SORTING BY SHORT BLOCK-MOVES

Lenwood S. Heath* John Paul C. Vergara†

Abstract

Sorting permutations by operations such as reversals and block-moves has received much interest because of its applications in computational biology, particularly in the study of genome rearrangements. A short block-move is an operation on a permutation that moves an element at most two positions away from its original position. This paper investigates the problem of finding a minimum-length sorting sequence of short block-moves for a given permutation. A $\frac{4}{3}$-approximation algorithm for this problem is presented. Exact polynomial-time algorithms are presented for woven bitonic permutations and woven double-strip permutations. A linear-time maximum matching algorithm for a special class of grid graphs is also discovered that improves the time complexity of one of these exact algorithms.

1 Introduction

Sorting permutations with the fewest number of operations has recently gained much interest in the area of computational biology [2, 3, 4, 15, 16]. An organism is uniquely determined by its *genome* and can be represented as a permutation of genes. When comparing closely related organisms, it can be assumed that the genome of each organism is a permutation of the same set of genes. A biologist wants to determine the minimum number of mutations or rearrangements required to transform one genome (permutation) into another as an estimate of the distance between two such genomes. Typical rearrangements are *reversals* or *block-moves* of a subsequence of contiguous genes in a genome.

Group theory allows us to view these rearrangements as permutations. If there are n genes, then each permutation is in S_n, the symmetric group on a set of n elements. Composition of permutations represents a mutation occurring on a genome. In particular, the expression

$$\pi_1 \cdot \gamma_1 \cdot \gamma_2 \cdots \gamma_k = \pi_2$$

[1]Department of Computer Science, Virginia Tech, Blacksburg, Virginia 24061-0106, heath@cs.vt.edu.
[2]Department of Computer Science, Ateneo de Manila University, Manila, Philippines, jpv@admu.edu.ph.

represents that the permutation π_2 is a result of composing the permutations $\gamma_1, \gamma_2, \ldots, \gamma_k$ with the permutation π_1. Equivalently, these permutations satisfy the expression

$$(\pi_2^{-1} \cdot \pi_1) \cdot \gamma_1 \cdot \gamma_2 \cdots \gamma_k = \boldsymbol{\iota},$$

where $\boldsymbol{\iota}$ is the identity permutation. Determining an evolutionary sequence for two genomes is therefore equivalent to sorting the permutation $\pi_2^{-1} \cdot \pi_1$.

We use a predicate P applicable to permutations to describe the set of allowed rearrangements. Vergara [20] develops the following formal generalization of the optimization problem.

MINIMUM SORTING BY P ($MinSort_P$)
INSTANCE: A permutation $\pi \in S_n$.
SOLUTION: A sequence of permutations $\gamma_1, \gamma_2, ..., \gamma_k$ such that $P(\gamma_i)$ is true for all i with $1 \leq i \leq k$, $\pi \cdot \gamma_1 \cdot \gamma_2 \cdots \gamma_k = \boldsymbol{\iota}$, and k is as small as possible.

The integer k in the above formulation is called the P *distance* of π, which we denote by $MinSort_P(\pi)$. The P *diameter* of the symmetric group S_n is the maximum possible P distance for length-n permutations.

Previous work on the $MinSort_P$ problem has concentrated on particular predicates P. $MinSort_P$ where P describes reversals has recently been shown NP-complete by Caprara [4]. Approximation algorithms for this problem have been devised [3, 15, 16] and the reversal diameter has been established [3]. Other predicates for which $MinSort_P$ has been studied include prefix reversals [9, 10, 12], fixed-length reversals [5], block-moves [2], first element re-insertions [1], and block-interchanges [6]. When P is considered part of the problem instance, Even and Goldreich [8] show that $MinSort_P$ is NP-hard. Jerrum [13] shows the stronger result that the problem is PSPACE-complete.

A *block* in a permutation π is a subsequence of contiguous elements in π. A *block-move* is a rearrangement that repositions a block in π. More precisely, the block-move $\beta\langle i, j, k\rangle$, where $1 \leq i < j < k \leq n+1$, is the permutation

$$\begin{pmatrix} 1 & 2 & \ldots & i-1 & i & i+1 & \ldots & j-1 & j & j+1 & \ldots & k-1 & k & k+1 & \ldots & n \\ 1 & 2 & \ldots & i-1 & j & j+1 & \ldots & k-1 & i & i+1 & \ldots & j-1 & k & k+1 & \ldots & n \end{pmatrix}.$$

In the product $\pi \cdot \beta\langle i, j, k\rangle$, the block $\pi(i)\ \pi(i+1)\ \ldots\ \pi(j-1)$ is moved to the immediate left of element $\pi(k)$. Equivalently, the block $\pi(j)\ \pi(j+1)\ \ldots\ \pi(k-1)$ is moved to the immediate left of element $\pi(i)$. Another equivalent description of the effect of block-move $\beta\langle i, j, k\rangle$ on π is that the adjacent blocks $\pi(i)\ \pi(i+1)\ \ldots\ \pi(j-1)$ and $\pi(j)\ \pi(j+1)\ \ldots\ \pi(k-1)$ are switched. Consider the permutation

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \mathbf{2} & \mathbf{3} & \mathbf{6} & \mathbf{7} & \mathbf{4} & \mathbf{5} & \mathbf{9} & \mathbf{8} & \mathbf{1} \end{pmatrix},$$

for example. Applying the block-move $\beta = \beta\langle 3, 5, 8\rangle$ to π yields

$$\pi \cdot \beta = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \mathbf{2} & \mathbf{3} & \mathbf{4} & \mathbf{5} & \mathbf{9} & \mathbf{6} & \mathbf{7} & \mathbf{8} & \mathbf{1} \end{pmatrix}.$$

The complexity of minimum sorting by block-moves remains unresolved, though a 1.5-approximation algorithm has been devised by Bafna and Pevzner [2]. *Block-interchanges* are a generalization that allow rearrangements to switch two blocks, not necessarily adjacent. Christie [6] shows that minimum sorting by block-interchanges is solvable in polynomial time. A *first element re-insertion* is a block-move of the form $\beta\langle 1, 2, k\rangle$. Aigner and West [1] show that minimum sorting by first element re-insertion is solvable in polynomial time.

Let Bk be the predicate that describes block-moves, so that $MinSort_{Bk}$ is the problem of minimum sorting by block-moves. Variants of this problem arise by the introduction of distance bounds on the predicate. For an integer bound ℓ, let Bk^ℓ be the predicate that describes the block-moves where the total length of the blocks switched is at most ℓ. Each integer ℓ implies a different set of allowed block-moves and thus a different sorting problem. Heath and Vergara [11] investigate sorting by such bounded block-moves and develop various results for $MinSort_{Bk^\ell}$ for different ℓ. The predicate Bk^3 describes *short block-moves*. They show that the Bk^3 diameter of S_n is $\lceil \binom{n}{2}/2 \rceil$ and present exact polynomial-time algorithms for $MinSort_{Bk^3}$ for two classes of permutations: bitonic permutations and double-strip permutations. This paper investigates the same problem and develops more general results. These results also appear in Vergara [20], where sorting by bounded rearrangements is studied in a more general context.

We summarize the results of this paper. Three different algorithms for $MinSort_{Bk^3}$ are devised. First, a $\frac{4}{3}$-approximation algorithm is presented in Section 4. Then, exact polynomial-time algorithms are devised for two special classes of permutations: woven bitonic permutations (Section 6) and woven double-strip permutations (Section 7). The algorithm for woven double-strip permutations uses matching, and we improve on its time complexity by developing a linear-time matching algorithm for a special class of grid graphs (Section 8).

2 Definitions

A permutation $\pi \in S_n$, when viewed as an ordered arrangement of integers, is denoted by the sequence of integers written in boldface and delimited by spaces; for example,

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ \mathbf{3} & \mathbf{4} & \mathbf{1} & \mathbf{5} & \mathbf{6} & \mathbf{2} \end{pmatrix} = \mathbf{3\ 4\ 1\ 5\ 6\ 2}.$$

A *short block-move* is a block-move that switches two adjacent blocks whose total length is at most 3. That is, $\beta\langle i,j,k\rangle$ is a short block-move when $k-i \leq 3$. In a short block-move, one block is a single element while the other block has at most two elements. Equivalently, a short block-move positions an element at most two positions away from its original position in the permutation. A short block-move that moves an element exactly one position away is called a *skip*. Otherwise, the short block-move positions an element exactly two positions away and it is called a *hop*. A hop is a *left hop* or a *right hop* depending on which direction the single element is moved.

Block-moves may be specified by the values of the elements instead of their positions. That is, in the context of the product $\pi \cdot \beta$, the block-move $\beta\langle i,j,k\rangle$ is alternately denoted by $B\langle s,t\rangle$, where $s = \pi(i)\ \pi(i+1)\ \ldots \pi(j-1)$ and $t = \pi(j)\ \pi(j+1)\ \ldots \pi(k-1)$ (the blocks moved in π). For example, let $\pi = \mathbf{2\ 4\ 5\ 1\ 3\ 6}$. The product $\pi \cdot \beta\langle 2,3,5\rangle$ may have been written as $\pi \cdot B\langle \mathbf{4}, \mathbf{5\ 1}\rangle$. A block-move specified by value is called a *potential* block-move since B may not be applicable to a given permutation. For example, the block-move $B\langle \mathbf{4}, \mathbf{3}\rangle$ does not apply to $\mathbf{2\ 4\ 5\ 1\ 3\ 6}$ but does apply to $\mathbf{3\ 4\ 1\ 5\ 6\ 2}$. Consider the permutation π and the potential short block-move B. The block-move B is an *actual* block-move with respect to π:

- If B is the skip $B\langle \pi(i), \pi(j)\rangle$ and $j = i+1$; or,
- If B is the hop $B\langle \pi(i), \pi(j)\ \pi(k)\rangle$ and $j = i+1, k = i+2$; or,
- If B is the hop $B\langle \pi(i)\ \pi(j), \pi(k)\rangle$ and $j = i+1, k = i+2$.

Observe that an actual block-move B can be immediately applied to π; that is, the product $\pi \cdot B$ is well defined.

In the product $\pi \cdot \beta\langle i,j,k\rangle$, β is a *correcting* block-move if $\pi(p) > \pi(q)$ for all p, q where $i \leq p < j \leq q < k$; that is, a correcting block-move corrects the relative order of the elements moved. An *inversion* in a permutation π is a pair of elements $\{\pi(i), \pi(j)\}$ in π that are not in their correct relative order ($i < j$ and $\pi(i) > \pi(j)$). Observe that a correcting block-move eliminates inversions.

3 Correcting Block-moves and Permutation Graphs

Heath and Vergara [11] show that it is sufficient to consider only correcting block-moves when seeking an optimal sorting sequence of short block-moves for a permutation.

Theorem 1 *[11] For a permutation π, there exists an optimal sequence of short block-moves $\beta_1, \beta_2, \ldots, \beta_k$ that sorts π such that each block-move is a correcting block-move.*

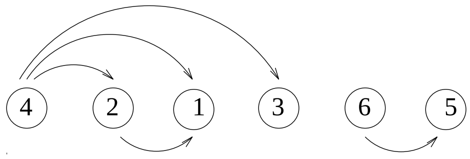


Figure 1: The permutation graph for $\pi = \mathbf{4\ 2\ 1\ 3\ 6\ 5}$.

Block-move	Arcs	Permutation graph
$B_1\langle\mathbf{4\ 2}, \mathbf{1}\rangle$	$(4, 1), (2, 1)$	Figure 2(a)
$B_2\langle\mathbf{4}, \mathbf{2\ 3}\rangle$	$(4, 2), (4, 3)$	Figure 2(b)
$B_3\langle\mathbf{6}, \mathbf{5}\rangle$	$(6, 5)$	Figure 2(c)

Table 1: Arcs erased by an optimal block-move sequence ($\pi = \mathbf{4\ 2\ 1\ 3\ 6\ 5}$).

The rest of this paper shall consider only sorting sequences of correcting short block-moves. Theorem 1 suggests the following graph representation of a permutation π. The *permutation graph* of π is the directed graph $G^p_\pi = (V^p_\pi, A^p_\pi)$ where $V^p_\pi = \{\pi(1), \pi(2), \ldots, \pi(n)\}$ and $(\pi(i), \pi(j)) \in A^p_\pi$ whenever $\pi(i) > \pi(j)$ and $i < j$. Every arc of G^p_π represents an inversion in π; in fact, in the context of a permutation graph, we shall not distinguish between inversions and arcs. Figure 1 illustrates the permutation graph for $\mathbf{4\ 2\ 1\ 3\ 6\ 5}$. The *in-degree* and *out-degree* of a vertex a in G^p_π follow the standard definitions of degrees in a directed graph. Permutation graphs have been studied extensively (e.g., [19]) and are examples of transitive graphs [18].

It can be verified that a correcting block-move corresponds to the removal of arcs in the permutation graph. In this context, we say that a block-move *erases* a set of arcs. For short block-moves, a correcting skip erases a single arc while a hop erases a pair of arcs. Let $\pi = \mathbf{4\ 2\ 1\ 3\ 6\ 5}$, whose permutation graph is shown in Figure 1. An optimal sorting sequence for π is:

$$B_1\langle\mathbf{4\ 2}, \mathbf{1}\rangle, B_2\langle\mathbf{4}, \mathbf{2\ 3}\rangle, B_3\langle\mathbf{6}, \mathbf{5}\rangle.$$

Figure 2 illustrates the effect of this sequence on G^p_π. Table 1 lists the block-moves and the dashed arcs of G^p_π that each block-move erases.

Two arcs in a permutation graph are *compatible* if their heads are identical or their tails are identical. The following proposition summarizes the potential of permutation graphs as a model for sorting by short block-moves.

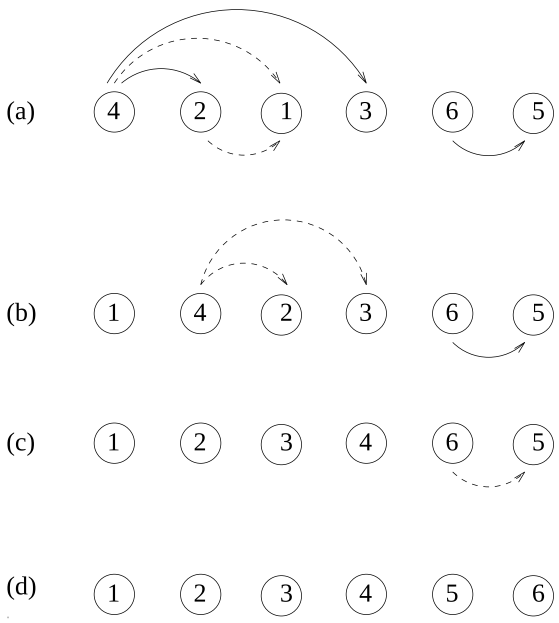


Figure 2: Effects of correcting short block-moves on a permutation graph.

Proposition 2 *Let π be a permutation, and let $\beta_1, \beta_2, \ldots, \beta_k$ be an optimal sequence of correcting short block-moves that sorts π. Then, each B_i erases a set of arcs A_i that contains a single arc or a pair of compatible arcs.*

The set $D_\pi = A_1, A_2, \ldots, A_k$ in Proposition 2 is called an *arc-partition* of G^p_π. The elements in this set could in fact be described by correcting short block-moves specified by value (potential block-moves) that erases the set of arcs. In general, the correcting skip $B\langle b, a\rangle$ erases the arc (b, a). The correcting hop $B\langle a, b\ c\rangle$ erases the pair of arcs (a, b) and (a, c). Finally, the correcting hop $B\langle b\ c, a\rangle$ erases the pair of arcs (b, a) and (c, a). Table 1 provides examples for each of these three cases.

It should be evident that the goal is to maximize the number of arc-pairs in the partition since this minimizes the total number of subgraphs derived from G^p_π, and, equivalently, the number of block-moves that sort π. Bounds for the length of an optimal sequence of block-moves that sorts π, $MinSort_{Bk^3}(\pi)$, can therefore be determined just from the size of A^p_π as stated in the following lemma.

Lemma 3 *The short block-move distance of a permutation π satisfies the inequality:*

$$\lceil |A^p_\pi|/2\rceil \leq MinSort_{Bk^3}(\pi) \leq |A^p_\pi|.$$

Recall that a necessary condition for a pair of arcs in A^p_π to be a hop is that they be compatible. This condition is by no means sufficient. It is possible that a pair of

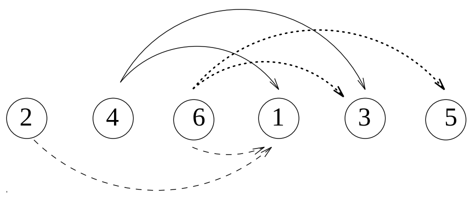


Figure 3: $B\langle \mathbf{2}\ \mathbf{6}, \mathbf{1}\rangle$ cannot be realized because of element **4**.

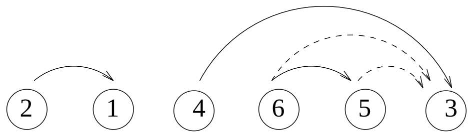
compatible arcs cannot be realized as a hop. Consider the example in Figure 3. The compatible arcs $\{(2,1),(6,1)\}$ cannot be a realized as a hop in any sequence of correcting block-moves because the element **4** occurs between elements **2** and **6** in the permutation and will therefore not "get out of the way" as it is already in its correct relative order with respect to elements **2** and **6**. In general, given a pair of compatible arcs $\{(a,b),(a,c)\}$ or $\{(b,a),(c,a)\}$ in a permutation graph G^p_π, the element x is an *obstacle* for the pair if $b < x < c$ and x occurs between b and c in π. A pair of arcs is a *feasible* hop if they are compatible and have no obstacle.

Naively obtaining an arc-partition and then deriving a corresponding sorting sequence is therefore an insufficient strategy for optimally sorting a permutation since the arc-partition may contain potential hops that are not feasible. In addition, as we find out later in Section 7, previously feasible hops may be rendered infeasible by the application of other block-moves.

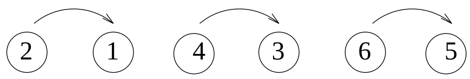
In the next sections, we use the permutation graph model to devise algorithms for minimum sorting by short block-moves. The first algorithm is an approximation algorithm with a provable approximation bound. The next two algorithms are exact algorithms for particular types of permutations.

4 A $\frac{4}{3}$-Approximation Algorithm

In this section, we present an algorithm for $MinSort_{Bk^3}$ that guarantees, for any permutation, a sorting sequence whose length is within $\frac{4}{3}$ of the optimal. The general strategy is to perform as many correcting hops as possible and avoid performing skips. Let G^p_π be a permutation graph. A *lone arc* (a,b) in G^p_π is an arc that is not compatible with any other arc (arc $(2,1)$ in Figure 4(a)). By Theorem 1, a lone arc must correspond to a skip in a sorting sequence The number of lone arcs in a graph thus gives us a lower bound on the number of correcting skips that *must* be performed when sorting a permutation and improves the lower bound of Lemma 3 for the short block-move distance of π.



(a)



(b)

Figure 4: Lone arcs in a permutation graph.

Lemma 4 *Let π be a permutation and let r be the number of lone arcs in G^p_π. Then,*

$$\lceil (|A^p_\pi| - r)/2 \rceil + r \leq MinSort_{Bk^3}(\pi).$$

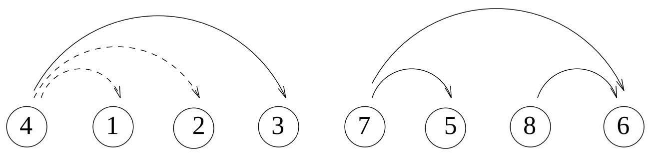
If (a, b) is a lone arc then the potential skip $B\langle a, b\rangle$ is a *lone skip*.

Lemma 5 *Let π be a permutation and let B be a lone skip in π. Then B is an actual block-move.*

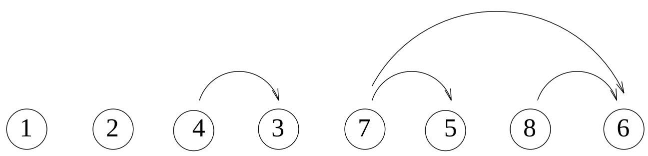
Proof: Let $B = B\langle a, b\rangle$ and let $a = \pi(i)$ and $b = \pi(j)$. We must show that $j = i + 1$. Suppose this is not the case. Then there exists an element x between a and b. Since (a, b) is an arc in G^p_π, either the arc (a, x) or the arc (x, b) is an arc in G^p_π. This results in a contradiction, because (a, b) is a lone arc implying that no other arc exists whose head is b or whose tail is a. The lemma follows. □

Consider an algorithm that sorts π using correcting block-moves. Such an algorithm may *introduce* lone arcs. Let $\pi = \mathbf{2\ 1\ 4\ 6\ 5\ 3}$. Its permutation graph is given in Figure 4(a). Observe that $(2, 1)$ is a lone arc for this permutation. However, upon the application of the correcting hop $B = B\langle \mathbf{6\ 5}, \mathbf{3}\rangle$, two additional lone arcs, $(4, 3)$ and $(6, 5)$, result. The permutation graph for $\pi \cdot B$ is given in Figure 4(b). This implies three more block-moves (lone skips) to complete the sort. Of course, an optimal sorting sequence for this permutation consists of correcting hops $B\langle \mathbf{6}, \mathbf{5\ 3}\rangle$ and $B\langle \mathbf{4\ 5}, \mathbf{3}\rangle$, followed by the lone skip $B\langle \mathbf{2}, \mathbf{1}\rangle$.

We prefer to apply a hop that does not introduce lone arcs. This is sometimes unavoidable as shown in Figure 5. Here, the permutation $\pi = \mathbf{4\ 1\ 2\ 3\ 7\ 5\ 8\ 6}$ allows only one correcting hop $B\langle \mathbf{4}, \mathbf{1\ 2}\rangle$ (shown in dashed arcs) and its application introduces the lone arc



(a)



(b)

Figure 5: Finding desirable correcting hops.

$(4, 3)$ in the permutation. A compromise is to always perform a correcting hop that introduces at most one lone arc. In this way, the resulting sorting sequence performs at least as many hops as there are lone skips (not counting the necessary ones implied by the lone arcs that were originally in the permutation graph). There is another problem to address that is also exhibited in Figure 5. After applying the only correcting hop possible, no correcting hops remain (Figure 5(b)). Define a *mushroom* in a permutation π as a subsequence of contiguous elements $a\ b\ c\ d$ in π, that satisfies $c > a > d > b$. The arc (a, d) is the *dome* of the mushroom, and the arcs (a, b) and (c, d) are its *shades*. In Figure 5(a), the subsequence **7 5 8 6** comprises a mushroom. Such subsequences in a permutation are crucial to the analysis of our algorithm, as we shall see later. For now, observe that the correcting skip $B\langle a, b\rangle$ and then the correcting hop $B\langle a\ c, d\rangle$ places the elements in a mushroom in their correct relative order.

Our algorithm, which we describe later in this section, distinguishes among three cases:

1. The permutation contains lone skips.

2. The permutation contains correcting hops.

3. The permutation contains a mushroom.

We now argue that at least one of these cases hold for an unsorted permutation. First, define an *adjacent inversion* in π to be a pair of elements $\{a, b\}$ such that $a > b$ and a occurs to the immediate left of b.

Lemma 6 *An unsorted permutation π contains at least one adjacent inversion.*

Proof: Let $a = \pi(i)$ be the largest unpositioned element in π. We claim that the element to its immediate right, $b = \pi(i+1)$, is less than a. Suppose this is not the case. Then, $b > a$ and is thus in its correct position; that is, $i + 1 = b$ so that $a < i + 1$. The element i is not in its correct position since a occupies its position. Hence, $i < a$. This implies $i < a < i + 1$, which contradicts the fact that a is an integer. Thus, $a > b$ and $\{a, b\}$ is an adjacent inversion in π. The lemma follows. $\square$

We locate a mushroom in a permutation by considering its adjacent inversions.

Lemma 7 *An unsorted permutation containing no lone skips or correcting hops contains a mushroom.*

Proof: Let π be an unsorted permutation. By Lemma 6, π contains at least one adjacent inversion. Let $a = \pi(i)$ and $b = \pi(i+1)$ be one such inversion. Now, this implies (a, b) is an arc in G^p_π. Since (a, b) is not a lone arc, there exists an arc (a, x) or an arc (x, b) for some element $x = \pi(j)$. We are interested in all elements x that satisfy this property. Define $span((a, b), x)$ for any element $x \neq a, b$ as follows:

$$span((a,b),x) = \begin{cases} j - (i+1) & \text{if } (a,x) \in A^p_\pi \\ i - j & \text{if } (x,b) \in A^p_\pi \\ \infty & \text{if neither } (a,x) \in A^p_\pi \text{ nor } (x,b) \in A^p_\pi. \end{cases}$$

Define $minspan(a, b)$ as follows:

$$minspan(a,b) = \min_{x \in V^p_\pi - \{a,b\}} span((a,b),x).$$

An x such that $span((a, b), x) = minspan(a, b)$ is a *closest connected neighbor* of $\{a, b\}$. Finally, among all adjacent inversions in π, let $\{\hat{a}, \hat{b}\}$ be an inversion that minimizes $minspan$ and let $\hat{x}$ be a closest connected neighbor of $\{\hat{a}, \hat{b}\}$.

Either $(\hat{a}, \hat{x})$ or $(\hat{x}, \hat{b})$ is an arc in G^p_π. We consider the first case and the second case follows by symmetry. We claim that the elements $\hat{a}$, $\hat{b}$, and $\hat{x}$ form part of a mushroom in π. Specifically, the arc $(\hat{a}, \hat{x})$ is the dome of this mushroom and the arc $(\hat{a}, \hat{b})$ is one of its shades. Since no correcting hops exist in π, there exists an obstacle for $\{(\hat{a}, \hat{x}), (\hat{b}, \hat{x})\}$. We claim that *exactly one* such obstacle y exists and that y forms with $\hat{x}$ the other shade $(y, \hat{x})$ of the mushroom, as illustrated in Figure 6. First, notice that any element y between $\hat{b}$ and $\hat{x}$ must be adjacent to either $\hat{a}$ or $\hat{x}$ because of the arc $(\hat{a}, \hat{x})$. Next, notice that the element y cannot be adjacent to $\hat{a}$ because this would mean y, not $\hat{x}$, is the closest connected neighbor of $\{\hat{a}, \hat{b}\}$. Thus, $(y, \hat{x}) \in A^p_\pi$ for every y between $\hat{b}$ and $\hat{x}$. It remains to show that only one such y exists. Suppose two or more elements occur between $\hat{b}$ and $\hat{x}$. Then a correcting hop exists between $\hat{x}$ and the two elements to its immediate left, contrary to the

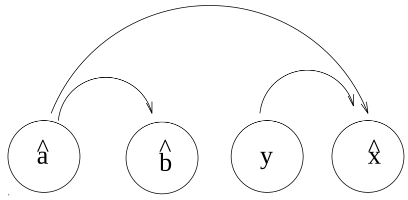


Figure 6: Finding a mushroom in a permutation.

assumptions of the lemma. We conclude that y is unique and that $\hat{a}\ \hat{b}\ y\ \hat{x}$ is a subsequence of contiguous elements. To show that this comprises a mushroom, we want $y > \hat{a} > \hat{x} > \hat{b}$. All relationships have been shown except for $\hat{x} > \hat{b}$. This easily follows because otherwise, the correcting hop $B\langle \hat{b}\ y, \hat{x}\rangle$ would exist. The lemma follows. □

Recall that our general strategy is to avoid skips. One way to do so is to prevent, whenever possible, the introduction of lone arcs. A lone arc is *pre-existing* if it is present in G^p_π initially. The corresponding skips are *required skips*. The goal is to show that every non-required skip performed during the sort is amortized by a hop. To do this, we need to precisely identify when a lone arc is introduced by a block-move.

Since a lone arc (a, b) is not compatible with other arcs, it should be clear that a has in-degree 0 and out-degree 1 and b has in-degree 1 and out-degree 0, An arc (a, b) is a *right hanging arc* if a has in-degree 0 and out-degree > 1 and b has in-degree 1 and out-degree 0; it is a *left hanging arc* if a has in-degree 0 and out-degree 1 and b has in-degree > 1 and out-degree 0; In a right hanging arc (a, b), a is called a *right critical element* and b is called a *a right leaf element*; in a left hanging arc (a, b), b is called a *left critical element* and a is called a *a left leaf element*. We state the following without proof.

Lemma 8 *Let π be a permutation and let B be a block-move that introduces the lone arc (a, b) when applied to π. Either a or b is a critical element in π.*

The following statements are directly implied by the above definitions.

1. A critical element cannot be a leaf element, and vice versa.

2. A right critical element cannot be a left critical leaf element, and vice versa.

3. A right leaf element cannot be a left leaf element, and vice versa.

A final observation is that, although a critical element may be associated with several leaf elements, only one such leaf element is relevant when analyzing whether a particular block-move introduces a lone arc.

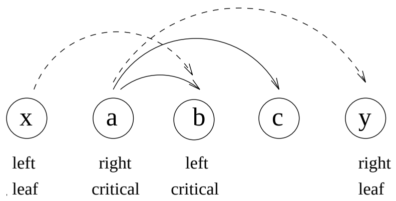


Figure 7: The case where $B\langle a, b\ c\rangle$ introduces lone arcs (x, b) and (a, y).

The following lemmas indicate when and how many lone arcs are introduced by lone skips, correcting hops, and block-moves that eliminate a mushroom.

Lemma 9 *Applying a lone skip to a permutation does not introduce a lone arc.*

Proof: Let π be a permutation with a lone skip $B\langle a, b\rangle$. The arc (a, b) is a lone arc and hence is not a hanging arc; this implies that a and b are non-critical elements. Since applying B to π affects only the degrees of the elements a and b, it does not introduce a lone arc. $\square$

Lemma 10 *For a permutation π containing correcting hops, there exists a correcting hop that, when applied to π, introduces at most one lone arc.*

Proof: Let B be a correcting hop in π. If B introduces at most one lone arc, then B satisfies the condition stated in the lemma. Otherwise, B is a block-move whose application introduces at least two lone arcs. By left-right symmetry, we may assume that B is a right hop. The objective is to identify an alternate correcting hop that introduces at most one lone arc. Let $B = B\langle a, b\ c\rangle$. Since B introduces at least 2 lone arcs, at least two of the following must hold:

(1) The element a is right critical.

(2) The element b is left critical.

(3) The element c is left critical.

Observe that if element c is left critical, its corresponding left leaf is b so that b cannot be left critical. This means at most two lone arcs may be introduced by the block-move and this can occur only when either statements **(1)** and **(2)** hold or when statements **(1)** and **(3)** hold.

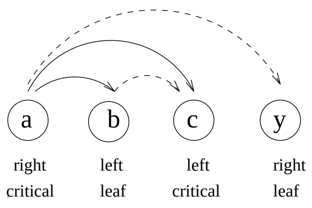


Figure 8: The case where $B\langle a, b\ c\rangle$ introduces lone arcs (b, c) and (a, y).

Suppose statements **(1)** and **(2)** hold. Figure 7 illustrates part of the corresponding permutation graph. Element x in the figure is a left leaf that corresponds to the left critical element b. Element y, on the other hand, is a right leaf that corresponds to the right critical element a. Note that the elements x, a, b, c, and y have to be contiguous in the permutation and that the right hop $B\langle a, b\ c\rangle$ does introduce two lone arcs, specifically, (x, b) and (a, y). Now, consider the correcting hop $B' = B\langle x\ a, b\rangle$. Upon application of B', the elements a and y are not adjacent in the resulting permutation so (a, y) does not occur as a lone arc in $\pi \cdot B'$. In addition, as x is a left leaf, it cannot cause the introduction of a lone arc (it is not right critical). This means that a lone arc could be introduced only through the left critical element b. Thus, B' is a correcting hop that introduces at most one lone arc.

For the second possibility, suppose statements **(1)** and **(3)** hold. Figure 8 illustrates part of the corresponding permutation graph. Again, element y in the figure is a right leaf that corresponds to the right critical element a. Recall that b is a left leaf that corresponds to the left critical element c. Note that the elements a, b, c, and y have to be contiguous in the permutation and that the right hop $B\langle a, b\ c\rangle$ does introduce two lone arcs, specifically, (b, c) and (a, y). Now, consider the correcting hop $B' = B\langle a\ b, c\rangle$. Upon application of B', the elements a and y are not adjacent in the resulting permutation so (a, y) does not occur as a lone arc in $\pi \cdot B'$. In addition, b is a left leaf so it cannot cause the introduction of a lone arc (it is not right critical). This means that a lone arc could be introduced only through the left critical element c. Thus, B' is a correcting hop that introduces at most one lone arc. The lemma follows. □

Lemma 11 *Let π be an unsorted permutation containing no lone skips or correcting hops. Let $a\ b\ c\ d$ be a mushroom in π. Applying $B\langle a, b\rangle$ followed by $B\langle a\ c, d\rangle$ does not introduce lone arcs.*

Proof: Let $\pi' = \pi \cdot B\langle a, b\rangle \cdot B\langle a\ c, d\rangle$. The four elements of the mushroom in π still occur contiguously in π'; specifically, they occur as the contiguous subsequence $b\ d\ a\ c$. In

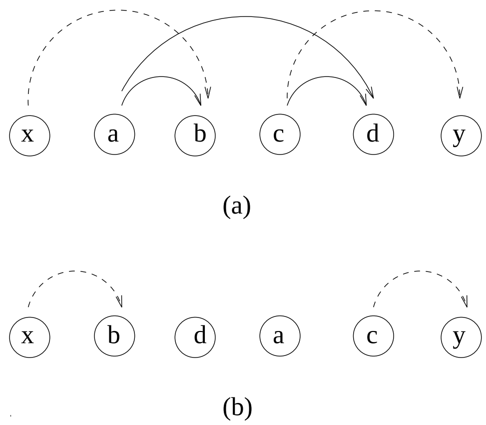


Figure 9: Arcs between mushroom elements.

addition, no arcs exist between them in $G^p_{\pi'}$ since the block-moves have eliminated the arcs of the mushroom, as Figure 9 illustrates.

Suppose π' contains a lone arc. The only possibilities are a lone arc (x, b) for some element x to the immediate left of b in π' or a lone arc (c, y) for some element y to the immediate right of c in π' (see Figure 9(b)). Neither of these possibilities can occur because this would mean that b is left critical in π with x as its left leaf ((x, b) is a left hanging arc) or that c is right critical in π with y as its right leaf ((c, y) is a right hanging arc), as illustrated in Figure 9(a). Either case implies the existence of a correcting hop in π ($B\langle x\ a, b\rangle$ or ($B\langle c, d\ y\rangle$), which contradicts the assumptions of the lemma. It follows that no lone arcs are introduced by the block-moves. $\square$

These results suggest a greedy approximation algorithm. Algorithm GreedyHops, shown in Figure 10, takes as input a permutation π and returns the length of a sorting sequence for π. The **while** loop in lines 2–16 repeatedly applies block-moves based on the existence of lone skips, correcting hops, or mushrooms in the permutation. Lemmas 9, 10, and 11 justify the three cases of the switch statement.

Theorem 12 *Algorithm* GreedyHops *sorts a permutation π with a sequence of short block-moves whose length is within $\frac{4}{3}$ of the optimal, in $O(n^3)$ time.*

Proof: Let π be a permutation. Lemma 7 guarantees that every iteration of the **while** loop in the algorithm performs correcting block-moves and thus eventually sorts π. Let r be the number of pre-existing lone arcs in G^p_{π}. GreedyHops first performs the r required skips. Any other lone arcs eliminated by the algorithm must be introduced by other block-moves performed during the sort. By Lemmas 9 and 11, lone arcs will not be introduced

GreedyHops(π) Sorting by short block-moves based on greedy hop choices.

INPUT: A permutation π.
OUTPUT: The length of a sorting sequence of block-moves for π.

1 $dist \leftarrow 0$
2 **while** $\pi \neq \iota$
3 **do switch case** there are lone skips in π
4 **do let** β be one such skip
5 $\pi \leftarrow \pi \cdot \beta$
6 $dist \leftarrow dist + 1$
7 **case** there are correcting hops for π
8 **do let** β be the hop that introduces the fewest lone arcs
9 $\pi \leftarrow \pi \cdot \beta$
10 $dist \leftarrow dist + 1$
11 **otherwise**
12 **do let** $a\ b\ c\ d$ be a mushroom in π
13 $\pi \leftarrow \pi \cdot B\langle a, b\rangle$
14 $\pi \leftarrow \pi \cdot B\langle a, c, d\rangle$
15 $dist \leftarrow dist + 2$
16 **return** $dist$

Figure 10: Algorithm GreedyHops.

by the block-moves in lines 5, 13, and 14 of the algorithm. In fact, by Lemma 10, at most one lone arc is possibly introduced whenever the block-move applied is the one indicated in line 9 of the algorithm, a correcting hop. This means that every non-required lone skip performed in line 5 is amortized by some correcting hop in line 9. Also, the correcting skip in line 13 is amortized by the correcting hop in line 14 (the mushroom case). Let s and h be the number of non-required skips and correcting hops, respectively, performed by the algorithm. By the above discussion, we have $h \geq s$. Noting that skips and hops eliminate 1 or 2 arcs from the graph G^p_π, respectively, the number of arcs in this graph is

$$|A^p_\pi| \quad = \quad 2h + s + r.$$

By Lemma 4,

$$\begin{aligned}
MinSort_{Bk^3}(\pi) &\geq \lceil (|A_\pi^p| - r)/2 \rceil + r \\
&= (2h+s)/2 + r \\
&= h + s/2 + r.
\end{aligned}$$

Let $GreedyHops(\pi)$ be the number of block-moves performed by the algorithm. By definition,

$$GreedyHops(\pi) \quad = \quad h + s + r.$$

If $s = 0$, then $GreedyHops(\pi) = MinSort_{Bk^3}(\pi)$; that is, GreedyHops returns an optimal solution. Otherwise, we have these inequalities on the approximation ratio:

$$\begin{aligned}
\frac{GreedyHops(\pi)}{MinSort_{Bk^3}(\pi)} &\leq \frac{h+s+r}{h+s/2+r} \\
&\leq \frac{s+s}{s+s/2} \quad \text{since } h \geq s > 0 \text{ and } r \geq 0 \\
&= \frac{4}{3}.
\end{aligned}$$

For the time complexity, the **while** loop in the algorithm performs $O(n^2)$ iterations. Each iteration requires at worst a linear scan of the elements since all it requires is consider every skip (first case) or every hop (second case) or every possible subsequence of 4 contiguous elements (third case). It follows that the algorithm runs in $O(n^3)$ time. □

5 Decomposing a Permutation into Subsequences

In this section, we define classes of permutations for which we have exact polynomial-time algorithms for the $MinSort_{Bk^3}$ problem.

First, define an *m-decomposition* of π to be a set $\{\chi_\pi^1, \chi_\pi^2, \ldots, \chi_\pi^m\}$ of pairwise-disjoint subsequences of π obtained as follows. Using a set of m labels $\{a_1, a_2, \ldots, a_m\}$, assign a label to each element of π. We then define the component χ_π^k, where $1 \leq k \leq m$, as the subsequence consisting of elements assigned the label a_k. For example, let

$$\pi = \mathbf{8\ 2\ 4\ 5\ 1\ 3\ 6\ 9\ 7}.$$

We obtain a 3-decomposition of π by assigning a_1 to **1** and **2**, a_2 to **4**, **7** and **8**, and, a_3 to **3**, **5**, **6**, and **9**. We arrive at the following 3-decomposition:

$$\{\mathbf{2\ 1},\ \ \mathbf{8\ 4\ 7},\ \ \mathbf{5\ 3\ 6\ 9}\}.$$

A *woven bitonic permutation* π is a permutation that has a 2-decomposition in which one sequence is increasing and the other sequence is decreasing. Specifically, there exists a 2-decomposition $\{inc_\pi, dec_\pi\}$ for π with increasing component

$$inc_\pi = \pi(i_1)\ \pi(i_2) \ldots \pi(i_r)$$

satisfying

$$\pi(i_1) < \pi(i_2) < \ldots < \pi(i_r)$$

and

$$dec_\pi = \pi(j_1)\ \pi(j_2) \ldots \pi(j_s),$$

satisfying

$$\pi(j_1) > \pi(j_2) > \ldots > \pi(j_s).$$

For example, the permutation

$$\pi = \mathbf{1\ 3\ 9\ 4\ 8\ 6\ 5\ 7\ 2}$$

is shown to be a woven bitonic permutation by this 2-decomposition

$$\boxed{\mathbf{1}}\ \boxed{\mathbf{3}}\ \mathbf{9}\ \boxed{\mathbf{4}}\ \mathbf{8\ 6\ 5}\ \boxed{\mathbf{7}}\ \mathbf{2}.$$

The elements in boxes comprise the increasing component while the remainder of the elements comprise the decreasing component; in particular, $inc_\pi = \mathbf{1\ 3\ 4\ 7}$ and $dec_\pi = \mathbf{9\ 8\ 6\ 5\ 2}$. A woven bitonic permutation generalizes the more familiar notion of a *bitonic* permutation, which is a permutation consisting of two subsequences of contiguous elements, an increasing one and a decreasing one. In a woven bitonic permutation, the elements in the subsequences need not be contiguous but are more generally interwoven.

A *woven double-strip permutation* π is a permutation that has a 2-decomposition whose components are both increasing. Specifically, there exists a 2-decomposition $\{inc^1_\pi, inc^2_\pi\}$ for π where both inc^1_π and inc^2_π are increasing subsequences. For example, the permutation

$$\pi = \mathbf{2}\ \boxed{\mathbf{1}}\ \boxed{\mathbf{4}}\ \mathbf{3}\ \boxed{\mathbf{5}}\ \boxed{\mathbf{7}}\ \mathbf{6\ 9}\ \boxed{\mathbf{8}}$$

is a woven double-strip permutation where $inc^1_\pi = \mathbf{1\ 4\ 5\ 7\ 8}$ and $inc^2_\pi = \mathbf{2\ 3\ 6\ 9}$.

In indicating a particular 2-decomposition for woven bitonic or a woven double-strip permutation, it is sufficient to specify one component as the other component is readily derived. Also, for simplicity, we allow empty subsequences in a decomposition; note that empty sequences can be considered as increasing or decreasing subsequences.

The identity permutation $\boldsymbol{\iota}$ is woven bitonic where the entire permutation comprises inc_π and dec_π is empty. The permutation $\boldsymbol{\iota}$ is also woven double-strip since any 2-decomposition

CheckWovenBitonic(π). Validating a woven bitonic permutation.

INPUT: A permutation π.
OUTPUT: Whether or not π is a woven bitonic permutation.

```
1  L ← a longest increasing subsequence of π
2  D ← the subsequence of elements in π not in L
3  if D is a decreasing subsequence
4    then return true
5    else  return false
```

Figure 11: Algorithm CheckWovenBitonic.

results in two increasing subsequences. The next two theorems show that it can be determined in polynomial time whether a given permutation is woven bitonic or woven double-strip.

Theorem 13 *Determining whether a permutation π is a woven bitonic permutation is solvable in $O(n \log \log n)$ time.*

Proof: Algorithm CheckWovenBitonic in Figure 11 returns **true** exactly when the permutation is woven bitonic. If the algorithm returns **true**, then $\{L, D\}$ is the required 2-decomposition. Now suppose the input permutation is woven bitonic. Let π be a woven bitonic permutation π with decomposition inc_π and dec_π, and let L be the LIS of π computed by the algorithm. Clearly, all elements in inc_π are in L and at most one element in dec_π may be in L. If none of the elements in dec_π are in L, then $D = dec_\pi$ and the algorithm returns **true**. If exactly one element a in dec_π is in L, then D is just dec_π without a. The sequence D is therefore either decreasing (empty, if $dec_\pi = a$) and the algorithm returns **true**.

Lines 2 and 3 of the algorithm require straightforward linear scans of π and D, respectively. The algorithm is therefore dominated by the determination of the LIS (line 1) which can be accomplished in $O(n \log \log n)$ time using Johnson's data structure [14]. □

Theorem 14 *Determining whether a permutation π is a woven double-strip permutation is solvable in $O(n)$ time.*

Proof: By Dilworth's Theorem (see [7]), a 2-decomposition exists for π if and only if the longest decreasing subsequence its longest decreasing subsequence is at most 2.

Determining an LDS for a permutation takes $O(n \log \log n)$ in general and involves maintaining a list of decreasing subsequences of lengths $1, 2, \ldots, n$. Such an algorithm can be revised so that it terminates whenever a decreasing subsequence of length greater than 2 is discovered. Determining whether or not the length of the LDS is at most 2 therefore requires just a linear scan of the permutation. It follows that an algorithm to determine whether a permutation is woven double-strip runs in $O(n)$ time. $\square$

In the next two sections, we solve $MinSort_{Bk^3}$ for both woven bitonic and woven double-strip permutations.

6 Woven Bitonic Permutations

In this section, we devise a polynomial-time algorithm for optimally sorting woven bitonic permutations by short block-moves. We first define the notion of pending elements. An element $a = \pi(p)$ in a permutation π is *right d-pending* for some integer d, where $0 < d \leq n-1$, if $\pi(q) < a$ for all q satisfying $p+1 \leq q \leq p+d$. Symmetrically, $a = \pi(p)$ is left d-pending if $\pi(q) > a$ for all q satisfying $p-d \leq q \leq p-1$.

Right (left) d-pending elements are associated with a non-zero magnitude (d) and a direction (right or left). In our discussion, we sometimes simplify by omitting one of these two parts (that is, right pending, left pending, or d-pending). whenever the appropriate generalization applies.

Right (left) pending elements are important in that correcting short block-moves can be immediately performed by moving the element a to its right (left) in π. The following lemma, whose straightforward proof is left to the reader, specifies the correcting hops that can be applied to a permutation given that a right or left d-pending element is identified.

Lemma 15 *Let π be a permutation and let $a = \pi(p)$ be a d-pending element in π such that d is even. Then, there is sequence of correcting hops $\beta_1, \beta_2, \ldots, \beta_{d/2}$, that, when applied to π results in a permutation*

$$\pi' = \pi \cdot \beta_1 \cdot \beta_2 \cdots \beta_{d/2},$$

whose arc set has cardinality $|A^p_{\pi'}| = |A^p_\pi| - d$. Specifically, if a is right pending, then

$$\beta_i = \beta\langle p+2i-2, p+2i-1, p+2i+1\rangle.$$

If a is left pending, then

$$\beta_i = \beta\langle p-2i, p-2i+2, p-2i+3\rangle.$$

A naive algorithm to sort π using short block-moves is to repeatedly identify d-pending elements and perform the corresponding hops. This is unfortunately insufficient because there will not always be a d-pending element where d is even. For example, all pending elements of **2 1 3 5 4 6** are 1-pending.

However, woven bitonic permutations are a special case. As an example, consider $\pi =$ **7 1 4 5 6 8 3 2** whose components are exhibited by

$$\mathbf{7}\ \boxed{1}\ \boxed{4}\ \boxed{5}\ \boxed{6}\ \boxed{8}\ \mathbf{3}\ \mathbf{2}.$$

The strategy of the upcoming algorithm is to insert elements from the decreasing component into the increasing component. For instance, observe that the element **7** is right 4-pending so that Lemma 15 applies and **7** can be moved to the right using 2 correcting hops. After the two hops, element **7** can be included in the increasing component as illustrated by

$$\boxed{1}\ \boxed{4}\ \boxed{5}\ \boxed{6}\ \boxed{7}\ \boxed{8}\ \mathbf{3}\ \mathbf{2}.$$

Now, element **2** is left 6-pending and therefore takes 3 correcting hops to be inserted in the increasing component resulting in

$$\boxed{1}\ \boxed{2}\ \boxed{4}\ \boxed{5}\ \boxed{6}\ \boxed{7}\ \boxed{8}\ \mathbf{3}.$$

Now we are left with element **3**, a left 5-pending element. Since it is 4-pending as well, 2 correcting hops and then one skip successfully inserts the element, resulting in

$$\boxed{1}\ \boxed{2}\ \boxed{3}\ \boxed{4}\ \boxed{5}\ \boxed{6}\ \boxed{7}\ \boxed{8},$$

the identity permutation. This is optimal since all but one block-move in the sort is a correcting hop (see Lemma 3). The above sort is performed using an "insert" metric, which we define next.

Let π be a woven bitonic permutation with components

$$\begin{aligned} inc_\pi &= \pi(i_1)\ \pi(i_2) \ldots \pi(i_r) \\ dec_\pi &= \pi(j_1)\ \pi(j_2) \ldots \pi(j_s). \end{aligned}$$

For notational convenience, set $i_0 = 0$ and $\pi(i_0) = 0$. Let $a = \pi(p)$ be an element in dec_π. Choose x satisfying $0 \leq x < r$ and

$$i_0 < i_1 < i_2 \cdots < i_x < p < i_{x+1} < \cdots < i_r.$$

Call i_x the *left neighbor index* of a; For example, let

$$\pi = \boxed{2}\ \mathbf{7}\ \boxed{4}\ \mathbf{5}\ \mathbf{3}\ \boxed{6}\ \mathbf{1}\ \boxed{8}.$$

For element $\pi(2) = \mathbf{7}$, its left neighbor index $i_x = 1$. The other elements in the decreasing sequence $\pi(4)$, $\pi(5)$, and $\pi(7)$ have left neighbor indices 3, 3, and 6, respectively. The intention is to determine the *insert position* in π where element a can be inserted in inc_π while retaining the increasing property of the sequence. We denote the insert position of an element by $insertpos(\pi, dec_\pi, a)$. There are three cases for insert position.

Case 1. $\pi(i_x) < \pi(p) < \pi(i_{x+1})$: In this case, a is already properly inserted in inc_π, so that $insertpos(\pi, dec_\pi, a) = p$. In the above example, $insertpos(\pi, dec_\pi, \mathbf{5}) = 4$.

Case 2. $\pi(i_{x+1}) < \pi(p)$: In this case, the insert position of a is to its right so that $insertpos(\pi, dec_\pi, a) = p$ is the position i_y where y is the largest integer $\leq r$ such that $\pi(i_y) < \pi(p)$. In the example, $insertpos(\pi, dec_\pi, \mathbf{7}) = 6$, the position of element **6**.

Case 3. $\pi(p) < \pi(i_x)$: In this case, the insert position of a is to its left so that $nsertpos(\pi, dec_\pi, 5)$ is the position i_y where y is the smallest integer > 0 such that $\pi(p) < \pi(i_y)$. In the example, $insertpos(\pi, dec_\pi, \mathbf{3}) = 3$, the position of element **4**. On the other hand, $insertpos(\pi, dec_\pi, \mathbf{1}) = 1$, the position of element **2**.

Define the *insert distance* $insertdist(\pi, dec_\pi, a)$ of an element $a = \pi(p)$ in dec_π to be the distance of a from its insert position; that is,

$$insertdist(\pi, dec_\pi, a) = insertpos(\pi, dec_\pi, a) - p.$$

The sign of $insertdist(\pi, dec_\pi, a)$ indicates whether the insert position is to the right or to the left of the element. The insert distances of the elements in the previous example are given in the following table.

Element	Insert Distance
7	4
5	0
3	-2
1	-6

An important observation is that when an element is moved to its insert position in a woven bitonic permutation, the resulting permutation remains woven bitonic, as stated in the following lemma.

Lemma 16 *Let π be a woven bitonic permutation with components inc_π and dec_π, and let $a = \pi(p)$ be an element in dec_π. Let $i = insertpos(\pi, dec_\pi, a)$ and let $d = insertdist(\pi, dec_\pi, a)$. If $d > 0$, then the permutation*

$$\pi' = \pi \cdot B\langle a, \pi(p+1)\ \pi(p+2) \ldots \pi(i)\rangle$$

is also a woven bitonic permutation. If $d < 0$, then the permutation

$$\pi' = \pi \cdot B\langle \pi(i)\ \pi(i+2) \ldots \pi(p-1), a\rangle$$

is also a woven bitonic permutation.

Proof: For either case, as long as dec_π contains at least two elements, the permutation π' is a woven bitonic permutation where $inc_{\pi'}$ is just inc_π with element a in its correct place and $dec_{\pi'}$ is just dec_π without the element a. If dec_π contains only one element, observe that $\pi' = \boldsymbol{\iota}$ and is therefore woven bitonic. $\square$

The following lemma determines pending elements from insert distances.

Lemma 17 *Let a be an element in the decreasing component of a woven bitonic permutation π and let $d = insertdist(\pi, dec_\pi, a)$. If $d > 0$, then a is right d-pending. If $d < 0$, then a is left $|d|$-pending.*

Proof: Let $i = insertpos(\pi, dec_\pi, a)$. First, suppose $d > 0$. We need to show that all d elements to the immediate right of element $a = \pi(p)$ are less than a. Let $b = \pi(q)$, where $p+1 \leq q \leq p+d$, be one such element. The element b can either be in dec_π or inc_π. If b is in dec_π, then $b < a$, since dec_π is decreasing and b occurs to the right of a. If b is in inc_π, then $b = \pi(q) < a$, by transitivity as $\pi(q) < \pi(i)$ since inc_π is increasing and $\pi(i) < a$ by definition. It follows that a is right d-pending.

Now, suppose $d < 0$. We need to show that all $|d|$ elements to the immediate left of element $a = \pi(p)$ are greater than a. Let $b = \pi(q)$, where $p + d \leq q \leq p-1$, be one such element. The element b can either be in dec_π or inc_π. If b is in dec_π, then $b > a$, since dec_π is decreasing and b occurs to the left of a. If b is in inc_π, then $b = \pi(q) > a$, by transitivity as $\pi(i) < \pi(q)$ since inc_π is increasing and $a < \pi(i)$ by definition. It follows that a is left $|d|$-pending. The lemma follows. $\square$

Algorithm BitonicSort, discussed later in this section, inserts pending elements into the increasing component depending on the parity of their insert distances. First, note that Lemma 17 implies that as long as the permutation being sorted contains an element with even insert distance, then sorting is straightforward since we can repeatedly insert each element using corresponding hops by Lemma 15. Algorithm InsertEven, shown in Figure 12, performs the hops for such an element. The algorithm takes as input, a permutation π, an element a, and an even integer d, such that either $d = 0$ or a is $|d|$-pending in π. If $d \neq 0$, then the element a is either right or left pending, depending on the sign of d. Presumably, d is the insert distance of a, computed by the algorithm (BitonicSort) that calls InsertEven. Algorithm InsertEven applies the corresponding hops to π and

InsertEven(π, a, d). Inserting a pending element in a permutation.

INPUT: A permutation π, an element a in π, and an even integer d such that $d = 0$ or a is $|d|$-pending in π.

OUTPUT: The resulting permutation after a has been inserted in π and the actual number of correcting hops applied.

```
if d > 0
   then π ← RightHops(π, a, d/2)
   else π ← LeftHops(π, a, |d|/2)
return (π, |d|/2)
```

Figure 12: Algorithm InsertEven.

returns the resulting permutation and $|d|/2$, the actual number of hops performed. The algorithm calls one of two functions, described as follows:

RightHops(π, a, k). This function applies k hops to π, specifically, hops that move element a to its right in π. The function returns the permutation that results after the hops have been applied to π.

LeftHops(π, a, k). This function is similar to RightHops but instead moves the element to the left.

For both these functions, if a is $2k$-pending, then Lemma 15 applies and the actual correcting hops are specified in the lemma.

We cannot guarantee that an element with positive, even insert distance exists in the permutation as it is sorted, particularly since this measure changes dynamically. In fact, in the example given earlier in this section, the last element inserted had odd insert distance. Algorithm InsertOdd, presented in Figure 13 is called by BitonicSort when one pending element remains and its insert distance d is odd. The algorithm spends one skip to insert a $|d|$-pending element. If $|d| = 1$, then a single skip suffices. If $|d| \geq 3$, then Lemma 15 then applies (the element is $(|d| - 1)$-pending) and $(|d| - 1)/2$ hops are performed An extra skip completes the insertion of a. In addition to the subroutines described earlier, the algorithm calls one of the functions RightSkip or LeftSkip described as follows.

RightSkip(π, a). This function applies a skip to π, specifically, the skip $\beta(p, p+1, p+2)$

InsertOdd(π, a, d). Inserting a pending element in a permutation.

INPUT: A permutation π, an element a in π, and an odd integer d such that a is $|d|$-pending in π.

OUTPUT: The resulting permutation after a has been inserted in π and the actual number of correcting block-moves applied.

if $d > 0$
  then $\pi \leftarrow$ RightHops$(\pi, a, \lfloor d/2 \rfloor)$
    $\pi \leftarrow$ RightSkip(π, a)
  else $\pi \leftarrow$ LeftHops$(\pi, a, \lfloor |d|/2 \rfloor)$
    $\pi \leftarrow$ LeftSkip(π, a)
return $(\pi, \lfloor |d|/2 \rfloor + 1)$.

Figure 13: Algorithm InsertOdd.

assuming $a = \pi(p)$. This function returns $\pi \cdot \beta$, the permutation that results.

LeftSkip(π, a). This function applies the skip $\beta(p-1, p, p+1)$ assuming $a = \pi(p)$ and returns $\pi \cdot \beta$.

Algorithm InsertOdd returns the resulting permutation and the actual number of block-moves applied $(\lfloor |d|/2 \rfloor + 1)$.

Algorithm BitonicSort repeatedly calls InsertEven whenever any elements with even insert distances remain in the decreasing component and then calls InsertOdd only when it is necessary, perhaps if only one element with odd insert distance remains. It is still possible, however, that several elements remain, all of which have odd insert distances. When this situation occurs, we select pairs of these elements and find a way for the elements to "correct" each other's insert distance parity, as the following example illustrates.

As an example, consider the woven bitonic permutation

$$\pi = \mathbf{13}\ \mathbf{12}\ \boxed{\mathbf{1}}\ \boxed{\mathbf{2}}\ \boxed{\mathbf{5}}\ \boxed{\mathbf{7}}\ \boxed{\mathbf{8}}\ \boxed{\mathbf{9}}\ \mathbf{6}\ \mathbf{4}\ \boxed{\mathbf{10}}\ \mathbf{3}\ \boxed{\mathbf{11}}.$$

The insert distances of the elements in the decreasing sequence are shown below:

Element	Insert Distance
13	12
12	11
6	-3
4	-5
3	-7

Element **13** is the only element with even insert distance so it is moved to the extreme right resulting in

$$\mathbf{12}\ \boxed{\mathbf{1}}\ \boxed{\mathbf{2}}\ \boxed{\mathbf{5}}\ \boxed{\mathbf{7}}\ \boxed{\mathbf{8}}\ \boxed{\mathbf{9}}\ \mathbf{6}\ \mathbf{4}\ \boxed{\mathbf{10}}\ \mathbf{3}\ \boxed{\mathbf{11}}\ \boxed{\mathbf{13}}.$$

The insert distances of the other elements remain unchanged after **13** has joined the increasing subsequence. We can insert elements **4** and **3** as follows. First, apply the following correcting hops:

$$B\langle \mathbf{4}\ \mathbf{10}, \mathbf{3}\rangle, B\langle \mathbf{6}, \mathbf{3}\ \mathbf{4}\rangle.$$

The first hop causes elements **3** and **4** to be adjacent. The second hop shifts the parity of the insert distance of both elements. The resulting permutation is

$$\mathbf{12}\ \boxed{\mathbf{1}}\ \boxed{\mathbf{2}}\ \boxed{\mathbf{5}}\ \boxed{\mathbf{7}}\ \boxed{\mathbf{8}}\ \boxed{\mathbf{9}}\ \mathbf{3}\ \mathbf{4}\ \mathbf{6}\ \boxed{\mathbf{10}}\ \boxed{\mathbf{11}}\ \boxed{\mathbf{13}}.$$

Note that the insert distance of **3** is now even, as is the insert distance of **4**, after **3** moves out of the way. We thus apply the following hops

$$B\langle \mathbf{8}\ \mathbf{9}, \mathbf{3}\rangle, B\langle \mathbf{5}\ \mathbf{7}, \mathbf{3}\rangle, B\langle \mathbf{8}\ \mathbf{9}, \mathbf{4}\rangle, B\langle \mathbf{5}\ \mathbf{7}, \mathbf{4}\rangle.$$

as direct applications of Lemma 15 resulting in the permutation

$$\mathbf{12}\ \boxed{\mathbf{1}}\ \boxed{\mathbf{2}}\ \boxed{\mathbf{3}}\ \boxed{\mathbf{4}}\ \boxed{\mathbf{5}}\ \boxed{\mathbf{7}}\ \boxed{\mathbf{8}}\ \boxed{\mathbf{9}}\ \mathbf{6}\ \boxed{\mathbf{10}}\ \boxed{\mathbf{11}}\ \boxed{\mathbf{13}},$$

where the insert distances for elements **12** and **6** remain as in the table above. These two elements are inserted by first applying the following correcting hops:

$$B\langle \mathbf{12}, \mathbf{1}\ \mathbf{2}\rangle, B\langle \mathbf{12}, \mathbf{3}\ \mathbf{4}\rangle, B\langle \mathbf{12}, \mathbf{5}\ \mathbf{7}\rangle.$$

These three hops move element **12** to the right. The third block-move, in particular, causes the insert distance of element **6** to shift from 3 to 4. At this point, the permutation is

$$\boxed{\mathbf{1}}\ \boxed{\mathbf{2}}\ \boxed{\mathbf{3}}\ \boxed{\mathbf{4}}\ \boxed{\mathbf{5}}\ \boxed{\mathbf{7}}\ \mathbf{12}\ \boxed{\mathbf{8}}\ \boxed{\mathbf{9}}\ \mathbf{6}\ \boxed{\mathbf{10}}\ \boxed{\mathbf{11}}\ \boxed{\mathbf{13}},$$

where **12** is still right 5-pending. Applying $B\langle \mathbf{8}\ \mathbf{9}, \mathbf{6}\rangle$ and $B\langle \mathbf{7}\ \mathbf{12}, \mathbf{6}\rangle$ to this permutation inserts element **6** resulting in the following:

$$\boxed{\mathbf{1}}\ \boxed{\mathbf{2}}\ \boxed{\mathbf{3}}\ \boxed{\mathbf{4}}\ \boxed{\mathbf{5}}\ \boxed{\mathbf{6}}\ \boxed{\mathbf{7}}\ \mathbf{12}\ \boxed{\mathbf{8}}\ \boxed{\mathbf{9}}\ \boxed{\mathbf{10}}\ \boxed{\mathbf{11}}\ \boxed{\mathbf{13}}.$$

Observe that the moves causes **12** to be right 4-pending so that the block-moves $B\langle \mathbf{12}, \mathbf{8\ 9}\rangle$ and $B\langle \mathbf{12}, \mathbf{10\ 11}\rangle$ complete the sort. To summarize, whenever elements with odd insert distances remain, a pair of elements is selected and the pair is inserted so that the parities of their insert distances are made even. This is always possible because any pair of elements from the decreasing sequence dec_π have to cross each other during the sort as the elements are not yet in their correct relative order. We shall see later that it is preferable that the insert distances of the pair have the same sign as in the first pair in the above sorting example (elements **3** and **4**).

We now describe two additional algorithms, InsertSameSign and InsertDiffSign that handle pairs of elements with odd insert distance. Algorithm InsertSameSign is shown in Figure 14 and takes as input the permutation π, two elements a and b, and two odd integers d_a and d_b with identical signs. The integers are presumed to be the insert distances of elements a and b so that these elements are $|d_a|$-pending and $|d_b|$-pending, respectively (Lemma 17). The goal is to insert these elements in inc_π using only correcting hops. The algorithm returns the resulting permutation and the number of hops performed. First, the algorithm establishes the positions of elements a and b (line 1) and then stores the distance between these two elements in the variable gap (line 2). Then, the algorithm distinguishes between two cases: whether the distances are positive (lines 4–12) or negative (lines 13–22). We discuss the positive case and the negative case follows by symmetry. The first step is to move element a to the right using correcting hops until it is next to element b. Depending on whether gap is odd or even, a will be immediately to the right (line 5) or immediately to the left (line 9) of b. Next, a correcting left hop is performed that shifts the remaining insert distance of the two elements (line 6 or 10). Finally, both elements are individually moved to the right using correcting hops until their insert positions are reached (lines 7–8 or lines 11–12). It is important to move a first particularly if gap is odd since a is already to the right of b and $a > b$.

Algorithm InsertDiffSign is shown in Figure 15 takes as input the permutation π, two elements a and b, and two odd integers d_a and d_b with different signs, presumed to be the insert distances of elements a and b. Assuming a occurs to the left of b, a and b are d_a-pending and $|d_b|$-pending, respectively. As in the previous algorithm, it begins by establishing the positions of elements a and b (line 1). It then stores in the variable $span$ the distance between these two elements (line 2). Next, the algorithm compares d_a and $|d_b|$ with $span$. Four cases result:

Case 1. $d_a < span$ **and** $|d_b| < span$ (lines 4–8): In this case, the element a is moved to the right using correcting hops and one correcting skip. The insertion of a causes the insert distance of b to be increased by 1. Lemma 15 and Lemma 17 then apply and b is inserted using correcting left hops.

InsertSameSign(π, a, d_a, b, d_b). Used in sorting woven bitonic permutations.

INPUT: A permutation π, elements a and b in π, and odd integers d_a and d_b such that a is $|d_a|$-pending, b is $|d_b|$-pending, and $d_a d_b > 0$.

OUTPUT: The resulting permutation after elements a and b have been inserted in π and the actual number of correcting hops applied.

```
let π(i) = a, π(j) = b   ▷ i < j
gap ← j − i   ▷ distance between elements a and b
if d_a > 0   ▷ direction is to the right
  then if gap is odd
         then π ← RightHops(π, a, (gap − 1)/2)
              π ← LeftHops(π, π(j + 1), 1)
              π ← RightHops(π, a, (d_a − 1)/2 − gap/2)
              π ← RightHops(π, b, (d_b − 1)/2)
         else π ← RightHops(π, a, gap/2 − 1)
              π ← LeftHops(π, π(j + 1), 1)
              π ← RightHops(π, a, (d_a − 1)/2 − (gap/2 − 1))
              π ← RightHops(π, b, (d_b − 1)/2)
  else   ▷ direction is to the left
       if gap is odd
         then π ← LeftHops(π, b, (gap − 1)/2)
              π ← RightHops(π, π(i − 1), 1)
              π ← LeftHops(π, b, (|d_a| − 1)/2 − gap/2)
              π ← LeftHops(π, a, (|d_b| − 1)/2)
         else π ← LeftHops(π, b, gap/2 − 1)
              π ← RightHops(π, π(i − 1), 1)
              π ← LeftHops(π, b, (|d_a| − 1)/2 − (gap/2 − 1))
              π ← LeftHops(π, a, (|d_b| − 1)/2)
return (π, (|d_a| + |d_b|)/2)
```

Figure 14: Algorithm InsertSameSign.

Case 2. $d_a < span$ **and** $|d_b| \geq span$ (lines 9–12): The element b is moved to the left until it is 1 or 2 (depending on the parity of $span$) positions to the right of a. This shifts the parity of the insert distance of a so Lemma 15 and Lemma 17 apply and right hops cause the insertion of element a. Inserting a, in turn, shifts the parity of the insert distance of b and element b is inserted using right hops.

Case 3. $d_a \geq span$ **and** $|d_b| < span$ (lines 13–16): Symmetric to Case 2.

Case 4. $d_a \geq span$ **and** $|d_b| \geq span$ (lines 17–20): Element a is inserted using right hops and a single skip, causing a shift in the parity of the insert distance of b. Then, element b is inserted using left hops.

We are now ready to describe our algorithm for sorting woven bitonic permutations. Algorithm BitonicSort, shown in Figure 16, takes as input a woven bitonic permutation π. The algorithm returns the short block-move distance of π and uses at most one skip in sorting π. It first establishes the decreasing component of π and sets a variable D to be this component (line 1). The variable D is a sequence of elements that represents the decreasing component of π even as π is updated. The **while** loop in lines 3–28 dynamically updates π and D by repeatedly inserting elements from D to the increasing component of π. The loop terminates when $|D| = 0$, equivalently, when π is sorted. In the loop, if an element with even insert distance exists in D (lines 4–8), then Algorithm InsertEven is called that inserts that element using correcting hops. If only elements with odd insert distances remain, then, when possible (lines 21–27), two elements whose insert distances have the same sign are selected from D and then InsertSameSign is called. There are cases when such elements do not exist in D such as when D has only one element. In this case (lines 9–13), that element is selected and Algorithm InsertOdd is called. Another case is when D contains exactly two elements whose insert distances differ in sign (lines 14–20). This is handled by a call to Algorithm InsertDiffSign. We complete the discussion of Algorithm BitonicSort with a description of the other functions used by this algorithm, particularly those that deal with the sequence D. Assume that π is a woven bitonic permutation and that D is its decreasing component.

InsertDistance(π, D, a)**.** Given π, D, and an element a from D, this function returns the insert distance of a in π. Computing the insert distance for a single element takes $O(n)$ time since this requires a linear scan of inc_π.

EvenExists(π, D)**.** Given π and D, this function returns **true** if an element from D with even insert distance exists in π; it returns **false** otherwise.

DiffSigns(π, D)**.** Given π and D, this function returns **true** if the insert distances of the two elements that D contains differ in sign; it returns **false** otherwise.

InsertDiffSign(π, a, d_a, b, d_b). Used in sorting woven bitonic permutations.

INPUT: A permutation π, elements a and b in π, and odd integers d_a and d_b such that a is $|d_a|$-pending, b is $|d_b|$-pending, and $d_a d_b < 0$.

OUTPUT: The resulting permutation after elements a and b have been inserted in π and the actual number of correcting hops applied.

let $\pi(i) = a, \pi(j) = b$ $\triangleright$ $i < j$
$span \leftarrow j - i$ $\triangleright$ distance between elements a and b
$dist \leftarrow (d_a + |d_b|)/2)$ $\triangleright$ value may be incremented; see first case below
switch case $d_a < span$ **and** $|d_b| < span$
  do $\pi \leftarrow$ RightHops($\pi, a, (d_a - 1)/2$)
    $\pi \leftarrow$ RightSkip(π, a)
    $dist \leftarrow dist + 1$
    $\pi \leftarrow$ LeftHops($\pi, b, (|d_b| + 1)/2$)
 case $d_a < span$ **and** $|d_b| \geq span$
  do $\pi \leftarrow$ LeftHops($\pi, b, \lceil span/2 \rceil - 1$)
    $\pi \leftarrow$ RightHops($\pi, a, (d_a + 1)/2$)
    $\pi \leftarrow$ LeftHops($\pi, b, (|d_b| - 1)/2 - \lceil span/2 \rceil + 1$)
 case $d_a \geq span$ **and** $|d_b| < span$
  do $\pi \leftarrow$ RightHops($\pi, a, \lceil span/2 \rceil - 1$)
    $\pi \leftarrow$ LeftHops($\pi, b, (|d_b| + 1)/2$)
    $\pi \leftarrow$ RightHops($\pi, a, (d_a - 1)/2 - \lceil span/2 \rceil + 1$)
 case $d_a \geq span$ **and** $|d_b| \geq span$
  do $\pi \leftarrow$ RightHops($\pi, a, (d_a - 1)/2$)
    $\pi \leftarrow$ RightSkip(π, a) $\triangleright$ no need to increment $dist$
    $\pi \leftarrow$ LeftHops($\pi, b, (|d_b| - 1)/2$)
return $(\pi, dist)$

Figure 15: Algorithm InsertDiffSign.

BitonicSort(π). Minimum sorting by short block-moves for woven bitonic permutations.

INPUT: A woven bitonic permutation π.
OUTPUT: The size of the shortest sorting sequence of block-moves.

```
D ← dec_π   ▷ choose an appropriate decomposition for π
dist ← 0
while |D| > 0
      do switch case EvenExists(π, D)
                    do a ← EvenElement(π, D)
                       d_a ← InsertDistance(π, D, a)
                       (π, d) ← InsertEven(π, a, d_a)
                       D ← RemoveElement(D, a)
               case |D| = 1
                    do a ← OneElement(D)
                       d_a ← InsertDistance(π, D, a)
                       (π, d) ← InsertOdd(π, a, d_a)
                       D ← RemoveElement(D, a)
               case |D| = 2 and DiffSigns(π, D)
                    do (a, b) ← TwoElements(D)
                       d_a ← InsertDistance(π, D, a)
                       d_b ← InsertDistance(π, D, b)
                       (π, d) ← InsertDiffSign(π, a, d_a, b, d_b)
                       D ← RemoveElement(D, a)
                       D ← RemoveElement(D, b)
               otherwise   ▷ default
                    do (a, b) ← SameSignElements(π, D)
                       d_a ← InsertDistance(π, D, a)
                       d_b ← InsertDistance(π, D, b)
                       (π, d) ← InsertSameSign(π, a, d_a, b, d_b)
                       D ← RemoveElement(D, a)
                       D ← RemoveElement(D, b)
         dist ← dist + d
return dist
```

Figure 16: Algorithm BitonicSort.

EVENELEMENT(π, D). Given π and D, this function returns an element in D that has even insert distance.

ONEELEMENT(D). This function returns the single remaining element in D.

TWOELEMENTS(D). This function returns the two remaining elements in D.

SAMESIGNELEMENTS(π, D). Given π and D, this function returns two elements from D whose insert distances have the same sign. Observe that if $|D| > 2$, such elements always exist.

REMOVEELEMENT(D, a). This function returns the sequence that results after removing the element a from D.

All of the above functions take $O(n)$ time (Algorithms ONEELEMENT and TWOELEMENTS take $O(1)$ time) and generally requires a linear scan of the elements in π or D.

Theorem 18 *Let π be a woven bitonic permutation. Algorithm* BITONICSORT *returns $MinSort_{Bk^3}(\pi)$, the length of an optimal sorting sequence of short block-moves for π, in $O(n^2)$ time.*

Proof: By Lemma 3, it suffices to show that at most one skip is performed in the sort and that all block-moves are correcting block-moves. The only possibilities where a skip is performed are with Algorithm INSERTODD and Algorithm INSERTDIFFSIGN. Algorithm INSERTODD is called only when a single element remains. Algorithm INSERTDIFFSIGN, on the other hand, is called only when two elements remain. Clearly, such a call may occur only on the last iteration in the loop of Algorithm BITONICSORT, since $|D| = 0$ after the call. This means that at most one of these algorithms is called and it follows that at most one skip is performed by the sort.

It remains to show that all the block-moves are correcting block-moves. Every iteration in the loop of Algorithm BITONICSORT inserts an element from the decreasing component D of π to its corresponding increasing component, after which both π and D are updated. The woven bitonic property of the updated permutation is therefore retained by Lemma 16. As a result, all hops and skips in the algorithms are justified by Lemmas 15 and 17 and are therefore correcting hops and skips.

Since all except perhaps one of the block-moves are correcting hops, all arcs in the permutation graph $G^p_\pi = (V^p_\pi, A^p_\pi)$ except perhaps one is paired with another as a correcting hop. Thus, the algorithm returns $\lceil |A^p_\pi|/2 \rceil$, the length of an optimal solution (Lemma 3).

There are $O(n)$ elements that need to be inserted in the increasing component of π so the loop in Algorithm BITONICSORT is executed $O(n)$ times. Each iteration involves a constant number of operations on D: the computation of insert distance for at most two elements, and the actual insertion of the elements. Recall that the operations on D as well as the computation of insert distance require $O(n)$ time. Also, inserting an element requires $O(n)$ (at most $n/2$) block-moves (calls to RightHops, LeftHops, RightSkip, and LeftSkip). It follows that the algorithm runs in $O(n^2)$ time. $\square$

In hindsight, the short block-move distance $MinSort_{Bk^3}(\pi)$ for woven bitonic permutations can be computed by constructing G^p_π and then determining $\lceil |A^p_\pi|/2 \rceil$, similarly an $O(n^2)$ process. However, A^p_π does not immediately exhibit the actual sorting sequence since an appropriate matching of arcs in A^p_π needs to be determined first. In the next section, we show that finding an appropriate matching is not a straightforward task and that the structure of the permutation is essential in guaranteeing that a matching does yield the sequence. Algorithm BitonicSort, on the other hand, can be easily revised so that it returns the actual sorting sequence, instead of just the short block-move distance.

7 Woven Double-Strip Permutations

In this section, we show that $MinSort_{Bk^3}$ is solvable in polynomial time for woven double-strip permutations. We begin by defining a graph model based on feasible arcs. For a permutation π, consider its permutation graph G^p_π the *arc graph* $G^a_\pi = (V^a_\pi, E^a_\pi)$ of π is an undirected graph with vertex set $V^a_\pi = A^p_\pi$ and $((a,b),(c,d)) \in E^a_\pi$ exactly when $\{(a,b),(c,d)\}$ is a pair of feasible arcs.

We develop an algorithm MatchSort to sort woven double-strip permutations. The algorithm first obtains a maximum matching M in G^a_π (recall that the intention is to maximize the hops, i.e., arc-pairs). The matching M is a set of edges from G^a_π, equivalently, a set of pairs of feasible arcs from G^p_π. Let U be the set of unmatched vertices in G^a_π; U, in turn corresponds to a set of arcs from G^p_π. From M and U, we derive the corresponding set D_π of subgraphs in G^p_π. The arc-pairs in D_π correspond to hops, single arcs to skips. The intention is to construct an optimal sorting sorting sequence from these hops and skips. Unfortunately, it is possible to *disable* a feasible arc-pair: a hop or skip may make a feasible hop infeasible. As an example, consider the permutation in Figure 17. When the hop for arc-pair $\{(6,2),(3,2)\}$ is applied, element **2** is placed between **1** and **3**, thereby becoming an obstacle for the arc-pair $\{(4,1),(4,3)\}$.

While this matching strategy does not work in all cases, it does apply to woven double-strip permutations. In addition, matching gives us a better lower bound for the length of an optimal sorting sequence for a permutation.

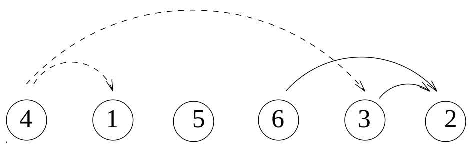


Figure 17: $\{(4,1),(4,3)\}$ is disabled when $B\langle \mathbf{6\ 3}, \mathbf{2}\rangle$ is applied.

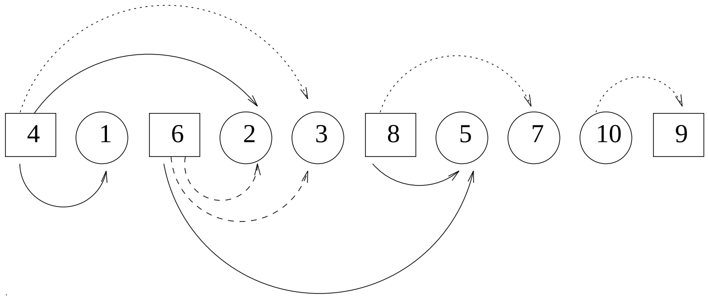


Figure 18: A maximum matching of feasible arc-pairs.

Lemma 19 *Let π be a permutation, and let M be a maximum matching in the arc graph G_π^a of π. Furthermore, let U denote the set of unmatched vertices in G_π^a. The short block-move distance of π satisfies the inequality:*

$$MinSort_{Bk^3}(\pi) \geq |M| + |U|.$$

We now consider woven double-strip permutations. The permutation

$$\pi = \boxed{4}\ \mathbf{1}\ \boxed{6}\ \mathbf{2\ 3}\ \boxed{8}\ \mathbf{5\ 7\ 10}\ \boxed{9}$$

is woven double-strip and its permutation graph G_π^p is shown in Figure 18. In the figure, the vertices are drawn using circles and squares to distinguish the two increasing components. The figure also illustrates a maximum matching of arcs. The unmatched arcs are drawn in dotted lines, while arc-pairs are drawn in solid or dashed. The derived set of hops and skips (described using potential block-moves) is

$$D_\pi = \{B\langle \mathbf{4}, \mathbf{1\ 2}\rangle, B\langle \mathbf{4}, \mathbf{3}\rangle, B\langle \mathbf{6}, \mathbf{2\ 3}\rangle, B\langle \mathbf{6\ 8}, \mathbf{5}\rangle, B\langle \mathbf{8}, \mathbf{7}\rangle, B\langle \mathbf{10}, \mathbf{9}\rangle\}.$$

The difficulty is that the sequence derived from D_π might not be realizable because of the possibility that a hop corresponding to an arc-pair may be disabled by an earlier block-move. Therefore, the goal is to transform the set D_π, if possible, into a realizable sorting sequence. Recall that an actual block-move B is a block-move that can be immediately applied to π. In our example, the potential block-move $B\langle \mathbf{6}, \mathbf{2}\ \mathbf{3} \rangle$ is an actual block-move with respect to π. The product $\pi \cdot B_1 \cdot B_2 \cdots B_k$ is a *realizable product* if each B_i is an actual block-move with respect to $\pi^{i-1} = \pi \cdot B_1 \cdot B_2 \cdots B_{i-1}$; the sequence $B_1, B_2, \ldots, B_k$ is then called a *realizable sequence.* The set D_π of potential block-moves is *realizable* if its elements can compose a realizable sequence.

We show in this section that for woven double-strip permutations, it can be guaranteed that the set D_π derived from M and U is realizable. Woven double-strip permutations have special characteristics, one of which is stated in the following lemma.

Lemma 20 *Let π be a woven double-strip permutation and let β be a correcting block-move. Then $\pi' = \pi \cdot \beta$ is a woven double-strip permutation.*

Proof: Since π is a woven double-strip permutation, a 2-decomposition $\{inc_\pi^1, inc_\pi^2\}$ for π exists. The block-move β preserves the correct relative order of the elements in π so that the same decomposition applies to $\pi' = \pi \cdot \beta$. It follows that π' is a woven double-strip permutation. □

The permutation graph for a woven double-strip permutation has the following property: all arcs (a, b) are such that a and b are from different increasing components. A feasible arc-pair $\{(a, b), (a, c)\}$ or $\{(b, a), (c, a)\}$ therefore has the element a in one component and the elements b and c in the other. Also, b and c have to appear consecutively in the component that they are in, as the following lemma states.

Lemma 21 *Suppose that π is a woven double-strip permutation. Further suppose that either $\{(a, b), (a, c)\}$ or $\{(b, a), (c, a)\}$ is a feasible arc-pair in G_π^p. If inc_π^1 (inc_π^2) is the component that contains b and c, then b and c occur consecutively in inc_π^1 (inc_π^2).*

Proof: Suppose b and c do not appear consecutively in inc_π^1 (or inc_π^2). Then, there exists an element x, where $b < x < c$, that is an obstacle between b and c (because the sequence that contains these elements is increasing). This implies that the arc-pair is not feasible, a contradiction. □

The next lemma shows that for woven double-strip permutations, feasible arc-pairs are never made infeasible.

Lemma 22 *Let π be a woven double-strip permutation. For a given feasible arc-pair in G_π^p, there is no sequence of correcting short block-moves that, when applied to π, will cause the arc-pair to be disabled.*

Proof: Let $\{(a, b), (a, c)\}$ be a feasible arc-pair. Without loss of generality, let inc_π^1 be the component that contains a, and let inc_π^2 be the component that contains b and c. Since

(a,b) and (a,c) are arcs in G^p_π, the relationships $a > b$ and $a > c$ hold. Let x be any element satisfying $b < x < c$ that occurs between b and c as a result of a sequence of correcting block-moves. Such an element x is not in inc^2_π as b and c are consecutive in this component (Lemma 21). Also, x must be less than a (by transitivity) and should therefore occur to the left of a in inc^1_π. Consequently, no sequence of correcting block-moves will place x to the right of a. However, elements b and c are to the right of a in π so x can never occur between b and c unless the hop $B\langle \mathbf{a}, \mathbf{b}\ \mathbf{c}\rangle$ that corresponds to arc-pair $\{(a,b),(a,c)\}$, has already been applied to the permutation. We conclude that no sequence of correcting block-moves can disable $\{(a,b),(a,c)\}$.

A symmetric argument applies to the other form for an arc-pair, $\{(b,a),(c,a)\}$. The lemma follows. □

Since disabling can be prevented as long as correcting short block-moves are performed, all that remains to be devised is a way to obtain an actual block-move from D_π.

Lemma 23 *Let π be a woven double-strip permutation. Let M be a maximum matching of the arc graph G^a_π and let U be the set of unmatched vertices in G^a_π. Furthermore, let D_π be the set of potential block-moves derived from M and U. Then, D_π contains an actual block-move.*

Proof: Algorithm GETACTUAL, shown in Figure 19, details a method of extracting an actual block-move given a permutation π and the set D_π. The algorithm first obtains the element t that is the tail of an arc in G^p_π and is as large as possible (line 1). It can be verified that this is equivalent to the arc-tail that appears rightmost in π. Without loss of generality, let inc^1_π be the increasing component that contains t. Consider the elements $h_1, h_2, \ldots h_m$ such that (t, h_i) is an arc. There are at least one of these elements, and such elements are all in inc^2_π, the other increasing component. Also, these elements have to be contiguous in π and occur to the immediate right of t, for otherwise arc-tails that occur to the right of t would exist. Consider h_1, the element to the immediate right of t (line 5); (t, h_1) is an arc in G^p_π and the algorithm calls a procedure ARCSEARCH that returns the potential block-move B in D_π that contains this arc (line 6). We show that B is either an actual skip, an actual right hop, or a left hop (not necessarily actual). If B is a skip, that is, $B = B\langle t, h_1\rangle$, then it is an actual block-move as t is immediately next to h_1 and the search is complete (lines 7–8). If B is some right hop $B\langle t, h_1\ h_j\rangle$, then $h_j = h_2$, the element on the immediate right of h_1 by Lemma 21 and B is an actual block-move (lines 9–10). The last possibility is that B is some left hop $B\langle t'\ t, h_1\rangle$ (line 14). If t' occurs on the immediate left of t then B is an actual block-move (line 13–14). Otherwise, we note that t' and t are consecutive elements in inc^1_π (Lemma 21) and that the elements between

GETACTUAL(π, D_π). Extracting an actual block-move.

INPUT: A woven double-strip permutation π and a set D_π of potential block-moves derived from a matching of G_π^a.
OUTPUT: An actual block-move from D_π.

1 $t \leftarrow \max\{a|(a,b)$ is an arc in $G_\pi^p\}$ ▷ get the rightmost arc-tail
2 $found \leftarrow$ **false**
3 **while not** $found$
4   **do let** $\pi(p) = t$
5   **let** $h_1 = \pi(p+1)$ ▷ element to t's right is an arc-head
6         $B \leftarrow$ ARCSEARCH($D_\pi, (t, h_1)$)
7   **switch case** $B = B\langle t, h_1\rangle$ ▷ an actual skip
8   **do** $found \leftarrow$ **true**
9   **case** $B = B\langle t, h_1\ h_2\rangle$ ▷ a right hop
10   **do** $found \leftarrow$ **true** ▷ an actual hop because $h_2 = \pi(p+2)$
11   **case** $B = B\langle t'\ t, h_1\rangle$ ▷ a left hop
12   **do let** $\pi(p') = t'$
13   **if** $p' = p - 1$
14   **then** $found \leftarrow$ **true** ▷ found an actual left hop
15   **else** $t \leftarrow t'$
16 **return** B

Figure 19: Algorithm GETACTUAL.

them in π, say, $h'_1, h'_2, \ldots h'_{m'}$, where $m \geq 1$, are in inc^2_π, precisely the component that contains h_1. In addition, since these elements are to the left of h_1 and (t', h_1) is an arc, (t', h'_i) is an arc for each h'_i. Figure 20 illustrates this situation. In the figure, the elements in circles are those in component inc^1_π while the elements in squares are those in inc^2_π; the pair of arcs in dashed lines is the hop $B\langle t'\ t, h_1\rangle$. Consider t' and observe that it possesses a property similar to the property t had at the beginning of this search. More precisely, the arc (t', h'_1) is part of a block-move that is either an actual skip, an actual hop $B\langle t', h'_1\ h_j\rangle$, where $h_j = h'_2$ (h_j cannot be h_1 since (t', h_1) is already part of the hop $B\langle t, t'\ h'_1\rangle$), or some left hop. The process can thus be repeated by setting t to t' (line 15). The **while** loop of the algorithm (lines 3–15) therefore eventually finds an actual block-move since the

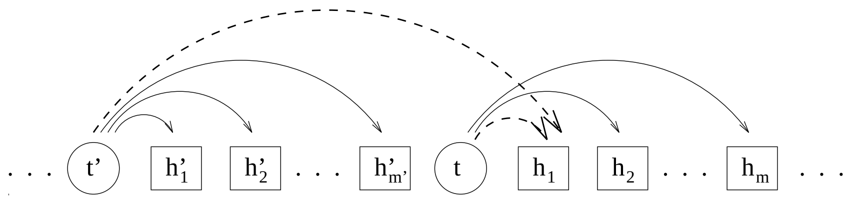


Figure 20: When a hop $B\langle t'\ t, h_1\rangle$ is not an actual block-move.

permutation does not go infinitely to the left. □

We now analyze the time complexity of Algorithm GetActual. Determining the rightmost arc-tail t (line 1) can be done in $O(n)$ time. It is not necessary to inspect all the arcs. Instead, since we now know that the element to t's immediate right is an arc-head, we simply search for the rightmost position p in π where $\pi(p) > \pi(p+1)$. The **while** loop is performed $O(n)$ times in the worst case, assuming the algorithm always finds a non-actual hop (line 15). The loop requires finding the potential block-move containing the arc (t, h_1). Assuming D_π is a stored as an array of block-moves sorted by its arcs, this takes $O(\log n)$ time. As a result, Algorithm GetActual runs in $O(n \log n)$ time.

We now present our main algorithm. Algorithm MatchSort, shown in Figure 21, returns an optimal sorting sequence of short block-moves given a woven double-strip permutation π. The algorithm first determines a maximum matching M of G^a_π (line 1) and then derives the set D_π of potential block-moves from M and the set of unmatched vertices U in G^a_π (line 2). Then, a **while** loop (lines 4–8) repeatedly selects an actual block-move B from D_π, applies B to π, and then removes B from D_π. The loop terminates when D_π is empty, after which, the sequence of block-moves applied is returned (line 9). We prove correctness of this algorithm in the following theorem.

Theorem 24 *Let π be a woven double-strip permutation. Algorithm* MatchSort *computes an optimal sorting sequence of short block-moves for π in $O(n^5)$ time.*

Proof: By Lemma 20, applying correcting block-moves to π does not destroy its woven double-strip property so that all the lemmas in this section apply at every iteration. In particular, Lemma 23 guarantees that there always exists an actual block-move from a set D_π derived from a maximum matching of G^a_π. We note that at every iteration, the permutation is transformed to a new permutation $\pi' = \pi \cdot B$, for some $B \in D_\pi$. A naive algorithm would be to compute at every iteration the graph G^a_π, a maximum matching M, and the set D_π. This is not necessary because by Lemma 22, no pair of arcs are rendered infeasible by a correcting block-move so that $G^a_{\pi'}$ is simply G^a_π minus the arcs

MatchSort(π). Minimum sorting by short block-moves using arc matching.

INPUT: A woven double-strip permutation π.
OUTPUT: The shortest sorting sequence of short block-moves, $B_1, B_2, \ldots, B_k$.

```
1  (M, U) ← a maximum matching in G^a_π   ▷ U represents the unmatched vertices
2  D_π ← potential block-moves derived from M and U
3  k ← 0
4  while |D_π| ≠ 0
5        do k ← k + 1
6           B_k ← GetActual(π, D_π)
7           π ← π · B_k
8           D_π ← D_π − B_k
9  return B_1, B_2, ..., B_k
```

Figure 21: Algorithm MatchSort.

associated with B. It follows that some $M' \subseteq M$ is a maximum matching of $G^a_{\pi'}$, and that $D_{\pi'} = D_\pi - B$. This justifies computing G^a_π, M, and D_π once before entering the loop and then removing the applied block-move B from D_π at every iteration. When the loop terminates, all arcs from the permutation graph are eliminated, or, equivalently, π is sorted. Since a block-move is removed from D_π at every iteration, the length of the sorting sequence returned is $|M| + |U|$, the number of potential block-moves in D_π before entering the loop. By Lemma 19, this is an optimal sorting sequence.

The **while** loop in the algorithm is performed $|D_\pi| = O(n^2)$ times. The computation therefore takes $O(n^3 \log n)$ time considering that Algorithm GetActual runs in $O(n \log n)$ time. However, line 1 of the algorithm involves obtaining a maximum matching in a graph which takes $O(v^{2.5})$ [17], where v is the number of vertices in the graph. Noting that $v = O(n^2)$, matching takes $O(n^5)$ time and, as a result, dominates the running time of the algorithm. $\square$

To illustrate how Algorithm MatchSort and Algorithm GetActual works, consider again, the example in Figure 18. Algorithm MatchSort returns the following sorting sequence:

$$B\langle \mathbf{10}, \mathbf{9} \rangle, B\langle \mathbf{6}, \mathbf{2\ 3} \rangle, B\langle \mathbf{6\ 8}, \mathbf{5} \rangle, B\langle \mathbf{8}, \mathbf{7} \rangle, B\langle \mathbf{4}, \mathbf{1\ 2} \rangle, B\langle \mathbf{4}, \mathbf{3} \rangle.$$

Figure 22 illustrates the permutation graphs for the intermediate permutations during the sort. The second block-move in the sequence, $B\langle \mathbf{6}, \mathbf{2}\ \mathbf{3} \rangle$, is a result of two iterations in Algorithm GetActual.

8 Maximum Matching in Stair Graphs

In this section, we improve the time complexity of Algorithm MatchSort by presenting a linear-time maximum matching algorithm for a special class of graphs. We first define this class of graphs and then show that the connected components of an arc graph of a woven double-strip permutation are in this class. An efficient matching algorithm for these graphs then implies a better running time for MatchSort.

Let $\mathbb{Z}$ be the set of integers and let G^g_∞ be the *infinite grid graph* that has vertex set $\mathbb{Z} \times \mathbb{Z}$, and edge set

$$\{(x_1, y_1), (x_2, y_2) \mid 1 = |x_1 - x_2| + |y_1 - y_2|\}.$$

A *grid graph* is a finite subgraph of G^g_∞. For a finite set $S \subset \mathbb{Z} \times \mathbb{Z}$, the *closed grid graph* $G^g_S = (S, E^g_S)$ is the subgraph of G^g_∞ induced by S.

If $x, a, b \in \mathbb{Z}$ satisfy $a \le b$, then the *column* $C[x : (a, b)]$ is the set

$$C[x : (a, b)] = \{(x, y) \mid a \le y \le b\}.$$

An integer p and a sequence of ordered pairs of integers $(a_1, b_1), (a_2, b_2), \dots, (a_m, b_m)$, satisfying $a_i \le b_i$, specify a subset S of $\mathbb{Z} \times \mathbb{Z}$ as follows:

$$S[p : (a_1, b_1), (a_2, b_2), \dots, (a_m, b_m)] = \bigcup_{i=1}^{m} C[p + i : (a_i, b_i)].$$

The set S is a *stair*, if it satisfies these conditions:

1. $a_i \le a_{i+1}$ for all i, $1 \le i < m$,

2. $b_i \le b_{i+1}$ for all i, $1 \le i < m$, and,

3. $b_i \ge a_{i+1}$ for all i, $1 \le i < m$.

A *stair graph* is the closed grid graph G^g_S of a given stair S. Figure 23 illustrates the stair graph for the stair

$$S[0 : (1, 3), (2, 4), (2, 4), (2, 5), (5, 6), (6, 6), (6, 6)].$$

It is straightforward to check that any stair graph is connected. In addition, if it satisfies only conditions 1 and 2, then every connected component of G^g_S is a stair graph.

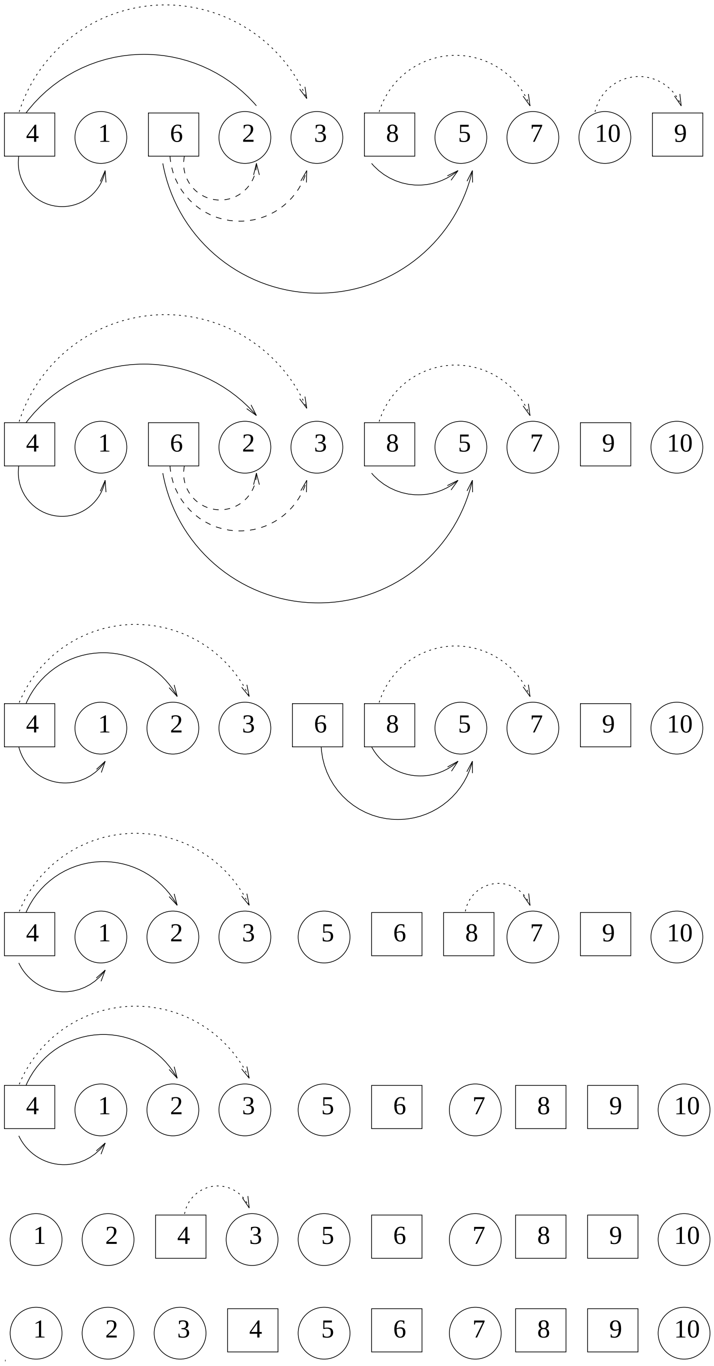


Figure 22: Sorting a woven double-strip permutation.

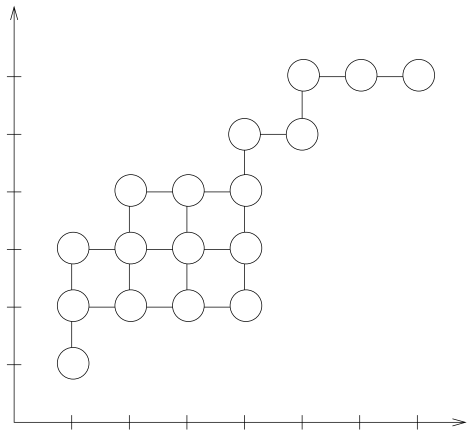


Figure 23: A stair graph.

Theorem 25 *Let π be a woven double-strip permutation. Each connected component of the arc graph G^a_π is a stair graph.*

Proof: Choose a 2-decomposition $\{inc^1_\pi, inc^2_\pi\}$ of π with

$$\begin{aligned} inc^1_\pi &= \pi(i_1)\ \pi(i_2)\ldots\pi(i_r) \\ inc^2_\pi &= \pi(j_1)\ \pi(j_2)\ldots\pi(j_s) \end{aligned}$$

such that inc^1_π is *minimal* in the following manner: no element of inc^1_π could be shifted to inc^2_π with inc^2_π remaining increasing. For each $\pi(i_v)$ in inc^1_π, we locate precisely its position relative to the elements of inc^2_π. Let $z(v)$ be the integer such that

$$j_1 < j_2 < \cdots < j_{z(v)} < i_v < j_{z(v)+1} < \cdots < j_s$$

(for notational convenience, $j_0 = 0$ and $j_{s+1} = n+1$). The element $\pi(i_v)$ is *early* if $\pi(i_v) < \pi(j_{z(v)})$; $\pi(i_v)$ is *late* if $\pi(i_v) > \pi(j_{z(v)+1})$. Since inc^1_π is chosen to be minimal, every element $\pi(i_v)$ is either early or late. Define $y(v)$ to be the integer that satisfies

$$\pi(j_1) < \pi(j_2) < \cdots < \pi(j_{y(v)}) < \pi(i_v) < \pi(j_{y(v)+1}) < \cdots < \pi(j_s).$$

Finally, define the interval (a_v, b_v), where $a_v \le b_v$, as follows:

- If $\pi(i_v)$ is early, then $(a_v, b_v) = (y(v)+1, z(v))$.
- If $\pi(i_v)$ is late, then $(a_v, b_v) = (z(v)+1, y(v))$.

It can be verified that the set $\{\{\pi(i_v), \pi(j_w)\} \mid a_v \le w \le b_v\}$ is the (nonempty) set of all inversions in π that contain the element $\pi(i_v)$.

Now, define a graph G' whose vertex set is

$$I = \{(v, w) \mid 1 \leq v \leq r, \{\pi(i_v), \pi(j_w)\} \text{ is an inversion in } \pi\}.$$

and whose edges connect two points whenever the corresponding arcs (inversions) are adjacent in G_π^a. We show that G' is a closed grid graph. By Lemma 21 in the previous section, feasible arcs $\{((\pi(i_v), \pi(j_w)), (\pi(i'_v), \pi(j'_w))\}$ (or $\{(\pi(j_w), \pi(i_v)), (\pi(j'_w), \pi(i'_v))\}$) imply $v' = v$ and $w' = w + 1$ or $w' = w$ and $v' = v + 1$, precisely the required adjacencies in a closed grid graph. Hence G' is the closed grid graph G_I^g.

We now prove that each connected component of G_I^g is a stair graph. The set I is alternately described as follows:

$$\begin{aligned} I &= \{(v, w) \mid 1 \leq v \leq r, a_v \leq w \leq b_v\} \\ &= \bigcup_{v=1}^{r} C[v : (a_v, b_v)] \\ &= S[0 : (a_1, b_1), (a_2, b_2), \ldots, (a_r, b_r)]. \end{aligned}$$

It suffices to show that I is a set of stairs, that is, that the first two conditions specified earlier hold. This is indeed the case as $a_v \leq a_{v+1}$ and $b_v \leq b_{v+1}$, for all v, $1 \leq v < r$, follows from the fact that $y(v) \leq y(v + 1)$ and $z(v) \leq z(v + 1)$. Consequently, I is a set of stairs and the components of G_I^g are stair graphs. The theorem follows. $\square$

We return our focus to stair graphs and on obtaining a more efficient maximum matching algorithm. Let

$$S = S[p : (a_1, b_1), (a_2, b_2), \ldots, (a_m, b_m)]$$

be a stair and let G_S^g be the stair graph it induces. Define the *outer path* P_S of G_S^g to be the path

$$\begin{gathered} (p + 1, a_1), (p + 1, a_1 + 1), \ldots, (p + 1, b_1), \\ (p + 2, b_1), (p + 1, b_1 + 1), \ldots, (p + 2, b_2), \\ (p + 3, b_2), (p + 1, b_2 + 1), \ldots, (p + 3, b_3), \\ \vdots \\ (p + m, b_{m-1}), (p + 1, b_{m-1} + 1), \ldots, (p + 1, b_m). \end{gathered}$$

Intuitively the outer path is the highest path in G_S^g from $(p + 1), a_1)$ to $(p + m, b_m)$. The outer path P_S of the example in Figure 23 is

$$(1, 1), (1, 2), (1, 3), (2, 3), (2, 4), (3, 4), (4, 4), (4, 5), (5, 5), (5, 6), (6, 6), (7, 6).$$

Each vertex in P_S is either *odd* or *even* depending on its position in the sequence. The first vertex $(p + 1, a_1)$ is odd and the succeeding vertices in the path are alternately even and

StairGraphMatch(G). Maximum matching in stair graphs.

INPUT: A stair graph G.
OUTPUT: A maximum matching of G.

let $\hat{P} = v_1, v_2, \ldots, v_k$ be the outer slice of G
$M \leftarrow \{(v_{2i-1}, v_{2i}) \mid 1 \leq i \leq \lfloor k/2 \rfloor\}$
$\triangleright$ If k is odd (no end-even vertex in G), leave v_k unmatched
let $H_1, H_2, \ldots, H_\ell$ $\triangleright$ be the connected components of $G - \hat{P}$
for $i \leftarrow 1$ **to** ℓ
    do $M \leftarrow M \cup$ StairGraphMatch(H_i)
return M

Figure 24: Algorithm StairGraphMatch.

odd. An *end-even* vertex (x, y) is an even vertex in P_S such that $y = b_i$ for some i. In the example, $(3,4)$, $(4,5)$, $(5,6)$, and $(7,6)$ are end-even vertices. Traverse the outer path P_S of G_S^g starting with $(p+1, a_1)$ until the first end-even vertex is encountered. The resulting path is called the *outer slice* $\hat{P}_S$ of G_S^g. If P_S contains no end-even vertices, we set $\hat{P}_S = P_S$. In the example, $\hat{P}_S$ is

$$(1,1), (1,2), (1,3), (2,3), (2,4), (3,4).$$

The reader may verify that each connected component of the graph $G_S^g - \hat{P}_S$ is a stair graph.

The recursive algorithm StairGraphMatch, shown in Figure 24, takes as input a stair graph G and returns a maximum matching of G. The algorithm first finds a maximum matching in the outer slice $\hat{P}$ and then recursively finds a matching in the connected components of $G - \hat{P}$.

Let $\hat{P} = v_1, v_2, \ldots, v_k$ be the outer slice of a stair graph G. We will show that the resulting matching M is a maximum matching of G. Divide $\hat{P}$ into segments (subpaths) as follows. Start a segment by traversing the path $\hat{P}$. Whenever an odd vertex x is encountered that occurs in a column different from the column of the preceding odd vertex, end the current segment and start a new segment. Continue until all vertices in the path have been traversed. Let $Q_1, Q_2, \ldots, Q_\ell$ be the segments of $\hat{P}$. Each Q_i is an even-length path, except when no end-even vertex exists in G (in which case Q_ℓ is a single vertex). Also,

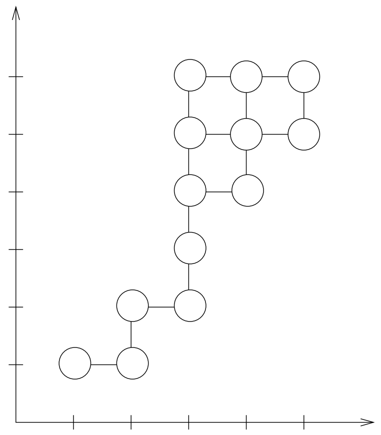


Figure 25: A stair graph with a straight segment.

each vertex in Q_i occurs in a single column, except perhaps the last vertex. If the last vertex in Q_i occurs in a separate column, the segment is a *bent segment*; otherwise, it is a *straight segment*. The example in Figure 23 has two segments $(1,1),(1,2),(1,3),(2,3)$ and $(2,4),(3,4)$, both of which are bent segments. Figure 25 shows an example of a stair graph whose outer slice has a straight segment. In the figure, $\hat{P}$ consists of segments $(1,1),(2,1)$, $(2,2),(3,2)$, and straight segment $(3,3),(3,4),(3,5),(3,6)$. Our goal is to show that there exists a maximum matching for G that contains the maximum matchings of each of these segments.

Fix a maximum matching M of G^g_S. A *rail* is a matched edge that is horizontal in G^g_S. Denote by $rails(M)$ the set of rails in M. A column $C = C[x:(a,b)] \subseteq S$ is *rail-free* if none of the rails in M contain an endpoint in C. We emphasize that C may be a *subcolumn*; that is, it need not contain all the points in S with the same x coordinate. A *track* $T[x:(a,b)] \subseteq S$ in G^g_S is the union of these two adjacent columns of points with identical intervals:

$$T[x:(a,b)] \quad = \quad C[x:(a,b)] \cup C[x+1:(a,b)].$$

The *length* of $T[x:(a,b)]$ is $b-a$. A track T *contains* a rail r if both endpoints of r are in T.

In general, we want to eliminate rails in our matching. The matching obtained from a straight segment, for instance, has no rails, while the matching obtained from a bent segment contains exactly one rail. The following lemma allows us to systematically eliminate particular rails from a given matching.

Lemma 26 *Let G be a stair graph and let M be a maximum matching in G. Let $T = T[x : (a, b)]$ be a track in G such that T contains the two rails $r_1 = ((x, a), (x + 1, a))$ and $r_2 = ((x, b), (x + 1, b))$ and the column $C[x : (a + 1, b - 1)]$ is rail-free. Then, there exists an alternate maximum matching M' in G such that*

$$rails(M') = \begin{cases} rails(M) - \{r_1\} & \text{if } b - a \text{ is even} \\ rails(M) - \{r_1, r_2\} & \text{if } b - a \text{ is odd} \end{cases}$$

Proof: By induction on $b - a$, the length of T. Let the theorem hold for tracks of length $< b - a$.

Suppose $b - a$ is even. Since $C[x : (a + 1, b - 1)]$ is rail-free, at least one of the points in this column is unmatched so that an alternating path from $(x + 1, a)$ to an unmatched vertex from this column exists. Switching matched and unmatched edges in that path results in a matching M' without rail r_1.

Suppose $b - a$ is odd. In the rail-free column $C = C[x : (a + 1, b - 1)]$, the entire column C comprises an alternating path for otherwise, it would contain two unmatched vertices and a larger maximum matching results. The alternating path starts with $(x, a + 1)$ and ends with $(x, b - 1)$. Now, consider the column $C' = C[x + 1 : (a + 1, b - 1)]$. By the induction hypothesis, we can assume that at most one rail exists in the track $T[x + 1 : (a + 1, b - 1)]$. Thus, C' is either rail-free or contains an endpoint of at most one rail. This means that either C' contains exactly one unmatched vertex or the entire column comprises an alternating path. If both C and C' comprise alternating paths, then together with r_1 and r_2, an alternating cycle is formed and a matching M' without r_1 and r_2 results. If exactly one unmatched vertex exists in C', then there is an alternating path that starts with such a vertex, includes r_1 and r_2, and passes through column C. Switching matched and unmatched edges in that path results in a our desired matching M'. $\square$

A *top rail* is an edge $((x, y), (x + 1, y))$ in M such that $(x, y + 1)$ is not a vertex in G. The next lemma shows that we can sometimes enforce that a rail in a track be a top rail.

Lemma 27 *Let G be a stair graph and let M be a maximum matching in G Let $T = T[x : (a, b)]$ be a track in G such that*

- *T contains the single rail $r = ((x, a), (x + 1, a))$,*
- *the point $(x, b + 1)$ is not a vertex in G, and,*
- *the column $C[x : (a + 1, b)]$ is rail-free.*

Then, there exists an alternate maximum matching M' in G such that

$$rails(M') = \begin{cases} rails(M) - \{r\} & \text{if } b - a \text{ is odd} \\ rails(M) - \{r\} \cup \{((x, b), (x + 1, b))\} & \text{if } b - a \text{ is even} \end{cases}$$

Proof: By induction on the distance of vertex (x, a) from the $(p+m, b_m)$.

Suppose $b-a$ is odd. Since $C[x:(a+1,b)]$ is rail-free, at least one of the points in this column is unmatched so that an alternating path from $(x+1,b)$ to an unmatched vertex from this column exists. Switching matched and unmatched edges in that path results in the a matching M' without rail r.

Suppose $b-a$ is even. In the rail-free column $C = C[x:(a+1,b)]$, the entire column C comprises an alternating path for otherwise, it would contain two unmatched vertices and the matching is not maximum. The alternating path starts with $(x, a+1)$ and ends with (x, b). Now, consider the column $C' = C[x+1:(a+1,b)]$. By Lemma 26, we may assume that there is at most one rail (in the track $T' = T[x+1:(a+1,b)]$) whose endpoint is in C'. If C' is rail-free, then, like column C, C' comprises an alternating path, this time from $(x+1, a+1)$ to $(x+1, b)$. Combined with r, we form an alternating cycle that results in a matching M' without the rail r but with the rail $((x,b),(x+1,b))$. On the other hand, if a rail exists whose endpoint is in C', then by the induction hypothesis, we can assume that this rail is a top rail of some track that contains T'. The only possibility where this rail r' is in T' is when $r' = ((x+1,b),(x+2,b))$. This, in turn, means that the column $C[x+1:(a+1,b-1)]$ is rail-free and must contain an unmatched vertex. There now exists an alternating path that starts with such a vertex, goes down to pass through rail r and column C, and ends at rail r'. The result is a matching M' that excludes r and includes $((x,b),(x+1,b))$. □

We are now ready to prove that Algorithm StairGraphMatch returns a maximum matching.

Theorem 28 *Let G_S^g be a stair graph. Algorithm* StairGraphMatch *returns a maximum matching for G_S^g in $O(|S|)$ time.*

Proof: It suffices to show that there is a maximum matching of G that contains a maximum matching of $\hat{P}$. This justifies first obtaining a maximum matching for the outer slice $\hat{P}$ of G.

Consider each segment $Q_1, Q_2, \ldots, Q_\ell$ in sequence. First consider the case of a bent segment

$$Q_i = (x,a), (x,a+1), \ldots, (x,b), (x+1,b).$$

Let a' be the the smallest integer where (a', b) is in S. The vertices in track $T[x:(a',b)]$ are all part of Q_i. The rail-free column $C[x:(a, a'-1)]$ comprises the rest of Q_i. By Lemma 26, we may assume at most one rail exists in track T. In addition, by Lemma 27, if such a rail exists, we may assume it is the top rail $((x,b),(x+1,b))$. Thus, there exists a maximum matching that contains the maximum matching of Q_i, specifically, the edges

$$((x,a),(x,a+1)),((x,a+2),(x,a+3)),\ldots,((x,b),(x+1,b)).$$

On the other hand, suppose that

$$Q_i = (x,a),(x,a+1),\ldots,(x,b)$$

is a straight segment. Let a' be the the smallest integer where (a',b) is in S. The vertices in track $T[x:(a',b)]$ are all part of Q_i. The rail-free column $C[x:(a,a'-1)]$ comprises the rest of Q_i By Lemma 26, we may assume at most one rail exists in track T. In fact, we may assume that no such rail exists because if it does, then a vertex from the segment must be unmatched ($|Q_i|$ is even) resulting in an alternating path from the unmatched vertex to the single rail, which eliminates the rail. It follows that there exists a matching that contains the maximum matching of Q_i, specifically, the edges

$$((x,a),(x,a+1)),(((x,a+2),(x,a+3)),\ldots,((x,b-1),(x,b)).$$

It remains to handle the special case where the last segment is a single vertex. Let (a,b) be this vertex, an odd vertex. Let (a',b') be the last vertex in $Q_{\ell-1}$. Since (a',b') is not end-even, the vertex $(a'-1,b')$ exists and is precisely (a,b). In addition $(a,b+1)$ does not exists because it would have been part of Q_ℓ. We conclude that (a,b) is adjacent only to (a',b') in G_S^g. This justifies leaving this single vertex unmatched.

It follows that separately obtaining the maximum matching of the outer slice $\hat{P}$ (line 2, in the algorithm) is appropriate and Algorithm StairGraphMatch obtains a maximum matching for G_S^g.

For the time complexity, obtaining an outer slice requires a simple scan of the intervals given for the stair. Moreover, he recursive calls process previously unvisited components of the graph. It follows that the algorithm runs in linear time with respect to $|S|$, the number of vertices in G_S^g. □

With this matching result, Algorithm MatchSort is no longer dominated by the computation of M.

Corollary 29 *Algorithm* MatchSort *computes an optimal sorting sequence of short block-moves for a woven double-strip permutation in $O(n^3 \log n)$ time.*

The arc graph of a permutation has $O(n^2)$ vertices. We can derive the short block-move distance of a woven double-strip permutation from the size of the obtained matching. Thus, computing just the short block-move distance is a simpler task.

Corollary 30 *There is an $O(n^2)$ algorithm to determine the short block-move distance of a woven double-strip permutation.*

References

[1] M. Aigner and D. B. West, *Sorting by insertion of leading element*, Journal of Combinatorial Theory A, 45 (1987), pp. 306–309.

[2] V. Bafna and P. A. Pevzner, *Sorting permutations by transpositions*, in Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, 1995, pp. 614–623.

[3] ——, *Genome rearrangements and sorting by reversals*, SIAM Journal on Computing, 25 (1996), pp. 272–289.

[4] A. Caprara, *Sorting by reversals is difficult.* Submitted to the *SIAM Journal on Computing*, 1996.

[5] T. Chen and S. Skiena, *Sorting with fixed-length reversals*, Discrete Applied Mathematics, 71 (1996), pp. 269–295.

[6] D. A. Christie, *Sorting permutations by block-interchanges*, Information Processing Letters, 60 (1996), pp. 165–169.

[7] B. A. Davey and H. A. Priestley, *Introduction to Lattices and Order*, Cambridge University Press, Cambridge, 1990.

[8] S. Even and O. Goldreich, *The minimum-length generator sequence problem is NP-hard*, Journal of Algorithms, 2 (1981), pp. 311–313.

[9] W. H. Gates and C. H. Papadimitriou, *Bounds for sorting by prefix reversal*, Discrete Mathematics, 27 (1979), pp. 47–57.

[10] E. Gyori and G. Turan, *Stack of pancakes*, Studia Scientarium Mathematicarum Hungarica, 13 (1978), pp. 133–137.

[11] L. S. Heath and J. P. C. Vergara, *Sorting by bounded block-moves.* Submitted to *Discrete Applied Mathematics;* TR 97-09, Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA, 1996.

[12] M. H. Heydari and I. H. Sudborough, *On sorting by prefix reversals and the diameter of pancake networks*, in Parallel Architectures and Their Efficient Use, Lecture Notes in Computer Science, vol. 678, 1992.

[13] M. R. Jerrum, *The complexity of finding minimum-length generator sequences*, Theoretical Computer Science, 36 (1985), pp. 265–289.

[14] D. B. Johnson, *A priority queue in which initialization and queue operations take $O(\log\log D)$ time*, Mathematical Systems Theory, 15 (1982), pp. 295–309.

[15] J. Kececioglu and D. Sankoff, *Efficient bounds for oriented chromosome inversion distance*, in Fifth Annual Symposium on Combinatorial Pattern Matching, Lecture Notes in Computer Science, vol. 807, 1994, pp. 307–325.

[16] ——, *Exact and approximation algorithms for sorting by reversals, with application to genome rearrangement*, Algorithmica, 13 (1995), pp. 180–210.

[17] P. A. Peterson and M. C. Loui, *The general maximum matching algorithm of Micali and Vazirani*, Algorithmica, 3 (1988), pp. 511–533.

[18] A. Pnueli, A. Lempel, and S. Even, *Transitive orientation of graphs and identification of permutation graphs*, Canadian Journal of Mathematics, 23 (1971), pp. 160–175.

[19] D. Rotem and J. Urrutia, *Circular permutation graphs*, Networks, 12 (1982), pp. 429–437.

[20] J. P. C. Vergara, *Sorting by Bounded Permutations*, PhD thesis, Virginia Polytechnic Institute and State University, 1997.