Improving Order with Queues

Andreas Karrenbauer
Max Planck Institute for Informatics, Saarbrücken, Germany

Kurt Mehlhorn
Max Planck Institute for Informatics and Saarland University

Pranabendu Misra
Chennai Mathematical Institute (CMI), Chennai, India

Paolo Luigi Rinaldi
Max Planck Institute for Informatics, Saarbrücken, Germany

Leo Wennmann
Computer Science Department, Goethe-Universität Frankfurt, Frankfurt, Germany

Siavash Rahimi Shateranloo
Max Planck Institute for Informatics, Saarbrücken, Germany

Alireza Haqi
Max Planck Institute for Informatics, Saarbrücken, Germany

Abstract

A classic problem in computer science is the following: given a sequence of $n$ numbers, and a collection of $k$ parallel queues, can we sort this sequence with these $k$ queues? The queues operate on a First-In-First-Out (FIFO) principle. We can push the elements of the input sequence, one by one, into the queues, and construct the output sequence by popping elements from the queues, one by one. This problem was studied by Tarjan, inspired by a problem posed by Knuth. It was studied in the context of certain card games (Patience and Floyd’s game), which led to an algorithm called Patience sort for it. Evan and Itai related it to chromatic number of Permutation graphs. It also arises in the context of planning and sorting items in assembly lines of factories, which has important applications.

We call the length of the Longest Decreasing Subsequence of a sequence its LDS. It is known that $k$ queues suffice if and only if the LDS of the input sequence is at most $k$. In this paper, we consider this problem when the number of queues $k$ is given and the LDS of the input sequence is possibly more than $k$. This is a natural generalization that appears in some important practical applications. Surprisingly, not much is known about it. We investigate the extent to which the disorder of the input sequence can be reduced. We prove the following results:

1. Let $L$ be the LDS of the input sequence. We give a simple algorithm that, using $k$ queues, reduces the LDS from $L$ to $L - k + 1$. It is based on Patience sort.

2. Then we show that, for any $L > 0$, there exist sequences of LDS $L$, such that it is impossible to reduce the LDS below $L - k + 1$ with only $k$ queues. Thus, the above algorithm is optimal. This completely characterizes the power of $k$ parallel queues in reducing the LDS and is the first main result of this paper.

3. Sorted sequences have LDS 1. Merging two sorted queues to produce a sorted sequence is a central step in efficient sorting algorithm, and it is always possible. In contrast, we show that if the two sequences have LDS 2, then it is not always possible to merge them into a sequence of LDS 2. Indeed, we give an algorithm that given two sequences of LDS 2, decides in polynomial time if it is possible to merge them.

4. Following the above barriers on LDS, we consider an alternative measure of disorder in a permutation, namely the number of downsteps. A downstep in a sequence is an item immediately followed by a smaller item. Intuitively, it represents a 'hiccup' in the sequence. We give an algorithm that for any number $k$ of queues, outputs a sequence with the least number of downsteps attainable with $k$ queues. This algorithm is online. This completely characterizes the power of $k$ parallel queues in reducing downsteps and is the second main result of our paper.

Our research was inspired by an application in car manufacturing.
Improving Order with Queues

2012 ACM Subject Classification  None

Keywords and phrases  Production, Patience Sort, Improving Order, Queues.
1 Introduction

A classic problem in computer science is the following: Given a sequence \( \pi \) of \( n \) numbers, and a collection of \( k \) parallel queues, can we sort \( \pi \) using these queues? Queues operate on a First-In-First-Out (FIFO) principle. The elements of \( \pi \) are, one by one, pushed into one of the \( k \) queues, and the output sequence is generated by popping elements from one of the \( k \) queues, one-by-one.

This problem was studied by Tarjan \[20\] inspired by a problem posed by Knuth \[13\], and also by Even and Itai \[9\]. It was studied even earlier by Schensted \[18\], and later by Mallows \[14\] in context of certain card games (Patience and Floyd’s game). It is known that we can sort \( \pi \) if and only if the number of queues, \( k \), is at least the length of the longest decreasing subsequence (LDS) of \( \pi \) \[20\]. It can achieved by using an algorithm called Patience sort \[1\]. This problem is related to the chromatic number of the Permutation graphs \[9,11\]: the length of the LDS of \( \pi \) is equal to the chromatic number of a certain conflict graph of \( \pi \), which is a permutation graph. It is also related to certain properties of the Young Tableaux \[1,18\].

A particularly important application of this problem appears in factories, where items are produced on assembly lines. Assembly lines, e.g., in car manufacturing, are partitioned by buffers to minimize the effect of line stops in one section to the other ones. The simplest buffer consists of a single lane that operates as a queue according to the First-In-First-Out (FIFO) principle, which maintains the order in which the items arrive and leave. Typically, there are two or more parallel lanes to which the arriving items are distributed and the leaving sequence is merged from. This presents a mechanism of buffers with sorting capabilities depending on the number of parallel queues it consists of. More sorting power allows more flexibility in the the planning of the build sequence because different sections may have other constraints and/or objectives \[5,8,15,19\]. Moreover, displacements in the sequence, e.g., caused by faults or other real-world events, can be mitigated to some extent as well. Understanding the sorting capabilities of such buffer mechanisms becomes very important in these applications.

Patience sort \[1\], a sorting algorithm inspired by the card game Patience, sorts a sequence of numbers using a minimal number of queues. The input is a sequence \( \pi_1 \) to \( \pi_n \) of \( n \) distinct numbers and the goal is to sort the numbers into increasing order. One starts with a set of empty queues \( Q_1, Q_2, \ldots \) and then enqueues the numbers one by one. When \( \pi_i \) is to be enqueued one determines the smallest index \( j \) such that either \( \pi_i \) is larger than the last element in the \( j \)-th queue or the \( j \)-th queue is empty and then appends \( \pi_i \) to the \( j \)-th queue. In this way, an increasing sequence is constructed in each queue. Once all elements are enqueued, the sorted sequence is extracted by repeatedly dequeuing the smallest first element of any queue. We call this dequeuing strategy Smallest Element First for later reference.

The number of queues required by Patience Sort is the length of the longest decreasing subsequence in the input sequence and no algorithm can sort with fewer queues. Moreover, if one considers all permutations of a sorted sequence that can be sorted again using a certain number of queues, say \( k \), this yields the set of permutations avoiding the pattern \((k + 1, k, \ldots, 1)\).

In this paper we ask: What can one do if the number of queues is fixed and may be smaller than the length of the longest decreasing subsequence of the input? This is a natural generalization, with some important practical applications, and also an interesting
Improving Order with Queues

combinatorial problem in its own right. Surprisingly not much was known about it. Our investigations lead us to consider: How does one measure the “disorder” of a sequence and how much of it can one remove with $k$ queues? We will consider two measures of disorder:

- The length of the longest decreasing subsequence of a sequence
- The number of down-steps in a sequence, i.e., the number of elements that are immediately followed by a smaller element.

Let $k$ be the number of available queues, $n$ be the length of the input sequence and $L$ be the LDS of this sequence. It is easy to observe that the LDS is 1 if and only if the number of downsteps is 0; which happens only for sorted sequences. Observe that, in general, the number of downsteps of a sequence is always at least as large as the LDS minus one; this follows directly from the properties of a longest decreasing subsequence. On the other hand, there exist sequences of length $n$, where the number of downsteps is $n/2$ while the LDS is just 2, e.g., 2, 1, 4, 3, ..., $n$, $n - 1$.

For LDS, we first give a simple algorithm based on Patience Sort that reduces the LDS from $L$ to $L - k + 1$. It can be considered as a truncated Patience Sort and runs in time $O(n \log k)$.

**Theorem 1.** Let $L \geq 1$ be arbitrary. With $k$ queues the LDS of any sequence with LDS $L$ can be reduced to $L - k + 1$. The algorithm can be made to run in time $O(n \log k)$, where $n$ is the length of the input sequence.

On the other hand, the LDS cannot be reduced below $L/k$ as at least one of the queues must contain at least that many elements from an LDS of the input and elements in the same queue cannot overtake each other. This lower bound suggests a natural strategy: distribute the elements of the input sequence to the $k$ queues s.t. the LDS of each queue is at most $\lceil L/k \rceil$ and then merge these sequences optimally. To this end, we consider the question of merging two queues without increasing the LDS. In this context, recall that merging two sorted lists to produce another sorted list is central to efficient sorting algorithms, such as Mergesort [7]. The LDS of a sorted list is 1. It is folklore that we can always merge two queues of LDS 1 to produce a sequence of LDS 1. But what happens if the two queues have LDS 2 or more?

We show that, there exist two sequences of LDS 2 that cannot be merged to a sequence of LDS of 2. We further give an algorithm that, given two sequences of LDS 2, either merges them into a sequence of LDS 2, or correctly concludes that they can’t be merged. This is accomplished via a reduction to the 2-SAT problem.

**Theorem 2.** There exist two sequences of LDS 2 such that they can’t be merged into a sequence of LDS 2.

Further, there is an algorithm that given two sequences of LDS 2, either merges them into another sequence of LDS 2, or correctly concludes that this is impossible. It can be implemented to run in linear time in $n$, the sum of the length of the two input sequences.

Based on this insight, we asked ourselves whether there is an algorithm that partitions a given an input sequence with LDS 4 into two mergeable sequences. We found out that this is not always possible. We prove even more generally that for every integer $L$, there exists a sequence LDS $L$ such that it is impossible to reduce the LDS to strictly less than $L - k + 1$ with only $k$ parallel queues. Hence, the above simple algorithm is tight, and no other other

---

1 To the best of our knowledge.
algorithm (no matter the running time) can do better! This lower-bound is unconditional and it is proved via a novel recursive construction. It completely characterizes the power of $k$ parallel queues in reducing the LDS of a sequence.

**Theorem 3.** For every $L$, there exist sequences of LDS $L$ which cannot be improved to an LDS strictly less than $L - k + 1$ with only $k$ parallel queues.

The above barriers on LDS motivated us to consider the second measure of disorder of a sequence, namely the number of down-steps. Recall that this is the number of items in the sequence that are immediately followed by a smaller item. Intuitively, each down-step represents a ‘hiccup’ in the sequence, and our objective is to minimize these.

We give an optimal algorithm $\text{Alg}_{\text{DS}}$ for decreasing the number of down-steps, i.e., no algorithm can output a sequence with fewer down-steps than $\text{Alg}_{\text{DS}}$ does. The algorithm is online, i.e., the decision where to enqueue a particular element $\pi_i$ does not depend on the elements after $\pi_i$ in the input sequence. Further, it is essentially the unique optimal on-line algorithm in the sense that if any other on-line algorithm $A$ deviates from $\text{Alg}_{\text{DS}}$ at any step, then there is a continuation of the sequence for which $A$ produces an output with more down-steps than $\text{Alg}_{\text{DS}}$. The algorithm runs in time $O(n \log k)$, where $n$ is the length of the input sequence and $k$ is the number of available queues. It completely characterizes the power of $k$ parallel queues in reducing the number of down-steps.

**Theorem 4.** There is an algorithm $\text{Alg}_{\text{DS}}$ that constructs an output sequence with a minimum number of down-steps. It is an on-line algorithm and can be implemented in (total) time $O(n \log k)$. Further, if any other online algorithm $A$ deviates from $\text{Alg}_{\text{DS}}$, the input sequence can be extended such that $A$ does strictly worse than $\text{Alg}_{\text{DS}}$.

**Our Motivation and Related Results**

Our research was motivated by an application in car manufacturing. The production in a car factory consists of three main stages: body (welding of the body), paint (painting the body), and assembly (adding engines, seats, ...). Nowadays, cars can be configured in many different variants (different colors, different engines, different interiors, different entertainment systems, different sensors, ...) and the sequence of cars passing through the production process must satisfy constraints of the factory floor and machinery. For example, no two subsequent cars may have the largest engine, only one out of three subsequent cars may have a roof-top, color changes cost money, and so on. Planning a sequence violating a (near) minimum number of constraints is presently an art [10,16,21].

Even if a sequence obeying all constraints is available at the beginning of the production process, this sequence will in general be perturbed in body and paint, e.g., because a welding point is not perfect and needs to be redone or the paint is not perfect and needs to be retouched. For this reason, there are buffers between body and paint and paint and assembly where the perturbed sequence can be improved to satisfy more constraints. In some factories, these buffers consist of a set of queues. This leads to our research question: “How much disorder can one remove using a given set of queues?” Surprisingly, we could not find any literature that addresses this problem for the two measures of order, i.e., LDS and number of down-steps, considered by us.

Boysen and Emde [5] study a closely related problem under the name parallel stack loading problem (PSLP). They consider only the enqueuing phase and the goal is to minimize the total number of down-steps in all queues. They also show that minimizing the number of down-steps in the enqueued sequences is NP-complete if the queue lengths are bounded: [5] gives
Improving Order with Queues

an alternative NP-completeness proof. Gopalan et al. [12] studies a streaming algorithm with the goal to approximately minimize the number of edit-operations required for transforming an input sequence into a sorted sequence. This measure is also called the Ulam metric. For a sequence of length \( n \), this measure is \( n \) minus the length of the longest decreasing subsequence. Parallel algorithms for computing the edit and the Ulam distance are given by Boroujeni et al. [4]. Aldous and Diaconis [1] discuss the many interesting properties of patience sort. Chandramouli et al. [6] give a high-performance implementation of patience sort for almost sorted data.

Organization of the paper

In Section 2, we present proofs of Theorem 2 and Theorem 3. Then in Sections 3 we present a proof of Theorem 4. We then conclude in Section 4 with a few open problems.

2 The Longest Decreasing Subsequence

We begin with the first measure of disorder of a sequence, namely the length of the longest decreasing subsequence (LDS). We are interested in the following question:

How much can one reduce the LDS of a sequence using \( k \) parallel queues?

We first present a simple algorithm based on Patience sort [1], that reduces the LDS from \( L \) to \( L - k + 1 \). Then, in Section 2.4 for each integer \( L \), we give a construction of sequences of LDS \( L \) such that it is impossible to reduce their LDS to strictly less than \( L - k + 1 \) with \( k \) queues. And finally in Section 2.2, we consider the related question of mergeability of two sequences of LDS 2 or more.

2.1 Reducing LDS via Patience Sort

▶ Theorem 5. There is an algorithm that using \( k \) queues reduces the LDS of a sequence from \( L \) to \( L - k + 1 \). The algorithm runs in \( O(n \log k) \) time.

Proof. Let us name these queues \( Q_1, Q_2, \ldots, Q_k \). We place the elements of the input sequence into the \( k \) queues such that the first \( k - 1 \) queues have LDS \( 1 \), and the last queue has LDS \( L - k + 1 \). This can be done by simulating Patience sort using \( L \) virtual queues \( Q'_1, Q'_2, \ldots, Q'_L \), and sending the elements from \( Q'_i \) to \( Q_i \) for \( i \in [k-1] \), and sending the elements of \( Q'_k, \ldots, Q'_L \) to \( Q_k \). The LDS of \( Q_k \) is \( L - k + 1 \) since it can be sorted using \( L - k + 1 \) queues, namely \( Q'_k, \ldots, Q'_L \).

Let us have a closer look at \( Q_k \). Let \( x \) and \( y \) be consecutive elements in \( Q'_k \). Then all elements in \( Q_k \) between \( x \) and \( y \) are smaller than \( x \) as an element larger than \( x \) would have been added to one of the queues \( Q'_1 \) to \( Q'_k \).

We now merge the queues \( Q_1 \) to \( Q_k \) using the Smallest Element First strategy. We claim this merges \( Q'_1, \ldots, Q'_k \) to an increasing sequence \( S \), thus, yielding an output that can be partitioned into \( L - k + 1 \) increasing sequences, namely \( S \) and \( Q'_{k+1} \) to \( Q'_L \). Indeed, assume we have just output an element \( x \) in \( Q'_k \) from \( Q_k \) and let \( y \) be the next element from \( Q'_k \) in \( Q_k \). Then the first elements of \( Q_1 \) to \( Q_{k-1} \) are larger than \( x \) and all elements in \( Q_k \) before \( y \) are smaller than \( x \). Thus they will be output next until \( y \) is at the front of \( Q_k \). Now the merge of \( Q_1 \) to \( Q_k \) will continue in the same way as a merge of \( Q'_1 \) to \( Q'_k \) until \( y \) is dequeued.
The running time of the algorithm is $O(n \log k)$, which follows from Patience sort because the tails of the queues always form a decreasing sequence and the right queue can be found using binary search \[1\].

The above algorithm is very simple. It is natural to ask whether a more sophisticated algorithm can give better results\[2\] perhaps by better balancing the LDS of the queues because the LDS in the output cannot be smaller than the maximum LDS over the $k$ queues. For example, this could be done by putting into $Q_i$, $i \in [k]$, all elements from $Q'_j$ for each $j \in [L]$ with $j \equiv i \mod k$. Then, the LDS of each of $Q_1, \ldots, Q_k$ is at most $\lceil L/k \rceil$.

We would then also have to look at a different merging strategy as Smallest Element First does not always yield the optimum merge as the following example shows. Consider $k = 2$ and the input sequence $5, 4, 1, 3, 2$ with LDS $L = 4$. The modulo procedure described above yields the two queues

- $Q_1 : 5, 1, 3$
- $Q_2 : 4, 2$

and Smallest Element First merges them to $4, 2, 5, 1, 3$ with LDS 3. However, we can also obtain $4, 5, 1, 2, 3$ with LDS 2, which is optimal as both queues have LDS 2 and at least one of the two queues must have an LDS of at least $L/k$.

Using two queues, two sequences of LDS two can always be merged into a sequence of LDS three (Theorem \[6\]) and maybe into a sequence of LDS two (the preceding example). In the next section, we show that two cannot always be obtained and derive an optimal merging strategy. We leave it as an open problem to generalize this result to more than two queues or larger LDS.

### 2.2 Mergeability of sequences

Merging two (or more) sorted sequences of numbers is central to all efficient sorting algorithms such as Mergesort \[7\]. Given $k$ queues, each containing a sorted sequence of numbers, we can always merge them to output a sorted list of all the numbers by the Smallest Element First strategy. The LDS of a sorted list is 1, and it is always possible to merge two (or more) queues of LDS 1 into an output sequence of LDS 1. Consider a generalization, where the LDS of the queues is $\ell > 1$. Then, is it always possible to merge them into an output sequence of LDS $\ell$?

The answer to this question is no. Indeed, even for 2 queues of LDS 2 it is not possible, as illustrated by the following example:

- $Q_1 : 7, 1, 9, 3$
- $Q_2 : 4, 2, 5, 8, 6$

It can be shown that any merging of these two queues will produce a sequence of LDS at least 3. In this section, we present a more general result: We give an algorithm that, given 2 queues of LDS 2, either merges them into a sequence of LDS 2, or correctly decides that they cannot be merged. We use the example above to illustrate it.

We have two sequences $Q_1$ and $Q_2$ each of LDS at most two. We want to decide whether $Q_1$ and $Q_2$ can be merged into a sequence of LDS at most two. A down-pair in a sequence is a pair $a, b$ with $a > b$ and $a$ preceding $b$ in the sequence.

We will define a precedence graph on the elements of $Q_1$ and $Q_2$. There are two kinds of edges. Strict edges and tentative edges. A strict edge $x \rightarrow y$ indicates that $x$ must be

\[2\] In fact, in Section 2.4 we show that no such strategy exists.
before \( y \) in the output. Tentative edges come in pairs that share a common element. If \( b \rightarrow e \rightarrow a \) are a pair of tentative edges then either \( b \) must be before \( e \) or \( e \) must be before \( a \) in the output; so the arrangement \( a, e, b \) is forbidden.

In a sequence of LDS at most two, we have three kinds of elements: Left elements in a down-pair, right elements in a down-pair, and elements that are in no down-pair. We call such elements solitaires. Note that no element can be a left and a right element in a down-pair as this would imply a decreasing sequence of length three. For a solitaire, all preceding elements are smaller and all succeeding elements are larger. We have the following precedence edges.

\[ \begin{align*}
\text{a)} & \quad \text{For each sequence, we have the edges } x \rightarrow y \text{ if } x \text{ precedes } y \text{ in the sequence.} \\
\text{b)} & \quad \text{Assume } a, b \text{ is a down-pair in one of the sequences and } c, d \text{ is a down-pair in the other sequence. We have the edges } \min(a, c) \rightarrow \max(a, c) \text{ and } \min(b, d) \rightarrow \max(b, d). \quad (3) \\
\text{c)} & \quad \text{Between two solitaires there are no constraints.} \\
\text{d)} & \quad \text{Assume } a, b \text{ is a down-pair in one of the sequences and } e \text{ is a solitaire in the other. We generate enough edges so that the elements cannot form a decreasing sequence of length } 3. \\
\text{1)} & \quad \text{If } e > a, \text{ then } a \rightarrow e. \\
\text{2)} & \quad \text{If } e < b, \text{ then } e \rightarrow b. \\
\text{3)} & \quad \text{If } a > e > b, \text{ then we have the pair } e \rightarrow a \text{ and } b \rightarrow e \text{ of tentative edges. We call them companions of each other. An edge may have many companions. For example, if } a, b, a', b' \text{ are down-pairs with } a' > e > b, a \rightarrow a' \text{ and } e \rightarrow a' \text{ are companions of } b \rightarrow e. \text{ Similarly, if } a, b, a', b' \text{ are down-pairs with } a > e > b \text{ and } a > e > b', b \rightarrow e \text{ and } b' \rightarrow e \text{ are companions of } e \rightarrow a. \\
\text{Tentative edges only end in solitaires or left endpoints.}
\end{align*} \]

\[ \begin{align*}
\text{Lemma 6.} & \quad \text{If the output sequence obeys all strict edges and one tentative edge of each pair of tentative edges, the output sequence has LDS at most two.} \\
\text{Proof.} & \quad \text{A decreasing subsequence of length three must consist of a down-pair } a, b \text{ from one of the sequences and an element } e \text{ from the other sequence; } e \text{ is either a solitaire or part of a down-pair } c, d \text{ with } e \in \{c, d\}. \\
& \quad \text{Assume first that } e \text{ is a solitaire. If } e > a, \text{ we have the edge } a \rightarrow e, \text{ if } b > e \text{ we have the edge } e \rightarrow b, \text{ and if } a > e > b, \text{ we have the pair of tentative edges } b \rightarrow e \rightarrow a. \text{ In either case, it is guaranteed that the three elements do not form a decreasing sequence of length three.} \\
& \quad \text{Assume next that } e \text{ is part of a down-pair } c, d. \text{ We will argue that the generated edges guarantee that the four elements } a, b, c, d \text{ do not produce a decreasing sequence of length three in the output. We have the edges } a \rightarrow b \text{ and } c \rightarrow d, \min(a, c) \rightarrow \max(a, c) \text{ and } \min(b, d) \rightarrow \max(b, d). \text{ Assume w.l.o.g. } a > c. \text{ Then we have the edge } c \rightarrow a. \text{ If } b > d, \text{ we also have the edge } d \rightarrow b \text{ and hence the two possible interleavings are } c, a, d, b \text{ and } c, d, a, b. \text{ Since } c \text{ precedes } a \text{ and } d \text{ precedes } b, \text{ a decreasing sequence can contain at most one of } a \text{ and } c \text{ and at most one of } b \text{ and } d. \text{ Thus its length is at most two. If } d > b, \text{ we have the edge } \min(b, d) \rightarrow \max(b, d). \text{ Otherwise, if } b > a, \text{ we have the edge } a \rightarrow b. \\
\end{align*} \]

\[ ^3 \text{We give two arguments why these constraints are necessary. Assume } a > c \text{ and } a \text{ precedes } c \text{ in the merge. Then } a, c, d \text{ is a decreasing sequence of length } 3. \text{ Assume } b > d \text{ and } b \text{ precedes } d \text{ in the merge. Then } a, b, d \text{ is a decreasing sequence length } 3. \text{ Alternatively, two down-pairs } a, b, c, d \text{ can either be disjoint, interleaving, or nested. Assume w.l.o.g. } a > c. \text{ If the pairs are nested, i.e., } a > c > d > b, \text{ the only possible order is } c, a, b, d. \text{ This is guaranteed by the edges } c \rightarrow a \rightarrow b \rightarrow d. \text{ If the pairs are interleaving, i.e., } a > b > c > d, \text{ or disjoint, i.e., } a > b > c > d, \text{ the two possible orders are } c, a, d, b \text{ and } c, d, a, b. \text{ This is guaranteed by the edges } a \rightarrow b, c \rightarrow d, c \rightarrow a \text{ and } d \rightarrow b. \]
b → d and hence the only arrangement is c, a, b, d. Since c precedes a and b precedes d, a decreasing sequence can contain at most one of a and c and at most one of b and d. Thus its length is at most two.

We next give an algorithm for merging two sequences. The algorithm always considers the elements at the front of the two queues and decides which one to move to the output. It may also decide that the two sequences cannot be merged. If a front element is moved to the output, all edges incident to the element are deleted and the companions of all outgoing tentative edges are deleted. We call a front element of a sequence free if it has no incoming edge, blocked if it has an incoming strict edge, and half-free if there is an incoming tentative edge, but no incoming strict edge. Here are the rules.

a) If only one of the front elements is free, move it. If both front elements are free, move the smaller one.

b) If both front elements are blocked, declare failure.

c) If one front element is blocked and the other one is half-free, move the half-free and turn the companion edges of the incoming tentative edges into strict edges.

d) If both front elements are half-free, move the smaller one. Turn the companion edges of the incoming tentative edges into strict edges.

Figure 1 illustrates the algorithm on the example from the beginning of the section.

Lemma 7. If the algorithm runs to completion, the output sequence obeys all strict edges and one of the tentative edges in each pair of tentative edges and hence the output has LDS at most two.

Proof. All strict edges are obeyed since blocked elements are never moved to the output. Also, one edge of each pair of tentative edges is realized. This can be seen as follows. When we move an element to the output all outgoing tentative edges are realized and hence we may remove their companions. When we move a half-free element to the output, the incoming tentative edges are not realized. We compensate for this by making their companions strict.

Lemma 8. If the algorithm fails, the two sequences cannot be merged.

Proof. We assume that we start with two sequences that can be merged and consider the first moment of time when the algorithm makes a move that destroys mergeability. Let e and f be the heads of the queues before the move of the algorithm. The algorithm moves e to the output, but the only move that maintains mergeability is moving f. Then e is not blocked as the algorithm moves it.

Claim 9. f is not blocked.

Proof. Assume f is blocked. Then there is a strict edge a → f from an element a in e’s queue. The strict edge was either created as a strict edge or was created as a tentative edge and later turned into a strict edge.

Assume first that either a or f is a solitaire. If the edge was created as a strict edge and f is a solitaire, then f > a and a is the left element of a down-pair a, b. The decreasing sequence f, a, b will be created, a contradiction. If the edge was created as a strict edge and a is a solitaire, then f > a and f is the right element of a down-pair g, f. The decreasing sequence g, f, a will be created. If the edge was created as a tentative edge and f is a solitaire, then there is a down-pair a', a in e’s input sequence with a' < f < a and we had the tentative edges a →→ f →→ a'. Since the edge a →→ f is now a solid edge, a' was already moved to
Improving Order with Queues

the output. Thus the decreasing sequence $a', f, a$ will be created. If the edge was created as a tentative edge and $a$ is a solitaire, then there is a down-pair $f, f'$ in $f$’s input sequence with $f > a > f'$ and we had the tentative edges $f' \rightarrow a \rightarrow f$. Since $f'$ is after $f$ in $f$’s input sequence, the tentative edge $a \rightarrow f$ is still there as a tentative edge.

We come to the case that $a$ and $f$ are elements of down-pairs. Then either both are left elements or both are right elements in their down-pairs, as we do not create edges (neither strict nor tentative) between left and right endpoints of down-pairs. Also, $a < f$ since edges always go from the smaller element to the larger element. If both are left elements and $a, b$ is a down-pair with $a$ on the left, the decreasing sequence $f, a, b$ will be created. If both are right elements and $g, f$ is a down-pair with $f$ on the right, the decreasing sequence $g, f, a$ will be created.

We have now shown that a blocked element should never be moved to the output. 

So we have to consider four cases depending on which of $e$ and $f$ are free or half-free.

Let $C = P fr Q er R$ be a correct merging; here $P$, $Q$ and $R$ are sequences. We want to argue that $M = Pf Q e R$ is also a correct merging. Assume otherwise. Then there must be a decreasing sequence of length 3 in $M$. As this decreasing sequence does not exist in $C$, it must involve $e$. All elements in $Q$ come from $f$’s queue as no element from $e$’s queue can overtake $e$ in $C$.

If the decreasing subsequence in $M$ does not involve $f$ (this is definitely the case if $e < f$), it must contain an element from $Q$, say $y$, as otherwise it would also exist in $C$.

If $f > e$, all elements in $P$ must be smaller than $f$ and all elements in $R$ must be larger than $e$, as otherwise $C$ would contain a decreasing subsequence of length 3.

$e$ and $f$ are free: We have $e < f$ since the algorithm moves $e$. So the decreasing sequence cannot contain $f$. Also, $e$ cannot be the first element of the decreasing sequence as otherwise replacing the first element by $f$ would result in a decreasing sequence of length 3 in $C$. It cannot be the last element of the decreasing sequence as the sequence contains an element of $Q$. So the decreasing sequence has the form $xey$ with $x$ in $P$ and $y$ in $Q$. Since $f > e > y$, $f, y$ is a down-pair in $f$’s sequence. If $e$ is a solitaire, we have the tentative edge $e \rightarrow f$ (Rule d3) and hence $f$ is not free, a contradiction. If $x$ is the left element in a down-pair, we have a strict edge $e \rightarrow f$ (Rule b1), a contradiction. If $e$ is the right element in a down-pair, we have the strict edge $y \rightarrow e$, a contradiction (Rule b1)

$e$ is free and $f$ is half-free: Since $f$ is half-free, there is a tentative edge $d \rightarrow f$ from an element $d$ in $e$’s queue. Thus $f$ is a solitaire or a left endpoint.

If $f$ is a solitaire, there is a down-pair $e, d$ in $e$’s sequence with $e > f > d$; $d$ is still in $e$’s queue. If $e < f$, the decreasing sequence has the form $xye$ with $x$ in $P$ and $y$ in $Q$ by the same argument as in the first case. Recall that all elements in $Q$ come from $f$’s queue. Thus $f, y$ is a down-pair, a contradiction to $f$ being a solitaire. If $e > f$, $e, d$ is a down-pair with $e > f > d$. Thus, we have the tentative edge $f \rightarrow e$ (Rule d3) and $e$ is not free, a contradiction.

If $f$ is a left endpoint, $d$ is a solitaire in $e$’s sequence. We have $f > d$. Since $d$ is a solitaire, we have $e < d$ and hence $e < f$. By the same argument as in the first case, the decreasing sequence has the form $xey$ with $x$ in $P$ and $y$ in $Q$. Thus $f, y$ is a down-pair with $f > d > e > y$. If $e$ is a left endpoint, we have the strict edge $e \rightarrow f$ (Rule b1) and $f$ is blocked. If $e$ is a right endpoint, we have the strict edge $y \rightarrow e$ (Rule b1) and $e$ is not free. If $e$ is a solitaire, we have the tentative edges $y \rightarrow e \rightarrow f$ (Rule d3) and $e$ is not free. In either case, we derived a contradiction.

$e$ is half-free and $f$ is free: This case cannot arise, as the algorithm would move $f$. 


\textbf{Lemma 10.} \emph{F is satisfiable if and only if the two sequences are mergeable.}

\textbf{Proof.} Assume \( F \) is satisfiable. Let \( E \) be the set of edges corresponding to true variables; \( E \) includes all strict edges, exactly one out of each pair of tentative edges, and at most one edge in each pair of crossing edges. Add the edges \( q_iq_{i+1} \) for \( 1 \leq i < n \) and \( p_ip_{j+1} \) for \( 1 \leq j < m \). The resulting graph is acyclic. Assume otherwise and let \( q_i \) and \( p_j \) be the elements with minimum index in a cycle. Then there must be an edge \( p_hq_i \) with \( h > j \) and \( q_kp_j \) with \( k > i \) or with \( i = k \) and \( j = h \). So we have a pair of realized crossing edges which is impossible.

Assume conversely that the sequences are mergeable. Then all strict edges are realized and exactly one out of each companion pair of tentative edges is realized. Set the variables corresponding to realized edges to true. Then the first two types of clauses are certainly true. The third type is also true since in a pair of crossing edges at most one can be realized.

For 2-SAT formulae there are simple witnesses of satisfiability and non-satisfiability [2], computable in polynomial time. We briefly review the argument. A satisfying assignment is a witness for satisfiability. For a witness of non-satisfiability, write all clauses involving two literals as two implications, i.e., replace \( x \lor y \) by \( \neg x \rightarrow y \) and \( \neg y \rightarrow x \). Clearly, all singleton clauses need to be satisfied. Do so and follow implications. If this leads to a contradiction, the formula is not satisfiable, and we have a witness of the form \( x \rightarrow^* y \) and \( x' \rightarrow^* \neg y \) for
Figure 1 The green brackets indicate down-pairs. In the left example, there is no solitaire and hence all edges are strict. The ordering 4, 7, 1, 2, 8, 9, 3, 6 is a solution. Element 4 is free and is moved to the output. Then 7 is free, then 1, then 2, and so on.

In the right example, element 5 is a solitaire and we have the pairs 1 ---→ 5 ---→ 7, 3 ---→ 5 ---→ 7 and 3 ---→ 5 ---→ 9 of tentative edges. We proceed as above and move 4, 7, 1, and 2 to the output. When we move 7, the tentative edges 1 ---→ 5 and 3 ---→ 5 are turned into solid edges. When we move 1, the first of these edges is realized. However, after removing 2, 5 and 9 are blocked by the solid edges 3 ---→ 5 and 8 ---→ 9 respectively.

In the 2-SAT problem we have to realize all solid edges. Since the solid edge 1 ---→ 2 crosses the tentative edge 5 ---→ 7, the tentative edge 3 ---→ 5 has to be realized. However, it crosses 8 ---→ 9 and hence the formula cannot be satisfied.

two singleton clauses $x$ and $x'$ and a literal $y$. After this process there are no singleton clauses. Create a graph with the literals as the nodes of the graph and the implications as the directed edges. If there is a cycle containing a variable and its complement, the formula is not satisfiable. So assume otherwise. Consider any variable $x$. We may have the implication $x \rightarrow^* \neg x$ or the implication $\neg x \rightarrow^* x$ but we cannot have both as we would have a cycle $x \rightarrow^* \neg x \rightarrow^* x$ involving $x$ and $\neg x$. If we have only the former, set $x$ to false, if we have only the latter, set $x$ to true, if we have neither, set $x$ to an arbitrary value. Follow implications. This cannot lead to an contradiction. Assume we have the implication $x \rightarrow^* \neg x$ and hence set $x$ to false and $\neg x$ to true. If this would lead to a contradiction, we would have implications $\neg x \rightarrow^* y$ and $\neg x \rightarrow^* \neg y$ for some $y$ and hence $y \rightarrow^* \neg x$ and hence $\neg x \rightarrow^* y \rightarrow^* x \rightarrow^* \neg x$.

The argument for the alternative case is symmetric.

We have now shown:

\textbf{Theorem 11.} There exist two sequences of LDS 2 that cannot be merged into a sequence of LDS 2 using two queues.

Further, there is an algorithm that given two sequences of LDS 2, either merges them into another sequence of LDS 2, or produces a witness that no such merge is possible. This algorithm runs in time polynomial in the sum of the length of the two input sequences.

Figure 1 illustrates the algorithm and the connection to 2-SAT. Figure 2 gives a more substantial example.

\section{Merging in linear time}

We give a linear time implementation of the algorithm of Section 2.2 in particular, we show that there is no need to construct the precedence graph, but that the decisions of which first element to move to the output can be made locally after a small amount of precomputation. This is the content of Lemma 12. After the Lemma, we show how to compute the information required for the refined rules in linear time.
Figure 2  The green brackets indicate down-pairs. The lower sequence contains three solitaires, namely 5, 8, and 12. The solitaire 5 is enclosed by the down-pair (6, 3) and hence we have the tentative edges 3 \rightarrow 5 \rightarrow 6. The solitaire 8 is enclosed by the down-pair (10, 7) and hence 7 \rightarrow 8 \rightarrow 10. The solitaire 12 is enclosed by (13, 11) and hence 11 \rightarrow 12 \rightarrow 13. We also have the strict edges 1 \rightarrow 2 and 15 \rightarrow 16. There are more edges not drawn here. The two sequence are not mergeable. Since we have the strict edge 1 \rightarrow 2, we cannot realize 5 \rightarrow 6. Thus we must realize 3 \rightarrow 5 and hence cannot realize 8 \rightarrow 10. Thus we must realize 7 \rightarrow 8 and hence cannot realize 12 \rightarrow 13. Thus we must realize 11 \rightarrow 12. This conflicts the strict edge 15 \rightarrow 16. Without the element 14 in the lower sequence, the two sequences can be merged into 4, 6, 1, 2, 10, 3, 5, 13, 7, 8, 16, 11, 12, 15. Instead of the solid edge 15 \rightarrow 16, we have the tentative edges 11 \rightarrow 15 \rightarrow 16. The former is realized.

Lemma 12. Let $c_i$ be the head of the $i$-th queue; here queue means the suffix of elements that are not dequeued yet.

1. If $c_i$ is smaller than all elements in the other queue, it is free.
2. If neither $c_1$ nor $c_2$ is a left element, the smaller of the two is free.
3. If $c_i$ is a left element, it is blocked iff the other queue contains a smaller left element.
   
   If there is such a left element in the other queue, it is the leftmost left element.
   
   If there is no such left element in the other queue, $c_i$ is half-blocked iff the other queue contains a solitaire $s$ such that $c_i > s > r_i$, where $r_i$ is the first right element after $c_i$.
4. We now assume that at least one first element is a left element and that both queues contain an element smaller than the first element in the other queue.
   
   a. If both first elements are left elements, the larger is blocked and the smaller is non-blocked.
   
   b. Assume the first queue starts with a left element $\ell_1$ and the second queue starts with either a right element or a solitaire. Let $r_1$ be the first right element after $\ell_1$. Then $\ell_1 > c_2 > r_1$.
   
   c. If one first element is a left element and the other is a right element, the right element is blocked.
   
   d. If one first element is a left element and the other is a solitaire and a left element larger than the solitaire has already been dequeued, the solitaire is blocked.
   
   e. If one first element is a left element and the other is a solitaire and no left element larger than the solitaire was already dequeued, the solitaire is half-blocked and smaller than the left element.
5. The above covers all cases.

Proof. 1. All constraint edges between sequences go from smaller to larger element.

2. If $c_i$ is a right element or a solitaire, all elements after $c_i$ in the $i$-th queue are larger than $c_i$. Thus, if neither first element is a left element, the smaller first element is free.

3. A solid incoming edge into a left element can only come from a smaller left element. The leftmost left element is the smallest left element in the other queue.
A tentative incoming edge can only come from a solitaire with $c_i > s > r_i$ where $r_i$ is a right element in $c_i$’s queue. If there is such an $r_i$, it is the first right element.

4. a. We have the strict edge $\min(\ell_1, \ell_2) \rightarrow \max(\ell_1, \ell_2)$ and hence the larger is blocked. The smaller is either free or half-blocked.

b. Assume the first queue starts with a left element $\ell_1$. There must be a right element $r_1$ after it in the first queue. $r_1$ is the smallest element in the first queue and hence $c_2 > r_1$. If the first element in the second queue is either a solitaire or a right element, it is the smallest element in the second queue. Thus $\ell_1 > c_2$.

c. Assume $c_1$ is a left element and $\ell_2$ is a right element. Let $r_1$ be the first right element in the first queue. Then $r_2 > r_1$ by the preceding item and hence we have a strict edge from $r_1$ to $r_2$ and hence $r_2$ is blocked.

d. Assume the first queue starts with a left element. Then $\ell_1 > s_2 > r_1$, where $r_1$ is the first right element in the first queue. If a left element $\ell'_1$ with $\ell'_1 > s_2$ was already dequeued, the tentative edge $s_1 \rightarrow \ell'_1$ was not realized and hence the tentative edge $r_1 \rightarrow s_1$ became solid.

e. Assume the first queue starts with a left element. Then $\ell_1 > s_2 > r_1$, where $r_1$ is the first right element in the first queue. All left elements already dequeued from the first queue are smaller than $s_2$ and hence all tentative edges $r \rightarrow s_1$ from a right element in the first queue into $s_2$ are still tentative. We also have the tentative edges $r_1 \rightarrow s_1 \rightarrow \ell_1$. Thus $s_1$ is half-blocked.

5. If neither first element is a left element, the smaller is free and moved to the output (item 2). If a first element is a left element, items 3 tells whether the element is free, half-blocked, or blocked. If both first elements are left elements, item 4a tells which one to move. If one first element is a left element, and the other is either a right element or a solitaire, items 4c, 4d, and 4f tell the status of the other element.

What is needed to implement the preceding Lemma?

- Each element needs to know its type: left or right element or solitaire.
- For each suffix of a queue, we need to know the smallest right element in the suffix, the smallest left element in the suffix, and the smallest solitaire in the suffix.
- The largest left element already dequeued.

Lemma 13. If the information above is available, all decisions in Lemma 12 can be made in constant time. The above information can be precomputed in linear time.

Proof. We go through the various items of Lemma 12 and argue that the decision can be made based on the information above. For the decisions in cases 1, 2, 3, 4a, 4b, 4c, and 4d, the information of the first two items suffices. For the case distinction 4e or 4f, one also needs the third item.

Item 3 needs an additional explanation. If $c_i$ is a left element and we have $c_i > s > r_i$ for some solitaire in the other queue and $r_i$ the first right element in $c_i$’s queue, $s$ is not necessarily the first solitaire in the other queue. However, if the first solitaire in the other queue is smaller than $c_i$ and $r_i$, then it is smaller than all elements in $c_i$’s queue. Thus the solitaire is free and it does not matter whether $c_i$ is free of half-blocked.

An element is a right element if there is a larger element to the left of it. We scan the sequence from left to right and maintain the maximum. Left elements are determined by a right to left scan. Elements that are neither left nor right are solitaires. For the second item, we do a right to left scan, and the third item is kept during the execution.
The algorithm above either produces an output of LDS two or stops because both first elements are blocked. Can we in the latter case at least produce an output of LDS three? Yes, we simply continue with the SMALLEST ELEMENT FIRST strategy.

**Theorem 14.** If both first elements are blocked, continuing with the SMALLEST ELEMENT FIRST strategy will produce an output of LDS three.

**Proof.** It is clear that SMALLEST ELEMENT FIRST runs to completion. So we only have to show that the output has LDS three. We first show that among two left elements, the smaller is always output before the larger. This is clear if they come from the same queue as the smaller precedes the larger. If they come from different queues and one of them is output before the algorithm blocks, this holds true, because we have a strict edge from the smaller to the larger. So assume that both are output by SMALLEST ELEMENT FIRST. Let \( \ell_1 \) and \( \ell_2 \) be two left elements with, say, \( \ell_1 < \ell_2 \), and assume that \( \ell_2 \) is output before \( \ell_1 \) by SMALLEST ELEMENT FIRST. Consider the situation just before \( \ell_2 \) is output. All elements preceding \( \ell_1 \) in its queue are smaller than \( \ell_1 \) and hence SMALLEST ELEMENT FIRST will output them and \( \ell_1 \) before \( \ell_2 \).

So a decreasing sequence can involve at most one left element. It can also involve at most one solitaire. So, if it has length four, it must involve two right elements, one from each queue. One of them must come from the same queue as the solitaire. But this is impossible, because solitaires are larger than all preceding elements and smaller than all succeeding elements. ▶

### 2.4 The limits of reducing LDS

The above algorithm can be used for the dequeuing step in algorithm that is supposed to reduce the LDS of a given sequence with an LDS of at most 4. However, it is not straightforward to see whether we can always enqueue such a given sequence into two queues so that both queues contain mergable sequences. It turns out this is not the case – there exist sequences of LDS \( L \), such that with \( k \) queues it is impossible to reduce it to a value lower than \( L - k + 1 \). We will first show this result for the case \( k = 2 \) in order to make it easier to build an intuition for the proof. Afterwards, we present the general case. We begin with the following definition.

**Definition 15.** Given a sequence \( A \), we define \( \Lambda_k(A) \) to be the lowest possible obtainable LDS of a sequence generated by enqueuing and dequeuing the sequence \( A \) using \( k \) parallel queues.

Let \( L \in \mathbb{N} \). In this section, we prove that for each \( L \in \mathbb{N} \), there is a sequence \( A \) of LDS \( L \) such that \( \Lambda_k(A) \geq L - k + 1 \). The proof is constructive, and the length of the constructed sequence \( A \) is \( \Theta(L^{2k-1}) \). Formally,

**Theorem 16.** For every \( L \) and \( k \), there exists a sequence \( A \) of LDS \( L \) such that \( \Lambda_k(A) \geq L - k + 1 \). The length of \( A \) is \( \Theta(L^{2k-1}) \).

We first present a construction when \( k = 2 \), and then extend it to any fixed \( k \). We then show that the bound the length of the constructed sequence. Towards this, we briefly recall the definition of the direct and skew sum of permutations, that is required for our construction. Given two permutations \( \pi \) and \( \sigma \), and their respective permutation matrices \( M_\pi \) and \( M_\sigma \), the direct sum and skew sum of these two permutations, in terms of permutation matrices, take the form

\[
M_\pi \oplus M_\sigma = \begin{bmatrix} M_\pi & 0 \\ 0 & M_\sigma \end{bmatrix} \quad \text{and} \quad M_\pi \ominus M_\sigma = \begin{bmatrix} 0 & M_\pi \\ M_\sigma & 0 \end{bmatrix},
\]

respectively.
Equivalently, we can define the direct sum of two permutations $\pi$ and $\sigma$ as

$$\pi \oplus \sigma = \pi \parallel \sigma',$$

where $\sigma'$ is obtained from $\sigma$ by adding $|\pi|$ to all its elements, and the skew sum as

$$\pi \ominus \sigma = \pi' \parallel \sigma,$$

where $\pi'$ is obtained from $\pi$ by adding $|\sigma|$ to all its elements.

### 2.4.1 Lower bound with 2 queues

Here, we fix an integer $L \in \mathbb{N}$ as the LDS of the constructed sequence. We present a permutation $\pi$ with LDS $L$ such that any manner of enqueueing it into $k = 2$ queues and then dequeueing it yields an outgoing permutation of LDS at least $L - 1$. We prove this claim by induction on the value of the LDS.

**Observation 17.** The following two observations are crucial for the understanding of the lower bound with 2 queues and with $k$ queues.

- Consider the direct sum $\pi \oplus \sigma$. In the resulting sequence, all the elements that originally belonged to $\pi$ are lower than those which originally belonged to $\sigma$. Furthermore, the LDS of $\pi \oplus \sigma$ is the maximum between the LDS of $\pi$ and the LDS of $\sigma$.

- Consider the skew sum $\pi \ominus \sigma$. In the resulting sequence, all the elements that originally belonged to $\pi$ are greater than those which originally belonged to $\sigma$. Furthermore, the LDS of $\pi \ominus \sigma$ is the sum of the LDS of $\pi$ and the LDS of $\sigma$.

**Theorem 18.** For every $L \in \mathbb{N}$, there exists a permutation $\pi$ with LDS $L$, such that $\Lambda_2(\pi) \geq L - 1$.

**Proof.** We set the base case to be $L = 3$. In this case $\pi = [3, 2, 1]$. It is easy to see that the LDS is 3 and that with only two queues it’s not possible to obtain an output LDS lower than 2. In other words it’s not possible to perfectly sort this sequence in ascending order.

To proceed with the inductive step, for each $L \in \mathbb{N}$ denote $A_L$ to be a specific permutation of LDS equals to $L$ such that $\Lambda_2(A_L) \geq L - 1$. A recursive formulation for $A_L$ is the following

$$A_{L+1} = (A_L \oplus D_L \oplus \cdots \oplus D_L) \ominus [1]$$

Where $D_L$ denotes the decreasing sequence of length $L$, and the corresponding permutation matrix is simply the anti-diagonal matrix of size $L$. The permutation matrix of $A_L$ is of the form

$$A_{L+1} = \begin{bmatrix}
0 & A_L & 0 & 0 & \cdots & 0 \\
0 & 0 & D_L & 0 & \cdots & 0 \\
0 & 0 & 0 & D_L & \cdots & 0 \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & 0 & 0 & \cdots & D_L \\
1 & 0 & 0 & 0 & \cdots & 0
\end{bmatrix}$$

From Observation 17 it follows that $A_{L+1}$ is a permutation of LDS $L + 1$. We prove that $\Lambda_2(A_{L+1}) \geq L$. We refer to $B_i$ and $C_i$ as the subsequences of the $i$-th copy of $D_L$ which have been enqueued into the first and the second queue respectively, where $1 \leq i \leq L - 1$. In
the same way, we call X and Y the sequences in the first and second queue dequeued from
A_L. W.l.o.g. assume that the element 1 is pushed in the first queue. This means that it is
dequeued after X and all the sequences B_j. We consider two following cases:

- The element 1 is dequeued after the last element of Y: Since the elements of X are in
  the same queue as the element 1, the 1 is also dequeued after the last element of X.
  Therefore, all the elements of X and Y will appear before the element 1 in the final
  sequence. By the inductive hypothesis, merging the two sequences X and Y will create
  a decreasing sequence of length L − 1. Since the element 1 is dequeued afterwards, we
  obtain a decreasing sequence of length L.

- The element 1 is dequeued before the last element of Y: Denote the LDS of the sequences
  B_1, B_2, ..., B_{L−1} as \(L_1, ..., L_{L−1}\), respectively. If any \(L_i \geq L − 1\), when the element 1 is
  appended at the end of the sequence, the sequence \(B_i \cup \{1\}\) will have an LDS \( \geq L\),
  trivially satisfying the claim of the theorem. Similarly, if \(L_i = 0\), the sequence \(C_i = D_L\),
  which has the LDS = L. We can therefore assume that \(1 \leq L_i \leq L - 2\) \(\forall i \in [L−1]\).
  By pigeonhole principle, there exists a pair \(i, j\), with \(i < j\), such that \(L_i = L_j\). Notice that,
  by the construction of \(A_{L+1}\), all the elements of \(B_j\) are greater than all the elements of \(C_i\).
  The element 1 is dequeued before the last element of Y', hence it is dequeued before any
  sequence \(C_i\). This means that all the sequences \(B_j\) are dequeued before all the sequences
  \(C_i\). Since \(L_i = L_j\), it's easy to see that the LDS given by the concatenation of \(B_j\), having
  LDS of \(L_i\), and \(C_i\), having LDS of \(L − L_i\), is equal to L, which proves the claim.

To analyze how quickly the size of \(A_L\) grows, we can compute its length recursively.
It is composed by the concatenation of \(A_{L−1}\), with unknown length, and \(L - 2\) sequences
\(D_{L−1}\), with length \(L − 1\), and finally of the element 1. Trivially, we obtain that \(\text{size}(A_L) = \text{size}(A_{L−1}) + (L - 2) \cdot (L − 1) + 1\), where \(\text{size}(A_3) = 3\). One can easily check that the closed
form is \(\text{size}(A_L) = \frac{L(L^2−3L+5)}{3} − 2\).

2.5 Lower bound in the generalized case with k queues

In this part, we present a permutation \(\pi\) with LDS \(L\) such that \(\Lambda_1(\pi) \geq L - k + 1\). The
intuition is similar to the case where \(k = 2\). The primary difference is that for \(k = 2\) we had
\(L - 1\) repetitions of \(D_L\) in the construction of \(A_L\). Instead of \(L - 1\) repetitions, we have
\((L − k + 1)(k−1)\) repetitions for a general \(k\).

**Definition 19.** Define a sequence of permutations \(\{B^k_L\}_{L=1}^{\infty}\) by following rules:

- For any L, set \(B^1_L\) to be a decreasing sequence of length L.
- For each \(1 \leq L \leq k + 1\), set \(B^k_L\) to be a decreasing sequence of length L.
- For every \(L \geq k + 2\), with \(k \geq 2\), consider the following recursion for \(B^k_{L+1}\):
  \[
  B^k_{L+1} = (B^k_L \oplus B^{k−1}_L \oplus \cdots \oplus B^1_L) \ominus [1].
  \]

Observe that, for \(k = 2\), the recursion above is the same as in previous subsection.
Similarly to the case for \(k = 2\), one can easily show that the LDS of \(B^k_L\) is exactly \(L\). So it
remains to prove that \(\Lambda_k(B^k_L) \geq L - k + 1\) for all \(L \geq k + 1\). Note that for every \(L \leq k\), it’s
possible to yield a sorted sequence as the result. It is also easy to see that for \(L = k + 1\), the
lowest obtainable LDS is \(2 = L - k + 1\), because, by the pigeonhole principle, at least two
elements will be enqueued on the same queue.

Before proceeding with the main Lemma, we first state and prove some lemmas that will
turn useful later.
Definition 20. Sequences $A'$ and $A''$ are called a split of $A$ if $A'$ and $A''$ are disjoint subsequences of $A$ and $A$ is an interleaving of $A'$ and $A''$.

Lemma 21. For any split $A, B$ of some sequence $C$, if $\Lambda_s(A) = \ell$ and $\Lambda_{s'}(B) = t$, then $\Lambda_{s+s'-1}(C) \leq \ell + t$.

Before proceeding with the proof we make some remarks and claims to make the proof more intuitive and easier to verify. Consider a sequence $D$ being completely enqueued and completely dequeued through $k$ queues, and resulting in a sequence $D'$. Let’s denote the enqueuing protocol, that is, the policy used to enqueue $D$, $S_e$, and the dequeuing protocol $S_d$. We can encode $S_e$ into a string, where the $i$-th element is an integer, indicating the index of the queue where the $i$-th element of $D$ should be enqueued to. For example, the string $\{1, 3, 2\}$ encodes a protocol that enqueues the first element into $Q_1$ the second element into $Q_3$ and the third element into $Q_2$. We can encode $S_d$ in the same way, but this time, each element indicates the index of the queue the $i$-th element of the resulting sequence should be dequeued from.

Claim 22. The following two ways to execute the protocols are equivalent and they both result in the same sequence $D'$.

1. First enqueue all the elements of $D$ according to $S_e$ and afterwards dequeue them according to $S_d$.
2. If the next element in $S_d$ is available in the respective queue, dequeue it, otherwise enqueue the next element according to $S_e$. Repeat until the starting sequence and all the queues are empty.

Proof. The first case coincides with the construction of $S_e$ and $S_d$. Therefore, it trivially leads to $D'$. By executing the protocol $S_e$, we are iterating over its encoded string until we reach the end, and afterward we iterate over the string of $S_d$.

Assume the resulting sequence $D''$ of the second case is different, and that the first position where it differs is $i$. It means that the $i$-th element which gets dequeued is different in $D'$ and $D''$. Since in both cases we used the same protocol $S_d$, the queue from which this element is dequeued is the same. Since we also execute the same enqueuing protocol $S_e$ as well, the relative ordering of the elements inside the same queues is preserved and cannot result in different elements at the same position.

Claim 23. In the second case of Claim 22 at each step, there is always at least one queue that either is empty, or that has a single element which is dequeued as soon as it is enqueued.

Proof. If all the queues have at least one element which is not immediately dequeued, it means that at the head of all the queues there is an element, and that no one of them can be dequeued. By enqueuing more elements in the queues, the elements at the head of the queues don’t change and the dequeuing protocol is blocked and cannot be executed, which, by construction, is a contradiction.

Notice that the queue is not necessarily the same all the time, however, the protocol can be modified to achieve this. We follow the same terminology of [17] and we denote this queue as express queue.

Claim 24. It is always possible to modify the protocols $S_e$ and $S_d$ such that the result is the same and such that there is a fixed queue that is always either empty or with a single element that is immediately dequeued.
We can now dequeue the elements of $Q_2$. Then, $Q_2$ gets emptied, and the dequeuing protocol requires an element from $Q_1$. If the enqueuing protocol enqueues elements in $Q_1$, the dequeuing protocol is still blocked by $Q_2$, which becomes the new express queue. In order to fix an express queue, the enqueuing and dequeuing protocol must be modified such that, whenever there is more than one empty queue, the last queue to receive an element from the enqueuing protocol is the desired express queue. By doing so, the fixed queue cannot be blocked by other empty queues, because it is the last one to stay empty.

We proceed with the proof of Lemma 21.

Proof. After enqueuing and dequeuing $A$ through $s$ queues we can obtain a sequence $A'$ with LDS of $\ell$. Denote the protocol to get this result as $S'$. Similarly, denote the protocol for $B$ that results in $B'$ with LDS of $t$, as $S''$. Now modify $S'$ and $S''$ according to Claim 24 such that they both have an express queue. W.l.o.g. we can assume that the express queue is the first queue in both cases. Now we show how to enqueue-dequeue $C$ through $s + s' - 1$ queues in order to get an LDS of $\ell + t$.

Enqueue the $i$-th element $e$ of $C$ as follows:

- If $e$ is an element of $A$, enqueue it according to $S'_d$;
- If $e$ is an element of $B$, let $q \in [s']$ be the index of the queue where it is supposed to be enqueued according to $S''_e$:
  - If $q = 1$, i.e., the express queue, enqueue $e$ in the first queue;
  - Otherwise, enqueue $e$ in the queue with index $q + s - 1$ and modify the dequeuing protocol $S''_d$ accordingly.

We can now dequeue the elements of $C$ according to $S'_d$ and $S''_d$ where the origin of the head of the first queue determines which of the two is currently active. We thereby obtain a sequence $C'$ with an LDS that is at most the sum of the LDS of $A'$ and $B'$, i.e., $\ell + t$. ▶

Lemma 25. For any split $A', A''$ of some sequence $A$, if $\Lambda_k(A) = L$ and $\Lambda_\ell(A') = \ell$, with $s < k$, then $\Lambda_{k-s+1}(A'') \geq L - \ell$.

Proof. For sake of contradiction, assume $t := \Lambda_{k-s+1}(A'') < L - \ell$. By Lemma 21 we can enqueue-dequeue $A$ with $k = s + k - s + 1 - 1$ queues, such that the LDS of the result is at most $\ell + t < \ell + L - \ell = L$, a contradiction to $\Lambda_\ell(A) = L$. Therefore, $t \geq L - \ell$. ◄

This Lemma is crucial for the proof and it encloses the main idea. The intuition is that we are going to split a sequence $B_{L-1}^k$ in two parts. The first part is sorted with $s$ queues and leads to an LDS of $\ell$, the second part is going to be sorted with $k-s$ queues and leads to an LDS of $L - k + 2 - \ell$, because we will assume by induction that $\Lambda_{k-1}(B_{L-1}^{k-1}) = L - k + 2$. The fact that if we split it in two parts, the sum of the LDSs, $\ell$ and $L - k + 2 - \ell$, is at least the original LDS, i.e., $L - k + 2$, will be the main point. Now we can state and prove the main Theorem.

Theorem 26. $\Lambda_k(B_L^k) \geq L - k + 1$ for all $k \geq 1$ and for all $L \geq k + 1$.

Proof. The proof follows by double induction on $L$ and $k$. First we set the base case for $k = 1$ and we prove the claim for every $L$. Recall that in this case $B_L^1$ is a decreasing sequence of length $L$. Obviously with a single queue the sequence does not change and the resulting sequence has an LDS of $L = L - k + 1$.
Now we set the base case for \( L = k + 1 \) and we prove the claim for every \( k \). We have already argued that, by the pigeonhole principle, the lowest possible obtainable LDS for \( B_k^{k+1} \) with \( k \) queues is \( 2 = L - k + 1 \).

Now we proceed with the induction step and we set \( k \geq 2 \) and \( L \geq k + 2 \). We want to prove that \( \Lambda_k(B_{k+1}^L) \geq L - k + 2 \), and we will make the following two assumptions by induction:

- \( \Lambda_k(B_{L+1}^L) \geq L - k + 1 \);
- \( \Lambda_{k-1}(B_{L+1}^{L-1}) \geq L - (k - 1) + 1 = L - k + 2 \).

Let’s denote \( P_i^j \) to be elements of the \( i \)th copy of \( B_{L+1}^{L-1} \) which are enqueued to the \( j \)th queue, with \( 1 \leq i \leq (L - k + 1)(k - 1) \). Similarly, let’s also denote \( P_0^j \) to be the elements of \( B_{L}^1 \). For example, in the proof of Theorem 18, \( P_i^j \) is completely dequeued before \( 1 \) or after all elements of each \( P_i^j \). So we are left with the case that \( \Lambda_k(B_{L+1}^L) \geq L - k + 2 \), and \( X \), in the first queue, would be \( P_0^1 \) in this case. W.L.O.G, assume that the element 1 is enqueued in the first queue.

Before proceeding, we prove another intermediate claim. For the sequel, we split the \((k - 1)(L - k + 1)\) occurrences of \( B_{L+1}^{L-1} \) into \( k - 1 \) disjoint blocks of \( L - k + 1 \) consecutive occurrences each. For a block \( Q \), we use \( Q^j \) to denote the elements that are enqueued into the \( j \)th queue.

\( \triangleright \) **Claim 27.** Given the construction of \( B_{L+1}^k \), at least one of the following is true:

1. All the sequences \( P_0^j \) are dequeued before the element 1;
2. There is a block \( Q \) such that each \( Q^j, j \in [k] \), is either completely dequeued before 1 or completely dequeued after 1. Moreover, at least one \( Q^j \) is completely dequeued before 1 and one is completely dequeued after 1.

**Proof.** Assume we are not in case 1, i.e., 1 is dequeued before the last element of some \( P_0^p \). Then \( p > 1 \), since 1 is dequeued after all elements in the first queue. Say that a queue \( j \) kills a block \( Q \) if 1 is dequeued after the first element of \( Q^j \) and before the last of \( Q^j \). Then queues 1 and \( p \) kill no block and any other queue can kill at most one block. So one block is not killed as we have \( k - 1 \) blocks but only \( k - 2 \) queues that can kill a block. \( \triangleright \)

We consider the two cases separately:

1. Recall that the sequences \( P_0^j \) are the result of enqueuing the elements of \( B_{L}^k \). By the inductive hypothesis \( \Lambda_k(B_{L}^k) \geq L - k + 1 \). By assumption 1 is output after all elements of \( B_{L}^k \) and hence an LDS of at least \( L - k + 2 \) will result.

2. We have a block \( Q \) of \( L - k + 1 \) occurrences of \( B_{L+1}^{L-1} \) such that 1 is either dequeued before or after all items of each \( Q^j \). Also 1 is dequeued after all elements in \( Q^1 \) and before all elements of \( Q^p \) for some \( p > 1 \).

We may assume that there is an \( s \) with \( 2 \leq s \leq k - 1 \) such that the \( Q^j \) with \( 1 \leq j \leq s \) are dequeued before 1 and the \( Q^j \) with \( s + 1 \leq j \leq k \) are dequeued after 1.

For \( i \in [L - k + 1] \), let \( P_i^1 \) be the \( i \)th occurrence of \( B_{L}^{L-1} \) in \( Q \) and let \( P_i^j \) be elements of the \( i \)th occurrence which are enqueued in queue \( j \). If some \( P_i^j \) is empty, \( P_i^j \) is enqueued into fewer than \( k \) queues and hence the result has an LDS of at least \( L - k + 2 \) by induction hypothesis.

We use \( P_i^j \) to denote the ensemble \( P_i^j, j \in [s] \). Let \( \ell_i \) be the minimum LDS that any dequeueing of \( P_i^j, j \in [s] \) can achieve. Since \( P_i^j \) is non-empty for all \( j, \ell_i \geq 1 \).

Assume next that there is an \( \ell_i \geq L - k + 1 \). Then together with the element 1 we obtain a decreasing subsequence of length \( L - k + 2 \).

So we are left with the case that \( 1 \leq \ell_i \leq L - k \) for all \( \ell \). Therefore there must be indices \( u \) and \( v \) such that \( 1 \leq u < v \leq L - k + 1 \) and \( \ell_u = \ell_v \). Note that all elements in \( P_v \) are...
greater than all elements in $P_u$. Note also that we are not claiming that $P_u$ and $P_v$ are enqueued in the same way. We are only claiming that both enqueuings give the same LDS.

We next apply Lemma 25 to $A = P_u$, $A'$ the subsequence of $P_u$ that is enqueued into the first $s$ queues and $A''$ the subsequence that is enqueued into the other $k-s$ queues. Since $\Lambda_{k-1}(B_L^{k-1}) = L - k + 2$ and $\Lambda_s(A') = \ell_u$, we have $\Lambda_{k-s}(A'') = \Lambda_{k-1-s+1}(A'') \geq L - k + 2 - \ell_u$.

Now we are done. In the output, we have a decreasing sequence of length $L - k + 2 - \ell_u$ resulting from the dequeuing of $P_v^j[s]$ followed by a decreasing sequence of length $L - k + 2 - \ell_u$ resulting from the dequeuing of the $P_u^j$, $s < j \leq k$. Note that all elements in the former sequence are larger that all elements in the latter sequence. Hence an LDS of $L - k + 2$ results.

This completes the proof of this theorem. \hfill $\blacksquare$

2.5.1 Growth of the sequence

We now show that the length of the sequence $B_L^k$ is $\Theta(L^{2k-1})$. Denote by $S_L^k$ the length of the sequence $B_L^k$. For the case $k = 2$ we have seen that $S_L^2 = L(L^2 - 3L + 5)/2 \in \Theta(L^3)$. For the general case, given the construction of $B_L^k$, $S_L^k$ will be of the following form:

$$S_L^k = S_{L-1}^k + (L - k)(k - 1)s_{L-1}^{k-1} + 1.$$

By expanding the recursive formula only on the first term, where only one index is decremented, we can reduce it to

$$S_L^k = L + (k - 1) \sum_{i=2}^{L-k} iS_{k-1+i}^{k-1} = L - (k - 1)k + (k - 1) \sum_{i=k}^{L-1} (i - k + 1)S_i^{k-1}$$

where we used that $S_{k+1}^k = k + 1$ and that $S_k^{k-1} = k$. Using this rewriting of the formula, we can prove the claim by induction.

The base case $S_L^1$ is trivial, as $S_L^1 = L$. We now assume that $S_{L-1}^{k-1} \in \Theta(L^{2k-3})$. The term that will result in the highest degree is $\sum_{i=k}^{L-1} iS_i^{k-1}$. In particular, $iS_i^{k-1} \in \Theta(i^{2k-2})$, by the inductive hypothesis. Since we sum the indices $i$ up to $L$, we get a polynomial one degree higher. Hence $S_L^k \in \Theta(L^{2k-1})$.

3 Minimizing Downsteps

With the results of the previous section, it is not clear whether having several multilane buffers in a row is better than having all lanes in parallel in a single buffer. However, with the measure of disorder that we consider in this section, namely the number of downsteps, we will see that a logarithmic number of multilane buffers in a row suffice to sort a sequence perfectly. Moreover, we present an optimal algorithm $\mathcal{A}_{DS}$ for minimizing the number of downsteps in a sequence using $k$ queues. This algorithm runs in time $O(n \log k)$, where $n$ is length of the sequence. Further, this algorithm is online.

The algorithm is as follows: Recall that we have $k$ queues and our input is a sequence of length $n$. For simplicity we assume that all numbers in the input are non-negative. We define the enqueuing and dequeuing strategy of our algorithm $\mathcal{A}_{DS}$ and then prove its optimality.
Improving Order with Queues

Enqueuing:

The queues are numbered 0 to \( k - 1 \). We denote the last element in queue \( i \) by \( \ell_i \). Initially, when all queues are empty, \( \ell_i \) has the fictitious value \( -i - 1 \). One of the queues is the base queue; we use \( b \in [k - 1] \) to denote the base queue. Initially, \( b = 0 \). We maintain the following invariant at all times:

\[
\ell_b > \ell_{b+1} > \ldots > \ell_{b+k-1}.
\]

Indices are to be read modulo \( k \). Note that the fictitious values are chosen such that the invariant holds initially.

Assume now that element \( e \) is to be enqueued. Let \( i \in \{0, 1, \ldots, k - 1\} \) be minimal (if any) such that \( e > \ell_{b+i} \). If \( i \) exists, either \( i = 0 \) and \( e > \ell_b \) or \( i > 0 \) and \( \ell_{b+i-1} > e > \ell_{b+i} \).

- If \( i \) exists, we append \( e \) to queue \( b+i \). This does not create a down-step and maintains the invariant.
- If \( i \) does not exist, i.e., \( \ell_{b+k-1} > e \), we append \( e \) to queue \( b \) and increase \( b \) by 1. This creates a down-step in queue \( b \). The invariant is maintained.

A run is an increasing sequence. At the end of the enqueuing all queues contain an equal number of runs up to one. Note that the down-steps are generated in round-robin fashion. The first down-step is created in queue 0, the second in queue 1, and so on. The number of runs in a queue is one more than the number of down-steps in the queue.

The enqueuing strategy is inspired by Patience Sort \([1]\). Patience Sort sorts a sequence of numbers using a minimum number of queues. It is as above with one difference: if \( e \) cannot be appended to an existing run, then it opens up a new queue.

Dequeuing:

We first merge the first runs in all queues into a single run, then merge the second runs in all queues, and so on. So the number of runs in the output sequence is the maximum number of runs in any of the queues.

It is clear that the algorithm can be made to run in time \( O(n \log k) \). We keep the values \( \ell_i \) in an array \( A \) of size \( k \). When an element \( e \) is to be enqueued, we perform binary search on the sorted array \( A[b..b+k-1] \); again indices are to be read modulo \( k \).

▶ Theorem 28. Algorithm \( \text{Alg}_\text{DS} \) constructs an output sequence with a minimum number of down-steps. It is an on-line algorithm and can be implemented to run in time \( O(n \log k) \). If an online algorithm \( A \) deviates from \( \text{Alg}_\text{DS} \), the input sequence can be extended such that \( A \) does worse than \( \text{Alg}_\text{DS} \).

We give three quite different proofs for the first claim of the theorem in Sections 3.1, 3.2, and 3.4 respectively. The first proof is the shortest, but also the least informative. It shows that the execution of any algorithm can be transformed into an execution of \( \text{Alg}_\text{DS} \) without increasing cost. The second proof uses a potential argument and establishes that \( \text{Alg}_\text{DS} \) is the unique optimal online algorithm. The third proof constructs witnesses for down-steps. In Section 3.3 we characterize the runs generated by the algorithm in the different queues without reference to the algorithm.

3.1 Transformation to \( \text{Alg}_\text{DS} \)

Let \( A \) be any algorithm. We will show that the queue contents constructed by \( A \) can be transformed into the contents constructed by \( \text{Alg}_\text{DS} \) without increasing cost.
Let \( t \) be minimal such that \( A \) and \( \text{Alg}_{\text{DS}} \) differ, i.e., \( A \) enqueues \( e = e_t \) into queue \( i \) and \( \text{Alg}_{\text{DS}} \) enqueues \( e \) into queue \( j \) with \( i \) different from \( j \). We concentrate on the \( i \)-th and the \( j \)-th queue. Let \( a_i \) and \( a_j \) be the contents of these queues just before enqueuing \( e_t \) and let \( s_i \) and \( s_j \) be their continuations by \( A \). \( s_i \) starts with \( e \). We swap \( s_i \) and \( s_j \), i.e., the contents of the \( i \)-th queue become \( a_i s_j \) and the contents of the \( j \)-queue becomes \( a_j s_i \). The number of down-steps stays the same except maybe for the down-steps at the borders between the \( a \)-parts and the \( s \)-parts.

We use \( \ell(a_i), \ell(a_j), f(s_i), \) and \( f(s_j) \) to denote the last and first elements of \( a_i, a_j, s_i \) and \( s_j \), respectively. We have \( e = f(s_i) \). We need to show that the number of down-steps in the pairs \( (\ell(a_j), f(s_i)) \) and \( (\ell(a_i), f(s_j)) \) is no larger than in the pairs \( (\ell(a_i), f(s_i)) \) and \( (\ell(a_j), f(s_j)) \).

We distinguish cases according to the action of \( \text{Alg}_{\text{DS}} \). If \( e \) is smaller than the last elements of all existing parts, \( \text{Alg}_{\text{DS}} \) incurs a down-step and adds \( e \) to the queue with largest last element. Otherwise, \( \text{Alg}_{\text{DS}} \) appends \( e \) to the queue whose last element is largest among the last elements smaller than \( e \).

**e is smaller than all last elements**: Then all \( k \) queues have a proper last element since \( e \) is a non-negative number and fictitious last elements are negative. Thus \( \ell(a_j) > \ell(a_i) > e \) where the first inequality follows from \( i \neq j \) and the fact that the \( j \)-th queue has the largest last element and the second inequality holds because \( e \) is smaller than all last elements. Thus we have a down-step in \( a_i s_j \) and in \( a_j s_i \). Let \( e' = f(s_i) \). If we have no down-step in \( a_j s_j \), then we have no down-step in \( a_i s_j \) and hence the swap does not increase the cost. If we have a down-step in \( a_j s_j \), then \( A \) incurred two down-steps before the swap and hence the swap cannot increase the cost. The swap decreases the number of down-steps if \( \ell(a_j) > e' > \ell(a_i) \).

**e is larger than some last element**: We have \( \ell(a_j) < e = f(s_i) \), because \( \text{Alg}_{\text{DS}} \) enqueues after the largest last element that is smaller than \( e \). Thus, we have no down-step at the border from \( a_j \) to \( s_i \) and hence \( A \) incurs at most one down-step after the swap. So, we only need to show that if the number of down-steps at the borders is zero before the swap then it is zero after the swap.

Assume the number is zero before the swap. Then \( \ell(a_i) < e = f(s_i) \) and \( \ell(a_j) < f(s_j) \). Also \( \ell(a_i) < \ell(a_j) < e \), since \( \text{Alg}_{\text{DS}} \) enqueues \( e \) into the queue with largest last element smaller than \( e \). Thus \( \ell(a_i) < f(s_j) \) and \( \ell(a_j) < f(s_i) \) and no down-step will be introduced by the swap.

We summarize:

\[\text{Theorem 29.} \quad \text{No algorithm creates a smaller total number of down-steps over all queues than algorithm } \text{Alg}_{\text{DS}}. \quad \text{Algorithm } \text{Alg}_{\text{DS}} \text{ generates an output with a minimal number of down-steps.}\]

**Proof.** We have shown how to convert the enqueuing of any algorithm into the enqueuing of \( \text{Alg}_{\text{DS}} \) without increasing the total number of down-steps. \( \text{Alg}_{\text{DS}} \) generates the same number of down-steps up to one in all queues and the number of down-steps in the output is equal to the maximum number of down-steps in any queue. Thus, every algorithm must generate at least as many down-steps in the output as \( \text{Alg}_{\text{DS}} \).

### 3.2 A Potential Function Argument

We will use a potential function argument to show that no algorithm can create fewer down-steps than \( \text{Alg}_{\text{DS}} \). We will also show that \( \text{Alg}_{\text{DS}} \) is the unique optimal on-line algorithm.
Let $A$ be any other algorithm. Let $a_0 \rightarrow a_{k-1}$ be the last elements in the queues of algorithm $A$ and let $\ell_1 > \ell_2 > \cdots > \ell_{b+k-1}$ be the last elements in the queues of $\text{Alg}_{\text{DS}}$: $b$ is the base queue. Let $d_A$ be the number of down-steps created by $A$ and let $d_O$ be the number of down-steps created by $\text{Alg}_{\text{DS}}$. As the elements are added to queues by both algorithms, we maintain a bijection $\pi : \{0, \ldots, k-1\} \rightarrow \{0, \ldots, k-1\}$ and call an index $i$ good if $a_{\pi(i)} \geq \ell_i$. Otherwise, the index is bad. We show that the following invariant is maintained:

$$d_A - d_O + \max_{\pi} \{i : a_{\pi(i)} \geq \ell_i \} \geq k.$$ 

The invariant holds initially, since all indices are good (we initialize all $\ell_i$ and $a_i$ with fictitious elements $-(i+1)$) and $d_A = d_O = 0$. We use $d$ to denote $d_A - d_O$.

Assume now we enqueue an element $e$. We may also assume that $b = 0$; otherwise renumber the queues of $\text{Alg}_{\text{DS}}$. We may assume that $\pi$ is the identity; otherwise renumber the queues of $A$. Let $a_0', a_1', \ldots, a_{k-1}'$ and $\ell_0', \ell_1', \ldots, \ell_{k-1}'$ denote the last elements of the queues of $A$ and $\text{Alg}_{\text{DS}}$, respectively, after adding $e$. Further, let $\pi'$ denote the bijection after adding $e$.

**Case, $e < \ell_{k-1}$**: Then $\text{Alg}_{\text{DS}}$ enqueues $e$ into queue 0 and $\ell_0' = e$. Algorithm $A$ adds $e$ to queue $j$; note that $j = 0$ is possible. Then $a_j' = e$. We modify $\pi$ to $\pi'$ by letting 0 map to $j$ and $j$ map to 0 (if $j = 0$, we do not modify $\pi$).

$\text{Alg}_{\text{DS}}$ incurs a down-step and $d_O$ increases by 1. Further the index $e$ is a good index after adding $e$, since $a_j' = e = \ell_0'$. Note that if $j$ was a good index, i.e., $a_j \geq \ell_j$, algorithm $A$ also incurs an down-step since $a_j \geq \ell_j \geq \ell_{k-1} > e$.

Observe that, if $j = 0$ and 0 was a good index before the addition, the invariant is maintained since both algorithms incur a down-step and 0 stays a good index. Otherwise if $j = 0$ and 0 was a bad index before the addition, then the number of good indices is increased by one and $d$ decreases by at most one. The invariant is maintained when $j = 0$.

So assume $j \neq 0$. If neither 0 nor $j$ was a good index before, the invariant is maintained since $d$ decreases by at most one and the number of good indices increases by at least one.

Assume next that exactly one of the indices was good: If 0 was good and $j$ was not, $a_0 \geq \ell_0$ and $a_j < \ell_j$ and hence $a_{\pi'(j)} = a_j' = a_0 \geq \ell_0 > \ell_j$ and $j$ is good after the addition. So the invariant is maintained since the number of good indices increases and $d$ decreases by at most one. Else if $j$ was good and 0 was not, $a_j \geq \ell_j$, and hence $A$ also incurs a down-step. Thus $d$ does not change and the number of good indices does not increase. In either case, the invariant is maintained.

Assume finally that both indices were good. Then $a_0 \geq \ell_0$ and $a_j \geq \ell_j$ before the addition of $e$. Then $j$ is good afterwards and $A$ incurs a down-step. So the number of good indices and $d$ do not change and the invariant is maintained.

**Case, $e \geq \ell_{k-1}$**: Let $i$ be minimal such that $e \geq \ell_i$. Then either $i = 0$ or $e < \ell_{i-1}$. Algorithm $\text{Alg}_{\text{DS}}$ adds $e$ to the $i$-th queue and algorithm $A$ adds $e$ to the $j$-th queue; $j = i$ is possible. Then $\ell_j' = e = a_j'$. Algorithm $\text{Alg}_{\text{DS}}$ does not incur a down-step and hence $d$ does not decrease. We change $\pi$ to let $i$ map to $j$ and $j$ map to $i$. If $i = j$, we do not modify $\pi$.

Then $i$ is a good index after the addition of $e$. So the invariant is maintained if at most one of the indices $i$ and $j$ were good before the addition of $e$. Note that this includes the case of $i = j$.

So assume $i \neq j$ and $i$ and $j$ are good before the addition of $e$. Then $a_i \geq \ell_i$ and $a_j \geq \ell_j$. If $i < j$, then $a_i \geq \ell_i > \ell_j$ and hence $j$ is good after the addition of $j$. Thus the invariant is maintained. If $j < i$, then $a_j \geq \ell_j \geq \ell_{i-1} > e$ and hence $A$ incurs a down-step. So the
number of good indices decreases by at most one and \( d \) increases by one and hence the invariant is maintained.

\textbf{Theorem 30.} No algorithm creates a smaller total number of down-steps over all queues than algorithm \( \text{Alg}_\text{DS} \). Algorithm \( \text{Alg}_\text{DS} \) generates an output with a minimal number of down-steps.

\textbf{Proof.} By the invariant

\[ d_A - d_O \geq k - \text{number of good indices} \geq 0, \]

and hence \( d_A \geq d_O \). This proves the first part. For the second part, we observe that the number of down-steps in different queues differs by at most one and hence the number of down-steps in the output sequence created by \( \text{Alg}_\text{DS} \) is at most \( \lceil d_O/k \rceil \). Algorithm \( A \) must create at least \( \lceil d_A/k \rceil \) down-steps in one of the queues and hence the output. \hfill \( \blacksquare \)

We will next show that \( \text{Alg}_\text{DS} \) is the unique optimal online algorithm. Recall that in an online algorithm the input is presented as a sequence \( e_1, e_2, \ldots, e_j, \ldots \) and we have to pick a queue for element \( e_j \) based on the current state of the queues and \( e_j \). The processing of \( e_j \) is independent of all the elements that follow it in the sequence.

\textbf{Theorem 31.} \( \text{Alg}_\text{DS} \) is the unique optimal online algorithm.

\textbf{Proof.} Consider any other online algorithm \( A \). Let \( e_1, e_2, \ldots, e_j, \ldots \) be the input sequence. When an element \( e_j \) is to be enqueued an online algorithm chooses the queue based on the value of \( e_j \) and the current contents of the queues, but independent of the continuation \( e_{j+1}, \ldots \). Assume now that \( A \) is not identical to \( \text{Alg}_\text{DS} \). Then there is an input sequence \( e_1, e_2, \ldots, e_j \) such that \( A \) and \( \text{Alg}_\text{DS} \) act the same up to element \( e_{j-1} \) but choose different queues for \( e_j \). We will show that there is a continuation \( e_{j+1}, \ldots \) that forces \( A \) to incur more down-steps than \( \text{Alg}_\text{DS} \).

Let us first assume that all queues are in use when \( e \) in enqueued. We may assume \( b = 0 \). So \( \ell_0 > \ell_1 > \ldots > \ell_{k-1} \) and these are also the tails of the queues for algorithm \( A \). An element \( e \) arrives.

Assume first that there is an \( i \) such that \( i = 0 \) and \( e > \ell_1 \) or \( \ell_{i-1} > e > \ell_i \). Algorithm \( \text{Alg}_\text{DS} \) adds \( e \) to the \( i \)-th queue and algorithm \( A \) adds \( e \) to the \( j \)-th queue with \( j \neq i \).

If \( j < i \), \( A \) incurs a down-step and we have an input sequence on which \( A \) does worse.

If \( j > i \), we have to work harder. The final elements of the queues are now:

\begin{align*}
\text{algorithm } \text{Alg}_\text{DS} & \quad \ell_0 > \ldots > \ell_{i-1} > e > \ell_{i+1} > \ldots > \ell_{j-1} > \ell_j > \ell_{j+1} > \ldots > \ell_{k-1} \\
\text{algorithm } A & \quad \ell_0 > \ldots > \ell_{i-1} > e > \ell_i > \ell_{i+1} > \ldots > \ell_{j-1} > \ell_{j+1} > \ldots > \ell_{k-1}
\end{align*}

We now insert \( k - 1 - i \) elements that algorithm \( \text{Alg}_\text{DS} \) can insert without incurring a down-step but \( A \) cannot. More precisely, we insert the decreasing sequence \( \ell_{i+1} + \epsilon, \ldots, \ell_j + \epsilon, \ldots, \ell_{k-1} + \epsilon \), where \( \epsilon \) is an infinitesimal. Algorithm \( \text{Alg}_\text{DS} \) can insert these elements into queues \( i + 1 \) to \( k \) without incurring a down-step. Algorithm \( A \) either has to put two of these elements in the same queue or one element in one of the first \( i + 1 \) queues. In either case, it incurs a down-step.

We come to the case \( e < \ell_k \). Algorithm \( \text{Alg}_\text{DS} \) puts \( e \) into queue 0, algorithm \( A \) puts \( e \) into queue \( j > 0 \). The final elements of the queues are now:

\begin{align*}
\text{algorithm } \text{Alg}_\text{DS} & \quad \ell_1 > \ell_2 > \ldots > \ell_{j-1} > \ell_j > \ell_{j+1} > \ldots > \ell_{k-1} > e \\
\text{algorithm } A & \quad \ell_0 > \ldots > \ell_{j-1} > \ell_{j+1} > \ldots > \ell_{k-1} > e
\end{align*}
We next insert \( \ell_j + \epsilon, \ell_{j+1} + \epsilon, \ldots, \ell_{k-1} + \epsilon \). Algorithm \( \text{Alg}_{\text{DS}} \) inserts these elements without incurring a down-step, but algorithm \( \mathcal{A} \) must incur a down-step. The argument is as above.

We have now handled the situation when \( \mathcal{A} \) differs from \( O \) after both algorithms fill all queues. We next deal with the situation where only \( h < k \) queues are used and we have \( \ell_0 > \ldots > \ell_{h-1} \). An element \( e \) is added. If \( e > \ell_{h-1} \), we argue as above. If \( e < \ell_{h-1} \) and \( \mathcal{A} \) does not open a new queue, it incurs a down-step. The argument is as above.

\[ \mathcal{A} \text{ does not open a new queue, it incurs an down-step. If } \mathcal{A} \text{ opens a new queue, it does the same as } \text{Alg}_{\text{DS}}. \]

### 3.3 Sequence Decomposition

In this section, we characterize the set of runs determined by algorithm \( \text{Alg}_{\text{DS}} \).

Let \( I_0 \) be the input sequence. We will decompose \( I_0 \) into runs \( s_0, s_1, \ldots \) and simultaneously construct modified input sequences \( I_1, I_2, \ldots \). The run \( s_i \) is a run in \( I_i \) and \( I_{i+1} \) is obtained from \( I_i \) by deleting the elements in \( s_i \).

Assume \( I_j \) is already defined. Let \( f_j \) be the leftmost element in \( I_j \) (if any) in which a decreasing sequence of length \( k + 1 \) ends. If there is no such element, let \( f_j \) be a fictitious element after the end of \( I_j \). Let \( s_j \) be the leftmost increasing subsequence of the prefix \( I_j[\text{up-to } f_j] \) of \( I_j \) ending just before \( f_j \), i.e., \( s_j \) starts with the first element of \( I_j \) and is then always extended by the first element that is larger than its last element.

\[ \textbf{Theorem 32.} \text{ Algorithm } \text{Alg}_{\text{DS}} \text{ constructs runs } s_i, s_{i+k}, s_{i+2k}, \ldots \text{ in queue } i, 0 \leq i < k. \text{ Exactly the non-fictitious } f_j \text{ create down-steps.} \]

\[ \textbf{Proof.} \text{ Let the rank of an element be the length of the longest decreasing subsequence ending in the element. Consider the execution of } \text{Alg}_{\text{DS}} \text{ on } I_j \text{ starting with empty queues and base queue } b. \text{ Just before } f_j \text{ is enqueued, the elements of rank } i, 0 \leq i < k, \text{ preceding } f_j \text{ in } I_j \text{ form the run in queue } q_{b+i}. \text{ In particular, queue } q_b \text{ contains exactly the elements in } s_j. \text{ When } f_j \text{ is enqueued into queue } b, b + 1 \text{ becomes the base queue and the contents of the queues } b + 1, \ldots b \text{ are now the elements of rank } 0, 1, \ldots k - 1 \text{ in } I_{j+1}[\text{up-to and including } f_j]. \]

### 3.4 Witnesses and Forced Down-steps

We associate witnesses with algorithm \( \text{Alg}_{\text{DS}} \) that explain why elements were put into particular queues. Assume \( e \) is the next element to be enqueued, \( b \) is the base queue, and \( \ell_b > \ell_{b+1} > \ldots > \ell_{b+k-1} \). Indices are to be read modulo \( k \) and queues are indexed 0 to \( k - 1 \).

Initially \( b = 0 \). We also set \( j \) to zero. \( b = j \mod k \) will be an invariant. When an element is added to a queue which is not the base queue, we let it point to the last element in the preceding queue. In this way, if an element is added to a queue \( b + j \), the pointer sequence has length (number of edges) at least \( j \). Pointer sequences end in elements that were inserted into the current base queue.

We also construct a sequence \( f_0, f_1, f_2, \ldots \). They are the first elements of runs in each of the queues except for the first run in each queue. So for any \( i \in [1, k - 1] \) and any \( j \geq 0 \), \( f_{j+b+k+1} \) is the first element of run \( j + 1 \) in queue \( i \); runs are numbered starting at zero.

Recall the enqueuing algorithm. Let \( e \) be the element to be enqueued. Let \( i \) be minimal (if any) such that \( e > \ell_{b+i} \).

\[ \begin{align*}
&\text{If } i \text{ exists, we append } e \text{ to } q_b+i \text{ and if } i > 0 \text{ we, in addition, let } e \text{ point to the last element of } q_{b+i-1}. \\
&\text{If } i \text{ does not exists, i.e., } e < \ell_{b+k-1} \text{, } e \text{ starts a new run in } q_b. \text{ We append } e \text{ to } q_b. \text{ This creates a down-step. We let } e \text{ point to the last element of } \ell_{b+k-1}. \text{ Also } e \text{ becomes } f_j \text{ and we increment } b \text{ and } j.
\end{align*} \]
Enqueuing $f_j$ makes queue $j + 1 \mod k$ the base queue.

Consider the first run in queue 1. The elements that were enqueued before $f_0$ have a pointer to queue 0, the elements that were enqueued after $f_0$ do not have a pointer to queue 0. Similarly for queue 2. The elements that were enqueued before $f_1$ have a pointer to queue 1, the elements that were enqueued after $f_1$ do not have a pointer to queue 1. Consider the second run in queue 2. The elements that were enqueued before $f_k$ have a pointer to queue 0, the elements that were enqueued after $f_k$ do not have a pointer to queue 0.

**Theorem 33.** Let $f_0$ to $f_{t-1}$ be the non-fictitious $f$’s in our input sequence. Any algorithm must have incurred $j + 1$ down-steps after it has enqueued $f_j$.

**Proof.** This is true for $j = -1$.

Consider $f_{t-1}$ and its backward chain in $I$. Since $f_{t-1}$ has rank $k + 1$ in $I_{t-1}$, the chain has length at least $k + 1$. It might be longer, say it has length $k + \ell$ for some $\ell \geq 1$ and contains elements $f_{t-1} = y_0, y_1, \ldots, y_{k+\ell-1}$ reading backwards. $f_{t-1}$ is in queue $t - 1 \mod k$, $y_1$ belongs to queue $t - 2 \mod k$, and $y_{k+\ell-1}$ belongs to queue $t - 1 - (k + \ell - 1) \mod k = t - \ell$. Reading the chain forward starting at $y_{k+\ell+1}$, we have a decreasing subsequence of $k + \ell$ elements.

Just before $y_0$ is enqueued, $y_1$ to $y_k$ belong the current runs of their respective queues, i.e., the runs we are building in the moment. The other elements belong to earlier runs. $y_k$ belongs to the same queue as $y_0$, i.e., queue $t - 1 \mod k$. When does $y_k$ have a back-pointer? $f_{t-2}$ belongs to queue $t - 2 \mod k$. $y_k$ has a back-pointer if it was enqueued before $f_{t-2}$ and does have a pointer if it was enqueued later.

We claim that the chain stop at $y_{k+\ell-1}$. This is because when $y_{k+\ell-1}$ was enqueued, it was enqueued into what is the base chain at that moment of time; chain $(t - 1) - k - \ell + 1 \mod k = t - \ell$. The chain became the base chain because of the enqueuing of $f_{t-1-\ell}$. We now have a decreasing subsequence of length $k + \ell$ starting after $f_{t-1-\ell}$. So any algorithm must incur $\ell$ down-steps after enqueuing $f_{t-1-\ell}$.

By induction hypothesis, it has incurred $t - 1 - \ell + 1$ down-steps up to and including the enqueuing of $f_{t+1-\ell}$. So the total number of down-steps is $t$.

4 Conclusion

We conclude with a few open problems:

- Our results in Theorem 3 depend on the fact that the $k$ queues are in parallel. It is an interesting open problem to consider the general setting where $k$ queues are arranged in an arbitrary directed acyclic network.
- Theorem 2 only applies to two sequences of LDS two each. Can we generalize this result to multiple queues and larger LDS?
- Besides the two measures of disorder, namely LDS and number of downsteps, it is interesting to develop other measures and understand their algorithmic properities.
- Stacks, which obey the Last-In-First-Out (LIFO) principle, have also been considered for sorting sequences [11,20]. It is an interesting open problem to characterize the sorting power of $k$ parallel stacks, for any fixed number $k$.

5 Acknowledgements

The research reported in this paper was carried out in the context of project MoDigPro (https://modigpro.saarland), which has been supported by the European Regional Develop-
ment Fund (ERDF). All authors were at the Max Planck Institute for Informatics when this research was performed.

References

1. David Aldous and Persi Diaconis. Longest increasing subsequences: from patience sorting to the baik-deift-johansson theorem. *Bulletin of the American Mathematical Society*, 36(4):413–432, 1999.
2. B. Aspvall, M. Plass, and R.E. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8:121–123, 1979.
3. Sven Boge and Sigrid Knust. The parallel stack loading problem minimizing the number of resuffles in the retrieval stage. *Eur. J. Oper. Res.*, 280(3):940–952, 2020.
4. Mahdi Boroujeni, Mohammad Ghodsi, and Saeed Seddighin. Improved mpc algorithms for edit distance and ulam distance. *IEEE Transactions on Parallel and Distributed Systems*, 32(11):2764–2776, 2021.
5. Nils Boysen and Simon Emde. The parallel stack loading problem to minimize blockages. *European Journal of Operational Research*, 249(2):618–627, 2016.
6. Badrish Chandramouli and Jonathan Goldstein. Patience is a virtue: Revisiting merge and sort on modern processors. New York, NY, USA, 2014. Association for Computing Machinery.
7. Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.
8. F-Y Ding and Hui Sun. Sequence alteration and restoration related to sequenced parts delivery on an automobile mixed-model assembly line with multiple departments. *International Journal of Production Research*, 42(8):1525–1543, 2004.
9. Shimon Even and Alon Itai. Queues, stacks and graphs. In *Theory of machines and computations*, pages 71–86. Elsevier, 1971.
10. C. Gagné, M. Gravel, S. Morin, and WL. Price. Impact of the pheromone trail on the performance of aco algorithms for solving the car-sequencing problem. *Journal of the Operational Research Society*, 59:1077–1090, 2008.
11. Martin Charles Golumbic. *Algorithmic graph theory and perfect graphs*. Elsevier, 2004.
12. P. Gopalan, TS. Jayram, R. Krauthgamer, and R. Kumar. Estimating the sortedness of a data stream. In *SODA*, 2007.
13. Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 1968.
14. Colin L Mallows. Patience sorting. *SIAM review*, 5(4):375, 1963.
15. Sebastian Meissner. Controlling just-in-sequence flow-production. *Logistics Research*, 2(1):45–53, 2010.
16. Sara Morin, Caroline Gagné, and Marc Gravel. Ant colony optimization with a specialized pheromone trail for the car-sequencing problem. *European Journal of Operational Research*, 197:1185–1191, 2009.
17. Vaughan R. Pratt. Computing permutations with double-ended queues, parallel stacks and parallel queues. page 268–277, 1973.
18. Craigie Schensted. Longest increasing and decreasing subsequences. *Canadian Journal of mathematics*, 13:179–191, 1961.
19. Sven Spieckermann, Kai Gutenschwager, and Stefan Voß. A sequential ordering problem in automotive paint shops. *International journal of production research*, 42(9):1865–1878, 2004.
20. Robert Tarjan. Sorting using networks of queues and stacks. *Journal of the ACM (JACM)*, 19(2):341–346, 1972.
21. Anna Katharina Twelsiek. Production planning of mixed-model assembly lines at the ford saarlouis plant. Master’s thesis, Universität des Saarlandes, 2021.