Fast and Longest Rollercoasters

Paweł Gawrychowski
Institute of Computer Science, University of Wrocław, Poland
gawry@cs.uni.wroc.pl

Florin Manea
Kiel University, Germany
flm@informatik.uni-kiel.de

Radosław Serafin
Institute of Computer Science, University of Wrocław, Poland
radserafin@gmail.com

───── **Abstract** ─────

For $k \geq 3$, a *k-rollercoaster* is a sequence of numbers whose every maximal contiguous subsequence, that is increasing or decreasing, has length at least k; 3-rollercoasters are called simply rollercoasters. Given a sequence of distinct real numbers, we are interested in computing its maximum-length (not necessarily contiguous) subsequence that is a k-rollercoaster. Biedl et al. (2018) have shown that each sequence of n distinct real numbers contains a rollercoaster of length at least $\lceil n/2 \rceil$ for $n > 7$, and that a longest rollercoaster contained in such a sequence can be computed in $O(n \log n)$-time (or faster, in $O(n \log \log n)$ time, when the input sequence is a permutation of $\{1, \ldots, n\}$). They have also shown that every sequence of $n \geqslant (k-1)^2 + 1$ distinct real numbers contains a k-rollercoaster of length at least $\frac{n}{2(k-1)} - \frac{3k}{2}$, and gave an $O(nk \log n)$-time (respectively, $O(nk \log \log n)$-time) algorithm computing a longest k-rollercoaster in a sequence of length n (respectively, a permutation of $\{1, \ldots, n\}$).

In this paper, we give an $O(nk^2)$-time algorithm computing the length of a longest k-rollercoaster contained in a sequence of n distinct real numbers; hence, for constant k, our algorithm computes the length of a longest k-rollercoaster in optimal linear time. The algorithm can be easily adapted to output the respective k-rollercoaster. In particular, this improves the results of Biedl et al. (2018), by showing that a longest rollercoaster can be computed in optimal linear time. We also present an algorithm computing the length of a longest k-rollercoaster in $O(n \log^2 n)$-time, that is, subquadratic even for large values of $k \leq n$. Again, the rollercoaster can be easily retrieved. Finally, we show an $\Omega(n \log k)$ lower bound for the number of comparisons in any comparison-based algorithm computing the length of a longest k-rollercoaster.

1 Introduction

The mathematical study of patterns occurring in sequences of numbers is a rather old and well developed topic in combinatorics and algorithms on sequences. Within this topic, of a particularly high interest is the study of long increasing and decreasing (not necessarily contiguous) subsequences occurring in a sequence. For example, already in 1749, Euler defined the Eulerian polynomials, which are the generating function for the number of descents in permutations. Almost 200 years later, Erdős and Szekeres [9] proved the existence of an increasing or a decreasing subsequence of length at least $a + 1$ in a sequence of at least $n = a^2 + 1$ distinct reals. More precisely, they have shown the following theorem.

▶ **Theorem 1** (Erdős and Szekeres, 1935). *Every sequence of $ab + 1$ distinct real numbers contains an increasing subsequence of length at least $a + 1$ or a decreasing subsequence of length at least $b + 1$.*

The theorem of Erdős–Szekeres is strongly related to, and in fact also follows from, the well-known decomposition of Dilworth (see [18]) regarding chains and antichains in a finite partially ordered set. Dilworth's result can be restated in the context of the combinatorics of patterns in sequences of numbers as follows.

▶ **Theorem 2** (Dilworth, 1950). *Any finite sequence S of distinct real numbers can be partitioned into k ascending sequences, where k is the maximum length of a descending sequence in S.*

Recent surveys on the combinatorics of patterns occurring in sequences are [14, 15].

The study of patterns in sequences of numbers also has a well developed algorithmic side (see, e.g., [4, 8, 10, 13]). For instance, finding a longest increasing subsequence (not necessarily contiguous) contained in the input sequence is a basic problem in theoretical computer science, studied already from the 1960s [3, 16, 17], with applications in areas such as bioinfomatics and physics (see [19] and the references therein). In particular, in 1975 Fredman [10] presented an algorithm (which he attributed to Knuth, now considered folklore) computing the length of a longest increasing subsequence (LIS) in an array of n numbers in $O(n \log n)$ time, and proved that this is optimal for comparison-based algorithms. If required, the algorithm can be extended to retrieve such a subsequence. If the input sequence can be sorted in linear time (in particular, when the input sequence is a permutation of $\{1, \dots, n\}$) and we do not require the algorithm to be comparison-based, the solution given by Fredman can be implemented in $O(n \log \log n)$ time, see [8] and the references therein. Fredman's algorithm is often called PATIENCE SORTING, and has some connections to constructing the so-called Young Tableaux [3, 16].

We consider a notion that is strongly related to longest increasing subsequences (and longest decreasing subsequences). A *run* in a sequence of numbers is a maximal contiguous subsequence that is either increasing or decreasing. A *k-rollercoaster*, where $k \geq 3$, is a sequence of numbers whose every run has length at least k; 3-rollercoasters are called, for short, rollercoasters. For example, the sequence $(3, 6, 8, 10, 9, 5, 1, 2, 4, 7, 11)$ is a 4-rollercoaster with runs $(3, 6, 8, 10)$, $(10, 9, 5, 1)$, $(1, 2, 4, 7, 11)$. Given a sequence $S[1 : n] = (S[1], S[2], \dots, S[n])$ of n distinct numbers, the k-rollercoaster problem is to find a maximum-size set of indices $i_1 < i_2 < \dots < i_m$ such that $(S[i_1], S[i_2], \dots, S[i_m])$ is a k-rollercoaster. In other words, this problem asks for a longest k-rollercoaster contained in the input sequence S.

There is a simple, but useful, geometrical interpretation of k-rollercoasters. The input sequence $S[1 : n]$ can be depicted as a set P of points in the plane by translating, for i from 1 to n, the number $S[i]$ to a point $p_i = (i, S[i])$. In this setting, a k-rollercoaster in S translates to a polygonal path in the plane, whose vertices are points of P, and such that every maximal sub-path, with positive- or negative-sloped edges, has at least k points. The rollercoaster $(3, 6, 8, 10, 9, 5, 1, 2, 4, 7, 11)$ is depicted in the left half of Figure 1. Two 4-rollercoasters occurring in the sequence $(3, 6, 1, 8, 7, 17, 13, 10, 11, 12, 9, 5, 14, 4, 2, 15, 16)$ are depicted in the right half of the same figure.

While rollercoasters seem interesting on their own as a combinatorial structure, the original motivation for their study was a connection to computational geometry and graph drawing, namely to point-set embeddings of *caterpillars* (see [5,6] and the references therein). More precisely, constructing a long rollercoaster in a sequence of numbers was used as an
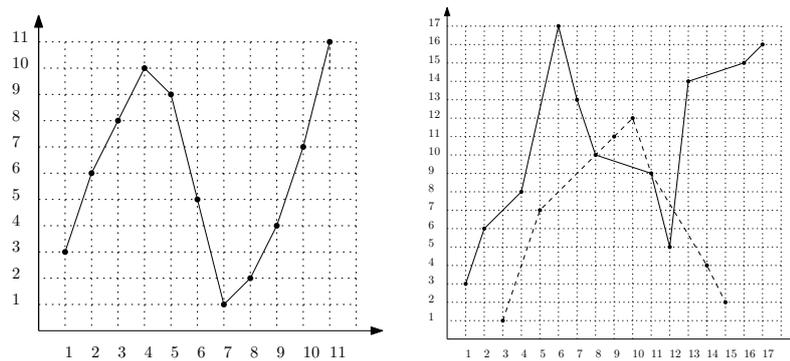
Figure 1 Left: a 4-rollercoaster $(3, 6, 8, 10, 9, 5, 1, 2, 4, 7, 11)$ with runs $(3, 6, 8, 10)$, $(10, 9, 5, 1)$, $(1, 2, 4, 7, 11)$. Right: two 4-rollercoasters, represented with a solid and, respectively, a dashed line, in $(3, 6, 1, 8, 7, 17, 13, 10, 11, 12, 9, 5, 14, 4, 2, 15, 16)$.

intermediate step towards obtaining a method of drawing a n-vertex top-view caterpillar, with L-shaped edges, on a set of $\frac{25}{3}n$ general orthogonal position points in the plane. This is currently the best known bound on the number of points required to draw such a graph.

In [5], the following results regarding k-rollercoasters were shown. First, from a combinatorial point of view, for $k = 3$, it was shown that the length of a longest rollercoaster contained in a sequence of $n \geq 7$ distinct numbers is at least $\lceil \frac{n}{2} \rceil$. As far as k-rollercoasters are concerned, it was shown that for $k \geqslant 4$ every sequence of $n \geqslant (k - 1)^2 + 1$ distinct numbers contains a k-rollercoaster of length at least $\frac{n}{2(k-1)} - \frac{3k}{2}$. From an algorithmic point of view, both previously mentioned results were constructive, leading to an $O(n)$-time (respectively $O(n \log k)$) algorithm computing a long (but not necessarily a longest) rollercoaster (respectively, k-rollercoaster) contained in a sequence of n distinct numbers. A longest rollercoaster contained in such a sequence was computed by an extension of Fredman's algorithm in $O(n \log n)$-time, and if the input sequence is a permutation of $\{1, \ldots, n\}$ (or, more generally, sortable in linear time) in $O(n \log \log n)$ time. By further generalising this approach, an $O(nk \log n)$-time (respectively, $O(nk \log \log n)$-time) algorithm computing a longest k-rollercoaster in a sequence of n distinct numbers (respectively, a permutation of $\{1, \ldots, n\}$) can be obtained. Note that, by the theorem of Erdös and Szekeres, a sequence of n distinct numbers always contains a $\lfloor \sqrt{n} \rfloor$-rollercoaster, and the aforementioned algorithm computes a longest such rollercoaster in $O(n^{1.5} \log n)$ time.

Our contributions. We consider the problem of computing a longest k-rollercoaster in an input sequence $S[1 : n]$ and provide three results.

Firstly, we design a comparison-based algorithm computing the length of a longest k-rollercoaster in a sequence of n distinct numbers in $O(nk^2)$ time. Thus, we obtain an optimal linear-time algorithm for constant values of k, in particular for $k = 3$. This significantly improves the results of [5] and shows that, even though longest rollercoasters are related to longest increasing subsequences, the rich combinatorial structure of the former makes them provably easier to find. The starting point of our algorithm is the following natural dynamic programming formulation. For each $2 \leq i \leq k$, and for each element $S[j]$, we compute a longest (not necessarily contiguous) subsequence of S ending with $S[j]$ and with every run of length at least k, except for the last run, which has only i elements if $i < k$ and at least k elements if $i = k$. Now the difficulty is to find the predecessor $S[j']$ of $S[j]$ in such a subsequence in time proportional to k, in particular avoiding any kind of binary search.

We greedily decompose the input sequence into blocks with a certain property related to Dilworth's theorem and prove, by a careful case analysis, that j' must belong to the previous few such blocks. This, together with the special structure of the blocks and appropriate data structures, allows us to find j' in $O(k)$ amortised time.

Secondly, we focus on the case of large k. Given that both the previous and the new algorithm have at least linear dependency on k, it might seem plausible that this is inherent to the problem, for example that for $k \geq \lfloor \sqrt{n} \rfloor$ the running time of any algorithm needs to be $\Omega(n^{1.5})$. We show that this is not the case by designing a subquadratic algorithm that computes a longest k-rollercoaster in a sequence of n distinct numbers in $O(n \log^2 n)$ time. To obtain this result, we exploit the fact that if an increasing (respectively, decreasing) run in a longest k-rollercoaster extends from $S[i]$ to $S[j]$, then that run should be LIS (respectively, longest decreasing sequence, LDS for short) in $S[i : j]$. If one arranges the length of LIS (respectively, LDS) in $S[i : j]$ in an $n \times n$ matrix then the matrix has the anti-Monge property. It is known that all row maxima of an anti-Monge matrix can be found in $O(n)$ time [2], that is, in sublinear time w.r.t. the size of the matrix (given an oracle access to the elements of the matrix). Such properties have been successfully exploited to speed up certain dynamic programming algorithms. We also follow this route, and construct a longest k-rollercoaster using dynamic programming, essentially by gluing together LISs and LDSs of consecutive contiguous subsequences of S.

Thirdly, we show that any comparison-based algorithm computing a longest k-rollercoaster needs $\Omega(n \log k)$ comparisons. Our reasoning is similar to the one used by Fredman to show that any comparison-based algorithm computing a LIS needs $\Omega(n \log n)$ comparisons. We leave as an open problem to close the gap between the lower and upper bounds shown here.

The paper is organised as follows. After a series of preliminaries, we describe the $O(nk^2)$-time algorithm for computing the length of a longest k-rollercoaster, followed by the $O(n \log^2 n)$-time algorithm. We conclude with the lower bound for the number of comparisons needed to compute the length of a longest k-rollercoaster in a sequence of length n. The proofs omitted here for space reasons can be found in [12].

2   Preliminaries

We consider sequences of distinct real numbers and work in the comparison-based model. If S is a sequence of n numbers, then $|S| = n$ is the length of the sequence, and $S[i]$ denotes its ith element. A *subsequence* of S is a sequence $(S[i_1], S[i_2], \ldots, S[i_m])$, defined by specifying the indices $1 \leq i_1 < i_2 < \ldots < i_m \leq n$. For $1 \leq i \leq j \leq n$, $S[i : j]$ denotes the *contiguous subsequence* $(S[i], S[i + 1], \ldots, S[j])$; in particular, $S[1 : n]$ denotes the entire S. Note that unless explicitly stated, a subsequence is not necessarily contiguous. An increasing subsequence (respectively, decreasing subsequence) of S is a subsequence $(S[i_1], S[i_2], \ldots, S[i_m])$ such that $S[i_j] < S[i_{j+1}]$, for all $1 \leq j \leq m - 1$ (respectively, $S[i_j] > S[i_{j+1}]$, for all $1 \leq j \leq m - 1$). A longest increasing (respectively, decreasing) sequence, for short LIS (respectively, LDS), is an increasing (respectively, decreasing) sequence with the largest possible length. Fredman gave an $O(n \log n)$-time algorithm for computing the length of LIS, denoted res in Algorithm 1. A byproduct of this algorithm is a partition of $S[1 : n]$ into res non-increasing subsequences that can be obtained by creating, for every $1 \leq j \leq$ res, a list of elements that has been stored in $R[j]$.

A *run* in a sequence of numbers is a maximal contiguous subsequence that is increasing or decreasing. A *k-rollercoaster* is a sequence of numbers such that every run has length at least k; 3-rollercoasters are called, for short, rollercoasters. Given a sequence $S[1 : n]$

Algorithm 1 Finding the length of LIS of S.

1: $R[0] \leftarrow 0$
2: res $\leftarrow 0$
3: **for** $i \leftarrow 1$ **to** n **do**
4:      $k \leftarrow \max\{j : R[j] < S[i]\}$ $\triangleright$ binary search over $R[0] < R[1] < R[2] < \ldots$
5:      $R[k+1] \leftarrow S[i]$
6:      res $\leftarrow \max\{\text{res}, k+1\}$
7: **return** res

---

we are interested in finding its longest subsequence that is a k-rollercoaster. To make the exposition easier to follow, we focus on finding the length of such a subsequence. Recovering the subsequence itself is, in all our algorithms, rather straightforward.

3   Computing a Longest k-Rollercoaster in $O(nk^2)$-Time

In this section we show how to find a longest k-rollercoaster of $S[1:n]$ in $O(nk^2)$ time.

    We begin our algorithm with a preprocessing phase. An alternating k-decomposition of $S[1:n]$ is a partition of $S[1:n]$ into contiguous subsequences (called parts) $S_1, S_2, \ldots, S_m$ such that the length of LIS in the odd parts (S_1, S_3, S_5, and so on) is k while the length of LDS in the even parts is k, possibly smaller for the very last part, and additionally by removing the last element of any odd (even) part we obtain a sequence with LIS (LDS) of length less than k. In other words, for $\ell \geq 1$, S_ℓ is either the shortest contiguous subsequence of S that follows directly after $S_1 \cdots S_{\ell-1}$ and has for ℓ odd (even) a LIS (respectively, LDS) of length k, if such a subsequence exists, or the whole remaining part of S otherwise. For example, an alternating 3-decomposition of $S = (1, 4, 2, 5, 8, 7, 6, 3)$ is $(1, 4, 2, 5), (8, 7, 6), (3)$.

▶ **Lemma 3.** *An alternating k-decomposition of $S[1:n]$ can be found in $O(n \log k)$ time.*

Proof. By terminating Algorithm 1 as soon as res $= k$ we can find the shortest prefix of S with LIS equal to k in $O(d \log k)$ time, where d is the length of the prefix. Then we find the shortest prefix of the remaining suffix of S with LDS equal to k, and repeat. Overall, this takes $O(n \log k)$ time because all parts are disjoint. ◀

▶ **Proposition 4.** *Let A be a k-rollercoaster in S. Any part S_ℓ contains elements of at most four consecutive runs of A.*

Proof. By contradiction. Let S'_ℓ be S_ℓ without the last element. If S_ℓ contains elements of five consecutive runs of A then S'_ℓ contains elements of four consecutive runs of A, and hence all elements of two such consecutive runs. Thus, if S_ℓ is an odd (even) part then S'_ℓ contains LIS (LDS) of length k, which contradicts the definition of an alternating k-decomposition. ◀

    By Dilworth's theorem, a part with LIS of length k can be decomposed into k decreasing subsequences, and such a decomposition can be obtained as a byproduct of Algorithm 1. Thus, we can decompose each part into up to k monotone (increasing or decreasing, depending on whether the part is odd or even) subsequences. These subsequences can be then merged to obtain a sorted list P_ℓ of all elements in the corresponding part S_ℓ in $O(n \log k)$ overall time, for example by first merging pairs of subsequences, then quadruples, and so on.

    Before moving on to the description of our algorithm, we need a combinatorial lemma that relates an alternating k-decomposition to a longest rollercoaster.

▶ **Lemma 5.** *Suppose that $x = S[j]$ is a non-first element occurring in an increasing run of a longest k-rollercoaster, and $y = S[j']$ is its predecessor in the same run, and consider an alternating k-decomposition of $S[1 : n]$. Then either x and y are in the same part S_i, or y is in one of the parts $S_{i-4}, S_{i-3}, S_{i-2}, S_{i-1}$.*

Proof. By contradiction. Suppose that there are at least four parts between x and y, i.e., x is in S_i and y is in some S_k with $k < i - 4$. Let r denote the run in the k-rollercoaster that contains x and y, let d be the length of r, and let ℓ be such that $r[\ell] = y$ and $r[\ell + 1] = x$. We assume that r is an increasing run (see Figure 2); the case when r is decreasing can be treated in the same way.
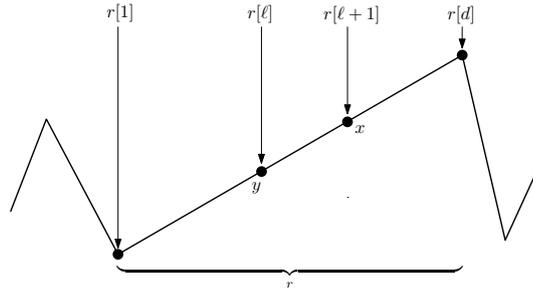
■ **Figure 2** The increasing run r from Lemma 5, with the points x and y highlighted.

Consider the following four cases:
1. $\ell \leq k - 1$ (i.e., there are at most $k - 2$ elements in r before y) and $k - 2 \geq d - \ell - 1$ (there are at most $k - 2$ elements in r after x).
2. $\ell \leq k - 1$ and $k - 1 \leq d - \ell - 1$ (there at least $k - 1$ elements in r after x).
3. $\ell \geq k$ (there are at least $k - 1$ elements in r before y) and $k - 2 \geq d - \ell - 1$.
4. $\ell \geq k$ and $k - 1 \leq d - \ell - 1$.

Recall that there are at least four whole parts between x and y. Therefore, in particular there are three consecutive parts $S_{i'}, S_{i'+1}$, and $S_{i'+2}$ such that the first has LIS of length k, the second has LDS of length k, and the third has LIS of length k.

In the first case, we replace $r[2 : d - 1]$ with LIS of $S_{i'}$, the LDS of $S_{i'+1}$, and LIS of $S_{i'+2}$. It is straightforward to verify that we obtain a valid k-rollercoaster, and because we remove at most $2k - 4$ elements and add at least $3k$, this creates a longer k-rollercoaster, which is a contradiction. In the second case, we replace $r[2 : \ell]$ with LIS of $S_{i'}$ and LDS of $S_{i'+1}$. Again, it is straightforward to verify that we obtain a valid longer k-rollercoaster, because we remove at most $k - 2$ elements and add at least $2k$. Similarly, in the third case, we replace $r[\ell + 1 : d - 1]$ with LDS of $S_{i'+1}$ and LIS of $S_{i'+2}$ to obtain a longer k-rollercoaster. Finally, in the fourth case we simply insert LDS of $S_{i'+1}$ between x and y to obtain a longer k-rollercoaster. ◀

After the initial preprocessing phase we apply dynamic programming. For $1 \leq i \leq k$, we say that a subsequence of S (not necessarily contiguous) is a $(k, i)_+$-rollercoaster if it ends with an increasing run of length exactly i when $i < k$ and at least k when $i = k$, while every other run is of length at least k. Additionally, we consider k-rollercoaster ending with a decreasing run as $(k, 1)_+$-rollercoaster. We want to construct, for every $1 \leq i \leq k$ and $1 \leq j \leq n$, a longest $(k, i)_+$-rollercoaster ending with $S[j]$. To this end we calculate $M_+[j, i]$, the position in S of the predecessor of $S[j]$ in such a $(k, i)_+$-rollercoaster, and $L_+[j, i]$, the length of the respective $(k, i)_+$-rollercoaster. A $(k, i)_-$-rollercoaster is defined similarly, except

that the last run should be decreasing, and we also calculate the values $M_-[j, i]$ and $L_-[j, i]$, defined similarly to the above and corresponding to such a $(k, i)_-$-rollercoaster. We only describe in detail how to compute $M_+[j, i]$ and $L_+[j, i]$, as $M_-[j, i]$ and $L_-[j, i]$ are computed analogously. The computation proceeds from left to right, that is, we iterate over the parts $S_1, S_2, \ldots$ and compute, for every element $S[j]$ of the current part S_ℓ, the values of $M_+[j, i]$ and $L_+[j, i]$ for every $1 \leq i \leq k$. See Algorithm 2 for a high-level overview of the algorithm.

Algorithm 2 Computing the length of a longest k-rollercoaster.

1: Find an alternating k-decomposition $S_1, \ldots, S_m$ of S.
2: **for** $1 \leq \ell \leq m$ **do**
3:     Merge the k monotone subsequences constituting S_ℓ to obtain a single sorted list P_ℓ.
4: **for** $1 \leq \ell \leq m$ **do**
    ▷ For each $S[j]$ in S_ℓ and $1 \leq i \leq k$, we compute the following:
        $M_+[j, i]$: position in S of the predecessor of $S[j]$ in its $(k, i)_+$-rollercoaster
        $L_+[j, i]$: length of the respective $(k, i)_+$-rollercoaster
5:     **for** $2 \leq i \leq k$ **do**
6:         **for** $1 \leq d \leq 4$ and each $S[j] \in S_\ell$ in the order of their occurrences in P_ℓ **do**
7:             Find $M_+^d[j, i]$ and $L_+^d[j, i]$.
8:         **for** each $S[j] \in S_\ell$ **do**
9:             $L_+[j, i] \leftarrow \max\{L_+^d[j, i] : 1 \leq d \leq 4\}$
10:             Set $M_+[j, i]$ so that it corresponds to $L_+[j, i]$.
11:     Compute, for each $S[j] \in S_\ell$, $L_-[j, i]$ and $M_-[j, i]$ with a similar approach.
12:     **repeat** 4 times
13:         **for** each $S[j] \in S_\ell$ **do**
14:             $L_+[j, 1] \leftarrow \max\{L_-[j, k], 1\}$
15:             $M_+[j, 1] \leftarrow M_-[j, k]$ if $L_-[j, k] > 0$ and 0 otherwise
16:         **for** $2 \leq i \leq k$ **do**
17:             **for** each $S[j] \in S_\ell$ in the order of their occurrences in P_ℓ **do**
18:                 Find $M_+'[j, i], L_+'[j, i]$ using decomposition of S_ℓ into k monotone sequences.
19:             **for** each $S[j] \in S_\ell$ **do**
20:                 $L_+[j, i] \leftarrow \max\{L_+[j, i], L_+'[j, i]\}$
21:                 Update $M_+[j, i]$ so that it corresponds to $L_+[j, i]$.
22:         Update, for each $S[j] \in S_\ell$, $L_-[j, i]$ and $M_-[j, i]$ with a similar approach.
23: **return** $\max\{\max\{L_-[j, k], L_+[j, k]\} : 1 \leq j \leq n\}$

When we begin computing the arrays $M_+[\cdot, i]$, $L_+[\cdot, i]$, $M_-[\cdot, i]$ and $L_-[\cdot, i]$, for $2 \leq i \leq k$, corresponding to all $S[j] \in S_\ell$, we have already computed $M_+[j', 1], M_+[j', 2], \ldots,$ $M_+[j', k]$ and $L_+[j', 1], L_+[j', 2], \ldots, L_+[j', k]$, as well as $M_-[j', 1], M_-[j', 2], \ldots, M_-[j', k]$ and $L_-[j', 1], L_-[j', 2], \ldots, L_-[j', k]$, for every $S[j'] \in S_{\ell'}$ such that $\ell' < \ell$.

We start with computing the values $M_+[\cdot, i]$, $L_+[\cdot, i]$, $M_-[\cdot, i]$ and $L_-[\cdot, i]$, for $2 \leq i \leq k$, assuming that the predecessor $S[j']$ of $S[j]$ in its corresponding rollercoaster belongs to $S_{\ell-d}$, for some $1 \leq d \leq 4$. In such case the longest rollercoaster ending at $S[j']$ has been already correctly determined and the computation is quite straightforward. If $S[j']$ also belongs to S_ℓ, we must be more careful to guarantee that the longest rollercoaster ending at $S[j']$ is already known. We proceed in iterations. In the tth iteration, we guarantee to compute the values such that at most t runs of the corresponding rollercoaster contain elements from S_ℓ. By Proposition 4, four iterations are enough. In a single iteration, we start with computing

the initial values $M_+[\cdot, 1]$, $L_+[\cdot, 1]$, $M_-[\cdot, 1]$ and $L_-[\cdot, 1]$ corresponding to $S[j]$ being the first element of its run. These values can be simply copied from the already known $M_-[\cdot, k]$, $L_-[\cdot, k]$, $M_+[\cdot, k]$ and $L_+[\cdot, k]$ corresponding to $S[j]$ being the last element of a rollercoaster with less than t runs containing elements from S_ℓ (or set to 1 corresponding to $S[j]$ being the only element in the rollercoaster). This is correct because a $(k, 1)_+$-rollercoaster is actually either a $(k, k)_-$-rollercoaster or a sequence consisting of a single element. Then, we calculate the values $M_+[\cdot, i]$, $L_+[\cdot, i]$, $M_-[\cdot, i]$ and $L_-[\cdot, i]$, for $2 \le i < k$, such that the predecessor $S[j'] \in S_\ell$ belongs to the same run as $S[j]$. By performing the calculation for $i = 2, 3 \ldots, k-1$ in this order we guarantee that the longest rollercoaster ending at the predecessor $S[j'] \in S_\ell$ is already known for all $S[j] \in S_\ell$, but the computation is still not completely trivial and requires a different approach depending on whether S_ℓ was decomposed into at most k increasing, respectively decreasing, subsequences. Finally, we extend this to $i = k$.

$M_+[j, i]$ belongs to $S_{\ell-d}$ for some $1 \le d \le 4$.   We process $S_{\ell-d}$ to identify some candidates, denoted $M_+^d[j, i]$ and $L_+^d[j, i]$, for $M_+[j, i]$ and $L_+[j, i]$, respectively, for every $S[j] \in S_\ell$. The idea is to compute these candidates in the order in which the elements $S[j]$ occur on the sorted list P_ℓ. So, let us consider P_ℓ and $P_{\ell-d}$. For each element $S[j]$ in the current part we want to identify a longest $(k, i-1)_+$-rollercoaster ending in $S_{\ell-d}$ with an element less than $S[j]$. Thus, as $P_{\ell-d}$ is increasing, for every element of the current part we need to consider all elements in a prefix of $P_{\ell-d}$. Also, if $S[j']$ is to the right of $S[j]$ in P_ℓ, that is, $S[j'] \ge S[j]$, then the prefix of $P_{\ell-d}$ that we need to consider to compute $M_+^d[j', i]$ is at least as long as the prefix that we need to consider to compute $M_+^d[j, i]$. Therefore, we can use two pointers to sweep through P_ℓ and $P_{\ell-d}$ from left to right, and obtain the information needed to compute $M_+^d[j, i]$ and $L_+^d[j, i]$, for every $S[j] \in S_\ell$. At the beginning the pointers point to the first element of P_ℓ and $P_{\ell-d}$, respectively. Say that the current element in P_ℓ and $P_{\ell-d}$ is $S[j]$ and $S[h]$, respectively (we update indices j and h along with the pointers). We keep moving forward the pointer corresponding to $S[h]$ until we find an element $S[h] > S[j]$. Then we set $M_+^d[j, i] = h'$ and $L_+^d[j, i] = L_+[h', i] + 1$, where $S[h']$ is an element occurring earlier than $S[h]$ in $P_{\ell-d}$ with the largest value of $L_+[h', i-1]$. The element $S[h']$ is maintained as we move from left to right in $P_{\ell-d}$. Then we proceed to the next element in P_ℓ. Overall, computing candidates $M_+^d[j, i]$ and $L_+^d[j, i]$, for every $S[j] \in S_\ell$, takes $O(|S_{\ell-d}| + |S_\ell|)$ time.

$M_+[j, i]$ belongs to S_ℓ decomposed into k increasing subsequences.   Recall that we have already computed $M_+[j', i']$ and $L_+[j', i']$ for every $i' < i$ and $S[j'] \in S_\ell$, and the goal is to identify candidates, denoted $M'_+[j, i]$ and $L'_+[j, i]$, for $M_+[j, i]$ and $L_+[j, i]$, respectively, for every $S[j] \in S_\ell$. Consider the decomposition of S_ℓ into k increasing subsequences $I_1, I_2, \ldots, I_k$. The elements of every sequence are increasing w.r.t. their value and w.r.t their position in S. Consider an element $S[j] \in I_a$ and $1 \le b \le k$ (possibly $a = b$). The elements of I_b that can be the predecessor of $S[j]$ in a $(k, i)_+$-rollercoaster (that is, possible candidates for $M_+[j, i]$) are both less w.r.t. value and w.r.t. position in S. Thus, these elements form a prefix of I_b, and for every $S[j] \in I_a$ and $1 \le b \le k$ we want to maximise $L_+[h', i]$ over all $S[h']$ in such a prefix. As in the previous case, we can use two pointers to sweep through I_a and I_b and compute, for every $S[j] \in I_a$, the element $S[h'] \in I_b$ that could precede $S[j]$ in a $(k, i)_+$-rollercoaster with the largest value of $L_+[h', i-1]$. Finally, we set $M'_+[j, i]$ and $L'_+[j, i]$ to correspond to the largest such value among all $1 \le b \le k$. Overall, computing the candidates $M'_+[j, i]$ and $L'_+[j, i]$, for every $S[j] \in S_\ell$, takes $O(k|S_\ell|)$ time.

$M_+[j,i]$ **belongs to** S_ℓ **decomposed into** k **decreasing subsequences.** This is the most complicated case. Recall that the decomposition into k decreasing subsequences $D_1, D_2, \ldots, D_k$ was obtained with Algorithm 1. In more detail, D_a consists of elements assigned to $R[a]$ throughout the execution of the algorithm. Thus, if $S[j] \in D_a$ then the predecessor of $S[j]$ in a sought longest $(k,i)_+$-rollercoaster, denoted $S[j']$, must belong to D_b for some $1 \leq b < a$. Indeed, Algorithm 1 first processes $S[j']$ and then $S[j]$, so if $S[j'] \in D_b$ then $R[b] \leq S[j']$ when processing $S[j]$ and consequently $S[j'] < S[j]$ implies that $S[j]$ is assigned to $R[a]$ with $a > b$. So, we first compute the candidates $M'_+[j,i]$ and $L'_+[j,i]$ for every $S[j] \in D_1$, then for every $S[j] \in D_2$, and so on. That is, consider a decreasing subsequence D_a and suppose that we have already computed the desired result for all elements in $D_1, D_2, \ldots, D_{a-1}$. Note that at this point we have already computed, for every $S[j] \in D_1 \cup \ldots \cup D_{a-1}$, the values of $M^d_+[j,i]$ and $L^d_+[j,i]$, for $1 \leq d \leq 4$, as well as the values $M'_+[j,i]$ and $L'_+[j,i]$ corresponding to the current iteration. Thus, we are already able to set $M_+[j,i]$ and $L_+[j,i]$ by choosing the option that maximises the length of the corresponding $(k,i)_+$-rollercoaster, which is important when extending this case to $i = k$.

Consider an element $S[j] \in D_a$ and $1 \leq b < a$. The elements of D_b that can be the predecessor of $S[j]$ in a $(k,i)_+$-rollercoaster (that is, possible candidates for $M_+[j,i]$) are both less w.r.t. value and w.r.t. position in S, similarly as in the previous case. The difference is that now these elements form contiguous subsequence X of D_b that is not necessarily a prefix. The first element of X can be found by searching for the first element with sufficiently small value, while its last element can be found by searching the last element with sufficiently small position (note that X might be empty). Let $S[j']$ be the next element after $S[j]$ in D_a, and Y be its corresponding contiguous subsequence of D_b consisting of possible predecessors in a $(k,i)_+$-rollercoaster. Clearly, $S[j] > S[j']$ while $j < j'$. Thus, the first element of Y is either the same as the first element of X or occurs after the first element of X in D_a, while the last element of Y is either the same as the last element of X or occurs after the last element of X in D_a (assuming that both X and Y are non-empty). Thus, we sweep through D_a while maintaining the current contiguous subsequence X of D_b corresponding to the possible predecessors of the current $S[j] \in D_a$. This requires the following tool.

▶ **Lemma 6** ([11]). *There is a data structure that maintains a list of elements under the following operations: pop an element from the front, push an element in the back, and return the maximum element in the current list, each in $O(1)$ time.*

When processing the current element $S[j] \in D_a$ we maintain the first element $S[f] \in D_b$ such that $S[f] < S[j]$ and the last element $S[\ell] \in D_b$ such that $\ell < j$. Then X consists of all elements between $S[f]$ and $S[\ell]$ in D_b (inclusive), and is maintained in a structure from Lemma 6 storing the lengths of their corresponding $(k,i)_+$-rollercoaster, that is, the already known value of $L_+[\cdot, i-1]$. This allows us to extract the element $S[j'] \in X$ with the largest value of $L_+[j', i-1]$, and set $M'_+[j,i] = j'$ and $L'_+[j,i] = L_+[j', i-1] + 1$ in constant time, while updating f and ℓ takes amortised constant time. Overall, computing the candidates $M'_+[j,i]$ and $L'_+[j,i]$, for every $S[j] \in S_\ell$, takes $O(k|S_\ell|)$ time.

Case $i = k$. To compute $M_+[j,k]$ and $L_+[j,k]$, we first use exactly the same approach as before for $i = k$, so consider the values of $M_+[\cdot, k-1]$ and $L_+[\cdot, k-1]$. But this only allows us to compute the length of a longest $(k,k)_+$-rollercoaster with the last run of length exactly k. To extend this to arbitrary $(k,k)_+$-rollercoasters with the last run of length greater than k we additionally run the same algorithm but instead of looking at $M_+[\cdot, k-1]$ and $L_+[\cdot, k-1]$ we use $M_+[\cdot, k]$ and $L_+[\cdot, k]$, including the values already computed in this extra step in the

third case. The reason why this works is that, due to the order in which we consider the elements of S_ℓ, at the moment when we compute the length of a longest $(k,k)_+$-rollercoaster ending with $S[j]$, and which may have more than k elements in the final run, we have already computed the length of a longest $(k,k)_+$-rollercoaster ending with any element $S[j']$ which may be a predecessor of $S[j]$ on the respective $(k,k)_+$-rollercoaster.

Conclusion. With these final remarks, our algorithm is completely described. It only remains to find the element $S[j]$ for which $\max\{L_-[j,k], L_+[j,k]\}$ is maximum. The correctness follows from the comments made throughout its description. To compute the complexity, it is enough to note that each part S_ℓ of the partition of S is processed in $O(k|S_\ell|)$ time, for each $2 \leq i \leq k$. Adding this up, we get that the total complexity of our algorithm is $O(nk^2)$.

▶ **Theorem 7.** *For every sequence $S[1:n]$ and $k \geq 3$, the length of a longest k-rollercoaster in S can be found in $O(nk^2)$-time.*

4 Computing a Longest k-Rollercoaster in $O(n \log^2 n)$-Time

Before we describe our algorithm, we introduce two preliminary procedures. Firstly, we introduce the definition of an anti-Monge matrix and the algorithm for finding the maximum in every column of such a matrix. Secondly, we describe the algorithm for finding LIS in contiguous subsequences of the input sequence. Finally, we describe the algorithm computing a longest k-rollercoaster in this sequence, using the previously developed tools as black boxes.

Monge matrices. Let A be an $n \times n$ matrix, and $A[i,j]$ denote its element in the ith row from the top and the jth column from the left. A is *Monge* (respectively, *anti-Monge*) if, for every $1 \leq i < j \leq n$ and $1 \leq k < \ell \leq n$, the *Monge equality* holds, namely $A[i,k] + A[j,\ell] \leq A[i,\ell] + A[j,k]$ (respectively, $A[i,k] + A[j,\ell] \geq A[i,\ell] + A[j,k]$). An $n \times n$ *falling staircase anti-Monge matrix* is a matrix with blanks such that for every blank all elements below and to the left are blanks, and the anti-Monge inequality holds whenever the four concerned elements are non-blank. Similarly, an $n \times n$ *reverse falling staircase anti-Monge matrix* is a matrix with blanks such that for every blank all elements above and to the right are blanks, and the anti-Monge inequality holds whenever the four concerned elements are non-blank. Finally, an $n \times n$ matrix A is *totally monotone* if, for every $1 \leq i < j \leq n$ and $1 \leq k < \ell \leq n$, $A[i,k] \leq A[i,\ell]$ implies $A[j,k] \leq A[j,\ell]$.

0	1	2	2	2
-1	0	1	1	2
-2	-1	0	1	2
-3	-2	-1	0	1
-4	-3	-2	-1	0

4				
3	2			
2	1			
4	4	2	6	

	1	2	2	2
			1	2
			1	2
				1

■ **Figure 3** Anti-Monge matrix, reverse falling staircase anti-Monge matrix, and falling staircase anti-Monge matrix.

Let us now recall some basic facts regarding Monge matrices.

▶ **Observation 8.** *Adding the same value to every element in a row (or a column) of an anti-Monge matrix results in an anti-Monge matrix.*

▶ **Observation 9.** *To check if an array is anti-Monge it is sufficient to check if every contiguous 2×2 submatrix is anti-Monge.*

The following lemma follows from the well-known SMAWK algorithm [2].

▶ **Lemma 10** (Lemma 3.3 in Aggarwal et al. [1]). *All row maxima in a reverse falling staircase totally monotone matrix can be found in $O(n)$ time.*

By transposing the matrix and observing that being anti-Monge implies being totally monotone we obtain the following.

▶ **Corollary 11.** *All column maxima in a falling staircase anti-Monge matrix can be found in $O(n)$ time.*

LIS-in-range queries. Let $S[1:n]$ be the input sequence. Define M as an $(n+1) \times (n+1)$ matrix with 0-indexed rows and columns, such that $M[i,j]$ is the length of LIS in $S[i+1:j]$ for $i < j$ and $M[i,j] = j - i$ otherwise (the anti-Monge matrix in Figure 3 is such a matrix for the sequence $(3, 4, 1, 2)$). As hinted by our example, this matrix turns out to have a rather special structure as observed by Tiskin [20]. We describe this structure in the following.

Let S' be the sequence obtained by sorting S (recall that S consists of distinct elements), and observe that LIS of S is the same as a longest common sequence (LCS, for short) of S and S'. Thus, we can think that $M[i,j]$ is LCS of S' and $S[i+1:j]$. As such, the following result can be shown (see [20] and the references therein).

▶ **Lemma 12.** *M is anti-Monge.*

Our algorithm needs to access the elements of M. Since the matrix contains $(n+1)^2$ elements, it is too large to be explicitly stored in memory. Fortunately, Tiskin also showed how to create in $O(n \log^2 n)$ time an $O(n)$-space implicit representation of M that allows us to obtain any of its elements in $O(\log n)$ time [20]. Before we present the internals of this representation, we need to introduce some additional definitions illustrated in Figure 4.

▶ **Definition 13.** *Let A be any $n \times n$ matrix. Its distribution matrix A^Σ is an $(n+1) \times (n+1)$ matrix defined by $A^\Sigma[x, y] = \sum_{i \geq x, j < y} A[i, j]$, for every $1 \leq x \leq n+1, 1 \leq y \leq n+1$.*

▶ **Definition 14.** *A permutation matrix is a square matrix that has exactly one 1 in every row and column, and the remaining elements are equal to 0.*

0	1	0
1	0	0
0	0	1

0	1	2	3
0	1	1	2
0	0	0	1
0	0	0	0

🟧 **Figure 4** A permutation matrix A and its distribution matrix A^Σ.

Now, we can provide the final ingredients of the construction. For two strings w_1 and w_2 of length d, Tiskin defines in [20] a $(2d+1) \times (2d+1)$ matrix L in the following way. Let w_2' be the string equal to $?^d w_2 ?^d$, whose positions are indexed from $-(d-1)$ to $2d$. The rows of L are indexed from $-d$ to d, while the columns of L are indexed from 0 to $2d$. The elements of L are defined by $L[i,j] = \text{LCS}(w_1, w_2'[i+1:j])$ if $j > i$, and $L[i,j] = j - i$ otherwise. In this definition, it is assumed that ? matches any character. If w_2 is the input sequence S and w_1 is S' then, for $0 \leq i, j \leq n$ we have $L[i,j] = M[i+1, j+1]$. Tiskin proved (Theorem 4.10 in [20]) that there exists $2d \times 2d$ permutation matrix P such that $L[i,j] = j - i - P^\Sigma[i,j]$. Furthermore, he provided an $O(n \log^2 n)$-time algorithm that finds all the

non-zero entries of P (Algorithm 8.2 in [20]). Having all the non-zero entries of P we can apply a dominance counting structure of Chazelle [7] that can be constructed in $O(n \log n)$ time, uses $O(n)$ space, and calculates $P^\Sigma[i, j]$ and hence also $M[i+1, j+1]$ in $O(\log n)$ time. Summarising, in $O(n \log^2 n)$ time we obtain a structure that returns any element of M in $O(\log n)$ time. We similarly obtain a matrix storing the length of LDS of every $S[i+1 : j]$.

Description of the algorithm.   Let $S[1 : n]$ be the input sequence. For every $1 \le x \le n$, let $\mathsf{res}[x]$ be the length of a longest k-rollercoaster in $S[1 : x]$, and $\mathsf{inc}[x]$ (respectively, $\mathsf{dec}[x]$) be the length of a longest k-rollercoaster in $S[1 : x]$ with the last run increasing (respectively, decreasing). Note that we do not require that these k-rollercoasters contain $S[x]$. Then, $\mathsf{res}[x] = \max\{\mathsf{dec}[x], \mathsf{inc}[x]\}$, for $1 \le x \le n$. Firstly, we introduce two structural lemmas.

▶ **Lemma 15.** *Let A be a k-rollercoaster in $S[1 : i]$ with the last run decreasing, and r be an increasing subsequence in $S[i : n]$ such that $|r| \ge k$. Then there exists a k-rollercoaster in $S[1 : n]$ of length at least $|A| + |r| - 1$ with the last run increasing.*

Proof.   Let A' be the sequence consisting of all elements from both A and r. Recall that a sequence is a k-rollercoaster if every run has length at least k. In order to show that A' is a k-rollercoaster with last run increasing we need to consider three cases: the first element of r is the last element of A, the first element of r is greater than the last element of A, and the first element of r is less than the last element of A.

In the first case, all runs in A' but the last are the same as in A, and the last run is equal to r. Since A is a k-rollercoaster and $|r| \ge k$ we conclude that A' is a k-rollercoaster. A and r have one common element, so $|A'| = |A| + |r| - 1$.

In the second case, all runs in A' but the last are also the same as in A, and the last run consists of the last element of A and r. Again we conclude that A' is a k-rollercoaster. Since A and r have no common elements, $|A'| = |A| + |r|$.

In the third case, all runs in A' but the last two are the same as in A. The second-to-last run in A' consist of the last run of A and the first element of r, and the last run in A' is r. Hence, A' is a k-rollercoaster. Since A and r have no common elements, $|A'| = |A| + |r|$.   ◀

▶ **Lemma 16.** *Consider a longest k-rollercoaster in $S[1 : n]$ with the last run increasing (respectively, decreasing), and let r be its last run with the first element $S[i]$. Then r is a longest increasing (respectively, decreasing) subsequence in $S[i : n]$.*

Proof.   By contradiction. Let A be a longest k-rollercoaster from the statement of the lemma, and suppose that there exists a longer increasing sequence r' in $S[i : n]$. Let A' be the prefix of A ending at $S[i]$. Observe that $|A'| = |A| - |r| + 1$. Then by Lemma 15 there exists a k-rollercoaster in S of length at least $|A'| + |r'| - 1 = |A| - |r| + |r'| > |A|$.   ◀

The above lemmas allow us to obtain the formula for calculating the arrays inc and dec. Recall that $M[i, j]$ is the length of LIS in $S[i+1 : j]$. Let M' be the matrix obtained from M by replacing all elements less than k by $-\infty$, and let $Z(j, j')$ be the set of indices $j \le i \le j'$ such that length of LIS in $S[i : j']$ is at least k (or, in other words, $M'[i-1, j'] \ne -\infty$).

▶ **Proposition 17.** *For every $1 \le x \le n$, the following holds:*

$$\mathsf{inc}'[x] = \max\{\mathsf{dec}[i] + M'[i-1, x] - 1 : i \in Z(1, x)\}, \mathsf{inc}[x] = \max\{\mathsf{inc}'[x], M'[0, x]\}.$$

If $Z(1, x)$ is empty then we set $\mathsf{inc}'[x] = 0$.

Proof. By Lemma 15 we obtain that for every $i \in Z(1, x)$ there exists a k-rollercoaster in $S[1 : x]$ with the last run increasing of length at least $\mathsf{dec}[i] + M'[i - 1, x] - 1$. We conclude that $\mathsf{inc}'[x]$ is less or equal to the length of a longest k-rollercoaster with the last run increasing in $S[1 : x]$. Observe that $M'[0, x]$ corresponds to an increasing run of length at least k or is equal to $-\infty$. We obtain that $\mathsf{inc}[x]$ is less or equal than the length of a longest k-rollercoaster with the last run increasing in $S[1 : x]$.

For the converse, consider a k-rollercoaster A with the last run increasing in $S[1 : x]$. If A consists of just a single run then its length is $M'[0, x]$. Otherwise, let $S[i]$ be the first element in the last run of A. Then by Lemma 16 the length of the last run is equal to $M'[i - 1, x]$ and the length of A is $\mathsf{dec}[i] + M'[i - 1, x] - 1$. Overall, the length of A is at most $\mathsf{inc}[x]$. ◄

Proposition 17 cannot be applied directly if we aim to achieve the announced $O(n \log^2 n)$ time complexity, and we need to introduce some auxiliary definitions. For every $1 \le d \le x$ we define $\mathsf{inc}_d[x]$ as follows:

$$\mathsf{inc}'_d[x] = \max\{\mathsf{dec}[i] + M'[i - 1, x] - 1 : i \in Z(1, d - 1)\}, \mathsf{inc}_d[x] = \max\{\mathsf{inc}'_d[x], M'[0, x]\}.$$

If $Z(1, d - 1)$ is empty then we set $\mathsf{inc}'_d[x] = 0$. In other words, $\mathsf{inc}_d[x]$ is equal to the length of a longest k-rollercoaster in $S[1 : x]$ with the last run increasing and starting at an element $S[i]$ with $i < d$ or LIS of $S[1 : n]$ of length at least k. Thus, $\mathsf{inc}_1[x]$ is equal to either 0 or the length of a LIS in $S[1 : x]$. We similarly define $\mathsf{dec}_d[x]$.

▶ **Observation 18.** *For every $j > i - k + 1$, $\mathsf{inc}_j[i] = \mathsf{inc}[i]$.*

We describe a function COMPUTE that receives a contiguous subsequence $S[i : j]$ together with the previously calculated arrays $\mathsf{inc}_i[i : j]$ and $\mathsf{dec}_i[i : j]$, and returns the arrays $\mathsf{inc}[i : j]$ and $\mathsf{dec}[i : j]$. To calculate the length of a longest k-rollercoaster in $S[1 : n]$ we invoke the function with the whole $S[1 : n]$ and the arrays $\mathsf{inc}_1[1 : n]$, $\mathsf{dec}_1[1 : n]$ as arguments, and return the maximum over the two resulting arrays. Note that $\mathsf{inc}_1[1 : n]$ and $\mathsf{dec}_1[1 : n]$ can be calculated in $O(n \log n)$ time using Algorithm 1.

Let $m = \lceil \frac{i+j}{2} \rceil$. The main idea of COMPUTE is to call the function recursively for the left half to calculate $\mathsf{inc}[i : m - 1]$ and $\mathsf{dec}[i : m - 1]$. The next step is to calculate $\mathsf{inc}_m[m : j]$ and $\mathsf{dec}_m[m : j]$ using tools from the previous paragraphs (as described below). Finally, we recursively calculate $\mathsf{inc}[m : j]$ and $\mathsf{dec}[m : j]$. Concatenating the results from both recursive calls gives us the desired result. This is summarised in Algorithm 3.

Algorithm 3 Computing the length of a longest k-rollercoaster.

1: **procedure** COMPUTE(k, $S[i : j]$, $\mathsf{inc}_i[i : j]$, $\mathsf{dec}_i[i : j]$)
2: **if** $j - i + 2 \le k$ **then**
3:   $\{\mathsf{inc}[i : j], \mathsf{dec}[i : j]\} \leftarrow \{\mathsf{inc}_i[i : j], \mathsf{dec}_i[i : j]\}$
4:   **return** $\{\mathsf{inc}[i : j], \mathsf{dec}[i : j]\}$
5: $m \leftarrow \lceil \frac{i+j}{2} \rceil$
6: $\{\mathsf{inc}[i : m-1], \mathsf{dec}[i : m-1]\} \leftarrow$ COMPUTE(k, $S[i : m-1]$, $\mathsf{inc}_i[i : m-1]$, $\mathsf{dec}_i[i : m-1]$)
7: Compute $\mathsf{inc}_m[m : j]$ and $\mathsf{dec}_m[m : j]$
8: $\{\mathsf{inc}[m : j], \mathsf{dec}[m : j]\} \leftarrow$ COMPUTE(k, $S[m : j]$, $\mathsf{inc}_m[m : j]$, $\mathsf{dec}_m[m : j]$)
9: **return** $\{\mathsf{inc}[i : j], \mathsf{dec}[i : j]\}$

Computing $\text{inc}_m[m:j]$ and $\text{dec}_m[m:j]$.   We only describe how to calculate $\text{inc}_m[m:j]$, as $\text{dec}_m[m:j]$ can be computed by a similar approach. Recall the previously introduced matrix M', obtained by replacing values less than k by $-\infty$ in M. Let A_{inc} be the $(m-i) \times (j+1-m)$ matrix with rows indexed from i to $m-1$ and columns indexed from m to j satisfying:

$$A_{\text{inc}}[x,y] = \begin{cases} \text{dec}[x] + M'[x-1,y] - 1 & \text{when } M'[x-1,y] \neq -\infty, \\ \text{blank} & \text{otherwise.} \end{cases}$$

Since we are able to retrieve any element of M' in $O(\log n)$ time using LIS-in-range queries, and the value of $\text{dec}[x]$, for every $i \leq x \leq m-1$, is already available, each element of A_{inc} can be calculated in $O(\log n)$ time. Furthermore, we have the following property.

▶ **Proposition 19.** *A is a falling staircase anti-Monge matrix.*

Proof. By Lemma 12 M is an anti-Monge matrix. By Observation 8 this is still the case if we add the same value to all elements in the same row.

To prove that A is a falling staircase matrix consider a non-blank element $A[i,j]$. Then $M[i,j] \geq k$. But this implies $M[i-1,j] \geq k$ and $M[i,j+1] \geq k$ (as long as $i > 1$ and $j < n$), so all elements above and to the right are also non-blank as required. ◀

▶ **Proposition 20.** *For every $m \leq \ell \leq j$, $\text{inc}_m[\ell]$ is equal to either $\text{inc}_i[\ell]$ or the maximum in the ℓth column of A.*

Proof. For every $m \leq \ell \leq j$, $\text{inc}_m[\ell]$ is equal to either $\text{inc}_i[\ell]$ or $\max\{\text{dec}[j] + M'[j-1,\ell] - 1 : j \in Z(i,m-1)\}$. However, the latter is exactly the maximum in the ℓth column of A. ◀

▶ **Lemma 21.** *We can compute $\text{inc}_m[m:j]$ and $\text{dec}_m[m:j]$ in $O((j-i+1)\log n)$ time.*

Proof. By Proposition 20 computing $\text{inc}_m[m:j]$ reduces to finding all the column maxima in A. Since A is a falling staircase anti-Monge matrix, we can use the algorithm from Corollary 11. Access to any element of A requires $O(\log n)$ time, so in total we obtain $O((j-i+1)\log n)$ time complexity. ◀

We can now state with the main result of this section.

▶ **Theorem 22.** *For every sequence $S[1:n]$ and $k \geq 3$, the length of a longest k-rollercoaster in S can be found in $O(n\log^2 n)$ time.*

Proof. The algorithm needs $O(n\log^2 n)$ preprocessing time to construct the LIS-in-range (and LDS-in-range) structure. We compute $\text{inc}_1[1:n]$ and $\text{dec}_1[1:n]$ in $O(n\log n)$ time using Algorithm 1. Then, we call the recursive function COMPUTE. By Lemma 21 a call of the function on $S[i:j]$ takes $O((j-i+1)\log n)$ time, so its running time is described by the recurrence $T(n) = 2T(n/2) + O(n\log n)$ that solves to $O(n\log^2 n)$. Thus, the overall time complexity is $O(n\log^2 n)$. ◀

5   Lower Bound

In the final section of our paper, we prove that any comparison-based algorithm computing the length of a longest k-rollercoaster in a permutation S of $\{1, \ldots, n\}$, for $4 \leq k \leq \frac{n}{3}$, performs at least $\Omega(n\log k)$ comparisons. Let T be a binary comparison tree associated with an algorithm that computes the result. The number of comparisons made in the algorithm is equal to the height of T, and this is a lower bound on the execution time of the algorithm.

Let A be a partial ordering associated with a path from the root to some leaf of T. Since the algorithm cannot distinguish between permutations following the same path, every permutation consistent with A has to give the same result. Our approach is to first identify a set U of permutations of $\{1, \ldots, n\}$ such that $\log |U| = \Theta(n \log k)$, and any ordering associated with a leaf of T can be consistent with at most one permutation from U. Hence, the number of leaves in T is at least $|U|$. Since the height of a binary tree is at least logarithm of the number of leaves, this will show that the height of T, and hence also the number of comparison performed by the algorithm, is at least $\Omega(\log |U|) = \Omega(n \log k)$.

We first recall the set Γ of $\ell^{n-2\ell}$ permutations of $\{1, \ldots, n\}$ proposed by Fredman in [10], where ℓ is a parameter. These permutations are essentially different inputs S for an algorithm computing the length of LIS, each leading to a different leaf in the comparison tree.

So, essentially, we want to construct input sequences $(x_1, \ldots, x_n)$, with their elements $x_1, \ldots, x_n$ chosen so that certain linear orderings of the x_is are induced. To create a permutation from Γ we partition $(x_1, \ldots, x_n)$ into ℓ subsequences $P_1, P_2, \ldots, P_\ell$. To simplify the exposure, let ℓ_{prefix} of a sequence be its prefix of length ℓ, while the ℓ_{suffix} is its suffix of length ℓ; the remaining $n - 2\ell$ elements are called ℓ_{middle} of the sequence. We partition $(x_1, \ldots, x_n)$ in the following way: the ith element of ℓ_{prefix} (that is, x_i) and the ith element of ℓ_{suffix} ($x_{n-\ell+i}$) belong to P_i. Each element from ℓ_{middle} of the sequence belongs to an arbitrary chosen part P_j. This gives us $\ell^{n-2\ell}$ different partitions. For a partition $P_1, \ldots, P_\ell$, we assign values from $\{1, \ldots, n\}$ to the input sequence in such a way, that the elements of each part P_i form a decreasing sequence and, for $1 \ i \le \ell$, each element of P_i is less than any element of P_{i+1} (see Figure 5). So, each such possible assignment gives us a permutation from Γ. LIS of any permutation from Γ is of length ℓ because it contains one element from each P_i. LDS of any permutation of Γ is no longer than $n - 2\ell + 2$ because it contains at most one element from ℓ_{prefix} and at most one from ℓ_{suffix}.

▶ **Proposition 23.** *Each permutation from Γ can be split into ℓ descending subsequences in only one way. For two different permutations from Γ these ways of splitting are different.*
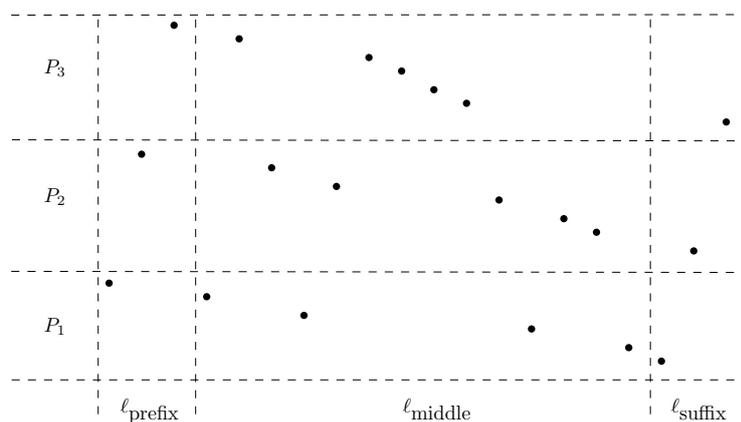
Figure 5 Example permutation $P \in \Gamma$ for $\ell = 3$ in a plane. In this figure, we have $P = (6, 13, 20, 5, 19, 12, 4, 11, 18, 17, 16, 15, 10, 3, 9, 8, 2, 1, 7, 14)$.

We now consider the algorithm computing the length of a longest k-rollercoaster. Using the permutations from Γ we create a set U of $k^{n\frac{k-3}{3k-3}}$ permutations of $\{1, \ldots, n\}$, again with the same principle behind: they should be input sequences which lead to different paths in the comparison tree associated to an algorithm computing the length of a longest

k-rollercoaster. Observe that $\log\left(k^{n\frac{k-3}{3k-3}}\right) = \Theta(n\log k)$, so this would imply the desired lower bound of $\Theta(n\log k)$ on the number of comparisons done by an algorithm to compute the length of a longest k-rollercoaster.

A permutation from U is obtained as follows. Suppose that $(3k-3)$ divides n. Split the sequence $(x_1, \ldots, x_n)$ into $\frac{n}{3k-3}$ *blocks* (contiguous subsequences) of size $3k-3$. We will assign to the elements of the ith contiguous block $(x_{i(3k-3)+1}, \ldots, x_{(i+1)(3k-3)})$ distinct values from the set $\{i(3k-3)+1, \ldots, (i+1)(3k-3)\}$, as follows. In every block, use one of the permutations from Γ (with the parameter ℓ set to k) to values to the elements $x_{i(3k-3)+1}, \ldots, x_{(i+1)(3k-3)}$ of that block, and then assign values to those elements according to that ordering. In this way, we can create $|\Gamma|^{\frac{n}{3k-3}} = (k^{k-3})^{\frac{n}{3k-3}}$ permutations of $\{1, \ldots, n\}$. Observe that in every block the length of a longest decreasing subsequence is less than k. Since every block consists of strictly greater values than the previous ones, a longest decreasing subsequence of every permutation from U is less than k. A longest increasing subsequence of every element of Γ is equal to k, so a longest k-rollercoaster for every element of U is equal to $\frac{kn}{3k-3}$ and consists only of longest increasing subsequences corresponding to all the blocks glued one after the other. We can now show a result similar to Proposition 23.

▶ **Proposition 24.** *Each permutation from U can be split into $\frac{kn}{3k-3}$ descending subsequence in only one way. For two different permutations from U these ways of splitting are different.*

Having constructed the set U, we can proceed with the lower bound. Let A be a partial ordering associated with a path to some leaf of T (the comparison tree associated to the algorithm computing the length of a longest k-rollercoaster). Since the algorithm cannot distinguish between permutations following the same path, every permutation consistent with A has to give the same result. We recall the following lemma.

▶ **Lemma 25** (Lemma 3.6 in [10]). *Let $\leq$ be a partial ordering defined on S. The maximum length of LIS in S associated with any linear embedding of this ordering, is equal to the minimum number of decreasing subsequences relative to $\leq$ into which S can be partitioned.*

Now we can prove the following.

▶ **Lemma 26.** *Let A be partial ordering associated with the path from the root to a leaf of T. Only one permutation from U can be consistent with A.*

Proof. Consider $S \in U$ that is consistent with A, and let $D = \frac{kn}{3k-3}$ be the length of its LIS. Now let m be the minimum number of decreasing subsequences relative to the results of the comparisons made on the path A into which S can be partitioned. If $m < d$ then S is consistent with A, so we can partition S into the same decreasing subsequences, but S cannot be divided into less than than d decreasing subsequences, a contradiction. If $m > d$ then by Lemma 25 there exists a permutation S' consistent with A with the length of LIS greater than d. S' follows the same path as S in the comparison tree, but has a longer k-rollercoaster (consisting only of LIS of S') than S, a contradiction. Thus, $m = d$ for any such S.

Consider two $S_1, S_2 \in U$ consistent with A. By Proposition 24, the only partition of S_1 into d decreasing sequences is different from the only such partition of S_2 (into d decreasing sequences), so A can be consistent with only one permutation, a contradiction. ◀

Thus, each permutation from U corresponds to a distinct leaf of T, making the depth of T at least $\log|U| = \Theta(n\log k)$ as required and proving the following theorem.

▶ **Theorem 27.** *For every k satisfying $4 \leq k \leq \frac{n}{3}$, any comparison-based algorithm that computes the length of a longest k-rollercoaster in a permutation of $\{1, \ldots, n\}$ performs at least $\Omega(n\log k)$ comparisons.*

──── **References** ────

1   Alok Aggarwal and Maria M. Klawe. Applications of generalized matrix searching to geometric algorithms. *Discrete Applied Mathematics*, 27(1-2):3–23, 1990.

2   Alok Aggarwal, Maria M. Klawe, Shlomo Moran, Peter W. Shor, and Robert E. Wilber. Geometric Applications of a Matrix-Searching Algorithm. *Algorithmica*, 2:195–208, 1987.

3   David Aldous and Persi Diaconis. Longest Increasing Subsequences: From Patience Sorting to the Baik-Deift-Johansson Theorem. *Bulletin of the American Mathematical Society*, 36:413–432, 1999.

4   Sergei Bespamyatnikh and Michael Segal. Enumerating longest increasing subsequences and patience sorting. *Information Processing Letters*, 76(1):7–11, 2000.

5   Therese C. Biedl, Ahmad Biniaz, Robert Cummings, Anna Lubiw, Florin Manea, Dirk Nowotka, and Jeffrey Shallit. Rollercoasters and Caterpillars. In *ICALP*, volume 107 of *LIPIcs*, pages 18:1–18:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

6   Therese C. Biedl, Timothy M. Chan, Martin Derka, Kshitij Jain, and Anna Lubiw. Improved Bounds for Drawing Trees on Fixed Points with L-Shaped Edges. In *Graph Drawing and Network Visualization - 25th International Symposium, GD 2017, Revised Selected Papers*, volume 10692 of *Lecture Notes in Computer Science*, pages 305–317. Springer, 2017.

7   Bernard Chazelle. A Functional Approach to Data Structures and Its Use in Multidimensional Searching. *SIAM J. Comput.*, 17(3):427–462, 1988.

8   Maxime Crochemore and Ely Porat. Fast Computation of a Longest Increasing Subsequence and Application. *Information and Computation*, 208(9):1054–1059, 2010.

9   Paul Erdős and George Szekeres. A combinatorial problem in geometry. *Compositio Mathematica*, 2:463–470, 1935.

10 Michael L. Fredman. On computing the length of longest increasing subsequences. *Discrete Mathematics*, 11(1):29–35, 1975.

11 Hania Gajewska and Robert Endre Tarjan. Deques with Heap Order. *Information Processing Letters*, 22(4):197–200, 1986.

12 P. Gawrychowski, F. Manea, and R. Serafin. Fast and Longest Rollercoasters. *arXiv*, 2018. `arXiv:1810.07422`.

13 James W. Hunt and Thomas G. Szymanski. A Fast Algorithm for Computing Longest Common Subsequences. *Communications of the ACM*, 20(5):350–353, 1977.

14 Sergey Kitaev. *Patterns in Permutations and Words*. Springer, 2011.

15 S. Linton, N. Ruškuc, and V. Vatter, editors. *Permutation Patterns*. London Mathematical Society Lecture Note Series, vol. 376, Cambridge, 2010.

16 Dan Romik. *The Surprising Mathematics of Longest Increasing Subsequences*. Cambridge, 2015.

17 Craige Schensted. Longest increasing and decreasing subsequences. *Canadian Journal of Mathematics*, 13:179–191, 1961.

18 J. Michael Steele. Variations on the monotone subsequence theme of Erdős and Szekeres. In David Aldous, Persi Diaconis, Joel Spencer, and J. Michael Steele, editors, *Discrete Probability and Algorithms*, pages 111–131. Springer New York, 1995.

19 Xiaoming Sun and David P. Woodruff. The Communication and Streaming Complexity of Computing the Longest Common and Increasing Subsequences. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07, pages 336–345. Society for Industrial and Applied Mathematics, 2007.

20 Alexander Tiskin. Fast Distance Multiplication of Unit-Monge Matrices. *Algorithmica*, 71(4):859–888, 2015.