LRM-Trees: Compressed Indices, Adaptive Sorting, and Compressed Permutations ⋆

Jérémy Barbay[1], Johannes Fischer[2], and Gonzalo Navarro[1]

[1] Department of Computer Science, University of Chile,
{jbarbay,gnavarro}@dcc.uchile.cl
[2] Computer Science Department, Karlsruhe University, johannes.fischer@kit.edu

Abstract. LRM-Trees are an elegant way to partition a sequence of values into sorted consecutive blocks. They have been used to encode ordinal trees and to index integer arrays in order to support range minimum queries on them. We describe how they yield many other convenient results in a variety of areas: compressed indices for range minimum queries on partially sorted arrays, a new adaptive sorting algorithm, and a compressed data structure for permutations supporting direct and inverse application in time inversely proportional to the compressibility of the permutation.

1 Introduction

Introduced by Fischer [12] as a data structure to support *Range Minimum Queries* (RMQs) in constant time with no access to the main data, and by Sadakane and Navarro [34] as an internal construction for supporting navigation on ordinal trees, *Left-to-Right-Minima Trees* (LRM-Trees for short) are an elegant way to partition a sequence of values into sorted consecutive blocks, and to express the relative position of the first element of each block within a previous block. In this article we describe how the use of LRM-Trees yields many other convenient results in the design of data structures and algorithms on permutations:

1. We define three compressed indices supporting RMQs (Theorem 1, 2 and 3), that use less space when the indexed array is partially sorted. This directly improves upon the $2n + o(n)$ bits of uncompressed solutions [12] (optimal in the worst case over instances of size n), and is incomparable with compressed data structures supporting RMQs that profit from repetitions in the input (instead of partial order) [13].
2. Based on LRM-Trees, we define a new *measure of presortedness* for permutations (Definition 4). It combines some of the advantages of two well-known

⋆ Partially funded by Fondecyt grants 1-110066 and 1-120054, Chile, and a DFG grant (German Research Foundation). An early partial version of this article appeared in the *Proceedings of the 22th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 285–298, 2011.

measures, *runs* and *shuffled up-sequences*. The new measure is computable in linear time (like the former), but considers sorted sub-*sequences* (instead of only contiguous sub-*arrays*) in the input (like the latter, with some restrictions).
3. Based on this measure, we introduce a new sorting algorithm and its adaptive analysis (Theorem 4), asymptotically superior to sorting algorithms based on runs [3], and on many instances faster than sorting algorithms based on subsequences [24].
4. Also based on this measure, we design a compressed data structure for permutations that supports access to the permutation and its inverse in time inversely proportional to the presortedness of the permutation (Theorem 5). This improves upon previous similar results [3].

All of our results hold in the $\Omega(\lg n)$-word RAM model, where machine words contain $w \in \Omega(\lg n)$ bits and where arithmetic and logical operations are performed in $\mathcal{O}(1)$ time on those. Without taking advantage of repetitions, our results on RMQ-indexing and sorting can be easily extended to multisets, and the results on the compression of permutations would require small additional work: the permutations are only the worst case. Taking advantage of the repetitions in multisets is unlikely to yield much better results for RMQ indexes given that, from the point of view of RMQ indexes, instances varying in repetitions such as $(2, 1, 3)$ and $(2, 1, 2)$ are indistinguishable. In the case of sorting, we expect further improvements from taking advantage of repetitions in addition of partial order, but do not describe them in this work.

In our algorithms and data structures, we distinguish between the work performed on the input (often called "data complexity" in the literature) and the accesses to the internal data structures ("index complexity"). Data structures with low data complexity are important when the input is large and cannot be stored in the same level of the memory hierarchy where the index is stored. For instance, in the context of compressed indices like our RMQ structures, given a fixed limited amount of main memory, separating data complexity from index complexity identifies the instances whose compressed index fits in main memory while the main data must reside on disk. On these instances, between two data structures that support operators with the same total asymptotic complexity but distinct index complexity, the one with the lowest data complexity (0 in the case of Theorems 1 and 3) is more desirable.

Our results integrate concepts from two different topics: compressed data structures and range minimum query indices. In these areas, distinct names have been introduced for similar structures (Left-to-Right Minimal Trees [34] and 2d-Min Heaps [12]) and concepts (integrated encoding [2], self-index [25], non-systematic data structure [12,16] or encoding data structure). An additional contribution of this article is an overview of those concepts (in Section 2.1). The rest of Section 2 reviews other concepts necessary to follow the article, while already introducing other contributions such as a simplified construction algorithm for LRM-Trees (Lemma 1) and an abstraction of the results on compressing permutations (Lemma 3).

Then the main results of the article follow. In Section 3 we give our new results for RMQ data structures on compressible arrays (Theorems 1 to 3). In Section 4 we show how LRM-Trees yield improved sorting algorithms (Theorem 4), and also give novel insights on the structure of LRM-Trees (Lemmas 4 and 5). In Section 5 we use LRM-Trees to improve the representation of compressible permutations (Theorem 5). Finally, we conclude in Section 6.

2 Previous Work and Concepts

2.1 On the Various Types of Succinct Data Structures

Some terms (e.g., succinct indices and systematic data structures) on succinct data structures were introduced more than once, at similar times but with distinct names, which makes their classification more complicated than necessary. Given that our results cross several areas (such as compressed data structures for permutations and indices supporting range minimum queries), each using distinct names, we aim in this section to clarify the potential overlaps of concepts, to the extent of our knowledge. Of course, the experienced reader is encouraged to skip this section and to jump to those more directly related to our results.

Data Structures A *Data Structure* $\mathcal{D}$ (e.g., a run encoding of permutations [3]) specifies how to encode data from some *Data Type* $\mathcal{T}$ (e.g., permutations) so as to support the operators specified by a given *Abstract Data Type* $\mathcal{T}$ (e.g., direct and inverse applications). An *Encoding* is merely a data structure that supports *access* to the data, either through queries specific to this data type (e.g., i-th value of a permutation) or through access to a binary representation of the object (e.g., the i-th block of $\lg n$ bits of the integer array representing the permutation). The `access` operator is any operator through which (after sufficient applications) the data object represented can be uniquely identified.

Without any consideration on the support time of the `access` operator, information theory indicates that the best possible encoding, in the worst case over the $f(n)$ instances of fixed size n, uses at least $\lg f(n)$ bits. This information theory lower bound is used as a baseline (or *uncompressed baseline*) to express the space taken by a given data structure.

By definition, any data structure supporting `access`, and possibly some other operators, will use at least that much space. The additional space used to support operators efficiently is called the *redundancy* of the data structure. A data structure whose redundancy is asymptotically negligible when the size of the instance goes to infinity, that is, with $o(\lg f(n))$ redundancy, is called a *Succinct Data Structure* [20].

An example that is relevant in this paper is the abstract data type of bitvectors $B[1, n]$ where the `access` operator $B[i]$ gives the bit value at any position, the operator $\texttt{rank}_1(B, i)$ gives the number of 1's in the prefix $B[1, i]$, the operator $\texttt{select}_1(B, i)$ gives the position of the i-th 1 in B, reading B from left to right ($1 \leq i \leq n$), and operators $\texttt{rank}_0(B, i)$ and $\texttt{select}_0(B, i)$ are defined analogously

for 0-bits. A succinct data structure for this abstract data type requires $n + o(n)$ bits of space and supports all those operations in constant time [9, 26].

On the other hand, a data structure that does not support the `access` operator can use less space than the information theory lower bound. Its space usage can still be expressed as a function of the information theory lower bound (e.g., there is a data structure supporting RMQs on arrays of n integers using $o(n \lg n)$ bits without any access to the array; this space is less than the uncompressed baseline of $n \lg n$ bits). We discuss this point in more detail at the end of this section.

Compression Generally, one desires a data structure that can take less space than the uncompressed baseline on specific classes of instances. In those cases, the space used is analyzed through the definition of a measure $\mu(I)$ of the *compressibility* of an instance I (e.g., the zero-order empirical entropy of a binary string with n_0 0s and n_1 1s is $n \cdot H_0$, where $H_0 = \frac{n_0}{n} \lg \frac{n}{n_0} + \frac{n_1}{n} \lg \frac{n}{n_1} \in \lg \binom{n}{n_1} + \mathcal{O}(\lg n)$). Again, without any consideration on the time to support an operator, information theory indicates that the best possible encoding, in the worst case over the $f(n, \mu)$ instances I of fixed size n and fixed compressibility $\mu = \mu(I)$, uses at least $\lg f(n, \mu)$ bits. This information theory lower bound is used as a *compressed baseline* in order to express the space taken by a given compressed data structure, for the specific measure $\mu()$. Note that distinct measures can yield distinct analyses and optimality results. When possible, compressibility measures are *reduced* one to another (e.g., the first-order entropy of a string is no larger than its zero-order entropy), but in many cases they are incomparable.

Formally, a *Compressed Data Structure* (also called "opportunistic data structure" [11] or "ultra-succinct data structure" [21]) for a compressibility measure μ is a data structure that requires $\lg f(n, \mu) + o(\lg f(n))$ bits to encode any instance of size n and compressibility μ. Note that, while the encoding of the data is limited by the compressed baseline, the redundancy is only required to be asymptotically negligible when compared with the uncompressed baseline. This is of course undesirable when the data is highly compressible, since the redundancy may dominate the overall space, but it is the case of most space-efficient data structures in the literature (e.g., a bit-vector representation supporting the `access`, `rank` and `select` operators in constant time and using $nH_0 + o(n)$ bits of space [31]). A *Compressed Encoding* is a compressed data structure supporting at least the `access` operator, and a *Compression Scheme* is the algorithm producing this compressed encoding.

A stronger concept is that of a *Fully Compressed Data Structure* for a compressibility measure μ, which is a data structure requiring $\lg f(n, \mu) + o(\lg f(n, \mu))$ bits on any instance of size n and compressibility μ. While the $o(\cdot)$ term is asymptotic in n, it is useful to allow μ to depend on n too. Barbay et al. [1] gave an example of such a structure for strings $s[1..n]$ over an alphabet of size σ, supporting the `access`, `rank` and `select` operators in time $\mathcal{O}(\lg \lg \sigma)$ while using $nH_0(s) + o(nH_0(s))$ bits of space, where $H_0(s)$ is the zero-order entropy of s.

Such a result is not always achievable. For example if we use a bitmap to mark $n_1 \in \mathcal{O}(n/\lg^2 n)$ sampled positions in an array of length n, then the zero-order entropy of this bitmap will be $nH_0 \in \mathcal{O}(n \lg\lg n/\lg^2 n) \in o(n)$. In this case, the compressed data structure mentioned [31] has a redundancy of $\Theta(n \lg\lg n/\lg n) \subset \omega(nH_0)$. A fully compressed data structure for this problem [28] requires $nH_0 + \mathcal{O}(n_1)$ bits of space (and supports the `access`, `rank` and `select` operators in super-constant time). This is fully compressed as long as $n_1 \in o(n)$; otherwise $nH_0 \in \Theta(n)$ and the previous structure [31] becomes fully compressed.

Indices An *Index* is a structure that, given access to some data structure $\mathcal{D}$ supporting a defined abstract data type $\mathcal{T}$ (e.g., a data structure for bit-vectors supporting the `access` operator), extends the set of operators supported to a more general abstract data type $\mathcal{T}'$ (e.g., `rank` and `select` operators). By analogy with succinct data structures, the space used by an index is called *redundancy*.

A *Succinct Index* [2] or *Systematic Data Structure* [16] $\mathcal{I}$ is simply an index whose redundancy is asymptotically negligible in comparison to the uncompressed baseline when the size n of the instance goes to infinity, that is, $o(\lg f(n))$ bits. Although the concept of separating the encoding from the index was introduced to prove lower bounds on the trade-off between space and operation times of succinct data structures [17] (it had only been implicitly used before [33]), it has other advantages. In particular, the concept of succinct index is additive in the sense that, if $\mathcal{D}$ is a succinct data structure, then the data structure formed by the union of $\mathcal{D}$ and $\mathcal{I}$ is a succinct data structure as well. This property permits the combination of succinct indices for distinct abstract data types on similar data types [2], and suggests the possibility of a library of succinct indices that can be combined for each particular application. For example, the discussed compressed data structure for bit-vectors [31] is indeed formed by a compressed encoding of the data using $nH_0 + o(n)$ bits and offering access to $\mathcal{O}(\lg n)$ consecutive bits, plus a succinct index using $o(n)$ bits. The succinct index can be used alone if we have a different encoding of the bitmap offering the same functionality.

A *Compressed Index* for a compressibility measure μ is an index whose redundancy is asymptotically negligible in comparison to the compressed baseline for μ when n goes to infinity, that is, it uses $o(\lg f(n, \mu))$ bits of space.

Some clarifications The terms *integrated encoding* [2], *self-index* [25], *non-systematic data structure* [12, 16] or *encoding data structure* [7] were introduced to emphasize the distinction between indexing data structures (with the possibility to access the data that is being indexed) and those data structures that do not require access to any other data structure than themselves. The introduction of these new terms might seem superfluous as indexing data structures are part of the larger class of data structures, and an indexing data structure $\mathcal{I}$ for the abstract data type $\mathcal{T}$ immediately yields a data structure $\mathcal{D}$ for $\mathcal{T}$ when combined with any encoding scheme for $\mathcal{T}$. Yet the property highlighted

by those terms matters since, dropping the requirement of accessing the data through certain operator `access`, those data structures yield potentially lower redundancies than the combination of an index and an encoding. For example, concerning bit vectors, Golynski [17] showed that if a bit vector B is stored verbatim using n bits, then every index supporting the operators `access`, `rank`, and `select` in constant time must have redundancy $\Omega(n \lg\lg n/\lg n)$ bits, while Pătraşcu [30] gave an integrated encoding for B with constant time support of these operators and redundancy $\mathcal{O}(n/\operatorname{polylg} n)$ bits. In general an "integrated encoding", having less restrictions than an index, is potentially stronger in terms of space and/or time.

In the case of non-systematic data structures [12, 16] and encoding data structures [7], the emphasis is that those indexing data structures require much less space than the data they index, and are able to answer some queries (other than `access`, obviously) without any access to the main data. Of course, such an index can be seen as a data structure by itself, *for a distinct data type* (e.g., a Lowest Common Ancestor non-systematic succinct index of $2n + o(n)$ bits for labeled trees is also a simple data structure for ordinal trees). Those notions are relative to their context.

2.2 On the various Complexity Measures

Data vs Index Complexity of an operator We distinguish between the *data complexity* and the *index complexity* of operators supported by succinct indices, both of which compose the overall complexity. These measure separately the number of operations performed on the data (i.e., supported by the encoding) and on the index, respectively. This distinction is important in two contexts:

- In the external memory model, if the data structure is too large to reside in main memory and is therefore kept in external memory (that is expensive to access), while its index is small enough to be stored in RAM, it is preferable to use a succinct index of minimal data complexity, even if it comes at the expense of a larger index complexity. Traditional time complexity mixing both index and data complexity does not give enough information to make an informed choice.
- When applying a succinct index $\mathcal{I}$ to a data structure $\mathcal{D}$ that supports the `access` operator in time $a(n) \in \omega(1)$, the time complexity $t(n)$ of a given operator of data complexity $d(n)$ and index complexity $i(n)$ is $t(n) = a(n)d(n) + i(n)$. Once again, the traditional time complexity of the operator counting just $d(n) + i(n)$ does not give enough information for an informed choice.

Following these definitions, a non-systematic data structure can be seen as a succinct index of data complexity equal to zero, and the usual complexity of an operator supported by a succinct index is the sum of its data complexity and its index complexity.

Data vs Internal Complexity of an algorithm In a similar way, for a given algorithm A receiving as input an instance $\mathcal{I}$ that it accesses through the abstract data type $\mathcal{T}$ without specifying which data structure encodes $\mathcal{I}$, we distinguish its *data complexity*, the number of accesses to $\mathcal{I}$ through the various operators of $\mathcal{T}$, from its *internal complexity*, the number of operations performed internally without any access to the input data. The sum of data and internal complexity is the overall algorithm complexity.

For example, consider the compression of a text of n symbols on alphabet $[1..\sigma]$ via Huffman's code, represented as a tree: the data complexity is linear ($2n$, to be precise), as each symbol is accessed exactly twice (once to count frequencies and once to encode), while the internal complexity is $\mathcal{O}(n \lg \sigma)$ to encode the output bit by bit.

This distinction matters for similar reasons as the distinction between the data and index complexity of succinct indices:

- In the external memory model, the data might be too large to reside in main memory, as opposed to the internal structures.
- When applying the algorithm to a data structure $\mathcal{D}$ that supports the `access` operator in time $a(n) \in \omega(1)$, the time complexity $t(n)$ of an algorithm of data complexity $d(n)$ and internal complexity $i(n)$ is $t(n) = a(n)d(n) + i(n)$.

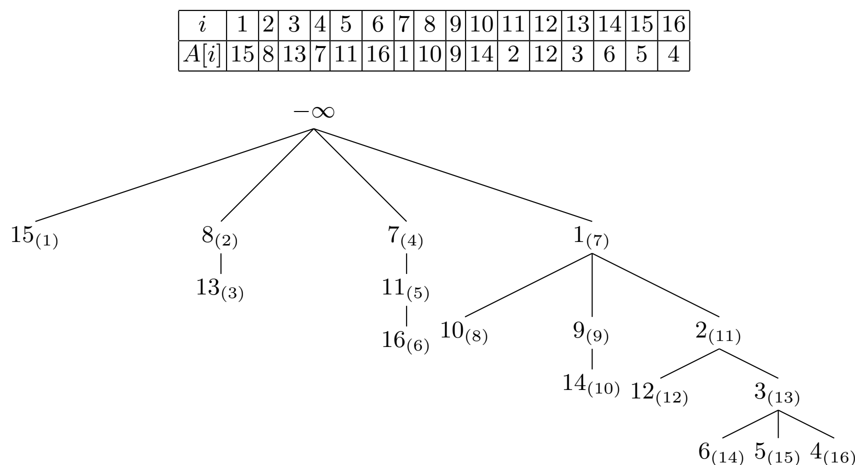
This is relevant to our results concerning the sorting of permutations and the construction of our compressed data structure for permutations, where some extensive computation is performed locally (e.g., computing an optimal partition of an LRM-Tree, computing an optimal merging tree of this partition) and other is performed on the data (e.g., identifying the runs, merging the sorted sequences).

2.3 Left-to-Right-Minima Trees

LRM-Trees partition a sequence of values into consecutive sorted blocks, and express the relative position of the first element of each block within a previous block. They were introduced under this name as an internal tool for basic navigational operations in ordinal trees [34] and, under the name "2d-Min Heaps", to index integer arrays in order to support range minimum queries on them [12].

Let $A[1..n]$ be an integer array. For technical reasons, we define $A[0] = -\infty$ as the "artificial" overall minimum of the array.

Definition 1 (Fischer [12]; Sadakane and Navarro [34]). *Consider an array $A[1..n]$ of n totally ordered objects, all larger than $A[0] = -\infty$. For $1 \leq i \leq n$, let $\text{PSV}_A(i) = \max\{j \in [0..i-1] \ : \ A[j] < A[i]\}$ denote the* Previous Smaller Value *(PSV) of position i. The* Left-to-Right-Minima Tree (LRM-Tree) *$\mathcal{T}_A$ of A is an ordered labeled tree with $n+1$ vertices, each labeled uniquely from $\{0, \ldots, n\}$. For $1 \leq i \leq n$, $\text{PSV}_A(i)$ is the parent node of i. The children of each node are ordered in increasing order from left to right.*



i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$A[i]$	15	8	13	7	11	16	1	10	9	14	2	12	3	6	5	4

Fig. 1. An example of an array and its LRM-Tree. The numbers at the nodes are the array values, and the smaller numbers in parentheses are their positions in the array.

Figure 1 gives an example of an integer array and its LRM-Tree. Fischer [12] gave a (complicated) linear-time construction algorithm with advantages that are not relevant for this paper. The following lemma shows a simpler way to construct the LRM-Tree in at most $2(n-1)$ comparisons within the array and overall linear time, which will be used in Theorems 4 and 5. The construction is similar to that of Cartesian Trees [15].

Lemma 1. *Given an array $A[1..n]$ of n totally ordered objects, there is an algorithm computing its LRM-Tree in at most $2(n-1)$ comparisons within A and $\mathcal{O}(n)$ total time.*

Proof. The computation of the LRM-Tree corresponds to a simple scan over the input array, starting at $A[0] = -\infty$, building down iteratively the current rightmost branch of the tree with increasing elements of the sequence until an element x smaller than its predecessor is encountered. At this point one climbs the rightmost branch up to the first node v holding a value smaller than x, and starts a new branch with a rightmost child of v of value x. As the root of the tree has value $A[0] = -\infty$ (smaller than all elements), the algorithm always terminates.

The construction algorithm performs at least $n-1$ comparisons (e.g., $A = (1, 2, \ldots, n)$) and at most $2(n-1)$ comparisons: the first two elements $A[0]$ and $A[1]$ can be inserted without any comparison as a simple path of two nodes (so $A[1]$ will be charged only once). For the remaining elements, we charge the last comparison performed during the insertion of an element x to the node of value x itself, and all previous comparisons to the elements already in the LRM-

Tree. Thus, each element is charged at most twice: once when it is inserted into the tree, and once when scanning it while searching for a smaller value on the rightmost branch. As in the latter case all scanned elements are removed from the rightmost path, this second charging occurs at most once for each element. Finally, the last element $A[n]$ is never scanned, and the first element $A[1]$ is inserted without any comparison. Hence the total number of comparisons is at most $2n - 2 = 2(n - 1)$. Since the number of comparisons within the array dominates the number of other operations, the overall construction time is also in $\mathcal{O}(n)$. $\square$

2.4 Range Minimum Queries

Range Minimum Queries (RMQ) have a wide range of applications in various data structures and algorithms, including text indexing [14], pattern matching [10], and more elaborate kinds of range queries [8]. We define them as follows:

Definition 2 (Range Minimum Queries [6]). *Consider an array $A[1..n]$ of n ordered objects. A* Range Minimum Query *consists of a pair of integers i and j such that $1 \leq i \leq j \leq n$, and its answer $\textsc{rmq}_A(i,j)$ is the leftmost position of a minimum in $A[i,j]$.*

For two given nodes i and j in a tree T, let $\textsc{lca}_T(i,j)$ denote their *Lowest Common Ancestor* (LCA) [6], that is, the deepest node that is an ancestor of both i and j. Now let $\mathcal{T}_A$ be the LRM-Tree of A. For two arbitrary nodes identified by their preorder ranks i and j in $\mathcal{T}_A$, $1 \leq i < j \leq n$, let $\ell = \textsc{lca}_{\mathcal{T}_A}(i,j)$. Then $\textsc{rmq}_A(i,j)$ is i if $\ell = i$ and is otherwise given by the child of ℓ that is on the path from ℓ to j [12].

Since there are succinct data structures supporting the LCA operator [12,21] in succinctly encoded trees in constant time, this yields a succinct index (which we improve in Theorems 1 and 3).

Lemma 2 (Fischer [12]). *Given an array $A[1..n]$ of n totally ordered objects, there is a non-systematic succinct index using $2n + o(n)$ bits and supporting RMQs in zero accesses to A and $\mathcal{O}(1)$ accesses to the index. This index can be built in $\mathcal{O}(n)$ time.*

Since we use LCA succinct data structures to define a succinct index for RMQ, it should be noted that, in these data structures, LCA queries are reduced to a particular case of RMQs, where the neighbors differ by exactly 1, denoted as ± 1RMQs. For ± 1RMQs, succinct data structures exist independently of LCAs, so there is no circular dependency in the definition of the solution.

2.5 Adaptive Sorting and Compression of Permutations

Sorting a permutation in the comparison model requires $\Theta(n \lg n)$ comparisons in the worst case over permutations of n elements. Yet, better results can be

achieved for some parameterized classes of permutations. Petersson and Moffat [29] summarized many of them; we describe here a few that are relevant to our work.

For a permutation π, Knuth [23] considered *Runs* (contiguous ascending subsequences), counted by $\rho = 1 + |\{i,\ 1 \leq i < n,\ \pi_{i+1} < \pi_i\}|$; Levcopoulos and Petersson [24] introduced *Shuffled Up-Sequences* and its generalization *Shuffled Monotone Sequences*, respectively counted by $|\mathsf{SUS}| = \min\{k,\ \pi$ is covered by k increasing subsequences $\}$, and $|\mathsf{SMS}| = \min\{k,\ \pi$ is covered by k monotone subsequences $\}$. Barbay and Navarro [3] introduced strict variants of some of those concepts, namely *Strict Runs* and *Strict Shuffled Up-Sequences*, where sorted subsequences are composed of consecutive integers (e.g., $(\mathbf{2}, \mathbf{3}, \mathbf{4}, 1, \mathit{5}, \mathit{6}, \mathit{7}, \mathit{8})$ has two runs but three strict runs), counted by $|\mathsf{SRuns}|$ and $|\mathsf{SSUS}|$, respectively. For any measure of disorder $\mathsf{X}()$ taken among those five, there is a variant of the merge-sort algorithm that sorts a permutation π, of size n and of measure of presortedness $\mathsf{X}(\pi) = \mathsf{x}$, in time $\mathcal{O}(n(1+\lg \mathsf{x}))$, which is within a constant factor of optimal in the worst case among instances of fixed size n and fixed values of $\mathsf{X}(\pi) = \mathsf{x}$ (this is not necessarily true for other measures of disorder [3, 24]). The idea central to those results is that, once we have identified x increasing subsequences in π that are already sorted, we can merge them in a balanced merging tree.

In this merging tree, each element is compared once in its way to the root. Thus the total merging cost is the sum of the number n_i of elements in each merging tree leaf (i.e., partition) times the depth l_i of such leaf, that is, $\sum_{i=1}^{r} n_i l_i$. When the leaves store partitions of different length, the merging cost can be reduced by rebalancing the tree so that leaves with longer sequences are closer to the root [3]. Actually, finding the optimal merging tree is exactly the same as finding an optimal code for the vector of frequencies $\mathsf{Seq} = \langle n_1, n_2, \ldots, n_r \rangle$, and the optimal tree is precisely the Huffman tree [19] of such sequence, as it minimizes $\sum_{i=1}^{r} n_i l_i$. The merging cost can then be expressed more precisely as a function of the *entropy* of the relative sizes of the sorted subsequences identified, where the entropy $\mathcal{H}(\mathsf{Seq})$ of a sequence $\mathsf{Seq} = \langle n_1, n_2, \ldots, n_r \rangle$ of r positive integers adding up to n is defined as $\mathcal{H}(\mathsf{Seq}) = \sum_{i=1}^{r} \frac{n_i}{n} \lg \frac{n}{n_i}$. Then the merging cost is upper bounded by $n(1+\mathcal{H}(\mathsf{Seq}))$. The overall sorting cost is that of merging plus the time to determine the partitions. Note the entropy satisfies $(r-1)\lg n \leq n\mathcal{H}(\mathsf{Seq}) \leq n \lg r$ by concavity of the logarithm.

Barbay and Navarro [3] observed that each adaptive sorting algorithm in the comparison model also describes an encoding of the permutation π that it sorts, so that it can be used to compress permutations from specific classes to less than the information-theoretic lower bound of $\lg(n!) \in n \lg n - n \lg e + (\lg n)/2 + \Theta(1)$ bits. Furthermore they used the similarity of the execution of the merge-sort algorithm with a Wavelet Tree [18], to support the application of $\pi()$ and its inverse $\pi^{-1}()$ in time logarithmic in the disorder of the permutation π (as measured by ρ, $|\mathsf{SRuns}|$, $|\mathsf{SUS}|$, $|\mathsf{SSUS}|$ or $|\mathsf{SMS}|$) in the worst case. We summarize their technique in Lemma 3 below, in a way parameterized by the partition chosen for the permutation, and focusing only on the merging part

of the sorting (e.g., the complexity of the SUS-Sorting algorithm differs from this result by a factor of two, because of the additional cost of computing the SUS-partition). We build on their slightly improved extended version [4], which reduces redundancy and improves time by combining compressed data structures for bitmaps with multiary wavelet trees.

Lemma 3 (Barbay and Navarro [3, 4]). *Given a partition of an array* π *of* n *totally ordered objects into* $|\mathsf{Seq}|$ *sorted subsequences of respective lengths* $\mathsf{Seq} = \langle n_1, n_2, \ldots, n_{|\mathsf{Seq}|}\rangle$*, these subsequences can be merged with* $n(1+\mathcal{H}(\mathsf{Seq}))$ *comparisons on* π *and* $\mathcal{O}(n(1+\mathcal{H}(\mathsf{Seq})))$ *total running time. This merging can be encoded using at most* $n\mathcal{H}(\mathsf{Seq}) + \mathcal{O}(|\mathsf{Seq}| \lg n) + o(n)$ *bits so that it supports the computation of* $\pi(i)$ *and* $\pi^{-1}(i)$ *in time* $\mathcal{O}(1 + \lg |\mathsf{Seq}| / \lg\lg n)$ *in the worst case* $\forall i \in [1..n]$*, and in time* $\mathcal{O}(1 + \mathcal{H}(\mathsf{Seq}) / \lg\lg n)$ *on average when* i *is chosen uniformly at random in* $[1..n]$*.*

The encoding of the merging is an encoding of the permutation: no further access is performed on the original array after the construction. Barbay et al. [1] give other similar tradeoffs by combining these ideas with novel sequence representations.

3 Compressed Indices for Range Minima

We now explain how to improve on the result of Lemma 2 for permutations that are partially ordered. We consider only the case where the input A is a permutation of $[1..n]$. If this is not the case, we can identify the elements in A with their rank, considering earlier occurrences of equal elements as smaller.

3.1 Strict Runs

Our first and simplest compressed data structure for RMQs uses an amount of space that is a function of $|\mathsf{SRuns}|$, the number of strict runs in π. Beside its simplicity, its interest resides in that it uses a total space within $o(n)$ bits on permutations such that $|\mathsf{SRuns}| \in o(n)$, and introduces techniques that we will use in Theorems 2 and 3.

Theorem 1. *Consider an array* $A[1..n]$ *of* n *totally ordered objects, composed of* $|\mathsf{SRuns}| \in o(n)$ *strict runs. Then there is a non-systematic compressed index using* $\lg\binom{n}{|\mathsf{SRuns}|} + o(n)$ *bits that can be computed in* $n-1+2(|\mathsf{SRuns}|-1) \in n+o(n)$ *data comparisons and* $\mathcal{O}(n)$ *overall time, and supports RMQs in zero accesses to* A *and* $\mathcal{O}(1)$ *accesses to the index.*

Proof. To build the index we mark the beginnings of the runs in A with a 1 in a bit-vector $B[1,n]$, and represent B with the compressed data structure from Raman et al. [32], using $\lg\binom{n}{|\mathsf{SRuns}|} + o(n)$ bits, in $n-1$ comparisons and $\mathcal{O}(n)$ overall time. Further, we define A' as the (conceptual) array consisting of the heads of A's runs ($A'[i] = A[\mathtt{select}_1(B,i)]$). We build the LRM-Tree from

Lemma 2 on A', using $2|\mathsf{SRuns}|(1+o(1))$ bits in $2(|\mathsf{SRuns}|-1)$ comparisons and $\mathcal{O}(|\mathsf{SRuns}|)$ overall time.

To answer query $\text{RMQ}_A(i,j)$, we compute $x = \mathtt{rank}_1(B,i)$ and $y = \mathtt{rank}_1(B,j)$, then compute $m' = \text{RMQ}_{A'}(x,y)$ as the minimum of the heads of those runs that overlap the query interval, and map it back to its position in A by $m = \mathtt{select}_1(B,m')$. Then if $m < i$, we return i as the final answer to $\text{RMQ}_A(i,j)$, otherwise we return m. The correctness of this algorithm follows from the fact that only i and the heads that are contained in the query interval can be the range minimum. Because the runs are strict, the former occurs if and only if the head of the run containing i is smaller than all other heads in the query range. □

For instance, given the permutation $A = (5,6,7,8,9,1,2,3,4)$ and a query from $i = 4$ to $j = 8$, the data structures can be represented as

$$\begin{array}{rl} & \overbrace{}\\ & ij \\ & 1\,2\,3\,4\,5\,6\,7\,8\,9 \\ A\ = & 5\,6\,7\,8\,9\,1\,2\,3\,4 \\ B\ = & 1\,0\,0\,0\,0\,1\,0\,0\,0 \\ A' = & 51 \end{array}$$

Then $x = 1$ and $y = 2$, $m' = 2$, $m = 6$, etc.

Note that, without the condition that $|\mathsf{SRuns}| \in o(n)$, the structure is not a compressed index, and holds little interest compared to the worst-case baseline. We explore in the following section a more general measure of partial order, ρ.

3.2 General Runs

The same idea of Theorem 1, applied to more general runs, yields another compressed index for RMQs, potentially smaller but this time requiring to compare two elements from the input to answer RMQs.

Theorem 2. *Consider an array $A[1..n]$ of n totally ordered objects, composed of $\rho \in o(n)$ runs. Then there is a systematic compressed index using $\lg\binom{n}{\rho} + o(n)$ bits, that can be computed in $n-1+2(\rho-1) \in n+o(n)$ data comparisons and $\mathcal{O}(n)$ overall time, and supports RMQs in 1 comparison within A and $\mathcal{O}(1)$ accesses to the index.*

Proof. We build the same data structures as in Theorem 1, using $\lg\binom{n}{\rho} + 2\rho + o(n)$ bits and in $n-1+2(\rho-1) \leq 3(n-1)$ data comparisons and $\mathcal{O}(n)$ overall time. To answer a query $\text{RMQ}_A(i,j)$, we compute $x = \mathtt{rank}_1(B,i)$ and $y = \mathtt{rank}_1(B,j)$. If $x = y$, we return i. Otherwise, we compute $m' = \text{RMQ}_{A'}(x+1,y)$, and map it back to its position in A by $m = \mathtt{select}_1(B,m')$. The final answer is i if $A[i] < A[m]$, and m otherwise. □

To achieve a non-systematic compressed index, which never accesses the array, and whose space usage is a function of ρ, we need more space and a more heavy machinery, as shown next. The main idea is that a permutation with few runs results in a compressible LRM-Tree, where many nodes have out-degree 1.

Theorem 3. *Consider an array $A[1..n]$ of n totally ordered objects, composed of $\rho \in O(n/\lg n)$ runs. Then there is a non-systematic compressed index using $2\rho \lg n - \rho \lg(\rho/e) + o(n)$ bits, that can be computed in $\mathcal{O}(n)$ overall time, and supports RMQs in zero accesses to A and $\mathcal{O}(1)$ accesses to the index.*

Proof. We build the LRM-Tree $\mathcal{T}_A$ from Lemma 1 directly on A, and then compress it with the tree representation of Jansson et al. [21], all in linear time.

To see that this results in the claimed space, let n_k denote the number of nodes in $\mathcal{T}_A$ with out-degree $k \geq 0$. Let $(i_1, j_1), \ldots, (i_\rho, j_\rho)$ be an encoding of the runs in A as (start, end), and look at a pair (i_x, j_x). We have $\text{PSV}_A(k) = k - 1$ for all $k \in [i_x + 1..j_x]$, and so the nodes in $[i_x..j_x]$ form a path in $\mathcal{T}_A$, possibly interrupted by branches stemming from heads i_y of other runs $y > x$ with $\text{PSV}_A(i_y) \in [i_x..j_x - 1]$. Hence $n_0 = \rho$, and $n_1 \geq n - \rho - (\rho - 1) > n - 2\rho$, as in the worst case the values $\text{PSV}_A(i_y)$ for $i_y \in \{i_2, i_3, \ldots, i_\rho\}$ are all different.

As an illustrative example, look again at the tree in Figure 1. It has $n_0 = 9$ leaves, corresponding to the runs (15), (8, 13), (7, 11, 16), (1, 10), (9, 14), (2, 12), (3, 6), (5), and (4) in A. The heads of the first four runs have a PSV of $A[0] = -\infty$, the next two head-PSVs point to $A[7] = 1$, the next one to $A[11] = 2$, and the last two to $A[13] = 3$. Hence, the heads of the runs "destroy" exactly four of the $n - n_0 + 1$ potential degree-1 nodes in the tree, so $n_1 = n - n_0 - 4 + 1 = 16 - 9 - 3 = 4$.

Now $\mathcal{T}_A$, with degree-distribution $n_0, \ldots, n_{n-1}$, is compressed into $nH^*(\mathcal{T}_A) + \mathcal{O}\left(\frac{n(\lg\lg n)^2}{\lg n}\right)$ bits [21], where

$$nH^*(\mathcal{T}_A) = \lg\left(\frac{1}{n}\binom{n}{n_0, n_1, \ldots, n_{n-1}}\right)$$

is the so-called *tree entropy* [21] of $\mathcal{T}_A$. This representation supports all navigational operations in $\mathcal{T}_A$ in constant time, and in particular those required for Lemma 2. A rough inequality yields a bound on the number of possible such LRM-Trees:

$$\begin{aligned}
\binom{n}{n_0, n_1, \ldots, n_{n-1}} &= \frac{n!}{n_0! n_1! \ldots n_{n-1}!} \\
&\leq \frac{n!}{n_0! n_1!} \\
&< \frac{n!}{\rho!(n - 2\rho)!} \\
&\leq \frac{n^{2\rho} e^\rho}{\rho^\rho} \ ,
\end{aligned}$$

from which one easily bounds the space usage of the compressed index:

$$\begin{aligned} nH^*(T) &\leq \lg\left(\frac{1}{n}\frac{n^{2\rho}e^{\rho}}{\rho^{\rho}}\right) \\ &= \lg\left(\frac{n^{2\rho-1}e^{\rho}}{\rho^{\rho}}\right) \\ &= (2\rho-1)\lg n + \rho\lg e - \rho\lg\rho \\ &< 2\rho\lg n - \rho\lg(\rho/e). \end{aligned}$$

Adding the space required to index the structure of Jansson et al. [21] yields the claimed space bound. □

Note that when there are "few" runs, $\rho \in o(n/\lg n)$, the overall space is simply $o(n)$, asymptotically less than the worst-case uncompressed baseline over instances of fixed size n. Without the hypothesis that $\rho \in O(n/\lg n)$, there would be too many runs for our analysis to give any interesting bound (as $2n + o(n)$ is already the worst-case baseline). Finally, if $\rho \in cn/\lg n + o(n/\lg n)$, for some constant $c < 2$, our analysis still shows an interesting space reduction, to $cn+o(n)$ bits.

4 Sorting Permutations

Barbay and Navarro [3] showed how to use the decomposition of a permutation π into ρ ascending consecutive runs of respective lengths Runs to sort adaptively to the entropy $\mathcal{H}(\mathsf{Runs})$. Those runs entirely partition the LRM-Tree of π: one can easily draw the partition corresponding to the runs considered by Barbay and Navarro [3] by iteratively tagging the leftmost maximal untagged leaf-to-root path of the LRM-Tree. For instance, the permutation of Figure 1 is of size $n = 16$ and has nine runs, $\{(15), (8,13), (7,11,16), (1,10), (9,14), (2,12), (3,6), (5), (4)\}$, of respective sizes given by the vector $\langle 1,2,3,2,2,2,2,1,1\rangle$, which has entropy $3 \times \frac{1}{16} \times \lg\frac{16}{1} + 5 \times \frac{2}{16} \times \lg\frac{16}{2} + 1 \times \frac{3}{16} \times \lg\frac{16}{3} \approx 3.0778$.

4.1 Optimal LRM-Partition

Note that *any* partition of the LRM-Tree into downward paths (so the values traversed by the paths are increasing) can be used to sort π, and a partition of smaller entropy yields a faster merging phase. To continue with the previous example, the nodes of the LRM-Tree of Figure 1 can be partitioned differently, so that the vector formed by the sizes of the increasing subsequences it describes has lower entropy. One such partition would be $\{(15), (8,13), (7,11,16),$ $(\mathbf{1},\mathbf{2},\mathbf{3},\mathbf{4}), (10), (9,14), (12), (6), (5)\}$, of respective sizes given by the vector $\mathsf{LRM} = \langle 1,2,3,\mathbf{4},\mathbf{1},2,\mathbf{1},1,1\rangle$, which has entropy $5 \times \frac{1}{16} \times \lg\frac{16}{1} + 2 \times \frac{2}{16} \times \lg\frac{16}{2} + 1 \times \frac{3}{16} \times \lg\frac{16}{3} + 1 \times \frac{4}{16} \times \lg\frac{16}{4} \approx 2.9528$, which is smaller than the entropy of 3.0778 of the partition given in the previous paragraph. See Figure 2 for a visual representation of this LRM-Partition, the one corresponding to the example

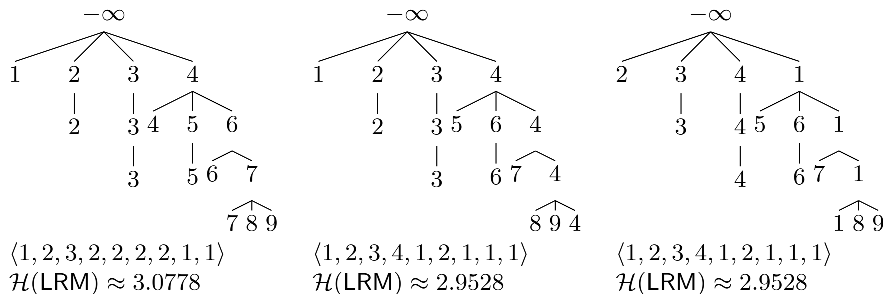


Fig. 2. Examples of LRM-Partitions of the LRM-Tree of Figure 1, where the numbers indicate the rank of the subsequence in the partition. On the left, the partition corresponding to a standard decomposition of the permutation into runs of consecutive positions; on the center, a general partition (here of optimal entropy); and finally on the right the left-most spinal LRM-Partition of the permutation.

of the previous paragraph, and the one corresponding to the optimal partition defined below.

Definition 3 (LRM-Partition). *An* LRM-Partition P *of an LRM-Tree* $\mathcal{T}$ *for an array* A *is a partition of the nodes of* $\mathcal{T}$ *into* ρ down-paths*, that is, paths starting at some branching node of the tree, and ending at a leaf. The* entropy *of* P *is* $\mathcal{H}(P) = \mathcal{H}(r_1, \ldots, r_\rho)$*, where* $r_1, \ldots, r_\rho$ *are the lengths of the down-paths in* P*.* P *is* optimal *if its entropy is minimal among all the LRM-partitions of* $\mathcal{T}$*. The entropy of such an optimal partition is the* LRM-entropy of the LRM-Tree $\mathcal{T}$ *and, by extension, the* LRM-entropy of the array A.

Note that, since there are exactly ρ leaves in the LRM-Tree, there will always be ρ down-paths in an LRM-partition. We first define a particular LRM-partition and prove that its entropy is minimal. Then we show how it can be computed in linear time.

Definition 4 (Left-Most Spinal LRM-Partition). *Given an LRM-Tree* $\mathcal{T}$*, the* left-most spinal chord *of* $\mathcal{T}$ *is the leftmost path among the longest root-to-leaf paths in* $\mathcal{T}$*; and the* left-most spinal LRM-partition *is defined recursively as follows. Removing the left-most spinal chord of* $\mathcal{T}$ *leaves a forest of shallower trees, which are partitioned recursively. The left-most spinal partition is the union of all the resulting LRM-partitions.* LRM *denotes the vector formed by the* ρ *lengths of the subsequences in the partition.*

For instance, the left-most spinal LRM-partition of the LRM-tree given in Figure 1 is quite easy to build: the first left-most spinal chord is $(-\infty, 1, 2, 3, 6) = (-\infty \to 6)$, whose removal leaves a forest of simple branches. The resulting partition is $\{(15), (8, 13), (7, 11, 16), (1, 2, 3, 6), (10), (9, 14), (12), (5), (4)\}$, of respective sizes given by the vector $\mathsf{LRM} = \langle 1, 2, 3, 4, 1, 2, 1, 1, 1 \rangle$, whose entropy is the same as our last example, ≈ 2.9528 (that was another optimal LRM-partition, different from the left-most spinal LRM-partition).

The left-most spinal LRM-partition, by successively extracting increasing subsequences of maximal length, actually yields a partition of minimal entropy, and hence an optimal LRM-partition, as shown in the following lemma.

Lemma 4. *The left-most spinal LRM-partition is optimal.*

Proof. Given an LRM-Tree $\mathcal{T}$, consider the leftmost leaf L_0 among the leaves of maximal depth in $\mathcal{T}$. We prove that there is always an optimal LRM-partition that contains the down-path $(-\infty \to L_0)$. Applying this property recursively in the trees produced by removing the nodes of $(-\infty \to L_0)$ from $\mathcal{T}$ yields the optimality of the left-most spinal LRM-partition.

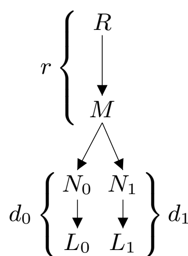


Fig. 3. Consider an arbitrary LRM-partition P and the down-path $(N_0 \to L_0)$ in P finishing at L_0. If $N_0 \neq -\infty$ (that is, N_0 is not the root), then consider the parent M of N_0 and the down-path $(R \to L_1)$ that contains M and finishes at some leaf L_1. Call N_1 the child of M on the path to L_1. Call r the number of nodes in $(R \to M)$, d_0 the number of nodes in $(N_0 \to L_0)$, and d_1 the number of nodes in $(N_1 \to L_1)$.

Consider an arbitrary LRM-partition P, the nodes R, M, N_0, N_1 and L_1 and the lengths r, d_0 and d_1 as described in Figure 3. Note that $d_1 \leq d_0$ because L_0 is one of the deepest leaves. Thus the LRM-partition P contains a down-path $(N_0 \to L_0)$ of length d_0 and another $(R \to L_1)$ of length $r + d_1$. We build a new LRM-partition P' by switching some parts of the down-paths, so that one is $(R \to L_0)$ and the other is $(N_1 \to L_1)$, with new down-path lengths $r + d_0$ and d_1, respectively.

Let $\langle n_1, n_2, \ldots, n_\rho \rangle$ be the lengths of the down-paths in P, such that $n\mathcal{H}(P) = n\mathcal{H}(n_1, n_2, \ldots, n_\rho) = n \lg n - \sum_{i=1}^{\rho} n_i \lg n_i$. Without loss of generality (the entropy is invariant to the order of the parameters), assume that $n_1 = d_0$ and $n_2 = r + d_1$ are the down-paths we have considered. They are replaced in P' by down-paths of length $n'_1 = r + d_0$ and $n'_2 = d_1$. The variation of $n\mathcal{H}(P)$ is $[(r+d_1)\lg(r+d_1)+d_0 \lg d_0]-[(r+d_0)\lg(r+d_0)+d_1 \lg d_1]$, which can be rewritten as $f(d_1) - f(d_0)$ with $f(x) = (r + x)\lg(r + x) - x \lg x$. Since the function $f(x)$ has a positive derivative and $d_1 \leq d_0$, the difference is non-positive (and strictly negative if $d_1 < d_0$, which would imply that P was not optimal). Iterating this argument until the path of the LRM-partition containing L_0 is rooted in $-\infty$ yields an LRM-partition of entropy no larger than that of the LRM-partition P, and one that contains the down-path $(-\infty \to L_0)$.

Applying this argument to an optimal LRM-partition demonstrates that there is always an LRM-partition that contains the down-path $(-\infty \to L_0)$. This, in turn, applied recursively to the subtrees obtained by removing the nodes

from the path $(-\infty \to L_0)$ from $\mathcal{T}$, shows the minimality of the entropy of the left-most spinal LRM-partition. □

The definition of the left-most spinal LRM-partition is constructive, but building this partition in linear time requires some more sophisticated techniques, described in the following lemma.

Lemma 5. *Given an LRM-Tree $\mathcal{T}$, there is an algorithm that computes its left-most spinal LRM-partition in linear overall time and zero accesses to the original array.*

Proof. Given an LRM-Tree $\mathcal{T}$ (and potentially no access to the array from which it originated), we first set up in linear time an array D containing the *depths* of the nodes in $\mathcal{T}$, listed in preorder. We then index D for range maximum queries in linear time using Lemma 2. Since D contains only internal data, the number of accesses to it matters only to the running time of the algorithm (they are distinct from accesses to the array at the construction of $\mathcal{T}$). Now the deepest node in $\mathcal{T}$ can be found by a range maximum query over the whole array, supported in constant time. From this node, we follow the path to the root, and save the corresponding nodes as the first subsequence. This divides A into disconnected subsequences, which can be processed recursively using the same algorithm, as the nodes in any subtree of $\mathcal{T}$ form an interval in D. We do so until all elements in A have been assigned to a subsequence. Note that, in the recursive steps, the numbers in D are not anymore the depths of the corresponding nodes in the remaining subtrees. Yet, as all depths listed in D differ by the same offset from their depths in any connected subtree, this does not affect the result of the range maximum queries. □

Of course, all operations can (and should) be performed directly on the array without the need to build an intermediary tree. For instance, each chord can be found recursively by parsing the array from the end to the begining, skipping elements larger than the last element added to the chord.

The left-most spinal LRM-partition is not much more expensive to compute than the partition into ascending consecutive runs [3]: at most $2(n-1)$ comparisons between elements of the array (to build the LRM-Tree) for the left-most spinal LRM-partition instead of $n-1$ for the Runs-partition. Note that $\mathcal{H}(\mathsf{LRM}) \leq \mathcal{H}(\mathsf{Runs})$, since the partition of π into consecutive ascending runs is just one LRM-partition among others.

4.2 LRM-Sorting

The concept of optimal LRM-partitions yields a new adaptive sorting algorithm, LRM-Sorting, which partitions a permutation using a LRM-Tree and merges the corresponding subsequences:

Theorem 4. *Let π be a permutation of size n, and of LRM-entropy $\mathcal{H}(\mathsf{LRM})$. The* LRM-Sorting *algorithm sorts π in a total of at most $n(3+\mathcal{H}(\mathsf{LRM}))-2$ comparisons between elements of π and in total running time of $\mathcal{O}(n(1+\mathcal{H}(\mathsf{LRM})))$.*

Proof. Obtaining the left-most spinal LRM-partition P composed of runs of respective lengths LRM through Lemmas 1 and 5 uses at most $2(n-1)$ comparisons between elements of π and $\mathcal{O}(n)$ total running time. Now sorting π is just a matter of applying Lemma 3: We merge the subsequences of P in $n(1+\mathcal{H}(\mathsf{LRM}))$ additional comparisons between elements of π and $\mathcal{O}(n(1+\mathcal{H}(\mathsf{LRM}))$ overall time. The sum of those complexities yields $n(3+\mathcal{H}(\mathsf{LRM}))-2$ data comparisons and $\mathcal{O}(n(1+\mathcal{H}(\mathsf{LRM})))$ running time. □

4.3 Comparison with Runs-Sorting

The additional sophistication of LRM-Sorting compared to Runs-Sorting comes with a cost: on instances where $\mathcal{H}(\mathsf{LRM}) = \mathcal{H}(\mathsf{Runs})$, that is, where the LRM-Partition is no better than that of contiguous runs, Runs-Sorting outperforms LRM-Sorting as it carries out $n-1$ fewer comparisons (this is the difference between building the LRM-Tree and just scanning for runs). Yet, the asymptotic performance of LRM-Sorting is never worse than that of Runs-Sorting, since $\mathcal{H}(\mathsf{LRM}) \leq \mathcal{H}(\mathsf{Runs})$. On the other hand, LRM-Sorting can be asymptotically faster than Runs-Sorting, by a factor of up to ρ on some instances. For instance, consider the permutation $\pi = (1, 2, 4, 3, 5, 7, 6, 8, 10, 9, 11, 12)$ of length $n = 12$. It has $\rho = 4$ runs whose lengths form the vector $\mathsf{Runs} = \langle 3, 3, 3, 3\rangle$ of entropy $\mathcal{H}(\mathsf{Runs}) = \lg 4 = 2$, while the LRM-Sorting algorithm yields a partition of sequences whose lengths form the vector $\mathsf{LRM} = \langle 9, 1, 1, 1\rangle$, of entropy $\mathcal{H}(\mathsf{LRM}) = \frac{9}{12}\lg\frac{12}{9} + 3 \times \frac{1}{12}\lg\frac{12}{1} \approx 1.2075 < 2$. A generalization of this example yields the following lemma.

Lemma 6. *LRM-Sorting performs asymptotically no worse than Runs-Sorting, and it outperforms Runs-Sorting on all instances where* $\mathcal{H}(\mathsf{LRM}) < \mathcal{H}(\mathsf{Runs}) - 1 + 1/n$*, by a factor of up to* $\Theta(\lg n)$ *for some classes of instances.*

Proof. The exact cost of both sorting algorithms is summarized as follows:

	Run-Sorting	LRM-Sorting
Partition	$n-1$	$2(n-1)$
Merge	$n(1+\mathcal{H}(\mathsf{Runs}))$	$n(1+\mathcal{H}(\mathsf{LRM}))$
Data comparisons	$n(2+\mathcal{H}(\mathsf{Runs}))-1$	$n(3+\mathcal{H}(\mathsf{LRM}))-2$
Time complexity	$\mathcal{O}(n(1+\mathcal{H}(\mathsf{Runs})))$	$\mathcal{O}(n(1+\mathcal{H}(\mathsf{LRM})))$

Since $\mathcal{H}(\mathsf{LRM}) \leq \mathcal{H}(\mathsf{Runs})$, $n(3+\mathcal{H}(\mathsf{LRM}))-2 \in \mathcal{O}(n(1+\mathcal{H}(\mathsf{Runs})))$ and hence LRM-Sorting performs asymptotically no worse than Runs-Sorting.

The extra data comparisons carried out by Runs-Sorting on top of LRM-Sorting are exactly $n(\mathcal{H}(\mathsf{Runs})-\mathcal{H}(\mathsf{LRM})-1)+1$, thus LRM-Sorting outperforms Runs-Sorting whenever $\mathcal{H}(\mathsf{Runs})-\mathcal{H}(\mathsf{LRM}) > 1-1/n$, and concedes to Runs-Sorting only when the value of $\mathcal{H}(\mathsf{LRM})$ is within $[\mathcal{H}(\mathsf{Runs})-1+1/n, \mathcal{H}(\mathsf{Runs})]$.

Now consider the class of permutations made of ρ runs of length n/ρ (we consider classes where ρ divides n to simplify), such that the insertion point (in the LRM-Tree) of each run is just before the last element of the previous run. Then, the Runs-partition has entropy $\mathcal{H}(\mathsf{Runs}) = \lg\rho$ (the lengths of the

runs form the vector $\mathsf{Runs} = \langle n/\rho, \dots, n/\rho \rangle)$, while the leftmost-spinal LRM-partition yields sequences forming the vector $\mathsf{LRM} = \langle n - \rho + 1, 1, \dots, 1 \rangle$, of entropy $\mathcal{H}(\mathsf{LRM}) = \frac{n-\rho+1}{n} \lg \frac{n}{n-\rho+1} + \frac{\rho-1}{n} \lg n \in \mathcal{O}((\rho/n) \lg n)$. For this class of permutations, the number of comparisons and time complexity of Runs-sorting are $\mathcal{O}(n \lg \rho)$, whereas those of LRM-sorting are $\mathcal{O}(n + \rho \lg n)$. The latter is asymptotically smaller whenever $\rho \in o(n)$ and $\rho \in \omega(1)$, by a factor of up to $\lg \rho$ whenever $\rho \in \mathcal{O}(n/\lg n)$. This factor can thus be as large as $\Theta(\lg n)$. □

4.4 Comparison with SUS-Sorting

The relation between the complexities of LRM-Sorting and SUS-Sorting is more complex. SUS-Sorting is asymptotically superior, as its performance is never more than a constant factor of the performance of LRM-Sorting, but the class of instances on which LRM-Sorting is in practice superior to SUS-Sorting is wider than the (narrow) class of instances on which Runs-Sorting is in practice superior to LRM-Sorting. Moreover, it is not easy to find the optimal SUS-partition.

SUS-Sorting can asymptotically outperform LRM-Sorting. Each down-path of the LRM-Tree corresponds to an ascending subsequence in π, but not all ascending subsequences correspond to down-paths of the LRM-Tree. For the array of Figure 1, for example, the SUS-partitioning algorithm described by Levcopoulos and Petersson [24] yields the partition $\{(15,16),(8,13,14),(7,11,12),(1,10),(9),(2,3,6),(5),(4)\}$, of $|\mathsf{SUS}| = 8$ runs (less than $\rho = 9$) of lengths forming the vector $\langle 2,3,3,2,1,3,1,1 \rangle$ of entropy $\mathcal{H}(\mathsf{SUS}) = 3 \times \frac{3}{16} \times \lg \frac{16}{3} + 2 \times \frac{2}{16} \times \lg \frac{16}{2} + 3 \times \frac{1}{16} \times \lg \frac{16}{1} \approx 2.8585$, lower than $\mathcal{H}(\mathsf{LRM}) \approx 2.9528$, itself already lower than the entropy of the decomposition into runs, $\mathcal{H}(\mathsf{Runs}) \approx 3.0778$. Hence, partitioning π optimally into $|\mathsf{SUS}|$ ascending subsequences potentially yields partitions with fewer runs and of smaller entropy.

On those instances where $\mathcal{H}(\mathsf{LRM})$ is much larger than $\mathcal{H}(\mathsf{SUS})$, the merging of an optimal LRM-partition can actually require many more comparisons than the merging of a SUS-partition. For example, for even $n > 2$, consider the permutation $\pi = (\mathit{1}, n/2{+}1,\ \mathit{2}, n/2{+}2, \dots,\ n/\mathit{2}, n)$. It has $\rho = n/2$ runs and the entropy of its left-most spinal LRM-partition is $\mathcal{H}(\mathsf{LRM}) = \lg \frac{n}{2}$, whereas it can be partitioned into $|\mathsf{SUS}| = 2$ increasing subsequences, which yields a SUS-partition of entropy $\mathcal{H}(\mathsf{SUS}) = \lg 2$. Figure 4 gives an example.

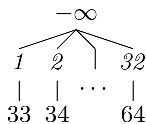


Fig. 4. The LRM-Tree of the permutation $\pi = (\mathit{1}, 33, \mathit{2}, 34, \dots, \mathit{32}, 64))$, where $\mathcal{H}(\mathsf{LRM})$ is much larger than $\mathcal{H}(\mathsf{SUS})$. Here $\rho = 32$ and $\mathcal{H}(\mathsf{LRM}) = \lg \frac{64}{2} = 5$, whereas $|\mathsf{SUS}| = 2$ and $\mathcal{H}(\mathsf{SUS}) = \lg 2 = 1$.

This shows that the time complexity of LRM-Sorting can be larger than that of SUS-Sorting, by a factor of up to $\Theta(\lg n)$, even over classes of instances of

fixed $\mathcal{H}(\mathsf{SUS})$. On the other hand, since $\mathcal{H}(\mathsf{SUS}) \leq \mathcal{H}(\mathsf{LRM})$, the complexity of SUS-Sorting is within a constant factor of the complexity of LRM-Sorting.

SUS-Sorting does not find an optimal SUS-Partition. The SUS-partitioning algorithm described by Levcopoulos and Petersson [24] makes a single pass over the permutation, keeping a set of increasing subsequences. Each new element is inserted at the end of the subsequence with the largest final element that is smaller than the new element. If there is none, the element starts a new subsequence. Their algorithm runs in time $\mathcal{O}(n(1 + \lg|\mathsf{SUS}|))$, and an improved variant [3] runs in time $\mathcal{O}(n(1 + \mathcal{H}(\mathsf{SUS})))$, where SUS is the vector formed by the lengths of the subsequences found by the algorithm.

This algorithm guarantees $|\mathsf{SUS}|$ to be minimal, but does not guarantee that the entropy $\mathcal{H}(\mathsf{SUS})$ of the partition produced is optimal, and no efficient algorithm is known to find the partition that minimizes this entropy. Furthermore, even though the *optimal* partition into increasing subsequences cannot have entropy higher than $\mathcal{H}(\mathsf{LRM})$, there exists permutations for which the partition found by both versions of the SUS-Sorting algorithm [3, 24] has entropy higher than $\mathcal{H}(\mathsf{LRM})$ (see Figure 5 for an example).

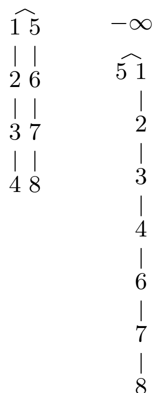


Fig. 5. On the permutation $\pi = (5, 1, 2, 3, 4, 6, 7, 8)$, the SUS-Sorting algorithm finds the partition $\{(5, 6, 7, 8), (1, 2, 3, 4)\}$ (on the left), which has entropy $\mathcal{H}(\mathsf{SUS}) = 1$, whereas the LRM-Sorting algorithm finds the partition $\{(5), (1, 2, 3, 4, 6, 7, 8)\}$ (LRM-Tree on the right), of entropy $\mathcal{H}(\mathsf{LRM}) = \frac{1}{8}\lg\frac{8}{1} + \frac{7}{8}\lg\frac{8}{7} \approx 0.5436$.

Lemma 7. *For every value of $n > 0$, there is a permutation over $[1..n]$ of LRM-Entropy $\mathcal{H}(\mathsf{LRM})$ on which both variants of the SUS-Sorting algorithm find a partition into increasing subsequences of entropy $\mathcal{H}(\mathsf{SUS})$ higher than $\mathcal{H}(\mathsf{LRM})$, by a factor of up to n^α for any constant $0 < \alpha < 1$.*

Proof. Without loss of generality, assume that $n = k \times a$ for some integers k and a. Consider the following permutation formed of $2k$ runs, basically "stealing" the first elements of $k-1$ straight ascending runs of length a to form a first descending sequence of length k: ($\boldsymbol{(k-1)a+1,\ (k-2)a+1,\ \ldots,\ 2a+1,\ a+1,\ 1}$, $2, 3, \ldots, a, \mathit{a+2}, \mathit{a+3}, \ldots, \mathit{2a}, 2a{+}2, 2a{+}3, \ldots, 3a, \ldots, \mathit{(k-1)a+2}, \ldots, \mathit{ka = n}$).

The SUS-Sorting algorithm [24] finds the partition $\{\ (1,2,\ldots,a),\ (a+1,a+2,\ldots,2a),\ (2a+1,2a+3,\ldots,3a),\ \ldots,\ ((k-1)a+1,((k-1)a)+2,\ldots,n)\}$, of size $|\mathsf{SUS}|=k$ and entropy $\mathcal{H}(\mathsf{SUS})=\lg k$.

However, the left-most spinal LRM-partition of the permutation has a much better entropy: it contains the singletons $(ia+1)$ for $i=1,\ldots,k-1$ and then a single increasing subsequence with all the rest. This decomposition has $\rho=k$ runs and entropy $\mathcal{H}(\mathsf{LRM})\in\mathcal{O}((k/n)\lg n)$. In particular, if $k=n^{1-\alpha}$ for any $0<\alpha<1$, then $\mathcal{H}(\mathsf{SUS})\in\Theta(\lg n)$ and $\mathcal{H}(\mathsf{LRM})\in\Theta(n^{-\alpha}\lg n)\subseteq o(1)$.

(Note that the SMS-Sorting algorithm [24] would find a much shorter partition into only two runs, $\{\ ((k-1)a+1,((k-2)a)+1,\ldots,2a+1,a+1,1),\ (2,3,\ldots,a,a+2,a+3,\ldots,2a,2a+2,2a+3,\ldots,3a,\ldots(k-1)a+2,\ldots,ka=n)\ \}$, of size $|\mathsf{SMS}|=2$ and entropy $\mathcal{H}(\mathtt{vSMS})=\mathcal{H}(k-1,n-k+1)\in\mathcal{O}((k/n)\lg(n/k))$.)
$\square$

LRM-Sorting outperforms SUS-Sorting in practice on a large class of instances. An optimal LRM-partition can be computed in $2(n-1)$ comparisons and $\mathcal{O}(n)$ time, while SUS-partitions are computed in $n(1+\mathcal{H}(\mathsf{SUS}))$ comparisons [3], and they may be suboptimal, as explained. Even if we assume that the optimal SUS-partitioning is found, on instances where the merging of the sequences identified by the LRM and SUS-partitions do not differ much in complexity (e.g., because they have the same entropy), this difference in the complexity of computing the partition makes LRM-Sorting a better choice.

For example, for $n>2$ multiple of 3, consider the permutation given by $\pi=(1,2,n,3,4,n-1,5,6,n-2,\ldots,2n/3-1,2n/3,2n/3+1)$. Its left-most spinal LRM-partition is the same as the SUS-partition of minimal size, resulting in the same vector $\mathsf{LRM}=\mathsf{SUS}=\langle 2n/3+1,1,\ldots,1\rangle$ and the same entropies, so that LRM-sorting outperforms SUS-sorting. Figure 6 exemplifies.

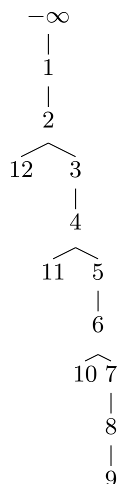


Fig. 6. The LRM-Tree of the permutation $\pi=(1,2,12,3,4,11,5,6,10,7,8,9)$, where $\mathsf{LRM}=\mathsf{SUS}=\langle 9,1,1,1\rangle$ so that $\mathcal{H}(\mathsf{LRM})=\mathcal{H}(\mathsf{SUS})$.

The high cost of computing the SUS-partition ($n(1 + \mathcal{H}(\mathsf{SUS}))$ additional comparisons within the array, as opposed to only $2(n-1)$ for the LRM-partition) means that on instances where $\mathcal{H}(\mathsf{LRM}) < 2\mathcal{H}(\mathsf{SUS}) - 1 + 2/n$, LRM-Sorting actually performs *fewer* comparisons within the array than SUS-Sorting.

Lemma 8. *On permutations such that $\mathcal{H}(\mathsf{LRM}) < 2\mathcal{H}(\mathsf{SUS}) - 1 + 2/n$, LRM-Sorting performs $n(2\mathcal{H}(\mathsf{SUS}) - \mathcal{H}(\mathsf{LRM}) - 1) + 2$ fewer comparisons than SUS-Sorting.*

Proof. Barbay and Navarro's improvement [3] of SUS-Sorting performs $2n(1 + \mathcal{H}(\mathsf{SUS}))$ comparisons within the array, which can be decomposed into two parts:

- $n(1 + \mathcal{H}(\mathsf{SUS}))$ comparisons within the array to compute a partition π into $|\mathsf{SUS}|$ sub-sequences that are minimal in size, yet not necessarily in entropy;
- $n(1 + \mathcal{H}(\mathsf{SUS}))$ additional comparisons within the array to merge the sub-sequences into a single ordered one (and $\mathcal{O}(n(1 + \mathcal{H}(\mathsf{SUS})))$ overall time complexity).

On the other hand, the combination of Lemmas 1 and 5 yields an optimal LRM-partition in $2(n-1)$ comparisons within the array (and $\mathcal{O}(n)$ additional internal operations), which is then merged in $n(1+\mathcal{H}(\mathsf{LRM}))$ comparisons within the array (and $\mathcal{O}(n(1 + \mathcal{H}(\mathsf{LRM})))$ additional internal operations) to merge the subsequences into a single ordered one.

The exact number of data comparisons performed by both sorting algorithms is summarized as follows:

	SUS-Sorting	LRM-Sorting
Partition	$n(1 + \mathcal{H}(\mathsf{SUS}))$	$2(n-1)$
Merge	$n(1 + \mathcal{H}(\mathsf{SUS}))$	$n(1 + \mathcal{H}(\mathsf{LRM}))$
Data comparisons	$2n(1 + \mathcal{H}(\mathsf{SUS}))$	$n(3 + \mathcal{H}(\mathsf{LRM})) - 2$
Time complexity	$\mathcal{O}(n(1 + \mathcal{H}(\mathsf{SUS})))$	$\mathcal{O}(n(1 + \mathcal{H}(\mathsf{LRM})))$

Comparing the $2n(1+\mathcal{H}(\mathsf{SUS}))$ comparisons performed by SUS-Sorting with the $n(3 + \mathcal{H}(\mathsf{LRM})) - 2$ comparisons performed by LRM-Sorting shows that LRM-Sorting performs fewer comparisons on instances where $\mathcal{H}(\mathsf{LRM}) < 2\mathcal{H}(\mathsf{SUS}) - 1 + 2/n$. □

We show in the next section that this new sorting algorithm yields another compressed data structure for permutations.

5 Compressing Permutations

As shown by Barbay and Navarro [3], sorting opportunistically in the comparison model yields a compression scheme for permutations and, with some more work, also yields a compressed data structure giving access to the direct and inverse permutations in time proportional to the entropy of the vector formed by the lengths of the runs (of whatever kind) of the permutation. We show that the sorting algorithm of Theorem 4 corresponds to a compressed data structure for

permutations that supports direct and inverse access in time proportional to the LRM-entropy (Definition 3 page 15), while often using less space than previous solutions.

The essential component of our solution is a data structure for encoding an LRM-partition P. In order to apply Lemma 3, our data structure must efficiently support two operators:

- the operator $\mathtt{map}(i)$ that indicates, for each position $i \in [1..n]$ in the input permutation π, the corresponding subsequence s of P, and the relative position p of i in this subsequence; and
- the operator $\mathtt{unmap}(s,p)$ that is the reverse of $\mathtt{map}()$: given a subsequence $s \in [1..\rho]$ of P and a position $p \in [1..n_s]$ in s, it indicates the corresponding position i in π.

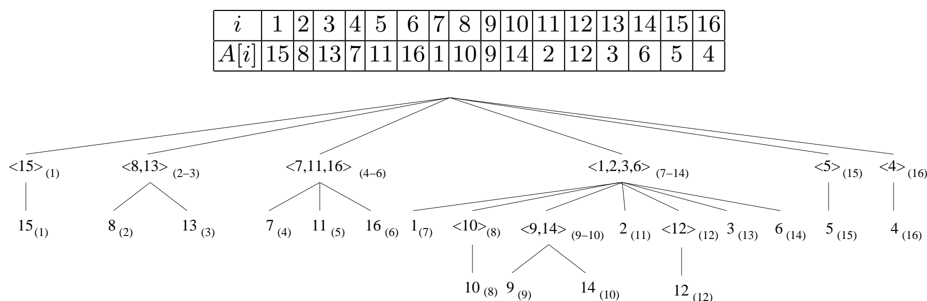
We obviously cannot afford to rewrite the numbers of π in the order described by the partition, which would use $\lg(n!) \geq n \lg n - n \lg e + \mathcal{O}(\lg n)$ bits. A naive solution would be to encode this partition as a string S over alphabet $[1..\rho]$, using a succinct data structure supporting the `access`, `rank` and `select` operators on S, as done for a general SUS-partition [3]. This solution is not suitable as it would require at the very least $n\mathcal{H}(\mathsf{LRM})$ bits *only to encode the LRM-partition*, plus other $n\mathcal{H}(\mathsf{LRM})$ bits to encode the merging tree. This would make this encoding worse than the optimal one for SUS-partition, which requires $2n\mathcal{H}(\mathsf{SUS})$ plus lower-order terms bits [3, 4], given that $\mathcal{H}(\mathsf{SUS}) \leq \mathcal{H}(\mathsf{LRM})$.[1]

We describe a more complex data structure that uses much less space, and that supports the desired operators in constant time.

Lemma 9. *Let P be an LRM-partition consisting of ρ subsequences of respective lengths given by the vector* LRM*, adding up to n. Then there is a data structure using $2\rho \lg n + \mathcal{O}(\rho) + o(n)$ bits that can be computed in time $\mathcal{O}(n)$ and supports the operators `map` and `unmap` on P in constant time and with zero accesses to the original data.*

Proof. The main idea of the data structure is that the subsequences of an LRM-partition P for a permutation π are not as general as, say, the subsequences of a partition into $|\mathsf{SUS}|$ up-sequences. For each pair of subsequences (u, v), either the positions of u and v belong to disjoint intervals of π, or the values corresponding to u (resp. v) all fall between two values from v (resp. u). To see this, assume position z is a child of position x in path u, and that y is a position of path v, so that y is between positions x and z, that is, $x < y < z$. Then, by the time y was processed by the LRM-Tree construction algorithm (Lemma 1), x must have been on the rightmost path and y must have been made a child of x or of a descendant of x, as otherwise z could not have been connected to x later. However, once z was added to the rightmost path as a child of x, y was not anymore on the rightmost path, so any child (and descendant) of y must be at positions between y and z.

[1] Assuming that we have spent all the necessary time to find the optimal SUS-partitioning.



i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$A[i]$	15	8	13	7	11	16	1	10	9	14	2	12	3	6	5	4

Fig. 7. The forest (with a fake root added) corresponding to the LRM-partition $\{(1, 2, 3, 6), (5), (4), (12), (9, 14), (10), (7, 11, 16), (8, 13), (15)\}$ of the permutation of Figure 1 (repeated on top). The smaller numbers below the nodes are the ranges of positions covered by each subsequence.

As a result, the subsequences in P can be organized into a forest of ordinal trees, where

– the internal nodes of the trees of this forest correspond to the ρ subsequences of P, organized so that node v is a descendant of node u if and only if the positions of the subsequence corresponding to v are contained between two positions of the subsequence corresponding to u;
– the leaves of the trees correspond to the n positions in π, and are children of the internal node u corresponding to the subsequence they belong to; and
– the children of a node are ordered in the same order as their corresponding subsequences (if they are internal nodes) or elements (if they are leaves) in the permutation.

It is easy to see that these conditions define a forest: An internal node v cannot have two parents u and u', as its subsequence would be inside two positions of u and of u', thus subsequences u and u' cannot be disjoint and hence, by the above property, one must descend from the other. We will use the balanced parentheses representation of this forest to implement operations `map` and `unmap`. Figure 7 shows the forest corresponding to the permutation of Figure 1.

Given a position $i \in [1..n]$ in π, the corresponding subsequence s of P is simply obtained by finding the parent of the i-th leaf, and returning its preorder rank among internal nodes. The relative position p of i in this subsequence is given by the number of its left siblings that are leaves. Conversely, given the rank $s \in [1..\rho]$ of a subsequence in P and a position $p \in [1..n_s]$ in this subsequence, the corresponding position i in π is computed by finding the s-th internal node in preorder, selecting its p-th child that is a leaf, and computing the preorder rank of this node among all the leaves of the tree.

We represent our forest, with the fake root added, using the structure of Jansson et al. [21]. The only required operation that this representation does not support is counting the number of leaf siblings to the left of a node, and

finding the p-th leaf child of a node. We show next how to extend the structure to support these.

Jansson et al.'s structure [21] encodes a DFUDS representation [5] of the tree, where the nodes are represented in preorder and each node with d children is represented as d 1s followed by a 0: "$1 \cdots 10$". In our example, this representation yields the bit sequence (where we added spaces between nodes for legibility):

1111110 10 0 110 0 0 1110 0 0 0 11111110 0 10 0 110 0 0 0 10 0 0 0 10 0 10 0.

Therefore, the preorder rank among internal nodes corresponds to the number of occurrences of the substring '10' before the position of the node, and conversely, the s-th internal node in preorder is found by locating the s-th occurrence of '10' in the bit-vector and then seeking the preceding 0. The number of leaves to the left of a position corresponds to the number of substrings '00' to the left of the node, and the i-th leaf is found by locating the position of the i-th occurrence of '00'. Those are simple extensions of `rank` and `select` operations on bitmaps that can be carried out in constant time and $o(n)$ bits of space.

For the operations not yet supported, we set up an additional bitmap, of the same length and aligned to the bit-vector of Jansson et al.'s structure, where we mark with a one each 1-bit (i.e., child) that corresponds to an internal node, and the remaining positions are set to zero. In our example, this yields the bit sequence

0111000 00 0 000 0 0 0000 0 0 0 01101000 0 00 0 000 0 0 0 00 0 0 0 00 0 00 0.

Then the operations are easily carried out using `rank` and `select` on this bit-vector and the one from Jansson et al.'s structure: Once we know that v is the i-th child of node u, and that u's children are at positions x to y in the bit-vectors (all of which is given by Jansson et al.'s structure), the number of left siblings of v that are leaves is the number of 0s in the zone $[x, x+i-2]$. Conversely, the p-th child of u that is a leaf is its i-th child, where i is the relative position of the p-th zero in $[x, y]$.

Since the forest T has n leaves and ρ internal nodes, Jansson et al.'s structure [21] takes space

$$nH^*(T) + \mathcal{O}\left(\frac{n(\lg\lg n)^2}{\lg n}\right) \in nH^*(T) + o(n)$$

bits, where

$$\begin{aligned} nH^*(T) &= \lg \binom{n+\rho}{n, n_1, \ldots, n_{n-1}} \\ &\leq \lg \frac{(n+\rho)!}{n!} \;\leq\; \lg\left((n+\rho)^\rho\right) \\ &= \rho \lg(n+\rho) \;=\; \rho \lg(n(1+\rho/n)) \\ &\in \rho \lg n + \mathcal{O}(\rho^2/n) \;\subset\; \rho \lg n + \mathcal{O}(\rho). \end{aligned}$$

On the other hand, the bitmap that we added is of length $2(n+\rho) \leq 4n$ and has exactly ρ 1s, and thus a compressed representation [32] requires $\rho \lg n + \mathcal{O}(\rho) + o(n)$ bits in total. Adding both we get $2\rho \lg n + \mathcal{O}(\rho) + o(n)$. □

Combining the data structure for LRM-partitions from Lemma 9 with the merging data structure from Lemma 3 yields a compressed data structure for permutations. Note that the index and the data are interwoven in a single data structure (i.e., this encoding is not a succinct index [2]), so we express the complexity of its operators as a single measure (as opposed to previous ones, for which we distinguished data and index complexity).

Theorem 5. *Let π be a permutation of size n, such that it has an optimal LRM-partition of size ρ and entropy $\mathcal{H}(\mathsf{LRM})$. Then there is a compressed data structure using $n\mathcal{H}(\mathsf{LRM}) + \mathcal{O}(\rho \lg n) + o(n)$ bits, supporting the computation of $\pi(i)$ and $\pi^{-1}(i)$ in time $\mathcal{O}(1 + \lg \rho / \lg \lg n)$ in the worst case $\forall i \in [1..n]$, and in time $\mathcal{O}(1 + \mathcal{H}(\mathsf{LRM}) / \lg \lg n)$ on average when i is chosen uniformly at random in $[1..n]$. It can be computed in at most $n(3 + \mathcal{H}(\mathsf{LRM})) - 2$ comparisons in π and total running time of $\mathcal{O}(n(1 + \mathcal{H}(\mathsf{LRM})))$.*

Proof. Lemma 5 yields an optimal LRM-partition for π in $2(n-1)$ data comparisons and linear overall time. Lemma 9 yields a data structure for this LRM-partition using $2\rho \lg n + \mathcal{O}(\rho) + o(n)$ bits, and supporting the `map` and `unmap` operators in constant time. This structure is built in linear time and with no further accesses to the permutation. The merging data structure from Lemma 3 requires $n\mathcal{H}(\mathsf{LRM}) + \mathcal{O}(\rho \lg n) + o(n)$ bits, is built using $n(1 + \mathcal{H}(\mathsf{LRM}))$ data comparisons, and supports the operators $\pi()$ and $\pi^{-1}()$ in the time described, through the additional (constant-time) calls to the operators `map`() and `unmap`(). Adding up, we obtain the result. □

Note that this is a compressed data structure, as we stated, as long as $\rho \lg n \in o(n\mathcal{H}(\mathsf{LRM}))$. Since $n\mathcal{H}(\mathsf{LRM}) = \sum n_i \lg \frac{n}{n_i}$, it suffices that there is one partition of length $n_i \in \omega(1)$ for the condition to hold. If the condition does not hold, on the other hand, we have $n\mathcal{H}(\mathsf{LRM}) \in \Theta(n \lg n)$ and there is no hope to compress the permutation under this model.

On the other hand, if we use a coarser-grained entropy measure, such as $\lg \rho$, then our data structure is a fully compressed one, as long as $\rho \in o(n)$. Again, the data is basically not compressible otherwise.

6 Final Remarks

LRM-Trees have proved to be a useful tool to describe a partitioning into increasing runs that is more general than the simple partitioning into contiguous subsequences, and more tractable than the one allowing any such partitioning. It is natural to ask whether other intermediate points exist, more powerful than (or incomparable with) LRM-Trees, and still manageable (i.e., allowing to efficiently

find the optimum partition). Each such result may yield improved sorting algorithms and compressed data structures for interesting classes of permutations that arise in practice.

When implementing these techniques, one should consider a variant of LRM-Trees, *Roller Coaster Trees* (RC-Trees), which take advantage of permutations formed by the combinations of ascending and descending runs. This approach is simple when considering subsequences of consecutive positions, but it gets more technical when considering the insertion of descending runs, and requires new techniques to adapt our compressed data structure to this new setting. Since finding a minimum-size partitioning into up- and down-sequences when considering general subsequences [24] is NP-complete [22], RC-Sorting seems a much desirable improvement on merging ascending and descending runs, as well as a more practical alternative to a hypothetical exponential-time SMS-Sorting algorithm, in an even stronger way than LRM-Trees improved on Runs-Sorting while staying more practical than SUS-Sorting.

Another perspective is the generalization of our results to the compression of general sequences and functions (our techniques already apply to the indexing and sorting of such multisets with redundant values), taking advantage of the redundancy in a general sequence to sort faster and encode in even less space, in function of both the entropy of the frequencies of the symbols and the entropy of the lengths of the subsequences of an optimal LRM-partition. Finally, studying the integration of those compressed data structures into compressed text indices like suffix arrays [27] is likely to yield interesting results, too, as demonstrated in previous work [3].

References

1. J. Barbay, T. Gagie, G. Navarro, and Y. Nekrich. Alphabet partitioning for compressed rank/select and applications. In O. Cheong, K.-Y. Chwa, and K.Parks, editors, *Proceedings of ISAAC 2010, LNCS*, volume 6507, pages 315–326, 2010.
2. J. Barbay, M. He, J. I. Munro, and S. S. Rao. Succinct indexes for strings, binary relations, and multi-labeled trees. In *Proc. SODA*, pages 680–689. ACM/SIAM, 2007.
3. J. Barbay and G. Navarro. Compressed representations of permutations, and applications. In *Proc. STACS*, pages 111–122. IBFI Schloss Dagstuhl, 2009.
4. J. Barbay and G. Navarro. On compressing permutations and adaptive sorting. *CoRR*, abs/1108.4408, 2011.
5. D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.
6. O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM J. Comput.*, 22(2):221–242, 1993.
7. G. S. Brodal, P. Davoodi, and S. S. Rao. On space efficient two dimensional range minimum data structures. In *Proc. ESA (Part II)*, LNCS 6347, pages 171–182. Springer, 2010.
8. K.-Y. Chen and K.-M. Chao. On the range maximum-sum segment query problem. In *Proc. ISAAC*, LNCS 3341, pages 294–305. Springer, 2004.

9. D. R. Clark. *Compact Pat Trees.* PhD thesis, University of Waterloo, Canada, 1996.
10. M. Crochemore, C. S. Iliopoulos, M. Kubica, M. S. Rahman, and T. Walen. Improved algorithms for the range next value problem and applications. In *Proc. STACS*, pages 205–216. IBFI Schloss Dagstuhl, 2008.
11. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. FOCS*, pages 390–398. IEEE, 2000.
12. J. Fischer. Optimal succinctness for range minimum queries. In *Proc. LATIN*, LNCS 6034, pages 158–169. Springer, 2010.
13. J. Fischer, V. Heun, and H. M. Stühler. Practical entropy bounded schemes for $O(1)$-range minimum queries. In *Proc. DCC*, pages 272–281. IEEE Press, 2008.
14. J. Fischer, V. Mäkinen, and G. Navarro. Faster entropy-bounded compressed suffix trees. *Theor. Comput. Sci.*, 410(51):5354–5364, 2009.
15. H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proc. STOC*, pages 135–143. ACM Press, 1984.
16. A. Gál and P. B. Miltersen. The cell probe complexity of succinct data structures. *Theor. Comput. Sci.*, 379(3):405–417, 2007.
17. A. Golynski. Optimal lower bounds for rank and select indexes. *Theor. Comput. Sci.*, 387(3):348–359, 2007.
18. R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA*, pages 841–850. ACM/SIAM, 2003.
19. D. Huffman. A method for the construction of minimum-redundancy codes. In *Proceedings of the I.R.E.*, volume 40, pages 1090–1101, 1952.
20. G. Jacobson. Space-efficient static trees and graphs. In *Proc. FOCS*, pages 549–554. IEEE Computer Society, 1989.
21. J. Jansson, K. Sadakane, and W.-K. Sung. Ultra-succinct representation of ordered trees. In *Proc. SODA*, pages 575–584. ACM/SIAM, 2007.
22. A. E. Kézdy, H. S. Snevily, and C. Wang. Partitioning permutations into increasing and decreasing subsequences. *J. Comb. Theory Ser. A*, 73(2):353–359, 1996.
23. D. E. Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition).* Addison-Wesley Professional, April 1998.
24. C. Levcopoulos and O. Petersson. Sorting shuffled monotone sequences. *Inf. Comput.*, 112(1):37–50, 1994.
25. V. Mäkinen and G. Navarro. Implicit compression boosting with applications to self-indexing. In *Proc. SPIRE*, LNCS 4726, pages 214–226. Springer, 2007.
26. J. I. Munro. Tables. In *Proc. FSTTCS*, volume 1180 of *LNCS*, pages 37–42. Springer, 1996.
27. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):Article No. 2, 2007.
28. D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proceedings of the 14th International Workshop on Algorithms and Data Structures (WADS)*, volume 1671 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, 2007.
29. O. Petersson and A. Moffat. A framework for adaptive sorting. *Discrete Applied Mathematics*, 59:153–179, 1995.
30. M. Pătraşcu. Succincter. In *Proc. FOCS*, pages 305–313. IEEE Computer Society, 2008.
31. R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. SODA*, pages 233–242. ACM/SIAM, 2002.

32. R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. *ACM Transactions on Algorithms*, 3(4):Art. 43, 2007.
33. K. Sadakane and R. Grossi. Squeezing succinct data structures into entropy bounds. In *Proc. SODA*, pages 1230–1239. ACM/SIAM, 2006.
34. K. Sadakane and G. Navarro. Fully-functional succinct trees. In *Proc. SODA*, pages 134–149. ACM/SIAM, 2010.