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, and to express the relative position of the first element of each block within a previous block. They were 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 succinct indices for range minimum queries on partially sorted arrays; a new adaptive sorting algorithm; and a compressed succinct data structure for permutations supporting direct and inverse application in time inversely proportional to the permutation's compressibility.

1 Introduction

Introduced by Fischer [9] as an indexing data structure which supports *Range Minimum Queries* (RMQs) in constant time with no access to the main data, and by Sadakane and Navarro [26] to support navigation operators on ordinal trees, *Left-to-Right-Minima Trees* (LRM-Trees) 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.

We describe how the use of LRM-Trees and variants yields many other convenient results in the design of data structures and algorithms:

1. We define three compressed succinct indices supporting RMQs, which use less space when the indexed array is partially sorted, improving in those cases on the $2n+o(n)$ bits usual space [9], and on other techniques of compression for RMQs such as taking advantage of repetitions in the input [10].
2. Based on LRM-Trees, we define a new *measure of presortedness* for permutations. It combines some of the advantages of two well-known 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 (similar, yet distinct, to the latter).

* First and third author partially funded by Fondecyt grant 1-110066, Chile; second author supported by a DFG grant (German Research Foundation).

3. Based on this measure, we propose a new sorting algorithm and its adaptive analysis, asymptotically superior to sorting algorithms based on runs [2], and on many instances faster than sorting algorithms based on subsequences [19].
4. We design a compressed succinct data structure for permutations based on this measure, which supports the access operator and its inverse in time inversely proportional to the permutation's presortedness, improving on previous similar results [2].

All our results are in the word RAM model, where it is assumed that we can do arithmetic and logical operations on w-bit wide words in $\mathcal{O}(1)$ time, and $w = \Omega(\lg n)$. In our algorithms and data structures, we distinguish between the work performed in the input (often called "data complexity" in the literature) and the accesses to the internal data structures ("index complexity"). This is important in cases where the input is large and cannot be stored in main memory, whereas the index is potentially small enough to be kept in fast main memory. For instance, in the context of compressed indexes like our RMQ structures, given a fixed limited amount of local memory, this additional precision permits identifying the instances whose compressed index fits in it while the main data does not. 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 index complexity is more desirable.

2 Previous Work and Concepts

2.1 Left-to-Right-Minima Trees

LRM-Trees partition a sequence of values into sorted consecutive 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 [26], and, under the name "2d-Min Heaps," to index integer arrays in order to support range minimum queries on them [9].

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 [9]; Sadakane and Navarro [26]). *For $1 \leq i \leq n$, let $\textsc{psv}_A(i) = \max\{j \in [0..i-1] \ : \ A[j] < A[i]\}$ denote the* previous smaller value *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 $\{0, \dots, n\}$. For $1 \leq i \leq n$, $\textsc{psv}_A(i)$ is the parent node of i. The children of each node are ordered in increasing order from left to right.*

See Fig. 1 for an example of LRM-Trees. Fischer [9] 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 Thms. 4 and 5.

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

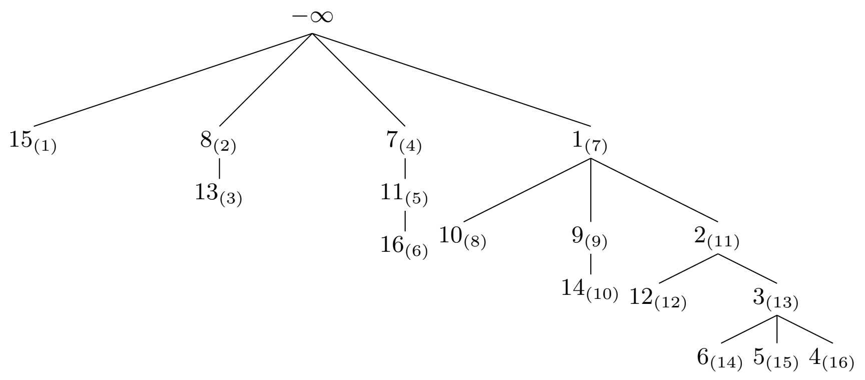

Fig. 1. An example of an array and its LRM-Tree.

Lemma 1. *Given an array $A[1,n]$ of 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 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 (apart from $A[1]$ and $A[n]$) 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 charged only once, as it will never be scanned: hence the total number of comparisons of $2n-2 = 2(n-1)$. Since the number of comparisons within the array dominates the number of other operations, the overall time is also in $\mathcal{O}(n)$. $\square$

2.2 Range Minimum Queries

We consider the following queries on a static array $A[1,n]$ (parameters i and j with $1 \leq i \leq j \leq n$):

Definition 2 (Range Minimum Queries). $\textsc{rmq}_A(i,j) =$ *position of a minimum in* $A[i,j]$.

RMQs have a wide range of applications for various data structures and algorithms, including text indexing [11], pattern matching [7], and more elaborate kinds of range queries [6].

For two given nodes i and j in a tree T, let $\textsc{lca}_T(i,j)$ denote their *Lowest Common Ancestor* (LCA), 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 arbitrary nodes i and j in $\mathcal{T}_A$, $1 \leq i < j \leq n$, let $\ell = \textsc{lca}_{\mathcal{T}_A}(i,j)$. Then if $\ell = i$, $\textsc{rmq}_A(i,j)$ is i, otherwise, $\textsc{rmq}_A(i,j)$ is given by the child of ℓ that is on the path from ℓ to j [9].

Since there are succinct data structures supporting the LCA operator [9,17]. in succinctly encoded trees in constant time, this yields a succinct index (which we improve with Thms. 1 and 3).

Lemma 2 (Fischer [9]). *Given an array $A[1,n]$ of totally ordered objects, there is a 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.*

2.3 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. For a fixed permutation π, Knuth [18] considered *Runs* (contiguous ascending subsequences), counted by $|\mathsf{Runs}| = 1 + |\{i : 1 \leq i < n, \pi_{i+1} < \pi_i\}|$; Levcopoulos and Petersson [19] introduced *Shuffled Up-Sequences* and its generalization *Shuffled Monotone Sequences*, respectively counted by $|\mathsf{SUS}| = \min\{k : \pi \text{ is covered by } k \text{ increasing subsequences}\}$, and $|\mathsf{SMS}| = \min\{k : \pi \text{ is covered by } k \text{ monotone subsequences}\}$. Barbay and Navarro [2] 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 of those five measures of disorder X, there is a variant of the merge-sort algorithm which sorts a permutation π, of size n and of measure of presortedness 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 X (this is not necessarily true for other measures of disorder).

As the merging cost induced by a subsequence is increasing with its length, the sorting time of a permutation can be improved by rebalancing the merging tree [2]. The complexity 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 $\mathcal{H}(\mathsf{Seq}) = \sum_{i=1}^{r} \frac{n_i}{n} \lg \frac{n}{n_i}$. This entropy satisfies $(r-1)\lg n \leq n\mathcal{H}(\mathsf{Seq}) \leq n \lg r$ by concavity of the logarithm, a formula which we will use later.

Barbay and Navarro [2] 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 $n \lg n$ bits. Furthermore they used the similarity of the execution of the merge-sort algorithm with a Wavelet Tree [14], to support the application of $\pi()$ and its inverse $\pi^{-1}()$ in time logarithmic in the disorder of the permutation π (as measured by $|\mathsf{Runs}|$, $|\mathsf{SRuns}|$, $|\mathsf{SUS}|$, $|\mathsf{SSUS}|$ or $|\mathsf{SMS}|$) in the worst case. We summarize their technique in Lemma 3 below, in a way independent of the partition chosen for the permutation, and focusing only on the merging part of the sorting.

Lemma 3 (Barbay and Navarro [2]). *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 $(1 + \mathcal{H}(\mathsf{Seq}))(n + o(n)) + \mathcal{O}(|\mathsf{Seq}| \lg n)$ bits so that it supports the computation of $\pi(i)$ and $\pi^{-1}(i)$ in time $\mathcal{O}(1+\lg |\mathsf{Seq}|)$ in the worst case $\forall i \in [1..n]$, and in time $\mathcal{O}(1 + \mathcal{H}(\mathsf{Seq}))$ on average when i is chosen uniformly at random in $[1..n]$.*

3 Compressed Succinct Indexes 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 sort the elements in A by rank, considering earlier occurrences of equal elements as smaller. Our first and simplest compressed data structure for RMQs uses an amount of space which 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 where $|\mathsf{SRuns}| \in o(n)$, and introduces techniques which we will use in Thms. 2 and 3.

Theorem 1. *Given an array $A[1,n]$ of totally ordered objects, composed of $|\mathsf{SRuns}|$ strict runs, there is a compressed succinct index using $\lceil \lg \binom{n}{|\mathsf{SRuns}|} \rceil + 2|\mathsf{SRuns}| + o(n)$ bits which supports RMQs in zero accesses to A and $\mathcal{O}(1)$ accesses to the index.*

Proof. 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 succinct data structure from Raman et al. [24], using $\lceil \lg \binom{n}{|\mathsf{SRuns}|} \rceil + o(n)$ bits. Further, we define A' as the (conceptual) array consisting of the heads of A's runs ($A'[i] = A[select_1(B,i)]$). We build the LRM-Tree from Lemma 2 on A' using $2|\mathsf{SRuns}|(1+o(1))$ bits. To answer a query

$\text{RMQ}_A(i,j)$, we compute $x = \mathit{rank}_1(B,i)$ and $y = \mathit{rank}_1(B,j)$, 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 $m = \mathit{select}_1(B,m')$. 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. $\square$

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

Theorem 2. *Given an array $A[1,n]$ of totally ordered objects, composed of $|\mathsf{Runs}|$ runs, there is a compressed succinct index using $2|\mathsf{Runs}|+\lceil \lg \binom{n}{|\mathsf{Runs}|}\rceil+o(n)$ bits and supporting RMQs in 1 comparison within A and $\mathcal{O}(1)$ accesses to the index.*

Proof. We build the same data structures as in Thm. 1, using $2|\mathsf{Runs}|+\lceil \lg \binom{n}{|\mathsf{Runs}|}\rceil$ $+o(n)$ bits. To answer a query $\text{RMQ}_A(i,j)$, we compute $x = \mathit{rank}_1(B,i)$ and $y = \mathit{rank}_1(B,j)$. If $x = y$, return i. Otherwise, compute $m' = \text{RMQ}_{A'}(x+1,y)$, and map it back to its position in A by $m = \mathit{select}_1(B,m')$. The final answer is i if $A[i] < A[m]$, and m otherwise. $\square$

To achieve a compressed succinct index which never accesses the array and whose space usage is a function of $|\mathsf{Runs}|$, 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. *Given an array $A[1,n]$ of totally ordered objects, composed of $|\mathsf{Runs}|$ runs, there is a compressed succinct index using $2|\mathsf{Runs}|\lg n + o(n)$ bits, and supporting RMQs in zero accesses to A and $\mathcal{O}(1)$ accesses to the index.*

Proof. We build the LRM-Tree $\mathcal{T}_A$ from Sect. 2.1 directly on A, and then compress it with the tree representation of Jansson et al. [17].

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 $(i_1,j_1),\ldots,(i_{|\mathsf{Runs}|},j_{|\mathsf{Runs}|})$ 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 $n_0 = |\mathsf{Runs}|$, and $n_1 \geq n - |\mathsf{Runs}| - (|\mathsf{Runs}|-1) > n - 2|\mathsf{Runs}|$, as in the worst case the values $\text{PSV}_A(i_y)$ for $i_y \in \{i_2, i_3, \ldots, i_{|\mathsf{Runs}|}\}$ are all different.

As an illustrative example, look again at the tree in Fig. 1. It has $n_0 = 9$ leaves, corresponding to the runs $\langle 15\rangle$, $\langle 8,13\rangle$, $\langle 7,11,16\rangle$, $\langle 1,10\rangle$, $\langle 9,14\rangle$, $\langle 2,12\rangle$, $\langle 3,6\rangle$, $\langle 5\rangle$, and $\langle 4\rangle$ in A. The first four runs have a PSV of $A[0] = -\infty$ for their corresponding head elements, 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 $nH^*(\mathcal{T}_A) + O\left(\frac{n(\lg\lg n)^2}{\lg n}\right)$ bits [17], 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* [17] 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:

$$\binom{n}{n_0, n_1, \ldots, n_{n-1}} = \frac{n!}{n_0! n_1! \ldots n_{n-1}!} \leq \frac{n!}{n_1!} \leq \frac{n!}{(n - 2|\mathsf{Runs}|)!} \leq n^{2|\mathsf{Runs}|} ,$$

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

$$nH^*(\mathcal{T}_A) \leq \lg\left(\frac{1}{n} n^{2|\mathsf{Runs}|}\right) = \lg\left(n^{2|\mathsf{Runs}|-1}\right) = (2|\mathsf{Runs}| - 1)\lg n < 2|\mathsf{Runs}|\lg n .$$

Adding the space required to index the structure of Jansson et al. [17] yields the claimed space bound. □

4 Sorting Permutations

Barbay and Navarro [2] showed how to use the decomposition of a permutation π into $|\mathsf{Runs}|$ ascending consecutive runs of respective lengths Runs to sort adaptively to their 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 [2] by iteratively tagging the leftmost maximal untagged leaf-to-root path of the LRM-Tree. For instance, the permutation of Figure 1 has nine runs ($\langle 15\rangle$, $\langle 8, 13\rangle$, $\langle 7, 11, 16\rangle$, $\langle 1, 10\rangle$, $\langle 9, 14\rangle$, $\langle 2, 12\rangle$, $\langle 3, 6\rangle$, $\langle 5\rangle$, and $\langle 4\rangle$), of respective sizes given by the vector $< 1, 2, 3, 2, 2, 2, 2, 1, 1 >$.

But *any* partition of the LRM-Tree into branches (such that the values traversed by the path 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 $\langle 15\rangle$, $\langle 8, 13\rangle$, $\langle 7, 11, 16\rangle$, $\langle \mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}\rangle$, $\langle 10\rangle$, $\langle 9, 14\rangle$, $\langle 12\rangle$, $\langle 6\rangle$, and $\langle 5\rangle$, of respective sizes given by the vector $< 1, 2, 3, \mathbf{4}, \mathbf{1}, 2, \mathbf{1}, 1, 1 >$.

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* $|\mathsf{LRM}|$ down-paths, *i.e. paths* *starting at some branching node of the tree, and ending at a leaf. The* entropy *of P is $\mathcal{H}(P) = \mathcal{H}(r_1, \ldots, r_{|\mathsf{LRM}|})$, where $r_1, \ldots, r_{|\mathsf{LRM}|}$ 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 this 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 $|\mathsf{Runs}|$ leaves in the LRM-Tree, there will always be $|\mathsf{Runs}|$ down-paths in the LRM-partition; hence $|\mathsf{LRM}| = |\mathsf{Runs}|$. 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 obtained by concatenating all resulting LRM-partitions in arbitrary order.* LRM *denotes the vector formed by the* $|\mathsf{LRM}|$ *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 $-\infty, 1, 2, 3, 6$, which removal leaves a forest of simple branches. The resulting partition is $\langle 15\rangle$, $\langle 8, 13\rangle$, $\langle 7, 11, 16\rangle$, $\langle 1, 2, 3, 6\rangle$, $\langle 10\rangle$, $\langle 9, 14\rangle$, $\langle 12\rangle$, $\langle 5\rangle$, and $\langle 4\rangle$, of respective sizes given by the vector $< 1, 2, 3, 4, 1, 2, 1, 1, 1 >$.

The LRM-partition, by successively extracting increasing subsequences of maximal length, actually yields a partition of minimal entropy, as shown in the following lemma.

Lemma 4. *The entropy of the left-most spinal LRM-partition is minimal among all LRM-partitions.*

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 which contains the down-path $(-\infty, L_0)$. Applying this property recursively in the trees produced by removing the nodes of $(-\infty, L_0)$ from $\mathcal{T}$ yields the optimality of the leftmost LRM-partition.

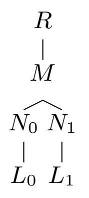

Fig. 2. Consider an arbitrary LRM-partition P and the down-path (N_0, L_0) in P finishing at L_0. If $N_0 \neq -\infty$ (that is, N_0 is not the root), then consider the parent M of N_0 and the down-path (R, L_1) which contains M and finishes at a leaf L_1. Call N_1 the child of M on the path to L_1.

Consider an arbitrary LRM-partition P and the nodes R, M, N_0, N_1 and L_1 as described in Figure 2. Call r the number of nodes in (R, M), d_0 the number of nodes in (N_0, L_0), and d_1 the number of nodes in (N_1, L_1). Note that $d_1 \leq d_0$ because L_0 is one of the deepest leaves. Thus the LRM-partition P has a down-path (N_0, L_0) of length d_0 and another (R, 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 goes from R to L_0 and the other from N_1 to L_1, with new down-path lengths $r + d_0$ and d_1, respectively.

Let $\langle n_1, n_2, \ldots, n_{|\mathsf{LRM}|} \rangle$ be the down-path lengths in P, such that $\mathcal{H}(P) = \mathcal{H}(n_1, n_2, \ldots, n_{|\mathsf{LRM}|}) = n \lg n - \sum_{i=1}^{|\mathsf{LRM}|} n_i \lg n_i$. 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 in entropy 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 $f(x) = (r+x) \lg(r+x) - x \lg x$ has 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 which contains the down-path $(-\infty, L_0)$.

Applying this argument to an optimal LRM-partition demonstrates that there is always an LRM-partition which contains the down-path $(-\infty, L_0)$. This, in turn, applied recursively to the subtrees obtained by removing the nodes from the path $(-\infty, L_0)$ from $\mathcal{T}$, shows the minimality of the entropy of the left-most spinal LRM-partition. □

While the definition of the left-most spinal LRM-partition is constructive, building this partition in linear time requires some sophistication, described in the following lemma:

Lemma 5. *Given an LRM-Tree $\mathcal{T}$, there is an algorithm which computes its left-most spinal LRM-partition in linear overall time (without accessing 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 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. $\square$

Note that the left-most spinal LRM-partition is not much more expensive to compute than the partition into ascending consecutive runs [2]: at most $2(n-1)$ comparisons between elements of the array for the LRM-partition instead of $n-1$ for the Runs-Partition. Note also 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 many. The concept of LRM-partitions yields a new adaptive sorting algorithm:

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 optimal LRM-partition P composed of runs of respective lengths LRM through Lemma 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: it merges the subsequences of P in $n(1+\mathcal{H}(\mathsf{LRM}))$ additional comparisons between elements of π and $\mathcal{O}(|\mathsf{LRM}| \lg |\mathsf{LRM}|)$ additional internal operations. The sum of those complexities yields $n(3+\mathcal{H}(\mathsf{LRM}))-2$ data comparisons, and since $|\mathsf{LRM}| \lg |\mathsf{LRM}| < n\mathcal{H}(\mathsf{LRM}) + \lg |\mathsf{LRM}|$ by concavity of the logarithm, the total time complexity is in $\mathcal{O}(n(1+\mathcal{H}(\mathsf{LRM})))$. $\square$

On instances where $\mathcal{H}(\mathsf{LRM}) = \mathcal{H}(\mathsf{Runs})$, LRM-Sorting can actually perform $n-1$ more comparisons than Runs-Sorting, due to the cost of the construction of the LRM-Tree. Yet, the entropy of the LRM-partition is never larger than the entropy of the Runs partition ($\mathcal{H}(\mathsf{LRM}) \leq \mathcal{H}(\mathsf{Runs})$), which ensures that LRM-sorting's asymptotical performance is never worse than Runs-sorting's performance [2]. Furthermore, LRM-Sorting is arbitrarily faster than Runs-Sorting on permutations with few consecutive inversions, as the lower entropy of the LRM-partition more than compensates for the additional cost of computing the LRM-Tree. For instance, for $n > 2$ odd and $\pi = 1, 3, 2, 5, 4, \ldots, 2i+1, 2i, \ldots, n, n-1$, $|\mathsf{Runs}| = |\mathsf{LRM}| = n/2$, $\mathsf{Runs} = \langle 2, \ldots, 2\rangle$ and $\mathsf{LRM} = \langle n/2+1, 1, \ldots, 1\rangle$, so that the entropy of LRM is arbitrarily smaller than the one of Runs.

When $\mathcal{H}(\mathsf{LRM})$ is much larger than $\mathcal{H}(\mathsf{SUS})$, the merging of the LRM-partition can actually require many more comparisons than the merging of the SUS partition produced by Levcopoulos and Petersson's algorithm [19]. For instance, for $n > 2$ even and $\pi = \mathit{1}, n/2+1, \mathit{2}, n/2+2, \ldots, \mathit{n/2}, n$, $|\mathsf{LRM}| = |\mathsf{Runs}| = n/2$ and $\mathcal{H}(\mathsf{LRM}) = \lg \frac{n}{2}$, whereas $|\mathsf{SUS}| = 2$ and $\mathcal{H}(\mathsf{SUS}) = \lg 2$.

Yet, the high cost of computing the SUS partition (up to $n(1+\mathcal{H}(\mathsf{SUS}))$ additional comparisons within the array, as opposed to only $2(n-1)$ for the LRM-partition) means that on instances where $\mathcal{H}(\mathsf{LRM}) \in [\mathcal{H}(\mathsf{SUS}), 2\mathcal{H}(\mathsf{SUS})-1]$, LRM-Sorting actually performs *fewer* comparisons within the array than SUS-Sorting (if only potentially half, given that $\mathcal{H}(\mathsf{SUS}) \leq \mathcal{H}(\mathsf{LRM})$). Consider for instance, for $n > 2$ multiple of 3, $\pi = 1, 2, n, 3, 4, n-1, 5, 6, n-2, \ldots 2n/3+1$:

there $\mathsf{LRM} = \mathsf{SUS} = \langle 2n/3+1, 1, \ldots, 1\rangle$, so that LRM and SUS have the same entropy, and LRM-sorting outperforms SUS-sorting. A similar reasoning applies to the comparison of the worst-case performances of LRM-Sorting and SMS-Sorting.

Another major advantage of LRM-Sorting over SUS and SMS sorting is that the optimal partition can be computed in linear time, whereas no such linear time algorithm is known to compute the partition of minimal entropy of π into Shuffled Up-Sequences or Shuffled Monotone Sequences; the notation $\mathcal{H}(\mathsf{SUS})$ is defined only as the entropy of the partition of π produced by Levcopoulos and Petersson's algorithm [19], which only promises the smallest number of Shuffled Up-Sequences [2].

LRM-Sorting generally improves on both Runs-Sorting and SUS-Sorting in the number of comparisons performed within the input array. As mentioned in the Introduction, this is important in cases where the internal data structures used by the algorithm do fit in main memory, but not the input itself. Furthermore, we show in the next section that this difference in performance implies an even more meaningful difference in the size of the compressed data structures for permutations corresponding to those sorting algorithms.

5 Compressing Permutations

As shown by Barbay and Navarro [2], sorting opportunistically in the comparison model yields a compression scheme for permutations, and with some more work a compressed succinct data structure supporting the direct and inverse operators in time logarithmic on the disorder of the permutation. We show that the sorting algorithm of Thm. 4 corresponds to a compressed succinct data structure for permutations which supports the direct and reverse operators in time logarithmic on its LRM-Entropy (defined in the previous section), 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 $map(i)$ 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;
- the operator $unmap(s,p)$ is the reverse of $map()$: given a subsequence $s \in [1..|\mathsf{LRM}|]$ 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 $n \lg n$ bits. A naive solution would be to encode this partition as a string S over alphabet $[1..|\mathsf{LRM}|]$, using a succinct data structure supporting the *access*, *rank* and *select* operators on it. This solution is not suitable as it would require at the very least $n\mathcal{H}(\mathsf{Runs})$ bits *only to encode the LRM-partition*, making this encoding worse than the $|\mathsf{Runs}|$ compressed succinct data structure [2]. We describe a more complex data structure which uses less space, and which supports the desired operators in constant time.

Lemma 6. *Let P be an LRM-partition consisting of $|\mathsf{LRM}|$ subsequences of respective lengths given by the vector LRM, summing to n. There is a succinct data structure using $2|\mathsf{LRM}|\lg n + \mathcal{O}(|\mathsf{LRM}|) + o(n)$ bits which supports the operators map and unmap on P in constant time (without accessing the original array).*

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).

As such, the subsequences in P can be organized into a forest of ordinal trees, where (1) the internal nodes of the trees correspond to the $|\mathsf{LRM}|$ subsequences of P, organized so that the node u is the parent of the node v if the positions of the subsequence corresponding to v are contained between two positions of the subsequence corresponding to u, (2) the children of a node are ordered in the same order as their corresponding subsequences in the permutation, and (3) the leaves of the trees correspond to the n positions in π, children of the internal node u corresponding to the subsequence they belong to.

For instance in Figure 3, the permutation $\pi = (4, 5, 9, 6, 8, 1, 3, 7, 2)$ has the LRM-partition $\langle 4, 5, 6, 8\rangle, \langle 9\rangle, \langle 1, 3, 7\rangle, \langle 2\rangle$, whose encoding can be visualized by the expression (45(9)68)(137)(2) and encoded by the balanced parenthesis expression (()()(())()())(()()())(()) (note that this is a forest, not a tree, hence the excess of '('s versus ')'s is going to zero several times inside the expression).

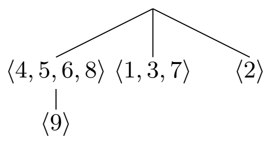

Fig. 3. Given a permutation $\pi = (4, 5, 9, 6, 8, 1, 3, 7, 2)$, its LRM-partition $\langle 4, 5, 6, 8\rangle, \langle 9\rangle, \langle 1, 3, 7\rangle, \langle 2\rangle$ can be visualized by the expression (45(9)68)(137)(2) and encoded as a forest.

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 which are leaves. Conversely, given the rank $s \in [1..|\mathsf{LRM}|]$ 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 which is a leaf, and computing the preorder rank of this node among all the leaves of the tree.

We represent such a forest using the structure of Jansson et al. [17] by adding a fake root node to the forest. The only operation it 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. Jansson et al.'s structure [17] encodes a DFUDS representation [4] of the tree, where each node with d children is represented as d opening parentheses followed by a closing parenthesis: "$(\cdots())$". Thus we set up an additional bitmap, of the same length and aligned to the parentheses string of Jansson et al.'s structure, where we mark with a one each opening parenthesis that corresponds to an internal node (the remaining parentheses, opening or closing, are set to zero). Then the operations are easily carried out using *rank* and *select* on this bitmap and the one from Jansson et al.'s structure.

Since the forest has n leaves and $|\mathsf{LRM}|$ internal nodes, Jansson et al.'s structure [17] takes space $H^*+o(n)$ bits, where $H^* = \lg \binom{n+|\mathsf{LRM}|}{n,n_1,\ldots,n_{n-1}} \leq \lg \frac{(n+|\mathsf{LRM}|)!}{n!} \leq \lg\left((n+|\mathsf{LRM}|)^{|\mathsf{LRM}|}\right) = |\mathsf{LRM}|\lg(n+|\mathsf{LRM}|) = |\mathsf{LRM}|\lg n + \mathcal{O}(|\mathsf{LRM}|)$. On the other hand, the bitmap that we added is of length $2(n+|\mathsf{LRM}|) \leq 4n$ and has exactly $|\mathsf{LRM}|$ 1s, and thus a compressed representation [24] requires $|\mathsf{LRM}|\lg n + \mathcal{O}(|\mathsf{LRM}|) + o(n)$ additional bits. $\square$

Given the data structure for LRM-partitions from Lemma 6, and applying the merging data structure from Lemma 3 immediately yields a compressed succinct 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 [1]), 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 $|\mathsf{LRM}|$ and entropy $\mathcal{H}(\mathsf{LRM})$. There is a compressed succinct data structure using $(1+\mathcal{H}(\mathsf{LRM}))(n+o(n)) + \mathcal{O}(|\mathsf{LRM}|\lg n)$ bits, supporting the computation of $\pi(i)$ and $\pi^{-1}(i)$ in time $\mathcal{O}(1+\lg|\mathsf{LRM}|)$ in the worst case $\forall i \in [1..n]$, and in time $\mathcal{O}(1+\mathcal{H}(\mathsf{LRM}))$ 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 6 yields a data structure for an optimal LRM-partition of π using $2|\mathsf{LRM}|\lg n + \mathcal{O}(|\mathsf{LRM}|) + o(n)$ bits, and supports the *map* and *unmap* operators in constant time. The merging data structure from Lemma 3 requires $(1+\mathcal{H}(\mathsf{LRM}))(n+o(n)) + \mathcal{O}(|\mathsf{LRM}|\lg n)$ bits, and supports the operators $\pi()$ and $\pi^{-1}()$ in the time described, through the additional calls to the operators *map*() and *unmap*(). The latter space is asymptotically dominant. $\square$

References

1. 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.
2. J. Barbay and G. Navarro. Compressed representations of permutations, and applications. In *Proc. STACS*, pages 111–122. IBFI Schloss Dagstuhl, 2009.
3. M. A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *J. Algorithms*, 57(2):75–94, 2005.

4. 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.
5. G. S. Brodal, P. Davoodi, and S. S. Rao. On space efficient two dimensional range minimum data structures. In *Proc. ESA (Part II)*, volume 6347 of *LNCS*, pages 171–182. Springer, 2010.
6. K.-Y. Chen and K.-M. Chao. On the range maximum-sum segment query problem. In *Proc. ISAAC*, volume 3341 of *LNCS*, pages 294–305. Springer, 2004.
7. 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.
8. C. Daskalakis, R. M. Karp, E. Mossel, S. Riesenfeld, and E. Verbin. Sorting and selection in posets. In *Proc. SODA*, pages 392–401. ACM/SIAM, 2009.
9. J. Fischer. Optimal succinctness for range minimum queries. In *Proc. LATIN*, volume 6034 of *LNCS*, pages 158–169. Springer, 2010.
10. 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.
11. J. Fischer, V. Mäkinen, and G. Navarro. Faster entropy-bounded compressed suffix trees. *Theor. Comput. Sci.*, 410(51):5354–5364, 2009.
12. A. Gál and P. B. Miltersen. The cell probe complexity of succinct data structures. *Theor. Comput. Sci.*, 379(3):405–417, 2007.
13. A. Golynski. Optimal lower bounds for rank and select indexes. *Theor. Comput. Sci.*, 387(3):348–359, 2007.
14. R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA*, pages 841–850. ACM/SIAM, 2003.
15. D. Huffman. A method for the construction of minimum-redundancy codes. In *Proceedings of the I.R.E.*, volume 40, pages 1090–1101, 1952.
16. G. Jacobson. Space-efficient static trees and graphs. In *Proc. FOCS*, pages 549–554. IEEE Computer Society, 1989.
17. J. Jansson, K. Sadakane, and W.-K. Sung. Ultra-succinct representation of ordered trees. In *Proc. SODA*, pages 575–584. ACM/SIAM, 2007.
18. D. E. Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)*. Addison-Wesley Professional, April 1998.
19. C. Levcopoulos and O. Petersson. Sorting shuffled monotone sequences. *Inf. Comput.*, 112(1):37–50, 1994.
20. V. Mäkinen and G. Navarro. Implicit compression boosting with applications to self-indexing. In *Proc. SPIRE*, LNCS 4726, pages 214–226. Springer, 2007.
21. J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Succinct representations of permutations. In *Proc. ICALP*, volume 2719 of *LNCS*, pages 345–356. Springer, 2003.
22. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):Article No. 2, 2007.
23. M. Pătraşcu. Succincter. In *Proc. FOCS*, pages 305–313. IEEE Computer Society, 2008.
24. 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.
25. K. Sadakane and R. Grossi. Squeezing succinct data structures into entropy bounds. In *Proc. SODA*, pages 1230–1239. ACM/SIAM, 2006.
26. K. Sadakane and G. Navarro. Fully-functional succinct trees. In *Proc. SODA*, pages 134–149. ACM/SIAM, 2010.