From Time to Space: Fast Algorithms that yield Small and Fast Data Structures

Jérémy Barbay

Departamento de Ciencias de la Computación (DCC),
Universidad de Chile, Santiago, Chile.
jbarbay@dcc.uchile.cl

Abstract. In many cases, the relation between encoding space and execution time translates into combinatorial lower bounds on the computational complexity of algorithms in the comparison or external memory models. We describe a few cases which illustrate this relation in a distinct direction, where fast algorithms inspire compressed encodings or data structures. In particular, we describe the relation between searching in an ordered array and encoding integers; merging sets and encoding a sequence of symbols; and sorting and compressing permutations.

Keywords: Adaptive (Analysis of) Algorithms, Compressed Data Structures, Permutation, Set Union, Sorting, Unbounded Search.

1 Introduction

The worst-case analysis of algorithmic constructs is the theoretical equivalent of a grumpy and bitter fellow who always predicts the worst outcome possible for any actions you might take, however far-fetched the prediction.

Consider for instance the case of data compression. In a modern digital camera, one does not assume that each picture of width w and height h measured in pixels of b bytes will require $h \times w \times b$ bytes of storage space. Rather, the design assumes that the pictures taken have some regularities (e.g. many pixels of similar colors grouped together), which permit to encode it in less space. Only a truly random set of pixels will require $h \times w \times b$ bytes of space, and those are usually not meaningful in the usage of a camera.

Similar to compression techniques, some opportunistic algorithms perform faster than the worst case on some instances. As a simple example, consider the sequential search algorithm on a sorted array of n elements, which performs n comparisons in the worst case but much less in other cases. More sophisticated search algorithms [9] uses $\mathcal{O}(\log n)$ comparisons in the worst case, yet much less in many particular cases. The study and comparison of the performance of such algorithms requires finer analysis techniques than the worst case among instances of fixed sizes. An analysis technique reinvented several times (under names such as *parameterized complexity*, *output-sensitive complexity*, *adaptive* (analysis of) *algorithms*, *distribution-sensitive algorithms* and others) is to study the worst-case performance among finer classes of instances, defined not only by a bound on their size but also by bounds on some defined measure of their *difficulty*.

This concept of opportunism, in common between encodings and algorithms, has yielded some dual analysis, where any fine-analysis of an encoding scheme implies a similar analysis of an algorithm, and vice-versa. For instance, already in 1976 Bentley and Yao [9] described *adaptive algorithms* inspired by Elias codes [13] for searching in a (potentially unbounded) sorted array. More recently in 2009, Laber and Avila [1] showed that any comparison-based merging algorithm can be adapted into a compression scheme for bit vectors. In particular, they discovered a relation between previous independent results on the distinct problems of comparison-based merging and compression schemes for bit vectors: the `Binary Merging` algorithm proposed by Hwang and Lin [24] is closely related to a runlength-based coder using `Rice coding` [35], while `Recursive Merging` is closely related to a runlength-based coder using the `Binary Interpolative Coder`. The very same year, Barbay and Navarro [7], observing that any comparison-based sorting algorithm yields an encoding for permutations, exploited the similarity between the *Wavelet Tree* [22], a data structure used for compression, and the execution of an adaptive variant of `Merge Sort`, to develop and analyze various *compressed data structures* for permutations and *adaptive sorting algorithms*. In this case, the construction algorithm of each data structure is in fact an adaptive algorithm sorting the permutation, in a number of comparisons roughly the same as the number of bits used by the data structure. Given those results, the following questions naturally arise:

1. All comparison-based adaptive sorting algorithms yield compressed encodings. One can wonder if all comparison-based adaptive sorting algorithm, and not only those based on `MergeSort`, can inspire compressed data structures, and if a similar relationship exists for all comparison-based algorithms on other problems than sorting. **Which adaptive algorithms yield compressed data structures?**
2. In the cases of permutations [7], techniques from data compression inspired improvements on the design of algorithms and the analysis of their performance. **Are there cases where such relations between compressed data structures and adaptive algorithms are bijective?**
3. Ignoring the relation between comparison-based merging algorithms and compressed encodings of bit vectors led to the independent discovery of similar techniques [1]. **Could a better understanding of such relations simplify future research?**

This survey aims to be a first step toward answering those questions, by reviewing some adaptive techniques (Section 2), some related data structures (Section 3), and various relations between them in the comparison model, such as between sorted search algorithms and encodings for sequences of integers (Section 4.1), merging algorithms and string data structures (Section 4.2), and sorting algorithms and permutation data structures (Section 4.3). We conclude with a selection of open problems in Section 5.

2 Adaptive Analysis

2.1 Sorted Search

A regular implementation of binary search returns the insertion rank r (defined as $r \in [1..n]$ such that $r = 1$ and $x \leq A[1]$ or $r > 1$ and $A[r-1] < x \leq A[r]$) of an element x in a sorted array A of n elements in $\lceil \lg n \rceil + 1 \in \mathcal{O}(\log n)$ comparisons[1] in the worst case and in $\lfloor \lg n \rfloor + 1 \in \mathcal{O}(\log n)$ comparisons in the best case. The performance of sequential search is much more variable: the algorithm will perform $\min(r, n) + 1$ comparisons (between 2 and $n + 1$), which corresponds to a worst-case complexity of $\mathcal{O}(n)$ comparisons. An interesting (if minor) fact is that, whenever the insertion rank of x is less than $\lceil \lg n \rceil + 1$, *linear search outperforms binary search*.

In 1976, inspired by Elias' codes [13] for sequences of integers, Bentley and Yao [9] described a family of algorithms for *unbounded search* in a (potentially infinite) sorted array. Among those, the doubling search algorithm [9] returns the insertion rank after $1 + 2\lfloor \lg r \rfloor$ comparisons. Hence, doubling search outperforms binary search whenever the insertion rank of x is less than $\sqrt{n}$, and never performs worse than twice the complexity of the binary search. It is widely used in practice, whereas its asymptotic worst-case complexity is the same as binary search, both optimal in the comparison model: the traditional worst-case analysis for a fixed value of n fails to distinguish the performance of those algorithms.

2.2 Union of Sorted Sets

A problem where the output size can vary but is not a good measure of difficulty, is the *description* of the *sorted union of sorted sets*: given k sorted sets, describe their union. On the one hand, the sorted union of $A = \{0, 1, 2, 3, 4\}$ and $B = \{5, 6, 7, 8, 9\}$ is easier to describe (all values from A in their original order, followed by all values from B, in their original order) than the union of $C = \{0, 2, 4, 6, 8\}$ and $D = \{1, 3, 5, 7, 9\}$. On the other hand, a deterministic algorithm must *find* this description, which can take much more time than to output it when computing the union of many sets at once.

Carlsson *et al.* [11] defined the adaptive algorithm `Adaptmerge` to compute the union of two sorted sets in adaptive time. This can be used to compute the union of k sets by merging them two by two, but Demaine *et al.* [12] proposed an algorithm whose complexity depends on the minimal encoding size $\mathcal{C}$ of a *certificate*, a set of comparisons required to check the correctness of the output of the algorithm (yielding worst-case complexity $\Theta(\mathcal{C})$) over instances of fixed size n and certificate-encoding-size $\mathcal{C}$). An alternative approach is to consider the non-deterministic complexity [6], the number of steps δ performed by a non deterministic algorithm to solve the instance, or equivalently the minimal number of comparisons of a certificate (yielding worst-case complexity $\Theta(\delta k \log(n/\delta k))$ over instances of fixed size n and certificate-comparison-size δ).

[1] We note $\lg n = \log_2 n$, $\lg^{(k)}(n) =$ the logarithm iterated k times, and $\log n$ when the base do not matter, such as in asymptotic notations.

2.3 Sorting

Sorting an array A of numbers is a basic problem, where the size of the output of an instance is always equal to its input size. Still, some instances are easier than others to sort (e.g. a sorted array, which can be checked/sorted in linear time). Instead of the output size, one can consider the disorder in an array as a measure of the difficulty of sorting this array [10, 28].

There are many ways to measure this disorder: one can consider the number of exchanges required to sort an array; the number of adjacent exchanges required; the number of pairs (i, j) such that $A[i] > A[j]$, but there are many others [33]. For each disorder measure, the logarithm of the number of instances with a fixed size and disorder forms a natural lower bound to the worst-case complexity of any sorting algorithm in the comparison model, as a correct algorithm must at least be able to distinguish all instances. As a consequence, there could be as many optimal algorithms as there are difficulty measures. Instead, one can reduce difficulty measures between themselves, which yields a hierarchy of disorder measures [29].

An algorithm is *adaptive with respect to* a given measure of presortedness M if its running time is a function of both the measure M and the size n of the input list $X = \langle x_1, \ldots, x_n \rangle$, being linear for small values of M and at most polylogarithmic in n for larger ones. Many measures have been defined [29], we list here only a few relevant ones:

- `nSRuns`, the number of *Strict Runs*, subsequences of consecutive positions in the input with a gap between successive values exactly 1, from beginning to end (e.g. $(1, 2, 6, 7, 8, 9, 3, 4, 5)$ is composed of `nSRuns` $= 3$ strict runs);
- `nRuns`, the number of *Runs*, subsequences of consecutive positions in the input with a positive gap between successive values, from beginning to end (e.g. $(1, 2, 6, 7, 8, 9, 3, 4, 5)$ is composed of `nRuns` $= 2$ runs);
- `nSSUS`, the number of *Strict Shuffled Up Sequences*, subsequences in the input with a gap between successive values exactly 1, from beginning to end (e.g. $(1, 5, 2, 6, 3, 8, 4, 9, 7)$ is composed of `nSSUS` $= 4$ strict shuffled up sequences);
- `nSUS`, the number of *Shuffled Up Sequences*, subsequences in the input with a positive gap between successive values, from beginning to end (e.g. $(1, 5, 2, 6, 3, 8, 4, 9, 7)$ is composed of `nSUS` $= 2$ shuffled up sequences);
- `nSMS`, the number of *Shuffled Monotone Sequences*, subsequences in the input with a positive gap between successive values, from beginning to end or from end to beginning (e.g. $(1, 9, 2, 8, 3, 7, 4, 6, 5)$ is composed of `nSMS` $= 2$ shuffled monotone sequences);
- `nInv`, the number $\texttt{nInv}(X) = |\{(i, j) : i < j \wedge x_i > x_j\}|$ of inversions (i.e., pairs in wrong order) in the input (e.g. $(2, 3, 4, 5, 6, 7, 8, 9, 1)$ has `nInv` $= 8$ inversions); and
- `nRem`, the minimum number of elements that must be removed from the input in order to obtain a sorted subsequence (e.g. $(2, 3, 4, 5, 6, 7, 8, 9, 1)$ needs only `nRem` $= 1$ removal to be sorted).

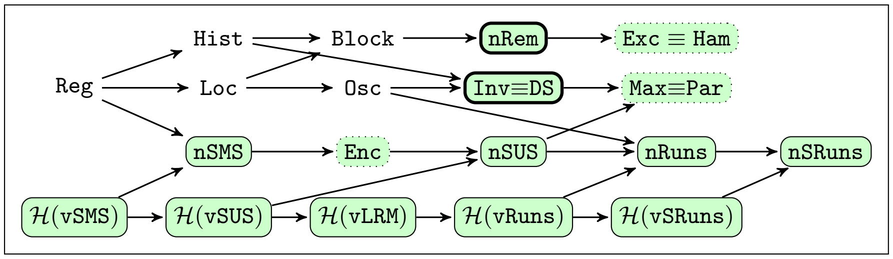

Fig. 1. Partial order on some measures of disorder for adaptive sorting, completed from Moffat and Petersson's survey [29] in 1992. A measure A dominates a measure B ($A \rightarrow B$ or $A \supseteq B$) if superior *asymptotically*. Round lined boxes signal the measures for which a compressed data structure supporting $\pi()$ and $\pi^{-1}()$ in sublinear time is known [3, 4, 7], round dotted boxes signal the results that can be inferred, and bold round boxes signal the additional compressed data structure presented in this article (see Section 4.3).

Moffat and Petersson [33] proposed a framework to compare those measures of presortedness, based on the cost function $C_M(|X|, k)$ representing the minimum number of comparisons to sort a list X such that $M(X) = k$, where M is a measure of presortedness. Given two measures of disorder M_1 and M_2:

- $M_1 \supseteq M_2$, M_1 is *superior* to M_2 if $C_{M_1}(|X|, M_1(X)) = O(C_{M_2}(|X|, M_2(X)))$;
- $M_1 \supset M_2$, M_1 is *strictly superior* to M_2 if $M_1 \supseteq M_2$ and $M_2 \not\subseteq M_1$;
- $M_1 \equiv M_2$, M_1 and M_2 are *equivalent* if $M_1 \supseteq M_2$ and $M_2 \supseteq M_1$;
- M_1 and M_2 are *independent* if $M_1 \not\supseteq M_2$ and $M_2 \not\supseteq M_1$.

This organization yields a partial on those measures and the corresponding algorithms, such as the one given in Figure 1.

3 Encodings and Data Structures

3.1 Integers

In 1975, Elias introduced the concept of *universal* prefix-free codeword set [13], such that given any countable set M of messages and any probability distribution P on M, the codewords in order of increasing length yield a code set of average cost within a constant factor of the source entropy, for any alphabet B of size $|B| \geq 2$. In the binary case ($|B| = 2$), he discussed the properties of existing representation of integers such as unary (code α) and binary (code $\hat{\beta}$), and presented several new techniques of binary representations of integers (γ, γ', δ, and ω), showing that the δ and ω codes are asymptotically optimal universal.

The γ code of an integer i is constructed by inserting after each of the bit of the binary representation $\hat{\beta}(i)$ of i a single bit of the unary representation $\alpha(|\hat{\beta}(i)|)$ of the length $|\hat{\beta}(j)|$ of $\hat{\beta}(j)$. Dropping the first bit (always equal to 1) of $\hat{\beta}(i)$ yields a code of length $1+2\lfloor \lg i \rfloor$ for any positive integer $i > 0$. Reordering the bits of $\gamma(i)$ yields a variant of the code, $\gamma'(i) = \alpha(|\hat{\beta}(i)|).\hat{\beta}(i)$ with the same length which "is easier for people to read" [13] and is usually referred to as the "Gamma Code". Both γ and γ' codes are universal but neither is asymptotically optimal, as the length of the codes stays at a factor of 2 of the optimal value of $\lfloor \lg i \rfloor$ suggested by information theory. The δ and ω codes below reduce this.

The δ code of an integer i is constructed in a similar way to the γ code, only encoding the length $|\hat{\beta}(i)|$ of the binary representation $\hat{\beta}(i)$ of i through the γ code $\gamma(|\hat{\beta}(i)|)$ instead of the unary code $\alpha(|\hat{\beta}(i)|)$, i.e. replacing $\alpha(|\hat{\beta}(i)|)$ by $\gamma(|\hat{\beta}(i)|)$. The δ code $\delta(i)$ of an integer i has length $|\delta(i)| = 1 + \lfloor \lg i \rfloor + 2\lfloor \lg(1 + \lfloor \lg i \rfloor) \rfloor$. This code is, this time, asymptotically optimal as the ratio $1 + \frac{1+2\lfloor \lg(1+\lfloor \lg i \rfloor)\rfloor}{\lg i}$ of its length to the information theory lower bound tends to 1 for large values of i. The same improvement techniques can be applied again in order to improve the convergence rate of this ratio.

The ω code is constructed by concatenating several groups of bits, the rightmost group being the binary representation $\hat{\beta}(i)$ of i, and each other group being the binary encoding $\hat{\beta}(l-1)$ of the length l less one of the following group, the process halting on the left with a group of length 2. Elias points out that there is a code performing even better than the ω code, but only for very large integer values ("much larger than Eddington's estimate of the number of protons and electrons in the universe").

3.2 Sets and Bit Vectors

Jacobson introduced the concept of succinct data structures encoding *Sets* and *bit vectors* [25] within space close to the information lower bounds while supporting efficiently some basic operators on it, as a constructing block for other data-structures, such as tree structures and planar graphs. Given a bit vector $B[1,\dots,n]$ (potentially representing a set $S \subset [1..n]$ such that $\alpha \in S$ if and only if $B[\alpha] = 1$), a bit $\alpha \in \{0,1\}$, a position $x \in [1..n] = \{1,\dots,n\}$ and an integer $r \in \{1,\dots,n\}$, the operator `bin_rank`(B,α,x) returns the number of occurrences of α in $B[1..x]$, and the operator `bin_select`(B,α,r) returns the position of the r-th label α in B, or n if none.

An index of $\frac{n \lg\lg n}{\lg n} + \mathcal{O}(\frac{n}{\log n})$ additional bits [19] support those operators in *constant time* in the $\Theta(\log n)$-word RAM model on a bit vector of length n. This space is asymptotically negligible (i.e. it is within $o(n)$) compared to the n bits required to encode the bit vector itself, in the worst case over all bit vectors of n bits, and is optimal up to asymptotically negligible terms among the encodings keeping the index separated from the encoding of the binary string [19]. As the index is separated from the encoding of the binary string, the result holds even if the binary string has exactly v bits set to one and is compressed to $\lceil \lg \binom{n}{v} \rceil$ bits, as long as the encoding supports in constant time the access to a machine word of the string [34].

3.3 Permutations and Functions

Another basic building block for various data structures is the representation of a *permutation* of the integers $\{1, \dots, n\}$, denoted by $[1..n]$. The basic operators on a permutation are the image of a number i through the permutation, through its inverse or through $\pi^k()$, the k-th power of it (i.e. $\pi()$ iteratively applied k times starting at i, where k can be any integer so that $\pi^{-1}()$ is the inverse of $\pi()$).

A straightforward array of n words encodes a permutation π and supports the application of the operator $\pi()$ in constant time. An additional index composed of n/t shortcuts [18] cutting the largest cycles of π in loops of less than t elements supports the inverse permutation $\pi^{-1}()$ in at most t word-accesses. Using such an encoding for the permutation mapping a permutation π to one of its cyclic representations, one can also support the application of the operator $\pi^k()$, the k times iterated application of operator $\pi()$, in at most t word-accesses (i.e. in time independent of k), with the same space constraints [31]. Those results extend to functions on finite sets [31] by a simple combination with the tree encodings from Jacobson [25].

3.4 Strings

Another basic abstract data type is the *string*, composed of n characters taken from an alphabet of arbitrary size σ (as opposed to binary for the bit vector). The basic operations on a string are to access it, and to search and count the occurrences of a pattern, such as a simple character from $[1..\sigma]$ in the simplest case [22]. Formally, for any character $\alpha \in [1..\sigma]$, position $x \in [1..n]$ in the string and positive integer r, those operations are performed via the operators `string_access`(x), the x-th character of the string; `string_rank`(α, x), the number of occurrences of α before position x; and `string_select`(α, r), the position of the r-th α-occurrence.

Golynski *et al.* [21] showed how to encode a string of length n over the alphabet $[1..\sigma]$ via n/σ permutations over $[1..\sigma]$ and a few bit vectors of length n; and how to support the string operators using the operators on those permutations. Choosing a value of $t = \lg \sigma$ in the encoding of the permutation from Munro *et al.* [31] yields an encoding using $n\big(\lg \sigma + o(\log \sigma)\big)$ bits in order to support the operators in at most $\mathcal{O}(\lg \lg \sigma)$ word accesses. Observing that the encoding of permutations already separates the data from the index, Barbay *et al.* [5] properly separated the data and the index of strings, yielding a succinct index using the same space and supporting the operators in $\mathcal{O}(\lg \lg \sigma \lg \lg \lg \sigma (f(n, \sigma) + \lg \lg \sigma))$ word accesses, if each word of the data can be accessed in $f(n, \sigma)$ word accesses. The space used by the resulting data-structures is optimal up to asymptotically negligible terms, among all possible succinct indexes [20] of fixed alphabet size.

A large body of work has further compressed strings to within entropy limits, culminating (or close) with full independence on the alphabet size from both the redundancy space of the compressed indexes and the time of support of the operators [8].

4 Fast Algorithms that yield Compression Schemes

4.1 From Unbounded Search to Integer Compression

Bentley and Yao [9] mentioned that each comparison-based unbounded search algorithm A implies a corresponding encoding for integers, by memorizing the bit result of each comparison performed by A: simulating A's execution with those bits will yield the same position in the array, hence those bits code the position of the searched element. Their search algorithms are clearly inspired from Elias' codes [13] yet Bentley and Yao do not give explicitly the correspondence between the codes generated by their unbounded search algorithms and the codes from Elias. We remedy this in the following table:

Search Algorithm	cmps and bits	Encoding Scheme
binary	$\lfloor \lg \rfloor n + 1$	binary
sequential	$\lfloor p \rfloor + 1$	unary
B_1 [9]	$2\lfloor \lg p \rfloor + 1$	γ' [13]
B_2 [9]	$2\lfloor \lg \lg p \rfloor + \lfloor \lg p \rfloor + 1$	δ [13]
U [9]	$\sum_i \lfloor \lg^{(i)} p \rfloor + \lfloor \lg^{\lfloor \lg^* p \rfloor} p \rfloor + \lfloor \lg^* p \rfloor + 1$	ω [13]

Each of those codes is readily extendable to a compressed data structure for integers supporting algebraic operations such as the sum, difference, and product in time linear in the sum of the sizes of the compressed encodings of the operands and results. The most advanced codes γ, γ', δ and ω even support, by construction, the extraction of the integer part of the logarithm in base two, in time linear in the sum of the sizes of the compressed encodings of the operands.

4.2 From Merging Algorithms to Set and String Compression

Consider k sorted arrays of integers $A_1, \ldots, A_k$. A *k-Merging Algorithm* computes the sorted union $A = A_1 \cup \ldots \cup A_k$ of those k arrays, with no repetitions.

Ávila and Laber observed that any comparison-based merging algorithm yields an encoding for strings [1]. The transformation is simple: given a string S of n symbols from alphabet $[1..k]$ and a comparison-based merging algorithm M, define the k sorted arrays $A_1, \ldots, A_k$ such that for each value $i \in [1..k]$, A_i is the set of positions in S of symbol i. Running algorithm M on input $(A_1, \ldots, A_k)$ yields the elements from $[1..n]$ in sorted order. Let C be the bit string formed by the sequence of results of each comparison performed by M. Since the execution of M on $(A_1, \ldots, A_k)$ can be simulated via C *without any access to* $(A_1, \ldots, A_k)$, the bit vector C encodes the string S, potentially in less than $n\lceil \lg k \rceil$ bits.

Many merging algorithms perform sublinearly in the size of the input on particular instances: this means that some strings of n symbols from an alphabet $[1..\sigma]$ of size σ are encoded in less than $n\lceil \lg \sigma \rceil$ bits, i.e. that the encoding implied by the merging algorithm is *compressing* the string. Ávila and Laber focused on binary merging algorithms and the compression schemes they implied for bit vectors and sets (i.e. binary sources). They observed that Hwang and

Lin's binary merging algorithm [24] yields an encoding equivalent to using Rice coding [35] in a runlength-based encoder of the string; and that the `Recursive Merging` algorithm [2] yields an encoding equivalent to Moffat and Stuiver's `Binary Interpolative coder` [30]. Furthermore, they note that at least one merging algorithm [15] yields a new encoding scheme (probabilistic in this case) for bit vectors, which was not considered before!

Merging Algorithm	cmps and bits	Source Encoding Scheme
Hwang and Lin [24]	$n\lg(1+\frac{m}{n})$	Rice coding+runlength [35]
`Recursive Merging` [2]	$n\lg(1+\frac{m}{n})$	`Binary Interpolative` [30]
`Probabilistic Binary` [15]	$(m{+}n)\lg(\frac{m}{m+n}){+}0.2355$	Randomized Rice Code [1]

4.3 From Sorting Algorithms to Permutations Data Structures

Barbay and Navarro [7] 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 $\lg(n!) \in n\log n - \frac{n}{\ln 2} + \frac{\log(n)}{2} + \Theta(1)$ bits. Furthermore they used the similarity of the execution of the `Merge Sort` algorithm with a wavelet tree [22], to support the application of the operator $\pi()$ and its inverse $\pi^{-1}()$ in time logarithmic in the disorder of the permutation π (as measured by `nRuns`, `nSRuns`, `nSUS`, `nSSUS` or `nSMS`) in the worst case. We describe below their results and some additional on additional preorder measures.

$\mathcal{H}$(`vRuns`)-Adaptive Sorting and Compression: The simplest way to partition a permutation into sorted chunks is to divide it into *runs* of consecutive positions forming already sorted blocks, in $n-1$ comparisons. For example, the permutation $(\mathit{8}, \mathit{9}, 1, 4, 5, 6, 7, \mathbf{2}, \mathbf{3})$ contains $\texttt{nRuns} = 3$ ascending runs, of lengths forming the vector $\texttt{vRuns} = \langle 2, 5, 2\rangle$.

Using a simple partition of the permutation into *runs*, merging those via a wavelet tree sorts the permutation and yields a data structure compressing a permutation to $n\mathcal{H}(\texttt{vRuns}) + \mathcal{O}(\texttt{nRuns}\log n) + o(n)$ bits in time $\mathcal{O}(n(1 + \mathcal{H}(\texttt{vRuns})))$, which is worst-case optimal in the comparison model. Furthermore, this data structure supports the operators $\pi()$ and $\pi^{-1}()$ in sublinear time $\mathcal{O}(1+\log\texttt{nRuns})$, with the average supporting time $\mathcal{O}(1+\mathcal{H}(\texttt{vRuns}))$ decreasing with the entropy of the partition of the permutation into runs [7].

Strict-Runs-Adaptive Sorting and Compression: A two-level partition of the permutation yields further compression [7]. The first level partitions the permutation into *strict ascending runs* (maximal ranges of positions satisfying $\pi(i+k) = \pi(i) + k$). The second level partitions the *heads* (first position) of those strict runs into conventional ascending runs.

For example, the permutation $\pi = (8, 9, 1, 4, 5, 6, 7, 2, 3)$ has $\texttt{nSRuns} = 4$ strict runs of lengths forming the vector $\texttt{vSRuns} = \langle 2, 1, 4, 2\rangle$. The run heads are

$\langle 8, 1, 4, 2\rangle$, which form 3 monotone runs, of lengths forming the vector $\texttt{vHRuns} = \langle 1, 2, 1\rangle$. The number of strict runs can be anywhere between $\texttt{nRuns}$ and n: for instance the permutation $(\mathit{6}, \mathit{7}, \mathit{8}, \mathit{9}, \mathit{10}, \mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5})$ contains $\texttt{nSRuns} = \texttt{nRuns} = 2$ strict runs while the permutation $(\mathit{1}, \mathit{3}, \mathit{5}, \mathit{7}, \mathit{9}, \mathbf{2}, \mathbf{4}, \mathbf{6}, \mathbf{8}, \mathbf{10})$ contains $\texttt{nSRuns} = 10$ strict runs, each of length 1, and 2 runs, each of length 5.

$\mathcal{H}$(vSUS)-Adaptive Sorting and Compression: The preorder measures seen so far have considered runs which group contiguous positions in π: this does not need to be always the case. A permutation π over $[1..n]$ can be decomposed in n comparisons into a minimal number $\texttt{nSUS}$ of *Shuffled Up Sequences*, defined as a set of, not necessarily consecutive, subsequences of increasing numbers that have to be removed from π in order to reduce it to the empty sequence [26]. Then those subsequences can be merged using the same techniques as above, which yields a new adaptive sorting algorithm and a new compressed data structure [7]. For example, the permutation $(\mathit{1}, \mathbf{6}, \mathit{2}, \mathbf{7}, \mathit{3}, \mathbf{8}, \mathit{4}, \mathbf{9}, \mathit{5}, \mathbf{10})$ contains $\texttt{nSUS} = 2$ shuffled up sequences of lengths forming the vector $\texttt{vSUS} = \langle 5, 5\rangle$, but $\texttt{nRuns} = 5$ runs, all of length 2.

Note that it is a bit more complicated to partition a permutation π over $[1..n]$ into a minimal number $\texttt{nSMS}$ of *Shuffled Monotone Sequences*, sequences of not necessarily consecutive subsequences of increasing or decreasing numbers: an optimal partition is NP-hard to compute [27].

$\mathcal{H}$(vLRM)-Adaptive Sorting and Compression: 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 [36] and, under the name "2d-Min Heaps", to index integer arrays in order to support range minimum queries on them [16]. Such a tree can be computed in $2(n-1)$ comparisons within the array and overall linear time, through an algorithm similar to that of Cartesian Trees [17].

The interest of LRM trees in the context of adaptive sorting and permutation compression is that the values are increasing in each root-to-leaf branch: they form a partition of the array into sub-sequences of increasing values. Barbay *et al.* [4] described how to compute the partition of the LRM-tree of minimal size-vector entropy, which yields a sorting algorithm asymptotically better than $\mathcal{H}(\texttt{vRuns})$-adaptive sorting, and better in practice than $\mathcal{H}(\texttt{vSUS})$-adaptive sorting; as well as a smaller compressed data structure.

nRem-Adaptive Sorting and Compression: The preorder measures described above are all variants of `MergeSort`, exploiting the similarity of its execution with a wavelet tree: they are all situated on the same "branch" of the graph from Figure 1 representing the measures of preorder and their relation.

The preorder measure `nRem` counts how many elements must be removed from a permutation so that what remains is already sorted. Its exact value is n

minus the length of the *Longest Increasing Subsequence*, which can be computed in time $n \lg n$, but in order to adaptively sort in time faster than this, $\texttt{nRem}$ can be approximated within a factor of 2 in n comparisons by an algorithm very similar to the one building a LRM-tree, which returns a partition of π into one part of $2\texttt{nRem}$ unsorted elements, and $n - 2\texttt{nRem}$ elements in increasing order. Sorting those $2\texttt{nRem}$ unsorted elements using any n-worst-case optimal comparison-based algorithm (ideally, one of the adaptive algorithms described above), and merging its result with the $n - 2\texttt{nRem}$ elements found to be already in increasing order, yields an adaptive sorting algorithm that performs $2n + 2\texttt{nRem} \lg(n/\texttt{nRem} + 1)$ comparisons [14, 29]. Similarly, partitioning π into those two parts by a bit vector of n bits; representing the order of the $2\texttt{nRem}$ elements in a wavelet tree (using any of the data structures described above) and representing the merging of both into n bits yields a compressed data structure using $2n + 2\texttt{nRem} \lg(n/\texttt{nRem}) + o(n)$ bits and supporting the operators $\pi()$ and $\pi^{-1}()$ in sublinear time, within $\mathcal{O}(\log(\texttt{nRem} + 1) + 1)$.

$\texttt{nInv}$-Adaptive Sorting and Compression: The preorder measure $\texttt{nInv}$ counts the number of pairs (i, j) of positions $1 \leq i < j \leq n$ in a permutation π over $[1..n]$ such that $\pi(i) > \pi(j)$. Its value is exactly the number of comparisons performed by the algorithm `Insertion Sort`, between n and n^2 for a permutation over $[1..n]$. A variant of `Insertion Sort`, named `Local Insertion Sort`, sorts π in $n(1 + \lg(\texttt{nInv}/n))$ comparisons [14, 29].

As before, the bit vector B listing the binary results of the comparisons performed by `Local Insertion Sort` on a permutation π identifies exactly π, because B is sufficient to simulate the execution of `Local Insertion Sort` on π without access to it. This yields an encoding of π into $n(1 + \lceil \lg(\texttt{nInv}/n) \rceil)$ bits, which is smaller than $n\lceil \lg n \rceil$ bits for permutations such that $\texttt{nInv} \in o(n^2)$. Yet it is not clear how to support the operator $\pi()$ (yet even its inverse $\pi^{-1}()$) on such an encoding without reading all the $n(1 + \lg(\texttt{nInv}/n))$ bits of B: the bits deciding of a single value can be spread in the whole encoding.

But reordering those bits does yield a compressed data structure supporting the operator $\pi()$ in constant time, by simply encoding the n values $(\pi(i)-i)_{i\in[1..n]}$ using the γ' code from Elias [13], and indexing the positions of the beginning of each code by a compressed bit vector. Following the execution of `Linear Insertion Sort` algorithm over a permutation π over $[1..n]$ presenting $\texttt{nInv}$ inversions, the number of swaps of the i-th element $\pi(i)$ required to reach its final position in the sorted list is $\pi(i) = i + g_i^+(\pi) - g_i^-(\pi)$, where $g_i^+(\pi) = |\{j \in [1..n] : \; j > i \text{ and } \pi(j) < \pi(i)\}|$ is the number of swaps to the right; and $g_i^-(\pi) = |\{j \in [1..n] : \; j < i \text{ and } \pi(j) > \pi(i)\}|$ is the number of swaps to the left. By definition of $\texttt{nInv}$, g^+ and g^-, $\sum_{i=1}^n g_i^+(\pi) = \sum_{i=1}^n g_i^-(\pi) = \texttt{nInv}$, and by property of the γ' code, the number of bits used to store the values of $g_i^+(\pi)$ (or $g_i^-(\pi)$) is $\texttt{Gap}(g_i^+(\pi))_{i\in[1..n]} = \sum_{i=1}^n \lg g_i^+(\pi) \leq n \lg\left(\frac{\sum_{i=1}^n g_i^+(\pi)}{n}\right) = n \lg \frac{\texttt{nInv}}{n}$, by concavity of the logarithm. Since $\pi(i) - i = g_i^+(\pi) - g_i^-(\pi)$, the data structure uses $n + \texttt{Gap}((\pi(i)-i)_{i\in[1..n]}) + o(n) \subset n + 2n \lg \frac{\texttt{nInv}}{n} + o(n) = n(1 + 2\lg \frac{\texttt{nInv}}{n}) + o(n)$ bits.

Note that the compressed data structure described so far supports the operator $\pi()$ in constant time, which is faster than the compressed data structure described above, but not the operator $\pi^{-1}()$ (other than in at least linear time, by reconstructing π). By definition of nInv, the inverse permutation π^{-1} has the same number nInv of inversions than π: the data structure for π^{-1} uses the same space as for π. Hence encoding both the permutations π and its inverse π^{-1} as described above yields a data structure using space within $2n(1+2\lg\frac{\texttt{nInv}}{n})+o(n)$ which supports both operators $\pi()$ and $\pi^{-1}()$ in constant time. This space is less than $n\log n + o(n)$ for instance where $\texttt{nInv}[0..n^{\frac{5}{4}}]$, but of course in the worst case where nInv is close to n^2, this space is getting close to $8n\log n + 2n + o(n)$, a solution four times as costly as merely encoding in a raw form both π and its inverse π^{-1}.

Other Adaptive Sorting and Compression: As we observed in Section 2.3, Moffat and Petersson [29] list many other measures of preorder and adaptive sorting techniques. Each measure explored above yields a compressed data structure for permutation supporting the operators $\pi()$ and $\pi^{-1}()$ in sublinear time. Figure 1 shows the relation between those measures, and the table below shows the relation between a selection of adaptive sorting algorithms and some permutation data structure (omitting both the $o()$ order terms in the space and the support time for the operators for sake of space).

Sorting Algorithm	cmps=bits	Permutation Data Structure
Natural MergeSort [23]	$n(1+\lg\texttt{nRuns})$	Runs [7]
BN-MergeSort [7]	$n(1+\mathcal{H}(\texttt{vRuns}))$	Huffman Runs [7]
$\mathcal{H}(\texttt{vSUS})$-Sort [7]	$2n\mathcal{H}(\texttt{vSUS})$	Huffman SUS [7]
$\mathcal{H}(\texttt{SMS})$-Sort [3]	$2n\mathcal{H}(\texttt{vSMS})$	Huffman SMS [7]
Rem-Sort (here)	$2n + 2\texttt{nRem}\lg(n/\texttt{nRem})$	Rem-encoding (here)
Local Ins Sort (here)	$2n(1+2\lg(\texttt{nInv}/n))$	Inv-encoding (here)

Note that all comparison-based adaptive sorting algorithms yield a compression scheme for permutations, but not all yield one for which we know how to support useful operators (such as $\pi()$ and $\pi^{-1}()$) in sublinear time.

5 Selected Open Problems

From Compression Schemes to Compressed Data Structures: In most current applications, compressed data is useless if it needs to be totally decompressed in order to access a small part of it. We saw how to support some operators on compressed data inspired from adaptive algorithms in the cases of permutations, integers and strings, but there are many other operators to study, from the iterated operator $\pi^k()$ on a compressed permutation π, non-algebraic operators on compressed integers, patern matching operators on compressed strings, etc...

From Compression Schemes to Adaptive Algorithms: Bentley and Yao [9] were already asking in 1976 if there are cases where such relations between compressed data structures and adaptive algorithms are bijective. Such a question comes naturally and applies to many other Abstract Data Types than integers or permutations, as both compression schemes and adaptive algorithms take advantage of forms of regularity in the instances considered. If a systematic transformation generating a distinct adaptive algorithm from each distinct compression scheme might not exist, at least one should be able to define a subclass of compression schemes which are in bijection with adaptive algorithms.

Other Compressed Data Structures for Permutations: Each adaptive sorting algorithm in the comparison model yields a compression scheme for permutations, but the encoding thus defined does not necessarily support the simple application of the permutation to a single element without decompressing the whole permutation, nor the application of the inverse permutation. Figure 1 represents the preorder measures for which opportunistic sorting algorithms are known, and in round boxes the ones for which compressed data structures for permutations (supporting in sublinear time the operators $\pi()$ and $\pi^{-1}()$) are known. **Are there compressed data structures for permutation**s, supporting the operators $\pi()$ and $\pi^{-1}()$ in sublinear time and **using space proportional to the other preorder measures?** What about other useful operators on permutations, such as $\pi^k()$?

Sorting and Encoding Multisets: Munro and Spira [32] showed how to sort multisets through `MergeSort`, `Insertion Sort` and `Heap Sort`, adapting them with counters to sort in time $\mathcal{O}(n(1+\mathcal{H}(\langle m_1, \ldots, m_r \rangle)))$ where m_i is the number of occurrences of i in the multiset (note this is totally different from our results, that depend on the distribution of the lengths of monotone runs). It seems easy to combine both approaches (e.g. on `Merge Sort` in a single algorithm using both runs and counters), yet quite hard to *analyze* the complexity of the resulting algorithm. The difficulty measure must depend not only on both the entropy of the partition into runs and the entropy of the repartition of the values of the elements, but also on their interaction.

Bibliography

[1] B. T. Ávila and E. S. Laber. Merge source coding. In *Proceedings of IEEE International Symposium on Information Theory (ISIT)*, pages 214–218. IEEE, 2009.

[2] R. A. Baeza-Yates. A fast set intersection algorithm for sorted sequences. In *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 400–408, 2004.

[3] J. Barbay, F. Claude, T. Gagie, G. Navarro, and Y. Nekrich. Efficient fully-compressed sequence representations. *Algorithmica*, 2013. To appear.

[4] J. Barbay, J. Fischer, and G. Navarro. LRM-trees: Compressed indices, adaptive sorting, and compressed permutations. *ELSEVIER Theoretical Computer Science (TCS)*, 459:26–41, 2012.

[5] J. Barbay, M. He, J. I. Munro, and S. R. Satti. Succinct indexes for strings, binary relations and multilabeled trees. *ACM Transactions on Algorithms*, 7(4):52, 2011.

[6] J. Barbay and C. Kenyon. Deterministic algorithm for the t-threshold set problem. In *Proceedings of the 14th International Symposium Algorithms and Computation (ISAAC)*, pages 575–584, 2003.

[7] J. Barbay and G. Navarro. Compressed representations of permutations, and applications. In *26th International Symposium on Theoretical Aspects of Computer Science (STACS 2009)*, volume 3, pages 111–122, 2009.

[8] D. Belazzougui and G. Navarro. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms (TALG)*, 2013. To appear.

[9] J. L. Bentley and A. C.-C. Yao. An almost optimal algorithm for unbounded searching. *Information processing letters*, 5(3):82–87, 1976.

[10] W. H. Burge. Sorting, trees, and measures of order. *Information and Control*, 1(3):181–197, 1958.

[11] S. Carlsson, C. Levcopoulos, and O. Petersson. Sublinear merging and natural mergesort. *Algorithmica*, 9(6):629–648, 1993.

[12] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the* 11^{th} *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 743–752, 2000.

[13] P. Elias. Universal codeword sets and representations of the integers. *Information Theory, IEEE Transactions on*, 21(2):194–203, 1975.

[14] V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys*, 24(4):441–476, 1992.

[15] W. Fernandez de la Vega, S. Kannan, and M. Santha. Two probabilistic results on merging. In *Proceedings of the international symposium on Algorithms*, SIGAL '90, pages 118–127, 1990.

[16] J. Fischer. Optimal succinctness for range minimum queries. In *Proc. LATIN*, LNCS 6034, pages 158–169. Springer, 2010.

[17] 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.

[18] R. Gennaro and L. Trevisan. Lower bounds on the efficiency of generic cryptographic constructions. In *IEEE Symposium on Foundations of Computer Science*, pages 305–313, 2000.

[19] A. Golynski. Optimal lower bounds for rank and select indexes. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, 2006.

[20] A. Golynski. *Upper and Lower Bounds for Text Indexing Data Structures*. PhD thesis, University of Waterloo, 2007.

[21] A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 368–373. ACM, 2006.

[22] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete algorithms (SODA)*, pages 841–850. ACM, 2003.

[23] J. D. Harris. Sorting unsorted and partially sorted lists using the natural merge sort. *Software: Practice and Experience*, 11(12):1339–1340, 1981.

[24] F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly-ordered sets. *SIAM J. Comput.*, 1(1):31–39, 1972.

[25] G. Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[26] C. Levcopoulos and O. Petersson. Sorting shuffled monotone sequences. In *SWAT '90: Proceedings of the 2nd Scandinavian Workshop on Algorithm Theory*, pages 181–191, London, UK, 1990. Springer-Verlag.

[27] C. Levcopoulos and O. Petersson. Sorting shuffled monotone sequences. *Inf. Comput.*, 112(1):37–50, 1994.

[28] H. Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Trans. Computers*, 34(4):318–325, 1985.

[29] A. Moffat and O. Petersson. An overview of adaptive sorting. *Australian Computer Journal*, 24(2):70–77, 1992.

[30] A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Inf. Retr.*, 3(1):25–47, 2000.

[31] J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Succinct representations of permutations and functions. *Theor. Comput. Sci.*, 438:74–88, 2012.

[32] J. I. Munro and P. M. Spira. Sorting and searching in multisets. *SIAM J. Comput.*, 5(1):1–8, 1976.

[33] O. Petersson and A. Moffat. A framework for adaptive sorting. *Discrete Applied Mathematics*, 59:153–179, 1995.

[34] R. Raman, V. Raman, and S. R. Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4), 2007.

[35] R. F. Rice and J. R. Plaunt. Adaptive variable-length coding for efficient compression of spacecraft television data. *IEEE Trans. Commun.*, COM-19:889–897, Dec. 1971.

[36] K. Sadakane and G. Navarro. Fully-functional succinct trees. In *Proc. SODA*, pages 134–149. ACM/SIAM, 2010.