New Algorithms and Lower Bounds for Sequential-Access Data Compression

Travis Gagie

PhD Candidate
Faculty of Technology
Bielefeld University
Germany

July 2009

Gedruckt auf alterungsbeständigem Papier °° ISO 9706.

Abstract

This thesis concerns sequential-access data compression, i.e., by algorithms that read the input one or more times from beginning to end. In one chapter we consider adaptive prefix coding, for which we must read the input character by character, outputting each character's self-delimiting codeword before reading the next one. We show how to encode and decode each character in constant worst-case time while producing an encoding whose length is worst-case optimal. In another chapter we consider one-pass compression with memory bounded in terms of the alphabet size and context length, and prove a nearly tight tradeoff between the amount of memory we can use and the quality of the compression we can achieve. In a third chapter we consider compression in the read/write streams model, which allows us passes and memory both polylogarithmic in the size of the input. We first show how to achieve universal compression using only one pass over one stream. We then show that one stream is not sufficient for achieving good grammar-based compression. Finally, we show that two streams are necessary and sufficient for achieving entropy-only bounds.

Acknowledgments

First and foremost, I thank my mother, without whom this thesis could not have been written (for the obvious reasons, and many more). I am also very grateful to Giovanni Manzini, who was essential to the research and writing; to Ferdinando Cicalese and Jens Stoye, who were essential to the submission; to Charlie Rackoff, who said "yes, you can go on vacation" and didn't ask "for how long?"; and to all the friends I've had over the years in Toronto, Perugia, Pisa, Alessandria and Bielefeld. This thesis is dedicated to the memory of my father.

Contents

Chapter 1

Introduction

Sequential-access data compression is by no means a new subject, but it remains interesting both for its own sake and for the insight it provides into other problems. Apart from the Data Compression Conference, several conferences often have tracks for compression (e.g., the International Symposium on Information Theory, the Symposium on Combinatorial Pattern Matching and the Symposium on String Processing and Information Retrieval), and papers on compression often appear at conferences on algorithms in general (e.g., the Symposium on Discrete Algorithms and the European Symposium on Algorithms) or even theory in general (e.g., the Symposium on Foundations of Computer Science, the Symposium on Theory of Computing and the International Colloquium on Algorithms, Languages and Programming). We mention these conference in particular because, in this thesis, we concentrate on the theoretical aspects of data compression, leaving practical considerations for later.

Apart from its direct applications, work on compression has inspired the design and analysis of algorithms and data structures, e.g., succinct or compact data structures such as indexes. Work on sequential data compression in particular has inspired the design and analysis of online algorithms and dynamic data structures, e.g., prediction algorithms for paging, web caching and computational finance. Giancarlo, Scaturro and Utro [GSU09] recently described how, in bioinformatics, compression algorithms are important not only for storage, but also for indexing, speeding up some dynamic programs, entropy estimation, segmentation, and pattern discovery.

In this thesis we study three kinds of sequential-access data compression: adaptive prefix coding, one-pass compression with memory bounded in terms of the al-

phabet size and context length, and compression in the read/write streams model. Adaptive prefix coding is perhaps the most natural form of online compression, and adaptive prefix coders are the oldest and simplest kind of sequential compressors, having been studied for more than thirty years. Nevertheless, in Chapter 2 we present the first one that is worst-case optimal with respect to both the length of the encoding it produces and the time it takes to encode and decode. In Chapter 3 we observe that adaptive alphabetic prefix coding is equivalent to online sorting with binary comparisons, so our algorithm from Chapter 2 can easily be turned into an algorithm for online sorting. Chapter 4 is also about online sorting but, instead of aiming to minimize the number of comparisons (which remains within a constant factor of optimal), we concentrate on trying to use sublinear memory, in line with research on streaming algorithms. We then study compression with memory constraints because, although compression is most important when space is in short supply and compression algorithms are often implemented in limited memory, most analyses ignore memory constraints as an implementation detail, creating a gap between theory and practice. We first study compression in the case where we can make only one pass over the data. One-pass compressors that use memory bounded in terms of the alphabet size and context length can be viewed as finite-state machines, and in Chapter 5 we use that property to prove a nearly tight tradeoff between the amount of memory we can use and the quality of the compression we can achieve. We then study compression in the read/write streams model, which allows us to make multiple passes over the data, change them, and even use multiple streams (see [Sch07]). Streaming algorithms have revolutionized the processing of massive data sets, and the read/write streams model is an elegant conversion of the streaming model into a model of external memory. By viewing read/write stream algorithms as simply more powerful automata, which can use passes and memory both polylogarithmic in the size of the input, in Chapter 6 we prove lower bounds on the compression we can achieve with only one stream. Specifically, we show that, although we can achieve universal compression with only one pass over one stream, we need at least two streams to achieve good grammar-based compression or entropy-only bounds. We also combine previously known results to prove that two streams are sufficient for us to compute the Burrows-Wheeler Transform and, thus, achieve low-entropy bounds. As corollaries of our lower bounds for compression, we obtain lower bounds for computing strings' minimum periods and for computing the Burrows-Wheeler

Transform [BW94], which came as something of a surprise to us. It seems no one has previously considered the problem of finding strings' minimum periods in a streaming model, even though some related problems have been studied (see, e.g., [EMŞ04], in which the authors loosen the definition of a repeated substring to allow approximate matches). We are currently investigating whether we can derive any more such results.

The chapters in this thesis were written separately and can be read separately; in fact, it might be better to read them with at least a small pause between chapters, so the variations in the models considered do not become confusing. Of course, to make each chapter independent, we have had to introduce some degree of redundancy. Chapter 2 was written specifically for this thesis, and is based on recent joint work [GN] with Yakov Nekrich at the University of Bonn; a summary was presented at the University of Bielefeld in October of 2008, and will be presented at the annual meeting of the Italy-Israel FIRB project "Pattern Discovery in Discrete Structures, with Applications to Bioinformatics" at the University of Palermo in February of 2009. Chapter 3 was also written specifically for this thesis, but Chapter 4 was presented at the 10th Italian Conference on Theoretical Computer Science [Gag07b] and then published in *Information Processing Letters* [Gag08b]. Chapter 5 is a slight modification of part of a paper [GM07b] written with Giovanni Manzini at the University of Eastern Piedmont, which was presented in 2007 at the 32nd Symposium on Mathematical Foundations of Computer Science. A paper [GKN09] we wrote with Marek Karpinski (also at the University of Bonn) and Yakov Nekrich that partially combines the results in these two chapters, will appear at the 2009 Data Compression Conference. Chapter 6 is a slight modification of a paper [Gag09] that has been submitted to a conference, with a very brief summary of some material from a paper [GM07a] written with Giovanni Manzini and presented at the 18th Symposium on Combinatorial Pattern Matching.

As we noted above, making the chapters in this thesis independent required us to introduce some degree of redundancy. In particular, some terms are defined several times, sometimes with different ideas emphasized in different chapters. For example, although we consider stable sorting in both Chapters 3 and 4, we give a more detailed definition in Chapter 4 because our algorithm there relies heavily on certain properties of the permutation that stably sorts the input; in Chapter 3, stability is less important in the upper bound and we are currently

trying to remove it from the lower bound. Nevertheless, in order to avoid excessive repetition, we now present some core terms and concepts. For a more thorough introduction to the field of data compression, we refer the reader to, e.g., the text by Cover and Thomas [CT06].

1.1 Entropy and empirical entropy

Let X be a random variable that takes on one of σ values according to $P = p_1, \ldots, p_\sigma$. Shannon [Sha48] proposed that any function $H(P)$ measuring our uncertainty about X should have three properties:

1. "H should be continuous in the p_i."
2. "If all the p_i are equal, $p_i = \frac{1}{\sigma}$, then H should be a monotonic increasing function of σ." [Shannon wrote n instead of σ; we have changed his notation for consistency with the rest of this thesis.]
3. "If a choice be broken down into two successive choices, the original H should be the weighted sum of the individual values of H."

Shannon proved the only function with these properties is $H(P) = \sum_{i=1}^{\sigma} p_i \log(1/p_i)$, which he called the *entropy* of P. The choice of the logarithm's base determines the unit; by convention, log means $\log_2$ and the units are bits.

If we are given a single individual string instead of a probability distribution over a set of strings, we cannot apply Shannon's definition directly. Many important papers have been written about how to measure the complexity of an individual string — by, e.g., Kolmogorov [Kol65] or Lempel and Ziv [LZ76, ZL77, ZL78] — but in this thesis we use the notion of empirical entropy. For any non-negative integer k, the *kth-order empirical entropy* of a string $s[1..n]$ (see, e.g., [Man01]) is our expected uncertainty about the random variable $s[i]$ given a context of length k, as in the following experiment: i is chosen uniformly at random from $\{1, \ldots, n\}$; if $i \leq k$, then we are told $s[i]$; otherwise, we are told $s[(i-k)..(i-1)]$. Specifically,

$$H_k(s) = \begin{cases} \displaystyle\sum_{a \in s} \frac{\mathrm{occ}\,(a, s)}{n} \log \frac{n}{\mathrm{occ}\,(a, s)} & \text{if } k = 0, \\ \displaystyle\frac{1}{n} \sum_{|\alpha|=k} |s_\alpha| H_0(s_\alpha) & \text{if } k \geq 1. \end{cases}$$

Here, $a \in s$ means character a occurs in s; $\operatorname{occ}(a, s)$ is the number of occurrences of a in s; and s_α is the string whose ith character is the one immediately following the ith occurrence of string α in s — the length of s_α is the number of occurrences of α in s, which we denote $\operatorname{occ}(\alpha, s)$, unless α is a suffix of s, in which case it is 1 less. We assume $s_\alpha = s$ when α is empty. Notice

$$0 \leq H_{k+1}(s) \leq H_k(s) \leq \log |\{a \,:\, a \in s\}| \leq \log \sigma$$

for $k \geq 0$. For example, if s is the string `mississippi`, then

$$\begin{aligned} H_0(s) &= \frac{4}{11}\log\frac{11}{4} + \frac{1}{11}\log 11 + \frac{2}{11}\log\frac{11}{2} + \frac{4}{11}\log\frac{11}{4} \approx 1.82\,, \\ H_1(s) &= \frac{1}{11}\big(3H_0(s_{\mathtt{i}}) + H_0(s_{\mathtt{m}}) + 2H_0(s_{\mathtt{p}}) + 4H_0(s_{\mathtt{s}})\big) \\ &= \frac{1}{11}\big(3H_0(\mathtt{ssp}) + H_0(\mathtt{i}) + 2H_0(\mathtt{pi}) + 4H_0(\mathtt{sisi})\big) \\ &\approx 0.63\,, \\ H_2(s) &= \frac{1}{11}\big(H_0(s_{\mathtt{ip}}) + 2H_0(s_{\mathtt{is}}) + H_0(s_{\mathtt{mi}}) + H_0(s_{\mathtt{pp}}) + 2H_0(s_{\mathtt{si}}) + 2H_0(s_{\mathtt{ss}})\big) \\ &= \frac{1}{11}\big(H_0(\mathtt{p}) + 2H_0(\mathtt{ss}) + H_0(\mathtt{s}) + H_0(\mathtt{i}) + 2H_0(\mathtt{sp}) + 2H_0(\mathtt{ii})\big) \\ &\approx 0.18\,, \end{aligned}$$

and all higher-order empirical entropies of s are 0. This means if someone chooses a character uniformly at random from `mississippi` and asks us to guess it, then our uncertainty is about 1.82 bits. If they tell us the preceding character before we guess, then on average our uncertainty is about 0.63 bits; if they tell us the preceding two characters, then our expected uncertainty decreases to 0.18 bits; given any more characters, we are certain of the answer.

Empirical entropy has a surprising connection to number theory. Let $(x)_{\sigma,n}$ denote the first n digits of the number x in base $\sigma \geq 2$. Borel [Bor09] called x *normal in base* σ if, for $\alpha \in \{0, \ldots, \sigma-1\}^*$, $\lim_{n\to\infty} \frac{\operatorname{occ}(\alpha, x_{\sigma,n})}{n} = 1/\sigma^{|\alpha|}$. For example, the Champernowne constant [Cha33] and Copeland-Erdös constant [CE46],

$$0\,.\,1\,2\,3\,4\,5\,6\,7\,8\,9\,10\,11\,12\ldots \qquad \text{and} \qquad 0\,.\,2\,3\,5\,7\,11\,13\,17\,19\,23\ldots$$

respectively, are normal in base 10. Notice x being normal in base σ is equivalent to $\lim_{n\to\infty} H_k((x)_{\sigma,n}) = \log\sigma$ for $k \geq 0$. Borel called x *absolutely normal* if it is normal in all bases. He proved almost all numbers are absolutely normal but Sierpinski [Sie17] was the first to find an example, which is still not known to be computable. Turing [Tur92] claimed there exist computable absolutely normal numbers but this was only verified recently, by Becher and Figueira [BF02]. Such numbers' representations have finite Kolmogorov complexity yet look random if we consider only empirical entropy — regardless of base and order. Of course, we are sometimes fooled whatever computable complexity metric we consider.

Now consider de Bruijn sequences [dB46] from combinatorics. An *σ-ary linear de Bruijn sequence of order k* is a string over $\{0,\ldots,\sigma-1\}$ containing every possible k-tuple exactly once. For example, the binary linear de Bruijn sequences of order 3 are the 16 10-bit substrings of 00010111000101110 and its reverse: $0001011100,\ldots,1000101110, 0111010001,\ldots,0011101000$. By definition, such strings have length $\sigma^k + k - 1$ and kth-order empirical entropy 0 (but $(k-1)$st-order empirical entropy $\frac{(\sigma^k-1)\log\sigma}{\sigma^k+k-1}$), even though there are $(\sigma!)^{\sigma^{k-1}}$ of them [vAEdB51, Knu67]. It follows that one randomly chosen has expected Kolmogorov complexity in $\Theta\left(\log(\sigma!)^{\sigma^{k-1}}\right) = \Theta(\sigma^k \log\sigma)$; whereas Borel's normal numbers can be much less complex than empirical entropy suggests, de Bruijn sequences can be much more complex.

Empirical entropy also has connections to algorithm design. For example, Munro and Spira [MS76] used 0th-order empirical entropy to analyze several sorting algorithms and Sleator and Tarjan [ST85] used it in the Static Optimality Theorem: Suppose we perform a sequence of n operations on a splay-tree, with $s[i]$ being the target of the ith operation; if s includes every key in the tree, then we use $O((H_0(s)+1)n)$ time. Of course, most of the algorithms analyzed in terms of empirical entropy are for data compression. Manzini's analysis [Man01] of the Burrows-Wheeler Transform [BW94] is particularly interesting. He proved an algorithm based on the Transform stores any string s of length n over an alphabet of size σ in at most about $(8H_k(s) + 1/20)n + \sigma^k(2\sigma\log\sigma + 9)$ bits, for all $k \geq 0$ simultaneously. Subsequent research by Ferragina, Manzini, Mäkinen and Navarro [FMMN07], for example, has shown that if $\sigma^{k+1}\log n \in o(n\log\sigma)$, then we can store an efficient index for s in $nH_k(s) + o(n\log\sigma)$ bits. Notice we cannot lift the restriction on σ and k to $\sigma^k \in O(n)$: If s is a randomly chosen σ-ary linear de Bruijn sequence of order k, then $n = \sigma^k + k - 1$ and $H_k(s) = 0$,

so $cnH_k(s) + o(n \log \sigma)) = o(\sigma^k \log \sigma)$ for any c, but $K(s) \in \Theta(\sigma^k \log \sigma)$ in the expected case.

1.2 Shannon codes and Huffman codes

The simplest representation of a code is as a binary code-tree: a rooted, ordered binary tree whose leaves are labelled with the characters in an alphabet and whose left and right edges are labelled with 0s and 1s, respectively. A code can be represented in this way if and only if it is prefix-free (often abbreviated simply as "prefix"), meaning that no one of its codewords is a prefix of any other. The codeword for the character labelling a leaf is simply the sequence of edge-labels on the path from the root to that leaf, so the expected length of a codeword is the same as the expected depth of a leaf. Shannon [Sha48] showed that, given a probability distribution $P = p_1, \ldots, p_\sigma$ with $p_1 \geq \cdots \geq p_\sigma$, we can build a prefix code whose codewords, in lexicographic order, have lengths $\lceil \log(1/p_1) \rceil, \ldots, \lceil \log(1/p_\sigma) \rceil$ (see Theorem 2.1); notice such a code has expected codeword length less than $H(P) + 1$. Moreover, he showed that no prefix code can have expected codeword length strictly less than $H(P)$. Shortly thereafter, Huffman [Huf52] gave an algorithm for building a code-tree having minimum expected depth, so prefix codes with minimum expected codeword length are often called Huffman codes (even when they cannot result from his algorithm). We refer the reader to an article by Abrahams' [Abr01] for a comprehensive survey of results on prefix codes.

Given a string $s[1..n]$ over an alphabet of size σ, if we set P to be the normalized and sorted distribution of characters in s, then a Shannon code assigns a codeword of length $\left\lceil \log \frac{n}{\operatorname{occ}(a,s)} \right\rceil$ to each character a that occurs in s, where $\operatorname{occ}(a, s)$ is the number of times a occurs in s. The sum of the codewords' length for all the characters in s is then

$$\sum_a \left\lceil \log \frac{n}{\operatorname{occ}(a, s)} \right\rceil < (H_0(s) + 1)n \,,$$

and recording the code takes $\mathcal{O}(\sigma \log \sigma)$ bits. Since a Huffman code minimizes the expected codeword length, we can obtain the same bound using a Huffman code instead of a Shannon code. Notice that, since every codeword has length at least 1 even when its character occurs nearly always, a Huffman code can also

have redundancy arbitrarily close to 1 bit per character. However, if we know something about the distribution (see [YY02] and references therein) then it is sometimes possible to prove stronger bounds for Huffman coding. For example, a result by Gallager [Gal78] implies the following:

Theorem 1.1 (Gallager, 1978)**.** *With a Huffman code, we can store s in*

$$(H + p_{\max} + 0.086)n + \mathcal{O}(\sigma \log \sigma)$$

bits, where $p_{\max} = \max_a\{\operatorname{occ}(a, s)\}/n$.

Although a Huffman code minimizes the expected codeword length and a Shannon code generally does not, a Shannon code has a better bound on each character a's pointwise redundancy, i.e., the amount by which the length of a's codeword exceeds $\log \frac{n}{\operatorname{occ}(a,s)}$. In a Shannon code, each character's pointwise redundancy is less than 1; in a Huffman code, the maximum pointwise redundancy is generally not bounded by a constant [KN76]. We consider pointwise redundancy because calculation shows that

$$\sum_{i=1}^{n} \log \frac{i + \sigma}{\operatorname{occ}(s[i], s[1..(i-1)]) + 1} \leq nH_0(s) + \mathcal{O}(\sigma \log n)$$

(see Lemmas 2.5 and 3.2) so upper bounds on pointwise redundancy can imply upper bounds for adaptive prefix coding and online sorting, as in Theorems 2.6 and 3.4, respectively. This perhaps explains why, in Chapter 2, we are able to improve the Vitter's longstanding bounds [Vit87] for adaptive prefix coding.

Theorem 1.2 (Vitter, 1987)**.** *With adaptive Huffman coding, we can store s using fewer than n more bits that we would use with Huffman coding. This is the best bound possible in the worst case, for any adaptive Huffman coding algorithm.*

Drmota and Szpankowski [DS02, DS04] showed that a generalization of Shannon codes actually have minimum maximum pointwise redundancy. In a recent paper [GG09] with Gawrychowski, we showed how their $\mathcal{O}(\sigma \log \sigma)$-time algorithm can be made to run in $\mathcal{O}(\sigma)$ time and we are now studying how to turn lower bounds for coding or sorting, such as Theorem 3.5, into lower bounds on pointwise redundancy.

When we are encoding a string of characters drawn from the alphabet, of course, it may be possible to achieve better compression by encoding more than

one character at a time. One method for doing this is run-length coding, in which we replace each run of a single distinct character by a single copy of the character and the length of the run. Another method is arithmetic coding (see, e.g., [HV92]), which encodes large blocks of characters at once, whether or not they are the same. To do this, we rearrange the alphabet so that characters are in non-increasing order by probability, then encode each block b by writing a prefix of the sum of the probabilities of all possible blocks lexicographically less than b; the more likely b is, the fewer bits we must write to distinguish its corresponding sum. The advantage of arithmetic coding is that, whereas Shannon or Huffman coding can waste nearly a whole bit for each character encoded, in theory arithmetic coding has asymptotically negligible redundancy.

1.3 The Burrows-Wheeler Transform

The Burrows-Wheeler Transform [BW94] (BWT) is an invertible transform that rearranges the characters in a string by sorting them into the lexicographical order of the suffixes that immediately follow them, as shown in Figure 1.1 (from [FGM06]). When using the BWT for compression, it is customary to append a special character \$ that is lexicographically less than any in the alphabet. To see why the BWT is invertible, consider the permutation π that stably sorts the transformed string. By definition, applying π to the the last column on the righthand matrix in Figure 1.1 (the transformed string), produces the first column (the stably sorted characters); moreover, notice that applying π to any other column produces the succeeding column. Therefore, since we can compute π from the tranformed string itself, the BWT is invertible. For a more thorough description of the BWT and its properties, we refer readers to Manzini's analysis [Man01].

Since the BWT does not change the distribution of characters, the 0th-order empirical entropy of the string remains the same; since it tends to move characters with similar contexts close together, however, the resulting string is often locally homogeneous. By applying an invertible transform such as move-to-front or distance coding, we can turn local homogeneity into global homogeneity, obtaining a string with low 0th-order empirical entropy, which we can then compress with a simple algorithm that does not use context, such as Huffman coding or arithmetic coding. Building Manzini's results [Man01], Kaplan, Landau and Verbin [KLV07]

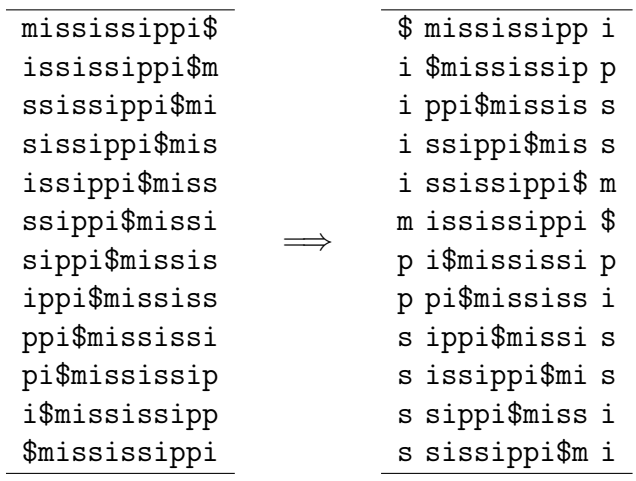

Figure 1.1: The Burrows-Wheeler Transform for the string $s =$ `mississippi`. The matrix on the right has the rows sorted in lexicographic order. The output of the BWT is the last column of the matrix, i.e., `ipssm$pissii`.

proved several upper bounds for this approach, the best of which is shown below, and Kaplan and Verbin [KV07] later also proved lower bounds.

Theorem 1.3 (Kaplan, Landau and Verbin, 2007)**.** *By applying the BWT followed by distance coding and arithmetic coding, we can store s in $1.73nH_k(s) + \mathcal{O}(\log n)$ bits for all k simultaneously.*

We note that the coefficients hidden in the asymptotic notation grow quickly in terms of σ and k so, unlike Manzini's analysis, the bound above does not guarantee good compression when s is very compressible. We wrote a paper with Manzini [GM07a] showing how to combine ideas from both analyses [Man01, KLV07] in order to obtain a good bound even in this case.

One alternative is to partition the result of the BWT into consecutive substrings, each with low 0th-order empirical entropy, and compress them separately. Ferragina, Giancarlo, Manzini and Sciortino [FGMS05] (see also [FGM06, FNV09]) gave an $\mathcal{O}(n)$-time algorithm Boost for computing a partition such that, if we compress each substring well in terms of its 0th-order empirical entropy, then the whole transformed string is compressed well in terms of its higher-order empirical entropies, as stated in the theorem below (based on the presentation in [FGM06]). Ferragina, Giancarlo and Manzini [FGM09] later combined this algorithm with other techniques to obtain the best known bounds for when s is very compressible.

Theorem 1.4 (Ferragina, Giancarlo, Manzini and Sciortino, 2005)**.** *Let* A *be a compression algorithm that encodes any string x in at most $\lambda|x|H_0(x) + \eta|x| + \mu$ bits, where λ, η and μ are constants. If we use* A *to compress the substrings in the partition computed by* Boost*, then the overall output size is bounded by $\lambda n H_k(s) + \log n + \eta n + g_k$ for any $k \geq 0$, where g_k depends only on σ and k.*

Although several other alternatives have been presented by, e.g., Mäkinen and Navarro [MN07] and Gupta, Grossi and Vitter [GGV08], the literature on the BWT is already too large, and growing too rapidly, for us to give anything like a thorough survey of it here.

Chapter 2

Adaptive Prefix Coding

Prefix codes are sometimes called instantaneous codes because, since no codeword is a prefix of another, the decoder can output each character once it reaches the end of its codeword. Adaptive prefix coding could thus be called "doubly instantaneous", because the encoder must produce a self-delimiting codeword for each character before reading the next one. The main idea behind adaptive prefix coding is simple: both the encoder and the decoder start with the same prefix code; the encoder reads the first character of the input and writes its codeword; the decoder reads that codeword and decodes the first character; the encoder and decoder now have the same information, and they update their copies of the code in the same way; then they recurse on the remainder of the input. The two main challenges are, first, to update efficiently the two copies of the code and, second, to prove the total length of the encoding is not much more than it would be if we were to use an optimal static prefix coder.

Because Huffman codes [Huf52] have minimum expected codeword length, early work on adaptive prefix coding naturally focused on efficiently maintaining a Huffman code for the prefix of the input already encoded. Faller [Fal73], Gallager [Gal78] and Knuth [Knu85] developed an adaptive Huffman coding algorithm — usually known as the FGK algorithm, for their initials — and showed it takes time proportional to the length of the encoding it produces. Vitter [Vit87] gave an improved version of their algorithm and showed it uses less than one more bit per character than we would use with static Huffman coding. (For simplicity we consider only binary encodings; therefore, by log we always mean $\log_2$.) Milidiú, Laber and Pessoa [MLP99] later extended Vitter's techniques to analyze the FGK algorithm, and showed it uses less than two more bits per character

than we would use with static Huffman coding. Suppose the input is a string s of n characters drawn from an alphabet of size $\sigma \ll n$; let H be the empirical entropy of s (i.e., the entropy of the normalized distribution of characters in s) and let r be the redundancy of a Huffman code for s (i.e., the difference between the expected codeword length and H). The FGK algorithm encodes s as at most $(H + 2 + r)n + o(n)$ bits and Vitter's algorithm encodes it as at most $(H + 1 + r)n + o(n)$ bits; both take $\mathcal{O}((H + 1)n)$ total time to encode and decode, or $\mathcal{O}(H + 1)$ amortized time per character. Table 2.1 summarizes bounds for various adaptive prefix coders.

If s is drawn from a memoryless source then, as n grows, adaptive Huffman coding will almost certainly "lock on" to a Huffman code for the source and, thus, use $(H + r)n + o(n)$ bits. In this case, however, the whole problem is easy: we can achieve the same bound, and use less time, by periodically building a new Huffman code. If s is chosen adversarially, then every algorithm uses at least $(H + 1 + r)n - o(n)$ bits in the worst case. To see why, fix an algorithm and suppose $\sigma = n^{1/2} = 2^\ell + 1$ for some integer ℓ, so any binary tree on σ leaves has at least two leaves with depths at least $\ell + 1$. Any prefix code for σ characters can be represented as a code-tree on σ leaves; the length of the lexicographically ith codeword is the depth of the ith leaf from the left. It follows that an adversary can choose s such that the algorithm encodes it as at least $(\ell + 1)n$ bits. On the other hand, a static prefix coding algorithm can assign codewords of length ℓ to the $\sigma - 2$ most frequent characters and codewords of length $\ell + 1$ to the two least frequent characters, and thus use at most $\ell n + 2n/\sigma + \mathcal{O}(\sigma \log \sigma) = \ell n + o(n)$ bits. Therefore, since the expected codeword length of a Huffman code is minimum, $(H + r)n \leq \ell n + o(n)$ and so $(\ell + 1)n \geq (H + 1 + r)n - o(n)$.

This lower bound seems to say that Vitter's upper bound cannot be significantly improved. However, to force the algorithm to use $(H + 1 + r)n - o(n)$ bits, it might be that the adversary must choose s such that r is small. In a previous paper [Gag07a] we were able to show this is the case, by giving an adaptive prefix coder that encodes s in at most $(H + 1)n + o(n)$ bits. This bound is perhaps a little surprising, since our algorithm was based on Shannon codes [Sha48], which generally do not have minimum expected codeword lengths. Like the FGK algorithm and Vitter's algorithm, our algorithm used $\mathcal{O}(H + 1)$ amortized time per character to encode and decode. Recently, Karpinski and Nekrich [KN09] showed how to combine some of our results with properties of canonical codes, defined by

Schwartz and Kallick [SK64] (see also [Kle00, TM00, TM01]), to achieve essentially the same compression while encoding and decoding each character in $\mathcal{O}(1)$ amortized time and $\mathcal{O}(\log(H+2))$ amortized time, respectively. Nekrich [Nek07] implemented their algorithm and observed that, in practice, it is significantly faster than arithmetic coding and slightly faster than Turpin and Moffat's GEO coder [TM01], although the compression it achieves is not quite as good. The rest of this chapter is based on joint work with Nekrich [GN] that shows how, on a RAM with $\Omega(\log n)$-bit words, we can speed up our algorithm even more, to both encode and decode in $\mathcal{O}(1)$ worst-case time. We note that Rueda and Oommen [RO04, RO06, RO08] have demonstrated the practicality of certain implementations of adaptive Fano coding, which is somewhat related to our algorithm, especially a version [Gag04] in which we maintained an explicit code-tree.

Table 2.1: Bounds for adaptive prefix coding: the times to encode and decode each character and the total length of the encoding. Bounds in the first row and last column are worst-case; the others are amortized.

	Encoding	Decoding	Length
Gagie and Nekrich [GN]	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$(H+1)n + o(n)$
Karpinski and Nekrich [KN09]	$\mathcal{O}(1)$	$\mathcal{O}(\log(H+2))$	$(H+1)n + o(n)$
Gagie [Gag07a]	$\mathcal{O}(H+1)$	$\mathcal{O}(H+1)$	$(H+1)n + o(n)$
Vitter [Vit87]	$\mathcal{O}(H+1)$	$\mathcal{O}(H+1)$	$(H+1+r)n + o(n)$
Knuth [Knu85] / Gallager [Gal78] / Faller [Fal73]	$\mathcal{O}(H+1)$	$\mathcal{O}(H+1)$	$(H+2+r)n + o(n)$

2.1 Algorithm

A Shannon code is one in which, if a character has probability p, then its codeword has length at most $\lceil \log(1/p) \rceil$. In his seminal paper on information theory, Shannon [Sha48] showed how to build such a code for any distribution containing only positive probabilities.

Theorem 2.1 (Shannon, 1948)**.** *Given a probability distribution $P = p_1, \ldots, p_\sigma$ with $p_1 \geq \cdots \geq p_\sigma > 0$, we can build a prefix code in $\mathcal{O}(\sigma)$ time whose codewords, in lexicographic order, have lengths $\lceil \log(1/p_1) \rceil, \ldots, \lceil \log(1/p_\sigma) \rceil$.*

We encode each character of s using a canonical Shannon code for a probability distribution that is roughly the normalized distribution of characters in the prefix of s already encoded. In order to avoid having to consider probabilities equal to 0, we start by assigning every character a count of 1. This means the smallest probability we ever consider is at least $1/(n+\sigma)$, so the longest codeword we ever consider is $\mathcal{O}(\log n)$ bits.

A canonical code is one in which the first codeword is a string of 0s and, for $1 \leq i < \sigma$, we can obtain the $(i+1)$st codeword by incrementing the ith codeword (viewed as a binary number) and appending some number of 0s to it. For example, Figure 2.1 shows the codewords in a canonical code, together with their lexicographic ranks. By definition, the difference between two codewords of the same length in a canonical code, viewed as binary numbers, is the same as the difference between their ranks. For example, the third codeword in Figure 2.1 is 0100 and the sixth codeword is 0111, and $(0111)_2 - (0100)_2 = 6 - 3$, where $(\cdot)_2$ means the argument is to be viewed as a binary number. We use this property to build a representation of the code that lets us quickly answer encoding and decoding queries.

1)	000	7)	1000
2)	001	8)	1001
3)	0100	9)	10100
4)	0101	10)	10101
5)	0110		⋮
6)	0111	16)	11011

Figure 2.1: The codewords in a canonical code.

We maintain the following data structures: an array A_1 that stores the codewords' ranks in lexicographic order by character; an array A_2 that stores the characters and their frequencies in order by frequency; a dictionary D_1 that stores the rank of the first codeword of each length, with the codeword itself as auxiliary information; and a dictionary D_2 that stores the first codeword of each length, with its rank as auxiliary information.

To encode a given character a, we first use A_1 to find a's codeword's rank; then use D_1 to find the rank of the first codeword of the same length and that codeword as auxiliary information; then add the difference in ranks to that codeword, viewed as a binary number, to obtain a's codeword. For example, if the codewords are as

shown in Figure 2.1 and a is the jth character in the alphabet and has codeword 0111, then

1. $A_1[j] = 6$,
2. $D_1.\text{pred}(6) = \langle 3, 0100 \rangle$,
3. $(0100)_2 + 6 - 3 = (0111)_2$.

To decode a given a binary string prefixed with its codeword, we first search in D_2 for the predecessor of the first $\lceil \log(n + \sigma) \rceil$ bits of the binary string, to find the first codeword of the same length and that codeword's rank as auxiliary information; then add that rank to the difference in codewords, viewed as binary numbers, to obtain a's codeword's rank; then use A_2 to find a. In our example above,

1. $D_2.\text{pred}(0111\ldots) = \langle 0100, 3 \rangle$,
2. $3 + (0111)_2 - (0100)_2 = 6$,
3. $A_2[6] = a_j$.

Admittedly, reading the first $\lceil \log(n + \sigma) \rceil$ bits of the binary string will generally result in the decoder reading past the end of most codewords before outputting the corresponding character. We see no way to avoid this without potentially requiring the decoder to read some codewords bit by bit.

Querying A_1 and A_2 takes $\mathcal{O}(1)$ worst-case time, so the time to encode and decode depends mainly on the time needed for predecessor queries on D_1 and D_2. Since the longest codeword we ever consider is $\mathcal{O}(\log n)$ bits, each dictionary contains $\mathcal{O}(\log n)$ keys, so we can implement each as an instance of the data structure described below, due to Fredman and Willard [FW93]. This way, apart from the time to update the dictionaries, we encode and decode each character in a total of $\mathcal{O}(1)$ worst-case time. Andersson, Miltersen and Thorup [ABT99] showed Fredman and Willard's data structure can be implemented with AC^0 instructions, and Thorup [Tho03] showed it can be implemented with AC^0 instructions available on a Pentium 4; admittedly though, in practice it might still be faster to use a sorted array to encode and decode each character in $\mathcal{O}(\log \log n)$ time.

Lemma 2.2 (Fredman & Willard, 1993)**.** *Given $\mathcal{O}\left(\log^{1/6} n\right)$ keys, we can build a dictionary in $\mathcal{O}\left(\log^{2/3} n\right)$ worst-case time that stores those keys and supports predecessor queries in $\mathcal{O}(1)$ worst-case time.*

Corollary 2.3. *Given $\mathcal{O}(\log n)$ keys, we can build a dictionary in $\mathcal{O}\left(\log^{3/2} n\right)$ worst-case time that stores those keys and supports predecessor queries in $\mathcal{O}(1)$ worst-case time.*

Proof. We store the keys at the leaves of a search tree with degree $\mathcal{O}\left(\log^{1/6} n\right)$, size $\mathcal{O}\left(\log^{5/6} n\right)$ and height at most 5. Each node stores an instance of Fredman and Willard's dictionary from Lemma 2.2: each dictionary at a leaf stores $\mathcal{O}\left(\log^{1/6} n\right)$ keys and each dictionary at an internal node stores the first key in each of its children's dictionaries. It is straightforward to build the search tree in $\mathcal{O}\left(\log^{2/3+5/6} n\right) = \mathcal{O}\left(\log^{3/2} n\right)$ time and implement queries in $\mathcal{O}(1)$ time. □

Since a codeword's lexicographic rank is the same as the corresponding character's rank by frequency, and a character's frequency is an integer than changes only by being incremented after each of its occurrences, we can use a data structure due to Gallager [Gal78] to update A_1 and A_2 in $\mathcal{O}(1)$ worst-case time per character of s. We can use $\mathcal{O}(\log n)$ binary searches in A_2 and $\mathcal{O}\left(\log^2 n\right)$ time to compute the number of codewords of each length; building D_1 and D_2 then takes $\mathcal{O}\left(\log^{3/2} n\right)$ time. Using multiple copies of each data structure and standard background-processing techniques, we update each set of copies after every $\lfloor \log^2 n \rfloor$ characters and stagger the updates, such that we need spend only $\mathcal{O}(1)$ worst-case time per character and, for $1 \leq i \leq n$, the copies we use to encode the ith character of s will always have been last updated after we encoded the $(i - \lfloor \log^2 n \rfloor)$th character.

Writing $s[i]$ for the ith character of s, $s[1..i]$ for the prefix of s of length i, and $\operatorname{occ}(s[i], s[1..i])$ for the number of times $s[i]$ occurs in $s[1..i]$, we can summarize the results of this section as the following lemma.

Lemma 2.4. *For $1 \leq i \leq n$, we can encode $s[i]$ as at most*

$$\left\lceil \log \frac{i + \sigma}{\max\left(\operatorname{occ}(s[i], s[1..i]) - \lfloor \log^2 n \rfloor, 1\right)} \right\rceil$$

bits such that encoding and decoding it takes $\mathcal{O}(1)$ worst-case time.

2.2 Analysis

Analyzing the length of the encoding our algorithm produces is just a matter of bounding the sum of the codewords' lengths. Fortunately, we can do this using a modification of the proof that adaptive arithmetic coding produces an encoding not much longer than the one produced by decrementing arithmetic coding (see, e.g., [HV92]).

Lemma 2.5.

$$\sum_{i=1}^{n} \left\lceil \log \frac{i+\sigma}{\max\big(\operatorname{occ}(s[i], s[1..i]) - \lfloor \log^2 n \rfloor, 1\big)} \right\rceil \leq (H+1)n + \mathcal{O}\big(\sigma \log^3 n\big) \ .$$

Proof. Since $\big\{\operatorname{occ}(s[i], s[1..i]) \ : \ 1 \leq i \leq n\big\}$ and $\left\{ j \ : \ \begin{array}{l} 1 \leq j \leq \operatorname{occ}(a, s), \\ a \text{ a character} \end{array} \right\}$ are the same multiset,

$$\begin{aligned}
&\sum_{i=1}^{n} \left\lceil \log \frac{i+\sigma}{\max\big(\operatorname{occ}(s[i], s[1..i]) - \lfloor \log^2 n \rfloor, 1\big)} \right\rceil \\
&< \sum_{i=1}^{n} \log(i+\sigma) - \sum_{i=1}^{n} \log \max\big(\operatorname{occ}(s[i], s[1..i]) - \lfloor \log^2 n \rfloor, 1\big) + n \\
&= \sum_{i=1}^{n} \log(i+\sigma) - \sum_{a} \sum_{j=1}^{\operatorname{occ}(a,s) - \lfloor \log^2 n \rfloor} \log j + n \\
&< \sum_{i=1}^{n} \log i + \sigma \log(n+\sigma) - \sum_{a} \sum_{j=1}^{\operatorname{occ}(a,s)} \log j + \sigma \log^3 n + n \\
&= \log(n!) - \sum_{a} \log(\operatorname{occ}(a, s)!) + n + \mathcal{O}\big(\sigma \log^3 n\big) \ .
\end{aligned}$$

Since

$$\log(n!) - \sum_{a} \log(\operatorname{occ}(a, s)!) = \log \frac{n!}{\prod_a \operatorname{occ}(a, s)!}$$

is the number of distinct arrangements of the characters in s, we could complete the proof by information-theoretic arguments; however, we will use straightforward calculation. Specifically, Robbins' extension [Rob55] of Stirling's Formula,

$$\sqrt{2\pi} x^{x+1/2} e^{-x+1/(12x+1)} < x! < \sqrt{2\pi} x^{x+1/2} e^{-x+1/(12x)} \, ,$$

implies that

$$x \log x - x \log e < \log(x!) \leq x \log x - x \log e + \mathcal{O}(\log x) \ ,$$

where e is the base of the natural logarithm. Therefore, since $\sum_a \operatorname{occ}(a, s) = n$,

$$\begin{aligned}
&\log(n!) - \sum_a \log(\operatorname{occ}(a, s)!) + n + \mathcal{O}\big(\sigma \log^3 n\big) \\
&= n \log n - \sum_a \operatorname{occ}(a, s) \log \operatorname{occ}(a, s) - \\
&\quad n \log e + \sum_a \operatorname{occ}(a, s) \log e + n + \mathcal{O}\big(\sigma \log^3 n\big) \\
&= \sum_a \operatorname{occ}(a, s) \log \frac{n}{\operatorname{occ}(a, s)} + n + \mathcal{O}\big(\sigma \log^3 n\big) \\
&= (H + 1)n + \mathcal{O}\big(\sigma \log^3 n\big) \ .
\end{aligned}$$

□

Combining Lemmas 2.4 and 2.5 and assuming $\sigma = o(n/\log^3 n)$ immediately gives us our result for this chapter.

Theorem 2.6. *We can encode s as at most $(H+1)n+o(n)$ bits such that encoding and decoding each character takes $\mathcal{O}(1)$ worst-case time.*

Chapter 3

Online Sorting with Few Comparisons

Comparison-based sorting is perhaps the most studied problem in computer science, but there remain basic open questions about it. For example, exactly how many comparisons does it take to sort a multiset? Over thirty years ago, Munro and Spira [MS76] proved distribution-sensitive upper and lower bounds that differ by $\mathcal{O}(n \log\log \sigma)$, where n is the size of the multiset s and σ is the number of distinct elements s contains. Specifically, they proved that $nH + \mathcal{O}(n)$ ternary comparisons are sufficient and $nH - (n - \sigma)\log\log\sigma - \mathcal{O}(n)$ are necessary, where $H = \sum_a \frac{\mathrm{occ}(a,s)}{n} \log \frac{n}{\mathrm{occ}(a,s)}$ denotes the entropy of the distribution of elements in s and $\mathrm{occ}\,(a, s)$ denotes the multiplicity of the distinct element a in s. Throughout, by log we mean $\log_2$. Their bounds have been improved in a series of papers, summarized in Table 3.1, so that the difference between the best upper and lower bounds (of which we are aware) is now slightly less than $(1 + \log e)n \approx 2.44n$, where e is the base of the natural logarithm.

Apart from the bounds shown in Table 3.1, there have been many bounds proven about, e.g., sorting multisets in-place or with minimum data movement, or in the external-memory or cache-oblivious models. In this chapter we consider online stable sorting; online algorithms sort s element by element and keep those already seen in sorted order, and stable algorithms preserve the order of equal elements. For example, splaysort (i.e., sorting by insertion into a splay tree [ST85]) is online, stable and takes $\mathcal{O}((H + 1)n)$ comparisons and time. In Section 3.1 we show how, if $\sigma = o(n^{1/2}/\log n)$, then we can sort s online and stably using $(H + 1)n + o(n)$ ternary comparisons and $\mathcal{O}((H + 1)n)$ time. In Section 3.2 we

prove $(H+1)n - o(n)$ comparisons are necessary in the worst case.

Table 3.1: Bounds for sorting a multiset using ternary comparisons.

	Upper bound	Lower bound
Munro and Raman [MR91]		$(H - \log e)n + \mathcal{O}(\log n)$
Fischer [Fis84]	$(H+1)n - \sigma$	$(H - \log H)n - \mathcal{O}(n)$
Dobkin and Munro [DM80]		$\left(H - n\log\left(\log n - \frac{\sum_a \operatorname{occ}(a,s)\log\operatorname{occ}(a,s)}{n}\right)\right)n - \mathcal{O}(n)$
Munro and Spira [MS76]	$nH + \mathcal{O}(n)$	$nH - (n-\sigma)\log\log\sigma - \mathcal{O}(n)$

3.1 Algorithm

Our idea is to sort s by inserting its elements into a binary search tree T, which we rebuild occasionally using the following theorem by Mehlhorn [Meh77]. We rebuild T whenever the number of elements processed since the last rebuild is equal to the number of distinct elements seen by the time of the last rebuild. This way, we spend $\mathcal{O}(n)$ total time rebuilding T.

Theorem 3.1 (Mehlhorn, 1977)**.** *Given a probability distribution $P = p_1, \ldots, p_k$ on k keys, with no $p_i = 0$, in $\mathcal{O}(k)$ time we can build a binary search tree containing those keys at depths at most $\log(1/p_1), \ldots, \log(1/p_k)$.*

To rebuild T after processing i elements of s, to each distinct element a seen so far we assign probability $\operatorname{occ}(a, s[1..i])/i$, where $s[1..i]$ denotes the first i elements of s; we then apply Theorem 3.1. Notice the smallest probability we consider is at least $1/n$, so the resulting tree has height at most $\log n$.

We want T always to contain a node for each distinct element a seen so far, that stores a as a key and stores a linked list of a's occurrences so far. After we use Mehlhorn's theorem, therefore, we extend T and then replace each of its leaves by an empty AVL tree [AL62]. To process an element $s[i]$ of s, we search for $s[i]$ in T; if we find a node v whose key is equal to $s[i]$, then we append $s[i]$ to v's linked list; otherwise, our search ends at a node of an AVL tree, into which we insert a new node whose key is equal to $s[i]$ and whose linked list contains $s[i]$.

If an element equal to $s[i]$ occurs by the time of the last rebuild before we process $s[i]$, then the corresponding node is at depth at most $\log \frac{i}{\max\left(\operatorname{occ}(s[i], s[1..i]) - \sigma, 1\right)}$ in T, so the number of ternary comparisons we use to insert $s[i]$ into T is at most

that number plus 1; the extra comparison is necessary to check that the algorithm should not proceed deeper into the tree. Otherwise, since our AVL trees always contain at most σ nodes, we use $\mathcal{O}(\log n + \log \sigma) = \mathcal{O}(\log n)$ comparisons. Therefore, we use a total of at most

$$\sum_{i=1}^{n} \log \frac{i}{\max\left(\operatorname{occ}\left(s[i], s[1..i]\right) - \sigma, 1\right)} + n + \mathcal{O}(\sigma \log n)$$

comparisons to sort s and, assuming each comparison takes $\mathcal{O}(1)$ time, a proportional amount of time. We can bound this sum using the following technical lemma, which says the logarithm of the number of distinct arrangements of the elements in s is close to nH, and the subsequent corollary. We write $a_1, \ldots, a_\sigma$ to denote the distinct elements in s.

Lemma 3.2.

$$nH - \mathcal{O}(\sigma \log(n/\sigma)) \leq \log \binom{n}{\operatorname{occ}\left(a_1, s\right), \ldots, \operatorname{occ}\left(a_\sigma, s\right)} \leq nH + \mathcal{O}(\log n) \ .$$

Proof. Robbins' extension [Rob55] of Stirling's Formula,

$$\sqrt{2\pi} x^{x+1/2} e^{-x+1/(12x+1)} < x! < \sqrt{2\pi} x^{x+1/2} e^{-x+1/(12x)} \,,$$

implies that

$$x \log x - x \log e < \log(x!) \leq x \log x - x \log e + \mathcal{O}(\log x) \ .$$

Therefore, since $\sum_a \operatorname{occ}(a, s) = n$, straightforward calculation shows that

$$\log \binom{n}{\operatorname{occ}\left(a_1, s\right), \ldots, \operatorname{occ}\left(a_\sigma, s\right)} = \log(n!) - \sum_a \log(\operatorname{occ}(a, s)!)$$

is at least $nH - \mathcal{O}(\sigma \log(n/\sigma))$ and at most $nH + \mathcal{O}(\log n)$. □

Corollary 3.3.

$$\sum_{i=1}^{n} \log \frac{i}{\max\left(\operatorname{occ}\left(s[i], s[1..i]\right) - \sigma, 1\right)} + n + \mathcal{O}(\sigma \log n)$$
$$\leq (H+1)n + \mathcal{O}\big(\sigma^2 \log n\big) \ .$$

Proof. Since $\big\{\operatorname{occ}(s[i], s[1..i]) \,:\, 1 \le i \le n\big\}$ and $\left\{ j \,:\, \begin{array}{l} 1 \le j \le \operatorname{occ}(a, s), \\ a \text{ an element} \end{array} \right\}$ are the same multiset,

$$\begin{aligned}
&\sum_{i=1}^{n} \log \frac{i}{\max\big(\operatorname{occ}(s[i], s[1..i]) - \sigma, 1\big)} \\
&= \log(n!) - \sum_{a} \sum_{j=1}^{\operatorname{occ}(a,s)-\sigma} \log j \\
&\le \log(n!) - \sum_{a} \sum_{j=1}^{\operatorname{occ}(a,s)} \log j + \mathcal{O}\big(\sigma^2 \log n\big) \\
&= \log(n!) - \sum_{a} \log(\operatorname{occ}(a, s)!) + \mathcal{O}\big(\sigma^2 \log n\big) \\
&= \log \binom{n}{\operatorname{occ}(a_1, s), \ldots, \operatorname{occ}(a_\sigma, s)} + \mathcal{O}\big(\sigma^2 \log n\big) \\
&\le nH + \mathcal{O}\big(\sigma^2 \log n\big) \ ,
\end{aligned}$$

by Lemma 3.2. It follows that

$$\begin{aligned}
&\sum_{i=1}^{n} \log \frac{i}{\max\big(\operatorname{occ}(s[i], s[1..i]) - \sigma, 1\big)} + n + \mathcal{O}(\sigma \log n) \\
&\le (H+1)n + \mathcal{O}\big(\sigma^2 \log n\big) \ .
\end{aligned}$$

□

Our upper bound follows immediately from Corollary 3.3.

Theorem 3.4. *When $\sigma = o(n^{1/2}/\log n)$, we can sort s online and stably using $(H+1)n + o(n)$ ternary comparisons and $\mathcal{O}((H+1)n)$ time.*

3.2 Lower bound

Consider any online, stable sorting algorithm that uses ternary comparisons. Since the algorithm is online and stable, it must determine each element's rank relative to the distinct elements already seen, before moving on to the next element. Since it uses ternary comparisons, we can represent its strategy for each element as an extended binary search tree whose keys are the distinct elements already seen. If the current element is distinct from all those already seen, then the algorithm reaches a leaf of the tree, and the minimum number of comparisons it

performs is equal to that leaf's depth. If the current element has been seen before, however, then the algorithm stops at an internal node and the minimum number of comparisons it performs is 1 greater than that node's depth; again, the extra comparison is necessary to check that the algorithm should not proceed deeper into the tree.

Suppose $\sigma = o(n/\log n)$ is a power of 2; then, in any binary search tree on σ keys, some key has depth $\log \sigma \geq H$ (the inequality holds because any distribution on σ elements has entropy at most $\log \sigma$). Furthermore, suppose an adversary starts by presenting one copy of each of σ distinct elements; after that, it considers the algorithm's strategy for the next element as a binary search tree, and presents the deepest key. This way, the adversary forces the algorithm to use at least $(n - \sigma)(\log \sigma + 1) \geq (H + 1)n - o(n)$ comparisons.

Theorem 3.5. *We generally need at least $(H + 1)n - o(n)$ ternary comparisons to sort s online and stably.*

Chapter 4

Online Sorting with Sublinear Memory

When in doubt, sort! Librarians, secretaries and computer scientists all know that when faced with lots of data, often the best thing is to organize them. For some applications, though, the data are so overwhelming that we cannot sort. The streaming model was introduced for situations in which the flow of data can be neither paused nor stored in its entirety; the model's assumptions are that we are allowed only one pass over the input and memory sublinear in its size (see, e.g., [Mut05]). Those assumptions mean we cannot sort in general, but in this chapter we show we can when the data are very compressible.

Our inspiration comes from two older articles on sorting. In the first, "Sorting and searching in multisets" from 1976, Munro and Spira [MS76] considered the problem of sorting a multiset s of size n containing σ distinct elements in the comparison model. They showed sorting s takes $\Theta((H+1)n)$ time, where $H = \sum_{i=1}^{\sigma}(n_i/n)\log(n/n_i)$ is the entropy of s, log means $\log_2$ and n_i is the frequency of the ith smallest distinct element. When σ is small or the distribution of elements in s is very skewed, this is a significant improvement over the $\Theta(n \log n)$ bound for sorting a set of size n.

In the second article, "Selection and sorting with limited storage" from 1980, Munro and Paterson [MP80] considered the problem of sorting a set s of size n using limited memory and few passes. They showed sorting s in p passes takes $\Theta(n/p)$ memory locations in the following model (we have changed their variable names for consistency with our own):

> "In our computational model the data is a sequence of n distinct elements stored on a one-way read-only tape. An element from the tape can be read into one of r locations of random-access storage. The elements are from some totally ordered set (for example the real numbers) and a binary comparison can be made at any time between any two elements within the random-access storage. Initially the storage is empty and the tape is placed with the reading head at the beginning. After each pass the tape is rewound to this position with no reading permitted. ... [I]n view of the limitations imposed by our model, [sorting] must be considered as the *determination* of the sorted order rather than any actual rearrangement."

An obvious question — but one that apparently has still not been addressed decades later — is how much memory we need to sort a multiset in few passes; in this chapter we consider the case when we are allowed only one pass. We assume our input is the same as Munro and Spira's, a multiset $s = \{s_1, \ldots, s_n\}$ with entropy H containing σ distinct elements. To simplify our presentation, we assume $\sigma \geq 2$ so $Hn = \Omega(\log n)$. Our model is similar to Munro and Paterson's but it makes no difference to us whether the tape is read-only or read-write, since we are allowed only one pass, and whereas they counted memory locations, we count bits. We assume machine words are $\Theta(\log n)$ bits long, an element fits in a constant number of words and we can perform standard operations on words in unit time. Since entropy is minimized when the distribution is maximally skewed,

$$Hn \geq n \left(\frac{n-\sigma+1}{n} \log \frac{n}{n-\sigma+1} + \frac{\sigma-1}{n} \log n \right) \geq (\sigma - 1) \log n \,;$$

thus, under our assumptions, $\mathcal{O}(\sigma)$ words take $\mathcal{O}(\sigma \log n) \subseteq \mathcal{O}(Hn)$ bits.

In Section 4.1 we consider the problem of determining the permutation π such that $s_{\pi(1)}, \ldots, s_{\pi(n)}$ is the stable sort of s (i.e., $s_{\pi(i)} \leq s_{\pi(i+1)}$ and, if $s_{\pi(i)} = s_{\pi(i+1)}$, then $\pi(i) < \pi(i+1)$). For example, if

$$s = a_1, b_1, r_1, a_2, c, a_3, d, a_4, b_2, r_2, a_5$$

(with subscripts serving only to distinguish copies of the same distinct element),

then the stable sort of s is

$$\begin{aligned} &a_1, a_2, a_3, a_4, a_5, b_1, b_2, c, d, r_1, r_2 \\ &= s_1, s_4, s_6, s_8, s_{11}, s_2, s_9, s_5, s_7, s_3, s_{10} \end{aligned}$$

and

$$\pi = 1, 4, 6, 8, 11, 2, 9, 5, 7, 3, 10\,.$$

We give a simple algorithm that computes π using one pass, $\mathcal{O}((H+1)n)$ time and $\mathcal{O}(Hn)$ bits of memory. In Section 4.2 we consider the simpler problem of determining a permutation ρ such that $s_{\rho(1)}, \ldots, s_{\rho(n)}$ is in sorted order (not necessarily stably-sorted). We prove that in the worst case it takes $\Omega(Hn)$ bits of memory to compute any such ρ in one pass.

4.1 Algorithm

The key to our algorithm is the fact $\pi = \ell_1 \cdots \ell_\sigma$, where ℓ_i is the sorted list of positions in which the ith smallest distinct element occurs. In our example, $s = a, b, r, a, c, a, d, a, b, r, a$,

$$\begin{aligned} \ell_1 &= 1, 4, 6, 8, 11 \\ \ell_2 &= 2, 9 \\ \ell_3 &= 5 \\ \ell_4 &= 7 \\ \ell_5 &= 3, 10\,. \end{aligned}$$

Since each ℓ_i is a strictly increasing sequence, we can store it compactly using Elias' gamma code [Eli75]: we write the first number in ℓ_i, encoded in the gamma code; for $1 \leq j < n_i$, we write the difference between the $(j+1)$st and jth numbers, encoded in the gamma code. The gamma code is a prefix-free code for the positive integers; for $x \geq 1$, $\gamma(x)$ consists of $\lfloor \log x \rfloor$ zeroes followed by the $(\lfloor \log x \rfloor + 1)$-bit binary representation of x. In our example, we encode ℓ_1 as

$$\gamma(1)\ \gamma(3)\ \gamma(2)\ \gamma(2)\ \gamma(3) = 1\ 011\ 010\ 010\ 011\ .$$

Lemma 4.1. *We can store π in $\mathcal{O}(Hn)$ bits of memory.*

Proof. Encoding the length of every list with the gamma code takes $\mathcal{O}(\sigma \log n) \subseteq \mathcal{O}(Hn)$ bits. Notice the numbers in each list ℓ_i sum to at most n. By Jensen's Inequality, since $|\gamma(x)| \leq 2 \log x + 1$ and log is concave, we store ℓ_i in at most $2n_i \log(n/n_i) + n_i$ bits. Therefore, storing $\ell_1, \ldots, \ell_\sigma$ as described above takes

$$\sum_{i=1}^{\sigma} (2n_i \log(n/n_i) + n_i) = \mathcal{O}((H+1)n)$$

bits of memory.

To reduce $\mathcal{O}((H+1)n)$ to $\mathcal{O}(Hn)$ — important when one distinct element dominates, so H is close to 0 and $Hn \ll n$ — we must avoid writing a codeword for each element in s. Notice that, for each run of length at least 2 in s (a run being a maximal subsequence of copies of the same element) there is a run of 1's in the corresponding list ℓ_i. We replace each run of 1's in ℓ_i by a single 1 and the length of the run. For each except the last run of each distinct element, the run-length is at most the number we write for the element in s immediately following that run; storing the last run-length for every character takes $\mathcal{O}(\sigma \log n) \subseteq \mathcal{O}(Hn)$ bits. It follows that storing $\ell_1, \ldots, \ell_\sigma$ takes

$$\sum_{i=1}^{\sigma} \mathcal{O}(r_i \log(n/r_i) + r_i) \leq \sum_{i=1}^{\sigma} \mathcal{O}(n_i \log(n/n_i)) + \mathcal{O}(r) = \mathcal{O}(Hn + r)$$

bits, where r_i is the number of runs of the ith smallest distinct element and r is the total number of runs in s. Mäkinen and Navarro [MN05] showed $r \leq Hn + 1$, so our bound is $\mathcal{O}(Hn)$. □

To compute $\ell_1, \ldots, \ell_\sigma$ in one pass, we keep track of which distinct elements have occurred and the positions of their most recent occurrences, which takes $\mathcal{O}(\sigma)$ words of memory. For $1 \leq j \leq n$, if s_j is an occurrence of the ith smallest distinct element and that element has not occurred before, then we start ℓ_i's encoding with $\gamma(j)$; if it last occurred in position $k \leq j-2$, then we append $\gamma(j-k)$ to ℓ_i; if it occurred in position $j-1$ but not $j-2$, then we append $\gamma(1)$ $\gamma(1)$ to ℓ_i; if it occurred in both positions $j-1$ and $j-2$, then we increment the encoded run-length at the end of ℓ_i's encoding. Because we do not know in advance how many bits we will use to encode each ℓ_i, we keep the encoding in an expandable binary array [CLRS01]: we start with an array of size 1 bit; whenever the array overflows, we create a new array twice as big, copy the contents from the old array

into the new one, and destroy the old array. We note that appending a bit to the encoding takes amortized constant time.

Lemma 4.2. *We can compute $\ell_1, \ldots, \ell_\sigma$ in one pass using $\mathcal{O}(Hn)$ bits of memory.*

Proof. Since we are not yet concerned with time, for each element in s we can simply perform a linear search — which is slow but uses no extra memory — through the entire list of distinct elements to find the encoding we should extend. Since an array is never more than twice the size of the encoding it holds, we use $\mathcal{O}(Hn)$ bits of memory for the arrays. □

To make our algorithm time-efficient, we use search in a splay tree [ST85] instead of linear search. At each node of the splay tree, we store a distinct element as the key, the position of that element's most recent occurrence and a pointer to the array for that element. For $1 \leq j \leq n$, we search for s_j in the splay tree; if we find it, then we extend the encoding of the corresponding list as described above, set the position of s_j's most recent occurrence to j and splay s_j's node to the root; if not, then we insert a new node storing s_j as its key, position j and a pointer to an expandable array storing $\gamma(j)$, and splay the node to the root. Figure 4.1 shows the state of our splay tree and arrays after we process the first 9 elements in our example; i.e., $a, b, r, a, c, a, d, a, b$. Figure 4.2 shows the changes when we process the next element, an r: we double the size of r's array from 4 to 8 bits in order to append $\gamma(10 - 3 = 7) = 00111$, set the position of r's most recent occurrence to 10 and splay r's node to the root. Figure 4.3 shows the final state of our splay tree and arrays after we process the last element, an a: we append $\gamma(11 - 8 = 3) = 011$ to a's array (but since only 10 of its 16 bits were already used, we do not expand it), set the position of a's most recent occurrence to 11 and splay a's node to the root.

Lemma 4.3. *We can compute $\ell_1, \ldots, \ell_\sigma$ in one pass using $\mathcal{O}((H+1)n)$ time and $\mathcal{O}(Hn)$ bits of memory.*

Proof. Our splay tree takes $\mathcal{O}(\sigma)$ words of memory and, so, does not change the bound on our memory usage. For $1 \leq i \leq \sigma$ we search for the ith largest distinct element once when it is not in the splay tree, insert it once, and search for it $n_i - 1$ times when it is in the splay tree. Therefore, by the Update Lemma [ST85] for

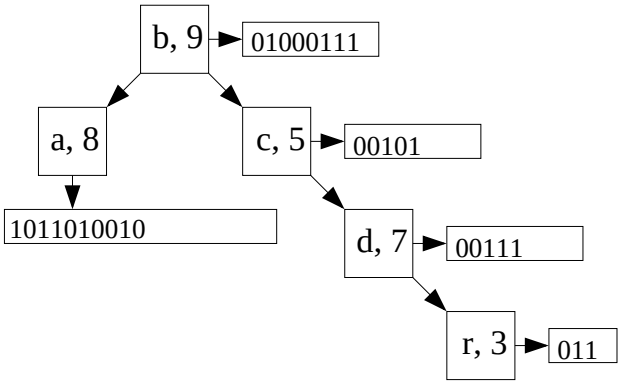

Figure 4.1: Our splay tree and arrays after we process $a, b, r, a, c, a, d, a, b$.

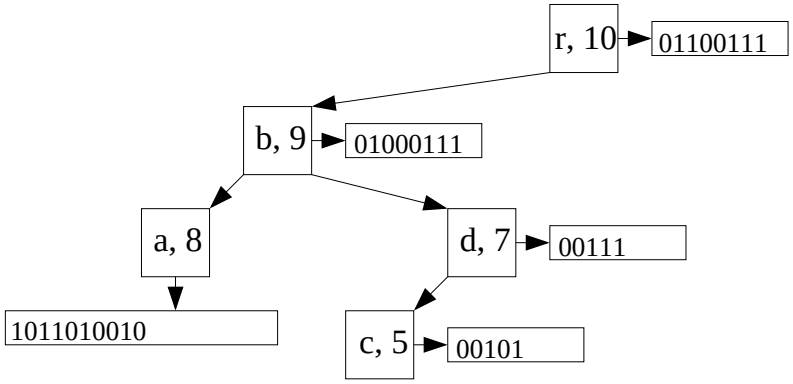

Figure 4.2: Our splay tree and arrays after we process $a, b, r, a, c, a, d, a, b, r$; notice we have doubled the size of the array for r, in order to append $\gamma(7) = 00111$.

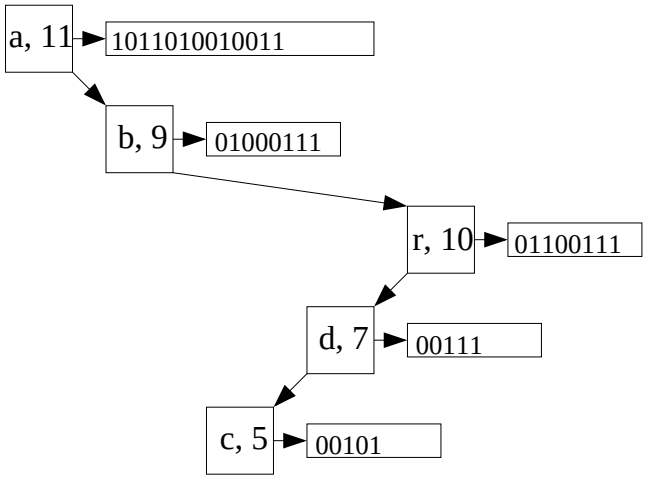

Figure 4.3: Our splay tree and arrays after we process $a, b, r, a, c, a, d, a, b, r, a$; notice we have not had to expand the array for a in order to append $\gamma(3) = 011$.

splay trees, the total time taken for all the operations on the splay tree is

$$\sum_{i=1}^{\sigma} \mathcal{O}\left(\log \frac{W}{\min(w_{i-1}, w_{i+1})} + n_i \log \frac{W}{w_i} + n_i\right) ,$$

where $w_1, \ldots, w_\sigma$ are any positive weights, W is their sum and $w_0 = w_{\sigma+1} = \infty$. Setting $w_i = n_i$ for $1 \leq i \leq \sigma$, this bound becomes

$$\sum_{i=1}^{\sigma} \mathcal{O}\big((n_i + 2)(\log(n/n_i) + 1)\big) = \mathcal{O}((H + 1)n) .$$

Because appending a bit to an array takes amortized constant time, the total time taken for operations on the arrays is proportional to the total length in bits of the encodings, i.e., $\mathcal{O}(Hn)$. □

After we process all of s, we can compute π from the state of the splay tree and arrays: we perform an in-order traversal of the splay tree; when we visit a node, we decode the numbers in its array and output their positive partial sums (this takes $\mathcal{O}(1)$ words of memory and time proportional to the length in bits of the encoding, because the gamma code is prefix-free); this way, we output the concatenation of the decoded lists in increasing order by element, i.e., $\ell_1 \cdots \ell_\sigma = \pi$. In our example, we visit the nodes in the order a, b, c, d, r; when we visit a's node we output

$$\begin{aligned} \gamma^{-1}(1) &= 1 \\ 1 + \gamma^{-1}(011) &= 1 + 3 = 4 \\ 4 + \gamma^{-1}(010) &= 4 + 2 = 6 \\ 6 + \gamma^{-1}(010) &= 6 + 2 = 8 \\ 8 + \gamma^{-1}(011) &= 8 + 3 = 11 . \end{aligned}$$

Our results in this section culminate in the following theorem:

Theorem 4.4. *We can compute π in one pass using $\mathcal{O}((H + 1)n)$ time and $\mathcal{O}(Hn)$ bits of memory.*

We note s can be recovered efficiently from our splay tree and arrays: we start with an empty priority queue Q and insert a copy of each distinct element, with priority equal to the position of its first occurrence (i.e., the first encoded number

in its array); for $1 \leq j \leq n$, we dequeue the element with minimum priority, output it, and reinsert it with priority equal to the position of its next occurrence (i.e., its previous priority plus the next encoded number in its array). This idea — that a sorted ordering of s partially encodes it — is central to our lower bound in the next section.

4.2 Lower bound

Consider any algorithm A that, allowed one pass over s, outputs a permutation ρ such that $s_{\rho(1)}, \ldots, s_{\rho(n)}$ is in sorted order (not necessarily stably-sorted). Notice A generally cannot output anything until it has read all of s, in case s_n is the unique minimum; also, given the frequency of each distinct element, ρ tells us the arrangement of elements in s up to equivalence.

Theorem 4.5. *In the worst case, it takes $\Omega(Hn)$ bits of memory to compute any sorted ordering of s in one pass.*

Proof. Suppose each $n_i = n/\sigma$, so $H = \log \sigma$ and the number of possible distinct arrangements of the elements in s is maximized,

$$\frac{n!}{\prod_{i=1}^{\sigma} n_i!} = \frac{n!}{((n/\sigma)!)^{\sigma}} .$$

It follows that in the worst case A uses at least

$$\begin{aligned}
&\log \left(n!/((n/\sigma)!)^{\sigma}\right) \\
&= \log n! - \sigma \log(n/\sigma)! \\
&\geq n \log n - n \log e - \sigma \left((n/\sigma) \log(n/\sigma) - (n/\sigma) \log e + \mathcal{O}(\log(n/\sigma))\right) \\
&= n \log \sigma - \mathcal{O}(\sigma \log(n/\sigma))
\end{aligned}$$

bits of memory to store ρ; the inequality holds by Stirling's Formula,

$$x \log x - x \log e < \log x! \leq x \log x - x \log e + \mathcal{O}(\log x) .$$

If $\sigma = \mathcal{O}(1)$ then

$$\sigma \log(n/\sigma) = \mathcal{O}(\log n) \subset o(n) ;$$

otherwise, since $\sigma \log(n/\sigma)$ is maximized when $\sigma = n/e$,

$$\sigma \log(n/\sigma) = \mathcal{O}(n) \subset o(n \log \sigma) ;$$

in both cases,

$$n \log \sigma - \mathcal{O}(\sigma \log(n/\sigma)) \geq n \log \sigma - o(n \log \sigma) \geq \Omega(Hn) . \qquad \square$$

Chapter 5

One-Pass Compression

Data compression has come of age in recent years and compression algorithms are now vital in situations unforeseen by their designers. This has led to a discrepancy between the theory of data compression algorithms and their use in practice: compression algorithms are often designed and analysed assuming the compression and decompression operations can use a "sufficiently large" amount of working memory; however, in some situations, particularly in mobile or embedded computing environments, the memory available is very small compared to the amount of data we need to compress or decompress. Even when compression algorithms are implemented to run on powerful desktop computers, some care is taken to be sure that the compression/decompression of large files do not take over all the RAM of the host machine. This is usually accomplished by splitting the input into blocks (e.g., bzip), using heuristics to determine when to discard the old data (e.g., compress, ppmd), or by maintaining a "sliding window" over the more recently seen data and forgetting the oldest data (e.g., gzip).

In this chapter we initiate the theoretical study of space-conscious compression algorithms. Although data compression algorithms have their own peculiarities, this study belongs to the general field of algorithmics in the streaming model (see, e.g., [BBD+02, Mut05]), in which we are allowed only one pass over the input and memory sublinear (possibly polylogarithmic or even constant) in its size. We prove tight upper and lower bounds on the compression ratio achievable by one-pass algorithms that use an amount of memory independent of the size of the input. By "one-pass", we mean that the algorithms are allowed to read each input symbol only once; hence, if an algorithm needs to access (portions of) the input more than once it must store it—consuming part of its precious working memory.

Our bounds are worst-case and given in terms of the empirical kth-order empirical entropy of the input string. More precisely we prove the following results:

(a) Let $\lambda \geq 1$, $k \geq 0$ and $\epsilon > 0$ be constants and let g be a function independent of n. In the worst case it is impossible to store a string s of length n over an alphabet of size σ in $\lambda H_k(s)n + o(n \log \sigma) + g$ bits using one pass and $\mathcal{O}\big(\sigma^{k+1/\lambda-\epsilon}\big)$ bits of memory.

(b) Given a $(\lambda H_k(s)+o(n \log \sigma)+g)$-bit encoding of s, it is impossible to recover s using one pass and $\mathcal{O}\big(\sigma^{k+1/\lambda-\epsilon}\big)$ bits of memory.

(c) Given $\lambda \geq 1$, $k \geq 0$ and $\mu > 0$, we can store s in $\lambda H_k(s)n + \mu n + \mathcal{O}\big(\sigma^{k+1/\lambda} \log \sigma\big)$ bits using one pass and $\mathcal{O}\big(\sigma^{k+1/\lambda} \log^2 \sigma\big)$ bits of memory, and later recover s using one pass and the same amount of memory.

While σ is often treated as constant in the literature, we treat it as a variable to distinguish between, say, $\mathcal{O}\big(\sigma^{k+1/\lambda-\epsilon}\big)$ and $\mathcal{O}\big(\sigma^{k+1/\lambda} \log^2 \sigma\big)$ bits. Informally, (a) provides a lower bound to the amount of memory needed to compress a string up to its kth-order entropy; (b) tells us the same amount of memory is required also for decompression and implies that the use of a powerful machine for doing the compression does not help if only limited memory is available when decompression takes place; (c) establishes that (a) and (b) are nearly tight. Notice λ plays a dual role: for large k, it makes (a) and (b) inapproximability results — e.g., we cannot use $\mathcal{O}\big(\sigma^k\big)$ bits of memory without worsening the compression in terms of $H_k(s)$ by more than a constant factor; for small k, it makes (c) an interesting approximability result — e.g., we can compress reasonably well in terms of $H_0(s)$ using, say, $\mathcal{O}(\sqrt{\sigma})$ bits of memory. The main difference between the bounds in (a)–(b) and (c) is a $\sigma^\epsilon \log^2 \sigma$ factor in the memory usage. Since μ is a constant, $\mu n \in o(n \log \sigma)$ and the bounds on the encoding's length match. Note that μ can be arbitrarily small, but the term μn cannot be avoided (Lemma 5.5).

We use s to denote the string we want to compress. We assume that s has length n and is drawn from an alphabet of size σ. Note that we measure memory in terms of alphabet size so σ is considered a variable. The 0th-order empirical entropy $H_0(s)$ of s is defined as $H_0(s) = \sum_a \frac{\mathrm{occ}(a,s)}{n} \log \frac{n}{\mathrm{occ}(a,s)}$, where $\mathrm{occ}\,(a, s)$ is the number of times character a occurs in s; throughout, we write log to mean $\log_2$ and assume $0 \log 0 = 0$. It is well known that H_0 is the maximum compression we can achieve using a fixed codeword for each alphabet symbol. We can achieve

a greater compression if the codeword we use for each symbol depends on the k symbols preceding it. In this case the maximum compression is bounded by the kth-order entropy $H_k(s)$ (see [KM99] for the formal definition). We use two properties of kth-order entropy in particular:

- $H_k(s_1)|s_1| + H_k(s_2)|s_2| \leq H_k(s_1 s_2)|s_1 s_2|$,
- since $H_0(s) \leq \log |\{a : a \text{ occurs in } s\}|$, we have

$$H_k(s) \leq \log \max_{|w|=k} \{j : w \text{ is followed by } j \text{ distinct characters in } s\}.$$

We point out that the empirical entropy is defined pointwise for any string and can be used to measure the performance of compression algorithms as a function of the string's structure, thus without any assumption on the input source. For this reason we say that the bounds given in terms of H_k are worst-case bounds.

Some of our arguments are based on Kolmogorov complexity [LV08]; the Kolmogorov complexity of s, denoted $K(s)$, is the length in bits of the shortest program that outputs s; it is generally incomputable but can be bounded from below by counting arguments (e.g., in a set of m elements, most have Kolmogorov complexity at least $\log m - \mathcal{O}(1)$). We use two properties of Kolmogorov complexity in particular: if an object can be easily computed from other objects, then its Kolmogorov complexity is at most the sum of theirs plus a constant; and a fixed, finite object has constant Kolmogorov complexity.

5.1 Algorithm

Move-to-front compression [BSTW86] is probably the best example of a compression algorithm whose space complexity is independent of the input length: keep a list of the characters that have occurred in decreasing order by recency; store each character in the input by outputting its position in the list (or, if it has not occurred before, its index in the alphabet) encoded in Elias' δ code [Eli75], then move it to the front of the list. Move-to-front stores a string s of length n over an alphabet of size σ in $\big(H_0(s) + \mathcal{O}(\log H_0(s))\big)\, n + \mathcal{O}(\sigma \log \sigma)$ bits using one pass and $\mathcal{O}(\sigma \log \sigma)$ bits of memory. When memory is scarce, we can use $\mathcal{O}\big(\sigma^{1/\lambda} \log \sigma\big)$ bits of memory by storing only the $\lceil \sigma^{1/\lambda} \rceil$ most recent characters; it is easy to see that this increases the number of bits stored by a factor of at most λ. On the

other hand, note that we can store s in $\big(H_k(s) + \mathcal{O}(\log H_k(s))\big)\, n + \mathcal{O}\big(\sigma^{k+1} \log \sigma\big)$ bits by keeping a separate list for each possible context of length k; this increases the memory usage by a factor of at most σ^k. In this section we first use a more complicated algorithm to get a better upper bound: given constants $\lambda \geq 1$, $k \geq 0$ and $\mu > 0$, we can store s in $(\lambda H_k(s) + \mu)n + \mathcal{O}\big(\sigma^{k+1/\lambda} \log \sigma\big)$ bits using one pass and $\mathcal{O}\big(\sigma^{k+1/\lambda} \log^2 \sigma\big)$ bits of memory.

We start with the following lemma — based on a previous paper [Gag06b] about compression algorithms' redundancies — that says we can store an approximation Q of a probability distribution P in few bits, so that the relative entropy between P and Q is small. The relative entropy $D(P\|Q) = \sum_{i=1}^{\sigma} p_i \log(p_i/q_i)$ between $P = p_1, \ldots, p_\sigma$ and $Q = q_1, \ldots, q_\sigma$ is the expected redundancy per character of an ideal code for Q when characters are drawn according to P.

Lemma 5.1. *Let s be a string of length n over an alphabet of size σ and let P be the normalized distribution of characters in s. Given s and constants $\lambda \geq 1$ and $\mu > 0$, we can store a probability distribution Q with $D(P\|Q) < (\lambda - 1)H(P) + \mu$ in $\mathcal{O}\big(\sigma^{1/\lambda} \log(n + \sigma)\big)$ bits using $\mathcal{O}\big(\sigma^{1/\lambda} \log(n + \sigma)\big)$ bits of memory.*

Proof. Suppose $P = p_1, \ldots, p_\sigma$. We can use an $\mathcal{O}(n \log n)$-time algorithm due to Misra and Gries [MG82] (see also [DLM02, KSP03]) to find the $t \leq r\sigma^{1/\lambda}$ values of i such that $p_i \geq 1/(r\sigma^{1/\lambda})$, where $r = 1 + \frac{1}{2^{\mu/2} - 1}$, using $\mathcal{O}\big(\sigma^{1/\lambda} \log \max(n, \sigma)\big)$ bits of memory; or, since we are not concerned with time in this chapter, we can simply make σ passes over s to find these t values. For each, we store i and $\lfloor p_i r^2 \sigma \rfloor$; since r depends only on μ, in total this takes $\mathcal{O}\big(\sigma^{1/\lambda} \log \sigma\big)$ bits. This information lets us later recover $Q = q_1, \ldots, q_\sigma$ where

$$q_i = \begin{cases} \dfrac{(1 - 1/r)\lfloor p_i r^2 \sigma \rfloor}{\sum \{\lfloor p_j r^2 \sigma \rfloor \,:\, p_j \geq 1/(r\sigma^{1/\lambda})\}} & \text{if } p_i \geq 1/(r\sigma^{1/\lambda}), \\ \dfrac{1}{r(\sigma - t)} & \text{otherwise.} \end{cases}$$

Suppose $p_i \geq \frac{1}{r\sigma^{1/\lambda}}$; then $p_i r^2 \sigma \geq r$. Since $\sum \big\{ \lfloor p_j r^2 \sigma \rfloor \,:\, p_j \geq \frac{r}{\sigma^{1/\lambda}} \big\} \leq r^2 \sigma$,

$$\begin{aligned} & p_i \log(p_i/q_i) \\ & \leq p_i \log \left(\frac{r}{r-1} \cdot \frac{p_i r^2 \sigma}{\lfloor p_i r^2 \sigma \rfloor} \right) \\ & < 2 p_i \log \frac{r}{r-1} \\ & = p_i \mu \,. \end{aligned}$$

Now suppose $p_i < 1/(r\sigma^{1/\lambda})$; then $p_i \log(1/p_i) > (p_i/\lambda)\log\sigma$. Therefore

$$\begin{aligned} & p_i \log(p_i/q_i) \\ & < p_i \log\left((\sigma - t)/\sigma^{1/\lambda}\right) \\ & \leq (\lambda - 1)(p_i/\lambda)\log\sigma \\ & < (\lambda - 1)p_i \log(1/p_i)\,. \end{aligned}$$

Since $p_i \log(p_i/q_i) < (\lambda - 1)p_i \log(1/p_i) + p_i\mu$ in both cases, $D(P\|Q) < (\lambda - 1)H(P) + \mu$. □

Armed with this lemma, we can adapt arithmetic coding to use $\mathcal{O}\big(\sigma^{1/\lambda}\log(n+\sigma)\big)$ bits of memory with a specified redundancy per character.

Lemma 5.2. *Given a string s of length n over an alphabet of size σ and constants $\lambda \geq 1$ and $\mu > 0$, we can store s in $(\lambda H_0(s) + \mu)n + \mathcal{O}\big(\sigma^{1/\lambda}\log(n+\sigma)\big)$ bits using $\mathcal{O}\big(\sigma^{1/\lambda}\log(n+\sigma)\big)$ bits of memory.*

Proof. Let P be the normalized distribution of characters in s, so $H(P) = H_0(s)$. First, as described in Lemma 5.1, we store a probability distribution Q with $D(P\|Q) < (\lambda-1)H(P) + \mu/2$ in $\mathcal{O}\big(\sigma^{1/\lambda}\log\sigma\big)$ bits using $\mathcal{O}\big(\sigma^{1/\lambda}\log(n+\lambda)\big)$ bits of memory. Then, we process s in blocks $s_1, \ldots, s_b$ of length $\lceil 4/\mu \rceil$ (except s_b may be shorter). For $1 \leq i < b$, we store s_i as the first $\lceil \log(2/\Pr[X = s_i]) \rceil$ bits to the right of the binary point in the binary representation of

$$\begin{aligned} f(s_i) &= \Pr[X < s_i] + \Pr[X = s_i]/2 \\ &= \sum_{j=1}^{\lceil 4/\mu \rceil} \Pr\Big[X[1] = s_i[1], \ldots, X[j-1] = s_i[j-1], X[j] < s_i[j]\Big] + \\ &\quad \Pr[X = s_i]/2\,, \end{aligned}$$

where X is a string of length $\lceil 4/\mu \rceil$ chosen randomly according to Q, $X < s_i$ means X is lexicographically less than s_i, and $X[j]$ and $s_i[j]$ indicate the indices in the alphabet of the jth characters of X and s_i, respectively. Notice that, since $|f(s_i) - f(y)| > \Pr[X = s_i]/2$ for any string $y \neq s_i$ of length $\lceil 4/\mu \rceil$, these bits uniquely identify $f(s_i)$ and, thus, s_i. Also, since the probabilities in Q are $\mathcal{O}(\log\sigma)$-bit numbers, we can compute $f(s_i)$ from s_i with $\mathcal{O}(\sigma)$ additions and $\mathcal{O}(1/\mu) = \mathcal{O}(1)$ multiplications using $\mathcal{O}(\log\sigma)$ bits of memory. (In fact, with appropriate data structures, $\mathcal{O}(\log\sigma)$ additions and $\mathcal{O}(1)$ multiplications suffice.)

Finally, we store s_b in $|s_b|\lceil \log \sigma \rceil = \mathcal{O}(\log \sigma)$ bits. In total we store s in

$$\begin{aligned}
&\sum_{i=1}^{b-1} \lceil \log(2/\Pr[X = s_i]) \rceil + \mathcal{O}\big(\sigma^{1/\lambda} \log \sigma\big) \\
&\leq \sum_{i=1}^{b-1} \left(\sum_{j=1}^{\lceil 4/\mu \rceil} \log(1/q_{s_i[j]}) + 2 \right) + \mathcal{O}\big(\sigma^{1/\lambda} \log \sigma\big) \\
&= n \sum_{i=1}^{\sigma} p_i \log(1/q_i) + 2(b-1) + \mathcal{O}\big(\sigma^{1/\lambda} \log \sigma\big) \\
&\leq n(D(P\|Q) + H(P)) + \mu n/2 + \mathcal{O}\big(\sigma^{1/\lambda} \log \sigma\big) \\
&\leq (\lambda H_0(s) + \mu)n + \mathcal{O}\big(\sigma^{1/\lambda} \log \sigma\big)
\end{aligned}$$

bits using $\mathcal{O}\big(\sigma^{1/\lambda} \log(n + \sigma)\big)$ bits of memory. □

We modify our space-conscious arithmetic coding algorithm to achieve a bound in terms of $H_k(s)$ instead of $H_0(s)$ by running a separate copy for each possible k-tuple, just as we modified move-to-front compression.

Lemma 5.3. *Given a string s of length n over an alphabet of size σ and constants $\lambda \geq 1$, $k \geq 0$ and $\mu > 0$, we can store s in $(\lambda H_k(s) + \mu)n + \mathcal{O}\big(\sigma^{k+1/\lambda} \log(n + \sigma)\big)$ bits using $\mathcal{O}\big(\sigma^{k+1/\lambda} \log(n + \sigma)\big)$ bits of memory.*

Proof. We store the first k characters of s in $\mathcal{O}(\log \sigma)$ bits then apply Lemma 5.2 to subsequences $s_1, \ldots, s_{\sigma^k}$, where s_i consists of the characters in s that immediately follow occurrences of the lexicographically ith possible k-tuple. Notice that although we cannot keep $s_1, \ldots, s_{\sigma^k}$ in memory, enumerating them as many times as necessary in order to apply Lemma 5.2 takes $\mathcal{O}(\log \sigma)$ bits of memory. □

To make our algorithm use one pass and to change the $\log(n + \sigma)$ factor to $\log \sigma$, we process the input in blocks $s_1, \ldots, s_b$ of length $\mathcal{O}\big(\sigma^{k+1/\lambda} \log \sigma\big)$. Notice each individual block s_i fits in memory — so we can apply Lemma 5.3 to it — and $\log(|s_i| + \sigma) = \mathcal{O}(\log \sigma)$.

Theorem 5.4. *Given a string s of length n over an alphabet of size σ and constants $\lambda \geq 1$, $k \geq 0$ and $\mu > 0$, we can store s in $(\lambda H_k(s) + \mu)n + \mathcal{O}\big(\sigma^{k+1/\lambda} \log \sigma\big)$ bits using one pass and $\mathcal{O}\big(\sigma^{k+1/\lambda} \log^2 \sigma\big)$ bits of memory, and later recover s using one pass and the same amount of memory.*

Proof. Let c be a constant such that, by Lemma 5.3, we can store any substring s_i of s in $(\lambda H_k(s_i) + \mu/2)|s_i| + c\sigma^{k+1/\lambda} \log \sigma$ bits using $\mathcal{O}\big(\sigma^{1/\lambda} \log(|s_i| + \sigma)\big)$ bits of memory. We process s in blocks $s_1, \ldots, s_b$ of length $\lceil (2c/\mu)\sigma^{k+1/\lambda} \log \sigma \rceil$ (except s_b may be shorter). Notice each block s_i fits in $\mathcal{O}\big(\sigma^{k+1/\lambda} \log^2 \sigma\big)$ bits of memory. When we reach s_i, we read it into memory, apply Lemma 5.3 to it — using

$$\mathcal{O}\big(\sigma^{k+1/\lambda} \log\big(\lceil (2c/\mu)\sigma^{k+1/\lambda} \log \sigma \rceil + \sigma\big)\big) = \mathcal{O}\big(\sigma^{k+1/\lambda} \log \sigma\big)$$

bits of memory — then erase it from memory. In total we store s in

$$\begin{aligned}
&\sum_{i=1}^{b} \big((\lambda H_k(s_i) + \mu/2)|s_i| + c\sigma^{k+1/\lambda} \log \sigma\big) \\
&\leq (\lambda H_k(s) + \mu/2)n + bc\sigma^{k+1/\lambda} \log \sigma \\
&\leq (\lambda H_k(s) + \mu)n + c\sigma^{k+1/\lambda} \log \sigma
\end{aligned}$$

bits using $\mathcal{O}\big(\sigma^{k+1/\lambda} \log^2 \sigma\big)$ bits of memory.

Notice the encoding of each block s_i also fits in $\mathcal{O}\big(\sigma^{k+1/\lambda} \log^2 \sigma\big)$ bits of memory. To decode each block later, we read its encoding into memory, search through all possible strings of length $\lceil (2c/\mu)\sigma^{k+1/\lambda} \log \sigma \rceil$ in lexicographic order until we find the one that yields that encoding — using $\mathcal{O}\big(\sigma^{k+1/\lambda} \log^2 \sigma\big)$ bits of memory — and output it. □

The method for decompression in the proof of Theorem 5.4 above takes exponential time but is very simple (recall we are not concerned with time here); reversing each step of the compression takes linear time but is slightly more complicated.

5.2 Lower bounds

Theorem 5.4 is still weaker than the strongest compression bounds that ignore memory constraints, in two important ways: first, even when $\lambda = 1$ the bound on the compression ratio does not approach $H_k(s)$ as n goes to infinity; second, we need to know k. It is not hard to prove these weaknesses are unavoidable when using fixed memory. In this section, we use the idea from these proofs to prove a nearly matching lower bound for compression: in the worst case it is impossible to store a string s of length n over an alphabet of size σ in $\lambda H_k(s)n + o(n \log \sigma) + g$

bits, for any function g independent of n, using one encoding pass and $\mathcal{O}\big(\sigma^{k+1/\lambda-\epsilon}\big)$ bits of memory. We close with a symmetric lower bound for decompression.

Lemma 5.5. *Let $\lambda \geq 1$ be a constant and let g be a function independent of n. In the worst case it is impossible to store a string s of length n in $\lambda H_0(s)n + o(n) + g$ bits using one encoding pass and memory independent of n.*

Proof. Let A be an algorithm that, given λ, stores s using one pass and memory independent of n. Since A's future output depends only on its state and its future input, we can model A with a finite-state machine M. While reading $|M|$ characters of s, M must visit some state at least twice; therefore either M outputs at least one bit for every $|M|$ characters in s — or $n/|M|$ bits in total — or for infinitely many strings M outputs nothing. If s is unary, however, then $H_0(s) = 0$. □

Lemma 5.6. *Let λ be a constant, let g be a function independent of n and let b be a function independent of n and k. In the worst case it is impossible to store a string s of length n over an alphabet of size σ in $\lambda H_k(s)n + o(n \log \sigma) + g$ bits for all $k \geq 0$ using one pass and b bits of memory.*

Proof. Let A be an algorithm that, given λ, g, b and σ, stores s using b bits of memory. Again, we can model it with a finite-state machine M, with $|M| = 2^b$ and M's Kolmogorov complexity $K(M) = K(\langle A, \lambda, g, b, \sigma\rangle) + \mathcal{O}(1) = \mathcal{O}(\log \sigma)$. (Since A, λ, g, and b are all fixed, their Kolmogorov complexities are $\mathcal{O}(1)$.)

Suppose s is a periodic string with period $2b$ whose repeated substring r has $K(r) = |r| \log \sigma - \mathcal{O}(1)$. We can specify r by specifying M, the states M is in when it reaches and leaves any copy of r in s, and M's output on that copy of r. (If there were another string r' that took M between those states with that output, then we could substitute r' for r in s without changing M's output.) Therefore M outputs at least

$$K(r) - K(M) - \mathcal{O}(\log |M|) = |r| \log \sigma - \mathcal{O}(\log \sigma + b) = \Omega(|r| \log \sigma)$$

bits for each copy of r in s, or $\Omega(n \log \sigma)$ bits in total. For $k \geq 2b$, however, $H_k(s)$ approaches 0 as n goes to infinity. □

The idea behind these proofs is simple: model a one-pass algorithm with a finite-state machine and evaluate its behaviour on a periodic string. Nevertheless,

combining it with the following simple results — based on the same previous paper [Gag06b] as Lemma 5.1 — we can easily show a lower bound that nearly matches Theorem 5.4. (In fact, our proofs are valid even for algorithms that make preliminary passes that produce no output — perhaps to gather statistics, like Huffman coding [Huf52] — followed by a single encoding pass that produces all of the output; once the algorithm begins the encoding pass, we can model it with a finite-state machine.)

Lemma 5.7. *Let $\lambda \geq 1$, $k \geq 0$ and $\epsilon > 0$ be constants and let r be a randomly chosen string of length $\lfloor \sigma^{k+1/\lambda-\epsilon} \rfloor$ over an alphabet of size σ. With high probability every possible k-tuple is followed by $\mathcal{O}\big(\sigma^{1/\lambda-\epsilon}\big)$ distinct characters in r.*

Proof. Consider a k-tuple w. For $1 \leq i \leq n-k$, let $X_i = 1$ if the ith through $(i+k-1)$st characters of s are an occurrence of w and the $(i+k)$th character in s does not occur in w; otherwise $X_i = 0$. Notice w is followed by at most $\sum_{i=1}^{n-k} X_i + k$ distinct characters in s and $\Pr[X_i = 1 \,|\, X_j = 1] \leq 1/\sigma^k$ and $\Pr[X_i = 1 \,|\, X_j = 0] \leq 1/(\sigma^k - 1)$ for $i \neq j$. Therefore, by Chernoff bounds (see [HR90]) and the union bound, with probability greater than

$$1 - \frac{\sigma^k}{2^{6\lfloor \sigma^{k+1/\lambda-\epsilon} \rfloor / (\sigma^k - 1)}} \geq 1 - \sigma^k / 2^{6\sigma^{1/\lambda-\epsilon}}$$

every k-tuple is followed by fewer than $6\lfloor \sigma^{k+1/\lambda-\epsilon} \rfloor / (\sigma^k - 1) + k \leq 12\sigma^{1/\lambda-\epsilon} + k$ distinct characters. □

Corollary 5.8. *Let $\lambda \geq 1$, $k \geq 0$ and $\epsilon > 0$ be constants. There exists a string r of length $\lfloor \sigma^{k+1/\lambda-\epsilon} \rfloor$ over an alphabet of size σ with $K(r) = |r| \log \sigma - \mathcal{O}(1)$ but $H_k(r^i) \leq (1/\lambda - \epsilon) \log \sigma + \mathcal{O}(1)$ for $i \geq 1$.*

Proof. If r is randomly chosen, then $K(r) \geq |r| \log \sigma - 1$ with probability greater than $1/2$ and, by Lemma 5.7, with high probability every possible k-tuple is followed by $\mathcal{O}\big(\sigma^{1/\lambda-\epsilon}\big)$ distinct characters in r; therefore there exists an r with both properties. Every possible k-tuple is followed by at most k more distinct characters in r^i than in r and, thus,

$$\begin{aligned} H_k(r^i) &\leq \log \max_{|w|=k} \left\{ j \,:\, \begin{array}{l} w \text{ is followed by } j \\ \text{distinct characters in } r^i \end{array} \right\} \\ &\leq \log \mathcal{O}\big(\sigma^{1/\lambda-\epsilon}\big) \\ &\leq (1/\lambda - \epsilon) \log \sigma + \mathcal{O}(1) \ . \end{aligned}$$

□

Consider what we get if, for some $\epsilon > 0$, we allow the algorithm A from Lemma 5.6 to use $\mathcal{O}\big(\sigma^{k+1/\lambda-\epsilon}\big)$ bits of memory, and evaluate it on the periodic string r^i from Corollary 5.8. Since r^i has period $\lfloor\sigma^{k+1/\lambda-\epsilon}\rfloor$ and its repeated substring r has $K(r) = |r|\log\sigma - \mathcal{O}(1)$, the finite-state machine M outputs at least

$$K(r) - K(M) - \mathcal{O}(\log|M|) = |r|\log\sigma - \mathcal{O}\big(\sigma^{k+1/\lambda-\epsilon}\big) = |r|\log\sigma - \mathcal{O}(|r|)$$

bits for each copy of r in r^i, or $n\log\sigma - \mathcal{O}(n)$ bits in total. Because $\lambda H_k(r^i) \leq (1-\epsilon)\log\sigma + \mathcal{O}(1)$, this yields the following nearly tight lower bound; notice it matches Theorem 5.4 except for a $\sigma^\epsilon\log^2\sigma$ factor in the memory usage.

Theorem 5.9. *Let $\lambda \geq 1$, $k \geq 0$ and $\epsilon > 0$ be constants and let g be a function independent of n. In the worst case it is impossible to store a string s of length n over an alphabet of size σ in $\lambda H_k(s)n + o(n\log\sigma) + g$ bits using one encoding pass and $\mathcal{O}\big(\sigma^{k+1/\lambda-\epsilon}\big)$ bits of memory.*

Proof. Let A be an algorithm that, given λ, k, ϵ and σ, stores s while using one encoding pass and $\mathcal{O}\big(\sigma^{k+1/\lambda-\epsilon}\big)$ bits of memory; we prove that in the worst case A stores s in more than $(\lambda H_k(s)+\mu)n + o(n\log\sigma) + g$ bits. Again, we can model it with a finite-state machine M, with $|M| = 2^{\mathcal{O}\left(\sigma^{k+1/\lambda-\epsilon}\right)}$ and $K(M) = \mathcal{O}(\log\sigma)$. Let r be a string of length $\lfloor\sigma^{k+1/\lambda-\epsilon}\rfloor$ with $K(r) \geq |r|\log\sigma - \mathcal{O}(1)$ and $H_k(r^i) \leq (1/\lambda-\epsilon)\log\sigma + \mathcal{O}(1)$ for $i \geq 1$, as described in Corollary 5.8, and suppose $s = r^i$ for some i. We can specify r by specifying M, the states M is in when it reaches and leaves any copy of r in s, and M's output on that copy. Therefore M outputs at least

$$K(r) - K(M) - \mathcal{O}\big(\sigma^{k+1/\lambda-\epsilon}\big) = |r|\log\sigma - \mathcal{O}(|r|)$$

bits for each copy of r in s, or $n\log\sigma - \mathcal{O}(n)$ bits in total — which is asymptotically greater than $\lambda H_k(s)n + o(n\log\sigma) + g \leq (1-\epsilon)n\log\sigma + o(n\log\sigma) + g$. □

With a good bound on how much memory is needed for compression, we turn our attention to decompression. Good bounds here are equally important, because often data is compressed once by a powerful machine (e.g., a server or base-station) and then transmitted to many weaker machines (clients or agents) who decompress it individually. Fortunately for us, compression and decompression are essentially symmetric. Recall Theorem 5.4 says we can recover s from a $\big(\lambda H_k(s)+\mu)n + \mathcal{O}\big(\sigma^{k+1/\lambda}\log\sigma\big)\big)$-bit encoding using $\mathcal{O}\big(\sigma^{k+1/\lambda}\log^2\sigma\big)$ bits of

memory and one pass. Using the same idea about finite-state machines and periodic strings gives us the following nearly matching lower bound:

Theorem 5.10. *Let $\lambda \geq 1$, $k \geq 0$ and $\epsilon > 0$ be constants and let g be a function independent of n. There exists a string s of length n over an alphabet of size σ such that, given a $(\lambda H_k(s)n + o(n \log \sigma) + g)$-bit encoding of s, it is impossible to recover s using one pass and $\mathcal{O}\big(\sigma^{k+1/\lambda-\epsilon}\big)$ bits of memory.*

Proof. Let r be a string of length $\lfloor \sigma^{k+1/\lambda-\epsilon} \rfloor$ with $K(r) = |r| \log \sigma - \mathcal{O}(1)$ but $H_k(r^i) \leq (1/\lambda - \epsilon) \log \sigma + \mathcal{O}(1)$ for $i \geq 1$, as described in Corollary 5.8, and suppose $s = r^i$ for some i. Let A be an algorithm that, given λ, k, ϵ, σ and a $(\lambda H_k(s)n + o(n \log \sigma) + g)$-bit encoding of s, recovers s using one pass; we prove A uses $\omega(\sigma^{k+1/\lambda-\epsilon})$ bits of memory. Again, we can model A with a finite-state machine M, with $\log |M|$ equal to the number of bits of memory A uses and $K(M) = \mathcal{O}(\log \sigma)$. We can specify r by specifying M, the state M is in when it starts outputting any copy of r in s, and the bits of the encoding it reads while outputting that copy of r; therefore

$$\begin{aligned} K(r) &\leq K(M) + \mathcal{O}(\log |M|) + \big(\lambda H_k(s)n + o(n \log \sigma) + g\big)/i \\ &\leq \mathcal{O}(\log \sigma) + \mathcal{O}(\log |M|) + |r| \big((1-\epsilon) \log \sigma + o(\log \sigma) + g/n\big) \\ &\leq (1-\epsilon)|r| \log \sigma + o(|r| \log \sigma) + \mathcal{O}(\log |M|) + g/n\,, \end{aligned}$$

so

$$\mathcal{O}(\log |M|) + g/n \geq \epsilon |r| \log \sigma - o(|r| \log \sigma) = \Omega(\sigma^{k+1/\lambda-\epsilon} \log \sigma)\ .$$

The theorem follows because n can be arbitrarily large compared to g. □

Chapter 6

Stream Compression

Massive datasets seem to expand to fill the space available and, in situations when they no longer fit in memory and must be stored on disk, we may need new models and algorithms. Grohe and Schweikardt [GS05] introduced read/write streams to model situations in which one wants to process data using mainly sequential accesses to one or more disks. As the name suggests, this model is like the streaming model (see, e.g., [Mut05]) but, as is reasonable with datasets stored on disk, it allows us to make multiple passes over the data, change them and even use multiple streams (i.e., disks). As Grohe and Schweikardt pointed out, sequential disk accesses are much faster than random accesses — potentially bypassing the von Neumann bottleneck — and using several disks in parallel can greatly reduce the amount of memory and the number of accesses needed. For example, when sorting, we need the product of the memory and accesses to be at least linear when we use one disk [MP80, GKS07] but only polylogarithmic when we use two [CY91, GS05]. Similar bounds have been proven for a number of other problems, such as checking set disjointness or equality; we refer readers to Schweikardt's survey [Sch07] of upper and lower bounds with one or more read/write streams, Heinrich and Schweikardt's recent paper [HS08] relating read/write streams to classical complexity theory, and Beame and Huỳnh-Ngọc's recent paper [BH08] on the value of multiple read/write streams for approximating frequency moments.

Since sorting is an important operation in some of the most powerful data compression algorithms, and compression is an important operation for reducing massive datasets to a more manageable size, we wondered whether extra streams could also help us achieve better compression. In this chapter we consider the problem of compressing a string s of n characters over an alphabet of size σ when

we are restricted to using $\log^{\mathcal{O}(1)} n$ bits of memory and $\log^{\mathcal{O}(1)} n$ passes over the data. In Section 6.1, we show how we can achieve universal compression using only one pass over one stream. Our approach is to break the string into blocks and compress each block separately, similar to what is done in practice to compress large files. Although this may not usually significantly worsen the compression itself, it may stop us from then building a fast compressed index [FMMN07] (unless we somehow combine the indexes for the blocks) or clustering by compression [CV05, FGG+07] (since concatenating files should not help us compress them better if we then break them into pieces again). In Section 6.2 we use a vaguely automata-theoretic argument to show one stream is not sufficient for us to achieve good grammar-based compression. Of course, by "good" we mean here something stronger than universal compression: we want the size of our encoding to be at most polynomial in the size of the smallest context-free grammar that generates s and only s. We still do not know whether any constant number of streams is sufficient for us to achieve such compression. Finally, in Section 6.3 we show that two streams are necessary and sufficient for us to achieve entropy-only bounds. Along the way, we show we need two streams to find strings' minimum periods or to compute the Burrows-Wheeler Transform. As far as we know, this is the first study of compression with read/write streams, and among the first studies of compression in any streaming model; we hope the techniques we use will prove to be of independent interest.

6.1 Universal compression

An algorithm is called universal with respect to a class of sources if, when a string is drawn from any of those sources, the algorithm's redundancy per character approaches 0 with probability 1 as the length of the string grows. The class most often considered, and which we consider in this section, is that of stationary, ergodic Markov sources (see, e.g., [CT06]). Since the kth-order empirical entropy $H_k(s)$ of s is the minimum self-information per character of s with respect to a kth-order Markov source (see [Sav97]), an algorithm is universal if it stores any string s in $nH_k(s) + o(n)$ bits for any fixed σ and k. The kth-order empirical entropy of s is also our expected uncertainty about a randomly chosen character

of s when given the k preceding characters. Specifically,

$$H_k(s) = \begin{cases} (1/n) \sum_a \mathsf{occ}(a, s) \log \frac{n}{\mathsf{occ}(a,s)} & \text{if } k = 0, \\ (1/n) \sum_{|w|=k} |w_s| H_0(w_s) & \text{otherwise,} \end{cases}$$

where $\mathsf{occ}(a, s)$ is the number of times character a occurs in s, and w_s is the concatenation of those characters immediately following occurrences of k-tuple w in s.

In a previous paper [GM07b] we showed how to modify the well-known LZ77 compression algorithm [ZL77] to use sublinear memory while still storing s in $nH_k(s)+\mathcal{O}(n \log \log n/ \log n)$ bits for any fixed σ and k. Our algorithm uses nearly linear memory and so does not fit into the model we consider in this chapter, but we mention it here because it fits into some other streaming models (see, e.g., [Mut05]) and, as far as we know, was the first compression algorithm to do so. In the same paper we proved several lower bounds using ideas that eventually led to our lower bounds in Sections 6.2 and 6.3 of this chapter.

Theorem 6.1 (Gagie and Manzini, 2007)**.** *We can achieve universal compression using one pass over one stream and $\mathcal{O}\big(n/ \log^2 n\big)$ bits of memory.*

To achieve universal compression with only polylogarithmic memory, we use a recent algorithm due to Gupta, Grossi and Vitter [GGV08]. Although they designed it for the RAM model, we can easily turn it into a streaming algorithm by processing s in small blocks and compressing each block separately.

Theorem 6.2 (Gupta, Grossi and Vitter, 2008)**.** *In the RAM model, we can store any string s in $nH_k(s) + \mathcal{O}\big(\sigma^k \log n\big)$ bits, for all k simultaneously, using $\mathcal{O}(n)$ time.*

Corollary 6.3. *We can achieve universal compression using one pass over one stream and $\mathcal{O}\big(\log^{1+\epsilon} n\big)$ bits of memory.*

Proof. We process s in blocks of $\lceil \log^{\epsilon} n \rceil$ characters, as follows: we read each block into memory, apply Theorem 6.2 to it, output the result, empty the memory, and move on to the next block. (If n is not given in advance, we increase the block size as we read more characters.) Since Gupta, Grossi and Vitter's algorithm uses $\mathcal{O}(n)$ time in the RAM model, it uses $\mathcal{O}(n \log n)$ bits of memory and we use

$\mathcal{O}\big(\log^{1+\epsilon} n\big)$ bits of memory. If the blocks are $s_1, \ldots, s_b$, then we store all of them in a total of

$$\sum_{i=1}^{b} \big(|s_i| H_k(s_i) + \mathcal{O}\big(\sigma^k \log\log n\big)\big) \leq nH_k(s) + \mathcal{O}\big(\sigma^k n \log\log n / \log^{\epsilon} n\big)$$

bits for all k simultaneously. Therefore, for any fixed σ and k, we store s in $nH_k(s) + o(n)$ bits. □

A bound of $nH_k(s)+\mathcal{O}\big(\sigma^k n \log\log n / \log^{\epsilon} n\big)$ bits is not very meaningful when k is not fixed and grows as fast as $\log\log n$, because the second term is $\omega(n)$. Notice, however, that Gupta *et al.*'s bound of $nH_k(s) + \mathcal{O}\big(\sigma^k \log n\big)$ bits is also not very meaningful when $k \geq \log n$, for the same reason. As we will see in Section 6.3, it is possible for s to be fairly incompressible but still to have $H_k(s) = 0$ for $k \geq \log n$. It follows that, although we can prove bounds that hold for all k simultaneously, those bounds cannot guarantee good compression in terms of $H_k(s)$ when $k \geq \log n$.

By using larger blocks — and, thus, more memory — we can reduce the $\mathcal{O}\big(\sigma^k n \log\log n / \log^{\epsilon} n\big)$ redundancy term in our analysis, allowing k to grow faster than $\log\log n$ while still having a meaningful bound. We conjecture that the resulting tradeoff is nearly optimal. Specifically, using an argument similar to those we use to prove the lower bounds in Sections 6.2 and 6.3, we believe we can prove that the product of the memory, passes and redundancy must be nearly linear in n. It is not clear to us, however, whether we can modify Corollary 6.3 to take advantage of multiple passes.

Open Problem 6.4. *With multiple passes over one stream, can we achieve better bounds on the memory and redundancy than we can with one pass?*

6.2 Grammar-based compression

Charikar *et al.* [CLL+05] and Rytter [Ryt03] independently showed how to build a context-free grammar APPROX that generates s and only s and is an $\mathcal{O}(\log n)$ factor larger than the smallest such grammar OPT, which is $\Omega(\log n)$ bits in size.

Theorem 6.5 (Charikar *et al.*, 2005; Rytter, 2003)**.** *In the RAM model, we can approximate the smallest grammar with* $|\mathsf{APPROX}| = \mathcal{O}(|\mathsf{OPT}|^2)$ *using* $\mathcal{O}(n)$ *time.*

In this section we prove that, if we use only one stream, then in general our approximation must be superpolynomially larger than the smallest grammar. Our idea is to show that periodic strings whose periods are asymptotically slightly larger than the product of the memory and passes, can be encoded as small grammars but, in general, cannot be compressed well by algorithms that use only one stream. Our argument is based on the following two lemmas.

Lemma 6.6. *If s has period ℓ, then the size of the smallest grammar for that string is $\mathcal{O}(\ell + \log n \log\log n)$ bits.*

Proof. Let t be the repeated substring and t' be the proper prefix of t such that $s = t^{\lfloor n/\ell \rfloor} t'$. We can encode a unary string $X^{\lfloor n/\ell \rfloor}$ as a grammar G_1 with $\mathcal{O}(\log n)$ productions of total size $\mathcal{O}(\log n \log\log n)$ bits. We can also encode t and t' as grammars G_2 and G_3 with $\mathcal{O}(\ell)$ productions of total size $\mathcal{O}(\ell)$ bits. Suppose S_1, S_2 and S_3 are the start symbols of G_1, G_2 and G_3, respectively. By combining those grammars and adding the productions $S_0 \to S_1 S_3$ and $X \to S_2$, we obtain a grammar with $\mathcal{O}(\ell + \log n)$ productions of total size $\mathcal{O}(\ell + \log n \log\log n)$ bits that maps S_0 to s. □

Lemma 6.7. *Consider a lossless compression algorithm that uses only one stream, and a machine performing that algorithm. We can compute any substring from*

- *its length;*
- *for each pass, the machine's memory configurations when it reaches and leaves the part of the stream that initially holds that substring;*
- *all the output the machine produces while over that part.*

Proof. Let t be the substring and assume, for the sake of a contradiction, that there exists another substring t' with the same length that takes the machine between the same configurations while producing the same output. Then we can substitute t' for t in s without changing the machine's complete output, contrary to our specification that the compression be lossless. □

Lemma 6.7 implies that, for any substring, the size of the output the machine produces while over the part of the stream that initially holds that substring, plus twice the product of the memory and passes (i.e., the number of bits needed to store the memory configurations), must be at least that substring's complexity.

Therefore, if a substring is not compressible by more than a constant factor (as is the case for most strings) and asymptotically larger than the product of the memory and passes, then the size of the output for that substring must be at least proportional to the substring's length. In other words, the algorithm cannot take full advantage of similarities between substrings to achieve better compression. In particular, if s is periodic with a period that is asymptotically slightly larger than the product of the memory and passes, and s's repeated substring is not compressible by more than a constant factor, then the algorithm's complete output must be $\Omega(n)$ bits. By Lemma 6.6, however, the size of the smallest grammar that generates s and only s is bounded in terms of the period.

Theorem 6.8. *With one stream, we cannot approximate the smallest grammar with* $|\mathsf{APPROX}| \leq |\mathsf{OPT}|^{\mathcal{O}(1)}$.

Proof. Suppose an algorithm uses only one stream, m bits of memory and p passes to compress s, with $mp = \log^{\mathcal{O}(1)} n$, and consider a machine performing that algorithm. Furthermore, suppose s is periodic with period $\lceil mp \log n \rceil$ and its repeated substring t is not compressible by more than a constant factor. Lemma 6.7 implies that the machine's output while over a part of the stream that initially holds a copy of t, must be $\Omega(mp \log n - mp) = \Omega(mp \log n)$. Therefore, the machine's complete output must be $\Omega(n)$ bits. By Lemma 6.6, however, the size of the smallest grammar that generates s and only s is $\mathcal{O}(mp \log n + \log n \log \log n) \subset \log^{\mathcal{O}(1)} n$ bits. Since $n = \log^{\omega(1)} n$, the algorithm's complete output is superpolynomially larger than the smallest grammar. □

As an aside, we note that a symmetric argument shows that, with only one stream, in general we cannot decode a string encoded as a small grammar. To prove this, instead of considering a part of the stream that initially holds a copy of the repeated substring t, we consider a part that is initially blank and eventually holds a copy of t. We can compute t from the machine's memory configurations when it reaches and leaves that part, so the product of the memory and passes must be greater than or equal to t's complexity. Also, we note that Theorem 6.8 has the following corollary, which may be of independent interest.

Corollary 6.9. *With one stream, we cannot find strings' minimum periods.*

Proof. Consider the proof of Theorem 6.8. If we could find s's minimum period, then we could store s in $\log^{\mathcal{O}(1)} n$ bits by writing n and one copy of its repeated substring t. □

We are currently working on a more detailed argument to show that we cannot even check whether a string has a given period. Unfortunately, as we noted earlier, our results for this section are still incomplete, as we do not know whether multiple streams are helpful for grammar-based compression.

Open Problem 6.10. *With $\mathcal{O}(1)$ streams, can we approximate the smallest grammar well?*

6.3 Entropy-only bounds

Kosaraju and Manzini [KM99] pointed out that proving an algorithm universal does not necessarily tell us much about how it behaves on low-entropy strings. In other words, showing that an algorithm encodes s in $nH_k(s)+o(n)$ bits is not very informative when $nH_k(s) = o(n)$. For example, although the well-known LZ78 compression algorithm [ZL78] is universal, $|\mathsf{LZ78}(1^n) = \Omega(\sqrt{n})$ while $nH_0(1^n) = 0$. To analyze how algorithms perform on low-entropy strings, we would like to get rid of the $o(n)$ term and prove bounds that depend only on $nH_k(s)$. Unfortunately, this is impossible since, as the example above shows, even $nH_0(s)$ can be 0 for arbitrarily long strings.

It is not hard to show that only unary strings have $H_0(s) = 0$. For $k \geq 1$, recall that $H_k(s) = (1/n)\sum_{|w|=k} |w_s|H_0(w_s)$. Therefore, $H_k(s) = 0$ if and only if each distinct k-tuple w in s is always followed by the same distinct character. This is because, if a w is always followed by the same distinct character, then w_s is unary, $H_0(w_s) = 0$ and w contributes nothing to the sum in the formula. Manzini [Man01] defined the kth-order modified empirical entropy $H_k^*(s)$ such that each context w contributes at least $\lfloor \log |w_s| \rfloor + 1$ to the sum. Because modified empirical entropy is more complicated than empirical entropy — e.g., it allows for variable-length contexts — we refer readers to Manzini's paper for the full definition. In our proofs in this chapter, we use only the fact that

$$nH_k(s) \leq nH_k^*(s) \leq nH_k(s) + \mathcal{O}\big(\sigma^k \log n\big) \ .$$

Manzini showed that, for some algorithms and all k simultaneously, it is possible to bound the encoding's length in terms of only $nH_k^*(s)$ and a constant that depends only on σ and k; he called such bounds "entropy-only". In particular, he showed that an algorithm based on the Burrows-Wheeler Transform

(BWT) [BW94] stores any string s in at most $(5+\epsilon)nH_k^*(s) + \log n + g_k$ bits for all k simultaneously (since $nH_k^*(s) \geq \log(n-k)$, we could remove the $\log n$ term by adding 1 to the coefficient $5+\epsilon$).

Theorem 6.11 (Manzini, 2001)**.** *Using the BWT, move-to-front coding, run-length coding and arithmetic coding, we can achieve an entropy-only bound.*

The BWT sorts the characters in a string into the lexicographical order of the suffixes that immediately follow them. When using the BWT for compression, it is customary to append a special character \$ that is lexicographically less than any in the alphabet. For a more thorough description of the BWT, we again refer readers to Manzini's paper. In this section we first show how we can compute and invert the BWT with two streams and, thus, achieve entropy-only bounds. We then show that we cannot achieve entropy-only bounds with only one stream. In other words, two streams are necessary and sufficient for us to achieve entropy-only bounds.

One of the most common ways to compute the BWT is by building a suffix array. In his PhD thesis, Ruhl introduced the StreamSort model [Ruh03, ADRR04], which is similar to the read/write streams model with one stream, except that it has an extra primitive that sorts the stream in one pass. Among other things, he showed how to build a suffix array efficiently in this model.

Theorem 6.12 (Ruhl, 2003)**.** *In the StreamSort model, we can build a suffix array using $\mathcal{O}(\log n)$ bits of memory and $\mathcal{O}(\log n)$ passes.*

Corollary 6.13. *With two streams, we can compute the BWT using $\mathcal{O}(\log n)$ bits of memory and $\mathcal{O}\big(\log^2 n\big)$ passes.*

Proof. We can compute the BWT in the StreamSort model by appending \$ to s, building a suffix array, and replacing each value i in the array by the $(i-1)$st character in s (replacing either 0 or 1 by \$, depending on where we start counting). This takes $\mathcal{O}(\log n)$ bits of memory and $\mathcal{O}(\log n)$ passes. Since we can sort with two streams using $\mathcal{O}(\log n)$ bits memory and $\mathcal{O}(\log n)$ passes (see, e.g., [Sch07]), it follows that we can compute the BWT using $\mathcal{O}(\log n)$ bits of memory and $\mathcal{O}\big(\log^2 n\big)$ passes. □

Now suppose we are given a permutation π on $n+1$ elements as a list $\pi(1), \ldots, \pi(n+1)$, and asked to rank it, i.e., to compute the list $\pi^0(1), \ldots, \pi^n(1)$. This

problem is a special case of list ranking (see, e.g., [ABD+07]) and has a surprisingly long history. For example, Knuth [Knu98, Solution 24] described an algorithm, which he attributed to Hardy, for ranking a permutation with two tapes. More recently, Bird and Mu [BM04] showed how to invert the BWT by ranking a permutation. Therefore, reinterpreting Hardy's result in terms of the read/write streams model gives us the following bounds.

Theorem 6.14 (Hardy, c. 1967)**.** *With two streams, we can rank a permutation using $\mathcal{O}(\log n)$ bits of memory and $\mathcal{O}\big(\log^2 n\big)$ passes.*

Corollary 6.15. *With two streams, we can invert the BWT using $\mathcal{O}(\log n)$ bits of memory and $\mathcal{O}\big(\log^2 n\big)$ passes.*

Proof. The BWT has the property that, if a character is the ith in $\mathsf{BWT}(s)$, then its successor in s is the lexicographically ith in $\mathsf{BWT}(s)$ (breaking ties by order of appearance). Therefore, we can invert the BWT by replacing each character by its lexicographic rank, ranking the resulting permutation, replacing each value i by the ith character of $\mathsf{BWT}(s)$, and rotating the string until \$ is at the end. This takes $\mathcal{O}(\log n)$ memory and $\mathcal{O}\big(\log^2 n\big)$ passes. □

Since we can compute and invert move-to-front, run-length and arithmetic coding using $\mathcal{O}(\log n)$ bits of memory and $\mathcal{O}(1)$ passes over one stream, by combining Theorem 6.11 and Corollaries 6.13 and 6.15 we obtain the following theorem.

Theorem 6.16. *With two streams, we can achieve an entropy-only bound using $\mathcal{O}(\log n)$ bits of memory and $\mathcal{O}\big(\log^2 n\big)$ passes.*

To show we need at least two streams to achieve entropy-only bounds, we use De Bruijn cycles in a proof similar to the one for Theorem 6.8. A kth-order De Bruijn cycle [dB46] is a cyclic sequence in which every possible k-tuple appears exactly once. For example, Figure 6.1 shows a 3rd-order and a 4th-order De Bruijn cycle. (We need consider only binary De Bruijn cycles.) Our argument this time is based on Lemma 6.7 and the following results about De Bruijn cycles.

Lemma 6.17. *If $s \in d^*$ for some kth-order De Bruijn cycle d, then $nH_k^*(s) = \mathcal{O}\big(2^k \log n\big)$.*

Proof. By definition, each distinct k-tuple is always followed by the same distinct character; therefore, $nH_k(s) = 0$ and $nH_k^*(s) = \mathcal{O}\big(2^k \log n\big)$. □

Figure 6.1: Examples of 3rd-order and 4th-order De Bruijn cycles.

Theorem 6.18 (De Bruijn, 1946)**.** *There are* $2^{2^{k-1}-k}$ *kth-order De Bruijn cycles.*

Corollary 6.19. *We cannot store most kth-order De Bruijn cycles in* $o(2^k)$ *bits.*

Since there are 2^k possible k-tuples, kth-order De Bruijn cycles have length 2^k, so Corollary 6.19 means that we cannot compress most De Bruijn cycles by more than a constant factor. Therefore, we can prove a lower bound similar to Theorem 6.8 by supposing that s's repeated substring is a De Bruijn cycle, then using Lemma 6.17 instead of Lemma 6.6.

Theorem 6.20. *With one stream, we cannot achieve an entropy-only bound.*

Proof. As in the proof of Theorem 6.8, suppose an algorithm uses only one stream, m bits of memory and p passes to compress s, with $mp = \log^{\mathcal{O}(1)} n$, and consider a machine performing that algorithm. This time, however, suppose s is periodic with period $2^{\lceil \log(mp \log n) \rceil}$ and that its repeated substring t is a kth-order De Bruijn cycle, $k = \lceil \log(mp \log n) \rceil$, that is not compressible by more than a constant factor. Lemma 6.7 implies that the machine's output while over a part of the stream that initially holds a copy of t, must be $\Omega(mp \log n - mp) = \Omega(mp \log n)$. Therefore, the machine's complete output must be $\Omega(n)$ bits. By Lemma 6.17, however, $nH_k^*(s) = \mathcal{O}\big(2^k \log n\big) = \mathcal{O}\big(mp \log^2 n\big) \subset \log^{\mathcal{O}(1)} n$. □

Notice Theorem 6.20 implies a lower bound for computing the BWT: if we could compute the BWT with one stream then, since we can compute move-to-front, run-length and arithmetic coding using $\mathcal{O}(\log n)$ bits of memory and $\mathcal{O}(1)$ passes over one stream, we could thus achieve an entropy-only bound with one stream, contradicting Theorem 6.20.

Corollary 6.21. *With one stream, we cannot compute the BWT.*

Grohe and Schweikardt [GS05] proved that, with $\mathcal{O}(1)$ streams, we generally cannot sort $n/\log n$ numbers, each consisting of $\log n$ bits, using $\mathcal{O}(n^{1-\epsilon})$ bits of memory and $o(\log n)$ passes. Combining this result with the following lemma, we immediately obtain a lower bound for computing the BWT with $\mathcal{O}(1)$ streams.

Lemma 6.22. *With two or more streams, sorting $\mathcal{O}(n/\log n)$ numbers, each of $\log n$ bits, takes $\mathcal{O}(\log n)$ more bits of memory and $\mathcal{O}(1)$ more passes than computing the BWT of a ternary string of length n.*

Proof. We reduce the problem of sorting a sequence $x_1, \ldots, x_m$ of $(\log n)$-bit binary numbers, $m = n/(2\log n + \log\log n + 2)$, to the problem of computing the BWT of a ternary string of length n. Let $x_i[j]$ denote the jth bit of x_i. Using two streams, $O(1)$ passes and $O(\log n)$ memory, we replace each $x_i[j]$ by $x_i[j]\ 2\ x_i\ i\ j$, writing 2 as a single character, x_i and i each as $\log n$ bits, and j as $\log\log n$ bits. Let X be the resulting string and consider the last $m \log n$ characters of the BWT of X: they are a permutation of the characters followed by 2s in X, i.e., the bits of $x_1, \ldots, x_m$; if $x_i < x_{i'}$ or $x_i = x_{i'}$ but $i < i'$ then, because $2\ x_i\ i$ is lexicographically less than $2\ x_{i'}\ i'$, each bit of x_i comes before each bit of $x_{i'}$; if $j < j'$ then, for any i, because $2\ x_i\ i\ j$ is lexicographically less than $2\ x_i\ i\ j'$, the bit $x_i[j]$ comes before the bit $x_i[j']$. In other words, the last $m \log n$ characters of the BWT of X are $x_1, \ldots, x_m$ in sorted order. □

Corollary 6.23. *With $\mathcal{O}(1)$ streams, we cannot compute the BWT of a ternary string of length n using $\mathcal{O}(n^{1-\epsilon})$ bits of memory and $o(\log n)$ passes.*

In another paper [GM07a] we improved the coefficient in Manzini's bound from $5 + \epsilon$ to 2.7, using a variant of distance coding instead of move-to-front and run-length coding. We conjecture this algorithm can also be implemented with two streams.

Open Problem 6.24. *With $\mathcal{O}(1)$ streams, can we achieve the same entropy-only bounds that we achieve in the RAM model?*

The main idea of distance coding [Bin00] is to write the starting position of each maximal run (i.e., subsequence consisting of copies of the same character), by writing the distance from the start of each maximal run to the start of the next maximal run of the same character. Notice we do not need to write the length of each run because the end of each run (except the last) is the position before the start of the next one. By symmetry, it makes essentially no difference to the length of the encoding if we write the distance to the start of each maximal run from the start of the previous maximal run of the same character, which is not difficult with $\mathcal{O}(\sigma \log n)$ bits of memory and $\mathcal{O}(1)$ passes.

Kaplan, Landau and Verbin [KLV07] showed how, using the BWT followed by distance coding and arithmetic coding, we can store s in $1.73nH_k(s)+\mathcal{O}(\log n)$ bits for any fixed σ and k. This bound holds only when we use an idealized arithmetic coder with $\mathcal{O}(\log n)$ total redundancy; if we use a 0th-order coder with per character redundancy μ, then the bound becomes $1.73nH_k(s) + \mu n + \mathcal{O}(\log n)$. In our paper [GM07a] we used a lemma due to Mäkinen and Navarro [MN05] bounding the number of runs in terms of the product of the length and the 0th-order empirical entropy, to change the latter bound into $(1.73 + \mu)nH_k(s) + \mathcal{O}(\log n)$, which is an improvement when $H_k(s) < 1$. Unfortunately, the presence of the $\mathcal{O}(\log n)$ term prevents this from being an entropy-only bound. To prove an entropy-only bound, we modified distance coding to use an escape mechanism, which we have not verified can be implemented in the read/write streams model.

Chapter 7

Conclusions and Future Work

In this thesis we have tried to provide a fairly complete but coherent view of our studies of sequential-access data compression, balancing discussion of previous work with presentation of our own results. We would like to highlight now what we consider our key ideas. The most important innovation in Chapter 2 was probably our use of predecessor queries for encoding and decoding with a canonical code. This, combined with our use of Fredman and Willard's data structure [FW93], Shannon coding and background processing, allowed us to encode and decode each character in constant worst-case time while producing an encoding whose length was worst-case optimal. Chapters 3 and 4 were, admittedly, somewhat tangential to our topic, but we included them to show how our interests shifted from the model we considered in Chapter 2 to the one we considered in Chapter 5. The key idea in Chapter 5 was to view one-pass algorithms with memory bounded in terms of the alphabet size and context length as finite-state machines. This, combined with the fact that short, randomly chosen strings almost certainly have low empirical entropy, allowed us to prove a lower bound on the amount of memory needed to achieve good compression, that nearly matched our upper bound (which was relatively easy to prove, given Lemma 5.1). Finally, the key idea in Chapter 6 was to extend the automata-theoretic arguments of Chapter 5 to algorithms that can make multiple passes and use an amount of memory that depends on the length of the input. This gave us our lower bound for achieving good grammar-based compression with one stream, our lower bound for finding strings' minimum periods and, combined with properties of De Bruijn sequences, our lower bound for achieving entropy-only bounds.

As we mentioned in the introduction, a paper [GKN09] we wrote with Marek

Karpinski and Yakov Nekrich at the University of Bonn that partially combines the results in Chapters 2 and 5, will appear at the 2009 Data Compression Conference. This paper concerns fast adaptive prefix coding with memory bounded in terms of the alphabet size and context length, and shows that we can encode s in $(\lambda H + \mathcal{O}(1))n + o(n)$ bits while using $\mathcal{O}\big(\sigma^{1/\lambda+\epsilon}\big)$ bits of memory and $\mathcal{O}(\log\log\sigma)$ worst-case time to encode and decode each character. Of course, we would like to improve these bounds, and perhaps implement and test how our algorithm performs with large alphabets such as Chinese, Unicode or the English vocabulary. We would also like to implement our algorithm from Chapter 2, testing several implementations of dictionaries to determine which is the fastest in practice; Fredman and Willard's analysis has enormous constants hidden in the asymptotic notation. Finally, we are preparing a paper with Nekrich that will give efficient algorithms for adaptive alphabetic prefix coding, adaptive prefix coding for unequal letter costs, and adaptive length-restricted prefix coding (see [Gag07a] for descriptions of these problems).

As we also mentioned in the introduction, we are currently trying to prove more results like the lower bound in Chapter 6 on finding strings' minimum periods. We are working on the open problems presented in Chapter 6, about using multiple passes to obtain smaller redundancy terms for universal compression with one stream, approximating the smallest grammar with $\mathcal{O}(1)$ streams, and achieving better entropy-only bounds with $\mathcal{O}(1)$ streams. Finally, we have been collaborating with Paolo Ferragina at the University of Pisa and Giovanni Manzini at the University of Eastern Piedmont on a paper [FGM] about BWT-based compression in the external memory model (see [Vit08]) with limited random disk accesses. For the moment, however, our curiosity about sequential-access data compression is mostly satisfied.

After proving our first results about adaptive prefix coding [Gag07a], we wrote several papers [Gag06a, Gag06b, Gag08a, Gag09] concerning the number of bits needed to store a good approximation of a probability distribution and, more generally, a Markov process. For one of these papers [Gag06b], about bounds on the redundancy in terms of the alphabet size and context length, we proved versions of Lemmas 5.1 and 5.7, which eventually led to Chapters 5 and 6. We are now curious whether our results can be combined with algorithms that build sophisticated probabilistic models, either for data compression (see, e.g., [FGMS05, FGM06, FM08]) or for inference (see, e.g., [Ris86, RST96, AB00, BY01] and subsequent

articles). These algorithms work by considering a class of probabilistic models that are, essentially, Markov sources with variable-length contexts, and finding the model that minimizes the sum of the length of the model's description and the self-information of the input with respect to the model; we note this sum is something like the kth-order modified empirical entropy. Similar kinds of models are used in both applications because many algorithms for inference are based on Rissanen's Minimum Description Length Principle [Ris78], which is based on ideas from data compression.

How we minimize the sum of the length of the model's description and the self-information depends on how we represent the model. At least some of the algorithms mentioned above assume that the length of description is proportional to the number of contexts used. However, it seems that, if some contexts occur frequently but the distributions of characters that follow them are nearly uniform, and others occur rarely but are always followed by the same character, then it might give better compression to prune the former and keep the latter, which take only $\mathcal{O}(\log \sigma)$ bits each to store. Of course, this is just speculation at the moment.

Bibliography

[AB00] A. Apostolico and G. Bejerano. Optimal amnesic probabilistic automata, or, How to learn and classify proteins in linear time and space. *Computational Biology*, 7(3–4):381–393, 2000.

[ABD+07] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. An optimal cache-oblivious priority queue and its application to graph algorithms. *SIAM Journal on Computing*, 36(6):1672–1695, 2007.

[Abr01] J. Abrahams. Code and parse trees for lossless source encoding. *Communications in Information and Systems*, 1(2):113–146, 2001.

[ABT99] A. Andersson, P. Bro Miltersen, and M. Thorup. Fusion trees can be implemented with AC^0 instructions only. *Theoretical Computer Science*, 215(1–2):337–344, 1999.

[ADRR04] G. Aggarwal, M. Datar, S. Rajagopalan, and M. Ruhl. On the streaming model augmented with a sorting primitive. In *Proceedings of the 45th Symposium on Foundations of Computer Science*, pages 540–549, 2004.

[AL62] G. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. *Doklady Akademi Nauk*, 146:263–266, 1962.

[BBD+02] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the 21st Symposium on Principles of Database Systems*, pages 1–16, 2002.

[BF02] V. Becher and S. Figueira. An example of a computable absolutely normal number. *Theoretical Computer Science*, 270(1–2):947–958, 2002.

[BH08] P. Beame and D.-T. Huỳnh-Ngọc. On the value of multiple read/write streams for approximating frequency moments. In *Proceedings of the 49th Symposium on Foundations of Computer Science*, pages 499–508, 2008.

[Bin00] E. Binder. Distance coder. Usenet group comp.compression, 2000.

[BM04] R. S. Bird and S.-C. Mu. Inverting the Burrows-Wheeler transform. *Journal of Functional Programming*, 14(6):603–612, 2004.

[Bor09] É. Borel. Les probabilités dénombrables et leur applications arithmétiques. *Rendiconti del Circolo Matematico di Palermo*, 27:247–271, 1909.

[BSTW86] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29:320–330, 1986.

[BW94] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 24, Digital Equipment Corporation, 1994.

[BY01] G. Bejerano and G. Yona. Variations on probabilistic suffix trees: Statistical modeling and prediction of protein families. *Bioinformatics*, 17(1):23–43, 2001.

[CE46] A. H. Copeland and P. Erdös. Note on normal numbers. *Bulletin of the American Mathematical Society*, 52:857–860, 1946.

[Cha33] D. G. Champernowne. The construction of decimals normal in the scale of 10. *Journal of the London Mathematical Society*, 8:254–260, 1933.

[CLL+05] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

[CT06] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley, 2nd edition, 2006.

[CV05] R. Cilibrasi and P. Vitányi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545, 2005.

[CY91] J. Chen and C.-K. Yap. Reversal complexity. *SIAM Journal on Computing*, 20(4):622–638, 1991.

[dB46] N. G. de Bruijn. A combinatorial problem. *Koninklijke Nederlandse Akademie van Wetenschappen*, 49:758–764, 1946.

[DLM02] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Frequency estimation of Internet packet streams with limited space. In *Proceedings of the 10th European Symposium on Algorithms*, pages 348–360, 2002.

[DM80] D. P. Dobkin and J. I. Munro. Determining the mode. *Theoretical Computer Science*, 12:255–263, 1980.

[DS02] M. Drmota and W. Szpankowski. Generalized Shannon code minimizes the maximal redundancy. In *Proceedings of the 5th Latin American Symposium on Theoretical Informatics*, pages 306–318, 2002.

[DS04] M. Drmota and W. Szpankowski. Precise minimax redundancy and regret. *IEEE Transactions on Information Theory*, 50(11):2686–2707, 2004.

[Eli75] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

[EMŞ04] F. Ergün, S. Muthukrishnan, and S. C. Şahinalp. Sublinear methods for detecting periodic trends in data streams. In *Proceedings of the 6th Latin American Symposium on Theoretical Informatics*, pages 16–28, 2004.

[Fal73] N. Faller. An adaptive system for data compression. In *Record of the 7th Asilomar Conference on Circuits, Systems and Computers*, pages 593–597, 1973.

[FGG+07] P. Ferragina, R. Giancarlo, V. Greco, G. Manzini, and G. Valiente. Compression-based classification of biological sequences and structures via the Universal Similarity Metric: Experimental assessment. *BMC Bioinformatics*, 8:252, 2007.

[FGM] P. Ferragina, T. Gagie, and G. Manzini. Space-conscious data indexing and compression in a streaming model. In preparation.

[FGM06] P. Ferragina, R. Giancarlo, and G. Manzini. The engineering of a compression boosting library: Theory vs. practice in BWT compression. In *Proceedings of the 14th European Symposium on Algorithms*, pages 756–767, 2006.

[FGM09] P. Ferragina, R. Giancarlo, and G. Manzini. The myriad virtues of wavelet trees. *Information and Computation*, 207(8):849–866, 2009.

[FGMS05] P. Ferragina, R. Giancarlo, G. Manzini, and M. Sciortino. Boosting textual compression in optimal linear time. *Journal of the ACM*, 52(4):688–713, 2005.

[Fis84] T. M. Fischer. On entropy decomposition methods and algorithm design. *Colloquia Mathematica Societatis János Bolyai*, 44:113–127, 1984.

[FM08] P. Ferragina and G. Manzini. Boosting textual compression. In M.-Y. Kao, editor, *Encyclopedia of Algorithms*. Springer, 2008.

[FMMN07] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2), 2007.

[FNV09] P. Ferragina, I. Nitto, and R. Venturini. On optimally partitioning a text to improve its compression. In *Proceedings of the 17th European Symposium on Algorithms*, 2009. To appear.

[FW93] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993.

[Gag04] T. Gagie. Dynamic Shannon coding. In *Proceedings of the 12th European Symposium on Algorithms*, pages 359–370, 2004.

[Gag06a] T. Gagie. Compressing probability distributions. *Information Processing Letters*, 97(4):133–137, 2006.

[Gag06b] T. Gagie. Large alphabets and incompressibility. *Information Processing Letters*, 99:246–251, 2006.

[Gag07a] T. Gagie. Dynamic Shannon coding. *Information Processing Letters*, 102(2–3):113–117, 2007.

[Gag07b] T. Gagie. Sorting streamed multisets. In *Proceedings of the 10th Italian Conference on Theoretical Computer Science*, pages 130–138, 2007.

[Gag08a] T. Gagie. Dynamic asymmetric communication. *Information Processing Letters*, 108(6):352–355, 2008.

[Gag08b] T. Gagie. Sorting streamed multisets. *Information Processing Letters*, 108(6):418–421, 2008.

[Gag09] T. Gagie. Compressed depth sequences. *Theoretical Computer Science*, 410(8–10):958–962, 2009.

[Gal78] R. G. Gallager. Variations on a theme by Huffman. *IEEE Transactions on Information Theory*, 24(6):668–674, 1978.

[GG09] P. Gawrychowski and T. Gagie. Minimax trees in linear time with applications. In *Proceedings of the 20th International Workshop on Combinatorial Algorithms*, 2009. To appear.

[GGV08] A. Gupta, R. Grossi, and J. S. Vitter. Nearly tight bounds on the encoding length of the Burrows-Wheeler Transform. In *Proceedings of the 4th Workshop on Analytic Algorithmics and Combinatorics*, pages 191–202, 2008.

[GKN09] T. Gagie, M. Karpinski, and Y. Nekrich. Low-memory adaptive prefix coding. In *Proceedings of the Data Compression Conference*, pages 13–22, 2009.

[GKS07] M. Grohe, C. Koch, and N. Schweikardt. Tight lower bounds for query processing on streaming and external memory data. *Theoretical Computer Science*, 380(1–3):199–217, 2007.

[GM07a] T. Gagie and G. Manzini. Move-to-front, distance coding, and inversion frequencies revisited. In *Proceedings of the 18th Symposium on Combinatorial Pattern Matching*, pages 71–82, 2007.

[GM07b] T. Gagie and G. Manzini. Space-conscious compression. In *Proceedings of the 32nd Symposium on Mathematical Foundations of Computer Science*, pages 206–217, 2007.

[GN] T. Gagie and Y. Nekrich. Worst-case optimal adaptive prefix coding. arXiv:0812.3306.

[GS05] M. Grohe and N. Schweikardt. Lower bounds for sorting with few random accesses to external memory. In *Proceedings of the 24th Symposium on Principles of Database Systems*, pages 238–249, 2005.

[GSU09] R. Giancarlo, D. Scaturro, and F. Utro. Textual data compression in the -omic sciences: A synopsis. *Bioinformatics*, 25(13):1575–1586, 2009.

[HR90] T. Hagerup and C. Rüb. A guided tour of Chernoff bounds. *Information Processing Letters*, 33(6):305–308, 1990.

[HS08] A. Hernich and N. Schweikardt. Reversal complexity revisited. *Theoretical Computer Science*, 401(1–3):191–205, 2008.

[Huf52] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[HV92] P. G. Howard and J. S. Vitter. Analysis of arithmetic coding for data compression. *Information Processing and Management*, 28(6):749–764, 1992.

[Kle00] S. T. Klein. Skeleton trees for the efficient decoding of Huffman encoded texts. *Information Retrieval*, 3(1):7–23, 2000.

[KLV07] H. Kaplan, S. Landau, and E. Verbin. A simpler analysis of Burrows-Wheeler-based compression. *Theoretical Computer Science*, 387(3):220–235, 2007.

[KM99] R. Kosaraju and G. Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM Journal on Computing*, 29(3):893–911, 1999.

[KN76] G. O. H. Katona and T. O. H. Nemetz. Huffman codes and self-information. *IEEE Transactions on Information Theory*, 22(3):337–340, 1976.

[KN09] M. Karpinski and Y. Nekrich. A fast algorithm for adaptive prefix coding. *Algorithmica*, 55(1):29–41, 2009.

[Knu67] D. E. Knuth. Oriented subtrees of an arc digraph. *Journal of Combinatorial Theory*, 3(4):309–314, 1967.

[Knu85] D. E. Knuth. Dynamic Huffman coding. *Journal of Algorithms*, 6(2):163–180, 1985.

[Knu98] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 2nd edition, 1998.

[Kol65] A. N. Kolmogorov. Three approaches to the quantitative definition of information. *Problems in Information Transmission*, 1:1–7, 1965.

[KSP03] R. M. Karp, S. Shenker, and C. H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems*, 28(1):51–55, 2003.

[KV07] H. Kaplan and E. Verbin. Most Burrows-Wheeler based compressors are not optimal. In *Proceedings of the 18th Symposium on Combinatorial Pattern Matching*, pages 107–118, 2007.

[LV08] M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer-Verlag, 3rd edition, 2008.

[LZ76] A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976.

[Man01] G. Manzini. An analysis of the Burrows-Wheeler Transform. *Journal of the ACM*, 48(3):407–430, 2001.

[Meh77] K. Mehlhorn. A best possible bound for the weighted path length of binary search trees. *SIAM Journal on Computing*, 6(2):235–239, 1977.

[MG82] J. Misra and D. Gries. Finding repeated elements. *Science of Computer Programming*, 2(2):143–152, 1982.

[MLP99] R. L. Milidiú, E. S. Laber, and A. A. Pessoa. Bounding the compression loss of the FGK algorithm. *Journal of Algorithms*, 32(2):195–211, 1999.

[MN05] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.

[MN07] V. Mäkinen and G. Navarro. Implicit compression boosting with applications to self-indexing. In *Proceedings of the 14th Symposium on String Processing and Information Retrieval*, pages 229–241, 2007.

[MP80] J. I. Munro and M. S. Paterson. Selection and sorting with limited storage. *Theoretical Computer Science*, 12:315–323, 1980.

[MR91] J. I. Munro and V. Raman. Sorting multisets and vectors in-place. In *Proceedings of the 2nd Workshop on Algorithms and Data Structures*, pages 473–480, 1991.

[MS76] J. I. Munro and P. M. Spira. Sorting and searching in multisets. *SIAM Journal on Computing*, 5(1):1–8, 1976.

[Mut05] S. Muthukrishnan. *Data Streams: Algorithms and Applications*. Foundations and Trends in Theoretical Computer Science. Now Publishers, 2005.

[Nek07] Y. Nekrich. An efficient implementation of adaptive prefix coding. In *Proceedings of the Data Compression Conference*, page 396, 2007.

[Ris78] J. Rissanen. Modeling by shortest data description. *Automatica*, 14:465–471, 1978.

[Ris86] J. Rissanen. Complexity of strings in the class of Markov sources. *IEEE Transactions on Information Theory*, 32(4):526–532, 1986.

[RO04] L. G. Rueda and B. J. Oommen. A nearly-optimal Fano-based coding algorithm. *Information Processing and Management*, 40(2):257–268, 2004.

[RO06] L. Rueda and B. J. Oommen. A fast and efficient nearly-optimal adaptive Fano coding scheme. *Information Sciences*, 176(12):1656–1683, 2006.

[RO08] L. Rueda and B. J. Oommen. An efficient compression scheme for data communication which uses a new family of self-organizing binary search trees. *International Journal of Communication Systems*, 21(10):1091–1120, 2008.

[Rob55] H. Robbins. A remark on Stirling's Formula. *American Mathematical Monthly*, 62(1):26–29, 1955.

[RST96] D. Ron, Y. Singer, and N. Tishby. The power of amnesia: Learning probabilistic automata with variable memory length. *Machine Learning*, 25(2–3):117–149, 1996.

[Ruh03] J. M. Ruhl. *Efficient Algorithms for New Computational Models*. PhD thesis, Massachusetts Institute of Technology, 2003.

[Ryt03] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1–3):211–222, 2003.

[Sav97] S. Savari. Redundancy of the Lempel-Ziv incremental parsing rule. *IEEE Transactions on Information Theory*, 43(1):9–21, 1997.

[Sch07] N. Schweikardt. Machine models and lower bounds for query processing. In *Proceedings of the 26th Symposium on Principles of Database Systems*, pages 41–52, 2007.

[Sha48] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 623–656, 1948.

[Sie17] M. W. Sierpinski. Démonstration élémentaire du théorème de m. borel sur les nombres absolument normaux et détermination dun tel nombre. *Bulletin de la Société Mathématiques de France*, 45:127–132, 1917.

[SK64] E. S. Schwartz and B. Kallick. Generating a canonical prefix encoding. *Communications of the ACM*, 7(3):166–169, 1964.

[ST85] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[Tho03] M. Thorup. On AC^0 implementations of fusion trees and atomic heaps. In *Proceedings of the 14th Symposium on Discrete Algorithms*, pages 699–707, 2003.

[TM00] A. Turpin and A. Moffat. Housekeeping for prefix coding. *IEEE Transactions on Communications*, 48(4), 2000.

[TM01] A. Turpin and A. Moffat. On-line adaptive canonical prefix coding with bounded compression loss. *IEEE Transactions on Information Theory*, 47(1):88–98, 2001.

[Tur92] A. M. Turing. A note on normal numbers. In J. L. Britton, editor, *Collected Works of A.M. Turing: Pure Mathematics*, pages 117–119. North-Holland, 1992.

[vAEdB51] T. van Aardenne-Ehrenfest and N. G. de Bruijn. Circuits and trees in oriented linear graphs. *Simon Stevin*, 48:203–217, 1951.

[Vit87] J. S. Vitter. Design and analysis of dynamic Huffman codes. *Journal of the ACM*, 1987(4):825–845, 1987.

[Vit08] J. S. Vitter. *Algorithms and Data Structures for External Memory*. Foundations and Trends in Theoretical Computer Science. Now Publishers, 2008.

[YY02] C. Ye and R. W. Yeung. A simple upper bound on the redundancy of Huffman codes. *IEEE Transactions on Information Theory*, 48(7):2132–2138, 2002.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[ZL78] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.