arXiv:2201.02514v4 [cs.IT] 17 Nov 2026

Efficiency of ANS Entropy Encoders

Dmitry Kosolobov[1,2]

[1]St. Petersburg State University, St. Petersburg, Russia.
[2]Ural Federal University, Ekaterinburg, Russia.
E-mail: dkosolobov@mail.ru. ORCID: 0000-0002-2909-2952

Abstract

Asymmetric Numeral Systems (ANS) is a class of entropy encoders that had an immense impact on the data compression, substituting arithmetic and Huffman coding. It was studied by different authors but the precise asymptotics of its redundancy (in relation to the entropy) was not completely understood. We obtain asymptotically tight bounds for the redundancy of the tabled ANS (tANS), the most popular ANS variant. Given a sequence $a_1, a_2, \ldots, a_n$ of symbols from an alphabet $\{0, 1, \ldots, \sigma-1\}$ such that each symbol a occurs in it f_a times and $n = 2^r$, the tANS encoder using Duda's "precise initialization" to fill tANS tables transforms this sequence into a bit string of the following length (the frequencies are not included in the encoding): $\sum_{a \in [0..\sigma)} f_a \cdot \log \frac{n}{f_a} + O(\sigma + r)$, where $O(\sigma + r)$ can be bounded by $\sigma \log e + r$. The r-bit term is an artifact indispensable to ANS; the rest incurs a redundancy of $O(\frac{\sigma}{n})$ bits per symbol. We complement this by examples showing that an $\Omega(\sigma + r)$ redundancy is necessary. We argue that similar examples exist for most adequate initialization methods for tANS. Thus, we refute Duda's conjecture that the redundancy is $O(\frac{\sigma}{n^2})$ bits per symbol. We also propose a variant of the range ANS (rANS), called rANS with fixed accuracy, parameterized by $k \geq 1$ that in certain conditions might be faster than the standard rANS because it avoids slow explicit division operations. We bound the redundancy for our rANS variant by $\frac{n}{2^k - 1} \log e + r + k$.

Keywords: Asymmetric numeral systems, ANS, finite state entropy, FSE, arithmetic coding, redundancy

1 Introduction

Asymmetric numeral systems (ANS) is a class of entropy encoders invented by Duda in 2009 [1–3]. These encoders had a huge impact on the data compression by providing the same rates of compression as the arithmetic coding [4–8] while being as fast as the Huffman coding [9] (and even faster in some scenarios). Since the invention of ANS and the emergence of its efficient implementation by

Collet [10], several high performance compressors based on ANS appeared [11–13] and it was integrated in some modern media formats [14,15]. The theoretical community also contributed to the study of ANS in a series of works [16–23], though less actively than practitioners [13, 24–26].

The primary focus of the theoretical analysis for entropy encoders is in the estimation of the redundancy, the difference between the number of bits produced by the encoder and the information theoretic entropy lower bound. The main result of the present paper is a tight asymptotic upper and lower bound on the redundancy of the most popular variant of ANS, called the tabled ANS (tANS) or, sometimes, the Finite State Entropy (FSE). Our analysis of the redundancy does not impose any randomized assumptions on the source that produced the input for the encoder; instead, we establish worst-case bounds in terms of the so-called *empirical entropy* [27, 28] (the definitions follow). In addition, we also introduce and analyze a novel variant of the range ANS (rANS), another version of ANS used in practice.

An entropy encoder receives as its input a sequence of n numbers from a set $\{0, 1, \ldots, \sigma - 1\}$ and transforms it into a bit string. Duda analyzed his tANS encoder and found that its redundancy is $O(\frac{\sigma}{n} \log n)$ bits per symbol[1] provided an appropriate initialization is used for tANS tables (details are explained in the sequel); see Equation (40) in [1]. His estimation, however, seems somewhat non-rigorous and in a different (more classical) setting: he considers a memoryless source generating the input at random and he estimates the expected length of the output encoding. Based on experimental evaluations, Duda conjectured that a tight bound for the redundancy is $O(\frac{\sigma}{n^2})$ bits per symbol [2]. Yokoo and Dubé [22] investigated the same problem in more rigorous terms closer to our setting and showed that the redundancy per symbol vanishes as the length n tends to infinity while σ is fixed (yet, they have some questionable assumptions in their derivations). Another analysis of the expected output length for the memoryless source was conducted in [20]: their bounds, however, are incomplete in a sense and, thus, cannot be easily compared to other results. In contrast, our analysis establishes upper bounds in the worst case without probabilistic assumptions.

In this paper we prove that the redundancy for tANS is $O(\frac{\sigma}{n})$ bits per symbol. We complement this upper bound by a series of examples showing that it is asymptotically tight when $\sigma > n/3$. As in the works cited above, here we did not include in this bound the $r = \log n$ redundant bits that are always produced by ANS encoders in the worst case (it is an artifact of the initial state of the encoder). After uncovering the constant under the big-O and including this r-bit term, the upper bound for the tANS redundancy that we establish can be expressed as $\sigma \log e + r$ bits over all symbols (not per symbol). Our lower bound examples show that a redundancy of $\frac{\sigma-1}{4} + r - 2$ bits is attainable, for $\sigma \approx n/3$. An important part of tANS is the initialization of its internal tables, which has a significant impact on the compression performance. The lower bound examples work for a wide class of initialization algorithms so that the same

[1] All logarithms in the paper are in base two.

redundancy can be observed for any adequate algorithm that generates the tANS tables using only the known frequencies of symbols without processing the sequence itself (however, methods that analyze the sequence might avoid this effect [17, 29]; these methods are usually infeasible in practice due to their relatively slow performance). We note that an analogous upper bound of $O(\frac{1}{2^w})$ bits per symbol is known for practical variants of the arithmetic coding that use w-bit machine words [30].

For the tANS encoder, it is natural to assume that the number of its internal states, $n = 2^r$, is comparable with the length of the input sequence since the tANS uses quite large internal tables for good performance. All above results are stated for the case when the length of the input sequence, m, is equal to n. When it is not the case and m is larger than n, we also obtain upper bounds; in particular, we rederive a very recent result from [23]: when n and σ are fixed and m tends to infinity, the redundancy per symbol can be upperbounded by $D(p||\hat{p}) + O(\frac{\sigma}{n})$, where $D(p||\hat{p})$ is the Kullback–Leibler divergence between the empirical distribution of symbols, p, and the approximate distribution, $\hat{p}$, that is used by the tANS with n internal states (details follow).

The second contribution of the present paper is a modification of the range ANS (rANS). The rANS is another variant of ANS invented by Duda. It has a number of advantages in some use scenarios because of which it was favoured by some practitioners (and because it is easier to learn). The rANS was first noticeably slower in practice than the tANS but its current SIMD implementations tend to be at least on a par with tANS. The main advantage of rANS is that it does not need tANS tables (however, fast variants still require some specific tables). Due to the less wasteful use of memory, the rANS is more suitable for (pseudo)adaptive compression or when several streams of data are encoded simultaneously [13, 24, 26, 31]. An inherent problem of rANS that slows down it significantly is a necessity to perform the integer division during the encoding, which is expensive on modern processors. The proposed modification of rANS, which we call rANS with fixed accuracy, tries to mitigate this issue making the encoder faster while preserving the same good properties of rANS.

The new rANS takes as a regulated parameter an integer $k \geq 1$. It is guaranteed that the division can be performed only in cases when its result belongs to the range $[2^k..2^{k+1})$. Thus, the division can be computed by faster methods provided k is small. We upperbound the redundancy for the rANS with fixed accuracy k by $\frac{n}{2^k-1}\log e + r$. In our experiments, the new rANS variant with $k = 3$ provides faster encoding than the standard rANS with explicit divisions but slower than the rANS in which the divisions are performed by multiplications and shifts on precomputed constants (see [32, 33]); the decoding in the new rANS was slower. We believe that the rANS with fixed accuracy might be suitable when the encoder should be adaptive (so that one cannot precompute constants for faster divisions) and the encoding speed is a priority.

The paper is organized as follows. After short preliminaries, Section 3 introduces a simple variant of the range ANS and, based on it, the tabled ANS. Section 4 presents a tight analysis of the redundancy for the tabled ANS, upper and lower bounds. Section 5 describes a novel variant of range ANS and

analyzes it. We conclude with some open problems in Section 6.

2 Preliminaries

A *symbol* is an element of a finite set of integers called an *alphabet*. We consider sequences of symbols. The sequences are also sometimes called *strings*. Integer intervals are denoted by $[i..j] = \{k \in \mathbb{Z} \colon i \le k \le j\}$ and $[i..j) = [i..j] \setminus \{j\}$. Fix an alphabet $[0..\sigma)$ with σ symbols and a sequence of symbols $a_1, a_2, \ldots, a_n$ from it. For each symbol a, denote by f_a the number of occurrences of a in the sequence. The number f_a is called the *frequency of* a. The number f_a/n is the *empirical probability of* a. An entropy encoder transforms this sequence into a sequence of bits that is then transmitted to a decoder. The encoder usually also transmits to the decoder the table of frequencies f_a. However, the problem of storage for the table is out of the scope of the present paper and we focus only on the encoding for the sequence itself implicitly assuming that both sides know the length n of the sequence and the frequencies of symbols. The following quantity is used as an information theoretic lower bound [27, 34] for the number of bits that any encoder should spend to encode some (least compressible) sequences with the given frequencies of symbols $\{f_a\}_{a\in[0..\sigma)}$ (even under the assumption that the table of frequencies f_a is known):

$$\sum_{a\in[0..\sigma)} f_a \log \frac{n}{f_a}, \tag{1}$$

assuming that $f_a \log \frac{n}{f_a} = 0$ whenever $f_a = 0$. We call (1) the *entropy formula* (though the empirically calculated entropy itself, i.e., the *empirical entropy*, is defined as the quantity (1) divided by n). The difference between the number of bits produced by an encoder and the optimal number of bits from (1) is called a *redundancy*. The redundancy is the primary focus of our analysis of ANS.

For each symbol a, denote $c_a = \sum_{a'\in[0..a)} f_{a'}$ (these numbers are called *cumulative frequencies*). Typically, encoders store two arrays of σ numbers c_a and f_a and some additional tables necessary for encoding. The ANS encoders are not an exception.

Remark 1. *We digress at this point to discuss how the quantity* (1) *can be interpreted as a lower bound. As it was mentioned in the introduction, we aim to avoid any probabilistic assumptions on the source of the input. In this case, the information theoretic lower bound for the length of the encoding is*

$$\left\lceil \log \binom{n}{f_0, f_1, \ldots, f_{\sigma-1}} \right\rceil = \left\lceil \log \frac{n!}{f_0! f_1! \cdots f_{\sigma-1}!} \right\rceil, \tag{2}$$

provided the length n of the sequence and the table of frequencies f_a are accessible to the encoder and decoder for free. The bound (2) *is tight: it is achieved by a simple encoder that uses the same number of bits as in* (2) *for all inputs whose symbol frequencies are $f_0, f_1, \ldots, f_{\sigma-1}$, and the encoding is the index of the*

input in the lexicographically sorted list of all possible inputs. The tight bound (2) *is less than the entropy bound* (1) *but close (e.g., see [35, Lem. II.2]): the difference is at most* $O(\sigma \log \frac{n}{\sigma})$*. Hence,* (1) *is not a lower bound in this strict sense.*

The quantity (1) *is equal to* $\min \log(1/\Pr(Q \text{ produces } a_1, a_2, \ldots, a_n))$*, where the minimum is taken for all memoryless sources* Q *randomly generating sequences of length* n *over the alphabet* $[0..\sigma)$ *(see [36]). If one restricts attention only to encoders that produce uniquely decodable codes (note that the encoder produces one codeword for the whole sequence of length* n *from the alphabet* $[0..\sigma)$*, not symbol by symbol), then* (1) *is a lower bound for the expected number of bits produced by such encoder provided each symbol of the length-*n *input sequence is generated by the memoryless source with respective probabilities* f_a/n *of the symbols [34, 37]. It is this fundamental fact that seems to justify the usage of* (1) *as a lower bound in [27, 28] and numerous subsequent papers. (The lower bound for non-uniquely decodable codes is close to* (1) *but smaller; see [38, 39].)*

It is still convenient to refer to (1) *as a kind of "optimal lower bound" and, following the established tradition, we call it as such in the sequel, despite the fact that it is not exacty the case.*

As a side note: if the table of frequencies cannot be accessed by the decoder for free and, thus, should be transmitted somehow, then the tight lower bound for the length of the uniquely decodable code is equal to (1) *plus* $\Theta(\sigma \log \frac{n}{\sigma})$ *when* $\sigma < n$ *(see [40–42]). Therefore,* (1) *is indeed a lower bound in this case but it is not tight (recall that this is not our setting: we consider the frequencies as freely transmitted to the decoder).*

3 ANS Encoders

On a high level, an ANS encoder can be described as a data structure that maintains a positive integer w that can be modified by two stack operations "$w = \mathsf{push}(w, a)$" and "$(w, a) = \mathsf{pop}(w)$": push encodes a symbol a into the number w and returns the modified value for w, and pop performs a reverse operation decoding the symbol a and restoring the old value for w. It is instructive to image the number w as a sequence of bits (potentially very long) from the highest bit of w, which is always 1, to lowest bits. Given a sequence of symbols $a_1, a_2, \ldots, a_n$, the encoder pushes them consecutively and transmits the resulting integer w to a decoder, which in turn retrieves the sequence from w performing pop operations. Note that the symbols appear in the reverse order during the decoding, which is a distinctive feature of ANS. For their correct coordinated work, both encoder and decoder should receive in advance the same table of (approximate) frequencies of symbols in the sequence $a_1, a_2, \ldots, a_n$.

For didactic reasons, ANS encoders are usually first described in their unbounded form in which they operate on very long integers w, which is unrealistic. In practice, *streaming ANS encoders* are used, which maintain the value of w during the construction within a fixed range fitting into a machine word and they store all excessive bits in an output buffer by performing a "renormaliza-

tion" (akin to arithmetic encoders). However, in what follows, we significantly diverge from this standard way of explanation for ANS and introduce only the streaming ANS but in an unconventional manner; the unbounded ANS variant is not discussed at all. The unconventional description of ANS was chosen since it fits more naturally in our upper bound analysis of ANS.

3.1 Preliminary ANS

Suppose that we are to encode a sequence of symbols $a_1, a_2, \ldots, a_n$ from the alphabet $[0..\sigma)$. For each symbol $a \in [0..\sigma)$, denote by f_a its frequency in the sequence (i.e., its empirical probability is $\frac{f_a}{n}$). We assume that n is a power of two, i.e., $n = 2^r$ for an integer $r \geq 0$. It is a standard assumption for both ANS and arithmetic coding that simplifies implementations. (If the length of the sequence is not a power of two, then the real empirical probabilities of symbols are approximated with numbers of the form $f_a/2^r$, where 2^r is a power of two close to n, f_a are not necessarily real frequencies of symbols, and $\sum_a f_a = 2^r$; we discuss this case briefly in the end of Section 4.1 but most details are omitted as they are not in the scope of this paper.)

The streaming ANS encoder reads the symbols $a_1, a_2, \ldots, a_n$ from left to right and, after processing $a_1, a_2, \ldots, a_i$, encodes the processed part into a positive integer w_i. Initially, $w_0 = 2^r$; the choice for w_0 is somewhat arbitrary, the only necessary condition is $w_0 \geq 2^r$. To encode a new symbol a_{i+1}, we transform the number w_i into w_{i+1} by replacing at most $r+1$ highest bits of w_i with exactly $r+1$ bits that store a larger value than what was stored in the replaced bits; see Figure 1. Thus, $w_0 < w_1 < \cdots < w_{i+1}$. According to our terminology, we have $w_{i+1} = \mathsf{push}(w_i, a_{i+1})$. We view the number w_i as a bit stream, from highest to lowest bits of w.

$$\begin{aligned} w_i &= \underline{10010}\,011001000010110 \\ w_{i+1} &= \underbrace{\underline{1001011}}_{r+1 \text{ bits}}\,011001000010110 \end{aligned}$$

Figure 1: A transformation of the number w_i into w_{i+1}.

The numbers w_i might be very long. However, since only the highest $r+1$ bits of w_i matter for the encoder, all lower bits can be stored in an output buffer. The integer r is chosen so that the $r+1$ bits can be stored into one machine word. For simplicity of the exposition, we omit this technical detail and continue to discuss the numbers w_i as if they were stored explicitly.

The goal is to encode the whole sequence optimally so that, ideally, the final number w_n occupies $\sum_a f_a \log \frac{n}{f_a}$ bits. Intuitively, one can achieve this by encoding each symbol a into $\log \frac{n}{f_a} = r - \log f_a$ bits. During the processing of $a = a_{i+1}$, we could have achieved this by replacing the highest $\log f_a + 1$ bits of w_i with $r+1$ bits that store a new larger value; then, the number of bits in w_{i+1} and w_i will differ by $r + 1 - (\log f_a + 1) = \log \frac{n}{f_a}$. But the number $\log f_a$ is not integer in general. Therefore, instead, we replace either $\lfloor \log f_a \rfloor + 1$ or

$\lfloor \log f_a \rfloor + 2$ highest bits of w_i with new $r+1$ bits. As a result, the number of bits in w_{i+1} increases by either $r - \lfloor \log f_a \rfloor$ or $r - \lfloor \log f_a \rfloor - 1$, which is approximately $\log \frac{n}{f_a}$. The cumulative growth of w_n may approach the optimal $\log \frac{n}{f_a}$ bits per symbol a on average if the case when $\lfloor \log f_a \rfloor + 2$ bits are replaced happens more often.

What is the content of the $r+1$ new highest bits in w_{i+1} and how do we decide whether $\lfloor \log f_a \rfloor + 1$ or $\lfloor \log f_a \rfloor + 2$ highest bits of w_i will be replaced?

Denote by x' the value stored in the highest $r+1$ bits of w_{i+1}, i.e., $w_{i+1} = x' \cdot 2^{\lfloor \log w_{i+1} \rfloor - r} + \Delta$, where $0 \le \Delta < 2^{\lfloor \log w_{i+1} \rfloor - r}$ and $2^r \le x' < 2^{r+1}$. Denote by x the value of the highest bits of w_i that were replaced with x', i.e., $w_i = x \cdot 2^{\lfloor \log w_{i+1} \rfloor - r} + \Delta$, where $x \ge 1$. (In the example of Figure 1 x and x' are emphasized and Δ is a common part of w_i and w_{i+1}.) The scheme must be reversible: the number x' must provide sufficient information for the decoder to restore the symbol a_{i+1} and the number x. Since $\sigma \le \sum_a f_a = 2^r$, there is enough place to encode a_{i+1} into the r lowest bits of x' (the highest, $(r+1)$th, bit of x' is 1). "Excessive" $2^r - \sigma$ possible values of x' will be used to restore the number x, the replaced highest bits of w_i.

The r lowest bits of x' can store any number from the range $[0..2^r)$. The encoder chooses one of these numbers depending on the values of x and a_{i+1}. We distribute the numbers from $[0..2^r)$ among the symbols according to their frequencies: for each symbol a, denote $c_a = \sum_{a' \in [0..a)} f_{a'}$ (a cumulative frequency corresponding to a); the subrange of values $[c_a..c_{a+1})$ is allocated for a. Hence, given a symbol $a = a_{i+1}$, we have a room in the range that can store any number from $[0..f_a)$ and this information should suffice to restore the replaced number x from x'. As an example, Figure 2 depicts a distribution of the range $2^r = 2^4$ among symbols a, b, c, d with frequencies $3, 5, 6, 2$, respectively.

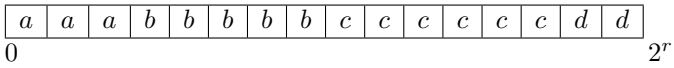


Figure 2: A distribution of symbols in a range of length 2^r.

Let $a = a_{i+1}$. Note that the number f_a occupies $\lfloor \log f_a \rfloor + 1$ bits. Denote by x_1 and x_2 the values stored in, respectively, the highest $\lfloor \log f_a \rfloor + 1$ and the highest $\lfloor \log f_a \rfloor + 2$ bits of w_i. We assume that $x = x_1$ if $x_1 \ge f_a$, and $x = x_2$ otherwise (see Figure 3, where the bits occupied by x_1 and x_2 are emphasized, respectively, on the left and right). Thus, the condition $x_1 \ge f_a$ determines whether $\lfloor \log f_a \rfloor + 1$ or $\lfloor \log f_a \rfloor + 2$ highest bits of w_i will be used for x. Note that $2^{\lfloor \log f_a \rfloor} \le x_1 < 2 \cdot f_a$ and $f_a < 2^{\lfloor \log f_a \rfloor + 1} \le x_2 \le 2 \cdot x_1 + 1$. If $x = x_1$ (i.e., $f_a \le x_1$), we have $f_a \le x < 2 \cdot f_a$. If $x = x_2$ (i.e., $f_a > x_1$), we have $x_2 \le 2 \cdot x_1 + 1 < 2 \cdot f_a$. Therefore, both cases imply $f_a \le x < 2 \cdot f_a$. Thus, $x \bmod f_a = x - f_a$.

In order to restore the value x from x', it suffices to encode somehow the value $x \bmod f_a$ into x': then, x is restored as $x = f_a + (x \bmod f_a)$. To this end, the range $[c_a..c_{a+1})$ allocated for the symbol a has exactly enough room. Thus,

$$\begin{aligned} w_i &= \underline{10110}011001000010110 \\ f_a &= 10011 \\ x \bmod f_a &= 11 \end{aligned} \qquad\qquad \begin{aligned} w_i &= \underline{10010}011001000010110 \\ f_a &= 10011 \\ x \bmod f_a &= 10001 \end{aligned}$$

Figure 3: Two cases: x occupies $\lfloor \log f_a \rfloor + 1$ highest bits of w_i (left) or $\lfloor \log f_a \rfloor + 2$ highest bits of w_i (right).

we have the following transformation to encode a:

$$x' = 2^r + c_a + (x \bmod f_a), \text{ where } f_a \le x < 2 \cdot f_a. \tag{3}$$

In other words, the operation $w_{i+1} = \mathsf{push}(w_i, a)$ assigns to a variable x the highest $\lfloor \log f_a \rfloor + 1$ or $\lfloor \log f_a \rfloor + 2$ bits of w_i (depending on the content of the bits, as was described above) and replaces these bits by x' computed in (3), where $x \bmod f_a = x - f_a$, thus transforming w_i into w_{i+1}. (In practice, instead of the entire w_i, the encoder stores only the highest $r + 1$ bits of w_i, which suffices for these calculations.)

With this approach, the decoder should perform a reverse transformation for the $r + 1$ highest bits of the number w_{i+1} in order to restore w_i. The decoding is straightforward:

$$x = f_a + (x' \bmod 2^r) - c_a, \text{ where } c_a \le x' \bmod 2^r < c_{a+1}. \tag{4}$$

The decoded symbol a is determined by examining to which range $[c_a..c_{a+1})$ the number $x' \bmod 2^r$ belongs. This is how the operation "$(w_i, a_{i+1}) = \mathsf{pop}(w_{i+1})$" is performed. It remains to observe that $x' > x$ since $x' \ge 2^r + (x \bmod f_a) > f_a + (x \bmod f_a) = x$. Therefore, the number w_{i+1} indeed is larger than w_i.

The described scheme is the simplest preliminary form of the so-called *range ANS (rANS)*. As will be seen later, the size of w_n in bits can be bounded by $\sum_a f_a \log \frac{n}{f_a} + O(n)$. The redundancy $O(n)$ is quite significant for many applications and, indeed, the described preliminary scheme does not perform well in practice and it is never used as is. To fix this issue, an additional "shuffling" step should be added to the encoder, which is described in the following section; the scheme with this step is called *tabled ANS (tANS)* or *Finite State Entropy (FSE)* and it has much smaller redundancy. There exists, however, a more elaborate version of the range ANS that also has a small redundancy and works well without shuffling; we discuss it briefly in Section 5 where we also present a novel variant of the range ANS with good compression guarantees.

3.2 Shuffling and Tabled ANS

The idea of the shuffling step enhancing the described preliminary ANS scheme is to shuffle the lower bits of x' in a random-like fashion. Thereby, the scheme (3) is changed as follows:

$$x' = 2^r + \mathsf{shuffle}[c_a + (x \bmod f_a)], \text{ where } f_a \le x < 2 \cdot f_a. \tag{5}$$

The array $\mathsf{shuffle}[0..2^r-1]$ is a permutation of the range $[0..2^r)$. Despite its name, the permutation is initialized not randomly but by a fixed deterministic algorithm, a *shuffling method*. Some methods are described in the next subsection. Each of the methods gives rise to a different ANS variant. Some of them use a pseudo-random generator with a fixed seed shared by the encoder and decoder, which somewhat justifies the name "shuffling". The permutation $\mathsf{shuffle}$ cannot be entirely random: in order to guarantee the inequality $x' > x$ (which implies $w_{i+1} > w_i$), it must satisfy the following property:

$$\mathsf{shuffle}[c_a + j] < \mathsf{shuffle}[c_a + k], \text{ whenever } 0 \le j < k < f_a. \tag{6}$$

Due to this condition, we have $x' > x$ since $x' \ge 2^r + (x \bmod f_a) > f_a + (x \bmod f_a) = x$. It is convenient to view $\mathsf{shuffle}$ as defined via an additional array $\mathsf{range}[0..2^r-1]$ that stores an (arbitrary) permutation of the array of symbols from Figure 2 in which every symbol a occurs exactly f_a times; then, for $j \in [0..f_a)$, $\mathsf{shuffle}[c_a + j]$ is equal to the index of the $(j+1)$th occurrence of the symbol a in range (see Figure 4). Thus, to define $\mathsf{shuffle}$, it suffices to initialize the array range; we will implicitly imply this relation in the sequel when the initialization of $\mathsf{shuffle}$ is discussed.

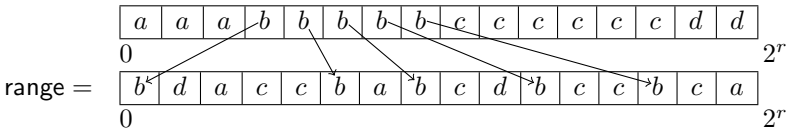


Figure 4: A shuffled distribution of symbols. Here, we have $\mathsf{shuffle}[c_b] = 0, \mathsf{shuffle}[c_b + 1] = 5, \mathsf{shuffle}[c_b + 2] = 7$.

The decoding procedure (4) performs a reverse transformation in an obvious way:

$$x = f_a + \mathsf{unshuffle}[x' \bmod 2^r] - c_a, \text{ where } \mathsf{range}[x' \bmod 2^r] = a.$$

Note the use of the array range in the decoder to determine the symbol a. The array $\mathsf{unshuffle}$ is the inverse of $\mathsf{shuffle}$ such that $\mathsf{unshuffle}[\mathsf{shuffle}[z]] = z$, for any $z \in [0..2^r)$. Thus, it "moves" the value $x' \bmod 2^r$ to its "correct" location and we have to add the frequency f_a and subtract the cumulative frequency c_a afterward. However, implementations usually construct, instead of the arrays $\mathsf{unshuffle}$ and range, an array decode that stores, for each number $x' \bmod 2^r$, an already corrected value for x and the corresponding symbol a; hence, the decoding is much simpler:

$$(x, a) = \mathsf{decode}[x' \bmod 2^r].$$

The described scheme is called a *tabled ANS (tANS)*. It is the most popular variant of ANS widely used in practice. The choice of the shuffling method is

crucial for its performance. Some methods are considered in the next section. There are several additional technical improvements that can be applied to this basic scheme. Perhaps, the most notable of them is that one can feed to the decoder more than $r+1$ bits at once, decoding many symbols in one step (the information about the decoded symbols and the new value for x must be stored in the array decode). Also, we point out again that $x \bmod f_a$ in (5) is computed as $x - f_a$ since $f_a \le x < 2 \cdot f_a$. We do not discuss these details further.

3.3 Shuffling Methods

The general rule for shuffling is to distribute symbols in the array range as uniformly as possible so that, for any symbol a, the distance between consecutive occurrences of a is approximately $\frac{n}{f_a}$. Implementations usually use heuristics for this [10] or Duda's method [2] (which is introduced below). Let us discuss some considerations on this regard that will be developed in a more rigorous way in Section 4.

The following informal argument suggests why the scheme with shuffling enhances the ANS so that it achieves the entropy bound (the rigorous analysis is in Section 4). The value $\log x' - \log x$ is, in a sense, an increase in bits from the number w_i to w_{i+1}; the bit length of w_n, the final encoding, is approximately the sum of the increases (this reasoning is formalized in Section 4). Denote $\delta = x \bmod f_a$. We have $x = f_a + \delta$. If the symbols in the array range are distributed uniformly, the distance between two consecutive symbols a is approximately $\frac{n}{f_a}$. Therefore, using (5), the encoder transforms x approximately to $x' \approx 2^r + \frac{n}{f_a}\delta = \frac{n}{f_a}(f_a + \delta) = \frac{n}{f_a}x$ (recall that $n = 2^r$). Hence, we obtain $\log x' - \log x \approx \log \frac{n}{f_a}$, which is precisely the optimal number of bits for the symbol a according to the entropy formula.

The argument suggests that the shuffling method should spread the symbols in the array range in such a way that the distance between two consecutive occurrences of a is approximately $\frac{n}{f_a}$ and the encoder transforms the number $x = f_a + \delta$ as close as possible to the number $2^r + \frac{n}{f_a}\delta$. Under this assumption of "uniformity", if the first occurrence of symbol a is at position p in range, then x is transformed into $x' \approx 2^r + \frac{n}{f_a}\delta + p$. The term p adds to the redundancy associated with f_a symbols a: the larger the value of p, the more bits are spent per symbol a. Therefore, the first occurrences of more frequent symbols should be closer to the beginning of the array range so that they produce less redundancies overall. Duda's method, which he called a *precise initialization*, tries to take into account all these considerations.

Duda's algorithm maintains a priority queue with the following operations: put(q, a) adds a pair of numbers (q, a) to the queue; $(q, a) =$ getmin() removes from the queue a pair (q, a) with the smallest value q (breaking ties arbitrarily). We first give in Algorithm 1 simplified Duda's algorithm, which is easier to analyze.

The array range corresponding to shuffle is not needed for the encoder and its construction is added here for the convenience of the reader.

Algorithm 1 A simple initialization algorithm.

for $a \in [0..\sigma)$ **do**
    $\mathsf{put}(0, a)$, $d_a = c_a$;
for $i = 0, 1, \ldots, 2^r - 1$ **do**
    $(q, a) = \mathsf{getmin}()$, $\mathsf{range}[i] = a$, $\mathsf{shuffle}[d_a] = i$;
    $\mathsf{put}(q + \frac{n}{f_a}, a)$, $d_a = d_a + 1$;

At every moment during the work of the algorithm, there is only one instance of each symbol in the priority queue and, if a symbol a was assigned to the array range exactly k times, then it is represented by the pair $(\frac{n}{f_a}k, a)$ in the queue. Therefore, after f_a assignments of a into range, the symbol is represented by (n, a) and all other symbols b that had less than f_b assignments in range are represented as (q, b) with $q < n$. Hence, in the end, each symbol a occurs in range exactly f_a times.

Duda's original "precise initialization" is the same as Algorithm 1 except that the operation "$\mathsf{put}(0, a)$" from the first loop is changed to "$\mathsf{put}(\frac{1}{2} \cdot \frac{n}{f_a}, a)$". Its correctness is proved analogously.

4 Analysis of the Tabled ANS

We are to estimate the redundancy of the output produced by the ANS encoder, i.e., the difference between $\lceil \log w_n \rceil$ and the lower bound $\sum_a f_a \log \frac{n}{f_a}$. We postpone the analysis of the ANS without shuffling to the next section, where its more general variant is considered. In this subsection we consider the ANS with shuffling, i.e., the tabled ANS (tANS). To the best of our knowledge, the following analysis, albeit quite simple, evaded the attention of researchers and was not present in prior works. Our proof methods, however, stem from observations and arguments from Dubé and Yokoo [22] and Duda [1, 2].

4.1 Upper Bounds

Let us upperbound $\log w_{i+1} - \log w_i = \log \frac{w_{i+1}}{w_i}$, a bit increase after one step of the encoding procedure. The total number of bits will be then estimated as follows (the term r appears because $w_0 = 2^r$ and $\log w_0 = r$):

$$\log w_n = \log \frac{w_n}{w_{n-1}} + \log \frac{w_{n-1}}{w_{n-2}} + \cdots + \log \frac{w_1}{w_0} + r. \tag{7}$$

Suppose that w_{i+1} was obtained from w_i by "inserting" a symbol a as described above. Denote $\ell = \lfloor \log w_{i+1} \rfloor - r$ so that $w_i = x \cdot 2^\ell + \Delta$ and $w_{i+1} = x' \cdot 2^\ell + \Delta$, where $0 \le \Delta < 2^\ell$. Then, $\log \frac{w_{i+1}}{w_i} = \log \left(\frac{x'2^\ell + \Delta}{x2^\ell + \Delta} \right) = \log \left(\frac{x'}{x} \left(\frac{1 + \Delta/(x'2^\ell)}{1 + \Delta/(x2^\ell)} \right) \right) = \log x' - \log x + \log \left(\frac{1 + \Delta/(x'2^\ell)}{1 + \Delta/(x2^\ell)} \right)$. Since $x' > x$, the additive term $\log \left(\frac{1 + \Delta/(x'2^\ell)}{1 + \Delta/(x2^\ell)} \right)$ is non-positive and, thus, we have obtained the

following inequality:

$$\log w_{i+1} - \log w_i \le \log x' - \log x. \tag{8}$$

It remains to estimate how close is $\log x' - \log x$ to the optimum $\log \frac{n}{f_a}$. We first consider the case when the encoder uses the shuffling produced by simplified Duda's algorithm (Algorithm 1).

Fix a symbol a and a number $\delta \in [0..f_a)$. Denote by k the index of the $(\delta+1)$th occurrence of a in range. Note that $\mathsf{shuffle}[c_a + \delta] = k$, by definition. For each $b \in [0..\sigma)$, denote by k_b the number of symbols b in the subrange $\mathsf{range}[0..k-1]$. Clearly, we have $k = \sum_b k_b$. The shuffling algorithm implies the following inequality:

$$(k_b - 1)\frac{n}{f_b} \le \delta \frac{n}{f_a}. \tag{9}$$

We express k_b from (9) as $k_b \le \delta \frac{f_b}{f_a} + 1$. Summing over all $b \in [0..\sigma)$, we deduce from this $k \le \delta \frac{n}{f_a} + \sigma$. It follows from (8) that, in order to analyze the number of bits per symbol produced by the encoder, we have to estimate $\log x' - \log x$, where, by (5), $x' = 2^r + \mathsf{shuffle}[c_a + (x \bmod f_a)]$. Assuming $\delta = x \bmod f_a$, we obtain $x = f_a + \delta$ and $x' = 2^r + k = n + k$. Therefore, $\log x' - \log x = \log(n+k) - \log x \le \log(n + \delta\frac{n}{f_a} + \sigma) - \log x = \log(\frac{n}{f_a}x + \sigma) - \log x = \log\frac{n}{f_a} + \log(1 + \frac{\sigma}{nx/f_a}) \le \log\frac{n}{f_a} + \frac{\sigma}{n}\log e$. Thus, we estimate the number of bits per symbol a as $\log\frac{n}{f_a} + \frac{\sigma}{n}\log e$, i.e., the redundancy is $\frac{\sigma}{n}\log e$ bits per symbols.

Now let us analyze Duda's original algorithm. The algorithm is the same as Algorithm 1 except that the operation "$\mathsf{put}(0, a)$" from the first loop is changed to "$\mathsf{put}(\frac{1}{2} \cdot \frac{n}{f_a}, a)$". The analysis is slightly more complicated. First, an equation analogous to (9) for this case looks as follows:

$$\left(k_b - \frac{1}{2}\right)\frac{n}{f_b} \le \left(\delta + \frac{1}{2}\right)\frac{n}{f_a}.$$

We then similarly deduce $k_b \le (\delta + \frac{1}{2})\frac{f_b}{f_a} + \frac{1}{2}$ and, summing over all b, $k \le (\delta + \frac{1}{2})\frac{n}{f_a} + \frac{\sigma}{2}$. Again, assuming $x = f_a + \delta$, it follows from this that $\log x' - \log x \le \log(\frac{n}{f_a}x + \frac{n}{2f_a} + \frac{\sigma}{2}) - \log x = \log\frac{n}{f_a} + \log(1 + \frac{1}{2x} + \frac{\sigma}{2nx/f_a}) \le \log\frac{n}{f_a} + (\frac{1}{2f_a} + \frac{\sigma}{2n})\log e$. The symbol a occurs in the sequence exactly f_a times. Hence, the redundancies $\frac{1}{2f_a}\log e$ for symbols a sum up to $\frac{1}{2}\log e$ over all these occurrence and, therefore, in the end we obtain $\frac{\sigma}{2n}\log e$ bits per symbol contributed by the terms $\frac{1}{2f_a}\log e$, for all symbols a, in the final encoding. Adding to this the $\frac{\sigma}{2n}\log e$ bits per symbol, we obtain $\frac{\sigma}{n}\log e$ redundant bits per symbol in the final encoding.

It remains to add the additive term r from (7) to the redundancy, which contributes $\frac{r}{n}$ bits per symbol, and the following theorem is proved.

Theorem 1. *Given a sequence $a_1, a_2, \ldots, a_n$ of symbols from an alphabet $[0..\sigma)$ such that each symbol a occurs in it f_a times and $n = 2^r$ for an integer r, the ANS encoder using [simplified] Duda's precise initialization transforms this sequence into a bit string of length*

$$\sum_{a \in [0..\sigma)} f_a \cdot \log\frac{n}{f_a} + O(\sigma + r),$$

where the $O(\sigma + r)$ term can be bounded by $\sigma \log e + r = \sigma \log e + \log n$. Thus, we have $O(\frac{\sigma+r}{n}) = O(\frac{\sigma+\log n}{n})$ redundant bits per symbol.

In practice, the length m of the encoded sequence $a_1, a_2, \ldots, a_m$ is not necessarily a power of two. A typical solution for this case is to approximate the real empirical probabilities f_a/m of symbols with approximate ones $\hat{f}_a/2^r$, where $\sum_a \hat{f}_a = 2^r$. The ANS encoder then processes the sequence as usually but using the "frequencies" $\hat{f}_a$ instead of f_a. The same analysis can be applied for this case: Equations (7) and (8) trivially hold and the value $\log x' - \log x$ is bounded in the same manner by $\log \frac{2^r}{\hat{f}_a} + \frac{\sigma}{2^r} \log e$, for simplified Duda's initialization, and by $\log \frac{2^r}{\hat{f}_a} + (\frac{1}{2\hat{f}_a} + \frac{\sigma}{2 \cdot 2^r}) \log e$, for Duda's initialization. Summing the redundancies over all m symbols, we obtain the following theorem (for simplicity, the theorem is stated only for simplified Duda's initialization).

Theorem 2. *Let $a_1, a_2, \ldots, a_m$ be a sequence of symbols from an alphabet $[0..\sigma)$ such that each symbol a occurs in it f_a times. Let the probabilities f_a/m be approximated by numbers $\hat{f}_a/n$ such that $\hat{f}_a$ are integers, $n = 2^r$, for an integer r, and $\sum_{a \in [0..\sigma)} \hat{f}_a = n$. The ANS encoder that uses the approximate probabilities and simplified Duda's initialization transforms this sequence into a bit string of length*

$$\sum_{a \in [0..\sigma)} f_a \cdot \log \frac{n}{\hat{f}_a} + O(\frac{\sigma m}{n} + r),$$

where the $O(\frac{\sigma m}{n} + r)$ term can be bounded by $\frac{\sigma m}{n} \log e + r$.

Theorem 2 can be used as follows to estimate the redundancy per symbol in terms of the real empirical distribution p, given by the probabilities $p_a = \frac{f_a}{m}$, and the approximate distribution $\hat{p}$, given by $\hat{p}_a = \frac{\hat{f}_a}{n}$. Denote the entropy of p and the Kullback–Leibler divergence between p and $\hat{p}$ as:

$$H(p) = \sum_{a \in [0..\sigma)} p_a \log \frac{1}{p_a}, \qquad D(p||\hat{p}) = \sum_{a \in [0..\sigma)} p_a \log \frac{p_a}{\hat{p}_a}$$

Then we derive:

$$\begin{aligned}
&\sum_{a \in [0..\sigma)} f_a \cdot \log \frac{n}{\hat{f}_a} + O(\frac{\sigma m}{n} + r) \\
&\quad = m \left[\sum_{a \in [0..\sigma)} p_a \cdot \log \frac{1}{\hat{p}_a} + O(\frac{\sigma}{n} + \frac{r}{m}) \right] \\
&\quad = m \left[\sum_{a \in [0..\sigma)} p_a \cdot \log \frac{1}{p_a} + \sum_{a \in [0..\sigma)} p_a \cdot \log \frac{p_a}{\hat{p}_a} + O(\frac{\sigma}{n} + \frac{r}{m}) \right] \\
&\quad = m \left[H(p) + D(p||\hat{p}) + O(\frac{\sigma}{n} + \frac{r}{m}) \right].
\end{aligned}$$

Thus, once n and σ are fixed and m tends to infinity, the redundancy per symbol can be bounded by $D(p||\hat{p}) + O(\frac{\sigma}{n})$. This is the same estimation that was obtained by Yamamoto and Iwata in [23] using different methods.

4.2 Lower Bound Example

Apparently, the r-bit redundancy incurred by the initial value w_0 is unavoidable in the described scheme. It is less clear whether an $O(\sigma)$ additive term is necessary in Theorem 1. An informal argument supporting that this is the case is as follows. Consider a sequence in which all symbols are (approximately) equiprobable, i.e., their frequencies f_a are $\sim\frac{n}{\sigma}$. The lower bound for the encoding of the sequence is $n \log \sigma$ bits. The array range constructed by Duda's initialization algorithm for the sequence looks (approximately) as n/σ blocks, each of which is of size σ and consists of consecutive symbols $0, 1, \ldots, \sigma - 1$. Hence, when the encoder receives a symbol $a_{i+1} = a \in [0..\sigma)$ during its work, it transforms the number $x = f_a + \delta$, where $\delta = x \bmod f_a$, occupying leading bits of w_i, into the number $x' = 2^r + \frac{n}{f_a}\delta + a = \frac{n}{f_a}x + a = \sigma x + a$ (note that $f_a = n/\sigma$, by our assumption). As in the previous section, one can deduce from this that $\log x' - \log x = \log \sigma + \log(1 + \frac{a}{\sigma x})$. Since $x < 2f_a = 2\frac{n}{\sigma}$ and $\frac{a}{2n} < 1$, the redundant additive term $\log(1+\frac{a}{\sigma x})$ can be estimated as $\log(1+\frac{a}{\sigma x}) \geq \log(1+\frac{a}{2n}) \geq \frac{a}{2n}$ (we used the inequality $\log(1+z) \geq z$, where $0 \leq z \leq 1$). Thus, the redundancy is approximately $\frac{a}{2n}$ bits per symbol a, which sums to the total redundancy of $\sum_a \frac{a}{2n} f_a = \sum_a \frac{a}{2\sigma} = \frac{\sigma-1}{4}$ bits over all symbols in the sequence.

This informal argument is only an intuition since the non-positive terms $\log\left(\frac{1+\Delta/(x'2^\ell)}{1+\Delta/(x2^\ell)}\right)$ that appear in the analysis of Section 4.1 could, in principle, diminish the described effect. Nevertheless, as we are to show, an $\Omega(\sigma)$ redundancy indeed appears in some instances.

Fix an even integer $r > 0$. Observe that $2^r \equiv 1 \pmod 3$ since r is even. Denote $n = 2^r$. The sequence under construction will contain $\sigma = (n-1)/3 + 1$ symbols $0, 1, \ldots, \sigma - 1$. Each symbol $a \in [0..\sigma-1)$ has exactly three occurrences in the sequence (i.e., $f_a = 3$) and the symbol $\sigma - 1$ occurs only once (i.e., $f_{\sigma-1} = 1$); note that $\sum_{a\in[0..\sigma)} f_a = 3(\sigma-1)+1 = n$. The entropy formula gives the following lower bound on the encoding size for the sequence:

$$(n-1)\log\frac{n}{3} + \log n = (n-1)(r - \log 3) + r = (n-1)(r - 1.58496...) + r. \quad (10)$$

It is straightforward that with such frequencies of symbols both Duda's initialization algorithm and its simplified variant (Algorithm 1) construct the same array range: the subrange range$[0..\sigma-1]$ contains consecutively the symbols $0, 1, \ldots, \sigma - 1$ (in this order) and the subranges range$[\sigma..2\sigma-2]$ and range$[2\sigma-1..n-1]$ are equal and both contain consecutively the symbols $0, 1, \ldots, \sigma - 2$ (in this order). Now let us arrange the symbols in the sequence $a_1, a_2, \ldots, a_n$.

The last symbol a_n is $\sigma - 1$. The rest, $a_1, a_2, \ldots, a_{n-1}$, consists of symbols $a \in [0..\sigma-1)$ whose frequencies are $f_a = 3$ (11_2 in binary). When the encoder

processes a symbol $a_{i+1} = a \in [0..\sigma{-}1)$ and modifies the number w_i representing the prefix $a_1, a_2, \ldots, a_i$, it replaces either two or three leading bits of w_i with new $r+1$ bits, thus producing the number w_{i+1}. The choice of whether to replace two or three bits depends on whether the two leading bits of w_i are 11 or 10, respectively (i.e., whether the two bits store a number less than $f_a = 3$ or not). We are to arrange the symbols $[0..\sigma{-}1)$ in the sequence $a_1, a_2, \ldots, a_{n-1}$ in such a way that the encoder chooses the two options alternatingly: it replaces three leading bits of w_i if i is even, and two bits if i is odd ($i = 0, 1, \ldots, n-2$). The total number of bits produced in this way is at least $\frac{n}{2}(r-2)+(\frac{n}{2}-1)(r-1)+r$ (the additive term r appears when the last symbol $a_n = \sigma - 1$ is encoded), which is equal to $(n-1)(r-1.5)+r-0.5$. Comparing this to (10), one can see that the encoding generated by ANS is larger than the optimum (10) by at least $(\log 3 - 1.5)(n-1) - 0.5 > 0.08496(n-1)$ (the estimation holds for large enough n, so that the term 0.5 disappears). By simple calculations, we deduce from the equality $\sigma = (n-1)/3+1$ that the redundancy $0.08496(n-1)$ is larger than $\frac{\sigma-1}{4}$. Let us describe an arrangement of symbols that produces such effect of "alternation".

The encoder consecutively transforms the initial value $w_0 = 2^r$ into $w_1, w_2, \ldots$ by performing the push operations: $w_{i+1} = \mathsf{push}(w_i, a_{i+1})$. Let us call the number $\lfloor w_i / 2^{\lfloor \log w_i \rfloor - r} \rfloor \bmod 2^r$ a *state*; it is the value stored in the highest $r+1$ bits of the number w_i currently processed by the encoder minus the highest bit 1. The range of possible states is $[0..2^r) = [0..n)$. We split this range into three disjoint segments numbered i, ii, iii: $[0..\sigma)$, $[\sigma..2\sigma-1)$, $[2\sigma-1..n)$, whose lengths are σ, $\sigma-1$, $\sigma-1$, respectively. The subarray $\mathsf{range}[0..\sigma{-}1]$ corresponding to the segment i contains all symbols $[0..\sigma)$; each of the subarrays $\mathsf{range}[\sigma..2\sigma-2]$ and $\mathsf{range}[2\sigma-1..n-1]$ corresponding to the segments ii and iii, respectively, contains all symbols $[0..\sigma{-}1)$.

Receiving the symbol $\sigma - 1$ (which occurs in the sequence only once), the encoder transforms any current state to the state $\sigma - 1$ (it is the index of the symbol $\sigma - 1$ in the array range). In order to describe how states are transformed by other symbols, let us split the state range $[0..2^r)$ into three segments called A, B, C: $[0..2^{r-2})$, $[2^{r-2}..2^{r-1})$, $[2^{r-1}..n)$, whose lengths are 2^{r-2}, 2^{r-2}, 2^{r-1}, respectively. The segments are related as follows: receiving a symbol from $[0..\sigma{-}1)$, the encoder translates the current state from the segment A (respectively, B, C) to a state from the segment ii (respectively, iii, i); see Figure 5 where this transition of states is illustrated by dashed lines connecting the corresponding segments.

Receiving $a_{i+1} = a \in [0..\sigma{-}1)$, the encoder replaces two leading bits of w_i with $r+1$ new bits if the current state is from the segment C; otherwise, it replaces three leading bits of w_i. This behaviour is determined by first bits of w_i, which are equal to 11... only for states from the segment C; see Figure 5. The execution of the encoder starts with the state 0, which belongs to the segment A. We are to arrange symbols in the sequence $a_1, a_2, \ldots, a_{n-1}$ so that, for odd i, the state corresponding to w_i belongs to the segment C, and for even i, to the segment A. The states in our arrangement will never belong to the segment B.

Since $\sigma + \sigma/2 - 1 = n/2$ and $n/2$ is the leftmost state from the segment C,

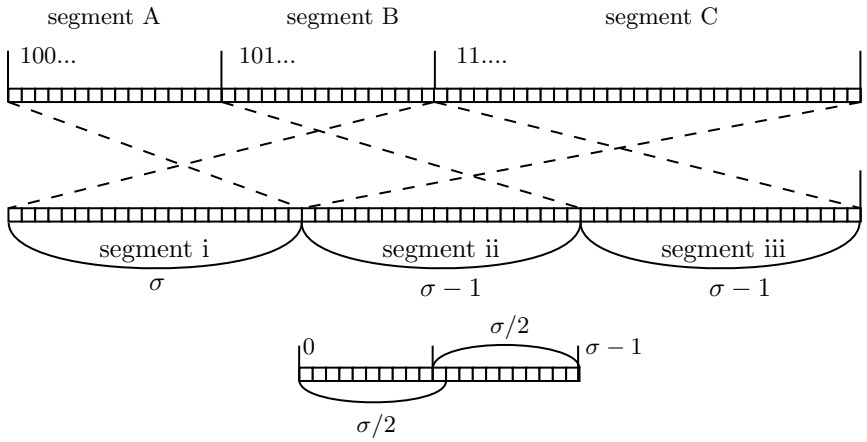


Figure 5: Two splits of the state range $[0..2^r)$ into segments. Here $r = 6$. Common first bits of numbers w_i corresponding to the states belonging to the segments A, B, C are written above the respective segments. Observe that the bits start with 11 only when a state belongs to the segment C. The range $[0..\sigma-1)$ is depicted under the segment ii: the encoder receives symbols from the right part $[\sigma/2 - 1..\sigma - 1)$ when the current state is from the segment A, and from the left part $[0..\sigma/2)$ when the state is from the segment C.

any state from the segment A transits to a state from the segment C when the encoder receives a symbol a from the range $[\sigma/2-1..\sigma-1)$; the new state is $\sigma+a$, the ath element of the segment ii (see Figure 5 for an illustration). Similarly, since $\sigma/2 < n/4$ and $n/4 - 1$ is the rightmost state from the segment A, any state from the segment C transits to a state from the segment A when the encoder receives a symbol $a \in [0..\sigma/2)$. Accordingly, for $i \in [0..n-1)$, we put in the sequence as the symbol a_{i+1} a symbol from $[\sigma/2 - 1..\sigma - 1)$ if i is even, and a symbol from $[0..\sigma/2)$ if i is odd (note that both ranges share a common symbol $\sigma/2 - 1$; it is not a mistake). The states thus "bounce" between the segments A and C during the processing of $a_1, a_2, \ldots, a_{n-1}$ by the encoder. Since the sizes of both ranges $[0..\sigma/2)$ and $[\sigma/2 - 1..\sigma - 1)$ are $\sigma/2$ and they share a common symbol $\sigma/2 - 1$, the symbols $[0..\sigma-1)$ can be distributed in the sequence $a_1, a_2, \ldots, a_{n-1}$ in such way that each symbol occurs exactly three times and each symbol a_{i+1} is from the range $[\sigma/2 - 1..\sigma - 1)$, for even i, and from the range $[0..\sigma/2)$, for odd i.

We thus have obtained a redundancy of $\frac{\sigma-1}{4}$ bits. Adding to this $r - 2$ bits produced by the $r - 2$ lowest bits of the initial value w_0, we have proved the following theorem.

Theorem 3. *For arbitrarily large $n = 2^r$ with even r, there exists an alphabet $[0..\sigma)$, where $\sigma > n/3$, and a sequence $a_1, a_2, \ldots, a_n$ of symbols from this alphabet such that the ANS encoder using [simplified] Duda's precise initialization*

transforms this sequence into a bit string of length at least

$$\sum_{a\in[0..\sigma)} f_a \cdot \log\frac{n}{f_a} + \frac{\sigma-1}{4} + r - 2,$$

where f_a is the number of occurrences for symbol a. Thus, the redundancy is $\frac{\sigma-1}{4} + r - 2 = \Theta(n)$ bits.

Note that Theorem 3 does not account for the space required to encode the table of symbol frequencies (which itself will be very large when $\sigma > n/3$) since, as it was stated in the preliminary section, the problem of storage for the frequency table is out of the scope of the present paper.

The example that attains the lower bound of Theorem 3 is simple (perhaps, unlike its tedious analysis). Hence, it is reasonable to assume that any adequate shuffling method would have the same redundancy $\Omega(\sigma+r)$ as in Theorem 3 on a similarly constructed sequence $a_1, a_2, \ldots, a_n$. We believe, therefore, that Duda's conjecture that the redundancy can be $O(\frac{\sigma}{n^2})$ bits per symbol when an appropriate shuffling method is used is disproved. This probably is not the case for algorithms that construct the shuffling tables after scanning the sequence first. However, such methods seem infeasible in practice due to incurring performance losses.

5 Range ANS with Fixed Accuracy

In Section 3.1, a simple range ANS (rANS) was described, which is just the ANS without shuffling. Now we are to introduce another variant of rANS, called *rANS with fixed accuracy* to distinguish it from the rANS as defined by Duda [2, 3] (which is also sketched below). Our exposition is less detailed than in the previous sections since we believe that all ideas and intuition necessary for understanding were developed above.

Let $a_1, a_2, \ldots, a_n$ be a sequence of symbols over an alphabet $[0..\sigma)$, where $n = 2^r$ and the frequencies of symbols are denoted by f_a (i.e., the empirical probability for symbol a is $\frac{f_a}{n}$). Fix an integer $k \geq 0$, which will serve as a user-defined accuracy parameter regulating the size of redundancy (typically, $0 \leq k \leq 4$). As in the simple rANS, the encoder starts its work with a number $w_0 = 2^{r+k}$; the value w_0 might be arbitrary, the only necessary condition is $w_0 \geq 2^{r+k}$. The encoder consecutively performs operations $w_{i+1} = \mathsf{push}(w_i, a_{i+1})$, for $i = 0, 1, \ldots, n-1$, but the operation $\mathsf{push}(w, a)$ works differently this time: it substitutes either $\lfloor \log f_a \rfloor + k + 1$ or $\lfloor \log f_a \rfloor + k + 2$ highest bits of w that store a number x by $r + k + 1$ new bits that store the number $x' = \lfloor x/f_a \rfloor 2^r + c_a + (x \bmod f_a)$, where $c_a = \sum_{b\in[0..a)} f_b$; the condition determining the number of bits occupied by x is essentially the same as in the simple rANS from Section 3.1.

More formally, the algorithm for $\mathsf{push}(w, a)$ is as follows. Denote by x_1 and x_2 the values stored in, respectively, the highest $\lfloor \log f_a \rfloor + k + 1$ and $\lfloor \log f_a \rfloor + k + 2$ bits of w. We assume that $x = x_1$ if $x_1 \geq f_a 2^k$, and $x = x_2$ otherwise. Note that $2^{\lfloor \log f_a \rfloor + k} \leq x_1 < f_a 2^{k+1}$ and $f_a 2^k < 2^{\lfloor \log f_a \rfloor + k + 1} \leq x_2 \leq 2 \cdot x_1 + 1$. If

$x = x_1$ (i.e., $f_a 2^k \le x_1$), we have $f_a 2^k \le x < f_a 2^{k+1}$. If $x = x_2$ (i.e., $f_a 2^k > x_1$), we have $x \le 2 \cdot x_1 + 1 < f_a 2^{k+1}$. Thus, both cases imply $f_a 2^k \le x < f_a 2^{k+1}$. Denote $\ell = \lfloor \log w \rfloor - \lfloor \log x \rfloor$. Note that $w = x \cdot 2^\ell + (w \bmod 2^\ell)$. The value $w' = \mathsf{push}(w, a)$ is computed as $w' = x' \cdot 2^\ell + (w \bmod 2^\ell)$ by replacing the part x of w by $r + k + 1$ bits representing a number x' defined as:

$$x' = \lfloor x / f_a \rfloor 2^r + c_a + (x \bmod f_a). \tag{11}$$

Since $f_a 2^k \le x < f_a 2^{k+1}$, we have $2^k \le \lfloor x/f_a \rfloor < 2^{k+1}$ and, therefore, the number x' indeed fits into $r + k + 1$ bits and its highest $(r + k + 1)$th bit is 1. The reverse operation $(w, a) = \mathsf{pop}(w')$ producing the old value w and the symbol a from w' is straightforward. Given a number x', which occupies $r + k + 1$ highest bits of w' (i.e., $x' = \lfloor w'/2^\ell \rfloor$, where $\ell = \lfloor \log w' \rfloor - r - k$), we first determine the symbol a by examining to which range $[c_a..c_{a+1})$ the number $x' \bmod 2^r$ belongs and, then, we compute x as follows:

$$x = \lfloor x'/2^r \rfloor f_a + (x' \bmod 2^r) - c_a. \tag{12}$$

Once x is known, we put $w = x \cdot 2^\ell + (w' \bmod 2^\ell)$. It remains to observe that $x' > x$ since $x' = \lfloor x/f_a \rfloor 2^r + (x \bmod f_a) > \lfloor x/f_a \rfloor f_a + (x \bmod f_a) = x$. Therefore, we have $w_0 < w_1 < \cdots < w_n$.

Note that, when $k = 0$, the described scheme degenerates simply to the ANS without shuffling.

To estimate the size of the final number w_n in bits, we derive by analogy to (7) and (8) using the condition $x' > x$ the following two equations (here we have $w_i = x \cdot 2^\ell + \Delta$ and $w_{i+1} = x' \cdot 2^\ell + \Delta$, where $\Delta \in [0..2^\ell)$):

$$\log w_n = \log \frac{w_n}{w_{n-1}} + \log \frac{w_{n-1}}{w_{n-2}} + \cdots + \log \frac{w_1}{w_0} + r + k;$$

$$\log w_{i+1} - \log w_i \le \log x' - \log x.$$

From (12), we deduce $\log x \ge \log(\lfloor x'/2^r \rfloor f_a) \ge \log f_a + \log(x'/2^r - 1) = \log f_a + \log(x'/2^r) + \log(1 - 2^r/x') = \log \frac{f_a}{2^r} + \log x' + \log(1 - 2^r/x')$. Since $2^{r+k} \le x'$, we derive further $\log(1 - 2^r/x') \ge \log(1 - 1/2^k) \ge -\frac{\log e}{2^k - 1}$ (due to the inequality $\ln(1 - z) \ge \frac{-z}{1-z}$, for $0 \le z < 1$). Therefore, we obtain

$$\log x' - \log x \le \log \frac{n}{f_a} + \frac{\log e}{2^k - 1}.$$

Now we derive $\log w_n = \sum_{i=0}^{n-1} (\log w_{i+1} - \log w_i) + r + k \le \sum_{i=0}^{n-1} (\log \frac{n}{f_{a_{i+1}}} + \frac{\log e}{2^k - 1}) + r + k = \sum_{a \in [0..\sigma)} f_a \cdot \log \frac{n}{f_a} + \frac{n \log e}{2^k - 1} + r + k$, getting the following theorem.

Theorem 4. *Given a sequence $a_1, a_2, \ldots, a_n$ of symbols from an alphabet $[0..\sigma)$ such that each symbol a occurs in it f_a times and $n = 2^r$ for an integer r, the rANS encoder with fixed accuracy $k \ge 1$ transforms this sequence into a bit string of length*

$$\sum_{a \in [0..\sigma)} f_a \cdot \log \frac{n}{f_a} + O\left(\frac{n}{2^k} + r + k\right),$$

where the $O\left(\frac{n}{2^k} + r + k\right)$ redundancy term can be bounded by $\frac{n \log e}{2^k - 1} + r + k$.

For completeness, let us briefly describe the standard rANS [2, 3]. The encoder similarly starts with the number $w_0 = 2^r$ and consecutively computes $w_1, w_2, \ldots, w_n$ for the sequence $a_1, a_2, \ldots, a_n$, where $n = 2^r$. The encoder maintains a number h (initially $h = r + 1$) and, receiving a new symbol, it replaces the highest h bits of the current number w_i with a larger value and increases h accordingly. A "renormalization" is sometimes performed by reducing the value h in order to contain numbers within a range fitting into a machine word. More precisely, receiving a symbol $a = a_{i+1}$, the encoder takes the value x stored in the h highest bits of w_i (i.e., $w_i = x \cdot 2^\ell + \Delta$, where $\ell = \lfloor \log w_i \rfloor + 1 - h$ and $\Delta \in [0..2^\ell)$), calculates a number x' by formula (11), and replaces x with x', thus producing $w_{i+1} = x' \cdot 2^\ell + \Delta$. After this, the value h is increased by $\lfloor \log x' \rfloor - \lfloor \log x \rfloor$. Once h is larger than a fixed threshold H, it is decreased but the resulting h should be larger than r. Usually, for performance reasons, encoders decrease h by a multiple of 8 or 16: $h = r + 1 + ((h - r - 1) \bmod b)$, where $b = 8$ or $b = 16$. Implementations maintain the h-bit number x, assigning $x = x'$ after processing each symbol, and the decreased number of bytes from x are "dumped" into an external stream. The decoder executes the same operations but in the reverse order computing x from x' as in (12).

The analysis of this rANS variant is not in the scope of the present paper; see [2, 20, 43].

Implementation notes The code of our implementation and tests is available at `https://github.com/dkosolobov/rans_with_fixed_accuracy`. See Table 1. We tested it against the standard rANS with explicit divisions and against a faster rANS with divisions performed by multiplications and shifts on precomputed constants; these two rANS variants were retrieved from the source code of `ryg_rans` by Giesen.[2] We note that neither of the four tested encoders implements the so-called interleaving streams: this technique can noticeably speed up the rANS [13] but it will do so equally for all four variants, if implemented, so we decided to omit it since, on a qualitative level, the results will likely remain the same with or without this technique.

All experiments were conducted on a laptop with AMD Ryzen 7 5800H (3.2 GHz, 512 Kb L1 cache, 16 Mb L3 cache) with 16 Gb RAM under 64-bit Windows 11 operating system. The code was compiled in Visual Studio 2019 using the `Clang` compiler; the essential flags of compilation were `-O2 -Oi -Ot -Gd -GS -arch:AVX2` (plus some technical flags included by the Visual Studio). The time was measured as average over five runs.

In our experiments the described rANS encoders with fixed accuracies $k = 3$ and $k = 2$ had approximately the same compression rate as the standard rANS (with a slight noticeable overhead for $k = 2$). In terms of speed, the rANS with accuracies $k = 3$ and $k = 2$ provide a faster encoding than the classical rANS with divisions but slower than the faster rANS implementation; however, the variant with $k = 2$ is very close to the faster rANS implementation. The rANS with accuracies $k = 3$ and $k = 2$ both were consistently slower in decoding.

[2] `https://github.com/rygorous/ryg_rans`

Table 1: Four rANS variants on three inputs. Each input has length 65536 with byte alphabet and it is generated either by a geometric or uniform distribution. The right column shows the compressed size (without frequency tables). For details, see `https://github.com/dkosolobov/rans_with_fixed_accuracy`.

Input	Entropy encoder	Comp/decomp time	Out bytes
Geom. dist. $p = 0.7$	rANS with acc 3:	341780/412960 ns	10366
	rANS with acc 2:	288740/399640 ns	10372
	rANS:	403560/286800 ns	10364
	rANS fast:	278200/286560 ns	10364
Geom. dist. $p = 0.3$	rANS with acc 3:	270120/391740 ns	24117
	rANS with acc 2:	237240/393480 ns	24135
	rANS:	317800/259460 ns	24112
	rANS fast:	218480/261520 ns	24112
Uniform dist.	rANS with acc 3:	266920/396760 ns	65521
	rANS with acc 2:	231240/392520 ns	65545
	rANS:	312480/252580 ns	65516
	rANS fast:	179380/253480 ns	65516

The key feature that allows the speed boost in encoding is that the operation of division $\lfloor x/f_a \rfloor$ in (11) guarantees that its result is in the range $[2^k..2^{k+1})$. Therefore, the division can be executed by simpler instructions in a branchless code: we used arithmetic and bit operations and the instruction `cmov` from x86 (it is also possible to use only arithmetic and bit instructions). The code, however, turns out to be quite cumbersome, which noticeably diminishes positive effects of the division-free branchless loop.

Denote $R = r + k$. The main loop of the encoder calls the function encode from Algorithm 2 consecutively for the symbols $a_1, a_2, \ldots, a_n$ in the encoded sequence. The function receives as its parameters an $(R+1)$-bit number w and a symbol a. The function stores some lowest bits of w in an external storage and returns an $(R+1)$-bit value x' computed as in (11) (details follow). The parameter w is actually an $(R+1)$-bit number x' produced by the previous call to the function encode in the encoding loop; the first call receives $w = 2^R$.

Denote $t = r - \lfloor \log f_a \rfloor$. The number x occupies either $R-t+1$ or $R-t+2$ highest bits of w. The presented pseudocode uses Collet's trick [10] to determine x with a branchless code. To this end, the array table stores, for each symbol $a \in [0..\sigma)$, besides the values c_a and f_a the number $d = (t << (R+1)) - (f_a << (t+k))$. The trick is that, in this case, the number $w + d = (t << (R+1)) + w - (f_a << (t+k))$ contains in its highest bits $R+1, R+2, \ldots$ either the number t or $t-1$ depending on whether $w \geq (f_a << (t+k))$ or not. Therefore, $x = w >> s$, where $s = (w + d) >> (R+1)$.

The code in lines 6–11 accumulates the quotient $\lfloor x/f_a \rfloor$ in bits $R+1, R+2, \ldots$ of the variable q and the remainder $x \bmod f_a$ in the variable x. It is done by subtracting the numbers $f_a << i$, for $i = 3, 2, 1, 0$, from x, thus, reconstructing q bit by bit; note, however, that the bits in q are inverted and, hence, in the

Algorithm 2 The encoding function of rANS with fixed accuracy $k = 3$.

1: **function** $\mathsf{encode}(w, a)$ ▷ rANS with fixed accuracy $k = 3$
2: $(c_a, f_a, d) = \mathsf{table}[a]$; ▷ $d = (t << (R+1)) - (f_a << (t+k))$, where $t = r - \lfloor \log f_a \rfloor$
3: $s = (w + d) >> (R + 1)$; ▷ Collet's trick
4: $\mathsf{outBits}(w, s)$; ▷ output s lowest bits of w
5: $x = w >> s$;
6: $x = x - (f_a << 3)$;
7: $q = 0$;
8: **for** $i = 2, 1, 0$ **do** ▷ the loop must be unrolled
9: $x_0 = x - (f_a << i)$;
10: **if** $(x_0 \geq 0)$ $x = x_0$; ▷ compiled to `cmov` on x86
11: $q = q$ **or** $(x_0$ **and** $(1 << (R + i)))$;
12: **return** $((q$ **xor** $(15 << R)) >> k) + c_a + x$;

end we have to perform xor with 15 (1111_2 in binary).

As is usual for such encoders, the function $\mathsf{outBits}(w, s)$ outputs s lower bits of w into a separate "buffer" machine word; this machine word is "dumped" into an external array in chunks of size 8 bits when too many bits are stored in it; see the details in `https://github.com/dkosolobov/rans_with_fixed_accuracy`.

As for the decoding process, it does not differ significantly from the decoding in the standard rANS and is quite straightforward; see the details in `https://github.com/dkosolobov/rans_with_fixed_accuracy`. However, the decoding has more instructions and, hence, is slightly slower.

Discussion. The standard rANS has a couple of advantages over tANS in some use cases [13]: it does not require a table of size 2^r like tANS and, hence, is more convenient for the (pseudo) adaptive mode when the table of frequencies is sometimes rebuilt during the execution of the encoder; due to this less heavy use of memory, the rANS might be better for interleaving several streams of data and utilizes more efficiently the instruction-level parallelism for this task. For these reasons, the rANS was used in some high performance compressors [13,44]. We note that in the (pseudo)adaptive setting the precomputation of constants for fast division is expensive and, hence, the standard rANS with explicit division is often used in this scenario. The described rANS with fixed accuracy shares the same good features of the standard rANS plus the described above benefits of the controlled division (at the expense of slower decoding, however).

An accurate integration of the rANS with fixed accuracy into existing adaptive encoders is an engineering task that we leave for a future work. One might expect that the speed up will be negligible if the time in the adaptive case is dominated by the updates of cumulative frequency tables. However, the (pseudo)adaptive setting is typically applied when the alphabet size is relatively small (at most 16), which allows to use SIMD instructions for fast updates of the cumulative frequencies, thus making the adaptive encoding not dramatically

slower than the non-adaptive encoding; e.g., see the TurboRC encoder in [44] and benchmarks therein. Still, in view of the above experimental results, only a minor speed up might be expected.

6 Conclusion and Open Problems

Theorems 1 and 3 describe the tight asymptotic behaviour of the redundancy for the tANS. We believe that Theorem 4, albeit not complemented with a lower bound, is asymptotically tight too. However, it is open to provided a series of examples supporting this claim. A number of other problems listed below still remain open too.

(i) The main remaining open problem concerning ANS encoders, as we see it, is to construct a genuine FIFO encoder with the same performance characteristics as tANS or rANS. The known ANS variants work as stacks (LIFO) while it is more natural and, in some scenarios, preferable to have an encoder that acts as a queue, like the arithmetic coding that fulfils this requirement but is noticeably worse than ANS in performance terms. For the same reason, the current ANS variants are not suitable enough for the adaptive encoding when frequencies of symbols change as the algorithm reads the sequence from left to right.

We note that the rANS can be applied for the adaptive mode as follows (for instance, see [44]): it first performs one left-to-right pass on the sequence to collect individual statistics per symbol adaptively and, then, encodes the sequence from right to left using these statistics; the decoder decodes the sequence from left to right adaptively computing the statistics by the same algorithm as was used by the encoder in the first pass. At first glance, this scheme is not much different from what the tANS or non-adaptive rANS do (note that they also perform a separate pass collecting statistics). Unfortunately, the adaptive rANS cannot utilize many optimizations that make a modern rANS comparable to tANS in terms of speed (in particular, the decoder cannot use SIMD and interleaving streams [13] as freely and cannot speed up divisions with special precomputed values). One of the goals for a FIFO encoder is to fix this issue.

(ii) Duda's precise initialization and its simplified variant from Algorithm 1 are not particularly suitable for practice due to the overhead incurred by operations on the priority queue and by floating point operations. Therefore, usually they are replaced with heuristics. An implementation of a fast and simple initialization method with good guarantees sufficient for Theorem 1 is an open problem.

(iii) The constants in our lower and upper bounds in Theorems 1 and 3 do not coincide and it remains open to find a tight constant for the tANS redundancy term. We believe also that the r-bit redundancy produced by the initial state w_0 of the encoder can be somehow reduced too by slightly modifying the scheme. Clearly, one can omit the trailing zeros in the bit representation of the resulting number w_n but it does not suffice to get rid of the r-bit redundancy entirely.

(iv) The encoding of the table of frequencies is rarely in the focus of theo-

retical research. However, it is very important in practice, mainly due to the following observation: a typical sequence fed to the entropy encoder is not homogeneous but consists of "chunks" drawn from different distributions; accordingly, an optimized compressor normally either uses an adaptive encoder (which seems best for this case but such encoders currently have issues described above) or splits the sequence into blocks and encodes each block separately, each with its own frequency table (e.g., see [45]). On relatively small chunks of data such as those produced in the latter case, the size of the frequency table is not negligible compared to the entropy of the data. For this reason, many practitioners resort to simpler Huffman encoders, which require much less space to encode their frequency tables. Surprisingly, this advantage in practice often suffices to compensate for the larger redundancy of Huffman compared to arithmetic or ANS encoders. Note that it is not an exceptional case: for instance, the Huffman encoder is *consistently superior* to arithmetic or ANS when encoding literals in LZ77 compressors (some LZ77 compressors often cautiously utilize the Huffman encoders in other parts as well, like to encode LZ77 phrase offsets and phrase lengths, without losses in the compression ratio; however, in most cases these parts still are processed by ANS or arithmetic encoders). Often the authors of compressors resort to the Huffman encoders after extensive experiments with ANS and arithmetic encoders, so the reason is not in simplicity.

Theoretical tight bounds are known for the size of encodings of the frequency tables: see [41], [40], [42], and references therein. However, works that investigate practical encoders for these tables are scarce. Ideally, such an encoder should take into account the entropy of the data to be able to relax some frequencies in order to make the total encoding size smaller, if necessary, thus never being worse than the Huffman encoders (see [17] and references in [46]). It seems that further investigations in this direction are needed.

Acknowledgments

I wish to thank the anonymous referees whose detailed reviews and suggestions helped to improve the paper (in particular, for pointing out that a recent result of [23] follows from Theorem 2).

Declarations

This work was supported by the Russian Science Foundation (RSF), project 24-71-00062. The author has no conflicts of interest to declare that are relevant to the content of this article.

References

[1] J. Duda. Asymmetric numeral systems. *arXiv preprint arXiv:0902.0271*, pages 1–47, 2009.

[2] J. Duda. Asymmetric numeral systems: entropy coding combining speed of huffman coding with compression rate of arithmetic coding. *arXiv preprint arXiv:1311.2540*, pages 1–24, 2013.

[3] J. Duda, K. Tahboub, N. J. Gadgil, and E. J. Delp. The use of asymmetric numeral systems as an accurate replacement for Huffman coding. In *2015 Picture Coding Symposium (PCS 2015)*, pages 65–69. IEEE, 2015.

[4] G. N. N. Martin. Range encoding: an algorithm for removing redundancy from a digitised message. In *Proc. Institution of Electronic and Radio Engineers International Conference on Video and Data Recording*, page 48, 1979.

[5] A. Moffat, R. M. Neal, and I. H. Witten. Arithmetic coding revisited. *ACM Transactions on Information Systems*, 16(3):256–294, 1998.

[6] R. C. Pasco. *Source coding algorithms for fast data compression*. PhD thesis, Stanford University CA, 1976.

[7] J. J. Rissanen. Generalized Kraft inequality and arithmetic coding. *IBM Journal of research and development*, 20(3):198–203, 1976.

[8] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.

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

[10] Y. Collet. FSE implementation. `https://github.com/Cyan4973/FiniteStateEntropy`. Accessed: 07.01.2022.

[11] Apple LZFSE compressor. `https://github.com/lzfse/lzfse`. Accessed: 07.01.2022.

[12] Y. Collet. Facebook Zstandard compressor. `https://github.com/facebook/zstd`. Accessed: 07.01.2022.

[13] F. Giesen. Interleaved entropy coders. *arXiv preprint arXiv:1402.3392*, pages 1–16, 2014.

[14] J. Alakuijala, R. van Asseldonk, S. Boukortt, M. Bruse, I-M. Comşa, M. Firsching, T. Fischbacher, E. Kliuchnikov, S. Gomez, R. Obryk, et al. JPEG XL next-generation image compression architecture and coding tools. In *Applications of Digital Image Processing XLII*, volume 11137, page 111370K. International Society for Optics and Photonics, 2019.

[15] T. Alonso, G. Sutter, and J. E. L. De Vergara. LOCO-ANS: An optimization of JPEG-LS using an efficient and low-complexity coder based on ANS. *IEEE Access*, 9:106606–106626, 2021.

[16] I. Blanes, M. Hernandez-Cabronero, J. Serra-Sagrista, and M. W. Marcellin. Redundancy and optimization of tANS entropy encoders. *IEEE Transactions on Multimedia*, pages 4341–4350, 2020.

[17] J. Duda. Encoding of probability distributions for asymmetric numeral systems. *arXiv preprint arXiv:2106.06438*, pages 1–5, 2021.

[18] A. Moffat and M. Petri. Large-alphabet semi-static entropy coding via asymmetric numeral systems. *ACM Transactions on Information Systems*, 38(4):1–33, 2020.

[19] N. Wang, C. Wang, and S-J. Lin. A simplified variant of tabled asymmetric numeral systems with a smaller look-up table. *Distributed and Parallel Databases*, 39(3):711–732, 2021.

[20] H. Yamamoto and K. Iwata. Encoding and decoding algorithms of ANS variants and evaluation of their average code lengths. *arXiv preprint arXiv:2408.07322*, 2024.

[21] H. Yokoo. On the stationary distribution of asymmetric binary systems. In *2016 IEEE International Symposium on Information Theory (ISIT)*, pages 11–15. IEEE, 2016.

[22] H. Yokoo and D. Dubé. Asymptotic optimality of asymmetric numeral systems. In *Proc. 42nd International Symposium on Information Theory and Its Applications (SITA 2019)*, pages 26–29, 2019.

[23] H. Yamamoto and K. Iwata. Asymmetric encoding-decoding schemes for lossless data compression. 2026.

[24] C. Bloom. cbloom rants blog. `http://cbloomrants.blogspot.com`. Accessed: 08.01.2022.

[25] Y. Collet. RealTime Data Compression blog. `http://fastcompression.blogspot.com`. Accessed: 08.01.2022.

[26] F. Giesen. The ryg blog. `https://fgiesen.wordpress.com`. Accessed: 07.01.2022.

[27] S. Rao Kosaraju and Giovanni Manzini. Compression of low entropy strings with Lempel–Ziv algorithms. *SIAM Journal on Computing*, 29(3):893–911, 1999.

[28] Giovanni Manzini. An analysis of the Burrows–Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.

[29] D. Dubé and H. Yokoo. Fast construction of almost optimal symbol distributions for asymmetric numeral systems. In *2019 IEEE International Symposium on Information Theory (ISIT)*, pages 1682–1686. IEEE, 2019.

[30] P. G. Howard and J. S. Vitter. Analysis of arithmetic coding for data compression. *Information processing & management*, 28(6):749–763, 1992.

[31] T. Strutz. Rescaling of symbol counts for adaptive rANS coding. In *2023 31st European Signal Processing Conference (EUSIPCO)*, pages 585–589. IEEE, 2023.

[32] R. Alverson. Integer division using reciprocals. In *IEEE Symposium on Computer Arithmetic*, pages 186–190, 1991.

[33] D. Lemire, C. Bartlett, and O. Kaser. Integer division by constants: optimal bounds. *Heliyon*, 7(6), 2021.

[34] T. M. Cover and J. A. Thomas. Information theory and statistics. *Elements of Information Theory*, 1(1):279–335, 1991.

[35] Imre Csiszár. The method of types. *IEEE Transactions on Information Theory*, 44(6):2505–2523, 2002.

[36] Travis Gagie. Large alphabets and incompressibility. *Information Processing Letters*, 99(6):246–251, 2006.

[37] B. McMillan. Two inequalities implied by unique decipherability. *IRE Transactions on Information Theory*, 2(4):115–116, 1956.

[38] N. Alon and A. Orlitsky. A lower bound on the expected length of one-to-one codes. *IEEE Transactions on information theory*, 40(5):1670–1672, 1994.

[39] W. Szpankowski and S. Verdu. Minimum expected length of fixed-to-variable lossless compression without prefix constraints. *IEEE Transactions on Information Theory*, 57(7):4017–4025, 2011.

[40] A. Orlitsky and N. P. Santhanam. Speaking of infinity [iid strings]. *IEEE Transactions on Information Theory*, 50(10):2215–2230, 2004.

[41] Yu. M. Shtar'kov. Universal sequential coding of single messages. *Problems of Information Transmission*, 23(3):3–17, 1987.

[42] W. Szpankowski and M. J. Weinberger. Minimax pointwise redundancy for memoryless models over large alphabets. *IEEE Transactions on Information Theory*, 58(7):4094–4104, 2012.

[43] J. Townsend. A tutorial on the range variant of asymmetric numeral systems. *arXiv preprint arXiv:2001.09186*, pages 1–11, 2020.

[44] TurboRC: Turbo range coder + rANS. `https://github.com/powturbo/Turbo-Range-Coder`. Accessed: 21.10.2023.

[45] J. Alakuijala, A. Farruggia, P. Ferragina, E. Kliuchnikov, R. Obryk, Z. Szabadka, and L. Vandevenne. Brotli: A general-purpose data compressor. *ACM Transactions on Information Systems (TOIS)*, 37(1):1–30, 2018.

[46] A. Painsky, S. Rosset, and M. Feder. Large alphabet source coding using independent component analysis. *IEEE Transactions on Information Theory*, 63(10):6514–6529, 2017.