# 9

# Integer Coding

Everything should be made as simple as possible, but no simpler

*Albert Einstein*



In this chapter we will address a basic encoding problem which occurs in many contexts, and whose efficient dealing is frequently underestimated for the impact it may have on the total space occupancy and speed of the underlying application [2, 8].

**Problem.** *Let $S = s_1, \ldots, s_n$ be a sequence of positive integers $s_i$, possibly repeated. The goal is to represent the integers of $S$ as binary sequences which are self-delimiting and use few bits.*

We note that the request about $s_i$ of being *positive integers* can be relaxed by mapping a positive integer $x$ to $2x$ and a negative integer $x$ to $-2x+1$, thus turning again the set $S$ to a set of just positive integers.

Let us comment two exemplar applications. Search engines store for each term $t$ the list of documents (i.e. Web pages, blog posts, tweets, etc. etc.) where $t$ occurs. Answering a user query, formulated as sequence of keywords $t_1 t_2 \ldots t_k$, then consists of finding the documents where all $t_i$s occur. This is implemented by intersecting the document lists for these $k$ terms. Documents are usually represented via integer IDs, which are assigned during the crawling of those documents from the Web. Storing these integers with a fixed-length binary encoding (i.e. 4 or 8 bytes) may require considerable space, and thus time for their retrieval, given that modern search engines index up to 20 billion documents. In order to reduce disk-space occupancy, as well as increase the amount of cached lists in internal memory, two kinds of compression tricks are adopted: the first one consists of sorting the document IDs in each list, and then encode each of them with the difference between it and its preceding ID in the list, the so called *d-gap*[1]; the second trick consists of encoding each d-gap with a variable-length sequence of bits which is short for small integers.

Another example of occurrence for the above problem relates to data compression. We have seen in Chapter 8 that the LZ77-compressor turns input files into sequence of triples in which the first two components are integers. Other known compressors (such as MTF, MPEG, RLE, BWT, etc.)

[1] Of course, the first document ID of a list is stored explicitly.

produce as intermediate output one or more sets of integers, with smaller values most probable and larger values increasingly less probable. The final coding stage of those compressors must therefore convert these integers into a bit stream, such that the total number of bits is minimized.

The main question we address in this chapter is how we design a variable-length binary representation for (unbounded) integers which takes as few bit as possible and is prefix-free, namely the encoding of $s_i$s can be concatenated to produce an output bit stream, which preserves decodability, in the sense that each individual integer encoding can be identified and decoded.

The first and simplest idea to solve this problem is surely that one to take $m = \max_j s_j$ and then encode each integer $s_i \in S$ by using $1 + \lfloor \log_2 m \rfloor$ bits. This fixed-size encoding is efficient whenever the set $S$ is not much spread and concentrated around the value zero. But this is a very unusual situation, in general, $m \gg s_i$ so that many bits are wasted in the output bit stream. So why not storing each $s_i$ by using its binary encoding with $1 + \lfloor \log_2 s_i \rfloor$ bits. The subtle problem with this approach would be that this code is not self-delimiting, and in fact we cannot concatenate the binary encoding of all $s_i$ and still be able to distinguish each codeword. As an example, take $S = \{1, 2, 3\}$ and the output bit sequence `11011` which would be produced by using their binary encoding. It is evident that we could derive many compatible sequence of integers from `11011`, such as $S$, but also $\{6, 1, 1\}$, as well as $\{1, 2, 1, 1\}$, and several others.

It is therefore clear that this simple encoding problem is challenging and deserves the attention that we dedicate in this chapter. We start by introducing one of the simplest integer codes known, the so called *unary code*. The unary code $U(x)$ for an integer $x \geq 1$ is given by a sequence of $x - 1$ bits set to 0, ended by a (delimiting) bit set to 1. The correctness of the condition that $x \neq 0$ is easily established. $U(x)$ requires $x$ bits, which is *exponentially longer* than the length $\Theta(\log x)$ of its binary code, nonetheless this code is efficient for very small integers and soon becomes *space inefficient* as $x$ increases.

This statement can be made more precise by recalling a basic fact coming from the Shannon's coding theory, which states that *the ideal code length $L(c)$ for a symbol $c$ is equal to $\log_2 \frac{1}{Pr[c]}$ bits, where $P[c]$ is the probability of occurrence of symbol $c$.* This probability can be known in advance, if we have information about the source emitting $c$, or it can be estimated empirically by examining the occurrences of integers $s_i$ in $S$. The reader should be careful in recalling that, in the scenario considered in this chapter, symbols are positive integers so the ideal code for the integer $x$ consists of $\log_2 \frac{1}{Pr[x]}$ bits. So, by solving the equation $|U(x)| = \log_2 \frac{1}{Pr[x]}$ with respect to $P[x]$, we derive the distribution of the $s_i$s for which the unary code is optimal. In this specific case it is $P[x] = 2^{-x}$. As far as efficiency is concerned, the unary code needs a lot of bit shifts which are slow to be implemented in modern PCs; again another reason to favor small integers.

***FACT 9.1*** *The unary code of a positive integer $x$ takes $x$ bits, and thus it is optimal for the distribution $P[x] = 2^{-x}$.*

Using this same argument we can also deduct that the fixed-length binary encoding, which uses $1 + \lfloor \log_2 m \rfloor$ bits, is optimal whenever integers in $S$ are *distributed uniformly* within the range $\{1, 2, \ldots, m\}$.

***FACT 9.2*** *Given a set $S$ of integers, of maximum value $m$, the fixed-length binary code represents each of them in $1 + \lfloor \log_2 m \rfloor$ bits, and thus it is optimal for the uniform distribution $P[x] = 1/m$.*

In general integers are not uniformly distributed, and in fact variable-length binary representations must be considered which eventually improve the simple unary code. There are many proposals in the literature, each offering a different *trade-off between space occupancy of the binary-code and*

*time efficiency for its decoding.* The following subsections will detail the most useful and the most used among these codes, starting from the most simplest ones which use *fixed encoding models* for the integers (such as, e.g., $\gamma$ and $\delta$ codes) and, then, moving to the more involved Huffman and Interpolative codes which use *dynamic* models that adapt themselves to the distribution of the integers in $S$. It is very well known that Huffman coding is *optimal*, but few times this optimality is dissected and made clear. In fact, this is crucial to explain some apparent contradictory statements about these more involved codes: such as the fact that in some cases Interpolative coding is better than Huffman coding. The reason is that Huffman coding is optimal among the family of *static* prefix-free codes, namely the ones that use a fixed model for encoding each single integer of $S$ (specifically, the Huffman code of an integer $x$ is defined according to $P[x]$). Vice versa, Interpolative coding uses a dynamic model that encodes $x$ according to the distribution of other integers in $S$, thus possibly adopting different codes for the occurrences of $x$. Depending on the distribution of the integers in $S$, this adaptivity might be useful and thus originate a shorter output bit stream.

## 9.1 Elias codes: $\gamma$ and $\delta$

These are two very simple *universal* codes for integers which use a fixed model, they have been introduced in the '60s by Elias [3]. The adjective "universal" here relates to the property that the length of the code is $O(\log x)$ for any integer $x$. So it is just a *constant factor* more than the optimal binary code $B(x)$ having length $1 + \lfloor \log x \rfloor$, with the additional wishful property of being prefix-free.

$\gamma$-code represents the integer $x$ as a binary sequence composed of two parts: a sequence of $|B(x)|-1$ zero, followed by the binary representation $B(x)$. The initial sequence of zeros is delimited by the 1 which starts the binary representation $B(x)$. So $\gamma(x)$ can be decoded easily: count the consecutive number of zeros up to the first 1, say they are $c$; then, fetch the following $c+1$ bits (included the 1), and interpret the sequence as the integer $x$.

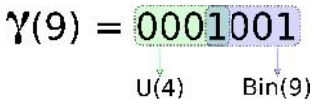


FIGURE 9.1: Representation for $\gamma(9)$.

The $\gamma$-code requires $2|B(x)|-1$ bits, which is $2(1+\lfloor \log_2 x \rfloor)-1 = 2\lfloor \log_2 x \rfloor + 1$. In fact, the $\gamma$-code of the integer 9 needs $2\lfloor \log_2 9 \rfloor + 1 = 7$ bits. From Shannon's condition on ideal codes, we derive that the $\gamma$-code is optimal whenever the distribution of the values follows the formula $Pr[x] \approx \frac{1}{2x^2}$.

***FACT 9.3*** *The $\gamma$-code of a positive integer $x$ takes $2\lfloor \log_2 x \rfloor + 1$ bits, and thus it is optimal for the distribution $P[x] \approx \frac{1}{2x^2}$, and it is a factor of 2 from the length of the optimal binary code.*

The inefficiency in the $\gamma$-code resides in the unary coding of the length $|B(x)|$ which is really costly as $x$ becomes larger and larger. In order to mitigate this problem, Elias introduced the $\delta$-code, which applies the $\gamma$-code in place of the unary code. So $\delta(x)$ consists of two parts: the first encodes $\gamma(|B(x)|)$, the second encodes $B(x)$. Notice that, since we are using the $\gamma$-code for $B(x)$'s length, the first and the second parts do not share any bits; moreover we observe that $\gamma$ is applied to $|B(x)|$ which guarantees to be a number greater than zero. The decoding of $\delta(x)$ is easy, first we decode $\gamma(|B(x)|)$ and then fetch $B(x)$, so getting the value $x$ in binary.

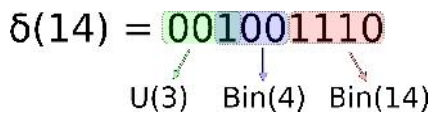


FIGURE 9.2: Representation for $\delta(14)$.

As far as the length in bits of $\delta(x)$ is concerned, we observe that it is $(1 + 2\lfloor \log_2 |B(x)| \rfloor) + |B(x)| \approx 1 + \log x + 2 \log \log x$. This encoding is therefore a factor $1 + o(1)$ from the optimal binary code, hence it is universal.

***FACT 9.4*** *The $\delta$-code of a positive integer $x$ takes about $1 + \log_2 x + 2 \log_2 \log_2 x$ bits, and thus it is optimal for the distribution $P[x] \approx \frac{1}{2x(\log x)^2}$, and it is a factor of $1 + o(1)$ from the length of the optimal binary code.*

In conclusion, $\gamma$- and $\delta$-codes are universal and pretty efficient whenever the set $S$ is concentrated around zero; however, it must be noted that these two codes need a lot of *bit shifts* to be decoded and this may be slow if numbers are larger and thus encoded in many bits. The following codes trade space efficiency for decoding speed and, in fact, they are preferred in practical applications.

## 9.2 Rice code

There are situations in which integers are concentrated around some value, different from zero; here, Rice coding becomes advantageous both in compression ratio and decoding speed. Its special feature is to be a *parametric code*, namely one which depends from a positive integer $k$, which may be fixed according to the distribution of the integers in the set $S$. The Rice code $R_k(x)$ of an integer $x$, given the parameter $k$, consists of two parts: the quotient $q = \lfloor \frac{(x-1)}{2^k} \rfloor$ and the remainder $r = x - 2^k q - 1$. The quotient is stored in unary using $q + 1$ bits, the remainder $r$ is stored in binary using $k$ bits. So the quotient is encoded in variable length, whereas the remainder is encoded in fixed length. The closer $2^k$ is to the value of $x$, the shorter is the representation of $q$, and thus the faster is its decoding. For this reason, $k$ is chosen in such a way that $2^k$ is concentrated around the mean of $S$'s elements.

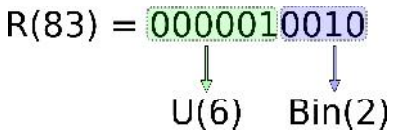


FIGURE 9.3: Representation for $R_4(83)$

The bit length of $R_k(x)$ is $q + k + 1$. This code is a particular case of the Golomb Code [8], it is optimal when the values to be encoded follow a geometric distribution with parameter $p$, namely $Pr[x] = (1 - p)^{x-1} p$. In this case, if $2^k \simeq \frac{ln(2)}{p} \simeq 0.69\ \texttt{mean}(S)$, the Rice and all Golomb codes generate an optimal prefix-code [8].

***FACT 9.5*** *The Rice code of a positive integer $x$ takes $\lfloor \frac{(x-1)}{2^k} \rfloor + 1 + k$ bits, and it is optimal for the geometric distribution $Pr[x] = (1 - p)^{x-1} p$.*

## 9.3 PForDelta code

This method for compressing integers supports extremely fast decompression and achieves a small size in the compressed output whenever $S$'s values follow a gaussian distribution. In detail, let us assume that most of $S$'s values fall in an interval $[base, base + 2^b - 1]$, we translate the values in the new interval $[0, 2^b - 1]$ in order to encode them in $b$ bits; the other values outside this range are called *exceptions* and they are represented in the compressed list with an *escape symbol* and also encoded explicitly in a separate list using a fixed-size representation of $w$ bits (namely, a whole memory word). The good property of this code is that all values in $S$ are encoded in fixed length, either $b$ bits or $w + b$ bits, so that they can be decoded very fast and possibly in parallel by packing few of them in a memory word.

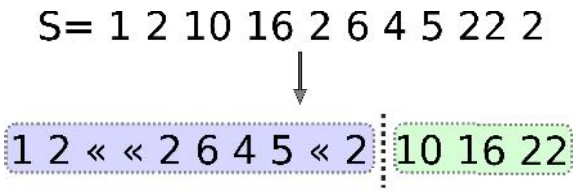


FIGURE 9.4: An example for PForDelta, with $b = 3$ and $base = 0$. The values in the range (blue box) are encoded using 3 bits, while the out-of-range values (green box) are encoded separately and an escape symbol $\ll$ is used as a place-holder.

***FACT 9.6*** *The PForDelta code of a positive integer $x$ takes either $b$ bits or $b+w$ bits, depending on the fact that $x \in [base, base + 2^b - 1]$ or not, respectively. This code is proper for a gaussian distribution of the integers to be encoded.*

The design of a PForDelta code needs to deal with two problems:

- *How to choose b:* in the original work, $b$ was chosen such that about the 90% of the values in $S$ are smaller than $2^b$. An alternative solution is to trade between space wasting (choosing a greater $b$) or space saving (more exceptions, smaller $b$). In [7] it has been proposed a method based on dynamic programming, that computes the optimal $b$ for a desired compression ratio. In particular, it returns the largest $b$ that minimizes the number of exceptions and, thus, ensures a faster decompression.
- *How to encode the escape character:* a possible solution is to assign a special bit sequence for it, thus leaving $2^b - 1$ configurations for the values in the range.

In conclusion PForDelta encodes blocks of $k$ consecutive integers so that they can be stored in a multi-word (i.e. multiple of 32 bits). Those integers that do not fit within $b$ bits are treated as exceptions and stored in another array that is merged to the original sequence of codewords during the decoding phase (thus paying $w+b$ bits). PForDelta is surprisingly succinct in storing the integers which occur in search-engine indexes; but the actual positive feature which makes it very appealing for developers is that it is incredibly fast in decoding because of the word-alignment and the fact that there exist implementations which do not use `if`-statements, and thus avoid *branch* mispredictions.

## 9.4 Variable-byte code and $(s, c)$-dense codes

Another class of codes which trade speed by succinctness is the one of the so called $(s, c)$-*dense codes*. Their simplest instantiation, originally used in the Altavista search engine, is the *variable-byte code* which uses a sequence of bytes to represent an integer $x$. This byte-aligned coding is useful to achieve a significant decoding speed. It is constructed as follows: the binary representation $B(x)$ is partitioned into groups of 7-bits, possibly the first group is padded by appending 0s to its front; a flag-bit is appended to each group to indicate whether that group is the last one (bit set to 0) or not (bit set to 1) of the representation. The decoding is simple, we scan the byte sequence until we find a byte whose value is smaller than 128.

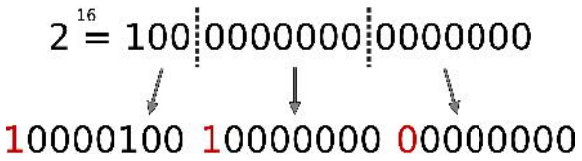


FIGURE 9.5: Variable-byte representation for the integer $2^{16}$

The minimum amount of bits necessary to encode $x$ is 8, and on average 4 bits are wasted because of the padding. Hence this method is proper for large values $x$.

***FACT 9.7*** *The Variable-byte code of a positive integer $x$ takes $\lceil \frac{|B(x)|}{7} \rceil$ bytes, and thus $8 * \lceil \frac{|B(x)|}{7} \rceil$ bits. This code is optimal for the distribution $P[x] \approx \sqrt[7]{1/x^8}$.*

The use of the status bit induces a subtle issue, in that it partitions the configurations of each byte into two sets: the values smaller than 128 (status bit equal to 0, called *stoppers*) and the values larger or equal than 128 (status bit equal to 1, called *continuers*). For the sake of presentation we denote the cardinalities of the two sets by $s$ and $c$, respectively. Of course, we have that $s + c = 256$ because they represent all possibly byte-configurations. During the decoding phase, whenever we encounter a continuer byte, we go on reading, otherwise we stop.

The drawback of this approach is that for any $x < 128$ we use always 1 byte. Therefore if the set $S$ consists of very-small integers, we are wasting bits. Vice versa, if $S$ consists of integers larger than 128, then it could be better to enlarge the set of stoppers. Indeed nobody prevents us to change the distribution of stoppers and continuers, provided that $s + c = 256$. Let us analyze how changes the number of integers which can be encoded with one of more bytes, depending on the choice of $s$ and $c$:

- One byte can encode the first $s$ integers;
- Two bytes can encode the subsequent $sc$ integers.
- Three bytes can encode the subsequent $sc^2$ integers.
- $k$ bytes can encode $sc^{k-1}$ integers.

It is evident, at this point, that the choice of $s$ and $c$ depends on the distribution of the integers to be encoded. For example, assume that we want to encode the values $1, \ldots, 15$ and they have decreasing frequency; moreover, assume that the word-length is 3 bits (instead of 8 bits), so that $s + c = 2^3 = 8$ (instead of 256).

Table 9.1 shows how the integers smaller than 15 are encoded by using two different choices for $s$ and $c$: in the first case, the number of stoppers and continuers is 4; in the second case, the number of

| Values | $s = c = 4$ | $s = 6, c = 2$ |
|---|---|---|
| 0 | 000 | 000 |
| 1 | 001 | 001 |
| 2 | 010 | 010 |
| 3 | 011 | 011 |
| 4 | 100 000 | 100 |
| 5 | 100 001 | 101 |
| 6 | 100 010 | 110 000 |
| 7 | 100 011 | 110 001 |
| 8 | 101 000 | 110 010 |
| 9 | 101 001 | 110 011 |
| 10 | 101 010 | 110 100 |
| 11 | 101 011 | 110 101 |
| 12 | 110 000 | 111 000 |
| 13 | 110 001 | 111 001 |
| 14 | 110 010 | 111 010 |
| 15 | 110 011 | 111 011 |

**TABLE 9.1** Example of $(s, c)$-encoding using two different values for $s$ and $c$.

stoppers is 6 and the number of continuers is 2. Notice that in both cases we correctly have $s+c = 8$. We point out that in both cases, two words of 3 bits (*i.e.* 6 bits) are enough to encode all the 15 integers; but, while in the former case we can encode only the first four values with one word, in the latter the values encoded using one word are six. This can lead to a more compressed sequence according to the skewness of the distribution of $\{1, \ldots, 15\}$.

This shows, surprisingly, that can be advantageous to adapt the number of stoppers and continuers to the probability distribution of $S$'s values. Figure 9.6 further details this observation, by showing the compression ratio as a function of $s$, for two different distributions `ZIFF` and `AP`, the former is the classic Zipfian distribution (i.e. $P[x] \approx 1/x$), the latter is the distribution derived from the words of the Associated-Press collection (i.e. $P[x]$ is the frequency of occurrence of the $x$-th most frequent word). When $s$ is very small, the number of high frequency values encoded with one byte is also very small, but in this case $c$ is large and therefore many words with low frequency will be encoded with few bytes. As $s$ grows, we gain compression in more frequent values and loose compression in less frequent values. At some later point, the compression lost in the last values is larger than the compression gained in values at the beginning, and therefore the global compression ratio worsens. That point give us the optimal $s$ value. In [1] it is shown that the minimum is unique and the authors propose an efficient algorithm to calculate that optimal $s$.

## 9.5 Interpolative code

This is an integer-encoding technique that is ideal whenever the sequence $S$ shows *clustered* occurrences of integers, namely subsequences which are concentrated in small ranges. This is a typical situation which arises in the storage of posting lists of search engines [7]. Interpolative code is designed in a *recursive* way by assuming that the integer sequence to be compressed consists of *increasing values*: namely $S' = s'_1, \ldots, s'_n$ with $s'_i < s'_{i+1}$. We can turn the original problem to this one, by just setting $s'_i = \sum_{j=1}^{i} s_j$.

At each iteration we know, for the current subsequence $S'_{l,r}$ to be encoded, the following 5 parameters:

- the left index $l$ and the right index $r$ delimiting the subsequence $S'_{l,r} = \{s'_l, s'_{l+1}, \ldots, s'_r\}$;

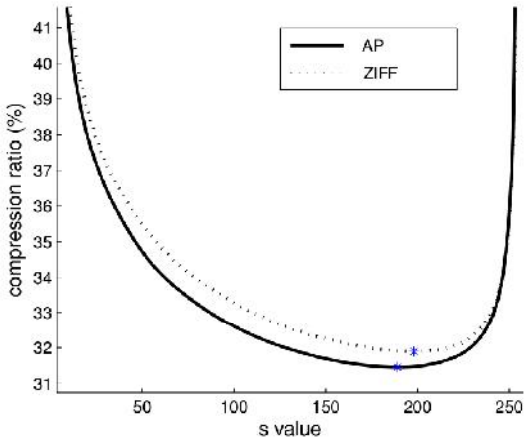


FIGURE 9.6: An example of how compression rate varies according to the choice of $s$, given that $c = 256 - s$.

- the number $n$ of elements in subsequence $S'_{l,r}$;
- a lower-bound *low* to the lowest value in $S'_{l,r}$, and an upper-bound *hi* to the highest value in $S'_{l,r}$, hence $low \leq s'_l$ and $hi \geq s'_r$.

Initially we have $n = |S|$, $l = 1$, $r = n$, $low = s'_1$ and $hi = s'_n$. At each step we first encode the middle element $s'_m$, where $m = \lfloor \frac{l+r}{2} \rfloor$, given the information available for the quintuple $\langle n, l, r, low, hi \rangle$, and then recursively encode the two subsequences $s'_l, \ldots, s'_{m-1}$ and $s'_{m+1}, \ldots, s'_r$, by using a properly recomputed parameters $\langle n, l, r, low, hi \rangle$ for each of them.

In order to succinctly encode $s'_m$ we deploy as much information as possible we can derive from $\langle n, l, r, low, hi \rangle$. Specifically, we observe that it is $s'_m \geq low + m - l$ (in the first half of $S'_{l,r}$ we have $m - l$ distinct values and the smallest one is larger than *low*) and $s'_m \leq hi - (r - m)$ (via a similar argument). Thus $s'_m$ lies in the range $[low + m - l, hi - r + m]$ so we can encode the value $s'_m - (low + m - l)$ by using $\lceil \log_2 \bar{l} \rceil$ bits, where $\bar{l} = hi - low - r + l$ is the size of that interval. In this way, interpolative coding can use very few bits per value whenever the sequence $S'_{l,r}$ is dense.

With the exception of the values of the first iteration, which must be known to both the encoder and the decoder, all values for the subsequent iterations can be easily derived from the previous ones. In particular,

- for the subsequence $s'_l, \ldots, s'_{m-1}$, the parameter *low* is the same of the previous step, since $s'_l$ has not changed; and we can set $hi = s'_m - 1$, since $s'_{m-1} < s'_m$ given that we assumed the integers to be distinct and increasing;
- for the subsequence $s'_{m+1}, \ldots, s'_r$, the parameter *hi* is the same as before, since $s'_r$ has not changed; and we can set $low = s'_m + 1$, since $s'_{m+1} > s'_m$;
- the parameters $l$, $r$ and $n$ are recomputed accordingly.

Figure 9.7 shows a running example of the behavior of the algorithm. We conclude the description of Interpolative coding by noticing that the encoding of an integer $s'_i$ is not fixed but depends on the distribution of the other integers in $S'$. This reflects onto the original sequence $S$ in such a way that the same integer $x$ may be encoded differently in its occurrences. This code is therefore *adaptive* and, additionally, it is *not* prefix-free; these two specialties may turn it better than Huffman code, which is optimal among the class of *static* prefix-free codes.

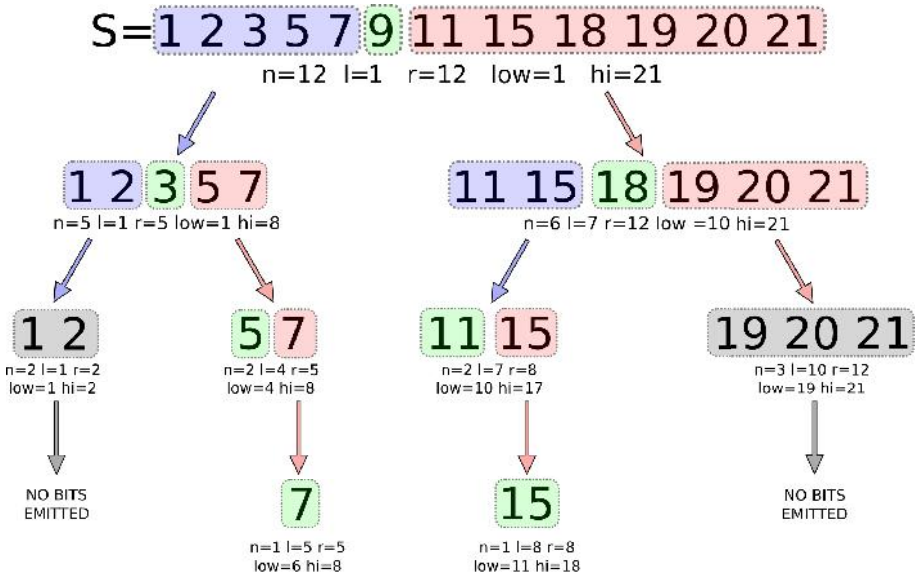


FIGURE 9.7: The blue and the red boxes are, respectively, the left and the right subsequence of each iteration. In the green boxes is indicated the integer $s'_m$ to be encoded. The procedure performs, in practice, a preorder traversal of a balanced binary tree whose leaves are the integers in $S$. When it encounters a subsequence of the form $[low, low+1, \ldots, low+n-1]$, it doesn't emit anything. Thus, the items are encoded in the following order (in brackets the actual number encoded): 9 (3), 3 (0), 5 (1), 7 (1), 18 (6), 11 (1), 15 (4).

## 9.6 Elias-Fano code

Unlike Interpolative coding, the code described in this section does not depend on the distribution of the integers to be encoded and, very importantly, it can be *indexed* (by proper compressed data structures) in order to efficiently *access randomly* the encoded integers. This is a positive feature in some settings, and a negative features in other settings. It is positive in the context of storing inverted lists of search engines and adjacency lists of large graphs (as it occurs in Facebook's Unicorn system); it is negative whenever integers occur *clustered* and space is a main concern of the underlying applications. Some authors [6] have recently proposed a (sort of) dynamic-programming approach that turns Elias-Fano coding into a *distribution-sensitive code*, like Interpolative code, thus combining its efficiency in randomly access the encoded integers and its space succinctness which derives from the possible clustering of these integers. Experiments have shown that Interpolative coding is only $2\% - 8\%$ smaller than the optimized Elias-Fano code but up to 5.5 times slower; and Variable-byte code is $10\% - 40\%$ faster than the optimized Elias-Fano code but $> 2.5$ times larger in space. This means that the Elias-Fano code is a competitive choice whenever an integer sequence must be compressed and (randomly) accessed.

As for the Interpolative code, Elias-Fano code works on a *monotonically increasing* sequence $S' = s'_1, \ldots, s'_n$ with $s'_i < s'_{i+1}$ and $s'_n < u$. We assume that each integer $s'_i$ is represented in binary over $b = \lceil \log_2 u \rceil$ bits. Let us consider a positive integer $\ell$, we partition the binary representation of $s'_i$ into two blocks: one is denoted by $H(s'_i)$ and consists of the $b - \ell$ most significant bits (the highest ones), whereas the other block is denoted by $L(s'_i)$ and consists of the $\ell$ less significant bits (the lowest ones). Clearly $|H(s'_i)| + |L(s'_i)| = b$ bits. The blocks $L(s'_i)$ are concatenated all together, in the order $i = 1, 2, \ldots, n$, thus forming the binary sequence $L$ whose length is $n\ell$ bits.

The blocks $H(s'_i)$ are encoded in an apparently strange way whose motivation will be clear when we will evaluate the overall space occupancy of the code. We start by observing that each block $H(s'_i)$ assumes values in $\{0, 1, 2, \ldots, \frac{u}{2^\ell} - 1\}$, each of these values is called *bucket*. So the Elias-Fano code iterates over the buckets $j = 0, 1, \ldots, \frac{u}{2^\ell} - 1$ and constructs the binary sequence $H$ by writing, for each bucket $j$, the negative unary representation of the number $x$ of elements $s'_i$ for which $H(s'_i) = j$: i.e., it writes $1^x\mathtt{0}$ (so that 1 is repeated $x$ times, with $x = 0$ if no $H(s'_i) = j$). The written binary sequence is denoted by $H$, its length is $n + \frac{u}{2^\ell}$ bits because we write $\frac{u}{2^\ell}$ negative-unary sequences (i.e. one per bucket) consisting of $\frac{u}{2^\ell}$ bits set to $\mathtt{0}$ (i.e. one bit set to $\mathtt{0}$ per bucket) and $n$ bits set to 1 (i.e. each $s'_i$ generates one bit set to 1 into some bucket $j$). The described process may be easily inverted so that, from $H$ and $L$, one can reconstruct $S'$. The total length of the Elias-Fano code is then $n\ell + n + \frac{u}{2^\ell}$ bits, which turns to be minimum for $\ell = \lceil \log \frac{u}{n} \rceil$.

***THEOREM 9.1*** *The Elias-Fano encoding of a monotonically increasing sequence of n integers in the range* $[0, u)$ *takes* $2n + n \log_2(u/n)$ *bits, regardless of their distribution. This is almost optimal (i.e.* $+2$ *bits per integer) if the integers are uniformly distributed in* $[0, u)$.

Figure 9.8 shows a running example of the coding process.

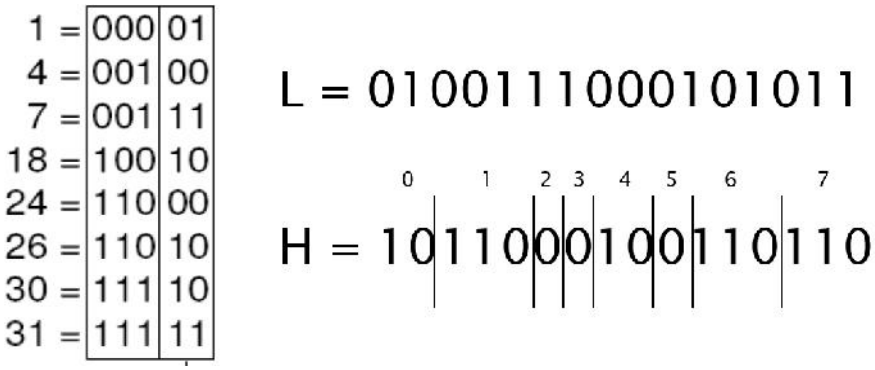


FIGURE 9.8: The Elias-Fano's code for the integer sequence $S' = 1, 4, 7, 18, 24, 26, 30, 31$. In this case $u = 32$ and $n = 8$, so that $\log_2 u = 5$ bits and $\ell = \log_2 n = 2$. Notice that the value $j = 1$ occurs twice as $H(s'_i)$, namely for the integers 4 and 7, so the binary sequence $H$ encodes $j = 1$ as $\mathtt{110}$, instead the value $j = 3$ does not occur as $H(s'_i)$ of any integer $s'_i$ and so the binary sequence $H$ encodes $j = 3$ as $\mathtt{0}$. The binary sequence $H$ consists of $\frac{u}{2^\ell} = 8$ unary sequences (and thus 8 bits set to $\mathtt{0}$) and $n = 8$ bits set to $\mathtt{1}$.

We can augment the Elias-Fano's coding of $S'$ to support efficiently the following two operations:

- Access(i) which, given an index $i \leq n$, returns $s'_i$;
- NextGEQ(x) which, given an integer $x \leq u$, returns the smallest element $s'_i$ which is greater than or equal to $x$.

We need to augment $H$ with an auxiliary data structure that efficiently, in time and space, answers a well-known primitive: $\mathsf{Select}_1(p, H)$ which returns the position in $H$ (counting from 1) of the $p$-th bit set to 1 (similarly it is defined $\mathsf{Select}_0(p, H)$). We do not want to enter here into the technicalities of the Select primitive, we content ourselves in pointing out that this query can be answered in constant time and $o(|H|) = o(n)$ bits of extra space (see [5] and refs therein). Given this

data structure built upon the binary array $H$ the two operations above can be implemented in $O(1)$ time as follows.

Access(i) needs to concatenate the higher and the lower bits of $s'_i$ present in $L$ and $H$, respectively. The block $L(s'_i)$ is easily retrieved by accessing the $i$-th block of $\ell$ bits in the binary sequence $L$. The retrieval of $H(s'_i)$ is a little bit more complicated and boils down to determine the rank of the negative unary sequence in $H$ which contains the 1 corresponding to $s'_i$; $H(s'_i)$ is eventually obtained by encoding that rank in $\log_2(u/n)$ bits. As an example consider Figure 9.8 and assume to execute Access(5). The 1 corresponding to the integer $s'_5$ in $H$, occurs at position 11 and falls in the 6th bucket; and in fact, $H(s'_5) = \texttt{110}$. More precisely the retrieval of $H(s'_i)$ first computes $\mathsf{Select}_1(i, H)$, which represents the position in $H$ of the 1 denoting $s'_i$; then, we substract $i$ (it is the $i$-th one!) to obtain the number of 0s occurring before that 1. By construction this value indicates the rank of the bucket which contains the 1 of $s'_i$. Referring again to the previous example, $L(s'_5) = \texttt{00}$ because this is the fifth block of $\ell = 2$ bits in $L$; and the retrieval of $H(s'_5)$ is obtained by computing $\mathsf{Select}_1(5, H) - 5 = 11 - 5 = 6$, and by encoding 6 in $\log_2(u/n) = 3$ bits as $H(s'_5) = \texttt{110}$. We have thus obtained $s'_5 = \texttt{110} \cdot \texttt{00} = 24$.

NextGEQ(x) is supported by observing that $p = \mathsf{Select}_0(H(x)) - H(x)$ is the number of integers in $S'$ whose highest bits are smaller than $H(x)$. Thus, $p + 1$ is the starting position in $S'$ of the elements whose highest bits are equal to $H(x)$ (if any) or larger. The integer answering NextGEQ(x) is then identified by scanning the elements at position $p + 1$ and beyond. If the element at position $p$ has its highest bits larger than $H(x)$, then we are done: it is enough to execute Access(p+1). Otherwise, the element at position $p + 1$ has the same highest bits as $x$, then we have to check whether there exist an element in that bucket which is larger or equal to $x$. So we compare the corresponding lowest bits in $L$ and scan them until an item $L(y) \geq L(x)$ is found in the bucket of $x$. But if not such element does exist, then we take the first item of the next bucket (which has larger highest bits than $H(x)$). As an example consider again Figure 9.8 and assume to execute NextGEQ(25). We compute $p = \mathsf{Select}_0(\texttt{110}) - \texttt{110} = \mathsf{Select}_0(6) - 6 = 10 - 6 = 4$, and then execute Access(p + 1) = Access(5) = 24, since this value is smaller than the queried 25 we scan the (usually small) bucket of integers whose highest bits are $H(x) = \texttt{110}$ until we meet the integer 26.

***FACT 9.8*** *It is possible to index the Elias-Fano encoding of a monotonically increasing sequence of n integers in the range* $[0, u)$*, taking* $o(n)$ *extra bits and support the random access to any of these integers (i.e.* Access(i) *operation) in constant time. Other operations (such as* NextGEQ(x)*) may be supported efficiently, too.*

A comment is in order at this point. Since Elias-Fano code represents a monotone sequence of integers regardless of its regularities, clustered sequences get significantly worse compression than what Interpolative code is able to achieve. Take, as an illustrative example, the sequence $S' = (1, 2, \ldots, n - 1, u - 1)$ of $n$ integers. This sequence is highly compressible since the length of the first run and the value of $u - 1$ can be encoded in $O(\log u)$ bits each. Conversely Elias-Fano code requires $2 + \log_2(u/n)$ bits per element. Some authors [6] have studied how to turn the Elias-Fano code into an *distribution-sensitive* code that takes advantage of the regularities present into the input sequence $S'$. They proposed two approaches. A simple one based on a two-level storage scheme: the sequence $S'$ is partitioned into $n/m$ chunks of $m$ integers each, then the "first level" is created by using Elias-Fano to encode the last integer of each chunk (we expect that these $m$ integers are well interspersed in $[0, u)$); then, the "second level" is created by using a specific Elias-Fano code on each chunk whose integers are delta-encoded with respect to the last integer of the previous chunk (available in the first level). This very simple scheme improves the space occupancy of the classic Elias-Fano code (which operates on the entire $S'$) by up to 30% but it slows down the decompression

time up to 10%; as far as Interpolative code is concerned, it worsen its space occupancy by up to 10% but it achieves three/four times faster decompression. A more sophisticated approach, based on a Shortest-Path interpretation of Elias-Fano's encoding of $S'$ on a suitably constructed graph, comes even closer in space to Interpolative code and still achieves very fast decompression.

## 9.7 Concluding remarks

We wish to convince the reader about the generality of the Integer Compression problem, because more and more frequently other compression problems, such as the classic Text Compression, boil down to compressing sequences of integers. An example was given by the LZ77-compressor in Chapter 8. Another example can be obtained by looking at any text $T$ as a sequence of tokens, being them words or single characters; each token can be represented with an integer (aka *token-ID*), so that the problem of compressing $T$ can be solved by compressing the sequence of token-IDs. In order to better deploy one of the previous integer-encoding schemes, one can adopt an interesting strategy which consists of sorting the tokens by decreasing frequency of occurrence in $T$, and then assign as token-ID their *rank* in the ordered sequence. This way, the more frequent is the occurrence of the token in $T$, the smaller is the token-ID, and thus the shorter will be the codeword assigned to it by anyone of the previous integer-encoding schemes. Therefore this simple strategy implements the *golden rule* of data compression which consists of assigning short codewords to frequent tokens. If the distribution of the tokens follows one of the distributions indicated in the previous sections, those codewords have optimal length; otherwise, the codewords may be sub-optimal. In [4] it is shown that, if the $i$-th word follows a Zipfian distribution, such as $P[i] = c(1/i)^{\alpha}$ where $c$ is a normalization constant and $\alpha$ is a parameter depending on the input text, then the previous algorithm using $\delta$-coding achieves a performance close to the *entropy* of the input text.

## References

[1] Nieves R. Brisaboa, Antonio Farina, Gonzalo Navarro, José R. Paramá. Lightweight natural language text compression. *Information Retrieval*, 10:1-33, 2007.

[2] Alistair Moffat. Compressing Integer Sequences and Sets. In *Encyclopedia of Algorithms*. Springer, 2009.

[3] Peter Fenwick. Universal Codes. In *Lossless Data Compression Handbook*. Academic Press, 2003.

[4] Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[5] Gonzalo Navarro and Eliana Providel. Fast, Small, Simple Rank/Select on Bitmaps. In *Procs of International Symposium on Experimental Algorithms (SEA)*, pp. 295-306, 2012.

[6] Giuseppe Ottaviano and Rossano Venturini. Partitioned Elias-Fano indexes. In *Procs of ACM SIGIR Conference*, pp. 273-282, 2014.

[7] Hao Yan, Shuai Ding, Torsten Suel. Inverted Index Compression and Query Processing with Optimized Document Ordering. In *Procs of WWW*, pp. 401-410, 2009.

[8] Ian H. Witten, Alistair Moffat, Timoty C. Bell. *Managing Gigabytes*. Morgan Kauffman, second edition, 1999.