8

Integer encoding

8.1	Unary code	8-2
8.2	Elias codes: γ and δ	8-2
8.3	Rice code	8-3
8.4	Interpolative coding	8-4
8.5	Variable-byte codes and (s,c)-dense codes	8-5
8.6	PForDelta encoding	8-7

scritto da: Tiziano De Matteis

Suppose that a sequence $S = s_1, \ldots, s_n$ has to be represented, where each symbol s_i is a positive integer, possibly unbounded. The goal of integer encoding techniques is to represent the integers of S over a binary output alphabet $\{0, 1\}$ using as few as possible output bits.
We may encounter this problem in many real situations. For example, in search engines, the information about the locations (the web pages) where a term occurs, are stored in inverted indexes: for each term t, the index contains an inverted list consisting of a number of index postings, usually the ID of the document, with some other information. Storing an index explicitly may require considerable space, but indexers can reduce demands on disk space and memory by using integer-encoding algorithms. Sometimes improvement is obtained by replacing each docID, except the first, with the difference between it and the previous one, the so called *d-gap*, and then encode the d-gap. This way, smaller integers have to be encoded with a possible reduction of the compressed output size.
Another example relates to data compressors: many of them produce as intermediate output one or more sets of integers (*e.g.* MTF, RLE, ...), with smaller values most probable and larger values increasingly less probable. The final coding stage must convert these integers into a bit stream, such that each value is self-delimiting and the total number of bits in the whole stream is minimized.

Why use a variable-length representation? The simplest code for a positive integer is its binary representation by fixed-length. If the sequence is bounded by a maximum value N, then the binary encoding takes $\lceil \log_2 N \rceil$ bits for each number. This might be too much if the integers to be encoded are skewed towards small values. From the Shannon's theory, ideal code length l_x for the symbol x and its probability $Pr[x]$ are related by the following equation:

$$l_x = \log_2 \frac{1}{Pr[x]} \tag{8.1}$$

Knowing the code length, we may determine the probability distribution for which that code is optimal. Namely we have:

$$Pr[x] = 2^{-l_x}$$

Hence, the implicit probability model associated with a fixed-length binary encoding is that each number will be uniformly distributed in $1, \ldots, N$ and this is often not a good reflection of reality. Thus, variable-length representations should be considered, in which smaller values (or, in general, the more frequent) are considered more likely and coded more economically than the larger (or less

© Paolo Ferragina, 2009-2012

frequent) ones. It goes without saying that Huffman encoding is optimal and could be applied in this content too by setting $\Sigma = \{1, 2, \ldots, N\}$. However it would require to store the model (of size $O(N \log N)$ bits) and to access it during the decoding phase (thus extra space and extra time are required). Conversely, the methods that we are going to discuss have an *implicit* model that has not to be stored and, in some cases (*e.g.* interpolative coding), could perform better than Huffman because they exploit some particular regularities of the sequence (they go beyond the 0-th order entropy H_0).

8.1 Unary code

This is one of the simplest codes: an integer $x \geq 1$ is encoded as $x - 1$ bits set to 0, ended by a bit set to 1. For example:

U(5) = 00001

FIGURE 8.1: Unary representation of 5

So the unary code requires x bits and thus strongly favors very small integers, resulting optimal whenever $Pr[x] = 2^{-x}$ (from Equation 8.1). This is a pretty much skewed distribution of the integers, indeed.

8.2 Elias codes: γ and δ

The γ-code represents the number x as a unary code for the value $length(x) = 1 + \lfloor \log_2 x \rfloor$ followed by the binary representation of x in $\lceil \log_2 x \rceil$ bits. The unary part specifies how many bits are required to code x (namely the length of x) and then the binary part actually codes x in that many bits. The final bit of the first field and the first bit of the binary representation (indeed, they are both 1) are shared, and thus the γ-code requires $2\lfloor \log_2 x \rfloor + 1$ bits.
For example, suppose that we want to code the integer 9. We have $length(9) = 1 + \lfloor \log_2 9 \rfloor = 4$ and thus:

γ(9) = 0001001

U(4) Bin(9)

FIGURE 8.2: γ representation of 9

From Equation 8.1 we derive that the γ-code is optimal whenever the distribution of x-values follows the formula:

$$Pr[x] = 2^{-l_x} = \frac{1}{2x^2}$$

The natural evolution of γ-code is δ-code: because the prefix of the γ-representation could be very long, it is useful to *recurse* and thus store $length(x)$ not in unary but via its γ-code. So, the δ representation of a number x is composed by two fields: the first is for the γ representation for the value $length(x)$ and the second for the binary representation of x.

For example, taking $x = 14$, we have length(x) = 4 and length(4)=3 and thus:

δ(14) = 001001110
U(3) Bin(4) Bin(14)

FIGURE 8.3: δ representation of 14

Notice that, since we are using the γ-code for x's length we can no longer hope that the first and the second fields share a bit, so they are distinguished. In fact, $Bin(4)$ ends with 0 whereas $Bin(14)$ starts with 1.

In general, in order to encode an arbitrary integer x, the δ-code requires

$$l_x = 2(\log_2 \mathit{length}(x)) + 1 + \lfloor \log_2 x \rfloor + 1 = \lfloor \log_2 x \rfloor + 2\lfloor \log_2 (1 + \lfloor \log_2 x \rfloor) \rfloor + 2 \text{ bits.}$$

From Equation 8.1 follows that the code is optimal when:

$$Pr[x] = 2^{-l_x} = \frac{1}{2x \log^2 x}$$

We notice that γ-code is a factor 2 from the optimal representation of an integer x (*i.e.* $\lfloor \log(x) \rfloor + 1$ bits), whereas δ-code is a factor

$$1 + O(\frac{\log \log x}{\log x}) = 1 + o(1)$$

optimal. This is the reason why γ and δ codes are called *Universal codes*, in that the representation they produce is a constant-factor from the optimal one. Nevertheless, these codes are not often used because they are particularly slow during the decoding phase, due to the numerous bits-operations required.

8.3 Rice code

There are situations in which integers are concentrated around some value, here Rice coding becomes advantageous both in compression ratio and decoding speed.
Rice code is a parametric code: fixed a positive integer k it encodes x computing the quotient $q = \lfloor \frac{(x-1)}{2^k} \rfloor$ and the remainder $r = x - 2^k q - 1$. Then the value is encoded in two parts: the quotient is stored in unary code using $q + 1$ bits and the remainder is stored in binary using k bits (field of fixed length). If k is less than or equal to the length of the machine word, this operation can be performed in constant time, since it is equivalent to bit shifts.

To encode an integer x, the rice code requires

$$l_x \leq \frac{x}{2^k} + k + 1 \text{ bits}$$

For example, suppose that we want to encode $x = 83$ with $k = 4$. So we have $q = \lfloor \frac{82}{16} \rfloor = 5$ and $r = 83 - 80 - 1 = 2$. The final representation is shown in Figure 8.4.

Rice code is useful when the integers are concentrated near 2^k: in this case the unary representation of q is short and thus fast decodable and the rest can be fetched in $O(1)$ time, being of fixed length.

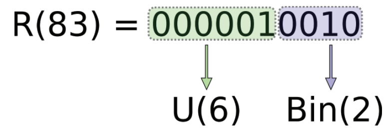

FIGURE 8.4: Rice representation for 83

This code is a particular case of the Golomb Code [4]. These codes are optimal when the x-values follow a geometric distribution corresponding to Bernoulli trials with the probability of success given by p. Namely we have:

$$Pr[x] = (1 - p)^{x-1} p$$

In this case, if 2^k is chosen so that

$$2^k \simeq \frac{ln(2)}{p} \simeq 0.69 mean(x)$$

where $mean(x)$ is the mean of the sequence to be encoded, these coding methods generate an optimal prefix code for that distribution.

8.4 Interpolative coding

The interpolative coding is a technique that is ideal for the types of clustered integer sequences that tipically arise in the storage of posting lists in search engines. This is a recursive coding that works on increasing integer sequences $s = s_1, \ldots, s_n$, so that $s_i < s_i + 1 \; \forall i < n$.
At each iteration we know, for the current subsequence to be encoded, the following parameters:

- the number n of elements in that subsequence;
- the left index l and the right index r, delimiting the subsequence (*i.e.* $s_l, s_{l+1}, \ldots, s_r$);
- a lower-bound *low* to the lowest value and an upper-bound *hi* to the highest value, hence $low \leq s_l$ and $hi \geq s_r$.

Initially we have $l = 1$, $r = n$, $low = s_1$ and $hi = s_n$. At each step we first encode the middle element s_m where $m = \lfloor \frac{l+r}{2} \rfloor$ 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 $[l, r, low, hi]$ for each of them. In order to encode the middle element, we deploy as much information as possible we can derive from the quadruple $[l, r, low, hi]$ so to use the fewest number of bits to encode it. Specifically, we observe that, for sure, it will be $s_m \geq low + m - l$ (in the first part of this subsequence we have $m - l$ distinct values and the smallest has value *low*) and $s_m \leq hi - (r - m)$ (same reasoning). Thus, s_m lies in the range $[low + m - l, hi - r + m]$ and we can encode it using $\lceil \log_2 \bar{l} \rceil$ bits, where $\bar{l} = hi - low - r + l$ is the size of that interval. We are actualy assuming that we are encoding $s_m - (low + m - l)$. In this way, interpolative coding can use very few bits per value for dense sequences or even zero bits whenever we have sequences of increasing numbers such as $[i, i + 1, \ldots, i + n - 1]$.

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, at the generic step of the encoding phase (but same reasoning hold for the decoding phase), we make the recursive call in the following way:

- 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, but $hi = s_m - 1$, since $s_{m-1} \leq s_m - 1$;
- for the subsequence $s_{m+1}, \ldots, s_r$ the parameter *hi* is the same as before, since s_r has not changed, but $low = s_m + 1$, since $s_{m+1} \geq s_m + 1$;

- the parameters l and r are modified accordingly.

The following figure shows a running example of the behavior of the algorithm:

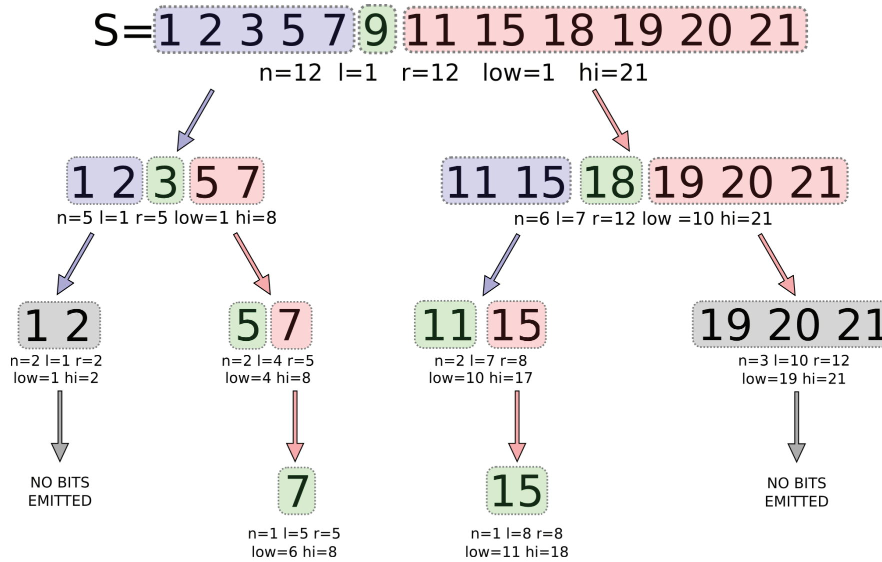

FIGURE 8.5: In the figure the blue and the red boxes are, respectively, the left and the right subsequence of each iteration. In the green boxes is indicated the number to be encoded. The procedure performs, in practice, a preorder traversal of a balanced binary tree. Notice that, when it encounters a subsequence in 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 (3), 5 (1), 7 (1), 18 (6), 11 (1), 15 (4).

8.5 Variable-byte codes and (s,c)-dense codes

Whenever speed is a primary concern, researcher have proposed different codes that trade compression ratio by decoding speed. Their main idea is to reduce as much as possible bit-operations on the integer encodings and thus force the codes to be of fixed-length or, possibly, byte aligned. An integer is represented in a variable number of bytes, where each byte consists of one status bit, followed by 7 data bits (padding if necessary). The status bit is zero if the actual byte is the last of the codeword, one otherwise. Figure 8.6 shows an exemple of this coding method.

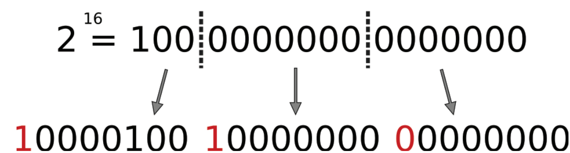

FIGURE 8.6: Var-byte representation for 2^{16}

The minimum amount of bits necessary to encode a number, is now one byte; thus, this method doesn't work very well for small values. In particular, in order to encode a value x it requires

$$l_x = \lceil \frac{length(x)}{7} \rceil \text{ bytes}$$

and thus we have an average waste of 4 bits per integer.

The use of the status bit induces a subtle issue, in that it partitions byte-configurations into two sets: the values smaller than 128 (status bit equal to 0, called stoppers hereafter) 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 s and c, with $s = c = 128$.
During the decoding phase, whenever we encounter a continuer byte, we continue to read, otherwise we stop. Instead of having the same number of continuers and stoppers, nothing prevent us to choose different values, provided that $s + c = 256$, just because we are working on 8 bits. In this way, if we have more stoppers, with one byte we may encode more items than in the previous case, but we reduce the number of items that can be encoded with more bytes. In particular, s items are encode with one byte, sc with two bytes, sc^2 with three and so on. Which values choose for s and c depend on the distribution of the numbers to be encode.
For example, assume that we want to encode a sequence containing the values $1, \ldots, 15$ of decreasing frequency. Supposing that our word-length is of 3 bits (instead of 8), The table below shows the encoded values using two different choices for s and c: in the first case the number of stoppers and continuers is 4; in the second, the number of stoppers is 6 and the number of continuers is 2. In

Values	$s = c = 4$	$s = 6, c = 2$
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	101 111	110 101
12	110 000	111 000
13	110 001	111 001
14	110 010	111 010
15	110 011	111 011

TABLE 8.1 Example of encoding using two different pairs (s, c)

both cases, two words (*i.e.* 6 bits) are enough to encode all the 15 numbers, 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 depending on the skewness of the distribution of $\{1, \ldots, 15\}$.
This example shows that can be advantageous to adapt the number of stoppers and continuers to the probability distribution of x-values.

Figure 8.7 shows the compression ratio as a funcion of s, for two different distributions of values. 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

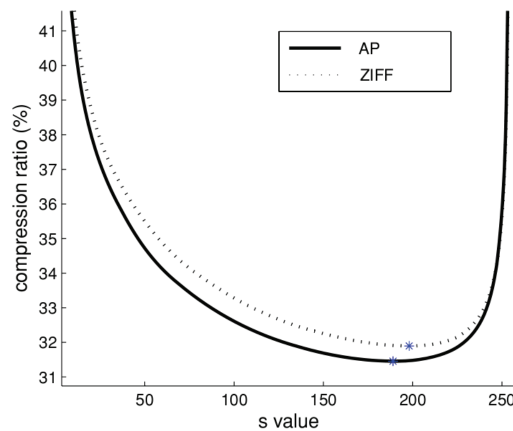

FIGURE 8.7: An example of how compression rate varies according to the choice of s

few bytes. From that point, as s grows, we gain compression in more frequent values and loose compression in less frequent ones. 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 [5] it is shown that the minimum is unique and an efficient algorithm is proposed to calculate it.

8.6 PForDelta encoding

This method for compressing integers has been recently proposed and supports extremely fast decompression achieving a small size in the compressed output. Suppose to have a gaussian distribution for the input values and that the majority of them fall in an interval $[base, base + 2^b - 1]$. We can translate the values in the new interval $[0, 2^b - 1]$ so to encode them in b bits. The values out of range (the so called *exceptions*), are encoded separately with a full representation, inserting in the compressed list an escape character where they occur.

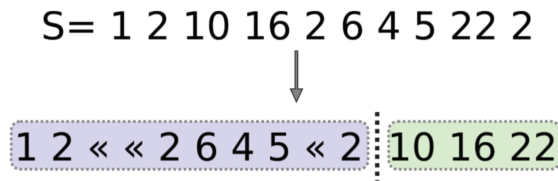

FIGURE 8.8: An example for PForDelta: suppose that $b = 3$ and $base = 0$. So the values in the blue box are encoded using 3 bits while the out of range values (green box) are encoded separately.

Now, two problems arise:

- How to choose b: in the original work, b was choosen such that about the 90% of the values 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 [3] it has been proposed an optimal method based on dynamic programming, that computes the optimal b for a desidered 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 $2^b - 2$ configurations for the values in the range.

References

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

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

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

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

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