1 Golomb Code

Let X be a random variable representing the number of times a coin is flipped until the outcome is heads. What is the best prefix-free encoding for X? If the coin is fair (heads and tails are equaly probable), it is easy to see that the *unary encoding* is the most adequate: the integer x should be represented by a sequence of $x-1$ ones followed by a zero. For example, 1 is encoded as `0`, 2 is encoded as `10`, 3 is encoded as `110`, and so on.

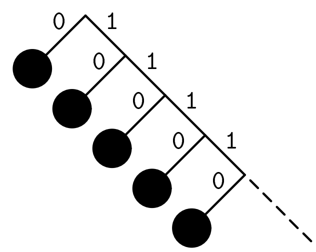

Figure 1: Optimal prefix-free encoding for a fair coin

Note that at any point of the coin-flipping experiment two outcomes are equally likely: we get heads (and stop) or we get tails (and flip the coin again). The first case is represented by `0`, the second by some string starting with `1`. In this sense, the tree above is perfectly balanced: for any given internal node, each branch is equally likely to be followed.

Now consider a more general case, where the probability of heads is some constant $p \in [0,1]$. For small values of p, the tree shown in Figure 1 no longer represents the best encoding for X. It would not be balanced anymore, since for any given internal node the probability of following the left branch (heads) would be significantly smaller than the probability of following the right branch (tails).

However, it is possible to change the tree in Figure 1 and obtain something with the same basic properties. Pick a value b such that

$$\text{prob}[X \leq b] \approx \text{prob}[X > b],$$

If we think of the first b symbols as a single block, we can do as in the previous case: represent them with strings starting with `0` and the remaining symbols with strings starting with a `1`. But note that the probabiliy distribution for the remaining symbols is the tail of an exponential distribution, and therefore is exponential itself, with the same ratio. Therefore, the same argument can be applied inductively, and we end up with the tree in Figure 2.

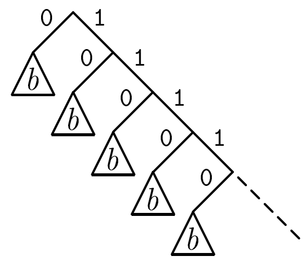

Figure 2: Golomb code (each triangle represents a set of b values)

This is the basic idea behind the *Golomb code*, which is optimal for this distribution. In practice, it is actually very simple. Given x, the positive integer we want to represent, define q to be the quotient and r to be the remainder of the division of $x-1$ by b:

$$\begin{aligned} q &= \frac{x-1}{b} \\ r &= x - q \cdot b - 1 \end{aligned}$$

To encode x, concatenate the unary representation of q with the binary representation of r. There are at most b possible remainders, which are encoded with either $\lceil \log b \rceil$ bits (for those that occur with lower probability) or $\lfloor \log b \rfloor$ bits (for the remaining ones).

1.1 Choosing b

Although the Golomb code could be defined for any integer b, we have seen that it is best to choose the value of b that results in a balanced prefix tree (with respect to the probabilities). We now show how this value of b can be derived from p.

First, note that the number of flips will be exactly some integer x if there are $x-1$ tails followed by heads. Therefore,

$$\text{prob}[x] = (1-p)^{x-1} p. \tag{1}$$

Given a certain number of flips ℓ, let us determine the probability that ℓ or more tosses are necessary until the outcome is heads:

$$\begin{aligned} \sum_{x \geq \ell} \text{prob}[x] &= \sum_{x \geq \ell} p\ (1-p)^x - 1 \\ &= p\ (1-p)^{\ell-1} \sum_{x \geq 0} (1-p)^x \\ &= p\ (1-p)^{\ell-1} (1/p) \\ &= (1-p)^{\ell-1} \end{aligned} \tag{2}$$

As mentioned before, b is a parameter chosen so as to make the probability that fewer then b coin flips are necessary be as close as possible to the probability that more than b coin flips are required. In other words, b must be the unique integer that satisfies the following inequality:

$$(1-p)^b + (1-p)^{b+1} \leq 1 < (1-p)^{b-1} + (1-p)^b \tag{3}$$

We can approximate this value using Equation 2:

$$\begin{aligned} (1-p)^b &\approx \frac{1}{2} \\ b \log \frac{1}{1-p} &\approx \log 2 \\ b &\approx \frac{\log 2}{\log \frac{1}{1-p}} \end{aligned}$$

For very small values of p, b can be further approximated by $(\log 2)/p$.

1.2 Optimality

The Golomb code was proven to be optimal for an exponential distribution by Gallager and Van Voorhin (1975). This section is based on their proof.

To simplify notation, let $\theta = 1 - p$. From Equation 1, we know that $p(i) = \theta^i(1-\theta)$ (we here assume that $i = 0, 1, 2, \ldots$, also to simplify notation). Furthermore, from equation 3 we know that

$$\theta^b + \theta^{b+1} \leq 1 < \theta^b + \theta^{b-1}. \tag{4}$$

To prove that Golomb encoding produces the optimal prefix-free code, our goal is to show that Huffman encoding leads to an expression similar to that above. Note, however, that we cannot do this directly, since the number of different values to be encoded is potentially infinite, while the algorithm that builds a Huffman code requires a finite set of probabilities.

To overcome this problem, we replace the original (infinite) source with a (finite) *m-reduced source*, which contains only $m + b + 1$ symbols. The first $m + 1$ symbols of the reduced source are exactly the same as the first $m + 1$ first symbols of the original source. Each of the remaining b symbols of the m-reduced source represent subsets of symbols of the original source, as shown in Figure 3:

$$\left|\begin{array}{c} m+1 \\ m+b+1 \\ m+2b+1 \\ m+3b+1 \\ \vdots \end{array}\right| \quad \left|\begin{array}{c} m+2 \\ m+b+2 \\ m+2b+2 \\ m+3b+2 \\ \vdots \end{array}\right| \quad \left|\begin{array}{c} m+3 \\ m+b+3 \\ m+2b+3 \\ m+3b+3 \\ \vdots \end{array}\right| \quad \cdots \quad \left|\begin{array}{c} m+b \\ m+b+b \\ m+2b+b \\ m+3b+b \\ \vdots \end{array}\right|$$

Figure 3: Last b symbols of an m-reduced source. Each symbol is a "bucket" containing infinite symbols of the original source.

In general, symbol $m+k$ (for $1 \leq k \leq b$) in the m-reduced source represents all symbols of the form $m+k+ib$ in the original source, for every $i > 0$. It is easy to see that the probability function p_m for the m-reduced source is as follows:

$$p_m(i) = \begin{cases} \theta^i(1-\theta) & \text{for } 0 \leq i \leq m \\ \frac{\theta^i(1-\theta)}{1-\theta^b} & \text{for } m < i \leq m+b \end{cases} \tag{5}$$

The first $m+1$ symbols are the same in both sources. Each of the remaining symbols of the reduced source has probability that is equal to the sum of the (infinitely many) corresponding symbols in the original source. From Equation 1 and Figure 3, we know that these probabilities form a geometric series with first term $\theta^i(1-\theta)$ and ratio θ^b between consecutive terms, and therefore adds up to the expression shown in Equation 5.

Lemma 1.1 *The two symbols with the smallest probabilities in the m-reduced sources are m and $m+b$.*

Proof. In the original source, probabilities are non-increasing. The first $m+1$ symbols of the m-reduced source have the same probability as the original source, which means they are also in non-increasing order. The same holds for the remaining b symbols of the m-reduced source $(m+1, m+2, \ldots, m+b)$, because of the way they are created. Therefore, we just have to prove that the second-to-last element of each of these subsequences is not smaller than the last element of the other subsequence. We prove each case independently.

a. $p_m[m] \leq p_m[m+b-1]$. From the probability distribution of the m-reduced source, we can rewrite this condition as

$$\begin{aligned} \theta^m(1-\theta) &\leq \frac{\theta^{m+b-1}(1-\theta)}{1-\theta^b} \\ 1-\theta^b &\leq \theta^{b-1} \\ 1 &\leq \theta^b + \theta^{b-1}, \end{aligned}$$

which is true according to Equation 4.

b. $p_m[m+b] \leq p_m[m-1]$. Again, we can rewrite this inequality as:

$$\begin{aligned} \frac{\theta^{m+b}(1-\theta)}{1-\theta^b} &\leq \theta^{m-1}(1-\theta) \\ \theta^{m+b} &\leq \theta^{m-1} - \theta^{m+b-1} \\ \theta^{b+1} &\leq 1-\theta b \\ \theta^b + \theta^{b+1} &\leq 1. \end{aligned}$$

This inequality is also true because of the definition of b (equation 4).

Together these two items are sufficient to prove the lemma. ■

This lemma implies that, when creating the Huffman code for the m-reduced source, the first pair of symbols to be combined will be m and $m+b$. The resulting symbol can be seen as an additional element in the set of symbols already represented by $m+b$ in the m-reduced source. The overall structure of the resulting set of symbols will be the same as that shown in Figure 3, but now there will be only $m-1$ symbols corresponding to a single symbol in the original distribution.

This means we will create an $(m-1)$-reduced source. Lemma 1.1 also holds for this new source (if we substitute $m-1$ for m, evidently). A simple inductive argument shows that we can keep combining the last two elements until we end up with a -1-reduced source, containing exactly b symbols with probabilities given by

$$p_{-1}(i) = \frac{\theta^i(1-\theta)}{1-\theta^b}, \text{ for } 0 \leq i \leq b-1.$$

In this distribution, the sum of the probabilities of the two least likely symbols is greater than the probability of the most likely symbol. As a result, the Huffman encoding of these symbols will result in an almost balanced tree, with every symbol represented with either $\lfloor \log b \rfloor$ or $\lceil \log b \rceil$ bits.

Optimality of the Golomb code. Let $\bar{\ell}_m$ be the average code length for the m-reduced source, and let $\bar{\ell}_G$ be the average code length resulting from the Golomb encoding. By definition, the probabilities in these two codes differ only for symbols beyond m. When we increase m, the sum of the probabilities for these symbols decreases. In particular, this sum goes to zero as m goes to infinity, which means that both codes end up modeling essentially the same set of symbols, with the same probability distribution:

$$\lim_{m\to\infty} \bar{\ell}_m = \bar{\ell}_G. \tag{6}$$

Given any encoding C, let ℓ_C denote its length. Morever, let $\bar{\ell}_C$ be the length of the of the smallest such encoding (for all possible encodings):

$$\bar{\ell}_C = \inf_C \ell_C \tag{7}$$

Note that $\bar{\ell}_C \leq \bar{\ell}_G$ by definition, since the Golomb code is just one particular code. Furthermore, since every prefix-free code can be converted into a code for the corresponding m-reduced source, it is true that $\bar{\ell}_m \leq \bar{\ell}_C$. We therefore have the following relation:

$$\bar{\ell}_m \leq \bar{\ell}_C \leq \bar{\ell}_G.$$

From Equation 6, we know that the two extremes of this relation converge as m goes to infinity. Since the relation holds for any value of m, it must be that $\bar{\ell}_C = \bar{\ell}_G$. We have therefore proven the following theorem:

Theorem 1.2 *The Golomb code is optimal for the exponential distribution.*

2 Indexing

In search engines, queries like "theory of computation" or "salt lake 2002" cannot be processed by matching each term with all documents in the database — this would be impractical (just think of a database consisting of all documents on the Web). Instead, an *index* is normally used. The original query is broken into *terms*, and each entry in the index relates a term in the database to a list of pointers to all documents in which the term occurs. Therefore, a query consists of retrieving the appropriate lists, determining their intersection, and ranking the resulting documents by some criterion.

The set of all possible terms in a database is called its *lexicon*. For example, it could be the set of all sequences of characters that are followed by a white space. In practice, however, it is interesting to keep the lexicon size within reasonable limits. Several strategies are used to achieve this goal:

- *Case-folding.* The lexicon is generally not case-sensitive: "olympics" and "Olympics" can be treated as a single term without any loss in meaning. (For some terms, however, case-sensitive queries may be relevant. This is the case of some acronyms, like SODA.)

- *Stemming.* Words that differ only in their suffix (or prefix) may be treated as a single term. For example, "compression", "compressor", and "compressing" could all be grouped under "compress".

- *Synonyms.* Synonyms are sometimes treated as a single term representing a "concept". For example, "fast", "quick", and "swift" could be taken as a single term.

- *Stop-words.* One could choose to select as terms only words that are descriptive (or meaningful) enough to represent documents. The word "complexity", for instance, is a good candidate to represent a book on Complexity Theory. Some other words, however, are clearly bad candidates: "a", "an", "the", "of", among others, appear in almost every document can hardly be used to select among them. These are called *stop-words*, and usually do not belong to the lexicon.

Another important design issue is the *granularity* of the index. Should each term be associated with a list of documents, or with a list of all occurrences within each document? Or maybe something in between? Clearly, an index containing just lists of documents tends to be smaller. On the other hand, if the index has more information, the number of documents that are actually looked at may be significantly reduced. There is a trade-off between space and time, which must be addressed when the index is being designed.

Here, we are interested on how an index is represented. For each term t, we need to store a list of all documents in which it occurs:

$$< f_t; d_1, d_2, d_3, \ldots, d_{f_t} >$$

The first term in the tuple above (f_t) is the *frequency* of term t: the number of documents in which t occurs. The remainder of the tuple are pointers to the documents. If N is the total number of pointers and we choose to represent each pointer with the same number of bits, $\lceil \log_2 N \rceil$ bits will be necessary for each pointer.

If we assume that the pointers are listed in non-decreasing order ($d_1 \leq d_2 \leq d_3 \leq \ldots \leq d_{f_t}$), the must be some gain simply by representing the *gaps* between consecutive pointers ($d_{k+1} - d_k$) instead of the pointers themselves. After all, the largest gap cannot be greater than N. However, we can do even better if we use coding schemes that take into account the fact that small gaps tend to be more common than large gaps.

We discuss several encoding schemes in the following subsections. In general, they can be classified as either *global* or *local*, depending on whether the same code is used for the entire file or not (different encodings are used for each term). Moreover, the encoding schemes can be either parameterized (they depend on the actual set of documents) or non-parameterized (there is a fixed encoding, regardless of the data set).

It is interesting to notice that each encoding scheme can be analyzed in terms of which assumptions about the distribution of the symbols. As we have seen before, if a certain symbol x appears with probability p_x, the number of bits used to represent it should be approximately

$$\ell_x \approx \log_2 \frac{1}{p_x}. \tag{8}$$

Therefore, for the values of ℓ_x, we will be able to estimate the underlying probability distribution of the methods discussed in the subsections that follow.

2.1 Binary Encoding

This is a global, non-parameterized model. It represents each gap using $\lceil \log_2 N \rceil$ bits. According to Equation 8, this scheme would be ideal for a uniform distribution, where the probability of occurrence for each symbol is roughly the same. For the application we are interested in (representing gaps in a sorted list of pointers), this is hardly the case, since small gaps tend to be more common than large gaps.

2.2 γ code

The γ code is also global and non-parameterized, but it employs variable-length encoding. The uses the fact that smaller numbers actually required fewer binary bits to be represented (for example, 3 can be represented with 2 bits, while 98 needs 7 bits). In the γ code starts with the length of the binary representation of a number in unary, followed by the binary representation itself (without the leading one). For instance, the binary number

```
101101
```

would be encoded as

```
11111001101.
```

From last to right, the first six digits (`111110`) represent the size of the binary representation (six bits) in unary (five `1`'s followed by a `0`, which marks the end of the first part of the representation). The remaining five digits (`01101`) are just the last five digits of the original number. There is no need to include the first number, because it will always be `1`. A few more examples: `1` is represented `0`, `10` by `100`, and `11` by `101`.

In particular, note that `0` cannot be directly represented in this method. This is usually not a problem, since the typical application for this method, as already mentioned, is to represent gaps between pointers, and these are never zero. If there is a need to represent zero, one can always "shift" the representation by one, using the code for $x+1$ to encode x.

Note that a number x whose standard binary representation requires s bits will be γ-encoded with $1+2(s-1)$ bits. More specifically, the number of bits ℓ_x used by this method to represent x is:

$$\ell_x = 1 + 2\lfloor \log_2 x \rfloor \text{ bits.}$$

Using Equation 8, we can determine the underlying probability distribution assumed by this method:

$$p_x \approx 2^{-\ell_x} = 2^{-(1+2\lfloor \log_2 x \rfloor)} = \frac{1}{2x^2}$$

2.3 δ code

The δ code (also global and non-parameterized) is very similar to the γ code. The only difference is that it replaces the unary representation of the length by the γ code representation itself. Therefore, the number of bits necessary to represent an integer x is[1]

$$\ell_x \approx (1 + 2\lfloor \log_2 \log_2 x \rfloor) + \log_2 x$$

According to Equation 8, this means that the underlying probability distribution for this method is given by

$$p_x \approx 2^{-(1+2\log_2 \log_2 x + \log_x 2)} \approx \frac{1}{2x(\log_2 x)^2}$$

Note that this method assigns greater probabilities to larger lenghts when compared to the γ code. In practice, this makes the δ code a better choice for larger number (for small numbers, however, it can actually be worse than the γ code). Decoding is reasonably easy for both methods, but slightly more straightforward in the γ code.

2.4 Parameterized Global Model

All methods discussed so far are non-parameterized: we define how each term should be represented regardless of how often it actually appears in the database being indexed. We now present a method that actually depends on the contents of the database.

[1] *This is what I had, but shouldn't there by a floor in the last $\log_2 x$ term?*

Let N be the total number of documents in the database, let n be the number of terms, and define f as the total number of gaps (or documents) in the database:

$$f = \sum_{\text{terms}} \#\text{occurrences}.$$

Furthermore, let

$$p = \frac{f}{nN}.$$

This value can be interpreted as a density of sorts for the gaps (note that $f \leq nN$). From this value of p, this method simply uses the Golomb code to represent the gaps.

2.5 Local Bernoulli Model

This method is similar to the previous one, but with a different Golomb encoding for each term. For term t, the parameter p_t for the encoding is

$$p_t = \frac{f_t}{N},$$

where N is the total number of documents and f_t is the number of documents in which term t appears.