CHAPTER 7

Lossless Compression

Computers can handle many different kinds of information like text, equations, games, sound, photos, and film. Some of these information sources require a huge amount of data and may quickly fill up your hard disk or take a long time to transfer across a network. For this reason it is interesting to see if we can somehow rewrite the information in such a way that it takes up less space. This may seem like magic, but does in fact work well for many types of information. There are two general classes of methods, those that do not change the information, so that the original file can be reconstructed exactly, and those that allow small changes in the data. Compression methods in the first class are called *lossless compression methods* while those in the second class are called *lossy compression methods*. Lossy methods may sound risky since they will change the information, but for data like sound and images small alterations do not usually matter. On the other hand, for certain kinds of information like for example text, we cannot tolerate any change so we have to use lossless compression methods.

In this chapter we are going to study lossless methods; lossy methods will be considered in a later chapter. To motivate our study of compression techniques, we will first consider some examples of technology that generate large amounts of information. We will then study two lossless compression methods in detail, namely *Huffman coding* and *arithmetic coding*. Huffman coding is quite simple and gives good compression, while arithmetic coding is more complicated, but gives excellent compression.

In section 7.3.2 we introduce the *information entropy* of a sequence of symbols which essentially tells us how much information there is in the sequence. This is useful for comparing the performance of different compression strategies.

7.1 Introduction

The potential for compression increases with the size of a file. A book typically has about 300 words per page and an average word length of four characters. A book with 500 pages would then have about 600 000 characters. If we write in English, we may use a character encoding like ISO Latin 1 which only requires one byte per character. The file would then be about 700 KB (kilobytes)[1], including 100 KB of formatting information. If we instead use UTF-16 encoding, which requires two bytes per character, we end up with a total file size of about 1300 KB or 1.3 MB. Both files would represent the same book so this illustrates straight away the potential for compression, at least for UTF-16 encoded documents. On the other hand, the capacity of present day hard disks and communication channels are such that a saving of 0.5 MB is usually negligible.

For sound files the situation is different. A music file in CD-quality requires 44 100 two-byte integers to be stored every second for each of the two stereo channels, a total of about 176 KB per second, or about 10 MB per minute of music. A four-minute song therefore corresponds to a file size of 40 MB and a CD with one hour of music contains about 600 MB. If you just have a few CDs this is not a problem when the average size of hard disks is approaching 1 TB (1 000 000 MB or 1 000 GB). But if you have many CDs and want to store the music in a small portable player, it is essential to be able to compress this information. Audio-formats like Mp3 and Aac manage to reduce the files down to about 10 % of the original size without sacrificing much of the quality.

Not surprisingly, video contains even more information than audio so the potential for compression is considerably greater. Reasonable quality video requires at least 25 images per second. The images used in traditional European television contain 576×720 small coloured dots, each of which are represented with 24 bits[2]. One image therefore requires about 1.2 MB and one second of video requires about 31MB. This corresponds to 1.9 GB per minute and 112 GB per hour of video. In addition we also need to store the sound. If you have more than a handful of films in such an uncompressed format, you are quickly going to exhaust the capacity of even quite large hard drives.

These examples should convince you that there is a lot to be gained if we can compress information, especially for video and music, and virtually all video formats, even the high-quality ones, use some kind of compression. With compression we can fit more information onto our hard drive and we can transmit information across a network more quickly.

[1] Here we use the SI prefixes, see Table 4.1.

[2] This is a digital description of the analog PAL system.

```
00000000000000000000000000000000000000000000
00000000000000000111000000000000000000000000
00000000000000000111000000000000000000000000
00000000000000000111000000000000000000000000
00000000000000000111000000000000000000000000
00000000000000000111000000000000000000000000
00000000000000000111000000000000000000000000
00000000000000000111000000000000000000000000
00000000000000000111000000000000000000000000
00000000000000000111000000000000000000000000
00000000000000000111000000000000000000000000
00000000000000000111000000000000000000000000
00000000000000000111000000000000000000000000
00000000000000000111000000000000000000000000
00000000000000000111000000000000000000000000
00000000000000000111000000000000000000000000
00000000000000000111000000000000000000000000
00000000000000000111000000000000000000000000
00000000000000000000000000000000000000000000
```

Figure 7.1. A representation of a black and white image, for example part of a scanned text document.

7.1.1 Run-length coding

We have already seen that English text stored in UTF-16 encoding can be compressed to at least half the number of bits by encoding in UTF-8 instead. Another simple strategy that sometimes work well is *run-length encoding*. One example is text documents scanned in black and white. This will produce an image of each page represented by a two-dimensional array of a large number of intensity values. At a point which is covered by text the intensity value will be 1, at all other points the value will be 0. Since most pages of text contain much more white than black, there will be long sequences of 0s in the intensity array, see figure 7.1. A simple way to compress such a file is to replace a sequence of n consecutive 0s by a code like n. Programs that read this file must then of course know that this code is to be interpreted as n consecutive 0s. As long as the sequences of 0s are long enough, this can lead to significant compression of the image file.

7.2 Huffman coding

The discussion in section 7.1 illustrates both the potential and the possibility of compression techniques. In this section we are going to approach the subject in more detail and describe a much used technique.

Before we continue, let us agree on some notation and vocabulary.

Definition 7.1 (Jargon used in compression)**.** *A sequence of symbols is called a* text *and is denoted* $\boldsymbol{x} = \{x_1, x_2, \ldots, x_m\}$*. The symbols are assumed to be taken from an* alphabet *that is denoted* $\mathscr{A} = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$*, and the number of times that the symbol* α_i *occurs in* $\boldsymbol{x}$ *is called its* frequency *and is denoted by* $f(\alpha_i)$*. For compression each symbol* α_i *is assigned a binary code* $c(\alpha_i)$*, and the text* $\boldsymbol{x}$ *is stored as the bit-sequence* $\boldsymbol{z}$ *obtained by replacing each character in* $\boldsymbol{x}$ *by its binary code. The set of all binary codes is called a* dictionary *or* code book*.*

If we are working with English text, the sequence $\boldsymbol{x}$ will just be a string of letters and other characters like $\boldsymbol{x}$ = {h,e, l, l, o, , a, g, a, i, n, .} (the character after 'o' is space, and the last character a period). The alphabet $\mathscr{A}$ is then the ordinary Latin alphabet augmented with the space character, punctuation characters and digits, essentially characters 32–127 of the ASCII table, see Table 4.3. In fact, the ASCII codes define a dictionary since it assigns a binary code to each character. However, if we want to represent a text with few bits, this is not a good dictionary because the codes of very frequent characters are no shorter than the codes of the characters that are hardly ever used.

In other contexts, we may consider the information to be a sequence of bits and the alphabet to be $\{0, 1\}$, or we may consider sequences of bytes in which case the alphabet would be the 256 different bit combinations in a byte.

Let us now suppose that we have a text $\boldsymbol{x} = \{x_1, x_2, \ldots, x_m\}$ with symbols taken from an alphabet $\mathscr{A}$. A simple way to represent the text in a computer is to assign an integer code $c(\alpha_i)$ to each symbol and store the sequence of codes $\{c(x_1), c(x_2), \ldots, c(x_m)\}$. The question is just how the codes should be assigned.

Small integers require fewer digits than large ones so a good strategy is to let the symbols that occur most frequently in $\boldsymbol{x}$ have short codes and use long codes for the rare symbols. This leaves us with the problem of knowing the boundary between the codes. Huffman coding uses a clever set of binary codes which makes it impossible to confuse the codes even though they have different lengths.

Fact 7.2 (Huffman coding)**.** *In Huffman coding the most frequent symbols in a text* $\boldsymbol{x}$ *get the shortest codes, and the codes have the* prefix property *which means that the bit sequence that represents a code is never a prefix of any other code. Once the codes are known the symbols in* $\boldsymbol{x}$ *are replaced by their codes and the resulting sequence of bits* $\boldsymbol{z}$ *is the compressed version of* $\boldsymbol{x}$*.*

Example 7.3. This may sound a bit vague, so let us consider an example. Suppose we have the four-symbol text $\boldsymbol{x}$ = DBACDBD of length 7. We note that

the different symbols occur with frequencies $f(A) = 1$, $f(B) = 2$, $f(C) = 1$ and $f(D) = 3$. We use the codes

$$c(D) = 1, \quad c(B) = 01, \quad c(C) = 001, \quad c(A) = 000. \tag{7.1}$$

We can then store the text as

$$\boldsymbol{z} = 1010000011011, \tag{7.2}$$

altogether 13 bits, while a standard encoding with one byte per character would require 56 bits. Note also that we can easily decipher the code since the codes have the prefix property. The first bit is 1 which must correspond to a 'D' since this is the only character with a code that starts with a 1. The next bit is 0 and since this is the start of several codes we read one more bit. The only character with a code that start with 01 is 'B' so this must be the next character. The next bit is 0 which does not uniquely identify a character so we read one more bit. The code 00 does not identify a character either, but with one more bit we obtain the code 000 which corresponds to the character 'A'. We can obviously continue in this way and decipher the complete compressed text. ∎

Compression is not quite as simple as it was presented in example 7.3. A program that reads the compressed code must clearly know the codes (7.1) in order to decipher the code. Therefore we must store the codes as well as the compressed text $\boldsymbol{z}$. This means that the text must have a certain length before it is worth compressing it.

7.2.1 Binary trees

The description of Huffman coding in fact 7.2 is not at all precise since it does not state how the codes are determined. The actual algorithm is quite simple, but requires a new concept.

Definition 7.4 (Binary tree)**.** *A* binary tree *T is a finite collection of nodes where one of the nodes is designated as the* root *of the tree, and the remaining nodes are partitioned into two disjoint groups T_0 and T_1 that are also trees. The two trees T_0 and T_1 are called the* subtrees *or* children *of T. Nodes which are not roots of subtrees are called* leaf nodes*. A connection from one node to another is called an* edge *of the tree.*

An example of a binary tree is shown in figure 7.2. The root node which is shown at the top has two subtrees. The subtree to the right also has two subtrees, both of which only contain leaf nodes. The subtree to the left of the root only has one subtree which consists of a single leaf node.

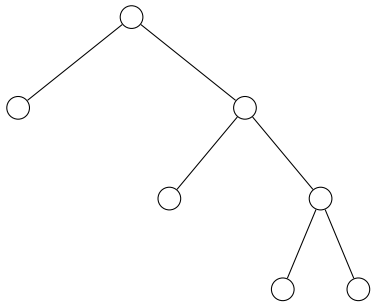


Figure 7.2. An example of a binary tree.

7.2.2 Huffman trees

It turns out that Huffman coding can conveniently be described in terms of a binary tree with some extra information added. These trees are usually referred to as *Huffman trees*.

Definition 7.5. *A Huffman tree is a binary tree that can be associated with an alphabet consisting of symbols* $\{\alpha_i\}_{i=1}^{n}$ *with frequencies* $f(\alpha_i)$ *as follows:*

1. *Each leaf node is associated with exactly one symbol* α_i *in the alphabet, and all symbols are associated with a leaf node.*
2. *Each node has an associated integer weight:*
   (a) *The weight of a leaf node is the frequency of the symbol.*
   (b) *The weight of a node is the sum of the weights of the roots of the node's subtrees.*
3. *All nodes that are not leaf nodes have exactly two children.*
4. *The Huffman code of a symbol is obtained by following edges from the root to the leaf node associated with the symbol. Each edge adds a bit to the code: a 0 if the edge points to the left and a 1 if it points to the right.*

Example 7.6. In figure 7.3 the tree in figure 7.2 has been turned into a Huffman tree. The tree has been constructed from the text CCDACBDC with the alphabet {A,B,C,D} and frequencies $f(A) = 1$, $f(B) = 1$, $f(C) = 4$ and $f(D) = 2$. It is easy to see that the weights have the properties required for a Huffman tree, and by

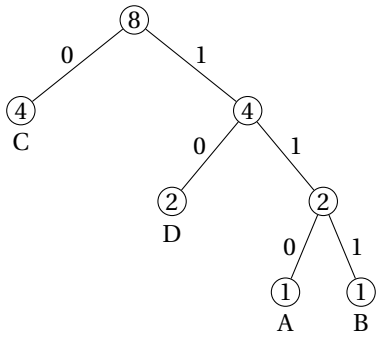


Figure 7.3. A Huffman tree.

following the edges we see that the Huffman codes are given by $c(C) = 0$, $c(D) = 10$, $c(A) = 110$ and $c(B) = 111$. Note in particular that the root of the tree has weight equal to the length of the text. ∎

We will usually omit the labels on the edges since they are easy to remember: An edge that points to the left corresponds to a 0, while an edge that points to the right yields a 1.

7.2.3 The Huffman algorithm

In example 7.6 the Huffman tree was just given to us; the essential question is how the tree can be constructed from a given text. There is a simple algorithm that accomplishes this.

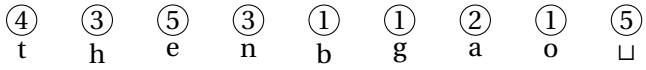
Algorithm 7.7 (Huffman algorithm)**.** *Let the text $\boldsymbol{x}$ with symbols $\{\alpha_i\}_{i=1}^n$ be given, and let the frequency of α_i be $f(\alpha_i)$. The Huffman tree is constructed by performing the following steps:*

1. *Construct a one-node Huffman tree from each of the n symbols α_i and its corresponding weight; this leads to a collection of n one-node trees.*
2. *Repeat until the collection consists of only one tree:*
   (a) *Choose two trees T_0 and T_1 with minimal weights and replace them with a new tree which has T_0 as its left subtree and T_1 as its right subtree.*
3. *The tree remaining after the previous step is a Huffman tree for the given text $\boldsymbol{x}$.*

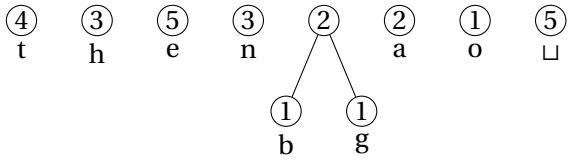
Most of the work in algorithm 7.7 is in step 2, but note that the number of trees is reduced by one each time, so the loop will run at most n times.

The easiest way to get to grips with the algorithm is to try it on a simple example.

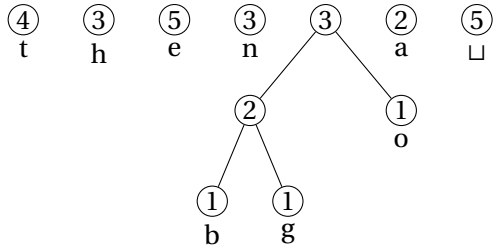
Example 7.8. Let us try out algorithm 7.7 on the text 'then the hen began to eat'. This text consists of 32 characters, including the five spaces. We first determine the frequencies of the different characters by counting. We find the collection of one-node trees



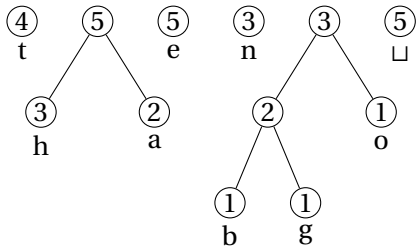
where the last character denotes the space character. Since 'b' and 'g' are two characters with the lowest frequency, we combine them into a tree,



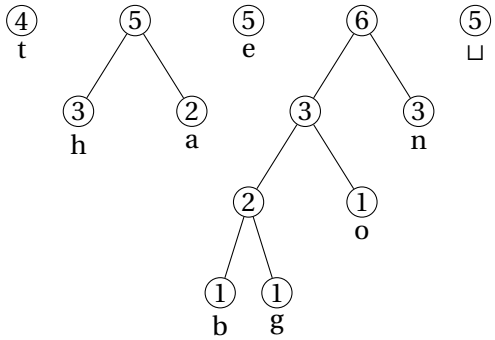
The two trees with the lowest weights are now the character 'o' and the tree we formed in the last step. If we combine these we obtain



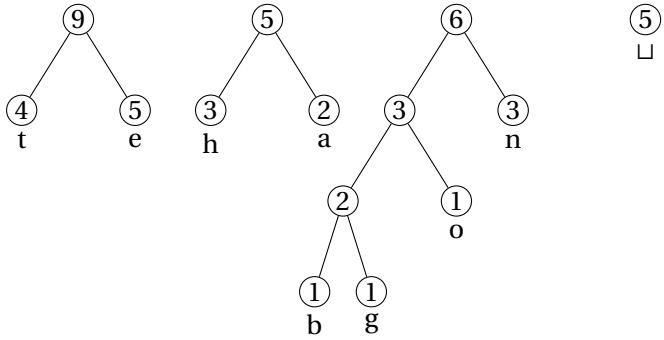
Now we have several choices. We choose to combine 'a' and 'h',



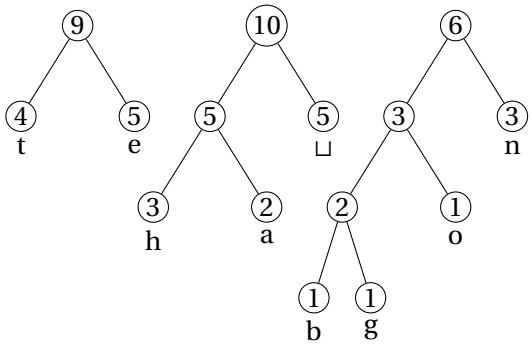
At the next step we combine the two trees with weight 3,



Next we combine the 't' and the 'e',



We now have two trees with weight 5 that must be combined



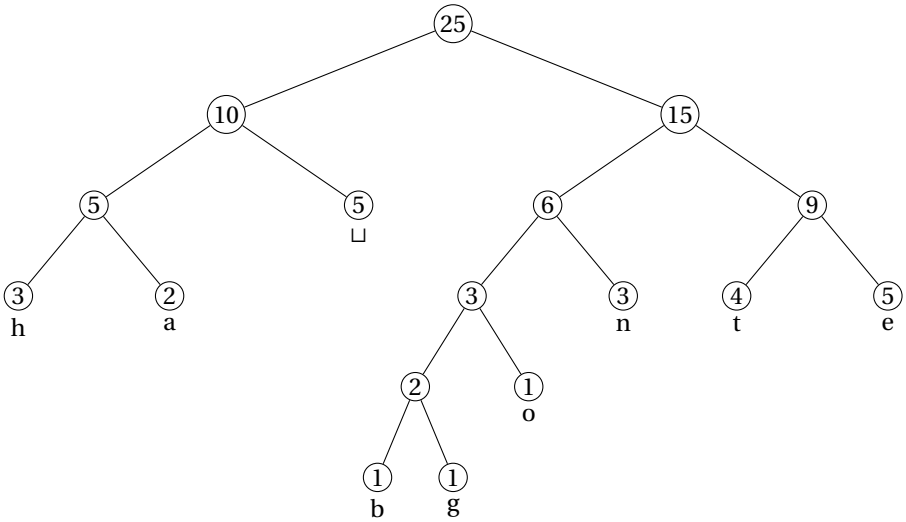
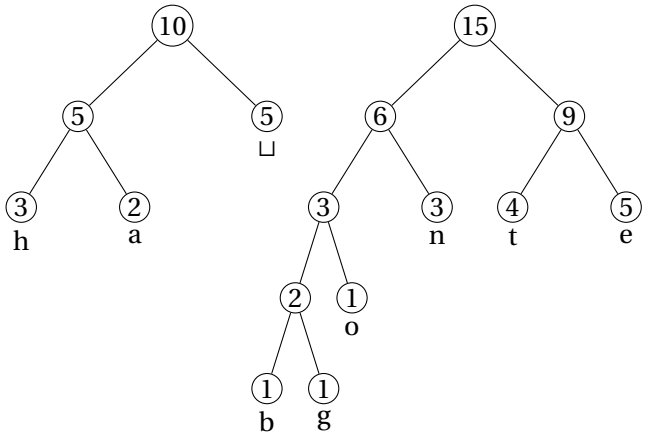


Figure 7.4. The Huffman tree for the text 'then the hen began to eat'.

Again we combine the two trees with the smallest weights,



By combining these two trees we obtain the final Huffman tree in figure 7.4. From this we can read off the Huffman codes as

$$\begin{aligned} c(h) &= 000, & c(b) &= 10000, & c(n) &= 101, \\ c(a) &= 001, & c(g) &= 10001, & c(t) &= 110, \\ c(\sqcup) &= 01, & c(o) &= 1001, & c(e) &= 111. \end{aligned}$$

so we see that the Huffman coding of the text 'then the hen began to eat' is

110 000 111 101 01 110 000 111 01 000 111 101 01 10000
111 10001 001 101 01 110 1001 01 111 001 110

The spaces and the new line have been added to make the code easier to read; on a computer these will not be present.

The original text consists of 25 characters including the spaces. Encoding this with standard eight-bit encodings like ISO Latin or UTF-8 would require 400 bits. Since there are only nine symbols we could use a shorter fixed width encoding for this particular text. This would require five bits per symbol and would reduce the total length to 125 bits. In contrast the Huffman encoding only requires 75 bits. ∎

7.2.4 Properties of Huffman trees

Huffman trees have a number of useful properties, and the first one is the prefix property, see fact 7.2. This is easy to deduce from simple properties of Huffman trees.

Proposition 7.9 (Prefix property)**.** *Huffman coding has the prefix property: No code is a prefix of any other code.*

Proof. Suppose that Huffman coding does not have the prefix property, we will show that this leads to a contradiction. Let the code c_1 be the prefix of another code c_2, and let n_i be the node associated with the symbol with code c_i. Then the node n_1 must be somewhere on the path from the root down to n_2. But then n_2 must be located further from the root than n_1, so n_1 cannot be a leaf node, which contradicts the definition of a Huffman tree (remember that symbols are only associated with leaf nodes). ∎

We emphasise that it is the prefix property that makes it possible to use variable lengths for the codes; without this property we would not be able to decode an encoded text. Just consider the simple case where $c(A) = 01$, $c(B) = 010$ and $c(C) = 1$; which text would the code 0101 correspond to?

In the Huffman algorithm, we start by building trees from the symbols with lowest frequency. These symbols will therefore end up the furthest from the root and end up with the longest codes, as is evident from example 7.8. Likewise, the symbols with the highest frequencies will end up near the root of the tree and therefore receive short codes. This property of Huffman coding can be quantified, but to do this we must introduce a new concept.

Note that any binary tree with the symbols at the leaf nodes gives rise to a coding with the prefix property. A natural question is then which tree gives the coding with the fewest bits?

Theorem 7.10 (Optimality of Huffman coding)**.** *Let $\boldsymbol{x}$ be a given text, let T be any binary tree with the symbols of $\boldsymbol{x}$ as leaf nodes, and let $\ell(T)$ denote the number of bits in the encoding of $\boldsymbol{x}$ in terms of the codes from T. If T^* denotes the Huffman tree corresponding to the text $\boldsymbol{x}$ then*

$$\ell(T^*) \leq \ell(T).$$

Theorem 7.10 says that Huffman coding is optimal, at least among coding schemes based on binary trees. Together with its simplicity, this accounts for the popularity of this compression method.

7.3 Probabilities and information entropy

Huffman coding is the best possible among all coding schemes based on binary trees, but could there be completely different schemes, which do not depend on binary trees, that are better? And if this is the case, what would be the best possible scheme? To answer questions like these, it would be nice to have a way to tell how much information there is in a text.

7.3.1 Probabilities rather than frequencies

Let us first consider more carefully how we should measure the quality of Huffman coding. For a fixed text $\boldsymbol{x}$, our main concern is how many bits we need to encode the text, see the end of example 7.8. If the symbol α_i occurs $f(\alpha_i)$ times and requires $\ell(\alpha_i)$ bits and we have n symbols, the total number of bits is

$$B = \sum_{i=1}^{n} f(\alpha_i)\ell(\alpha_i). \tag{7.3}$$

However, we note that if we multiply all the frequencies by the same constant, the Huffman tree remains the same. It therefore only depends on the relative frequencies of the different symbols, and not the length of the text. In other words, if we consider a new text which is twice as long as the one we used in example 7.8, with each letter occurring twice as many times, the Huffman tree would be the same. This indicates that we should get a good measure of the quality of an encoding if we divide the total number of bits used by the length of

the text. If the length of the text is m this leads to the quantity

$$\bar{B} = \sum_{i=1}^{n} \frac{f(\alpha_i)}{m} \ell(\alpha_i). \tag{7.4}$$

If we consider longer and longer texts of the same type, it is reasonable to believe that the relative frequencies of the symbols would converge to a limit $p(\alpha_i)$ which is usually referred to as the *probability* of the symbol α_i. As always for probabilities we have $\sum_{i=1}^{n} p(\alpha_i) = 1$.

Instead of referring to the frequencies of the different symbols in an alphabet we will from now on refer to the probabilities of the symbols. We can then translate the bits per symbol measure in equation 7.4 to a setting with probabilities.

Observation 7.11 (Bits per symbol)**.** *Let $\mathscr{A} = \{\alpha_1, \ldots, \alpha_n\}$ be an alphabet where the symbol α_i has probability $p(\alpha_i)$ and is encoded with $\ell(\alpha_i)$ bits. Then the average number of bits per symbol in a text encoded with this alphabet is*

$$\bar{b} = \sum_{i=1}^{n} p(\alpha_i)\ell(\alpha_i). \tag{7.5}$$

Note that the Huffman algorithm will work just as well if we use the probabilities as weights rather than the frequencies, as this is just a relative scaling. In fact, the most obvious way to obtain the probabilities is to just divide the frequencies with the number of symbols for a given text. However, it is also possible to use a probability distribution that has been determined by some other means. For example, the probabilities of the different characters in English have been determined for typical texts. Using these probabilities and the corresponding codes will save you the trouble of processing your text and computing the probabilities for a particular text. Remember however that such pre-computed probabilities are not likely to be completely correct for a specific text, particularly if the text is short. And this of course means that your compressed text will not be as short as it would be had you computed the correct probabilities.

In practice, it is quite likely that the probabilities of the different symbols change as we work our way through a file. If the file is long, it probably contains different kinds of information, as in a document with both text and images. It would therefore be useful to update the probabilities at regular intervals. In the case of Huffman coding this would of course also require that we update the Huffman tree and therefore the codes assigned to the different symbols. This

may sound complicated, but is in fact quite straightforward. The key is that the decoding algorithm must compute probabilities in exactly the same way as the compression algorithm and update the Huffman tree at exactly the same position in the text. As long as this requirement is met, there will be no confusion as the compression end decoding algorithms will always use the same codes.

7.3.2 Information entropy

The quantity $\bar{b}$ in observation 7.11 measures the number of bits used per symbol for a given coding. An interesting question is how small we can make this number by choosing a better coding strategy. This is answered by a famous theorem.

Theorem 7.12 (Shannon's theorem)**.** *Let $\mathscr{A} = \{\alpha_1, \ldots, \alpha_n\}$ be an alphabet where the symbol α_i has probability $p(\alpha_i)$. Then the minimal number of bits per symbol in an encoding using this alphabet is given by*

$$H = H(p_1, \ldots, p_n) = -\sum_{i=1}^{n} p(\alpha_i) \log_2 p(\alpha_i).$$

where $\log_2$ denotes the logarithm to base 2. The quantity H is called the information entropy *of the alphabet with the given probabilities.*

Example 7.13. Let us return to example 7.8 and compute the entropy in this particular case. From the frequencies we obtain the probabilities

$$\begin{aligned} c(t) &= 4/25, & c(n) &= 3/25, & c(a) &= 2/25, \\ c(h) &= 3/25, & c(b) &= 1/25, & c(o) &= 1/25, \\ c(e) &= 1/5, & c(g) &= 1/25, & c(\sqcup) &= 1/5. \end{aligned}$$

We can then compute the entropy to be $H \approx 2.93$. If we had a compression algorithm that could compress the text down to this number of bits per symbol, we could represent our 25-symbol text with 74 bits. This is only one bit less than what we obtained in example 7.8, so Huffman coding is very close to the best we can do for this particular text. ∎

Note that the entropy can be written as

$$H = \sum_{i=1}^{n} p(\alpha_i) \log_2\big(1/p(\alpha_i)\big).$$

If we compare this expression with equation (7.5) we see that a compression strategy would reach the compression rate promised by the entropy if the length

of the code for the symbol α_i was $\log_2\big(1/p(\alpha_i)\big)$. But we know that this is just the number of bits in the number $1/p(\alpha_i)$. This therefore indicates that an optimal compression scheme would represent α_i by the number $1/p(\alpha_i)$. Huffman coding necessarily uses an integer number of bits for each code, and therefore only has a chance of reaching entropy performance when $1/p(\alpha_i)$ is a power of 2 for all the symbols. In fact Huffman coding does reach entropy performance in this situation, see exercise 3.

7.4 Arithmetic coding

When the probabilities of the symbols are far from being fractions with powers of 2 in their denominators, the performance of Huffman coding does not come close to entropy performance. This typically happens in situations with few symbols as is illustrated by the following example.

Example 7.14. Suppose that we have a two-symbol alphabet $\mathscr{A} = \{0,1\}$ with the probabilities $p(0) = 0.9$ and $p(1) = 0.1$. Huffman coding will then just use the obvious codes $c(0) = 0$ and $c(1) = 1$, so the average number of bits per symbol is 1, i.e., there will be no compression at all. If we compute the entropy we obtain

$$H = -0.9\log_2 0.9 - 0.1\log_2 0.1 \approx 0.47.$$

So while Huffman coding gives no compression, there may be coding methods that will reduce the file size to less than half the original size. ∎

7.4.1 Arithmetic coding basics

Arithmetic coding is a coding strategy that is capable of compressing files to a size close to the entropy limit. It uses a different strategy than Huffman coding and does not need an integer number of bits per symbol and therefore performs well in situations where Huffman coding struggles. The basic idea of arithmetic coding is quite simple.

Idea 7.15 (Basic idea of arithmetic coding)**.** *Arithmetic coding associates sequences of symbols with different subintervals of* $[0,1)$*. The width of a subinterval is proportional to the probability of the corresponding sequence of symbols, and the* arithmetic code *of a sequence of symbols is a floating-point number in the corresponding interval.*

To illustrate some of the details of arithmetic coding, it is easiest to consider an example.

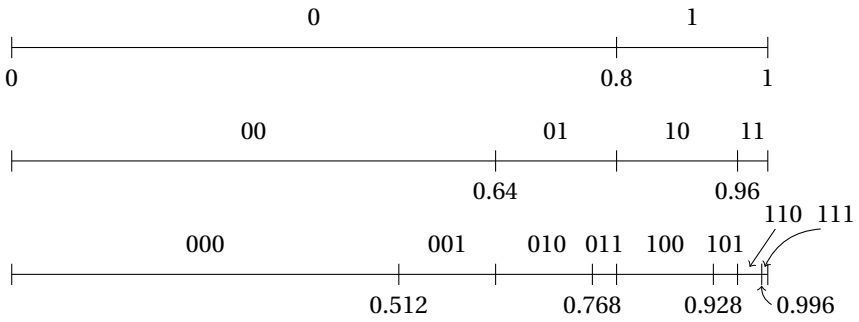


Figure 7.5. The basic principle of arithmetic coding applied to the text in example 7.16.

Example 7.16 (Determining an arithmetic code)**.** We consider the two-symbol text '00100'. As for Huffman coding we first need to determine the probabilities of the two symbols which we find to be $p(0) = 0.8$ and $p(1) = 0.2$. The idea is to allocate different parts of the interval $[0,1)$ to the different symbols, and let the length of the subinterval be proportional to the probability of the symbol. In our case we allocate the interval $[0,0.8)$ to '0' and the interval $[0.8,1)$ to '1'. Since our text starts with '0', we know that the floating-point number which is going to represent our text must lie in the interval $[0,0.8)$, see the first line in figure 7.5.

We then split the two subintervals according to the two probabilities again. If the final floating point number ends up in the interval $[0,0.64)$, the text starts with '00', if it lies in $[0.64,0.8)$, the text starts with '01', if it lies in $[0.8,0.96)$, the text starts with '10', and if the number ends up in $[0.96,1)$ the text starts with '11'. This is illustrated in the second line of figure 7.5. Our text starts with '00', so the arithmetic code we are seeking must lie in the interval $[0,0.64)$.

At the next level we split each of the four sub-intervals in two again, as shown in the third line in figure 7.5. Since the third symbol in our text is '1', the arithmetic code must lie in the interval $[0.512,0.64)$. We next split this interval in the two subintervals $[0.512,0.6144)$ and $[0.6144,0.64)$. Since the fourth symbol is '0', we select the first interval. This interval is then split into $[0.512,0.59392)$ and $[0.59392,0.6144)$. The final symbol of our text is '0', so the arithmetic code must lie in the interval $[0.512,0.59392)$.

We know that the arithmetic code of our text must lie in the half-open interval $[0.512,0.59392)$, but it does not matter which of the numbers in the interval we use. The code is going to be handled by a computer so it must be represented in the binary numeral system, with a finite number of bits. We know that any number of this kind must be on the form $i/2^k$ where k is a positive integer and i is an integer in the range $0 \le i < 2^k$. Such numbers are called *dyadic*

numbers. We obviously want the code to be as short as possible, so we are looking for the dyadic number with the smallest denominator that lies in the interval $[0.512, 0.59392)$. In our simple example it is easy to see that this number is $9/16 = 0.5625$. In binary this number is 0.1001_2, so the arithmetic code for the text '00100' is 1001. ∎

Example 7.16 shows how an arithmetic code is computed. We have done all the computations in decimal arithmetic, but in a program one would usually use binary arithmetic.

It is not sufficient to be able to encode a text; we must be able to decode as well. This is in fact quite simple. We split the interval $[0,1]$ into the smaller pieces, just like we did during the encoding. By checking which interval contains our code, we can extract the correct symbol at each stage.

7.4.2 An algorithm for arithmetic coding

Let us now see how the description in example 7.16 can be generalised to a systematic algorithm in a situation with n different symbols. An important tool in the algorithm is a function that maps the interval $[0,1]$ to a general interval $[a,b]$.

Observation 7.17. *Let $[a,b]$ be a given interval with $a < b$. The function*

$$g(z) = a + z(b-a)$$

will map any number z in $[0,1]$ to a number in the interval $[a,b]$. In particular the endpoints are mapped to the endpoints and the midpoint to the midpoint,

$$g(0) = a, \quad g(1/2) = \frac{a+b}{2}, \quad g(1) = b.$$

We are now ready to study the details of the arithmetic coding algorithm. As before we have a text $\boldsymbol{x} = \{x_1, \ldots, x_m\}$ with symbols taken from an alphabet $\mathscr{A} = \{\alpha_1, \ldots, \alpha_n\}$, with $p(\alpha_i)$ being the probability of encountering α_i at any given position in $\boldsymbol{x}$. It is much easier to formulate arithmetic coding if we introduce one more concept.

Definition 7.18 (Cumulative probability distribution)**.** *Let $\mathscr{A} = \{\alpha_1, \ldots, \alpha_n\}$ be an alphabet where the probability of α_i is $p(\alpha_i)$. The* cumulative probability distribution *F is defined as*

$$F(\alpha_j) = \sum_{i=1}^{j} p(\alpha_i), \quad \text{for } j = 1, 2, \ldots, n.$$

The related function L is defined by $L(\alpha_1) = 0$ and

$$L(\alpha_j) = F(\alpha_j) - p(\alpha_j) = F(\alpha_{j-1}), \quad \textit{for } j = 2, 3, \ldots, n.$$

It is important to remember that the functions F, L and p are defined for the symbols in the alphabet $\mathscr{A}$. This means that $F(x)$ only makes sense if $x = \alpha_i$ for some i in the range $1 \le i \le n$.

The basic idea of arithmetic coding is to split the interval $[0,1)$ into the n subintervals

$$\big[0, F(\alpha_1)\big), \quad \big[F(\alpha_1), F(\alpha_2)\big), \ \ldots, \ \big[F(\alpha_{n-2}), F(\alpha_{n-1})\big), \quad \big[F(\alpha_{n-1}), 1\big) \tag{7.6}$$

so that the width of the subinterval $\big[F(\alpha_{i-1}), F(\alpha_i)\big)$ is $F(\alpha_i) - F(\alpha_{i-1}) = p(\alpha_i)$. If the first symbol is $x_1 = \alpha_i$, the arithmetic code must lie in the interval $[a_1, b_1)$ where

$$\begin{aligned} a_1 &= p(\alpha_1) + p(\alpha_2) + \cdots + p(\alpha_{i-1}) = F(\alpha_{i-1}) = L(\alpha_i) = L(x_1), \\ b_1 &= a_1 + p(\alpha_i) = F(\alpha_i) = F(x_1). \end{aligned}$$

The next symbol in the text is x_2. If this were the first symbol of the text, the desired subinterval would be $\big[L(x_2), F(x_2)\big)$. Since it is the second symbol we must map the whole interval $[0,1)$ to the interval $[a_1, b_1]$ and pick out the part that corresponds to $\big[L(x_2), F(x_2)\big)$. The mapping from $[0,1)$ to $[a_1, b_1)$ is given by $g_2(z) = a_1 + z(b_1 - a_1) = a_1 + zp(x_1)$, see observation 7.17, so our new interval is

$$[a_2, b_2) = \Big[g_2\big(L(x_2)\big), g_2\big(F(x_2)\big)\Big) = \big[a_1 + L(x_2)p(x_1), a_1 + F(x_2)p(x_1)\big).$$

The third symbol x_3 would be associated with the interval $\big[L(x_3), F(x_3)\big)$ if it were the first symbol. To find the correct subinterval, we map $[0,1)$ to $[a_2, b_2)$ with the mapping $g_3(z) = a_2 + z(b_2 - a_2)$ and pick out the correct subinterval as

$$[a_3, b_3) = \Big[g_3\big(L(x_3)\big), g_3\big(F(x_3)\big)\Big).$$

This process is then continued until all the symbols in the text have been processed.

With this background we can formulate a precise algorithm for arithmetic coding of a text of length m with n distinct symbols.

Algorithm 7.19 (Arithmetic coding)**.** *Let the text* $\boldsymbol{x} = \{x_1, \ldots, x_m\}$ *be given, with the symbols being taken from an alphabet* $\mathscr{A} = \{\alpha_1, \ldots, \alpha_n\}$*, with probabilities* $p(\alpha_i)$ *for* $i = 1, \ldots, n$*. Generate a sequence of* m *subintervals of* $[0,1)$*:*

1. *Set* $[a_0, b_0] = \big[0,1\big)$*.*
2. *For* $k = 1, \ldots, m$
   (a) *Define the linear function* $g_k(z) = a_{k-1} + z(b_{k-1} - a_{k-1})$*.*
   (b) *Set* $[a_k, b_k] = \Big[g_k\big(L(x_k)\big), g_k\big(F(x_k)\big)\Big)$*.*

The arithmetic code of the text $\boldsymbol{x}$ *is the midpoint* $C(\boldsymbol{x})$ *of the interval* $[a_m, b_m)$*, i.e., the number*

$$\frac{a_m + b_m}{2},$$

truncated to

$$\Big\lceil -\log_2\big(p(x_1)p(x_2)\cdots p(x_m)\big)\Big\rceil + 1$$

binary digits. Here $\lceil w \rceil$ *denotes the smallest integer that is larger than or equal to* w*.*

A program for arithmetic coding needs to output a bit more information than just the arithmetic code itself. For the decoding we also need to know exactly which probabilities were used and the ordering of the symbols (this influences the cumulative probability function). In addition we need to know when to stop decoding. A common way to provide this information is to store the length of the text. Alternatively, there must be a unique symbol that terminates the text so when we encounter this symbol during decoding we know that we are finished.

Let us consider another example of arithmetic coding in a situation with a three-symbol alphabet.

Example 7.20. Suppose we have the text $\boldsymbol{x} = \{ACBBCAABAA\}$ and we want to encode it with arithmetic coding. We first note that the probabilities are given by

$$p(A) = 0.5, \quad p(B) = 0.3, \quad p(C) = 0.2,$$

so the cumulative probabilities are $F(A) = 0.5$, $F(B) = 0.8$ and $F(C) = 1.0$. This means that the interval $[0,1)$ is split into the three subintervals

$$[0, 0.5), \quad [0.5, 0.8), \quad [0.8, 1).$$

The first symbol is A, so the first subinterval is $[a_1, b_1) = [0, 0.5)$. The second symbol is C so we must find the part of $[a_1, b_1)$ that corresponds to C. The mapping from $[0,1)$ to $[0, 0.5)$ is given by $g_2(z) = 0.5z$ so $[0.8, 1]$ is mapped to

$$[a_2, b_2) = \big[g_2(0.8), g_2(1)\big) = [0.4, 0.5).$$

The third symbol is B which corresponds to the interval $[0.5, 0.8)$. We map $[0,1)$ to the interval $[a_2, b_2)$ with the function $g_3(z) = a_2 + z(b_2 - a_2) = 0.4 + 0.1z$ so $[0.5, 0.8)$ is mapped to

$$[a_3, b_3) = \big[g_3(0.5), g_3(0.8)\big) = [0.45, 0.48).$$

Let us now write down the rest of the computations more schematically in a table,

$$\begin{aligned}
&g_4(z) = 0.45 + 0.03z, \quad x_4 = B, \quad [a_4, b_4) = \big[g_4(0.5), g_4(0.8)\big) = [0.465, 0.474),\\
&g_5(z) = 0.465 + 0.009z, \quad x_5 = C, \quad [a_5, b_5) = \big[g_5(0.8), g_5(1)\big) = [0.4722, 0.474),\\
&g_6(z) = 0.4722 + 0.0018z, \quad x_6 = A, \quad [a_6, b_6) = \big[g_6(0), g_6(0.5)\big) = [0.4722, 0.4731),\\
&g_7(z) = 0.4722 + 0.0009z, \quad x_7 = A, \quad [a_7, b_7) = \big[g_7(0), g_7(0.5)\big) = [0.4722, 0.47265),\\
&g_8(z) = 0.4722 + 0.00045z, \quad x_8 = B, \quad [a_8, b_8) = \big[g_8(0.5), g_8(0.8)\big) = [0.472425, 0.47256),\\
&g_9(z) = 0.472425 + 0.000135z, \quad x_9 = A,\\
&\qquad\qquad [a_9, b_9) = \big[g_9(0), g_9(0.5)\big) = [0.472425, 0.4724925),\\
&g_{10}(z) = 0.472425 + 0.0000675z, \quad x_{10} = A,\\
&\qquad\qquad [a_{10}, b_{10}) = \big[g_{10}(0), g_{10}(0.5)\big) = [0.472425, 0.47245875).
\end{aligned}$$

The midpoint M of this final interval is

$$M = 0.472441875 = 0.011110001111100011111_2,$$

and the arithmetic code is M rounded to

$$\Big\lceil -\log_2\big(p(A)^5 p(B)^3 p(C)^2\big)\Big\rceil + 1 = 16$$

bits. The arithmetic code is therefore the number

$$C(\boldsymbol{x}) = 0.0111100011110001_2 = 0.472427,$$

but we just store the 16 bits 0111100011110001. In this example the arithmetic code therefore uses 1.6 bits per symbol. In comparison the entropy is 1.49 bits per symbol. ∎

7.4.3 Properties of arithmetic coding

In example 7.16 we chose the arithmetic code to be the dyadic number with the smallest denominator within the interval $[a_m, b_m)$. In algorithm 7.19 we have chosen a number that is a bit easier to determine, but still we need to prove that the truncated number lies in the interval $[a_m, b_m)$. This is necessary because when we throw away some of the digits in the representation of the midpoint, the result may end up outside the interval $[a_m, b_m]$. We combine this with an important observation on the length of the interval.

Theorem 7.21. *The width of the interval $[a_m, b_m)$ is*

$$b_m - a_m = p(x_1)p(x_2)\cdots p(x_m) \tag{7.7}$$

and the arithmetic code $C(\boldsymbol{x})$ lies inside this interval.

Proof. The proof of equation(7.7) is by induction on m. For $m = 1$, the length is simply $b_1 - a_1 = F(x_1) - L(x_1) = p(x_1)$, which is clear from the last equation in Definition 7.18. Suppose next that

$$b_{k-1} - a_{k-1} = p(x_1)\cdots p(x_{k-1});$$

we need to show that $b_k - a_k = p(x_1)\cdots p(x_k)$. This follows from step 2 of algorithm 7.19,

$$\begin{aligned} b_k - a_k &= g_k\big(F(x_k)\big) - g_k\big(L(x_k)\big) \\ &= \big(F(x_k) - L(x_k)\big)(b_{k-1} - a_{k-1}) \\ &= p(x_k)p(x_1)\cdots p(x_{k-1}). \end{aligned}$$

In particular we have $b_m - a_m = p(x_1)\cdots p(x_m)$.

Our next task is to show that the arithmetic code $C(\boldsymbol{x})$ lies in $[a_m, b_m)$. Define the number μ by the relation

$$\frac{1}{2^\mu} = b_m - a_m = p(x_1)\cdots p(x_m) \quad \text{or} \quad \mu = -\log_2\big(p(x_1)\cdots p(x_m)\big).$$

In general μ will not be an integer, so we introduce a new number λ which is the smallest integer that is greater than or equal to μ,

$$\lambda = \lceil \mu \rceil = \Big\lceil -\log_2\big(p(x_1)\cdots p(x_m)\big)\Big\rceil.$$

This means that $1/2^\lambda$ is smaller than or equal to $b_m - a_m$ since $\lambda \geq \mu$. Consider the collection of dyadic numbers D_λ on the form $j/2^\lambda$ where j is an integer in

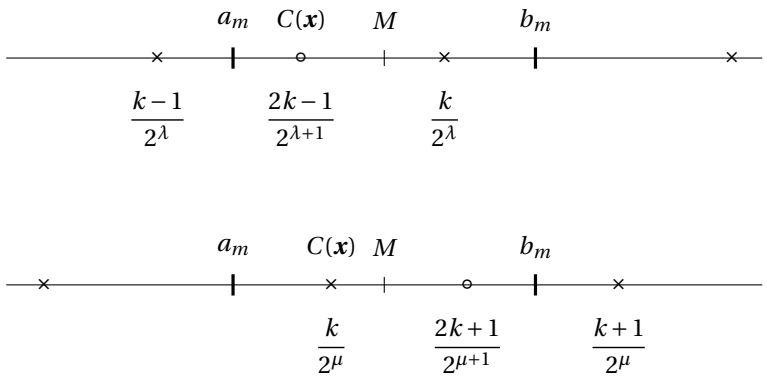


Figure 7.6. The two situations that can occur when determining the number of bits in the arithmetic code.

the range $0 \leq j < 2^\lambda$. At least one of them, say $k/2^\lambda$, must lie in the interval $[a_m, b_m)$ since the distance between neighbouring numbers in D_λ is $1/2^\lambda$ which is at most equal to $b_m - a_m$. Denote the midpoint of $[a_m, b_m)$ by M. There are two situations to consider which are illustrated in figure 7.6.

In the first situation shown in the top part of the figure, the number $k/2^\lambda$ is larger than M and there is no number in D_λ in the interval $[a_m, M]$. If we form the approximation $\tilde{M}$ to M by only keeping the first λ binary digits, we obtain the number $(k-1)/2^\lambda$ in D_λ that is immediately to the left of $k/2^\lambda$. This number may be smaller than a_m, as shown in the figure. To make sure that the arithmetic code ends up in $[a_m, b_m)$ we therefore use one more binary digit and set $C(\boldsymbol{x}) = (2k-1)/2^{\lambda+1}$, which corresponds to keeping the first $\lambda+1$ binary digits in M.

In the second situation there is a number from D_λ in $[a_m, M]$ (this was the case in example 7.16). If we now keep the first λ digits in M we would get $C(\boldsymbol{x}) = k/2^\lambda$. In this case algorithm 7.19 therefore gives an arithmetic code with one more bit than necessary. In practice the arithmetic code will usually be at least thousands of bits long, so an extra bit does not matter much. ∎

Now that we know how to compute the arithmetic code, it is interesting to see how the number of bits per symbol compares with the entropy. The number of bits is given by

$$\left\lceil -\log_2\bigl(p(x_1)p(x_2)\cdots p(x_m)\bigr)\right\rceil + 1.$$

Recall that each x_i is one of the n symbols α_i from the alphabet so by properties of logarithms we have

$$\log_2\big(p(x_1)p(x_2)\cdots p(x_m)\big) = \sum_{i=1}^{n} f(\alpha_i)\log_2 p(\alpha_i)$$

where $f(\alpha_i)$ is the number of times that α_i occurs in $\boldsymbol{x}$. As m becomes large we know that $f(\alpha_i)/m$ approaches $p(\alpha_i)$. For large m we therefore have that the number of bits per symbol approaches

$$\begin{aligned}\frac{1}{m}\Big\lceil -\log_2\big(p(x_1)p(x_2)\cdots p(x_m)\big)\Big\rceil + \frac{1}{m} &\leq -\frac{1}{m}\log_2\big(p(x_1)p(x_2)\cdots p(x_m)\big) + \frac{2}{m}\\ &= -\frac{1}{m}\sum_{i=1}^{n} f(\alpha_i)\log_2 p(\alpha_i) + \frac{2}{m}\\ &\approx -\sum_{i=1}^{n} p(\alpha_i)\log_2 p(\alpha_i)\\ &= H(p_1,\ldots,p_n).\end{aligned}$$

In other words, arithmetic coding gives compression rates close to the best possible for long texts.

Corollary 7.22. *For long texts the number of bits per symbol required by the arithmetic coding algorithm approaches the minimum given by the entropy, provided the probability distribution of the symbols is correct.*

7.4.4 A decoding algorithm

We commented briefly on decoding at the end of section 7.4.1. In this section we will give a detailed decoding algorithm similar to algorithm 7.19.

We will need the linear function that maps an interval $[a,b]$ to the interval $[0,1]$, i.e., the inverse of the function in observation 7.17.

Observation 7.23. *Let $[a,b]$ be a given interval with $a<b$. The function*

$$h(y) = \frac{y-a}{b-a}$$

will map any number y in $[a,b]$ to the interval $[0,1]$. In particular the endpoints are mapped to the endpoints and the midpoint to the midpoint,

$$h(a) = 0, \quad h\big((a+b)/2\big) = 1/2, \quad h(b) = 1.$$

Linear functions like h in observation 7.23 play a similar role in decoding as the g_ks in algorithm 7.19; they help us avoid working with very small intervals. The decoding algorithm assumes that the number of symbols in the text is known and decodes the arithmetic code symbol by symbol. It stops when the correct number of symbols have been found.

Algorithm 7.24. *Let $C(\boldsymbol{x})$ be a given arithmetic code of an unknown text $\boldsymbol{x}$ of length m, based on an alphabet $\mathscr{A} = \{\alpha_1, \ldots, \alpha_n\}$ with known probabilities $p(\alpha_i)$ for $i = 1, \ldots, n$. The following algorithm determines the symbols of the text $\boldsymbol{x} = \{x_1, \ldots, x_m\}$ from the arithmetic code $C(\boldsymbol{x})$:*

1. *Set $z_1 = C(\boldsymbol{x})$.*
2. *For $k = 1, \ldots, m$*
   (a) *Find the integer i such that $L(\alpha_i) \leq z_k < F(\alpha_i)$ and set*
   $$[a_k, b_k) = \big[L(\alpha_i), F(\alpha_i)\big).$$
   (b) *Output $x_k = \alpha_i$.*
   (c) *Determine the linear function $h_k(y) = (y - a_k)/(b_k - a_k)$.*
   (d) *Set $z_{k+1} = h_k(z_k)$.*

The algorithm starts by determining which of the n intervals

$$\big[0, F(\alpha_1)\big), \quad \big[F(\alpha_1), F(\alpha_2)\big), \ldots, \big[F(\alpha_{n-2})\big), F(\alpha_{n-1})\big), \quad \big[F(\alpha_{n-1}), 1\big)$$

it is that contains the arithmetic code $z_1 = C(\boldsymbol{x})$. This requires a search among the cumulative probabilities. When the index i of the interval is known, we know that $x_1 = \alpha_i$. The next step is to decide which subinterval of $[a_1, b_1) = \big[L(\alpha_i), F(\alpha_i)\big)$ that contains the arithmetic code. If we stretch this interval out to $[0,1)$ with the function h_k, we can identify the next symbol just as we did with the first one. Let us see how this works by decoding the arithmetic code that we computed in example 7.16.

Example 7.25 (Decoding of an arithmetic code)**.** Suppose we are given the arithmetic code 1001 from example 7.16 together with the probabilities $p(0) = 0.8$ and $p(1) = 0.2$. We also assume that the length of the code is known, the probabilities, and how the probabilities were mapped into the interval $[0,1]$; this is the typical output of a program for arithmetic coding. Since we are going to do this

manually, we start by converting the number to decimal; if we were to program arithmetic coding we would do everything in binary arithmetic.

The arithmetic code 1001 corresponds to the binary number 0.1001_2 which is the decimal number $z_1 = 0.5625$. Since this number lies in the interval $[0, 0.8)$ we know that the first symbol is $x_1 = 0$. We now map the interval $[0, 0.8)$ and the code back to the interval $[0, 1)$ with the function

$$h_1(y) = y/0.8.$$

We find that the code becomes

$$z_2 = h_1(z_1) = z_1/0.8 = 0.703125$$

relative to the new interval. This number lies in the interval $[0, 0.8)$ so the second symbol is $x_2 = 0$. Once again we map the current interval and arithmetic code back to $[0, 1)$ with the function h_2 and obtain

$$z_3 = h_2(z_2) = z_2/0.8 = 0.87890625.$$

This number lies in the interval $[0.8, 1)$, so our third symbol must be a $x_3 = 1$. At the next step we must map the interval $[0.8, 1)$ to $[0, 1)$. From observation 7.23 we see that this is done by the function $h_3(y) = (y - 0.8)/0.2$. This means that the code is mapped to

$$z_4 = h_3(z_3) = (z_3 - 0.8)/0.2 = 0.39453125.$$

This brings us back to the interval $[0, 0.8)$, so the fourth symbol is $x_4 = 0$. This time we map back to $[0, 1)$ with the function $h_4(y) = y/0.8$ and obtain

$$z_5 = h_4(z_4) = 0.39453125/0.8 = 0.493164.$$

Since we remain in the interval $[0, 0.8)$ the fifth and last symbol is $x_5 = 0$, so the original text was '00100'. ∎

7.4.5 Arithmetic coding in practice

Algorithms 7.19 and 7.24 are quite simple and appear to be easy to program. However, there is one challenge that we have not addressed. The typical symbol sequences that we may want to compress are very long, with perhaps millions or even billions of symbols. In the coding process the intervals that contain the arithmetic code become smaller for each symbol that is processed which means that the ends of the intervals must be represented with extremely high precision. A program for arithmetic coding must therefore be able to handle arbitrary precision arithmetic in an efficient way. For a time this prevented the method from

being used, but there are now good algorithms for handling this. The basic idea is to organise the computations of the endpoints of the intervals in such a way that early digits are not influenced by later ones. It is then sufficient to only work with a limited number of digits at a time (for example 32 or 64 binary digits). The details of how this is done is rather technical though.

Since the compression rate of arithmetic coding is close to the optimal rate predicted by the entropy, one would think that it is often used in practice. However, arithmetic coding is protected by many patents which means that you have to be careful with the legal details if you use the method in commercial software. For this reason, many prefer to use other compression algorithms without such restrictions, even though these methods may not perform quite so well.

In long texts the frequency of the symbols may vary within the text. To compensate for this it is common to let the probabilities vary. This does not cause problems as long as the coding and decoding algorithms compute and adjust the probabilities in exactly the same way.

7.5 Lempel-Ziv-Welch algorithm

The Lempel-Ziv-Welch algorithm is named after the three inventors and is usually referred to as the LZW algorithm. The original idea is due to Lempel and Ziv and is used in the LZ77 and LZ78 algorithms.

LZ78 constructs a *code book* during compression, with entries for combinations of several symbols as well as for individual symbols. If, say, the ten next symbols already have an entry in the code book as individual symbols, a new entry is added to represent the combination consisting of these next ten symbols. If this same combination of ten symbols appears later in the text, it can be represented by its code.

The LZW algorithm is based on the same idea as LZ78, with small changes to improve compression further.

LZ77 does not store a list of codes for previously encountered symbol combinations. Instead it searches previous symbols for matches with the sequence of symbols that are presently being encoded. If the next ten symbols match a sequence 90 symbols earlier in the symbol sequence, a code for the pair of numbers $(90, 10)$ will be used to represent these ten symbols. This can be thought of as a type of run-length coding.

7.6 Lossless compression programs

Lossless compression has become an important ingredient in many different contexts, often coupled with a lossy compression strategy. We will discuss this

in more detail in the context of digital sound and images in later chapters, but want to mention two general-purpose programs for lossless compression here.

7.6.1 Compress

The program `compress` is a much used compression program on UNIX platforms which first appeared in 1984. It uses the LZW-algorithm. After the program was published it turned out that part of the algorithm was covered by a patent.

7.6.2 gzip

To avoid the patents on `compress`, the alternative program `gzip` appeared in 1992. This program is based on the LZ77 algorithm, but uses Huffman coding to encode the pairs of numbers. Although `gzip` was originally developed for the Unix platform, it has now been ported to most operating systems, see `www.gzip.org`.

Exercises

7.1 In this exercise we are going to use Huffman coding to encode the text 'There are many people in the world', including the spaces.

- **a)** Compute the frequencies of the different symbols used in the text.
- **b)** Use algorithm 7.7 to determine the Huffman tree for the symbols.
- **c)** Determine the Huffman coding of the complete text. How does the result compare with the entropy?

7.2 We can generalise Huffman coding to numeral systems other than the binary system.

- **a)** Suppose we have a computer that works in the ternary (base-3) numeral system; describe a variant of Huffman coding for such machines.
- **b)** Generalise the Huffman algorithm so that it produces codes in the base-n numeral system.

7.3 In this exercise we are going to do Huffman coding for the text given by $\boldsymbol{x} = \{ABACABCA\}$.

- **a)** Compute the frequencies of the symbols, perform the Huffman algorithm and determine the Huffman coding. Compare the result with the entropy.
- **b)** Change the frequencies to $f(A) = 1$, $f(B) = 1$, $f(C) = 2$ and compare the Huffman tree with the one from (a).

7.4 Recall from section 4.3.1 in chapter 4 that ASCII encodes the 128 most common symbols used in English with seven-bit codes. If we denote the alphabet by $\mathscr{A} = \{\alpha_i\}_{i=1}^{128}$, the codes are

$$c(\alpha_1) = 0000000, \quad c(\alpha_2) = 0000001, \quad c(\alpha_3) = 0000010, \quad \ldots$$
$$c(\alpha_{127}) = 1111110, \quad c(\alpha_{128}) = 1111111.$$

Explain how these codes can be associated with a certain Huffman tree. What are the frequencies used in the construction of the Huffman tree?

7.5 In this exercise we use the two-symbol alphabet $\mathscr{A} = \{A, B\}$.

a) Compute the frequencies $f(A)$ and $f(B)$ in the text

$$\boldsymbol{x} = \{AAAAAAAABAA\}$$

and the probabilities $p(A)$ and $p(B)$.

b) We want to use arithmetic coding to compress the sequence in (a); how many bits do we need in the arithmetic code?

c) Compute the arithmetic code of the sequence in (a).

7.6 The four-symbol alphabet $\mathscr{A} = \{A, B, C, D\}$ is used throughout this exercise. The probabilities are given by $p(A) = p(B) = p(C) = p(D) = 0.25$.

a) Compute the information entropy for this alphabet with the given probabilities.

b) Construct the Huffman tree for the alphabet. How many bits per symbol is required if you use Huffman coding with this alphabet?

c) Suppose now that we have a text $\boldsymbol{x} = \{x_1, \ldots, x_m\}$ consisting of m symbols taken from the alphabet $\mathscr{A}$. We assume that the frequencies of the symbols correspond with the probabilities of the symbols in the alphabet.

How many bits does arithmetic coding require for this sequence and how many bits per symbol does this correspond to?

d) The Huffman tree you obtained in (b) is not unique. Here we will fix a tree so that the Huffman codes are

$$c(A) = 00, \quad c(B) = 01, \quad c(C) = 10, \quad c(D) = 11.$$

Compute the Huffman coding of the sequence 'ACDBAC'.

e) Compute the arithmetic code of the sequence in (d). What is the similarity with the result obtained with Huffman coding in (d)?

7.7 The three-symbol alphabet $\mathscr{A} = \{A, B, C\}$ with probabilities $p(A) = 0.1$, $p(B) = 0.6$ and $p(C) = 0.3$ is given. A text $\boldsymbol{x}$ of length 10 has been encoded by arithmetic coding and the code is 1001101. What is the text $\boldsymbol{x}$?

7.8 We have the two-symbol alphabet $\mathscr{A} = \{A, B\}$ with $p(A) = 0.99$ and $p(B) = 0.01$. Find the arithmetic code of the text

$$\overbrace{AAA\cdots AAA}^{\text{99 times}}B.$$

7.9 The two linear functions in observations 7.17 and 7.23 are special cases of a more general construction. Suppose we have two nonempty intervals $[a, b]$ and $[c, d]$, find the linear function which maps $[a, b]$ to $[c, d]$.

Check that your solution is correct by comparing with observations 7.17 and 7.23.