# 2
# Huffman Coding

## Prelude

Huffman coding is a popular method for compressing data with variable-length codes. Given a set of data symbols (an alphabet) and their frequencies of occurrence (or, equivalently, their probabilities), the method constructs a set of variable-length codewords with the shortest average length and assigns them to the symbols. Huffman coding serves as the basis for several applications implemented on popular platforms. Some programs use just the Huffman method, while others use it as one step in a multistep compression process. The Huffman method [Huffman 52] is somewhat similar to the Shannon–Fano method, proposed independently by Claude Shannon and Robert Fano in the late 1940s ([Shannon 48] and [Fano 49]). It generally produces better codes, and like the Shannon–Fano method, it produces the best variable-length codes when the probabilities of the symbols are negative powers of 2. The main difference between the two methods is that Shannon–Fano constructs its codes from top to bottom (and the bits of each codeword are constructed from left to right), while Huffman constructs a code tree from the bottom up (and the bits of each codeword are constructed from right to left).

Since its inception in 1952 by D. Huffman, the method has been the subject of intensive research in data compression. The long discussion in [Gilbert and Moore 59] proves that the Huffman code is a minimum-length code in the sense that no other encoding has a shorter average length. A much shorter proof of the same fact was discovered by Huffman himself [Motil 07]. An algebraic approach to constructing the Huffman code is introduced in [Karp 61]. In [Gallager 78], Robert Gallager shows that the redundancy of Huffman coding is at most $p_1 + 0.086$ where $p_1$ is the probability of the most-common symbol in the alphabet. The redundancy is the difference between the average Huffman codeword length and the entropy. Given a large alphabet, such as the set of letters, digits and punctuation marks used by a natural language, the largest symbol probability is typically around 15–20%, bringing the value of the quantity $p_1 + 0.086$ to around 0.1. This means that Huffman codes are at most 0.1 bit longer (per symbol) than an ideal entropy encoder, such as arithmetic coding (Chapter 4).

This chapter describes the details of Huffman encoding and decoding and covers related topics such as the height of a Huffman code tree, canonical Huffman codes, and an adaptive Huffman algorithm. Following this, Section 2.4 illustrates an important application of the Huffman method to facsimile compression.

**David Huffman (1925–1999)**

Being originally from Ohio, it is no wonder that Huffman went to Ohio State University for his BS (in electrical engineering). What is unusual was his age (18) when he earned it in 1944. After serving in the United States Navy, he went back to Ohio State for an MS degree (1949) and then to MIT, for a PhD (1953, electrical engineering).

That same year, Huffman joined the faculty at MIT. In 1967, he made his only career move when he went to the University of California, Santa Cruz as the founding faculty member of the Computer Science Department. During his long tenure at UCSC, Huffman played a major role in the development of the department (he served as chair from 1970 to 1973) and he is known for his motto "my products are my students." Even after his retirement, in 1994, he remained active in the department, teaching information theory and signal analysis courses.

Huffman developed his celebrated algorithm as a term paper that he wrote in lieu of taking a final examination in an information theory class he took at MIT in 1951. The professor, Robert Fano, proposed the problem of constructing the shortest variable-length code for a set of symbols with known probabilities of occurrence.

It should be noted that in the late 1940s, Fano himself (and independently, also Claude Shannon) had developed a similar, but suboptimal, algorithm known today as the Shannon–Fano method ([Shannon 48] and [Fano 49]). The difference between the two algorithms is that the Shannon–Fano code tree is built from the top down, while the Huffman code tree is constructed from the bottom up.

Huffman made significant contributions in several areas, mostly information theory and coding, signal designs for radar and communications, and design procedures for asynchronous logical circuits. Of special interest is the well-known Huffman algorithm for constructing a set of optimal prefix codes for data with known frequencies of occurrence. At a certain point he became interested in the mathematical properties of "zero curvature" surfaces, and developed this interest into techniques for folding paper into unusual sculptured shapes (the so-called computational origami).

# 2.1 Huffman Encoding

The Huffman encoding algorithm starts by constructing a list of all the alphabet symbols in descending order of their probabilities. It then constructs, from the bottom up, a binary tree with a symbol at every leaf. This is done in steps, where at each step two symbols with the smallest probabilities are selected, added to the top of the partial tree, deleted from the list, and replaced with an auxiliary symbol representing the two original symbols. When the list is reduced to just one auxiliary symbol (representing the entire alphabet), the tree is complete. The tree is then traversed to determine the codewords of the symbols.

This process is best illustrated by an example. Given five symbols with probabilities as shown in Figure 2.1a, they are paired in the following order:

1. $a_4$ is combined with $a_5$ and both are replaced by the combined symbol $a_{45}$, whose probability is 0.2.
2. There are now four symbols left, $a_1$, with probability 0.4, and $a_2$, $a_3$, and $a_{45}$, with probabilities 0.2 each. We arbitrarily select $a_3$ and $a_{45}$ as the two symbols with smallest probabilities, combine them, and replace them with the auxiliary symbol $a_{345}$, whose probability is 0.4.
3. Three symbols are now left, $a_1$, $a_2$, and $a_{345}$, with probabilities 0.4, 0.2, and 0.4, respectively. We arbitrarily select $a_2$ and $a_{345}$, combine them, and replace them with the auxiliary symbol $a_{2345}$, whose probability is 0.6.
4. Finally, we combine the two remaining symbols, $a_1$ and $a_{2345}$, and replace them with $a_{12345}$ with probability 1.

The tree is now complete. It is shown in Figure 2.1a "lying on its side" with its root on the right and its five leaves on the left. To assign the codewords, we arbitrarily assign a bit of 1 to the top edge, and a bit of 0 to the bottom edge, of every pair of edges. This results in the codewords 0, 10, 111, 1101, and 1100. The assignments of bits to the edges is arbitrary.

The average size of this code is $0.4 \times 1 + 0.2 \times 2 + 0.2 \times 3 + 0.1 \times 4 + 0.1 \times 4 = 2.2$ bits/symbol, but even more importantly, the Huffman code is not unique. Some of the steps above were chosen arbitrarily, because there were more than two symbols with smallest probabilities. Figure 2.1b shows how the same five symbols can be combined differently to obtain a different Huffman code (11, 01, 00, 101, and 100). The average size of this code is $0.4 \times 2 + 0.2 \times 2 + 0.2 \times 2 + 0.1 \times 3 + 0.1 \times 3 = 2.2$ bits/symbol, the same as the previous code.

◇ **Exercise 2.1:** Given the eight symbols A, B, C, D, E, F, G, and H with probabilities 1/30, 1/30, 1/30, 2/30, 3/30, 5/30, 5/30, and 12/30, draw three different Huffman trees with heights 5 and 6 for these symbols and compute the average code size for each tree.

◇ **Exercise 2.2:** Figure Ans.1d shows another Huffman tree, with height 4, for the eight symbols introduced in Exercise 2.1. Explain why this tree is wrong.

It turns out that the arbitrary decisions made in constructing the Huffman tree affect the individual codes but not the average size of the code. Still, we have to answer the obvious question, which of the different Huffman codes for a given set of symbols is best? The answer, while not obvious, is simple: The best code is the one with the

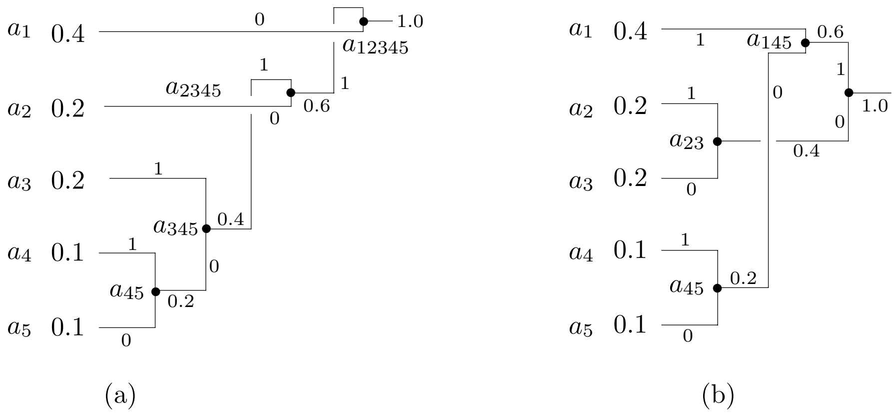



Figure 2.1: Huffman Codes.

smallest variance. The variance of a code measures how much the sizes of the individual codewords deviate from the average size. The variance of the code of Figure 2.1a is

$$0.4(1-2.2)^2+0.2(2-2.2)^2+0.2(3-2.2)^2+0.1(4-2.2)^2+0.1(4-2.2)^2=1.36,$$

while the variance of code 2.1b is

$$0.4(2-2.2)^2+0.2(2-2.2)^2+0.2(2-2.2)^2+0.1(3-2.2)^2+0.1(3-2.2)^2=0.16.$$

Code 2.1b is therefore preferable (see below). A careful look at the two trees shows how to select the one we want. In the tree of Figure 2.1a, symbol $a_{45}$ is combined with $a_3$, whereas in the tree of 2.1b $a_{45}$ is combined with $a_1$. The rule is: When there are more than two smallest-probability nodes, select the ones that are lowest and highest in the tree and combine them. This will combine symbols of low probability with symbols of high probability, thereby reducing the total variance of the code.

If the encoder simply writes the compressed data on a file, the variance of the code makes no difference. A small-variance Huffman code is preferable only in cases where the encoder transmits the compressed data, as it is being generated, over a network. In such a case, a code with large variance causes the encoder to generate bits at a rate that varies all the time. Since the bits have to be transmitted at a constant rate, the encoder has to use a buffer. Bits of the compressed data are entered into the buffer as they are being generated and are moved out of it at a constant rate, to be transmitted. It is easy to see intuitively that a Huffman code with zero variance will enter bits into the buffer at a constant rate, so only a short buffer will be needed. The larger the code variance, the more variable is the rate at which bits enter the buffer, requiring the encoder to use a larger buffer.

The following claim is sometimes found in the literature:

> It can be shown that the size of the Huffman code of a symbol $a_i$ with probability $P_i$ is always less than or equal to $\lceil -\log_2 P_i \rceil$.

Even though it is correct in many cases, this claim is not true in general. It seems to be a wrong corollary drawn by some authors from the Kraft–McMillan inequality, Equation (1.4). The author is indebted to Guy Blelloch for pointing this out and also for the example of Table 2.2.

| $P_i$ | Code | $-\log_2 P_i$ | $\lceil -\log_2 P_i \rceil$ |
|---|---|---|---|
| .01 | 000 | 6.644 | 7 |
| *.30 | 001 | 1.737 | 2 |
| .34 | 01 | 1.556 | 2 |
| .35 | 1 | 1.515 | 2 |

Table 2.2: A Huffman Code Example.

◇ **Exercise 2.3:** Find an example where the size of the Huffman code of a symbol $a_i$ is greater than $\lceil -\log_2 P_i \rceil$.

◇ **Exercise 2.4:** It seems that the size of a code must also depend on the number $n$ of symbols (the size of the alphabet). A small alphabet requires just a few codes, so they can all be short; a large alphabet requires many codes, so some must be long. This being so, how can we say that the size of the code of $a_i$ depends just on the probability $P_i$?

Figure 2.3 shows a Huffman code for the 26 letters.

As a self-exercise, the reader may calculate the average size, entropy, and variance of this code.

◇ **Exercise 2.5:** Discuss the Huffman codes for equal probabilities.

Exercise 2.5 shows that symbols with equal probabilities don't compress under the Huffman method. This is understandable, since strings of such symbols normally make random text, and random text does not compress. There may be special cases where strings of symbols with equal probabilities are not random and can be compressed. A good example is the string $a_1 a_1 \ldots a_1 a_2 a_2 \ldots a_2 a_3 a_3 \ldots$ in which each symbol appears in a long run. This string can be compressed with RLE but not with Huffman codes.

Notice that the Huffman method cannot be applied to a two-symbol alphabet. In such an alphabet, one symbol can be assigned the code 0 and the other code 1. The Huffman method cannot assign to any symbol a code shorter than one bit, so it cannot improve on this simple code. If the original data (the source) consists of individual bits, such as in the case of a bi-level (monochromatic) image, it is possible to combine several bits (perhaps four or eight) into a new symbol and pretend that the alphabet consists of these (16 or 256) symbols. The problem with this approach is that the original binary data may have certain statistical correlations between the bits, and some of these correlations would be lost when the bits are combined into symbols. When a typical bi-level image (a painting or a diagram) is digitized by scan lines, a pixel is more likely to be followed by an identical pixel than by the opposite one. We therefore have a file that can start with either a 0 or a 1 (each has 0.5 probability of being the first bit). A zero is more likely to be followed by another 0 and a 1 by another 1. Figure 2.4 is a finite-state machine illustrating this situation. If these bits are combined into, say, groups of eight,

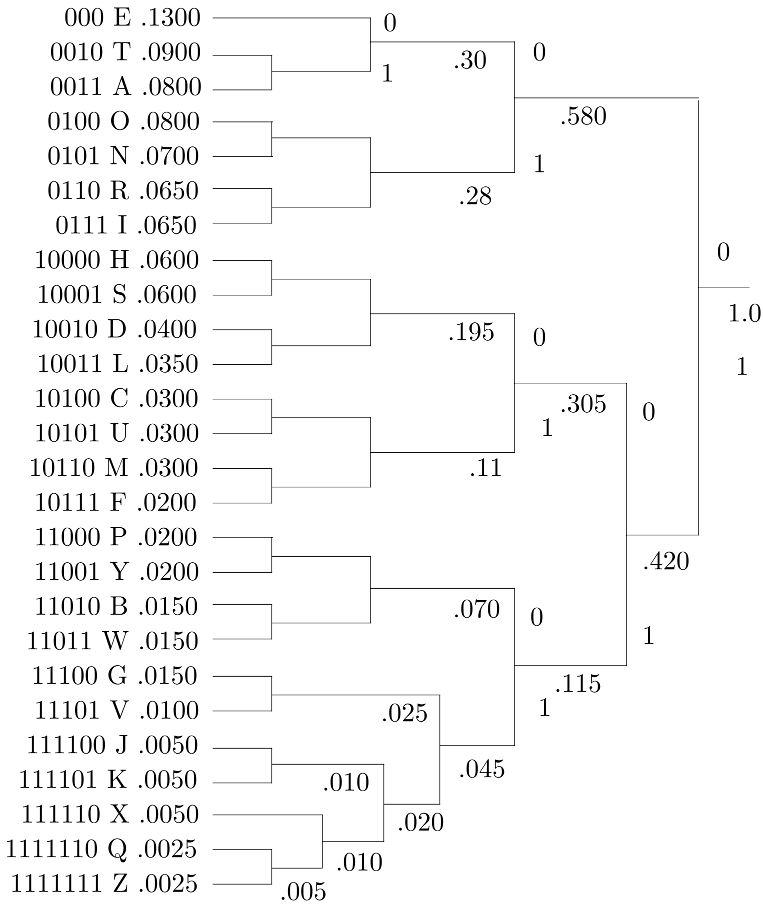



Figure 2.3: A Huffman Code for the 26-Letter Alphabet.

the bits inside a group will still be correlated, but the groups themselves will not be correlated by the original pixel probabilities. If the input data contains, e.g., the two adjacent groups 00011100 and 00001110, they will be encoded independently, ignoring the correlation between the last 0 of the first group and the first 0 of the next group. Selecting larger groups improves this situation but increases the number of groups, which implies more storage for the code table and longer time to calculate the table.

◇ **Exercise 2.6:** How does the number of groups increase when the group size increases from $s$ bits to $s+n$ bits?

A more complex approach to image compression by Huffman coding is to create several complete sets of Huffman codes. If the group size is, e.g., eight bits, then several sets of 256 codes are generated. When a symbol S is to be encoded, one of the sets is selected, and S is encoded using its code in that set. The choice of set depends on the symbol preceding S.

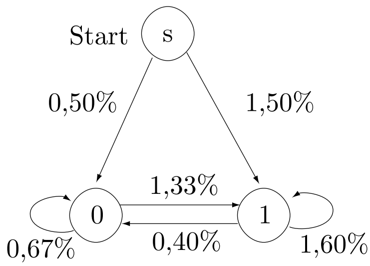



Figure 2.4: A Finite-State Machine.

◇ **Exercise 2.7:** Imagine an image with 8-bit pixels where half the pixels have values 127 and the other half have values 128. Analyze the performance of RLE on the individual bitplanes of such an image, and compare it with what can be achieved with Huffman coding.

Which two integers come next in the infinite sequence 38, 24, 62, 12, 74, . . . ?

# 2.2 Huffman Decoding

Before starting the compression of a data file, the compressor (encoder) has to determine the codes. It does that based on the probabilities (or frequencies of occurrence) of the symbols. The probabilities or frequencies have to be written, as side information, on the output, so that any Huffman decompressor (decoder) will be able to decompress the data. This is easy, because the frequencies are integers and the probabilities can be written as scaled integers. It normally adds just a few hundred bytes to the output. It is also possible to write the variable-length codes themselves on the output, but this may be awkward, because the codes have different sizes. It is also possible to write the Huffman tree on the output, but this may require more space than just the frequencies.

In any case, the decoder must know what is at the start of the compressed file, read it, and construct the Huffman tree for the alphabet. Only then can it read and decode the rest of its input. The algorithm for decoding is simple. Start at the root and read the first bit off the input (the compressed file). If it is zero, follow the bottom edge of the tree; if it is one, follow the top edge. Read the next bit and move another edge toward the leaves of the tree. When the decoder arrives at a leaf, it finds there the original, uncompressed symbol (normally its ASCII code), and that code is emitted by the decoder. The process starts again at the root with the next bit.

This process is illustrated for the five-symbol alphabet of Figure 2.5. The four-symbol input string $a_4a_2a_5a_1$ is encoded into 1001100111. The decoder starts at the root, reads the first bit 1, and goes up. The second bit 0 sends it down, as does the third bit. This brings the decoder to leaf $a_4$, which it emits. It again returns to the root, reads 110, moves up, up, and down, to reach leaf $a_2$, and so on.

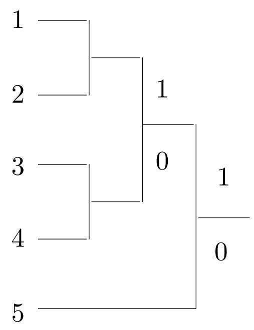



Figure 2.5: Huffman Codes for Equal Probabilities.

## 2.2.1 Fast Huffman Decoding

Decoding a Huffman-compressed file by sliding down the code tree for each symbol is conceptually simple, but slow. The compressed file has to be read bit by bit and the decoder has to advance a node in the code tree for each bit. The method of this section, originally conceived by [Choueka et al. 85] but later reinvented by others, uses preset partial-decoding tables. These tables depend on the particular Huffman code used, but not on the data to be decoded. The compressed file is read in chunks of $k$ bits each (where $k$ is normally 8 or 16 but can have other values) and the current chunk is used as a pointer to a table. The table entry that is selected in this way can decode several symbols and it also points the decoder to the table to be used for the next chunk.

As an example, consider the Huffman code of Figure 2.1a, where the five codewords are 0, 10, 111, 1101, and 1100. The string of symbols $a_1a_1a_2a_4a_3a_1a_5\ldots$ is compressed by this code to the string $0|0|10|1101|111|0|1100\ldots$. We select $k=3$ and read this string in 3-bit chunks $001|011|011|110|110|0\ldots$. Examining the first chunk, it is easy to see that it should be decoded into $a_1a_1$ followed by the single bit 1 which is the prefix of another codeword. The first chunk is $001 = 1_{10}$, so we set entry 1 of the first table (table 0) to the pair $(a_1a_1, 1)$. When chunk 001 is used as a pointer to table 0, it points to entry 1, which immediately provides the decoder with the two decoded symbols $a_1a_1$ and also directs it to use table 1 for the next chunk. Table 1 is used when a partially-decoded chunk ends with the single-bit prefix 1. The next chunk is $011 = 3_{10}$, so entry 3 of table 1 corresponds to the encoded bits $1|011$. Again, it is easy to see that these should be decoded to $a_2$ and there is the prefix 11 left over. Thus, entry 3 of table 1 should be $(a_2, 2)$. It provides the decoder with the single symbol $a_2$ and also directs it to use table 2 next (the table that corresponds to prefix 11). The next chunk is again $011 = 3_{10}$, so entry 3 of table 2 corresponds to the encoded bits $11|011$. It is again obvious that these should be decoded to $a_4$ with a prefix of 1 left over. This process continues until the end of the encoded input. Figure 2.6 is the simple decoding algorithm in pseudocode.

Table 2.7 lists the four tables required to decode this code. It is easy to see that they correspond to the prefixes $\Lambda$ (null), 1, 11, and 110. A quick glance at Figure 2.1a shows that these correspond to the root and the four interior nodes of the Huffman code tree. Thus, each partial-decoding table corresponds to one of the four prefixes of this code. The number $m$ of partial-decoding tables therefore equals the number of interior nodes (plus the root) which is one less than the number $N$ of symbols of the alphabet.

```
i←0; output←null;
repeat
  j←input next chunk;
  (s,i)←Table_i[j];
  append s to output;
until end-of-input
```

Figure 2.6: Fast Huffman Decoding.

| $T_0 = \Lambda$ | | | $T_1 = 1$ | | | $T_2 = 11$ | | | $T_3 = 110$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 000 | $a_1a_1a_1$ | 0 | 1\|000 | $a_2a_1a_1$ | 0 | 11\|000 | $a_5a_1$ | 0 | 110\|000 | $a_5a_1a_1$ | 0 |
| 001 | $a_1a_1$ | 1 | 1\|001 | $a_2a_1$ | 1 | 11\|001 | $a_5$ | 1 | 110\|001 | $a_5a_1$ | 1 |
| 010 | $a_1a_2$ | 0 | 1\|010 | $a_2a_2$ | 0 | 11\|010 | $a_4a_1$ | 0 | 110\|010 | $a_5a_2$ | 0 |
| 011 | $a_1$ | 2 | 1\|011 | $a_2$ | 2 | 11\|011 | $a_4$ | 1 | 110\|011 | $a_5$ | 2 |
| 100 | $a_2a_1$ | 0 | 1\|100 | $a_5$ | 0 | 11\|100 | $a_3a_1a_1$ | 0 | 110\|100 | $a_4a_1a_1$ | 0 |
| 101 | $a_2$ | 1 | 1\|101 | $a_4$ | 0 | 11\|101 | $a_3a_1$ | 1 | 110\|101 | $a_4a_1$ | 1 |
| 110 | – | 3 | 1\|110 | $a_3a_1$ | 0 | 11\|110 | $a_3a_2$ | 0 | 110\|110 | $a_4a_2$ | 0 |
| 111 | $a_3$ | 0 | 1\|111 | $a_3$ | 1 | 11\|111 | $a_3$ | 2 | 110\|111 | $a_4$ | 2 |

Table 2.7: Partial-Decoding Tables for a Huffman Code.

Notice that some chunks (such as entry 110 of table 0) simply send the decoder to another table and do not provide any decoded symbols. Also, there is a trade-off between chunk size (and thus table size) and decoding speed. Large chunks speed up decoding, but require large tables. A large alphabet (such as the 128 ASCII characters or the 256 8-bit bytes) also requires a large set of tables. The problem with large tables is that the decoder has to set up the tables after it has read the Huffman codes from the compressed stream and before decoding can start, and this process may preempt any gains in decoding speed provided by the tables.

To set up the first table (table 0, which corresponds to the null prefix $\Lambda$), the decoder generates the $2^k$ bit patterns 0 through $2^k - 1$ (the first column of Table 2.7) and employs the decoding method of Section 2.2 to decode each pattern. This yields the second column of Table 2.7. Any remainders left are prefixes and are converted by the decoder to table numbers. They become the third column of the table. If no remainder is left, the third column is set to 0 (use table 0 for the next chunk). Each of the other partial-decoding tables is set in a similar way. Once the decoder decides that table 1 corresponds to prefix $p$, it generates the $2^k$ patterns $p|00\ldots0$ through $p|11\ldots1$ that become the first column of that table. It then decodes that column to generate the remaining two columns.

This method was conceived in 1985, when storage costs were considerably higher than today (early 2007). This prompted the developers of the method to find ways to cut down the number of partial-decoding tables, but these techniques are less important today and are not described here.

## 2.2.2 Average Code Size

Figure 2.8a shows a set of five symbols with their probabilities and a typical Huffman tree. Symbol A appears 55% of the time and is assigned a 1-bit code, so it contributes $0.55 \cdot 1$ bits to the average code size. Symbol E appears only 2% of the time and is assigned a 4-bit Huffman code, so it contributes $0.02 \cdot 4 = 0.08$ bits to the code size. The average code size is therefore easily computed as

$$0.55 \cdot 1 + 0.25 \cdot 2 + 0.15 \cdot 3 + 0.03 \cdot 4 + 0.02 \cdot 4 = 1.7 \text{ bits per symbol.}$$

Surprisingly, the same result is obtained by adding the values of the four internal nodes of the Huffman code tree $0.05 + 0.2 + 0.45 + 1 = 1.7$. This provides a way to calculate the average code size of a set of Huffman codes without any multiplications. Simply add the values of all the internal nodes of the tree. Table 2.9 illustrates why this works.

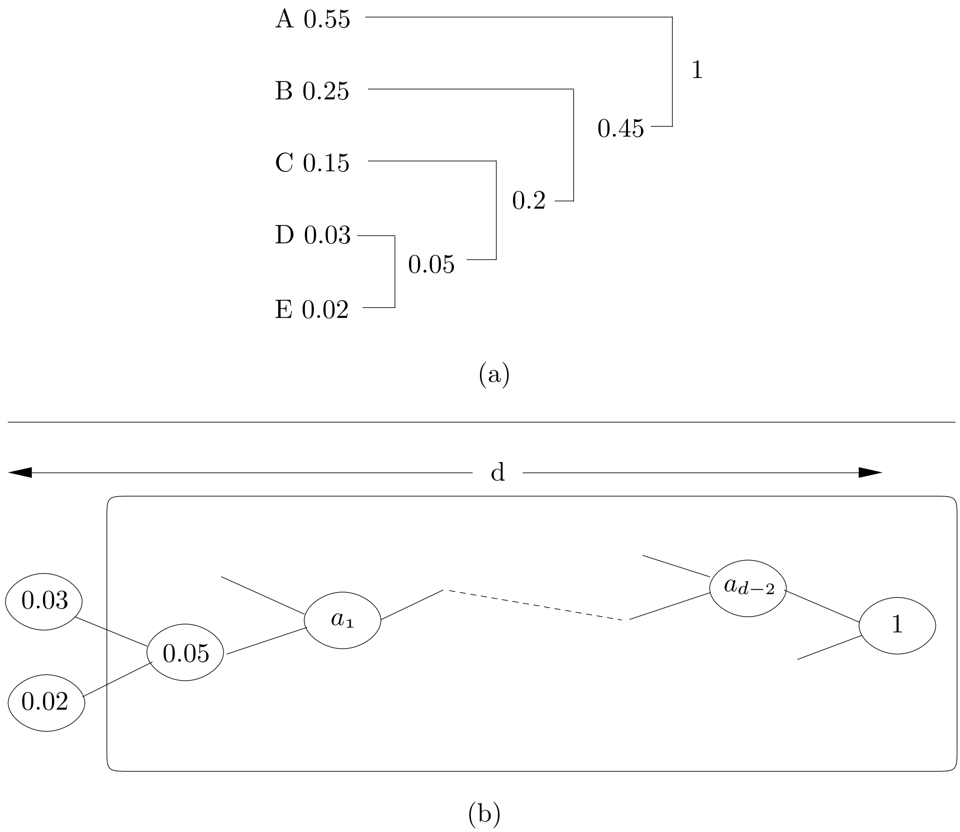



Figure 2.8: Huffman Code Trees.

(Internal nodes are shown in italics in this table.) The left column consists of the values of all the internal nodes. The right columns show how each internal node is the sum of

| | | |
|---|---|---|
| $\mathit{.05}$ = | | $.02+.03$ |
| $\mathit{.20}$ | $=\mathit{.05}+.15$ | $=.02+.03+.15$ |
| $\mathit{.45}$ | $=\mathit{.20}+.25$ | $=.02+.03+.15+.25$ |
| $\mathit{1.0}$ | $=\mathit{.45}+.55$ | $=.02+.03+.15+.25+.55$ |

Table 2.9: Composition of Nodes.

| | | |
|---|---|---|
| $\mathit{0.05}$ = | | $= 0.02 + 0.03 + \cdots$ |
| $a_1$ | $= \mathit{0.05} + \ldots$ | $= 0.02 + 0.03 + \cdots$ |
| $a_2$ | $= a_1 \quad + \ldots$ | $= 0.02 + 0.03 + \cdots$ |
| $\vdots$ | = | |
| $a_{d-2}$ | $= a_{d-3} + \ldots$ | $= 0.02 + 0.03 + \cdots$ |
| $\mathit{1.0}$ | $= a_{d-2} + \ldots$ | $= 0.02 + 0.03 + \cdots$ |

Table 2.10: Composition of Nodes.

some of the leaf nodes. Summing the values in the left column yields 1.7, and summing the other columns shows that this 1.7 is the sum of the four values 0.02, the four values 0.03, the three values 0.15, the two values 0.25, and the single value 0.55.

This argument can be extended to the general case. It is easy to show that, in a Huffman-like tree (a tree where each node is the sum of its children), the weighted sum of the leaves, where the weights are the distances of the leaves from the root, equals the sum of the internal nodes. (This property has been communicated to the author by J. Motil.)

Figure 2.8b shows such a tree, where we assume that the two leaves 0.02 and 0.03 have $d$-bit Huffman codes. Inside the tree, these leaves become the children of internal node 0.05, which, in turn, is connected to the root by means of the $d-2$ internal nodes $a_1$ through $a_{d-2}$. Table 2.10 has $d$ rows and shows that the two values 0.02 and 0.03 are included in the various internal nodes exactly $d$ times. Adding the values of all the internal nodes produces a sum that includes the contributions $0.02 \cdot d + 0.03 \cdot d$ from the two leaves. Since these leaves are arbitrary, it is clear that this sum includes similar contributions from all the other leaves, so this sum is the average code size. Since this sum also equals the sum of the left column, which is the sum of the internal nodes, it is clear that the sum of the internal nodes equals the average code size.

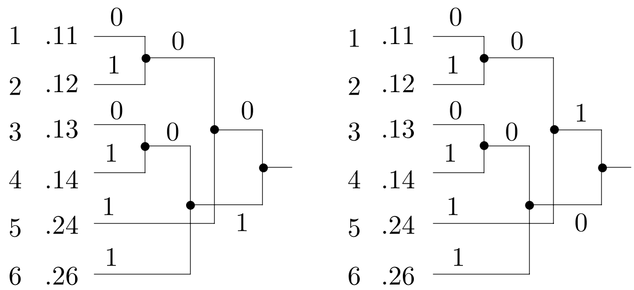
Notice that this proof does not assume that the tree is binary. The property illustrated here exists for any tree where a node contains the sum of its children.

## 2.2.3 Number of Codes

Since the Huffman code is not unique, the natural question is: How many different codes are there? Figure 2.11a shows a Huffman code tree for six symbols, from which we can answer this question in two different ways as follows:

Answer 1. The tree of 2.11a has five interior nodes, and in general, a Huffman code tree for $n$ symbols has $n-1$ interior nodes. Each interior node has two edges coming out of it, labeled 0 and 1. Swapping the two labels produces a different Huffman code tree, so the total number of different Huffman code trees is $2^{n-1}$ (in our example, $2^5$ or 32). The tree of Figure 2.11b, for example, shows the result of swapping the labels of the two edges of the root. Table 2.12a,b lists the codes generated by the two trees.

Answer 2. The six codes of Table 2.12a can be divided into the four classes $00x$, $10y$, 01, and 11, where $x$ and $y$ are 1-bit each. It is possible to create different Huffman codes by changing the first two bits of each class. Since there are four classes, this is the same as creating all the permutations of four objects, something that can be done in $4! = 24$ ways. In each of the 24 permutations it is also possible to change the values




(a) (b)

Figure 2.11: Two Huffman Code Trees.

| (a) | (b) | (c) |
|---|---|---|
| 000 | 100 | 000 |
| 001 | 101 | 001 |
| 100 | 000 | 010 |
| 101 | 001 | 011 |
| 01 | 11 | 10 |
| 11 | 01 | 11 |

Table 2.12.

of $x$ and $y$ in four different ways (since they are bits) so the total number of different Huffman codes in our six-symbol example is $24 \times 4 = 96$.

The two answers are different because they count different things. Answer 1 counts the number of different Huffman code trees, while answer 2 counts the number of different Huffman codes. It turns out that our example can generate 32 different code trees but only 94 different codes instead of 96. This shows that there are Huffman codes that cannot be generated by the Huffman method! Table 2.12c shows such an example. A look at the trees of Figure 2.11 should convince the reader that the codes of symbols 5 and 6 must start with different bits, but in the code of Table 2.12c they both start with 1. This code is therefore impossible to generate by any relabeling of the nodes of the trees of Figure 2.11.

## 2.2.4 Ternary Huffman Codes

The Huffman code is not unique. Moreover, it does not have to be binary! The Huffman method can easily be applied to codes based on other number systems. Figure 2.13a shows a Huffman code tree for five symbols with probabilities 0.15, 0.15, 0.2, 0.25, and 0.25. The average code size is

$$2\times0.25 + 3\times0.15 + 3\times0.15 + 2\times0.20 + 2\times0.25 = 2.3\,\text{bits/symbol}.$$

Figure 2.13b shows a ternary Huffman code tree for the same five symbols. The tree is constructed by selecting, at each step, three symbols with the smallest probabilities and merging them into one parent symbol, with the combined probability. The average code size of this tree is

$$2\times0.15 + 2\times0.15 + 2\times0.20 + 1\times0.25 + 1\times0.25 = 1.5\,\text{trits/symbol}.$$

Notice that the ternary codes use the digits 0, 1, and 2.

◇ **Exercise 2.8:** Given seven symbols with probabilities 0.02, 0.03, 0.04, 0.04, 0.12, 0.26, and 0.49, construct binary and ternary Huffman code trees for them and calculate the average code size in each case.

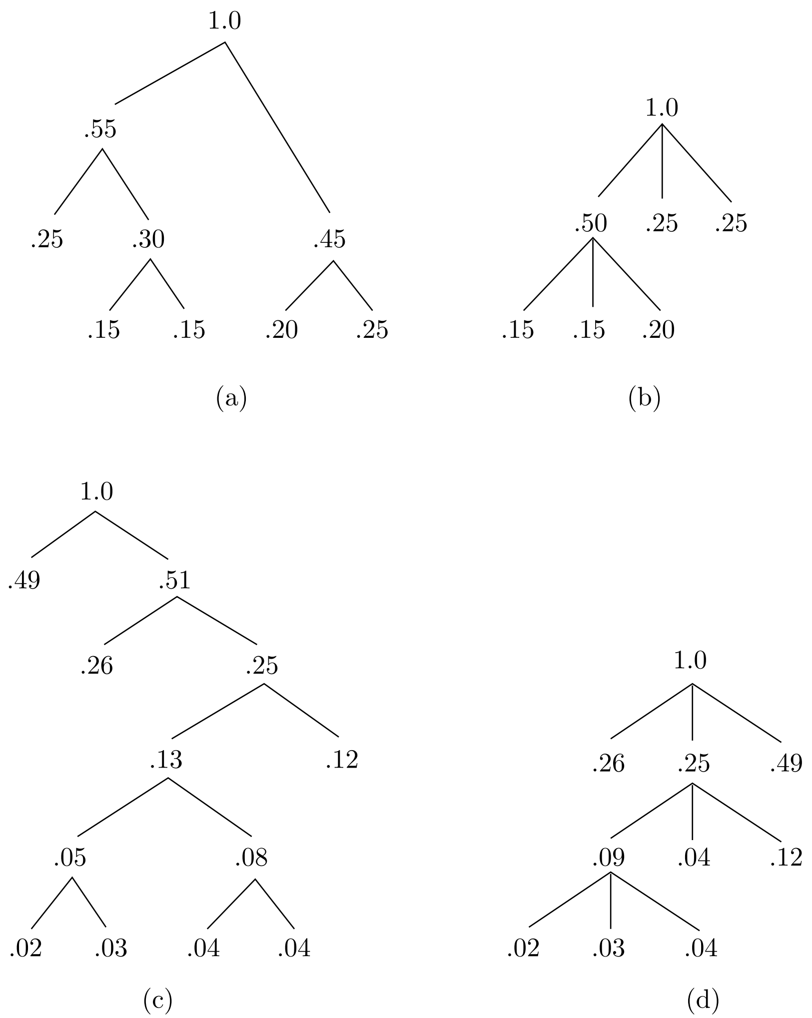



Figure 2.13: Binary and Ternary Huffman Code Trees.

## 2.2.5 Height of a Huffman Tree

The height of the code tree generated by the Huffman algorithm may sometimes be important because the height is also the length of the longest code in the tree. The Deflate method (Section 3.3), for example, limits the lengths of certain Huffman codes to just three bits.

It is easy to see that the shortest Huffman tree is created when the symbols have equal probabilities. If the symbols are denoted by A, B, C, and so on, then the algorithm combines pairs of symbols, such A and B, C and D, in the lowest level, and the rest of the tree consists of interior nodes as shown in Figure 2.14a. The tree is balanced or close to balanced and its height is $\lceil \log_2 n \rceil$. In the special case where the number of symbols $n$ is a power of 2, the height is exactly $\log_2 n$. In order to generate the tallest tree, we need to assign probabilities to the symbols such that each step in the Huffman method will increase the height of the tree by 1. Recall that each step in the Huffman algorithm combines two symbols. Thus, the tallest tree is obtained when the first step combines two of the $n$ symbols and each subsequent step combines the result of its predecessor with one of the remaining symbols (Figure 2.14b). The height of the complete tree is therefore $n-1$, and it is referred to as a lopsided or unbalanced tree.

It is easy to see what symbol probabilities result in such a tree. Denote the two smallest probabilities by $a$ and $b$. They are combined in the first step to form a node whose probability is $a+b$. The second step will combine this node with an original symbol if one of the symbols has probability $a+b$ (or smaller) and all the remaining symbols have greater probabilities. Thus, after the second step, the root of the tree has probability $a+b+(a+b)$ and the third step will combine this root with one of the remaining symbols if its probability is $a+b+(a+b)$ and the probabilities of the remaining $n-4$ symbols are greater. It does not take much to realize that the symbols have to have probabilities $p_1=a$, $p_2=b$, $p_3=a+b=p_1+p_2$, $p_4=b+(a+b)=p_2+p_3$, $p_5=(a+b)+(a+2b)=p_3+p_4$, $p_6=(a+2b)+(2a+3b)=p_4+p_5$, and so on (Figure 2.14c). These probabilities form a Fibonacci sequence whose first two elements are $a$ and $b$. As an example, we select $a=5$ and $b=2$ and generate the 5-number Fibonacci sequence 5, 2, 7, 9, and 16. These five numbers add up to 39, so dividing them by 39 produces the five probabilities 5/39, 2/39, 7/39, 9/39, and 15/39. The Huffman tree generated by them has a maximal height (which is 4).

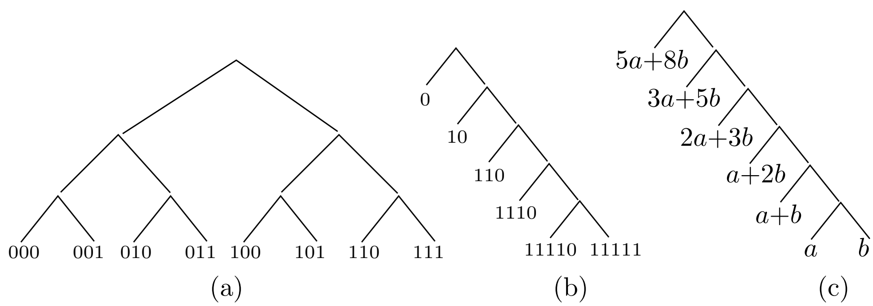



Figure 2.14: Shortest and Tallest Huffman Trees.

In principle, symbols in a set can have any probabilities, but in practice, the probabilities of symbols in an input file are computed by counting the number of occurrences of each symbol. Imagine a text file where only the nine symbols A through I appear. In order for such a file to produce the tallest Huffman tree, where the codes will have lengths from 1 to 8 bits, the frequencies of occurrence of the nine symbols have to form a Fibonacci sequence of probabilities. This happens when the frequencies of the symbols are 1, 1, 2, 3, 5, 8, 13, 21, and 34 (or integer multiples of these). The sum of these frequencies is 88, so our file has to be at least that long in order for a symbol to have 8-bit Huffman codes. Similarly, if we want to limit the sizes of the Huffman codes of a set of $n$ symbols to 16 bits, we need to count frequencies of at least 4,180 symbols. To limit the code sizes to 32 bits, the minimum data size is 9,227,464 symbols.

If a set of symbols happens to have the Fibonacci probabilities and therefore results in a maximal-height Huffman tree with codes that are too long, the tree can be reshaped (and the maximum code length shortened) by slightly modifying the symbol probabilities, so they are not much different from the original, but do not form a Fibonacci sequence.

## 2.2.6 Canonical Huffman Codes

The code of Table 2.12c has a simple interpretation. It assigns the first four symbols the 3-bit codes 0, 1, 2, and 3, and the last two symbols the 2-bit codes 2 and 3. This is an example of a *canonical Huffman code*. The word "canonical" means that this particular code has been selected from among the several (or even many) possible Huffman codes because its properties make it easy and fast to use.

> Canonical (adjective): Conforming to orthodox or well-established rules or patterns, as of procedure.

Table 2.15 shows a slightly bigger example of a canonical Huffman code. Imagine a set of 16 symbols (whose probabilities are irrelevant and are not shown) such that four symbols are assigned 3-bit codes, five symbols are assigned 5-bit codes, and the remaining seven symbols are assigned 6-bit codes. Table 2.15a shows a set of possible Huffman codes, while Table 2.15b shows a set of canonical Huffman codes. It is easy to see that the seven 6-bit canonical codes are simply the 6-bit integers 0 through 6. The five codes are the 5-bit integers 4 through 8, and the four codes are the 3-bit integers 3 through 6. We first show how these codes are generated and then how they are used.

| | (a) | (b) | | (a) | (b) |
|---|---|---|---|---|---|
| 1: | 000 | 011 | 9: | 10100 | 01000 |
| 2: | 001 | 100 | 10: | 101010 | 000000 |
| 3: | 010 | 101 | 11: | 101011 | 000001 |
| 4: | 011 | 110 | 12: | 101100 | 000010 |
| 5: | 10000 | 00100 | 13: | 101101 | 000011 |
| 6: | 10001 | 00101 | 14: | 101110 | 000100 |
| 7: | 10010 | 00110 | 15: | 101111 | 000101 |
| 8: | 10011 | 00111 | 16: | 110000 | 000110 |

Table 2.15.

| | | | | | | |
|---|---|---|---|---|---|---|
| length: | 1 | 2 | 3 | 4 | 5 | 6 |
| numl: | 0 | 0 | 4 | 0 | 5 | 7 |
| first: | 2 | 4 | 3 | 5 | 4 | 0 |

Table 2.16.

The top row (length) of Table 2.16 lists the possible code lengths, from 1 to 6 bits. The second row (numl) lists the number of codes of each length, and the bottom row (first) lists the first code in each group. This is why the three groups of codes start with values 3, 4, and 0. To obtain the top two rows we need to compute the lengths of all the Huffman codes for the given alphabet (see below). The third row is computed by setting "`first[6]:=0;`" and iterating

```
for l:=5 downto 1 do first[l]:=⌈(first[l+1]+numl[l+1])/2⌉;
```

This guarantees that all the 3-bit prefixes of codes longer than three bits will be less than `first[3]` (which is 3), all the 5-bit prefixes of codes longer than five bits will be less than `first[5]` (which is 4), and so on.

Now for the use of these unusual codes. Canonical Huffman codes are useful in cases where the alphabet is large and where fast decoding is mandatory. Because of the way the codes are constructed, it is easy for the decoder to identify the length of a code by reading and examining input bits one by one. Once the length is known, the symbol can be found in one step. The pseudocode listed here shows the rules for decoding:

```
l:=1; input v;
while v<first[l]
append next input bit to v; l:=l+1;
endwhile
```

As an example, suppose that the next code is 00110. As bits are input and appended to v, it goes through the values 0, 00 = 0, 001 = 1, 0011 = 3, 00110 = 6, while `l` is incremented from 1 to 5. All steps except the last satisfy `v<first[l]`, so the last step determines the value of `l` (the code length) as 5. The symbol itself is found by subtracting $\texttt{v} - \texttt{first[5]} = 6 - 4 = 2$, so it is the third symbol (numbering starts at 0) in group $\texttt{l} = 5$ (symbol 7 of the 16 symbols).

The last point to be discussed is the encoder. In order to construct the canonical Huffman code, the encoder needs to know the length of the Huffman code of every symbol. The main problem is the large size of the alphabet, which may make it impractical or even impossible to build the entire Huffman code tree in memory. There is an algorithm—described in [Hirschberg and Lelewer 90], [Sieminski 88], and [Salomon 07]—that solves this problem. It calculates the code sizes for an alphabet of $n$ symbols using just one array of size $2n$.

> Considine's Law. Whenever one word or letter can change the entire meaning of a sentence, the probability of an error being made will be in direct proportion to the embarrassment it will cause.
>
> —Bob Considine

One morning I was on my way to the market and met a man with four wives (perfectly legal where we come from). Each wife had four bags, containing four dogs each, and each dog had four puppies. The question is (think carefully) how many were going to the market?

## 2.3 Adaptive Huffman Coding

The Huffman method assumes that the frequencies of occurrence of all the symbols of the alphabet are known to the compressor. In practice, the frequencies are seldom, if ever, known in advance. One approach to this problem is for the compressor to read the original data twice. The first time, it only counts the frequencies; the second time, it compresses the data. Between the two passes, the compressor constructs the Huffman tree. Such a two-pass method is sometimes called semiadaptive and is normally too slow to be practical. The method that is used in practice is called adaptive (or dynamic) Huffman coding. This method is the basis of the UNIX `compact` program. The method was originally developed by [Faller 73] and [Gallager 78] with substantial improvements by [Knuth 85].

The main idea is for the compressor and the decompressor to start with an empty Huffman tree and to modify it as symbols are being read and processed (in the case of the compressor, the word "processed" means compressed; in the case of the decompressor, it means decompressed). The compressor and decompressor should modify the tree in the same way, so at any point in the process they should use the same codes, although those codes may change from step to step. We say that the compressor and decompressor are synchronized or that they work in *lockstep* (although they don't necessarily work together; compression and decompression normally take place at different times). The term *mirroring* is perhaps a better choice. The decoder mirrors the operations of the encoder.

Initially, the compressor starts with an empty Huffman tree. No symbols have been assigned codes yet. The first symbol being input is simply written on the output in its uncompressed form. The symbol is then added to the tree and a code assigned to it. The next time this symbol is encountered, its current code is written on the output, and its frequency incremented by 1. Since this modifies the tree, it (the tree) is examined to see whether it is still a Huffman tree (best codes). If not, it is rearranged, an operation that results in modified codes.

The decompressor mirrors the same steps. When it reads the uncompressed form of a symbol, it adds it to the tree and assigns it a code. When it reads a compressed (variable-length) code, it scans the current tree to determine what symbol the code belongs to, and it increments the symbol's frequency and rearranges the tree in the same way as the compressor.

It is immediately clear that the decompressor needs to know whether the item it has just input is an uncompressed symbol (normally, an 8-bit ASCII code, but see Section 2.3.1) or a variable-length code. To remove any ambiguity, each uncompressed symbol is preceded by a special, variable-size *escape code*. When the decompressor reads this code, it knows that the next eight bits are the ASCII code of a symbol that appears in the compressed file for the first time.

> Escape is not his plan. I must face him. Alone.
>
> —David Prowse as Lord Darth Vader in *Star Wars* (1977)

The trouble is that the escape code should not be any of the variable-length codes used for the symbols. These codes, however, are being modified every time the tree is rearranged, which is why the escape code should also be modified. A natural way to do this is to add an empty leaf to the tree, a leaf with a zero frequency of occurrence, that's always assigned to the 0-branch of the tree. Since the leaf is in the tree, it is assigned a variable-length code. This code is the escape code preceding every uncompressed symbol. As the tree is being rearranged, the position of the empty leaf—and thus its code—change, but this escape code is always used to identify uncompressed symbols in the compressed file. Figure 2.17 shows how the escape code moves and changes as the tree grows.

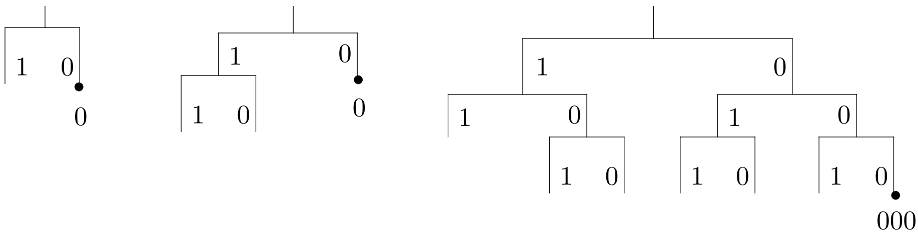


Figure 2.17: The Escape Code.

## 2.3.1 Uncompressed Codes

If the symbols being compressed are ASCII characters, they may simply be assigned their ASCII codes as uncompressed codes. In the general case where there may be any symbols, uncompressed codes of two different sizes can be assigned by a simple method. Here is an example for the case $n = 24$. The first 16 symbols can be assigned the numbers 0 through 15 as their codes. These numbers require only 4 bits, but we encode them in 5 bits. Symbols 17 through 24 can be assigned the numbers $17-16-1 = 0$, $18-16-1 = 1$ through $24-16-1 = 7$ as 4-bit numbers. We end up with the sixteen 5-bit codes 00000, $00001, \ldots, 01111$, followed by the eight 4-bit codes 0000, $0001, \ldots, 0111$.

In general, we assume an alphabet that consists of the $n$ symbols $a_1, a_2, \ldots, a_n$. We select integers $m$ and $r$ such that $2^m \le n < 2^{m+1}$ and $r = n - 2^m$. The first $2^m$ symbols are encoded as the $(m+1)$-bit numbers 0 through $2^m - 1$. The remaining symbols are encoded as $m$-bit numbers such that the code of $a_k$ is $k - 2^m - 1$. This code is also called a phased-in binary code (also a minimal binary code).

## 2.3.2 Modifying the Tree

The chief principle for modifying the tree is to check it each time a symbol is input. If the tree is no longer a Huffman tree, it should be rearranged to become one. A glance at Figure 2.18a shows what it means for a binary tree to be a Huffman tree. The tree in the figure contains five symbols: $A$, $B$, $C$, $D$, and $E$. It is shown with the symbols and their frequencies (in parentheses) after 16 symbols have been input and processed. The property that makes it a Huffman tree is that if we scan it level by level, scanning each level from left to right, and going from the bottom (the leaves) to the top (the root), the frequencies will be in sorted, nondescending order. Thus, the bottom-left node ($A$) has the lowest frequency, and the top-right node (the root) has the highest frequency. This is called the sibling property.

◇ **Exercise 2.9:** Why is this the criterion for a tree to be a Huffman tree?

Here is a summary of the operations needed to update the tree. The loop starts at the current node (the one corresponding to the symbol just input). This node is a leaf that we denote by $X$, with frequency of occurrence $F$. Each iteration of the loop involves three steps as follows:

1. Compare $X$ to its successors in the tree (from left to right and bottom to top). If the immediate successor has frequency $F + 1$ or greater, the nodes are still in sorted order and there is no need to change anything. Otherwise, some successors of $X$ have

identical frequencies of $F$ or smaller. In this case, $X$ should be swapped with the last node in this group (except that $X$ should not be swapped with its parent).
2. Increment the frequency of $X$ from $F$ to $F + 1$. Increment the frequencies of all its parents.
3. If $X$ is the root, the loop stops; otherwise, it repeats with the parent of node $X$.

Figure 2.18b shows the tree after the frequency of node $A$ has been incremented from 1 to 2. It is easy to follow the three rules above to see how incrementing the frequency of $A$ results in incrementing the frequencies of all its parents. No swaps are needed in this simple case because the frequency of $A$ hasn't exceeded the frequency of its immediate successor $B$. Figure 2.18c shows what happens when $A$'s frequency has been incremented again, from 2 to 3. The three nodes following $A$, namely, $B$, $C$, and $D$, have frequencies of 2, so $A$ is swapped with the last of them, $D$. The frequencies of the new parents of $A$ are then incremented, and each is compared with its successor, but no more swaps are needed.

Figure 2.18d shows the tree after the frequency of $A$ has been incremented to 4. Once we decide that $A$ is the current node, its frequency (which is still 3) is compared to that of its successor (4), and the decision is not to swap. $A$'s frequency is incremented, followed by incrementing the frequencies of its parents.

In Figure 2.18e, $A$ is again the current node. Its frequency (4) equals that of its successor, so they should be swapped. This is shown in Figure 2.18f, where $A$'s frequency is 5. The next loop iteration examines the parent of $A$, with frequency 10. It should be swapped with its successor $E$ (with frequency 9), which leads to the final tree of Figure 2.18g.

## 2.3.3 Counter Overflow

The frequency counts are accumulated in the Huffman tree in fixed-size fields, and such fields may overflow. A 16-bit unsigned field can accommodate counts of up to $2^{16} - 1 = 65{,}535$. A simple solution to the counter overflow problem is to watch the count field of the root each time it is incremented, and when it reaches its maximum value, to *rescale* all the frequency counts by dividing them by 2 (integer division). In practice, this is done by dividing the count fields of the leaves, then updating the counts of the interior nodes. Each interior node gets the sum of the counts of its children. The problem is that the counts are integers, and integer division reduces precision. This may change a Huffman tree to one that does not satisfy the sibling property.

A simple example is shown in Figure 2.18h. After the counts of the leaves are halved, the three interior nodes are updated as shown in Figure 2.18i. The latter tree, however, is no longer a Huffman tree, since the counts are no longer in sorted order. The solution is to rebuild the tree each time the counts are rescaled, which does not happen very often. A Huffman data compression program intended for general use should therefore have large count fields that would not overflow very often. A 4-byte count field overflows at $2^{32} - 1 \approx 4.3 \times 10^9$.

It should be noted that after rescaling the counts, the new symbols being read and compressed have more effect on the counts than the old symbols (those counted before the rescaling). This turns out to be fortuitous since it is known from experience that the probability of appearance of a symbol depends more on the symbols immediately preceding it than on symbols that appeared in the distant past.

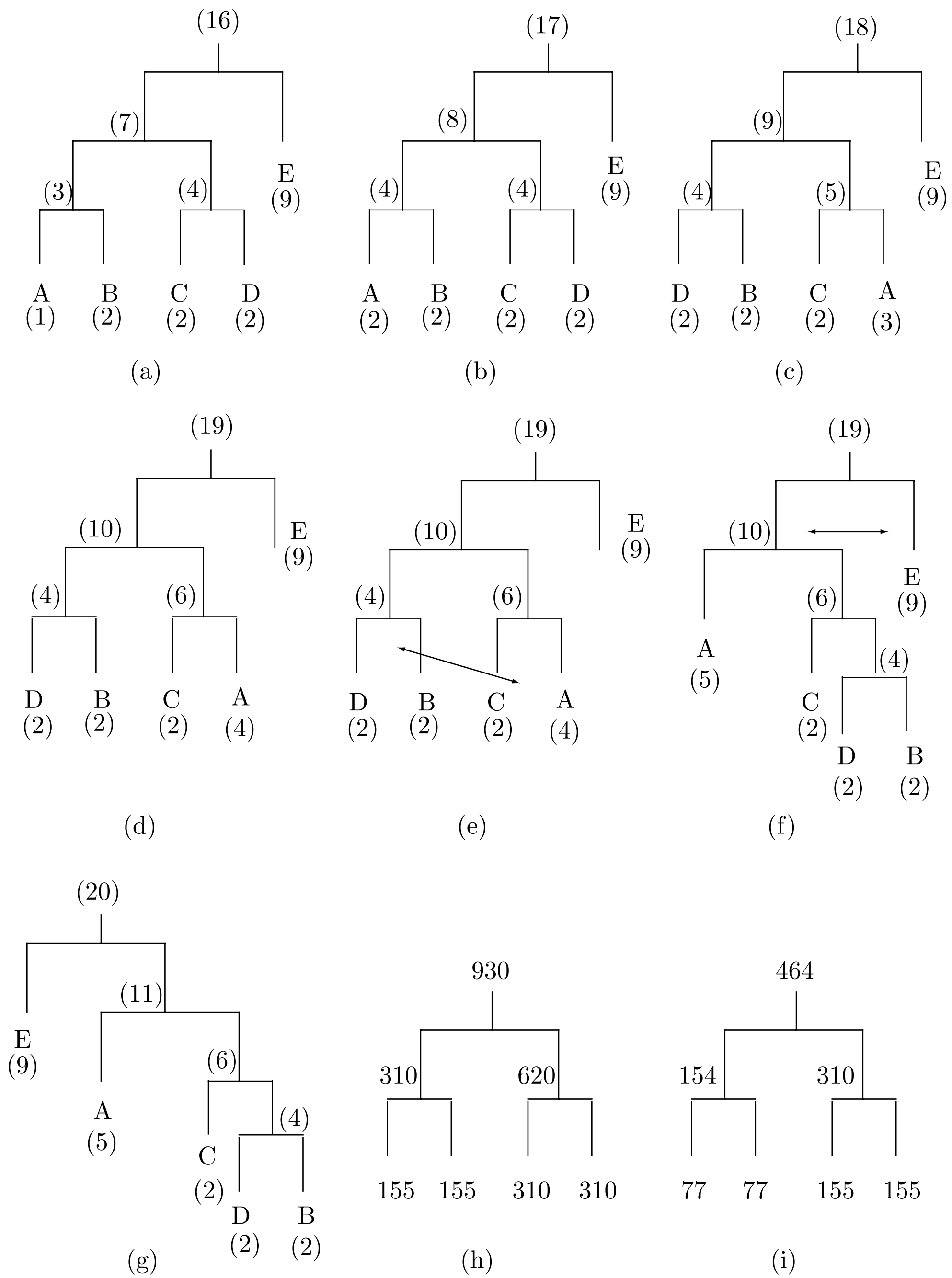



Figure 2.18: Updating the Huffman Tree.

## 2.3.4 Code Overflow

An even more serious problem is code overflow. This may happen when many symbols are added to the tree, and it becomes tall. The codes themselves are not stored in the tree, since they change all the time, and the compressor has to figure out the code of a symbol $X$ each time $X$ is input. Here are the details of this process:

1. The encoder has to locate symbol $X$ in the tree. The tree has to be implemented as an array of structures, each a node, and the array is searched linearly.
2. If $X$ is not found, the escape code is emitted, followed by the uncompressed code of $X$. $X$ is then added to the tree.
3. If $X$ is found, the compressor moves from node $X$ back to the root, building the code bit by bit as it goes along. Each time it goes from a left child to a parent, a "1" is appended to the code. Going from a right child to a parent appends a "0" bit to the code (or vice versa, but this should be consistent because it is mirrored by the decoder). Those bits have to be accumulated someplace, since they have to be emitted in the *reverse order* in which they are created. When the tree gets taller, the codes get longer. If they are accumulated in a 16-bit integer, then codes longer than 16 bits would cause a malfunction.

One solution to the code overflow problem is to accumulate the bits of a code in a linked list, where new nodes can be created, limited in number only by the amount of available memory. This is general but slow. Another solution is to accumulate the codes in a large integer variable (perhaps 50 bits wide) and document a maximum code size of 50 bits as one of the limitations of the program.

Fortunately, this problem does not affect the decoding process. The decoder reads the compressed code bit by bit and uses each bit to move one step left or right down the tree until it reaches a leaf node. If the leaf is the escape code, the decoder reads the uncompressed code of the symbol off the compressed data (and adds the symbol to the tree). Otherwise, the uncompressed code is found in the leaf node.

◇ **Exercise 2.10:** Given the 11-symbol string `sir␣sid␣is`, apply the adaptive Huffman method to it. For each symbol input, show the output, the tree after the symbol has been added to it, the tree after being rearranged (if necessary), and the list of nodes traversed left to right and bottom up.

## 2.3.5 A Variant

This variant of the adaptive Huffman method is simpler but less efficient. The idea is to calculate a set of $n$ variable-length codes based on equal probabilities, to assign those codes to the $n$ symbols at random, and to change the assignments "on the fly," as symbols are being read and compressed. The method is inefficient because the codes are not based on the actual probabilities of the symbols in the input. However, it is simpler to implement and also faster than the adaptive method described earlier, because it has to swap rows in a table, rather than update a tree, when updating the frequencies of the symbols.

The main data structure is an $n \times 3$ table where the three columns store the names of the $n$ symbols, their frequencies of occurrence so far, and their codes. The table is always kept sorted by the second column. When the frequency counts in the second

| Name | Count | Code | Name | Count | Code | Name | Count | Code | Name | Count | Code |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $a_1$ | 0 | 0 | $a_2$ | 1 | 0 | $a_2$ | 1 | 0 | $a_4$ | 2 | 0 |
| $a_2$ | 0 | 10 | $a_1$ | 0 | 10 | $a_4$ | 1 | 10 | $a_2$ | 1 | 10 |
| $a_3$ | 0 | 110 | $a_3$ | 0 | 110 | $a_3$ | 0 | 110 | $a_3$ | 0 | 110 |
| $a_4$ | 0 | 111 | $a_4$ | 0 | 111 | $a_1$ | 0 | 111 | $a_1$ | 0 | 111 |
| | (a) | | | (b) | | | (c) | | | (d) | |

Table 2.19: Four Steps in a Huffman Variant.

column change, rows are swapped, but only columns 1 and 2 are moved. The codes in column 3 never change. Table 2.19 shows an example of four symbols and the behavior of the method when the string $a_2$, $a_4$, $a_4$ is compressed.

Table 2.19a shows the initial state. After the first symbol $a_2$ is read, its count is incremented, and since it is now the largest count, rows 1 and 2 are swapped (Table 2.19b). After the second symbol $a_4$ is read, its count is incremented and rows 2 and 4 are swapped (Table 2.19c). Finally, after reading the last symbol $a_4$, its count is the largest, so rows 1 and 2 are swapped (Table 2.19d).

The only point that can cause a problem with this method is overflow of the count fields. If such a field is $k$ bits wide, its maximum value is $2^k - 1$, so it will overflow when incremented for the $2^k$th time. This may happen if the size of the input file is not known in advance, which is very common. Fortunately, we do not really need to know the counts, we just need them in sorted order, which makes it easy to solve this problem.

One solution is to count the input symbols and, after $2^k - 1$ symbols are input and compressed, to (integer) divide all the count fields by 2 (or shift them one position to the right, if this is easier).

Another, similar, solution is to check each count field every time it is incremented, and if it has reached its maximum value (if it consists of all ones), to integer divide all the count fields by 2, as mentioned earlier. This approach requires fewer divisions but more complex tests.

Naturally, whatever solution is adopted should be used by both the compressor and decompressor.

## 2.3.6 Vitter's Method

An improvement of the original algorithm, due to [Vitter 87], which also includes extensive analysis is based on the following key ideas:

1. A different scheme should be used to number the nodes in the dynamic Huffman tree. It is called *implicit numbering*, and it numbers the nodes from the bottom up and in each level from left to right.

2. The Huffman tree should be updated in such a way that the following will always be satisfied. For each weight $w$, all leaves of weight $w$ precede (in the sense of implicit numbering) all the internal nodes of the same weight. This is an *invariant*.

These ideas result in the following benefits:

1. In the original algorithm, it is possible that a rearrangement of the tree would move a node down one level. In the improved version, this does not happen.

2. Each time the Huffman tree is updated in the original algorithm, some nodes may be moved up. In the improved version, at most one node has to be moved up.

3. The Huffman tree in the improved version minimizes the sum of distances from the root to the leaves and also has the minimum height.

A special data structure, called a floating tree, is proposed to make it easy to maintain the required invariant. It can be shown that this version performs much better than the original algorithm. Specifically, if a two-pass Huffman method compresses an input file of $n$ symbols to $S$ bits, then the original adaptive Huffman algorithm can compress it to at most $2S+n$ bits, whereas the improved version can compress it down to $S+n$ bits—a significant difference! Notice that these results do not depend on the size of the alphabet, only on the size $n$ of the data being compressed and on its nature (which determines $S$).

> "I think you're begging the question," said Haydock, "and I can see looming ahead one of those terrible exercises in probability where six men have white hats and six men have black hats and you have to work it out by mathematics how likely it is that the hats will get mixed up and in what proportion. If you start thinking about things like that, you would go round the bend. Let me assure you of that!"
>
> —Agatha Christie, *The Mirror Crack'd*

## Intermezzo

**History of Fax**. Fax machines have been popular since the mid-1980s, so it is natural to assume that this is new technology. In fact, the first fax machine was invented in 1843, by Alexander Bain, a Scottish clock and instrument maker and all-round inventor. Among his many other achievements, Bain also invented the first electrical clock (powered by an electromagnet propelling a pendulum), developed chemical telegraph receivers and punch-tapes for fast telegraph transmissions, and installed the first telegraph line between Edinburgh and Glasgow.

The patent for the fax machine (grandly titled "improvements in producing and regulating electric currents and improvements in timepieces and in electric printing and signal telegraphs") was granted to Bain on May 27, 1843, 33 years before a similar patent (for the telephone) was given to Alexander Graham Bell.

Bain's fax machine transmitter scanned a flat, electrically conductive metal surface with a stylus mounted on a pendulum that was powered by an electromagnet. The stylus picked up writing from the surface and sent it through a wire to the stylus of the receiver, where the image was reproduced on a similar electrically conductive metal surface. Reference [hffax 07] has a figure of this apparatus.

Unfortunately, Bain's invention was not very practical and did not catch on, which is easily proved by the well-known fact that Queen Victoria never actually said "I'll drop you a fax."

In 1850, Frederick Bakewell, a London inventor, made several improvements on Bain's design. He built a device that he called a copying telegraph, and received a patent on it. Bakewell demonstrated his machine at the 1851 Great Exhibition in London.

In 1862, Italian physicist Giovanni Caselli built a fax machine (the pantelegraph), that was based on Bain's invention and also included a synchronizing apparatus. It was more successful than Bain's device and was used by the French Post and Telegraph agency between Paris and Lyon from 1856 to 1870. Even the Emperor of China heard about the pantelegraph and sent officials to Paris to study the device. The Chinese immediately recognized the advantages of facsimile for Chinese text, which was impossible to handle by telegraph because of its thousands of ideograms. Unfortunately, the negotiations between Peking and Caselli failed.

Elisha Gray, arguably the best example of the quintessential loser, invented the telephone, but is virtually unknown today because he was beaten by Alexander Graham Bell, who arrived at the patent office a few hours before Gray on the fateful day of March 7, 1876. Born in Barnesville, Ohio, Gray invented and patented many electrical devices, including a facsimile apparatus. He also founded what later became the Western Electric Company.

Ernest A. Hummel, a watchmaker from St. Paul, Minnesota, invented, in 1895 a device he dubbed a copying telegraph, or telediagraph. This machine was based on synchronized rotating drums, with a platinum stylus as an electrode in the transmitter. It was used by the New York Herald to send pictures via telegraph lines. An improved version (in 1899) was sold to several newspapers (the Chicago Times Herald, the St. Louis Republic, the Boston Herald, and the Philadelphia Inquirer) and it, as well as other, very similar machines, were in use to transmit newsworthy images until the 1970s.

A practical fax machine (perhaps the first practical one) was invented in 1902 by Arthur Korn in Germany. This was a photoelectric device and it was used to transmit photographs in Germany from 1907.

In 1924, Richard H. Ranger, a designer for the Radio Corporation of America (RCA), invented the wireless photoradiogram, or transoceanic radio facsimile. This machine can be considered the true forerunner of today's fax machines. On November 29, 1924, a photograph of the American President Calvin Coolidge that was sent from New York to London became the first image reproduced by transoceanic wireless facsimile.

The next step was the belinograph, invented in 1925 by the French engineer Edouard Belin. An image was placed on a drum and scanned with a powerful beam of light. The reflection was converted to an analog voltage by a photoelectric cell. The voltage was sent to a receiver, where it was converted into mechanical movement of a pen to reproduce the image on a blank sheet of paper on an identical drum rotating at the same speed. The fax machines we all use are still based on the principle of scanning a document with light, but they are controlled by a microprocessor and have a small number of moving parts.

In 1924, the American Telephone & Telegraph Company (AT&T) decided to improve telephone fax technology. The result of this effort was a telephotography machine that was used to send newsworthy photographs long distance for quick newspaper publication.

By 1926, RCA invented the Radiophoto, a fax machine based on radio transmissions.

The Hellschreiber was invented in 1929 by Rudolf Hell, a pioneer in mechanical image scanning and transmission. During WW2, it was sometimes used by the German military in conjunction with the Enigma encryption machine.

In 1947, Alexander Muirhead invented a very successful fax machine.

On March 4, 1955, the first radio fax transmission was sent across the continent.

Fax machines based on optical scanning of a document were developed over the years, but the spark that ignited the fax revolution was the development, in 1983, of the Group 3 CCITT standard for sending faxes at rates of 9,600 bps.

More history and pictures of many early fax and telegraph machines can be found at [hffax 07] and [technikum29 07].

## 2.4 Facsimile Compression

Data compression is especially important when images are transmitted over a communications line because a person is often waiting at the receiving end, eager to see something quickly. Documents transferred between fax machines are sent as bitmaps, so a standard compression algorithm was needed when those machines became popular. Several methods were developed and proposed by the ITU-T.

The ITU-T is one of four permanent parts of the International Telecommunications Union (ITU), based in Geneva, Switzerland (`http://www.itu.ch/`). It issues recommendations for standards applying to modems, packet switched interfaces, V.24 connectors, and similar devices. Although it has no power of enforcement, the standards it recommends are generally accepted and adopted by industry. Until March 1993, the ITU-T was known as the Consultative Committee for International Telephone and Telegraph (Comité Consultatif International Télégraphique et Téléphonique, or CCITT).

CCITT: Can't Conceive Intelligent Thoughts Today

The first data compression standards developed by the ITU-T were T2 (also known as Group 1) and T3 (Group 2). These are now obsolete and have been replaced by T4 (Group 3) and T6 (Group 4). Group 3 is currently used by all fax machines designed to operate with the Public Switched Telephone Network (PSTN). These are the machines we have at home, and at the time of writing, they operate at maximum speeds of 9,600 baud. Group 4 is used by fax machines designed to operate on a digital network, such as ISDN. They have typical speeds of 64K bits/sec (baud). Both methods can produce compression factors of 10 or better, reducing the transmission time of a typical page to about a minute with the former, and a few seconds with the latter.

**One-dimensional coding**. A fax machine scans a document line by line, converting each scan line to many small black and white dots called *pels* (from Picture ELement). The horizontal resolution is always 8.05 pels per millimeter (about 205 pels per inch). An 8.5-inch-wide scan line is therefore converted to 1728 pels. The T4 standard, though, recommends to scan only about 8.2 inches, thereby producing 1664 pels per scan line (these numbers, as well as those in the next paragraph, are all to within $\pm 1\%$ accuracy).

The word facsimile comes from the Latin *facere* (make) and *similis* (like).

The vertical resolution is either 3.85 scan lines per millimeter (standard mode) or 7.7 lines/mm (fine mode). Many fax machines have also a very-fine mode, where they scan 15.4 lines/mm. Table 2.20 assumes a 10-inch-high page (254 mm), and shows the total number of pels per page, and typical transmission times for the three modes without compression. The times are long, illustrating the importance of compression in fax transmissions.

| Scan lines | Pels per line | Pels per page | Time (sec.) | Time (min.) |
|---|---|---|---|---|
| 978 | 1664 | 1.670M | 170 | 2.82 |
| 1956 | 1664 | 3.255M | 339 | 5.65 |
| 3912 | 1664 | 6.510M | 678 | 11.3 |

Ten inches equal 254 mm. The number of pels is in the millions, and the transmission times, at 9600 baud without compression, are between 3 and 11 minutes, depending on the mode. However, if the page is shorter than 10 inches, or if most of it is white, the compression factor can be 10 or better, resulting in transmission times of between 17 and 68 seconds.

Table 2.20: Fax Transmission Times.

To derive the Group 3 code, the committee appointed by the ITU-T counted all the run lengths of white and black pels in a set of eight "training" documents that they felt represent typical text and images sent by fax, and then applied the Huffman algorithm to construct a variable-length code and assign codewords to all the run length. (The eight documents are described in Table 2.21 and can be found at [funet 07].) The most common run lengths were found to be 2, 3, and 4 black pixels, so they were assigned the shortest codes (Table 2.22). Next come run lengths of 2–7 white pixels, which were assigned slightly longer codes. Most run lengths were rare and were assigned long, 12-bit codes. Thus, Group 3 uses a combination of RLE and Huffman coding.

| Image | Description |
|---|---|
| 1 | Typed business letter (English) |
| 2 | Circuit diagram (hand drawn) |
| 3 | Printed and typed invoice (French) |
| 4 | Densely typed report (French) |
| 5 | Printed technical article including figures and equations (French) |
| 6 | Graph with printed captions (French) |
| 7 | Dense document (Kanji) |
| 8 | Handwritten memo with very large white-on-black letters (English) |

Table 2.21: The Eight CCITT Training Documents.

◇ **Exercise 2.11:** A run length of 1,664 white pels was assigned the short code 011000. Why is this length so common?

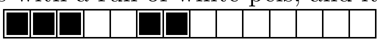
Since run lengths can be long, the Huffman algorithm was modified. Codes were assigned to run lengths of 1 to 63 pels (they are the termination codes in Table 2.22a) and to run lengths that are multiples of 64 pels (the make-up codes in Table 2.22b). Group 3 is therefore a *modified Huffman code* (also called MH). The code of a run length is either a single termination code (if the run length is short) or one or more make-up codes, followed by one termination code (if it is long). Here are some examples:

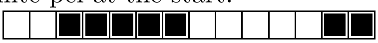
1. A run length of 12 white pels is coded as 001000.
2. A run length of 76 white pels ($= 64 + 12$) is coded as 11011|001000.
3. A run length of 140 white pels ($= 128 + 12$) is coded as 10010|001000.
4. A run length of 64 black pels ($= 64 + 0$) is coded as 0000001111|0000110111.
5. A run length of 2,561 black pels ($2560 + 1$) is coded as 000000011111|010.

◇ **Exercise 2.12:** There are no runs of length zero. Why then were codes assigned to runs of zero black and zero white pels?

◇ **Exercise 2.13:** An 8.5-inch-wide scan line results in 1,728 pels, so how can there be a run of 2,561 consecutive pels?

Each scan line is coded separately, and its code is terminated by the special 12-bit EOL code 000000000001. Each line also gets one white pel appended to it on the left when it is scanned. This is done to remove any ambiguity when the line is decoded on the receiving end. After reading the EOL for the previous line, the receiver assumes that the new line starts with a run of white pels, and it ignores the first of them. Examples:

1. The 14-pel line  is coded as the run lengths 1w 3b 2w 2b 7w EOL, which become 000111|10|0111|11|1111|000000000001. The decoder ignores the single white pel at the start.

2. The line  is coded as the run lengths 3w 5b 5w 2b EOL, which becomes the binary string 1000|0011|1100|11|000000000001.

◇ **Exercise 2.14:** The Group 3 code for a run length of five black pels (0011) is also the prefix of the codes for run lengths of 61, 62, and 63 white pels. Explain this.

> In computing, a newline (also known as a line break or end-of-line / EOL character) is a special character or sequence of characters signifying the end of a line of text. The name comes from the fact that the next character after the newline will appear on a new line—that is, on the next line below the text immediately preceding the newline. The actual codes representing a newline vary across hardware platforms and operating systems, which can be a problem when exchanging data between systems with different representations.
>
> —From `http://en.wikipedia.org/wiki/End-of-line`

The Group 3 code has no error correction, but many errors can be detected. Because of the nature of the Huffman code, even one bad bit in the transmission can cause the receiver to get out of synchronization, and to produce a string of wrong pels. This is why each scan line is encoded separately. If the receiver detects an error, it skips

(a)

| Run length | White codeword | Black codeword | Run length | White codeword | Black codeword |
|---|---|---|---|---|---|
| 0 | 00110101 | 0000110111 | 32 | 00011011 | 000001101010 |
| 1 | 000111 | 010 | 33 | 00010010 | 000001101011 |
| 2 | 0111 | 11 | 34 | 00010011 | 000011010010 |
| 3 | 1000 | 10 | 35 | 00010100 | 000011010011 |
| 4 | 1011 | 011 | 36 | 00010101 | 000011010100 |
| 5 | 1100 | 0011 | 37 | 00010110 | 000011010101 |
| 6 | 1110 | 0010 | 38 | 00010111 | 000011010110 |
| 7 | 1111 | 00011 | 39 | 00101000 | 000011010111 |
| 8 | 10011 | 000101 | 40 | 00101001 | 000001101100 |
| 9 | 10100 | 000100 | 41 | 00101010 | 000001101101 |
| 10 | 00111 | 0000100 | 42 | 00101011 | 000011011010 |
| 11 | 01000 | 0000101 | 43 | 00101100 | 000011011011 |
| 12 | 001000 | 0000111 | 44 | 00101101 | 000001010100 |
| 13 | 000011 | 00000100 | 45 | 00000100 | 000001010101 |
| 14 | 110100 | 00000111 | 46 | 00000101 | 000001010110 |
| 15 | 110101 | 000011000 | 47 | 00001010 | 000001010111 |
| 16 | 101010 | 0000010111 | 48 | 00001011 | 000001100100 |
| 17 | 101011 | 0000011000 | 49 | 01010010 | 000001100101 |
| 18 | 0100111 | 0000001000 | 50 | 01010011 | 000001010010 |
| 19 | 0001100 | 00001100111 | 51 | 01010100 | 000001010011 |
| 20 | 0001000 | 00001101000 | 52 | 01010101 | 000000100100 |
| 21 | 0010111 | 00001101100 | 53 | 00100100 | 000000110111 |
| 22 | 0000011 | 00000110111 | 54 | 00100101 | 000000111000 |
| 23 | 0000100 | 00000101000 | 55 | 01011000 | 000000100111 |
| 24 | 0101000 | 00000010111 | 56 | 01011001 | 000000101000 |
| 25 | 0101011 | 00000011000 | 57 | 01011010 | 000001011000 |
| 26 | 0010011 | 000011001010 | 58 | 01011011 | 000001011001 |
| 27 | 0100100 | 000011001011 | 59 | 01001010 | 000000101011 |
| 28 | 0011000 | 000011001100 | 60 | 01001011 | 000000101100 |
| 29 | 00000010 | 000011001101 | 61 | 00110010 | 000001011010 |
| 30 | 00000011 | 000001101000 | 62 | 00110011 | 000001100110 |
| 31 | 00011010 | 000001101001 | 63 | 00110100 | 000001100111 |

(b)

| Run length | White codeword | Black codeword | Run length | White codeword | Black codeword |
|---|---|---|---|---|---|
| 64 | 11011 | 0000001111 | 1344 | 011011010 | 0000001010011 |
| 128 | 10010 | 000011001000 | 1408 | 011011011 | 0000001010100 |
| 192 | 010111 | 000011001001 | 1472 | 010011000 | 0000001010101 |
| 256 | 0110111 | 000001011011 | 1536 | 010011001 | 0000001011010 |
| 320 | 00110110 | 000000110011 | 1600 | 010011010 | 0000001011011 |
| 384 | 00110111 | 000000110100 | 1664 | 011000 | 0000001100100 |
| 448 | 01100100 | 000000110101 | 1728 | 010011011 | 0000001100101 |
| 512 | 01100101 | 0000001101100 | 1792 | 00000001000 | same as |
| 576 | 01101000 | 0000001101101 | 1856 | 00000001100 | white |
| 640 | 01100111 | 0000001001010 | 1920 | 00000001101 | from this |
| 704 | 011001100 | 0000001001011 | 1984 | 000000010010 | point |
| 768 | 011001101 | 0000001001100 | 2048 | 000000010011 | |
| 832 | 011010010 | 0000001001101 | 2112 | 000000010100 | |
| 896 | 011010011 | 0000001110010 | 2176 | 000000010101 | |
| 960 | 011010100 | 0000001110011 | 2240 | 000000010110 | |
| 1024 | 011010101 | 0000001110100 | 2304 | 000000010111 | |
| 1088 | 011010110 | 0000001110101 | 2368 | 000000011100 | |
| 1152 | 011010111 | 0000001110110 | 2432 | 000000011101 | |
| 1216 | 011011000 | 0000001110111 | 2496 | 000000011110 | |
| 1280 | 011011001 | 0000001010010 | 2560 | 000000011111 | |

Table 2.22: Group 3 and 4 Fax Codes: (a) Termination Codes, (b) Make-Up Codes.

bits, looking for an EOL. This way, one error can cause at most one scan line to be received incorrectly. If the receiver does not see an EOL after a certain number of lines, it assumes a high error rate, and it aborts the process, notifying the transmitter. Since the codes are between two and 12 bits long, the receiver detects an error if it cannot decode a valid code after reading 12 bits.

Each page of the coded document is preceded by one EOL and is followed by six EOL codes. Because each line is coded separately, this method is a *one-dimensional coding* scheme. The compression ratio depends on the image. Images with large contiguous black or white areas (text or black and white images) can be highly compressed. Images with many short runs can sometimes produce negative compression. This is especially true in the case of images with shades of gray (such as scanned photographs). Such shades are produced by halftoning, which covers areas with many alternating black and white pels (runs of length 1).

◇ **Exercise 2.15:** What is the compression ratio for runs of length one (i.e., strictly alternating pels)?

The T4 standard also allows for fill bits to be inserted between the data bits and the EOL. This is done in cases where a pause is necessary, or where the total number of bits transmitted for a scan line must be a multiple of 8. The fill bits are zeros.

Example: The binary string 000111|10|0111|11|1111|000000000001 becomes 000111|10|0111|11|1111|00|0000000001 after two zeros are added as fill bits, bringing the total length of the string to 32 bits ($= 8 \times 4$). The decoder sees the two zeros of the fill, followed by the 11 zeros of the EOL, followed by the single 1, so it knows that it has encountered a fill followed by an EOL.

**Two-dimensional coding**. This variant was developed because one-dimensional coding produces poor results for images with gray areas. Two-dimensional coding is optional on fax machines that use Group 3 but is the only method used by machines intended to work in a digital network. When a fax machine using Group 3 supports two-dimensional coding as an option, each EOL is followed by one extra bit, to indicate the compression method used for the next scan line. That bit is 1 if the next line is encoded with one-dimensional coding, and 0 if it is encoded with two-dimensional coding.

The two-dimensional coding method is also called MMR, for *modified modified READ*, where READ stands for *relative element address designate*. The term "modified modified" is used because this is a modification of one-dimensional coding, which itself is a modification of the original Huffman method. The two-dimensional coding method is described in detail in [Salomon 07] and other references, but here are its main principles. The method compares the current scan line (called the *coding line*) to its predecessor (referred to as the *reference line*) and records the differences between them, the assumption being that two consecutive lines in a document will normally differ by just a few pels. The method assumes that there is an all-white line above the page, which is used as the reference line for the first scan line of the page. After coding the first line, it becomes the reference line, and the second scan line is coded. As in one-dimensional coding, each line is assumed to start with a white pel, which is ignored by the receiver.

The two-dimensional coding method is more prone to errors than one-dimensional coding, because any error in decoding a line will cause errors in decoding all its successors and will propagate throughout the entire document. This is why the T.4 (Group 3) standard includes a requirement that after a line is encoded with the one-dimensional method, at most $K - 1$ lines will be encoded with the two-dimensional coding method. For standard resolution $K = 2$, and for fine resolution $K = 4$. The T.6 standard (Group 4) does not have this requirement, and uses two-dimensional coding exclusively.

# Chapter Summary

Huffman coding is one of the basic techniques of data compression. It is also fast, conceptually simple, and easy to implement. The Huffman encoding algorithm starts with a set of symbols whose probabilities are known and constructs a code tree. Once the tree is complete, it is used to determine the variable-length, prefix codewords for the individual symbols. Each leaf of the tree corresponds to a data symbol and the prefix code of a symbol $S$ is determined by sliding down the tree from the root to the leaf that corresponds to $S$. It can be shown that these codewords are the best possible in the sense that they feature the shortest average length. However, the codewords are not unique and there are several (perhaps even many) different sets of codewords that have the same average length. Huffman decoding starts by reading bits from the compressed file and using them to slide down the tree from node to node until a leaf (and thus a data symbol) is reached. Section 2.2.1 describes an interesting method for fast decoding.

The Huffman method requires knowledge of the symbols' probabilities, but in practice, these are not always known in advance. This chapter lists the following methods for handling this problem.

- Use a set of training documents. The implementor of a Huffman codec (compressor/decompressor) selects a set of documents that are judged typical or average. The documents are analyzed once, counting the number of occurrences (and hence also the probability) of each data symbol. Based on these probabilities, the implementor constructs a Huffman code (a set of codewords for all the symbols in the alphabet) and hard-codes this code in both encoder and decoder. Such a code may not conform to the symbols' probabilities of any particular input file that's being compressed, and so does not produce the best compression, but this approach is simple and fast. The compression method used by fax machines (Section 2.4) is based on this approach.

- A two-pass compression job produces the ideal codewords for the input file, but is slow. In this approach, the input file is read twice. In the first pass, the encoder counts symbol occurrences. Between the passes, the encoder uses this information to compute symbol probabilities and constructs a set of Huffman codewords for the particular data being compressed. In the second pass the encoder actually compresses the data by replacing each data symbol with its Huffman codeword.

- An adaptive compression algorithm achieves the best of both worlds, being both effective and fast, but is more difficult to implement. The principle is to start with an empty Huffman code tree and to update the tree as input symbols are read and processed. When a symbol is input, the tree is searched for it. If the symbol is in the tree, its codeword is used; otherwise, it is added to the tree and a new codeword is assigned to it. In either case, the tree is examined and may have to be rearranged to keep it a Huffman code tree. This process has to be designed carefully, to make sure that the decoder can perform it in the same way as the encoder (in lockstep). Such an adaptive algorithm is discussed in Section 2.3.

The Huffman method is simple, fast, and produces excellent results, but is not as effective as arithmetic coding (Chapter 4). The conscientious reader may benefit from the discussion in [Bookstein and Klein 93], where the authors argue in favor of Huffman coding.

### Self-Assessment Questions

1. In a programming language of your choice, implement Huffman encoding and test it on the five symbols of Figure 2.1.

2. Complete the decoding example in the second paragraph of Section 2.2.1.

3. The fax compression standard of Section 2.4 is based on eight training documents selected by the CCITT (the predecessor of the ITU-T). Select your own set of eight training documents (black and white images on paper) and scan them at 200 dpi to determine the frequencies of occurrence of all the runs of black and white pels. Sort the runs in ascending order and compare their probabilities to the lengths of the codes of Table 2.22 (your most-common runs should correspond to the shortest codes of this table).

> The novelty of waking up to a fax machine next to your bed going off in the middle of the night with an order from Japan wears off.
>
> —Naomi Bradshaw

http://www.springer.com/978-1-84800-071-1

A Concise Introduction to Data Compression
Salomon, D.
2008, XIV, 314 p. 89 illus. With online files/update.,
Softcover
ISBN: 978-1-84800-071-1