Chapter 1

Text Compression

1.1 Introduction

Data compression is useful and necessary in a variety of applications. These applications can be broadly divided into two groups: transmission and storage. Transmission involves sending a file, from a *sender* to a *receiver*, over a channel. Compression reduces the number of bits to be transmitted, thus making the transmission process more efficient. Storing a file in a compressed form typically requires fewer bits, thus utilizing storage resources (including main memory itself) more efficiently.

Data compression can be applied to any kind of data: text, image (such as fax), audio and video. A 1-second video without compression takes around 20 megabytes (i.e., 170 megabits) and a 2-minute CD-quality uncompressed music (44,100 samples per second with 16 bits per sample) requires more than 84 megabits. Impressive gains can be made by compressing video, for instance, because successive frames are very similar to each other in their contents. In fact, real-time video transmission would be impossible without considerable compression. There are several new applications that generate data at prodigious rates; certain earth orbiting satellites create around half a terabyte (10^{12}) of data per day. Without compression there is no hope of storing such large files in spite of the impressive advances made in storage technologies.

Lossy and Lossless Compression Most data types, except text, are compressed in such a way that a very good approximation, but not the exact content, of the original file can be recovered by the receiver. For instance, even though the human voice can range up to 20kHz in frequency, telephone transmissions retain only up to about 5kHz.[1] The voice that is reproduced at the receiver's end is a close approximation to the real thing, but it is not exact. Try lis-

[1] A famous theorem, known as the *sampling theorem*, states that the signal must be sampled at twice this rate, i.e., around 10,000 times a second. Typically, 8 to 16 bits are produced for each point in the sample.

tening to your favorite CD played over a telephone line. Video transmissions often sacrifice quality for speed of transmission. The type of compression in such situations is called *lossy*, because the receiver cannot exactly reproduce the original contents. For analog signals, all transmissions are lossy; the degree of loss determines the quality of transmission.

Text transmissions are required to be *lossless*. It will be a disaster to change even a single symbol in a text file.[2] In this note, we study several lossless compression schemes for text files. Henceforth, we use the terms *string* and *text file* synonymously.

Error detection and correction can be applied to uncompressed as well as compressed strings. Typically, a string to be transmitted is first compressed and then encoded for errors. At the receiver's end, the received string is first decoded (error detection and correction are applied to recover the compressed string), and then the string is decompressed.

What is the typical level of compression? The amount by which a text string can be compressed depends on the string itself. A repetitive lyric like "Old McDonald had a farm" can be compressed significantly, by transmitting a single instance of a phrase that is repeated.[3] I compressed a postscript file of 2,144,364 symbols to 688,529 symbols using a standard compression algorithm, `gzip`; so, the compressed file is around 32% of the original in length. I found a web site[4] where *The Adventures of Tom Sawyer*, by Mark Twain, is in uncompressed form at 391 Kbytes and compressed form (in zip format) at 172 Kbytes; the compressed file is around 44% of the original.

1.2 A Very Incomplete Introduction to Information Theory

Take a random string of symbols over a given alphabet; imagine that there is a source that spews out these symbols following some probability distribution over the alphabet. If all symbols of the alphabet are equally probable, then you can't do any compression at all. However, if the probabilities of different symbols are non-identical —say, over a binary alphabet "0" occurs with 90% frequency and "1" with 10%— you may get significant compression. This is because you are likely to see runs of zeros more often, and you may encode such runs using short bit strings. A possible encoding, using 2-bit blocks, is: 00 for 0, 01 for 1, 10 for 00 and 11 for 000. We are likely to see a large number of "000" strings which would be compressed by one bit, whereas for encoding "1" we lose a bit.

[2]There are exceptions to this rule. In some cases it may not matter to the receiver if extra white spaces are squeezed out, or the text is formatted slightly differently.

[3]Knuth [30] gives a delightful treatment of a number of popular songs in this vein.

[4]`http://www.ibiblio.org/gutenberg/cgi-bin/sdb/t9.cgi/t9.cgi?entry=74 &full=yes&ftpsite=http://www.ibiblio.org/gutenberg/`

In 1948, Claude E. Shannon [45] published "A Mathematical Theory of Communication", in which he presented the concept of *entropy*, which gives a quantitative measure of the compression that is possible. I give below an extremely incomplete treatment of this work.

Consider a finite alphabet; it may be binary, the Roman alphabet, all the symbols on your keyboard, or any other finite set. A random source outputs a string of symbols from this alphabet; it has probability p_i of producing the ith symbol. Productions of successive symbols are independent, that is, for its next output, the source selects a symbol with the given probabilities independent of what it has produced already. The *entropy*, h, of the alphabet is given by

$$h \;=\; -\textstyle\sum_i p_i \,(\log p_i)$$

where log stands for logarithm to base 2. (We use the convention that $0 \log 0 = 0$.) Shannon showed that for lossless transmission of a (long) string of n symbols, you need at least nh bits, i.e., h bits on the average to encode each symbol of the alphabet. And, it is possible to transmit at this rate!

Figure 1.1 below shows the entropy function for an alphabet of size 2 where the probabilities of the two symbols are p and $(1-p)$. Note that the curve is symmetric in p and $(1-p)$, and its highest value, 1.0, is achieved when both symbols are equiprobable.

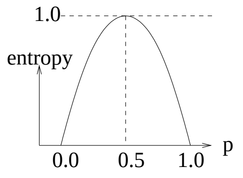

Figure 1.1: Entropy function: $-p \log p - (1-p) \log (1-p)$

Let us compute a few values of the entropy function. Suppose we have the binary alphabet where the two symbols are equiprobable. Then, as is shown in Figure 1.1,

$$\begin{aligned} h &= -0.5 \times (\log 0.5) - 0.5 \times (\log 0.5) \\ &= -\log 0.5 \\ &= 1 \end{aligned}$$

That is, you need 1 bit on the average to encode each symbol, so you cannot compress such strings at all! Next, suppose the two symbols are not equiprobable; "0" occurs with probability 0.9 and "1" with 0.1. Then,

$$\begin{aligned} h &= -0.9 \times (\log 0.9) - 0.1 \times (\log 0.1) \\ &= 0.469 \end{aligned}$$

The text can be compressed to less than half its size. If the distribution is even more lop-sided, say 0.99 probability for "0" and 0.01 for "1", then $h = 0.080$; it is possible to compress the file to 8% of its size. Note that Shannon's theorem does not say how to achieve this bound; we will see some schemes that will asymptotically achive this bound.

Exercise 1
Show that for an alphabet of size m where all symbols are equally probable, the entropy is $\log m$. □

Next, consider English text. The source alphabet is usually defined as the 26 letters and the space character. There are then several models for entropy. The zero-order model assumes that the occurrence of each character is equally likely. Using the zero-order model, the entropy is $h = \log 27 = 4.75$. That is, a string of length n would have no less than $4.75 \times n$ bits.

The zero-order model does not accurately describe English texts: letters occur with different frequency. Six letters — 'e', 't', 'a', 'o', 'i', 'n'— occur over half the time; see Tables 1.1 and 1.2. Others occur rarely, such as 'q' and 'z'. In the first-order model, we assume that each symbol is statistically independent (that is, the symbols are produced independently) but we take into account the probability distribution. The first-order model is a better predictor of frequencies and it yields an entropy of 4.219 bits/symbol. For a source Roman alphabet that also includes the space character, a traditional value is 4.07 bits/symbol.

Letter	Frequency	Letter	Frequency	Letter	Frequency	Letter	Frequency
a	0.08167	b	0.01492	c	0.02782	d	0.04253
e	0.12702	f	0.02228	g	0.02015	h	0.06094
i	0.06966	j	0.00153	k	0.00772	l	0.04025
m	0.02406	n	0.06749	o	0.07507	p	0.01929
q	0.00095	r	0.05987	s	0.06327	t	0.09056
u	0.02758	v	0.00978	w	0.02360	x	0.00150
y	0.01974	z	0.00074				

Table 1.1: Frequencies of letters in English texts, alphabetic order

Higher order models take into account the statistical dependence among the letters, such as that 'q' is almost always followed by 'u', and that there is a high probability of getting an 'e' after an 'r'. A more accurate model of English yields lower entropy. The third-order model yields 2.77 bits/symbol. Estimates by Shannon [46] based on human experiments have yielded values as low as 0.6 to 1.3 bits/symbol.

Compression Techniques from Earlier Times Samuel Morse developed a code for telegraphic transmissions in which he encoded the letters using a binary alphabet, a dot ($\cdot$) and a dash ($-$). He assigned shorter codes to letters like 'e'($\cdot$) and 'a'($\cdot\,-$) that occur more often in texts, and longer codes to rarely-occurring letters, like 'q'($-\,-\,\cdot\,-$) and 'j'($\cdot\,-\,-\,-$).

Letter	Frequency	Letter	Frequency	Letter	Frequency	Letter	Frequency
e	0.12702	t	0.09056	a	0.08167	o	0.07507
i	0.06966	n	0.06749	s	0.06327	h	0.06094
r	0.05987	d	0.04253	l	0.04025	c	0.02782
u	0.02758	m	0.02406	w	0.02360	f	0.02228
g	0.02015	y	0.01974	p	0.01929	b	0.01492
v	0.00978	k	0.00772	j	0.00153	x	0.00150
q	0.00095	z	0.00074				

Table 1.2: Frequencies of letters in English texts, descending order

The Braille code, developed for use by the blind, uses a 2×3 matrix of dots where each dot is either flat or raised. The 6 dots provide $2^6 = 64$ possible combinations. After encoding all the letters, the remaining combinations are assigned to frequently occurring words, such as “and” and “for”.

1.3 Huffman Coding

We are given a set of symbols and the probability of occurrence of each symbol in some long piece of text. The symbols could be $\{0, 1\}$ with probability 0.9 for 0 and 0.1 for 1, Or, the symbols could be $\{a, c, g, t\}$ from a DNA sequence with appropriate probabilities, or Roman letters with the probabilities shown in Table 1.1. In many cases, particularly for text transmissions, we consider frequently occurring words —such as “in”, “for”, “to”— as symbols. The problem is to devise a *code*, a binary string for each symbol, so that (1) any encoded string can be decoded (i.e., the code is *uniquely decodable*, see below), and (2) the expected code length —probability of each symbol times the length of the code assigned to it, summed over all symbols— is minimized.

Example Let the symbols $\{a, c, g, t\}$ have the probabilities 0.05, 0.5, 0.4, 0.05 (in the given order). We show three different codes, C1, C2 and C3, and the associated expected code lengths in Table 1.3.

Symbol	Prob.	C1	avg. length	C2	avg. length	C3	avg. length
a	0.05	00	$0.05 \times 2 = 0.1$	00	$0.05 \times 2 = 0.1$	000	$0.05 \times 3 = 0.15$
c	0.5	0	$0.5 \times 1 = 0.5$	01	$0.5 \times 2 = 1.0$	1	$0.5 \times 1 = 0.5$
g	0.4	1	$0.4 \times 1 = 0.4$	10	$0.4 \times 2 = 0.8$	01	$0.4 \times 2 = 0.8$
t	0.05	11	$0.05 \times 2 = 0.1$	11	$0.05 \times 2 = 0.1$	001	$0.05 \times 3 = 0.15$
			exp. length = 1.1		2.0		1.6

Table 1.3: Three different codes for $\{a, c, g, t\}$

Code C1 is not uniquely decodable because *cc* and *a* will both be encoded by 00. Code C2 encodes each symbol by a 2 bit string; so, it is no surprise that the expected code length is 2.0 in this case. Code C3 has variable lengths for the codes. It can be shown that C3 is optimal, i.e., it has the minimum expected code length.

1.3.1 Uniquely Decodable Codes and Prefix Codes

We can get low expected code length by assigning short codewords to every symbol. If we have n symbols we need n distinct codewords. But that is not enough. As the example above shows, it may then be impossible to decode a piece of text unambiguously. A code is *uniquely decodable* if every string of symbols is encoded into a different string.

A *prefix code* is one in which no codeword is a prefix of another.[5] The codewords 000, 1, 01, 001 for $\{a, c, g, t\}$ constitute a prefix code. A prefix code is uniquely decodable: if two distinct strings are encoded identically, either their first symbols are identical (then, remove their first symbols, and repeat this step until they have distinct first symbols), or the codeword for one first symbol is a prefix of the other first symbol, contradicting that we have a prefix code.

It can be shown —but I will not show it in these notes— that there is an optimal uniquely decodable code which is a prefix code. Therefore, we can limit our attention to prefix codes only, which we do in the rest of this note.

A prefix code can be depicted by a labeled binary tree, as follows. Each leaf is labeled with a symbol (and its associated probability), a left edge by 0 and a right edge by 1. The codeword associated with a symbol is the sequence of bits on the path from the root to the corresponding leaf. See Figure 1.2 for a prefix code for $\{a, c, g, t\}$ which have associated probabilities of $0.05,\ 0.5,\ 0.4,\ 0.05$ (in the given order).

The length of a codeword is the corresponding pathlength. The *weighted pathlength* of a leaf is the probability associated with it times its pathlength. The *expected code length* is the sum of the weighted pathlengths over all leaves. Henceforth, the expected code length of a tree will be called its *weight*, and a tree is *best* if its weight is minimum. Note that there may be several best trees for the given probabilities.

Since the symbols themselves play no role —the probabilities identify the associated symbols— we dispense with the symbols and work with the probabilities only. Since the same probability may be associated with two different symbols, we have a bag, i.e., a multiset, of probabilities. Also, it is immaterial that the bag elements are probabilities; the algorithm applies to any bag of nonnegative numbers. We use the set notation for bags below.

Exercise 2

Try to construct the best tree for the following values $\{1,\ 2,\ 3,\ 4,\ 5,\ 7,\ 8\}$. The weight of the best tree is 78. □

[5]String s is a prefix of string t if $t = s +\!\!+\ x$, for some string x, where $+\!\!+$ denotes concatenation.

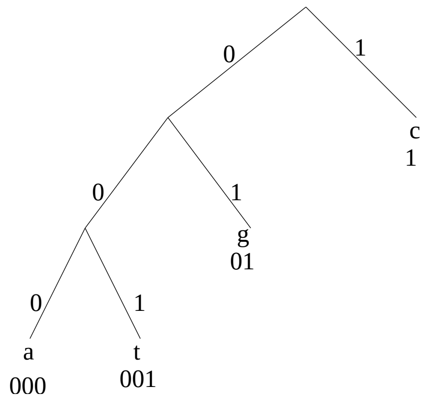

Figure 1.2: Prefix code for $\{a, c, g, t\}$

Remark: In a best tree, there is no dangling leaf; i.e., each leaf is labeled with a distinct symbol. Therefore, every internal node (i.e., nonleaf) has exactly two children. Such a tree is called a *full binary tree*.

Exercise 3

Show two possible best trees for the alphabet {0, 1, 2, 3, 4} with probabilities {0.2, 0.4, 0.2, 0.1, 0.1}. The trees should not be mere rearrangements of each other through reflections of subtrees. □

Solution One possible solution is shown below.

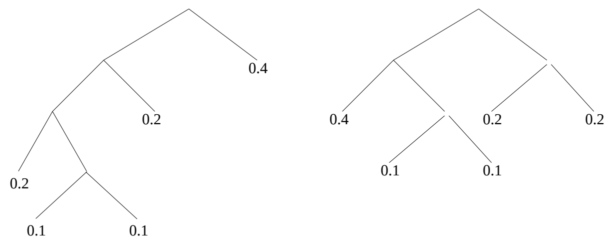

Figure 1.3: Two different best trees over the same probabilities

1.3.2 Constructing An Optimal Prefix Code

Huffman has given an extremely elegant algorithm for constructing a best tree for a given set of symbols with associated probabilities.[6]

The optimal prefix code construction problem is: given a bag of nonnegative numbers, construct a best tree. That is, construct a binary tree and label its

[6]I call an algorithm elegant if it is easy to state and hard to prove correct.

leaves by the numbers from the bag so that the weight, i.e., the sum of the weighted pathlengths to the leaves, is minimized.

The Huffman Algorithm If bag b has a single number, create a tree of one node, which is both a root and a leaf, and label the node with the number. Otherwise (the bag has at least two numbers), let u and v be the two smallest numbers in b, not necessarily distinct. Let $b' = b - \{u, v\} \cup \{u + v\}$, i.e., b' is obtained from b by replacing its two smallest elements by their sum. Construct a best tree for b'. There is a leaf node in the tree labeled $u + v$; expand this node to have two children that are leaves and label them with u and v.

Illustration of Huffman's Algorithm Given a bag $\{0.05,\ 0.5,\ 0.4,\ 0.05\}$, we obtain successively

$b_0 = \{0.05,\ 0.5,\ 0.4,\ 0.05\}$	, the original bag
$b_1 = \{0.1,\ 0.5,\ 0.4\}$	, replacing $\{0.05,\ 0.05\}$ by their sum
$b_2 = \{0.5,\ 0.5\}$	, replacing $\{0.1,\ 0.4\}$ by their sum
$b_3 = \{1.0\}$	, replacing $\{0.5,\ 0.5\}$ by their sum

The trees corresponding to these bags are shown in Figure 1.4:

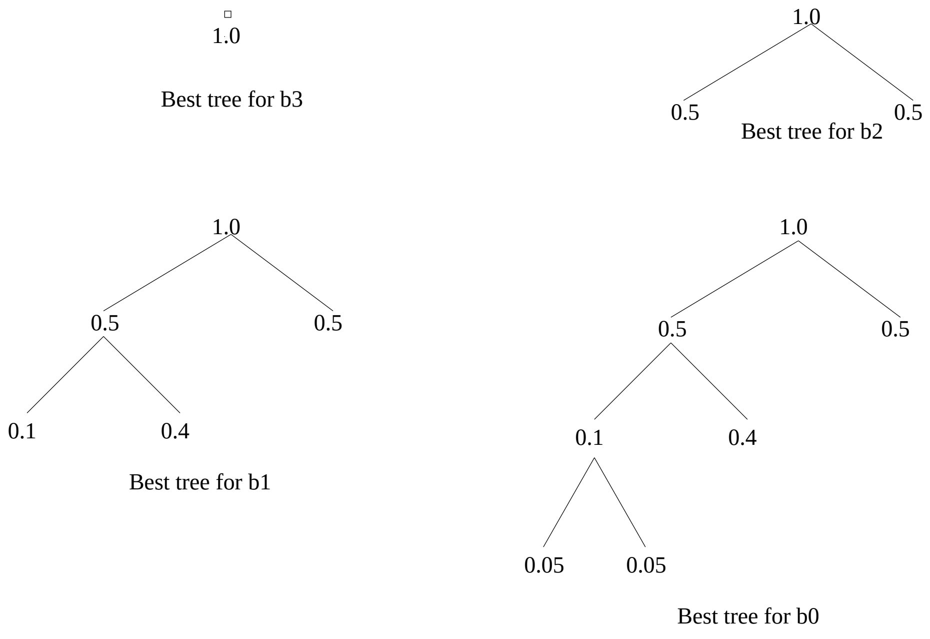

Figure 1.4: Illustration of Huffman algorithm

1.3.3 Proof of Correctness

We prove that Huffman's algorithm yields a best tree.

Lemma 1: Let x and y be two values in a bag where $x < y$. In a best tree for the bag, $Y \leq X$, where X and Y are the pathlengths to x and y.

Proof: Let T be a best tree. Switch the two values x and y in T to obtain a new tree T'. The weighted pathlengths to x and y in T are xX and yY, respectively. And, they are xY and yX, respectively, in T'. Weighted pathlengths to all other nodes in T and T' are identical. Since T is a best tree, weight of T is less than equal to that of T'. Therefore,

$$\begin{array}{ll} & xX + yY \leq xY + yX \\ \Rightarrow & \{\text{arithmetic}\} \\ & yY - xY \leq yX - xX \\ \Rightarrow & \{\text{arithmetic}\} \\ & (y-x)Y \leq (y-x)X \\ \Rightarrow & \{\text{since } x < y,\ (y-x) > 0;\ \text{arithmetic}\} \\ & Y \leq X \end{array}$$

Lemma 2: Let u and v be two smallest values, respectively, in bag b. Then, there is a best tree for b in which u and v are siblings.

Proof: Let T be a best tree for b. Let U and V be the pathlengths to u and v, respectively. Let the sibling of u be x and X be the pathlength to x. (In a best tree, u has a sibling. Otherwise, delete the edge to u, and let the parent of u become the node corresponding to value u, thus lowering the cost.)

If $v = x$, the lemma is proven. Otherwise, $v < x$.

$$\begin{array}{ll} & v < x \\ \Rightarrow & \{\text{from Lemma 1}\} \\ & X \leq V \\ \Rightarrow & \{u \leq v \text{ and } v < x.\ \text{So, } u < v.\ \text{from Lemma 1}\} \\ & X \leq V \text{ and } V \leq U \\ \Rightarrow & \{X = U, \text{ because } x \text{ and } u \text{ are siblings}\} \\ & X = V \end{array}$$

Switch the two values x and v (they may be identical). This will not alter the weight of the tree because $X = V$, while establishing the lemma.

Lemma 3: Let T be an optimal tree for bag b in which u and v are siblings. Let T' be all of T except the two nodes u and v; see Figure 1.5. Then T' is a best tree for bag $b' = b - \{u, v\} \cup \{u + v\}$.

Proof: Let $W(T)$ and $W(T')$ be the weighted pathlengths of T and T', respectively. Let the pathlength to $u + v$ in T' be p. Then the pathlengths to u and v

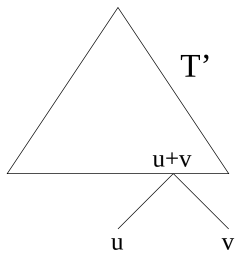

Figure 1.5: The entire tree is T for b; its upper part is T' for b'.

in T are $p + 1$. The weighted pathlengths to all other nodes are identical in T and T'; let the combined pathlengths to all other nodes be q. Then

$$\begin{array}{ll} & W(T) \\ = & \{\text{definition of weighted pathlength of } T\} \\ & q + (p+1) \times u + (p+1) \times v \\ = & \{\text{arithmetic}\} \\ & q + p \times (u+v) + (u+v) \\ = & \{\text{definition of weighted pathlength of } T'\} \\ & W(T') + (u+v) \end{array}$$

Since T is the best tree for b, T' is the best tree for b'. Otherwise, we may replace T' by a tree whose weight is lower than $W(T')$, thus reducing $W(T)$, a contradiction since T is the best tree.

We combine Lemma 2 and 3 to get the following theorem, which says that Huffman's algorithm constructs a best tree.

Theorem: Given is a bag b. Let u and v be two smallest values in b. And, $b' = b - \{u, v\} \cup \{u + v\}$. There is a best tree T for b such that

1. u and v are siblings in T, and

2. T' is a best tree for b', where T' is all of T except the two nodes u and v; see Figure 1.5.

Exercise 4

1. What is the structure of the Huffman tree for 2^n, $n \geq 0$, equiprobable symbols?

2. Show that the tree corresponding to an optimal prefix code is a *full* binary tree.

3. In a best tree, consider two nodes labeled x and y, and let the corresponding pathlengths be X and Y, respectively. Show that

$x < y \;\Rightarrow\; X \geq Y$

4. Prove or disprove (in the notation of the previous exercise)

   $x \leq y \;\Rightarrow\; X \geq Y$, and
   $x = y \;\Rightarrow\; X \geq Y$

5. Consider the first n fibonacci numbers (start at 1). What is the structure of the tree constructed by Huffman's algorithm on these values?

6. Prove that the weight of any tree is the sum of the values in the non-leaf nodes of the tree. For example in Figure 1.4, the weight of the final tree is 1.6, and the sum of the values in the non-leaf nodes is $1.0+0.5+0.1 = 1.6$.

   Observe that a tree with a single leaf node and no non-leaf node has weight 0, which is the sum of the values in the non-leaf nodes (vacuously).

   Does the result hold for non-binary trees?

7. Show that the successive values computed during execution of Huffman's algorithm (by adding the two smallest values) are nondecreasing.

8. Combining the results of the last two exercises, give an efficient algorithm to compute the weight of the optimal tree; see Section 1.3.4.

9. (Research) As we have observed, there may be many best trees for a given bag. We may wish to find the *very best* tree that is a best tree and in which the maximum pathlength to any node is as small as possible, or the sum of the pathlengths to the leaves is minimized. The following procedure achieves both of these goals simultaneously: whenever there is a tie in choosing values, always choose an original value rather than a combined value. Show the correctness of this method and also that it minimizes the maximum pathlength as well as the sum of the pathlengths among all best trees. See Knuth [28], Section 2.3.4.5, page 404. □

How Good is Huffman Coding? We know from Information theory (see Section 1.2) that it is not possible to construct code whose weight is less than the entropy, but it is possible to find codes with this value (asymptotically). It can be shown that in any alphabet where the probabilities of symbols are non-zero and the entropy is h, the Huffman code with weight H satisfies:

$$h \leq H < h + 1$$

So, the ratio $(H - h)/h$ will be very low if h is large.

However, in another sense, Huffman coding leaves much to be desired. The probabilities are very difficult to estimate if you are compressing something other than standard English novels. How do you get the frequencies of symbols in a postscript file? And, which ones should we choose as symbols in such a file? The latter question is very important because files tend to have bias toward certain phrases, and we can compress much better if we choose those as our basic symbols.

The Lempel-Ziv code, described in the following section addresses some of these issues.

1.3.4 Implementation

During the execution of Huffman's algorithm, we will have a bag of elements where each element holds a value and it points to either a leaf node —in case it represents an original value— or a subtree —if it has been created during the run of the algorithm. The algorithm needs a data structure on which the following operations can be performed efficiently: (1) remove the element with the smallest value and (2) insert a new element. In every step, operation (1) is performed twice and operation (2) once. The creation of a subtree from two smaller subtrees is a constant-time operation, and is left out in the following discussion.

A priority queue supports both operations. Implemented as a heap, the space requirement is $O(n)$ and each operation takes $O(\log n)$ time, where n is the maximum number of elements. Hence, the $O(n)$ steps of Huffman's algorithm can be implemented in $O(n \log n)$ time.

If the initial bag is available as a sorted list, the algorithm can be implemented in linear time, as follows. Let *leaf* be the list of initial values sorted in ascending order. Let *nonleaf* be the list of values generated in sequence by the algorithm (by summing the two smallest values in $\textit{leaf} \cup \textit{nonleaf}$).

The important observation is that

- (monotonicity) *nonleaf* is an ascending sequence.

You are asked to prove this in part 7 of the exercises in Section 1.3.3.

This observation implies that the smallest element in $\textit{leaf} \cup \textit{nonleaf}$ at any point during the execution is the smaller of the two items at the heads of *leaf* and *nonleaf*. That item is removed from the appropriate list, and the monotonicity property is still preserved. An item is inserted by adding it at the tail end of *nonleaf*, which is correct according to monotonicity.

It is clear that *leaf* is accessed as a list at one end only, and *nonleaf* at both ends, one end for insertion and the other for deletion. Therefore, *leaf* may be implemented as a stack and *nonleaf* as a queue. Each operation then takes constant time, and the whole algorithm runs in $O(n)$ time.

1.4 Lempel-Ziv Coding

As we have noted earlier, Huffman coding achieves excellent compression when the frequencies of the symbols can be predicted, and when we can identify the interesting symbols. In a book, say *Hamlet*, we expect the string *Ophelia* to occur quite frequently, and it should be treated as a single symbol. Lempel-Ziv coding does not require the frequencies to be known a priori. Instead, the sender scans the text from left to right identifying certain strings (henceforth, called *words*) that it inserts into a *dictionary*. Let me illustrate the procedure when the dictionary already contains the following words. Each word in the dictionary has an *index*, simply its position in the dictionary.

index	word
0	$\langle\rangle$
1	t
2	a
3	ta

Suppose the remaining text to be transmitted is *taaattaaa*. The sender scans this text from left until it finds a string that is *not* in the dictionary. In this case, *t* and *ta* are in the dictionary, but *taa* is not in the dictionary. The sender adds this word to the dictionary, and assigns it the next higher index, 4. Also, it transmits this word to the receiver. But it has no need to transmit the whole word (and, then, we will get no compression at all). The prefix of the word excluding its last symbol, i.e., *ta*, is a dictionary entry (remember, the sender scans the text just one symbol beyond a dictionary word). Therefore, it is sufficient to transmit $(3, a)$, where 3 is the index of *ta*, the prefix of *taa* that is in the dictionary, and *a* is the last symbol of *taa*.

The receiver recreates the string *taa*, by loooking up the word with index 3 and appending *a* to it, and then it appends *taa* to the text it has created already; also, it updates the dictionary with the entry

index	word
4	taa

Initially, the dictionary has a single word, the empty string, $\langle\rangle$, as its only (0th) entry. The sender and receiver start with this copy of the dictionary and the sender continues its transmissions until the text is exhausted. To ensure that the sender can always find a word which is not in the dictionary, assume that the end of the file, written as #, occurs nowhere else in the string.

Example Consider the text *taccagtaccagtaccacta#*. The dictionary and the transmissions are shown in Table 1.4. □

It should be clear that the receiver can update the dictionary and recreate the text from the given transmissions. Therefore, the sequence of transmissions constitutes the compressed file. In the small example shown above, there is hardly any compression. But for longer files with much redundancy, this scheme achieves excellent results. Lempel-Ziv coding is asymptotically optimal, i.e., as the text length tends to infinity, the compression tends to the optimal value predicted by information theory.

The dictionary size is not bounded in this scheme. In practice, the dictionary is limited to a fixed size, like 4096 (so that each index can be encoded in 12 bits). Beyond that point, the transmissions continue in the same manner, but the dictionary is not updated. Also, in practical implementations, the dictionary is initially populated by all the symbols of the alphabet.

index	word	transmission
0	$\langle\rangle$	none
1	t	$(0, t)$
2	a	$(0, a)$
3	c	$(0, c)$
4	ca	$(3, a)$
5	g	$(0, g)$
6	ta	$(1, a)$
7	cc	$(3, c)$
8	ag	$(2, g)$
9	tac	$(6, c)$
10	cac	$(4, c)$
11	$ta\#$	$(6, \#)$

Table 1.4: Transmission of *taccagtaccagtaccacta#* using Lempel-Ziv Code

There are a number of variations of the Lempel-Ziv algorithm, all having the prefix LZ. What I have described here is known as LZ78 [54]. Many popular compression programs —Unix utility "compress", "gzip", Windows "Winzip"— are based on some variant of the Lempel-Ziv algorithm. Another algorithm, due to Burrows and Wheeler [9], is used in the popular "bzip" utility.

Implementation of the Dictionary We develop a data structure to implement the dictionary and the two operations on it: (1) from a given string find the (shortest) prefix that is not in the dictionary, and (2) add a new entry to the dictionary. The data structure is a special kind of tree (sometimes called a "trie"). Associated with each node of the tree is a word of the dictionary and its index; associated with each branch is a symbol, and branches from a node have different associated symbols. The root node has the word $\langle\rangle$ (empty string) and index 0 associated with it. The word associated with any node is the sequence of symbols on the path to that node. Initially, the tree has only the root node.

Given a text string, the sender starts matching its symbols against the symbols at the branches, starting at the root node. The process continues until a node, n, is reached from which there is no branch labelled with the next input symbol, s. At this point, index of n and the symbol s are transmitted. Additionally, node n is extended with a branch labelled s.

The receiver does not need to maintain the tree. The receiver merely maintains an array D, where $D(i)$ is the string of index i in the tree.

The tree shown in Figure 1.6 results from the string *taccagtaccagtaccacta#*. The first 8 dictionary entries result from the prefix *taccagtaccag*. Now consider transmission of the remaining portion of the string, *taccacta#*. The prefix *ta* matches the symbols along the path up to the node with index 6. Therefore, index 6 and the next symbol, c, are transmitted, the tree is updated by adding a branch out of node 6, labelled c, and the new node acquires the next index at that point, 9. Transmission of the remaining portion of the string follows the same procedure.

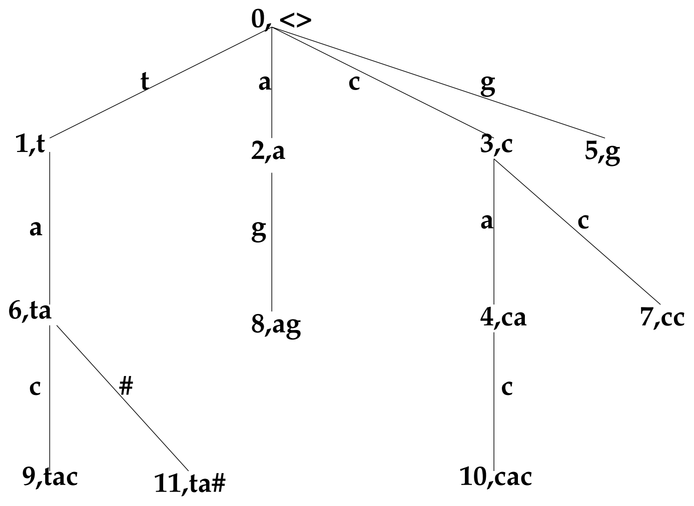

Figure 1.6: Trie resulting from transmission of *taccagtaccagtaccacta#*

Exercise 5

1. Is it necessary to maintain the word at each node?

2. If your input alphabet is large, it will be non-trivial to look for a branch out of a node that is labeled with a specific symbol. Devise an efficient implementation of the tree in this case.

3. Suppose that the string *Ophelia* appears in the text as a separate word (surrounded by white speces), but none of its prefixes do. What is the minimum number of times this string must be seen before it is encoded as a word? □

Acknowledgment I am grateful to Thierry Joffrain for helping me write part of Section 1.2.