Chapter 3

Variable Length Codes

In Section 2.1, we pointed out that using varying length sequences of digits to encode letters of the alphabet can get us into trouble. We then focused our attention on schemes that used a fixed number of digits to represent each letter. Fixed length codes are indeed very important. As we have seen, ASCII, the most widely used code for representing text in computer systems, is a fixed length code. With care, however, it is possible to design a viable code in which the number of digits used to encode letters varies. Such codes are used in many applications because they can be more efficient than fixed length codes. That is, the number of digits required to represent a given message with a variable length code is often smaller that what would be required if a fixed length code were used. In this chapter, we will explore the use of variable length codes. As we did in the preceding section, we will begin by considering codes based on using the 10 decimal digits. Then, after illustrating the underlying ideas with such examples, we will discuss the use of binary codes with variable length codewords.

3.1 Unique Decoding

In Section 2.1, we considered a scheme in which the first nine letters of the alphabet were represented by the single digits 1 through 9, while the remaining letters were represented by pairs of digits corresponding to the numbers 10 through 26 as shown below.

1. a	11. k	21. u
2. b	12. l	22. v
3. c	13. m	23. w
4. d	14. n	24. x
5. e	15. o	25. y
6. f	16. p	26. z
7. g	17. q	
8. h	18. r	
9. i	19. s	
10. j	20. t	

We showed two possible interpretations for the sequence of digits “211814” under this scheme. If the first digit is interpreted as a “b”, then the code appears to represent the word “barn”. On

the other hand, if the first two digits are interpreted as the code for "u", then the code appear to represent "urn".

Actually, there are many possible interpretations of "211814". The middle digits, "18", can be interpreted as a single "r", or as the pair of letters "ah". The final digits, "14", could be interpreted as "n" or "ad". As a result, the sequence "211814" could represent "barn", "urn", "bahn", "uahn", "barad", "urad", "baahad", or "uahad". "Barn" and "urn" are the only interpretations that correspond to English words, but "bahn" is a word in German and several of the other possible interpretations are used as proper names. (Just use them as search terms in Google.)

By way of contrast, consider the interpretation of the sequence "5105320". Since the pair "51" is not used as a codeword in our scheme, the first digit must stand for the letter "e". If we try to interpret the following "1" as an "a", then the next codeword would start with "0". There are no codewords that start with "0", so we must interpret the "1" as part of the two digit codeword "10" which stands for "j". Again, there are no two digit codes that start with "5" or "3", so the next two letters must be "ec". Finally, the last two digits can only be interpreted as "t" so the encoded word must be "eject". In this case, there are no alternatives. We say that such a coded message is *uniquely decodable*.

The coded message "5105320" is uniquely decodable because the codewords used have properties that make it possible to determine the boundaries between codewords without either using delimiters or fixed length codes. For example, the fact that no codeword used in our scheme begins with "0" enables us to recognize that the sequences "10" and "20" must be interpreted as two digit codewords. With care, we can design coding schemes that use variable length codewords selected to ensure that all coded message can be decoded in only one way. That is, we can design variable length coding schemes that are uniquely decodable.

Exercise 3.1.1 *In fact, our list of possible interpretations of "211814" is still incomplete. There are more than eight. Find as many additional ways to interpret "211814" using the coding table shown above as you can. Hint: The remaining interpretations may be even less word-like than the eight we have identified.*

Exercise 3.1.2 *We showed that the coded message produced for the word "eject" was uniquely decodable. Find another example of an English word that produces a coded message that is uniquely decodable.*

When analyzing the coded message "5105320", we determined that the digits "5" and "3" had to be interpreted as single digit codewords because the scheme we were using did not include any two digit codewords that began with "5" or "3". With just a few changes to our coding scheme, we can ensure that a similar rule can be used to distinguish all single digit codewords from two digit codewords.

In our original scheme, the two digit codewords all begin with either "1" or "2". Thus, the only single digit codewords that can cause confusion are those used for "a" and "b". If we replace the codes for "a" and "b" with two digit codes as shown in Figure 3.1 these sources of confusion are removed.

Now, "barn" is encoded as "28271814" while "urn" is encoded as "211814" and each of these codes is uniquely decodable. In fact, any codeword produced using our new scheme is certain to be uniquely decodable. Therefore, even though this is a variable length code, it can be used to encode messages without requiring any additional delimiters to separate codewords.

27. a	11. k	21. u
28. b	12. l	22. v
3. c	13. m	23. w
4. d	14. n	24. x
5. e	15. o	25. y
6. f	16. p	26. z
7. g	17. q	
8. h	18. r	
9. i	19. s	
10. j	20. t	

Figure 3.1: A prefix-free code for the alphabet

Our revised code is an example of a *prefix-free* code. We say that a coding scheme is prefix-free if no codeword used in the scheme appears as a prefix of any other codeword. Any coding scheme that is prefix-free is guaranteed to produce coded messages that are uniquely decodable.

3.2 Exploiting Symbol Frequencies

The possibility of using variable length codewords is of interest because a variable length code may require fewer symbols to represent a message than a coding scheme based on fixed length codewords. For example, if we encode the word "shorter" using our variable length scheme, the resulting coded message "198151820518" is 12 digits long. On the other hand, encoding "shorter" using the fixed length scheme we proposed in Section 2.1 would require 14 digits since each letter uses two digits. The variable length coding scheme saves two digits because "shorter" contains two letters, "e" and "h" that have single digit codewords.

Reducing the number of symbols used to encode a message can be quite important. If the message is going to be transmitted through a computer network, the amount of time required for the transmission will be proportional to the number of symbols required to encode it. If we can reduce the number of symbols in the encoding by 10%, then the message can be transmitted 10% more quickly.

To maximize the savings obtained by using a code with variable length codewords, we should arrange to assign short codewords to the symbols that will appear most frequently in the messages we encode and longer codewords to the symbols that appear less frequently. To do this, we need information about how frequently various symbols appear in the text we want to encode.

It is difficult to give accurate letter occurrence values for the English language. The frequencies with which letters occur vary slightly depending on what samples of English text you examine. Fortunately, all we need is a set of representative frequency estimates that we can use to motivate our consideration of various coding schemes. With this in mind, we will use the frequencies with which various letters appear in the text of one book, Alice in Wonderland.

In Figure 3.2 we show the fractions of the text of Alice in Wonderland accounted for by each of the 26 letters of the alphabet. That is, assuming that there are a total of N letters in the book, there must be $0.126N$ e's, $0.046N$ d's, and $0.001N$ z's. (In fact, the fractions given in the table have clearly been rounded to the nearest thousandth, so $0.126N$ and $0.046N$ do not describe the

e	0.126	d	0.046	p	0.014
t	0.099	l	0.043	b	0.014
a	0.082	u	0.032	k	0.011
o	0.076	w	0.025	v	0.008
i	0.070	g	0.024	q	0.002
h	0.069	c	0.022	x	0.001
n	0.064	y	0.021	j	0.001
s	0.060	m	0.020	z	0.001
r	0.051	f	0.018		

Figure 3.2: Fractional distribution of letters in text of Alice in Wonderland

0. e	80. b	90. q
1. t	81. c	91. r
2. a	82. d	92. u
3. o	83. f	93. v
4. i	84. g	94. w
5. h	85. j	95. x
6. n	86. k	96. y
7. s	87. l	97. z
	88. m	
	89. p	

Figure 3.3: A more efficient prefix-free code

exact numbers of e's and d's. For the purpose of our discussion, however, we ask you to assume that the numbers in the table are exact.)

The code presented in Figure 3.1 uses single digit codes for the seven letters c, d, e, f, g, h, and i. Examining the table in Figure 3.2, we can see that this list only includes three of the most frequently occurring letters. Accordingly, if we construct a new prefix-free coding scheme in which more of the commonly occurring letters are assigned one digit codewords then we would expect this code to require fewer digits to encode text. The code in Figure 3.1 also wastes one short codeword. The single digit "0" could be used as a codeword while preserving the prefix-free property. Figure 3.3 shows an example of a coding scheme designed to address both of these issues.

The new coding scheme uses different prefix values to indicate code lengths. In the original code, the two digit codewords began with "1" and "2". In our new scheme, we have used the prefixes "8" and "9" to identify two digit codewords. The digits "0" through "7" are used as single digit codewords. The letters associated with these codewords are those associated with the largest frequency values in Figure 3.2: e, t, a, o, i, h, n, and s.

Exercise 3.2.1 *Show how to encode the words in the sentence:*

"Curiouser and curiouser" cried Alice

using the coding schemes shown in Figures 3.1 and 3.3 (while ignoring punctuation, spaces, and capitalization). How many digits are required by each scheme?

0. e	90. d	990. p
1. t	91. l	991. b
2. a	92. u	992. k
3. o	93. w	993. v
4. i	94. g	994. q
5. h	95. c	995. x
6. n	96. y	996. j
7. s	97. m	997. z
8. r	98. f	

Figure 3.4: A prefix-free code using three codeword lengths

The coding scheme shown in Figure 3.3 is as efficient a code as possible given the letter frequencies shown in Figure 3.2 and the assumption that we should only use codewords of length one and two. We could, however, possibly obtain a more efficient code by using longer codewords. This may seem like a ridiculous suggestion. How could using longer codewords produce a more efficient code? It is worth trying, however, because using longer codewords makes it possible to use more short codewords.

In the scheme shown in Figure 3.3, we used 8 single digit codewords. It is obvious that we can't use all 10 digits as single digit codewords. If we did this, the resulting code could not be prefix-free. We might, however, hope to use 9 single digit codewords rather than just 8. For example, we might make 8 the codeword for "r", the ninth most common letter. If we do this, all longer codes will have to start with 9. There are only 10 two digit codes that begin with 9. Therefore, if we limit ourselves to using at most 2 digits, we will only have 19 codewords. We need 26.

The alternative is to use only 9 of the two digit codewords and then to use the remaining two digit pair as a prefix for three digit codewords. For example, we could use the codewords 90, 91, ... 98 and then reserve 99 as a prefix indicating the use of a three digit codeword. If we want to make this code as efficient as possible, we will want to associate the letters that occur least frequently with the three digit codewords, the letters that occur most frequently with one digit codewords, and the remaining letters with two digit codewords. These three groups of letters correspond to the three columns shown in the table of letter frequencies in Figure 3.2. Therefore, the coding scheme shown in Figure 3.4 would be a reasonable way to use three digit codewords. As an example, using this coding scheme, the word "reject" would be encoded using the digits "809960951".

Exercise 3.2.2 *Show how to encode the words in the sentence:*

"Curiouser and curiouser" cried Alice

using the coding schemes shown in Figures 3.4 (while ignoring punctuation, spaces, and capitalization). How many digits are required?

3.3 Evaluating a Variable Length Code

We have now considered three different variable length coding schemes for encoding text. In the first, we simply assigned short codes to the first few letters of the alphabet and long codes to the remaining letters. In the other two schemes, we attempted to reduce the number of digits required

by associating short codes with the letters that are used most often and longer codes with the remaining letters. How can we accurately compare the performance of these codes?

If we know exactly what message will be encoded, we can compare various schemes by simply encoding the message to be encoded with each scheme. For example, given that our last two codes were based on the frequency with which letters appear in Alice in Wonderland, we might evaluate them by encoding the entire text of Alice in Wonderland with each of the schemes.

Fortunately, we can use the table in Figure 3.2 to compute the number of digits required to encode Alice in Wonderland without having to actually encode the text. Suppose that the total number of letters that appear in the text of Alice in Wonderland is N. If we take the fraction of the text that a given letter accounts for and multiply by N, we get the number of times that letter appears in the text. In the case of Alice in Wonderland, we can therefore conclude that there must be $.082N$ a's and $.014N$ b's in the book. Continuing in this way we can describe exactly how many letters in the book would be represented using single digit codewords in any of our schemes. For example, the coding table in Figure 3.1 uses single digit codewords for the letters c, d, e, f, g, h, and i. Looking at Figure 3.2, we can conclude that taken together, these letters account for

$$.022N + .046N + .126N + .018N + .024N + .069N + .070N$$

or

$$.375N$$

of the letters in Alice in Wonderland. All the other letters in the book will be encoded using 2 digit codewords. There must be a total of $N - .375N$ or $.625N$ such words if there are a total of N words in the book. Therefore, we can conclude that the total number of digits that would be used to encode the text would be

$$1(.375N) + 2(.625N)$$

which can be simplified to

$$(1 \times .375 + 2 \times .625)N$$

or

$$1.625N$$

Using a similar process, we can derive formulas for the numbers of digits required to encode the text using the schemes described by Figures 3.3 and 3.4. The code in Figure 3.3 encodes the letters e, t, a, o, i, h, n, and s using single digit codewords. These letters account for

$$(.126 + .099 + .082 + .076 + .070 + .069 + .064 + .060)N$$

or

$$.646N$$

of the letters in the text. The remaining letters are encoded using 2 digit codes. Therefore, the number of digits required to encode the text will be

$$1(.646N) + 2(1 - .646)N$$

which simplifies to

$$(1 \times .646 + 2 \times .354)N$$

or

$$1.354N$$

This second scheme is clearly better than the first scheme. It requires $1.354N$ digits to encode the book which is less than the $1.624N$ digits used by the other scheme.

Finally, if you compare the encoding table shown in Figure 3.4 with the table of letter frequencies in Figure 3.2, you will notice that we arranged these two tables so that the letters in both tables are arranged in the same order. As a result, all the letters represented using single digit codewords fall in the first column of both tables, all the letters represented by two digit codewords fall in the second column, and all the letters represented by three digit codewords fall in the third column. By summing the percentages in each column, we can see that when the third scheme is used, 69.7% of the text will be encoded using single digits, 25.1% with double digits and only 5.2% with three digit codewords. This implies that the total number of digits used will be

$$(1 \times .697 + 2 \times .251 + 3 \times .052)N$$

or

$$1.355N$$

The value of this formula will be just a little bit bigger than the formula we obtained for the second scheme, $1.354N$.

We can see why the second scheme performs just a tiny bit better than the third scheme by carefully considering the differences between the two schemes. The third scheme encode the letter r with one digit instead of 2. Since 5.1% of the letters are r's, this will reduce the number of digits required by $0.051N$. Unfortunately, the third scheme uses one extra digit for all the letters that appear in the last column. Since these letters account for a total of 5.2% of the total, this causes an increase of 0.052. The difference between this increase and the decrease resulting from using only one digit for r's is $0.001N$.

Our analysis of the third coding scheme, however, makes it clear that there are cases where a variable length code can be made more efficient by using longer codes for some letters. If r's accounted for 5.5% of the letters in the text and p's only accounted for 1.0%, then we would save $0.055N$ digits by using a one digit codeword for r while switching the eight least common letters to three digit codewords would only require an additional $0.048N$ digits. In this case, the code with three digit codewords would save $0.007N$ digits. Clearly, finding the best coding scheme for a particular document requires careful attention to the frequency with which various letters occur in the document.

3.4 Probabilities and Expected Values

In many practical applications, a single coding scheme may be used to encode many different messages over time. In such situations, it is more useful to measure or predict the average performance of the scheme on a variety of typical messages than to determine the number of digits required to encode a particular message. The trick is to define "typical." Is it reasonable to assume that the text of Alice in Wonderland is typical of all English writing (including email and IM messages)? If not, is there some collection of books, or network messages that we can identify as "typical"?

The good news is that once a collection of typical text is identified, we can predict the behavior of encoding scheme using a table of letter frequencies for the selected text much like that shown

in Figure 3.2. In fact, the calculations performed to predict the behavior of a scheme on typical message will be very similar to those used to calculate the number of digits required to encode Alice in Wonderland in the last section. The interpretations associated with the values used in the computation, however, will be different in subtle but important ways.

To appreciate these differences, suppose that instead of trying to encode all of Alice in Wonderland, we just encoded Chapter 4 using the scheme described in Figure 3.3. Just as we used N to represent the number of letters in the complete book, we will let M represent the number of letters in Chapter 4. Obviously, $M < N$.

It is unlikely, that the numbers in Figure 3.2 will exactly describe the distribution of letters in Chapter 4. Chances are that Chapter 4 may contain proportionally more a's or less b's or less c's, etc. than the entire book. Therefore, while we concluded that encoding the entire book would require exactly $1.354N$ digits, the formula $1.354M$ will not tell us the exact number of digits required to encode Chapter 4. We would, however, expect the percentages that describe the distribution of letters in Chapter 4 to be similar to those for the entire book. Therefore, we would expect $1.354M$ to be a good estimate of the number of digits required to encode the chapter.

The same logic applies if we encode even smaller sections of the book. Instead of encoding an entire Chapter, we might encode just a page or even a single paragraph. If M denotes the number of letters in whatever subsection of the text we choose, then we would still expect $1.354M$ to provide a reasonable estimate of the number of digits that will be required to encode the subsection.

Taking this line of thought to the limit, suppose that we encode just a single, randomly chosen letter from the book. That is, let $M = 1$. We would then conclude that we expect that our encoding scheme will use 1.354 digits to encode the randomly chosen letter. Clearly, our scheme will never use 1.354 digits to encode any letter. Any individual letter is encoded using either exactly 1 or 2 digits. So we should think a little about how we should interpret the value 1.354.

The number 1.354 in our example is an example of what is called an *expected value*. This is a term that comes from the mathematical field of probability theory. In the terminology of probability theory, picking a random letter to encode from Alice in Wonderland is an experiment with 26 possible *outcomes*. The likelihood of each outcome is represented by a number between 0 and 1 which is called the *probability* of the outcome. The larger the probability associated with an outcome, the more likely it is that the outcome will actually occur. The sum of the probabilities of all possible outcomes must equal 1. Informally, the probability of a particular outcome equals the frequency with which the outcome would occur if the experiment were conducted many, many times. In particular, we can interpret the values in Figure 3.2 as the probabilities that a randomly selected letter from Alice in Wonderland will match a particular letter of the alphabet. Using the notation $P(x)$ to denote the probability of outcome x, we would then say that $P(\text{a}) = 0.082, P(\text{b}) = 0.014, P(\text{c}) = 0.022$, and so on.

Probability theory recognizes that sometimes, it is not the actual outcome of an experiment that matters. Instead, some particular property of the outcome may be all that is of interest. For example, a biologist might plant and observe the growth of 100 plants. Each plant produced would represent an "outcome" with many features including the number of leaves on each plant, the color of the flowers, etc. It may be the case, however, that all the biologist cares about is the height of the plants.

To capture this idea, probability theory uses the notion of a *random variable*. Unfortunately, this is a very misleading name. A random variable is not a variable at all. Instead, a random variable is a function from the set of all possible outcomes of an experiment to some other set of

values that represent a feature of interest. For example, the random variable of interest in our biology example would be the function that mapped a plant to it height.

Applying the same terminology to our Alice in Wonderland example, the random variable we are interested in is the function that maps a letter of the alphabet to the length of the codeword associated with that letter in Figure 3.3. If we name this random variable L (for length), then $L(\text{a}) = 1, L(\text{b}) = 2, L(\text{c}) = 2, L(\text{d}) = 2, L(\text{e}) = 1$, and so on.

Finally, given a random variable, the *expected value* of the variable is defined as the sum over all possible outcomes of the product of the probability of the outcome and the value of the random variable for that outcome. If X is a random variable, we use $E(X)$ to denote the expected value of X. Thus, in the general case, we have

$$E(X) = \sum_{x \in \{outcomes\}} P(x)X(x)$$

In the case of our Alice in Wonderland example, we would write

$$1.354 = E[L] = \sum_{x \in \{\text{a, b, c, ... }\}} P(x)L(x)$$

3.5 Variable Length Binary Codes

We can construct variable-length binary codes that are uniquely decodable in much the same way we constructed such codes using the digits 0 through 9. We simply have to ensure that the set of codewords we select is prefix free. That is, no codeword can appear as a prefix of any other codeword.

We have already seen that binary codes require longer codewords than codes based on decimal digits. To avoid having to use very long binary codes, we will begin by considering an alphabetic coding example that uses considerably fewer than the 26 letters of the Roman alphabet. Suppose that you are charged with the task of designing a binary encoding scheme for letter grades assigned to students in courses at some university. That is, you need a scheme for encoding the five letters A, B, C, D, and F. This scheme will then be used to encode exciting "messages" like "ABFABCDABBC".

There are many different prefix-free variable-length binary codes that could be used to encode these five letters. Three such schemes are shown in Figure 3.5. With a bit of effort we can verify that each of these codes is prefix free. For example, code (a), shown on the left in the figure, uses three 2-digit codewords and two 3-digit codewords. Both of the 3-digit codewords start with the prefix "11" which is not used as a 2-digit codeword. As a result, we can conclude that this code is prefix free.

Because these three codes assign codewords of different lengths to the various letters of the alphabet, the total number of digits required to encode a given message will depend on the code used. For example, suppose 10 students were assigned the grades B, C, A, B, A, F, C, A, B, and C. This sequence of grades, "BCABAFCABC", would be encoded as:

- `01101100111010111101100110` using the scheme described by table (a),
- `00000110001010110011000001` by scheme (b), and
- `01001101100010011011001` by scheme (c).

Letter Grade	Code-word
A	110
B	01
C	10
D	00
F	111

(a)

Letter Grade	Code-word
A	1
B	000
C	001
D	010
F	011

(b)

Letter Grade	Code-word
A	1
B	01
C	001
D	0000
F	0001

(c)

Figure 3.5: Three variable-length binary codes for letter grades

While it is not difficult to verify that these binary coding schemes are prefix free, larger alphabets will require much longer codewords making it more difficult to verify that a code is prefix-free by simply examining the list of codewords. Fortunately, there is a simple way to diagram the structure of a set of binary codewords that both makes it easy to decode messages and makes it clear that a given code satisfies the prefix free constraint.

The diagramming system is similar to decision trees sometimes used to represent the process a person should follow in reaching a decision of one sort or another. As an example of such a decision tree, in Figure 3.6 we provide a helpful guide to selecting a meal plan found at the Housing and Food Service website of the University of Nevada at Reno.[1] As the instructions provided with the diagram indicate, a student using it to select a food plan proceeds by answering a series of yes/no questions. Each answer determines the next step along a path that ultimately leads to a box that contains the food plan the student should select. These diagrams are called *trees* because they "branch" at the places that correspond to questions. Based on this analogy, the starting point is called the *root* (even though it is usually drawn at the top of the diagram rather than the bottom), and the endpoints of the paths where the final answers are placed are called *leaves*.

We can easily build decision trees that can be used to decode messages encoded with schemes like those shown in Figure 3.5. All of the questions in such decision trees will be the same: "Is the next digit in the encoded message a 0 or a 1?" As a result, we can draw these trees more compactly than the tree shown in Figure 3.6 by omitting the questions and simply labeling the paths between nodes with "0"s and "1"s. For example, trees describing how to decode the codes shown in Figure 3.5 are shown in Figure 3.7.

The diagram in Figure 3.8 illustrates how the tree from Figure 3.7(a) would be used to decode the message "0110". We begin at the root of the tree and with the first digit of the message, "0". Because the first digit is "0", we follow the path from the root that is labeled with "0". We show this in Figure 3.8(I) by darkening this path. The next digit, "1", would then be used to determine the path to take from the end of the edge highlighted in Figure 3.8(I). This tells us to take the path leading to the right as shown in Figure 3.8(II). This two-step path leads to a leaf labeled with the letter "B". Therefore, we conclude that the first letter encoded by the message is "B".

After reaching a leaf, we return to the root of the tree to continue decoding any binary digits remaining in the message. In this example, the first digit after the code for "B" is "1", so we follow the edge leading right from the root as shown in Figure 3.8(III). The last digit of the message, "0"

[1]Really. We didn't make this up. Visit http://www.reslife.unr.edu/decisionchart.html.

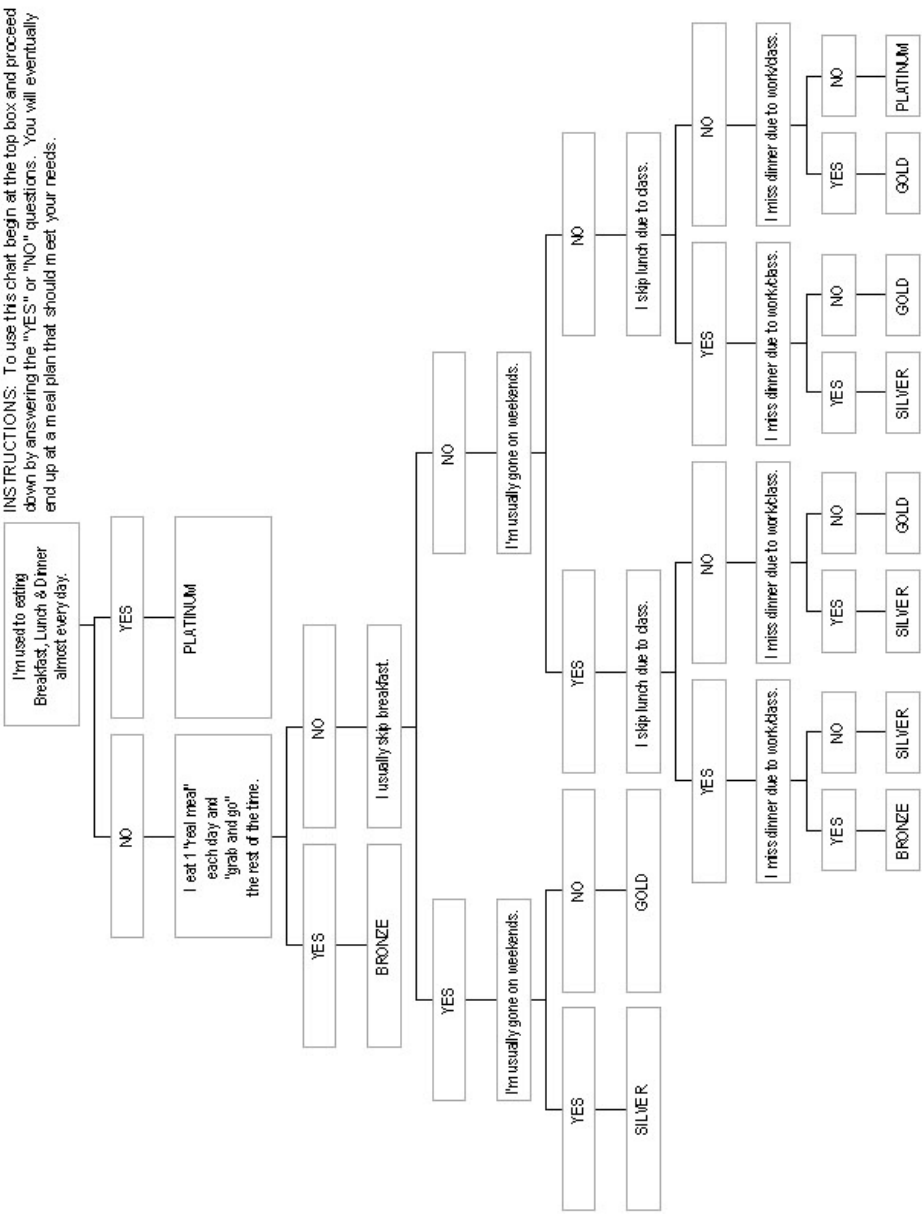


Figure 3.6: A decision chart for indecisive students

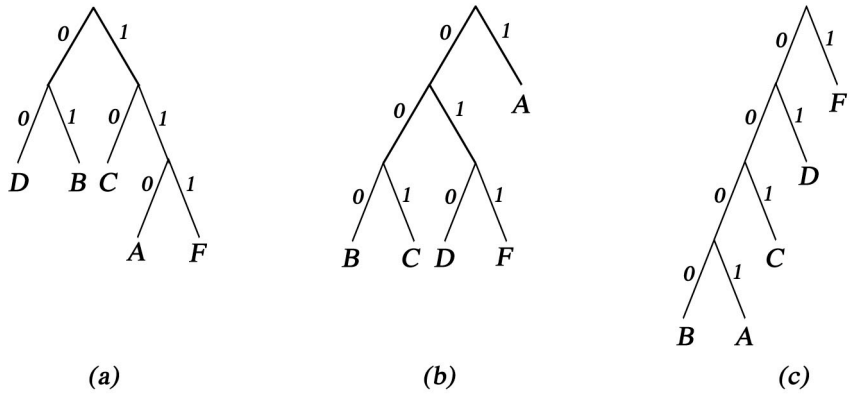


Figure 3.7: Decision trees for the coding schemes shown in Figure 3.5

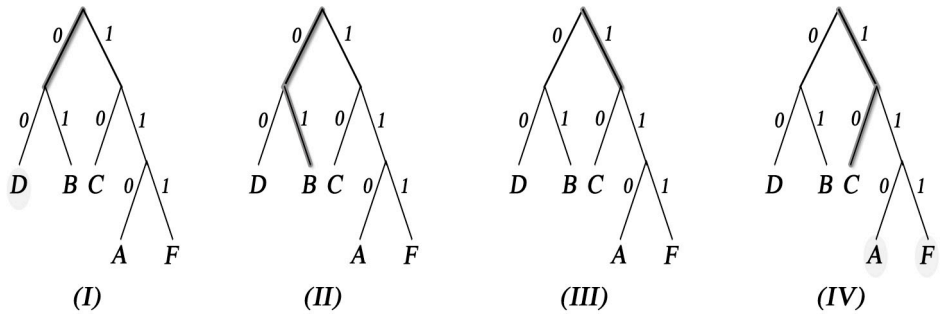


Figure 3.8: Using the decision tree from Figure 3.7(a) to decode 0110

Grade	Percent
A	20%
B	29%
C	25%
D	19%
F	7%

Figure 3.9: Distribution of letter grades

tells us to go to the left leading to a leaf labeled "C" as shown in Figure 3.8(IV). The complete message is therefore decoded as "BC".

3.6 Huffman Codes

Because the coding schemes shown in Figure 3.5 use different length codewords for different letters, the number of digits required to encode a given message may vary from one code to another. We can see this by looking at the encodings of the string of letter grades "BCABAFCABC" used as an example above. Schemes (a) and (b) use 24 binary digits to encode this sequence, but scheme (c) uses 22. Given such differences, we should try to use information about the messages that will be encoded to select the coding scheme that is likely to require the smallest number of binary digits.

As an example, suppose that at some college the registrar's office tallied all the grades assigned during the academic year and concluded that the percentages of As, Bs, Cs, Ds, and Fs assigned were those shown in Figure 3.9.[2] Given this data, if we are interested in encoding the grades for a typical course at this institution, it is be reasonable to assume that the probability that a randomly selected student received an A is 0.2, that the probability the student received a B is 0.29 and so on. We can then use these probabilities to compute the expected number of binary digits that will be used to encode letter grades using each of our three coding schemes.

Figure 3.10 shows tables that summarize the process of computing expected values for these coding schemes.

The figure contains one table for each of our three encoding schemes. Each row of a given table summarize all the information about the encoding of one letter grade within a given coding scheme. The first entry in each row is one of the five letter grades. This is followed by the codeword for the letter grade and the number of binary digits in the codeword. Next, we list the probability that a particular letter grade will be used. Finally, we show the product of the probability that a letter grade will be used and the length of its codeword. The sum of these products is the expected value of the length of the codeword for a randomly selected letter grade. These expected values are shown under the last column of each of the three tables. That is, we would predict that encoding grades with scheme (a) would require an average of 2.27 binary digits per letter grade, scheme (b) would require 2.6 digits per letter grade, and scheme (c) would require 2.57 digits per grade.

Clearly, of the three codes we have been considering, it would be best to use scheme (a). It should also be obvious, however, that scheme (a) is not THE best encoding scheme to use given the frequencies in Figure 3.9. According to the frequency figures, a grade of A occurs slightly more often than a D. In scheme (a), however, the codeword for A is longer than the codeword for D.

[2]This data is clearly from an institution that has resisted grade inflation much more effectively than most schools!

Letter Grade	Code Word	Code-word Length	Grade Proba-bility	Length x Prob
A	110	3	0.20	0.60
B	01	2	0.29	0.58
C	10	2	0.25	0.50
D	00	2	0.19	0.38
F	111	3	0.07	0.21

Expected digits/letter = 2.27

(a)

Letter Grade	Code Word	Code-word Length	Grade Proba-bility	Length x Prob
A	1	1	0.20	0.20
B	000	3	0.29	0.87
C	001	3	0.25	0.75
D	010	3	0.19	0.57
F	011	3	0.07	0.21

Expected digits/letter = 2.60

(b)

Letter Grade	Code Word	Code-word Length	Grade Proba-bility	Length x Prob
A	1	1	0.20	0.20
B	01	2	0.29	0.58
C	001	3	0.25	0.75
D	0000	4	0.19	0.76
F	0001	4	0.07	0.28

Expected digits/letter = 2.57

(c)

Figure 3.10: Expected digits per grade for coding schemes from Figure 3.5

Grade	Probability
A	0.20
B	0.29
C	0.25
E	0.26

Figure 3.11: Distribution of letter grades

Interchanging these codeword would therefore produce a more efficient code for this data. This reveals the fact that our three codes are just a sample of all the possible codes that could be used to encode letter grades. How can we be sure that we really have the best scheme without considering every possible coding system?

Fortunately, there is an algorithm that can be used to construct a variable length binary coding scheme that is guaranteed to be "best" in the sense that the expected number of digits it uses per symbol is less than or equal to that used by any other scheme. The algorithm was devised by David Huffman in 1951 while he was a graduate student at MIT. The codes produced are known as Huffman codes.

Huffman's algorithm exhibits a technique that often provides the key to solving a difficult problem: Find a way to solve a slightly simpler problem such that you can use the solution to the simpler problem as the basis for solving the original. We can illustrate this idea in the context of our letter grade example.

Given that we don't know how to find a code for the standard set of five letter grades, suppose that we instead tackle the problem of encoding grades from a system that only uses four letters. Looking at the statistics in Figure 3.9, it is clear that the grades D and F are the two grades that are the least frequently used. One can imagine that a school with such a grade distribution might decide that is was not worth keeping two grades that were used so rarely. They might decide to replace all Ds and Fs with some new grade, possibly an E. Assuming that the the new grade of E is used whenever a D or F would have been assigned, the new grading distribution would be described by the table in Figure 3.11. That is, the fraction of Es assigned would be the sum of the factions of Ds and Fs that had been assigned, 0.19 + 0.07.

We still don't know how to find the best coding scheme for the four letter grades, A, B, C, and E. We do know, however, that it will be slightly easier to do this than to find the best scheme for a five letter grading scheme, because there are simply fewer possible codes to consider. Best of all, once we choose a scheme to use for the four letter grades, there is a simple way to extend it for the original five letter system.

Suppose, for example, that we somehow determine that the best coding scheme to use for the grade distribution shown in Figure 3.11 is the one shown in Figure 3.12. In this scheme, the codeword for E is 11. We know that E is being used for all grades that would have been Ds and Fs. Therefore, a natural way to extend the four letter code to support the original five letter grades is to derive the codewords for D and F from the codeword for E. We do this by adding one extra digit to the codeword for E. For example, we might add a 0 to E's codeword, 11, to obtain the codeword for D, 110. Then, we would add a 1 to E's codeword to obtain the codeword for F, 111. The scheme for the five letter grades would then use the codewords listed in Figure 3.13.

The key to understanding Huffman's algorithm is to realize that we can apply the process of "simplifying" the problem by combining two letters over and over again. That is, in order to find

Grade	Codeword
A	00
B	01
C	10
E	11

Figure 3.12: Huffman Code for four letter grades

Grade	Codeword
A	00
B	01
C	10
D	110
F	111

Figure 3.13: Huffman Code for five letter grades

the coding scheme for the four letter grades shown in Figure 3.12, we will combine two of these grades to form a three letter grading system. To find a coding scheme for these three letters, we will again combine two of them to form a two letter grading system. Luckily, at this point we can stop because there is only one reasonable way to encode a pair of letter. We use a single 0 as the codeword for one letter and a single 1 for the other.

The standard scheme for applying Huffman's algorithm takes advantage of the fact that decision trees provide an alternative to tables like those in Figure 3.5 for describing coding schemes. That is, while we first showed a table of codewords and then later drew the corresponding decision tree, we can also work in the opposite direction. Given a decision tree, we can easily build a table of the codewords for all symbols. This is how Huffman's algorithm works. It tells us how to build a decision tree from which we can derive an optimal coding scheme.

When we apply Huffman's algorithm, we will keep track of fragments of what will eventually be the complete decision tree as we repeatedly reduce the number of symbols we are trying to encode. In particular, when we make a reduction like combining D and F into a single symbol, we will represent this single symbol as a small decision tree rather than using a letter like E. For example, after combining D and F as described above, we would use the table shown in Figure 3.14 to describe the four letter code rather than a table like the one we showed earlier in Figure 3.11.

Grade	Probability
A	0.20
B	0.29
C	0.25
	0.26

Figure 3.14: First step in application of Huffman algorithm to letter grades

Grade	Probability
	0.45
B	0.29
	0.26

Figure 3.15: Second step in application of Huffman algorithm to letter grades

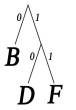
Grade	Probability
	0.45
	0.55

Figure 3.16: Third step in application of Huffman algorithm to letter grades

At each step in the algorithm, we will simplify the problem by replacing the two symbols that are least likely to be used in a message with a single symbol. Given this rule and the percentages shown in Figure 3.14, we can see that the two symbols that needed to be combined next are A and C. A table showing the three-symbol coding problem that results is shown in Figure 3.15. Note that the probability associated with the symbol that replaces A and C is the sum of the probabilities that had been associated with A and C, 0.20 and 0.25.

For the next step, we combine B and the tree fragment that represents D and F, because the probabilities associated with these two symbols are the two smallest remaining probabilities. When we combine a symbol that is already represented by a fragment of a decision tree with another symbol, we simply make the existing tree fragment a leaf in the tree that represent the new reduced symbol. As a result, the table shown in Figure 3.16. is used to represent the result of this step in the algorithm.

We can conclude the process by constructing the tree that would describe the single symbol that would represent all five of the original letter grades. This tree is shown in Figure 3.16. The

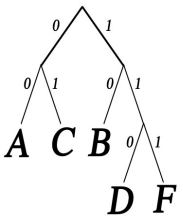


Figure 3.17: Huffman code decision tree for letter grades

Grade	Codeword
A	00
B	10
C	01
D	110
F	111

Figure 3.18: Huffman Code for five letter grades

code described by this tree is shown in Figure 3.18. This code is optimal in the sense that the average number of binary digits that will be required to encode messages with the given probability distribution will be less than or equal to that required for any other code.

We will not attempt to provide a complete justification for the claim that Huffman codes are optimal. We will, however, mention one fact that helps explain why this is true. Consider the longest codewords used in the three coding schemes we presented in Figure 3.5. You will notice that they come in pairs. In scheme (a), the longest codewords are those used for D and F, 110 and 110. These form a pair in the sense that they share the prefix 11. The only difference between the two members of this pair is that one ends with 0 and the other ends with 1. In scheme (c), the codes for D and F behave in the same way, but they are both four digits long. In scheme (b), there are two pairs instead of just one. The codes for B and C both start with 00, and the codes for D and F both start with 01.

It turns out that in any variable-length, prefix-free, binary coding scheme that is optimal, the longest codewords used will come in such pairs. That is, for each codeword that is of maximal length there will be another codeword of the same length that is identical except for its last digit. Suppose for example, that in some coding scheme, 0110010 is a maximal length codeword. Then there must be another symbol that is assigned the codeword 0110011 within the scheme. Otherwise, you could simply drop the last digit from 0110010 and obtain a more efficient code. Huffman's algorithm exploits this fact. This is why at each step we combine the two least common symbols rather than the most common symbols or any other pair. The fact that we know that the codes associated with the least common symbol will be of the same length is what makes it possible to derive codewords for these two symbols from the codeword used for the combined symbol that represented them in the reduced problem.

Exercise 3.6.1 *Which of the following sets of binary codes could not be a Huffman code for any message? Explain/justify your answer.*

a. *11, 101, 100, 01, 000*

b. *0, 10, 110, 1110, 1111*

c. *10, 01, 010, 110, 111*

Exercise 3.6.2 *In this question, you are asked to construct some Huffman codes for a specific message.*

a. *Construct a Huffman code for the message "bananarama". Show both the tree you construct and the binary codes used for each of the 5 symbols in the message. How many bits are*

required to represent the message using this code? (Just the message. Not the description of the code itself.)

b. *You might have noticed that at some points in the process of constructing the first tree, you had the ability to choose several different sets of characters to merge. See what happens if you choose differently. In particular, try to make choices that produce a Huffman tree with a different structure than the tree you constructed for part a. Show both the tree you construct and the binary codes used for each of the 5 symbols in the message. How many bits are required to represent the message bananarama" using this code?*

Exercise 3.6.3 *For this question, we would like you to construct a Huffman code given information about the probabilities of letter occurrences, rather than an actual message.*

a. *Give a Huffman code for messages composed of the six symbols a, b, c, d, e, and f assuming that the letters appear in messages with the following probabilities:*

Letter	*Probability*	*Letter*	*Probability*	*Letter*	*Probability*
a	*7/30*	*b*	*10/30*	*c*	*3/30*
d	*4/30*	*e*	*4/30*	*f*	*2/30*

b. *What is the expected number of bits per symbol when the code you constructed is used to encode messages with the given distribution?*

Exercise 3.6.4 *For this question, we would like you to investigate an alternative to Huffman codes known as Shannon-Fano codes. The Huffman algorithm builds a tree from the bottom up. It starts by merging single letters into pairs. Then, it repeatedly joins small collections of letters to form larger collections. The Shannon-Fano algorithm works instead from the top down. It starts by splitting the letters into two sets, then splits each of these sets into smaller sets until eventually all the letters have been split into sets of their own. Each split leads to the construction of a branch in the tree. When the algorithm splits a set of letters into two smaller subsets it first sorts the letters in decreasing order by their frequencies and second, finds the point in the sorted list where the sum of the frequencies on one side roughly equals the sum of the frequencies on the other side. We give a complete description of the Shannon-Fano algorithm along with an example execution of it on the message "bananarama" below homework.*

Sometimes the Shannon-Fano algorithm produces a code that is as efficient as the Huffman code, but not always. We want you to provide an example message where the code produced by the Huffman algorithm is more efficient than the code produced by the Shannon-Fano algorithm. Show both the Shannon-Fano code and the Huffman code for this message and indicate how many bits are required by each scheme.

Here is a formal description of the Shannon-Fano algorithm.

a. *Order all the symbols in the set by their frequency.*

b. *If the set is a single symbol then produce the "tree" consisting of just that symbol.*

c. *Otherwise, split the ordered list of symbols into two subsets X and Y such that*

(a) all of the letters in X occur before the letters in Y,

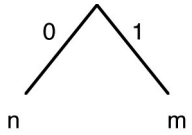
(b) *the sum of the frequencies of the letters in X is at least the sum of the frequencies of the letters in Y, but*

(c) *the size of X is as small as it can be subject to (b).*

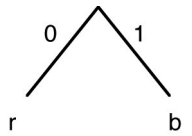
d. *Now generate trees for X and Y independently using the same algorithm. Build a tree by joining the two subtrees that result as branches under a single tree node. Label the branch for the first subset as 0 and the branch for the second subset as 1.*

For clarity, we illustrate an application of the Shannon-Fano algorithm to the example message: bananarama.

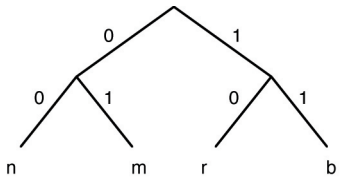
- *Step 1 produces the list $\{a, n, m, r, b\}$ since a appears most frequently and b, r, and m appear once each.*
- *Step 2 does not apply at this point since the set $\{a, n, m, r, b\}$ contains 5 symbols.*
- *The first letter in the list $\{a, n, m, r, b\}$, the single symbol a, has frequency 5. The frequencies of the remaining symbols also sum to 5 so following step 3 we split the list of 5 symbols into the subset $\{a\}$ and the subset $\{n, m, r, b\}$.*
- *As stated in step 4, we now apply the steps over again – first to build a tree for the subset $\{a\}$ and then to build one for the subset $\{n, m, r, b\}$.*
- *Building a tree for $\{a\}$ is handled by step 2. Since there is only one symbol, the tree is trivial. It just consists of the symbol a.*
- *To build a tree for the list $\{n, m, r, b\}$ we start with step 1. Luckily, the list is already in order and it is not of length 1. Therefore, we skip to step 3. This calls for us to divide the list up into two subsets again. This time, the first subset will be $\{n, m\}$ since it has total frequency 3 which is just more than half of the total frequency of 5 for the set $\{n, m, r, b\}$.*
- *Now, we go through the algorithm again for the sets $\{n, m\}$ and $\{r, b\}$. Luckily, sets of length two are handled fairly simple. The only way a set of two items can be divided into two subsets that satisfy the conditions in step 3 is to place one symbol in each subset. For example, $\{n, m\}$ is broken up into $\{n\}$ and $\{m\}$. Each of these lists falls under step 2. So we end up with two trivial trees. Step 4 combines them under a branch with the 0 edge going to n and the 1 edge going toward m:*



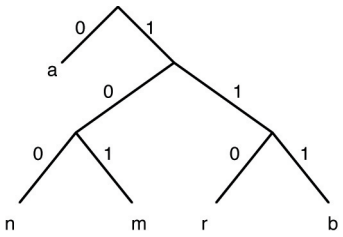
Similarly, the list $\{r, b\}$ will result in the building of the tree:



Now, we get to apply Step 4 again to these two trees, since they are the trees built from the prefix and suffix of the list $\{n, m, r, b\}$*. The result is to build the tree:*



Finally, we follow Step 4 one last time to combine the trivial tree built for the prefix $\{a\}$ *with the tree we just obtained for* $\{n, m, r, b\}$ *to obtain the complete tree:*



Notice that the the Shannon-Fano tree for bananarama *is very similar to the Huffman tree you may have created in Exercise 3.6.2. But, as we note above, this is not always the case.*