NCS-085 Data Compression Syllabus (DC) Syllabus

Unit–I

Compression Techniques: Loss less compression, Lossy Compression, Measures of performance, Modeling and coding, Mathematical Preliminaries for Lossless compression: A brief introduction to information theory, Models: Physical models, Probability models, Markov models, composite source model, Coding: uniquely decodable codes, Prefix codes.

Unit-II

The Huffman coding algorithm: Minimum variance Huffman codes, Adaptive Huffman coding: Update procedure, Encoding procedure, Decoding procedure. Golomb codes, Rice codes, Tunstall codes, Applications of Hoffman coding: Loss less image compression, Text compression, Audio Compression

Unit–III

Coding a sequence, Generating a binary code, Comparison of Binary and Huffman coding, Applications: Bi-level image compression-The JBIG standard, JBIG2, Image compression. Dictionary Techniques: Introduction, Static Dictionary: Diagram Coding, Adaptive Dictionary. The LZ77 Approach, The LZ78 Approach, Applications: File Compression-UNIX compress, Image Compression: The Graphics Interchange Format (GIF), Compression over Modems: V.42 bits, Predictive Coding: Prediction with Partial match (ppm): The basic algorithm, The ESCAPE SYMBOL, length of context, The Exclusion Principle, The Burrows-Wheeler Transform: Moveto-front coding, CALIC, JPEG-LS, Multi-resolution Approaches, Facsimile Encoding, Dynamic Markoy Compression.

Unit–IV

Distortion criteria, Models, Scalar Ouantization: The Quantization problem, Uniform Quantizer, Adaptive Quantization, Non uniform Quantization.

U nit –V

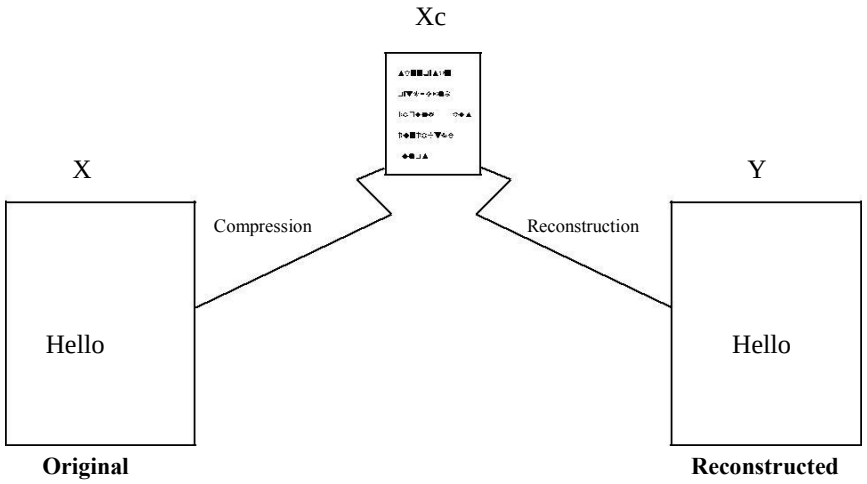
Advantages of Vector Quantization over Scalar Quantization, The Linde-Buzo-Gray Algorithm, Tree structured Vector Quantizers. Structured VectorQuantizers

REFERENCES: 1. Khalid Sayood, Introduction to Data Compression, Morgan Kaufmann Publishers
2. Elements of Data Compression,Drozdek, Cengage Learning
3.Introduction to Data Compression, Second Edition,KhalidSayood,The Morgan aufmannSeries
4.Data Compression: The Complete Reference 4th Edition byDavid Salomon, Springer
5.Text Compression1st Edition by Timothy C. Bell Prentice Hall

UNIT-I

Compression Techniques:

When we speak of a compression technique or compression algorithm we are actually referring to two algorithms. There is the compression algorithm that takes an input X and generates a representation X_c that requires fewer bits, and there is a reconstruction algorithm that operates on the compressed representation X_c to generate the reconstruction Y. These operations are shown schematically in Figure.We will follow convention and refer to both the compression and reconstruction algorithms together to mean the compression algorithm.



Compression and Reconstruction

Based on the requirements of reconstruction, data compression schemes can be divided into two broad classes: lossless compression schemes, in which Y is identical to X , and lossy compression schemes, which generally provide much higher compression than lossless compression but allow Y to be different from X .

Lossless Compression:

Lossless compression techniques, as their name implies, involve no loss of information. If data have been losslessly compressed, the original data can be recovered exactly from the compressed data. Lossless compression is generally used for applications that cannot tolerate any difference between the original and reconstructed data.

Text compression is an important area for lossless compression. It is very important that the reconstruction is identical to the original text, as very small differences can result in statements with very different meanings. Consider the sentences “Do not send money” and “Do now send money.” A similar argument holds for computer files and for certain types of data such as bank records.

If data of any kind are to be processed or “enhanced” later to yield more information, it is important that the integrity be preserved. For example, suppose we compressed a radiological image in a lossy fashion, and the difference between the reconstruction Y and the original X was visually undetectable. If this image was later enhanced, the previously undetectable differences may cause the appearance of artifacts that could seriously mislead the radiologist. Because the price for this kind of mishap may be a human life, it makes sense to be very careful about using a compression scheme that generates a reconstruction that is different from the original.

There are many situations that require compression where we want the reconstruction to be

identical to the original. There are also a number of situations in which it is possible to relax this requirement in order to get more compression. In these situations, we look to lossy compression techniques.

Lossy Compression:

Lossy compression techniques involve some loss of information, and data that have been compressed using lossy techniques generally cannot be recovered or reconstructed exactly. In return for accepting this distortion in the reconstruction, we can generally obtain much higher compression ratios than is possible with lossless compression.

In many applications, this lack of exact reconstruction is not a problem. For example, when storing or transmitting speech, the exact value of each sample of speech is not necessary. Depending on the quality required of the reconstructed speech, varying amounts of loss of information about the value of each sample can be tolerated. If the quality of the reconstructed speech is to be similar to that heard on the telephone, a significant loss of information can be tolerated. However, if the reconstructed speech needs to be of the quality heard on a compact disc, the amount of information loss that can be tolerated is much lower.

Similarly, when viewing a reconstruction of a video sequence, the fact that the reconstruc-tion is different from the original is generally not important as long as the differences do not result in annoying artifacts. Thus, video is generally compressed using lossy compression.

Measures of Performance:

A compression algorithm can be evaluated in a number of different ways. We could measure the relative complexity of the algorithm, the memory required to implement the algorithm, how fast the algorithm performs on a given machine, the amount of compression, and how closely the reconstruction resembles the original.

A very logical way of measuring how well a compression algorithm compresses a given set of data is to look at the ratio of the number of bits required to represent the data before compression to the number of bits required to represent the data after compression. This ratio is called the compression ratio. Suppose storing an image made up of a square array of 256×256 pixels requires 65,536 bytes. The image is compressed and the compressed version requires 16,384 bytes. We would say that the compression ratio is 4:1. We can also represent the compression ratio by expressing the reduction in the amount of data required as a percentage of the size of the original data. In this particular example, the compression ratio calculated in this manner would be 75%.

Another way of reporting compression performance is to provide the average number of bits required to represent a single sample. This is generally referred to as the *rate*. For example, in the case of the compressed image described above, if we assume 8 bits per byte (or pixel), the average number of bits per pixel in the compressed representation is 2. Thus, we would say that the rate is 2 bits per pixel.

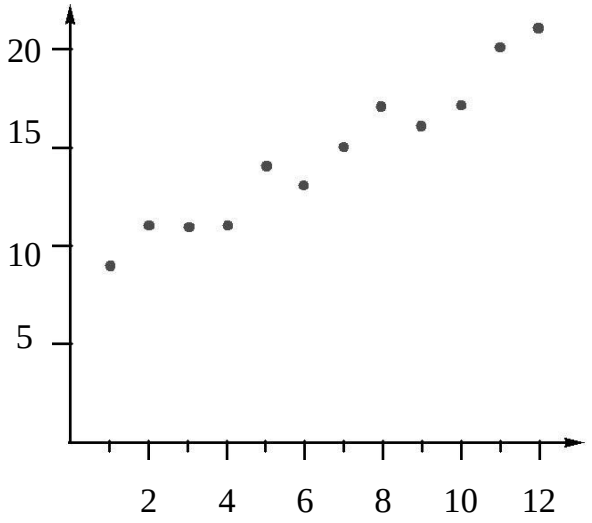
In lossy compression, the reconstruction differs from the original data. Therefore, in order to determine the efficiency of a compression algorithm, we have to have some way of quantifying the difference. The difference between the original and the reconstruction is often called the distortion. Lossy techniques are generally used for the compression of data that originate as analog signals, such as speech and video. In compression of speech and video, the final arbiter of quality is human. Because human responses are difficult to model mathematically, many approximate measures of distortion are used to determine the quality of the reconstructed waveforms.

Other terms that are also used when talking about differences between the reconstruction and the original are fidelity and quality. When we say that the fidelity or quality of a reconstruction is high, we mean that the difference between the reconstruction and the original is small. Whether this difference is a mathematical difference or a perceptual difference should be evident from the

context.

Modeling and Coding:

The development of data compression algorithms for a variety of data can be divided into two phases. The first phase is usually referred to as modeling. In this phase, we try to extract information about any redundancy that exists in the data and describe the redundancy in the form of a model. The second phase is called coding. A description of the model and a "description" of how the data differ from the model are encoded, generally using a binary alphabet. The difference between the data and the model is often referred to as the residual.



Sequence of Data Values

In the following three examples, we will look at three different ways that data can be modeled. We will then use the model to obtain compression.

Example -1:

Consider the following sequence of numbers $\{x_1, x_2, x_3, \ldots\}$:

9	11	11	11	14	13	15	17	16	17	20	21

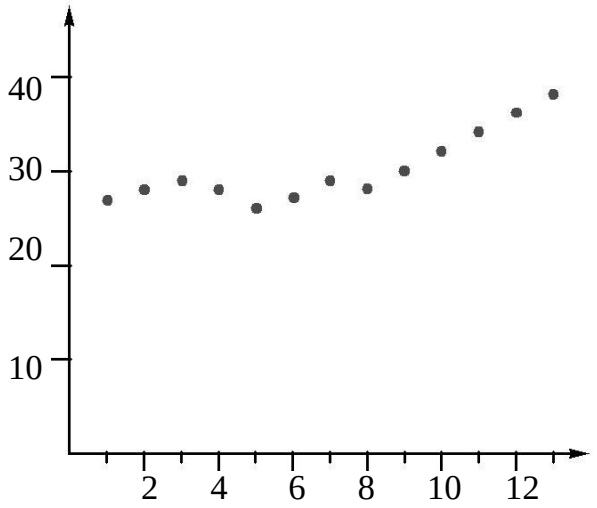
If we were to transmit or store the binary representations of these numbers, we would need to use 5 bits per sample. However, by exploiting the structure in the data, we can represent the sequence using fewer bits. If we plot these data as shown in Figure, we see that the data seem to fall on a straight line. A model for the data could, therefore, be a straight line given by the equation

$$\hat{x}_n = n + 8 \quad n = 1, 2, \ldots$$

The structure in this particular sequence of numbers can be characterized by an equation. Thus, $\hat{x}_1 = 9$, while $x_1 = 9$, $\hat{x}_2 = 10$, while $x_2 = 11$, and so on. To make use of this structure, let's examine the difference between the data and the model. The difference (or residual) is given by the sequence

$$e_n = x_n - \hat{x}_n : 010 - 11 - 101 - 1 - 111$$

The residual sequence consists of only three numbers $\{-1, 0, 1\}$. If we assign a code of 00 to -1, a code of 01 to 0, and a code of 10 to 1, we need to use 2 bits to represent each element of the residual sequence. Therefore, we can obtain compression by transmitting or storing the parameters of the model and the residual sequence. The encoding can be exact if the required compression is to be lossless, or approximate if the compression can be lossy.



The type of structure or redundancy that existed in these data follows a simple law. Once we recognize this law, we can make use of the structure to *predict* the value of each element in the sequence and then encode the residual. Structure of this type is only one of many types of structure.

Example-2 :

Consider the following sequence of numbers:

27	28	29	28	26	27	29	28	30	32	34	36	38

The sequence is plotted in Figure.

The sequence does not seem to follow a simple law as in the previous case. However, each value in this sequence is close to the previous value. Suppose we send the first value, then in place of subsequent values we send the difference between it and the previous value. The sequence of transmitted values would be

27	1	1	-1	-2	1	2	-1	2	2	2	2	2

Like the previous example, the number of distinct values has been reduced. Fewer bits are required to represent each number, and compression is achieved. The decoder adds each received value to the previous decoded value to obtain the reconstruction corresponding to the received value. Techniques that use the past values of a sequence to predict the current value and then encode the error in prediction, or residual, are called predictive coding schemes.

Example-3 :

Suppose we have the following sequence:

abbarrayaran/barray/bran/b/ f arb/ f aarb/ f aaarba/way

which is typical of all sequences generated by a source (*b/* denotes a blank space). Notice that the sequence is made up of eight different symbols. In order to represent eight symbols, we need to use 3 bits per symbol. Suppose instead we used the code shown in Table. Notice that we have assigned a codeword with only a single bit to the symbol that occurs most often (*a*) and correspondingly longer codewords to symbols that occur less often. If we substitute the codes for each symbol, we will use 106 bits to encode the entire sequence. As there are 41 symbols in the sequence, this works out to approximately 2.58 bits per symbol. This means we have obtained a compression ratio of 1.16:1.

TABLE	A code with codewords of varying length.
a	1
b/	001
b	01100
f	0100
n	0111
r	000
w	01101
y	0101

When dealing with text, along with statistical redundancy, we also see redundancy in the form of words that repeat often. We can take advantage of this form of redundancy by constructing a list of these words and then representing them by their position in the list. This type of compression scheme is called a dictionary compression scheme. Often the structure or redundancy in the data becomes more evident when we look at groups of symbols. Finally, there will be situations in which it is easier to take advantage of the structure if we decompose the data into a number of components. We can then study each component separately and use a model appropriate to that component.

Mathematical Preliminaries for Lossless compression:

A brief introduction to information theory:

Shannon defined a quantity called self-information. Suppose we have an event *A*, which is a set of outcomes of some random experiment. If P (A) is the probability that the event *A* will occur, then the information associated with A is given by

$$i(A) = \log_b \frac{1}{P(A)} = -\log_b P(A)$$

Note that we have not specified the base *b* of the log function. We will discuss the choice of the base later in this section. The use of the logarithm to obtain a measure of information was not an arbitrary choice as we shall see in Section. But first let's see if the use of a logarithm in this context makes sense from an intuitive point of view. Recall that log(1)= 0, and −log*(x)* increases as *x* decreases from one to zero. Therefore, if the probability of an event is low, the amount of self-information associated with it is high; if the probability of an event is high, the information associated with it is low. Even if we ignore the mathematical definition of information and simply use the definition we use in everyday language, this makes some intuitive sense. The barking of a dog during a burglary is a high-probability event and, therefore, does not contain too much information. However, if the dog did not bark during a burglary, this is a low-probability event and contains a lot of information. Although this equivalence of the mathematical and semantic definitions of information holds true most of the time, it does not hold all of the time. For example, a totally random string of letters will contain more information than a well-thought-out treatise on information theory.

Another property of this mathematical definition of information that makes intuitive sense is that the information obtained from the occurrence of two independent events is the sum of the information obtained from the occurrence of the individual events. Suppose *A* and *B* are two independent events. The self-information associated with the occurrence of both event A and event B.

The unit of information depends on the base of the log. If we use log base 2, the unit is *bits*; if we use log base *e*, the unit is nats and if we use log base 10, the unit is hartleys. In general, if we do not

explicitly specify the base of the log we will be assuming a base of 2.

Because the logarithm base 2 probably does not appear on your calculator, let's briefly review logarithms. Recall that

$$\log_b x = a$$

means that

$$b^a = x$$

Therefore, if we want to take the log base 2 of x

$$\log_2 x = a \Rightarrow 2^a = x$$

we want to find the value of a. We can take the natural log (log base e), which we will write as ln, or log base 10 of both sides (which do appear on your calculator). Then

$$\ln(2^a) = \ln x \Rightarrow a \ln 2 = \ln x$$

Example :

Let H and T be the outcomes of flipping a coin. If the coin is fair, then

$$P(H) = P(T) = \frac{1}{2}$$

and

$$i(H) = i(T) = 1 \text{ bit}$$

If the coin is not fair, then we would expect the information associated with each event to be different. Suppose

$$P(H) = \frac{1}{8}, \quad P(T) = \frac{7}{8}$$

Then

$$i(H) = 3 \text{ bits}, \quad i(T) = 0.193 \text{ bits}$$

At least mathematically, the occurrence of a head conveys much more information than the occurrence of a tail. As we shall see later, this has certain consequences for how the information conveyed by these outcomes should be encoded.

Models:

Having a good model for the data can be useful in estimating the entropy of the source. As we will see in later chapters, good models for sources lead to more efficient compression algorithms. In general, in order to develop techniques that manipulate data using mathematical operations, we need to have a mathematical model for the data. Obviously, the better the model, the more likely it is that we will come up with a satisfactory technique. There are several approaches to building mathematical models.

Physical Models:

If we know something about the physics of the data generation process, we can use that information to construct a model. For example, in speech-related applications, knowledge about the physics of speech production can be used to construct a mathematical model for the sampled speech process. Sampled speech can then be encoded using this model.

Models for certain telemetry data can also be obtained through knowledge of the underlying process. For example, if residential electrical meter readings at hourly intervals were to be coded, knowledge about the living habits of the populace could be used to determine when electricity usage

would be high and when the usage would be low. Then instead of the actual readings, the difference (residual) between the actual readings and those predicted by the model could be coded.

In general, however, the physics of data generation is simply too complicated to understand, let alone use to develop a model. Where the physics of the problem is too complicated, we can obtain a model based on empirical observation of the statistics of the data.

Probability Models:

The simplest statistical model for the source is to assume that each letter that is generated by the source is independent of every other letter, and each occurs with the same probability. We could call this the ignorance model, as it would generally be useful only when we know nothing about the source. (Of course, that *really* might be true, in which case we have a rather unfortunate name for the model!) The next step up in complexity is to keep the independence assumption, but remove the equal probability assumption and assign a probability of occurrence to each letter in the alphabet. For a source that generates letters from an alphabet $A = \{a_1, a_2, \ldots, a_M\}$, we can have a probability model $P = \{ P(a_1), P(a_2), \ldots, P(a_M)\}$.

Given a probability model (and the independence assumption), we can compute the entropy of the source using Equation (4). As we will see in the following chapters using the probability model, we can also construct some very efficient codes to represent the letters in *A*. Of course, these codes are only efficient if our mathematical assumptions are in accord with reality.

If the assumption of independence does not fit with our observation of the data, we can generally find better compression schemes if we discard this assumption. When we discard the independence assumption, we have to come up with a way to describe the dependence of elements of the data sequence on each other.

Markov Models:

One of the most popular ways of representing dependence in the data is through the use of Markov models

For models used in lossless compression, we use a specific type of Markov process called a *discrete time Markov chain*. Let $\{x_n\}$ be a sequence of observations. This sequence is said to follow a *k*th-order Markov model if

$$P(x_n | x_{n-1}, \ldots, x_{n-k}) = P(x_n | x_{n-1}, \ldots, x_{n-k}, \ldots) \quad (13)$$

In other words, knowledge of the past *k* symbols is equivalent to the knowledge of the entire past history of the process. The values taken on by the set $\{x_{n-1}, \ldots, x_{n-k}\}$ are called the *states* of the process. If the size of the source alphabet is *l* then the number of states is l^k . The most commonly used Markov model is the first-order Markov model, for which

$$P(x_n | x_{n-1}) = P(x_n | x_{n-1}, x_{n-2}, x_{n-3}, \ldots) \quad (14)$$

Equations (13) and (14) indicate the existence of dependence between samples. However, they do not describe the form of the dependence. We can develop different first-order Markov models depending on our assumption about the form of the dependence between samples.

If we assumed that the dependence was introduced in a linear manner, we could view the data sequence as the output of a linear filter driven by white noise. The output of such a filter can be given by the difference equation

$$x_n = \rho x_{n-1} + \epsilon_n \quad (15)$$

where ϵ_n is a white noise process. This model is often used when developing coding algorithms for speech and images.

The use of the Markov model does not require the assumption of linearity. For example, consider a binary image. The image has only two types of pixels, white pixels and black pixels. We know

that the appearance of a white pixel as the next observation depends, to some extent, on whether the current pixel is white or black. Therefore, we can model the pixel process as a discrete time Markov chain. Define two states S_w and S_b (S_w would correspond to the case where the current pixel is a white pixel, and S_b corresponds to the case where the current pixel is a black pixel). We define the transition probabilities $P(w/b)$ and $P(b/w)$ and the probability of being in each state $P(S_w)$ and $P(S_b)$. The Markov model can then be represented by the state diagram shown in Figure 2.3.

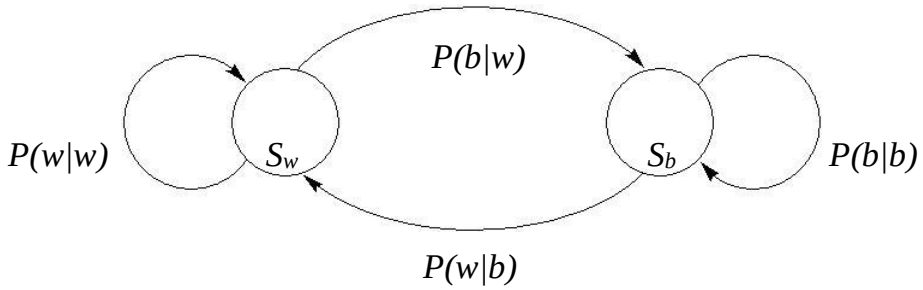
The entropy of a finite state process with states S_i is simply the average value of the entropy at each state:

$$H = \sum_{i=1}^{M} P(S_i)\, H(S_i) \qquad (16)$$

For our particular example of a binary image

$$H(S_w) = -P(b|w) \log P(b|w) - P(w|w) \log P(w|w)$$

where $P(w|w) = 1 - P(b|w)$. $H(S_b)$ can be calculated in a similar manner.



F I G U R E A two-state Markov model for binary images.

Example of Markov Model:

To see the effect of modeling on the estimate of entropy, let us calculate the entropy for a binary image, first using a simple probability model and then using the finite state model described above. Let us assume the following values for the various probabilities:

$$P(S_w) = 30/31 \quad P(S_b) = 1/31$$
$$P(w|w) = 0.99 \quad P(b|w) = 0.01 \quad P(b|b) = 0.7 \quad P(w|b) = 0.3$$

Then the entropy using a probability model and the *iid* assumption is

$$H = -0.8 \log 0.8 - 0.2 \log 0.2 = 0.206 \text{ bits}$$

Now using the Markov model

$$H(S_b) = -0.3 \log 0.3 - 0.7 \log 0.7 = 0.881 \text{ bits}$$

and

$$H(S_w) = -0.01 \log 0.01 - 0.99 \log 0.99 = 0.081 \text{ bits}$$

which, using Equation(16), results in an entropy for the Markov model of 0.107 bits, about half of the entropy

Composite Source Model:

In many applications, it is not easy to use a single model to describe the source. In such cases, we can define a composite source, which can be viewed as a combination or composition of several sources, with only one source being *active* at any given time. A composite source can be represented as a number of individual sources S_i , each with its own model M_i and a switch that selects a source S_i with probability P_i (as shown in Figure 2.4). This is an exceptionally rich model and can be used to describe some very complicated processes. We will describe this model in more detail when we need it.

Coding:

When we talk about coding we mean the assignment of binary sequences to elements of an alphabet. The set of binary sequences is called a code, and the individual members of the set are called codewords. An *alphabet* is a collection of symbols called *letters*. For example, the alphabet used in writing most books consists of the 26 lowercase letters, 26 uppercase letters, and a variety of punctuation marks. In the terminology used in this book, a comma is a letter. The ASCII code for the letter *a* is 1000011, the letter *A* is coded as 1000001, and the letter "," is coded as 0011010. Notice that the ASCII code uses the same number of bits to represent each symbol. Such a code is called a *fixed-length code*. If we want to reduce the number of bits required to represent different messages, we need to use a different number of bits to represent different symbols. If we use fewer bits to represent symbols that occur more often, on the average we would use fewer bits per symbol. The average number of bits per symbol is often called the *rate* of the code. The idea of using fewer bits to represent symbols that occur more often is the same idea that is used in Morse code: the codewords for letters that occur more frequently are shorter than for letters that occur less frequently. For example, the codeword for *E* is while the codeword for *Z* is

T A B L E Four different codes for a four-letter alphabet.

Letters	Probability	Code 1	Code 2	Code 3	Code 4
a_1	0.5	0	0	0	0
a_2	0.25	0	1	10	01
a_3	0.125	1	00	110	011
$^a4$	0.125	10	11	111	0111
Average length		1.125	1.25	1.75	1.875

Uniquely Decodable Codes:

The average length of the code is not the only important point in designing a "good" code. Consider the following example adapted from [6]. Suppose our source alphabet consists of four letters a_1, a_2, a_3, and a_4, with probabilities $P(a_1) = \frac{1}{2}$, $P(a_2) = \frac{1}{4}$, and $P(a_3) = P(a_4) = \frac{1}{8}$.

The average length l for each code is given by

$$l = \sum_{i=1}^{4} P(a_i)n(a_i)$$

where $n(a_i)$ is the number of bits in the codeword for letter a_i and the average length is given in bits/symbol. Based on the average length, Code 1 appears to be the best code. However, to be

useful, a code should have the ability to transfer information in an unambiguous manner. This is obviously not the case with Code 1. Both a_1 and a_2 have been assigned the codeword 0. When a 0 is received, there is no way to know whether an a_1 was transmitted or an a_2. We would like each symbol to be assigned a *unique* codeword.

At first glance, Code 2 does not seem to have the problem of ambiguity; each symbol is assigned a distinct codeword. However, suppose we want to encode the sequence $a_2\ a_1\ a_1$. Using Code 2, we would encode this with the binary string 100. However, when the string 100 is received at the decoder, there are several ways in which the decoder can decode this string. The string 100 can be decoded as $a_2a_1a_1$, or as a_2a_3. This means that once a sequence is encoded with Code 2, the original sequence cannot be recovered with certainty. In general, this is not a desirable property for a code. We would like *unique decodability* from the code; that is, any given sequence of codewords can be decoded in one, and only one, way.

We have already seen that Code 1 and Code 2 are not uniquely decodable. How about Code 3? Notice that the first three codewords all end in a 0. In fact, a 0 always denotes the termination of a codeword. The final codeword contains no 0s and is 3 bits long. Because all other codewords have fewer than three 1s and terminate in a 0, the only way we can get three 1s in a row is as a code for a_4. The decoding rule is simple. Accumulate bits until you get a 0 or until you have three 1s. There is no ambiguity in this rule, and it is reasonably easy to see that this code is uniquely decodable. With Code 4 we have an even simpler condition. Each codeword starts with a 0, and the only time we see a 0 is in the beginning of a codeword

TABLE 2.3 Code 5. A code that is uniquely decodable but not instantaneous.

Letter	Codeword
a_1	0
a_2	01
a_3	11

Therefore, the decoding rule is to accumulate bits until you see a 0. The bit before the 0 is the last bit of the previous codeword.

There is a slight difference between Code 3 and Code 4. In the case of Code 3, the decoder knows the moment a code is complete. In Code 4, we have to wait till the beginning of the next codeword before we know that the current codeword is complete. Because of this property, Code 3 is called an *instantaneous* code. Although Code 4 is not an instantaneous code, it is almost that. While this property of instantaneous or near-instantaneous decoding is a nice property to have, it is not a requirement for unique decodability. Consider the code shown in Table 2.3. Let's decode the string 0111111111111111. In this string, the first codeword is either 0 corresponding to a_1 or 01 corresponding to a_2. We cannot tell which one until we have decoded the whole string. Starting with the assumption that the first codeword corresponds to a_1, the next eight pairs of bits are decoded as a_3. However, after decoding eight a_3s, we are left with a single (dangling) 1 that does not correspond to any codeword. On the other hand, if we assume the first codeword corresponds to a_2, we can decode the next 16 bits as a sequence of eight a_3s, and we do not have any bits left over. The string can be uniquely decoded. In fact, Code 5, while it is certainly not instantaneous, is uniquely decodable.

We have been looking at small codes with four letters or less. Even with these, it is not immediately evident whether the code is uniquely decodable or not. In deciding whether larger codes are uniquely decodable, a systematic procedure would be useful. Actually, we should include a caveat with that last statement. Later in this chapter we will include a class of variable-length codes that are always uniquely decodable, so a test for unique decodability may not be

that necessary. You might wish to skip the following discussion for now, and come back to it when you find it necessary.

Before we describe the procedure for deciding whether a code is uniquely decodable, let's take another look at our last example. We found that we had an incorrect decoding because we were left with a binary string (1) that was not a codeword. If this had not happened, we would have had two valid decodings. For example, consider the code shown in Table 2.4. Let's encode the sequence a_1 followed by eight a_3s using this code. The coded sequence is 01010101010101010. The first bit is the codeword for a_1. However, we can also decode it as the first bit of the codeword for a_2. If we use this (incorrect) decoding, we decode the next seven pairs of bits as the codewords for a_2. After decoding seven a_2s, we are left with a single 0 that we decode as a_1. Thus, the incorrect decoding is also a valid decoding, and this code is not uniquely decodable.

TABLE 2.4 Code 6. A code that is not uniquely decodable.

Letter	Codeword
a_1	0
a_2	01
a_3	10

A Test for Unique Decodability

In the previous examples, in the case of the uniquely decodable code, the binary string left over after we had gone through an incorrect decoding was not a codeword. In the case of the code that was not uniquely decodable, in the incorrect decoding what was left was a valid codeword. Based on whether the dangling suffix is a codeword or not, we get the following test [7,8].

We start with some definitions. Suppose we have two binary codewords a and b, where a is k bits long, b is n bits long, and $k < n$. If the first k bits of b are identical to a, then a is called a *prefix* of b. The last $n - k$ bits of b are called the *dangling suffix* [7]. For example, if $a = 010$ and $b = 01011$, then a is a prefix of b and the dangling suffix is 11.

Construct a list of all the codewords. Examine all pairs of codewords to see if any codeword is a prefix of another codeword. Whenever you find such a pair, add the dangling suffix to the list unless you have added the same dangling suffix to the list in a previous iteration. Now repeat the procedure using this larger list. Continue in this fashion until one of the following two things happens:

1. You get a dangling suffix that is a codeword.
2. There are no more unique dangling suffixes.

If you get the first outcome, the code is not uniquely decodable. However, if you get the second outcome, the code is uniquely decodable.

Let's see how this procedure works with a couple of examples.

Example 1:

Consider Code 5. First list the codewords:

$$\{0, 01, 11\}$$

The codeword 0 is a prefix for the codeword 01. The dangling suffix is 1. There are no other pairs for which one element of the pair is the prefix of the other. Let us augment the codeword list with the dangling suffix:

{0, 01, 11, 1}

Comparing the elements of this list, we find 0 is a prefix of 01 with a dangling suffix of 1. But we have already included 1 in our list. Also, 1 is a prefix of 11. This gives us a dangling suffix of 1, which is already in the list. There are no other pairs that would generate a dangling suffix, so we cannot augment the list any further. Therefore, Code 5 is uniquely decodable.

Example 2:

Consider Code 6. First list the codewords:

{0, 01, 10}

The codeword 0 is a prefix for the codeword 01. The dangling suffix is 1. There are no other pairs for which one element of the pair is the prefix of the other. Augmenting the codeword list with 1, we obtain the list

{0, 01, 10, 1}

In this list, 1 is a prefix for 10. The dangling suffix for this pair is 0, which is the codeword for a_1. Therefore, Code 6 is not uniquely decodable.

Prefix Codes:

The test for unique decodability requires examining the dangling suffixes initially generated by codeword pairs in which one codeword is the prefix of the other. If the dangling suffix is itself a codeword, then the code is not uniquely decodable. One type of code in which we will never face the possibility of a dangling suffix being a codeword is a code in which no codeword is a prefix of the other. In this case, the set of dangling suffixes is the null set, and we do not have to worry about finding a dangling suffix that is identical to a codeword. A code in which no codeword is a prefix to another codeword is called a prefix code. A simple way to check if a code is a prefix code is to draw the rooted binary tree corresponding to the code. Draw a tree that starts from a single node (the root node) and has a maximum of two possible branches at each node. One of these branches corresponds to a 1 and the other branch corresponds to a 0. In this book, we will adopt the convention that when we draw a tree with the root node at the top, the left branch corresponds to a 0 and the right branch corresponds to a 1.

Note that apart from the root node, the trees have two kinds of nodes , nodes that give rise to other nodes and nodes that do not. The first kind of nodes are called internal nodes