Introduction to Data Compression*

Guy E. Blelloch

September 18, 2000

Contents

1 Introduction **2**

2 Information Theory **4**
- 2.1 Entropy . . . 4
- 2.2 The Entropy of the English Language . . . 5
- 2.3 Conditional Entropy and Markov Chains . . . 7

3 Probability Coding **8**
- 3.1 Prefix Codes . . . 9
  - 3.1.1 Relationship to Entropy . . . 10
- 3.2 Huffman Codes . . . 12
  - 3.2.1 Combining Messages . . . 13
  - 3.2.2 Minimum Variance Huffman Codes . . . 14
- 3.3 Arithmetic Coding . . . 15
  - 3.3.1 Integer Implementation . . . 18

4 Applications of Probability Coding **22**
- 4.1 Run-length Coding . . . 22
- 4.2 Move-To-Front Coding . . . 24
- 4.3 Residual Coding: JPEG-LS . . . 25
- 4.4 Context Coding: JBIG . . . 26
- 4.5 Context Coding: PPM . . . 27

5 The Lempel-Ziv Algorithms **30**
- 5.1 Lempel-Ziv 77 (Sliding Windows) . . . 30
- 5.2 Lempel-Ziv-Welch . . . 32

6 Other Lossless Compression **35**
- 6.1 Burrows Wheeler . . . 35

*This is an early draft of a chapter of a book I'm starting to write on "algorithms in the real world". There are surely many mistakes, and **please feel free to point them out**. General comments are also welcome. Currently this document only goes into lossless compression—I'm working on the lossy compression sections.

1 Introduction

Compression is used just about everywhere. All the images you get on the web are compressed, typically in the JPEG or GIF formats, most modems use compression, HDTV will be compressed using MPEG-2, and several file systems automatically compress files when stored, and the rest of us do it by hand. The neat thing about compression, as with the other topics we will cover in this course, is that the algorithms used in the real world make heavy use of a wide set of algorithmic tools, including sorting, hash tables, tries, and FFTs. Furthermore, algorithms with strong theoretical foundations play a critical role in real-world applications.

In this chapter we will use the generic term *message* for the objects we want to compress, which could be either files or messages. The task of compression consists of two components, an *encoding* algorithm that takes a message and generates a "compressed" representation (hopefully with fewer bits), and a *decoding* algorithm that reconstructs the original message or some approximation of it from the compressed representation. These two components are typically intricately tied together since they both have to understand the shared compressed representation.

We distinguish between *lossless algorithms*, which can reconstruct the original message exactly from the compressed message, and *lossy algorithms*, which can only reconstruct an approximation of the original message. Lossless algorithms are typically used for text, and lossy for images and sound where a little bit of loss in resolution is often undetectable, or at least acceptable. Lossy is used in an abstract sense, however, and does not mean random lost pixels, but instead means loss of a quantity such as a frequency component, or perhaps loss of noise. For example, one might think that lossy text compression would be unacceptable because they are imagining missing or switched characters. Consider instead a system that reworded sentences into a more standard form, or replaced words with synonyms so that the file can be better compressed. Technically the compression would be lossy since the text has changed, but the "meaning" and clarity of the message might be fully maintained, or even improved. In fact Strunk and White might argue that good writing is the art of lossy text compression.

Is there a lossless algorithm that can compress all messages? There has been at least one patent application that claimed to be able to compress all files (messages)—Patent 5,533,051 titled "Methods for Data Compression". The patent application claimed that if it was applied recursively, a file could be reduced to almost nothing. With a little thought you should convince yourself that this is not possible, at least if the source messages can contain any bit-sequence. We can see this by a simple counting argument. Lets consider all 1000 bit messages, as an example. There are 2^{1000} different messages we can send, each which needs to be distinctly identified by the decoder. It should be clear we can't represent that many different messages by sending 999 or fewer bits for all the messages — 999 bits would only allow us to send 2^{999} distinct messages. The truth is that if any one message is shortened by an algorithm, then some other message needs to be lengthened. You can verify this in practice by running GZIP on a GIF file. It is, in fact, possible to go further and show that for a set of input messages of fixed length, if one message is compressed, then the average length of the compressed messages over all possible inputs is always going to be longer than the original input messages. Consider, for example, the 8 possible 3 bit messages. If one is compressed to two bits, it is not hard to convince yourself that two messages will have to expand to 4 bits, giving an average of 3 1/8 bits. Unfortunately, the patent was granted.

Because one can't hope to compress everything, all compression algorithms must assume that there is some bias on the input messages so that some inputs are more likely than others, *i.e.* that there is some unbalanced probability distribution over the possible messages. Most compression algorithms base this "bias" on the structure of the messages – *i.e.*, an assumption that repeated characters are more likely than random characters, or that large white patches occur in "typical" images. Compression is therefore all about probability.

When discussing compression algorithms it is important to make a distinction between two components: the model and the coder. The *model* component somehow captures the probability distribution of the messages by knowing or discovering something about the structure of the input. The *coder* component then takes advantage of the probability biases generated in the model to generate codes. It does this by effectively lengthening low probability messages and shortening high-probability messages. A model, for example, might have a generic "understanding" of human faces knowing that some "faces" are more likely than others (*e.g.*, a teapot would not be a very likely face). The coder would then be able to send shorter messages for objects that look like faces. This could work well for compressing teleconference calls. The models in most current real-world compression algorithms, however, are not so sophisticated, and use more mundane measures such as repeated patterns in text. Although there are many different ways to design the model component of compression algorithms and a huge range of levels of sophistication, the coder components tend to be quite generic—in current algorithms are almost exclusively based on either Huffman or arithmetic codes. Lest we try to make to fine of a distinction here, it should be pointed out that the line between model and coder components of algorithms is not always well defined.

It turns out that information theory is the glue that ties the model and coder components together. In particular it gives a very nice theory about how probabilities are related to information content and code length. As we will see, this theory matches practice almost perfectly, and we can achieve code lengths almost identical to what the theory predicts.

Another question about compression algorithms is how does one judge the quality of one versus another. In the case of lossless compression there are several criteria I can think of, the time to compress, the time to reconstruct, the size of the compressed messages, and the generality—*i.e.*, does it only work on Shakespeare or does it do Byron to. In the case of lossy compression the judgement is further complicated since we also have to worry about how good the lossy approximation is. There are typically tradeoffs between the amount of compression, the runtime, and the quality of the reconstruction. Depending on your application one might be more important than another and one would want to pick your algorithm appropriately. Perhaps the best attempt to systematically compare lossless compression algorithms is the Archive Comparison Test (ACT) by Jeff Gilchrist. In reports times and compression ratios for 100s of compression algorithms over many databases. It also gives a score based on a weighted average of runtime and the compression ratio.

This chapter will be organized by first covering some basics of information theory. Section 3 then discusses the coding component of compressing algorithms and shows how coding is related to the information theory. Section 4 discusses various models for generating the probabilities needed by the coding component. Section 5 describes the Lempel-Ziv algorithms, and Section 6 covers other lossless algorithms (currently just Burrows-Wheeler).

2 Information Theory

2.1 Entropy

Shannon borrowed the definition of *entropy* from statistical physics to capture the notion of how much information is contained in a and their probabilities. For a set of possible messages S, Shannon defined entropy[1] as,

$$H(S) = \sum_{s \in S} p(s) \log_2 \frac{1}{p(s)}$$

where $p(s)$ is the probability of message s. The definition of Entropy is very similar to that in statistical physics—in physics S is the set of possible states a system can be in and $p(s)$ is the probability the system is in state s. We might remember that the second law of thermodynamics basically says that the entropy of a system and its surroundings can only increase.

Getting back to messages, if we consider the individual messages $s \in S$, Shannon defined the notion of the *self information* of a message as

$$i(s) = \log_2 \frac{1}{p(s)} \ .$$

This self information represents the number of bits of information contained in it and, roughly speaking, the number of bits we should use to send that message. The equation says that messages with higher probability will contain less information (*e.g.*, a message saying that it will be sunny out in LA tomorrow is less informative than one saying that it is going to snow).

The entropy is simply a weighted average of the information of each message, and therefore the average number of bits of information in the set of messages. Larger entropies represent more information, and perhaps counter-intuitively, the more random a set of messages (the more even the probabilities) the more information they contain on average.

Here are some examples of entropies for different probability distributions over five messages.

$$\begin{aligned}
p(S) &= \{0.25, 0.25, 0.25, 0.125, 0.125\} \\
H &= 3 \times 0.25 \times \log_2 4 + 2 \times 0.125 \times \log_2 8 \\
&= 1.5 + 0.75 \\
&= 2.25
\end{aligned}$$

$$p(s) = \{0.5, 0.125, 0.125, 0.125, 0.125\}$$

[1]Technically this definition is for *first-order* Entropy. We will get back to the general notion of Entropy.

	bits/char
bits $\lceil\log(96)\rceil$	7
entropy	4.5
Huffman Code (avg.)	4.7
Entropy (Groups of 8)	2.4
Asymptotically approaches:	1.3
Compress	3.7
Gzip	2.7
BOA	2.0

Table 1: Information Content of the English Language

$$\begin{aligned} H &= 0.5 \times \log_2 2 + 4 \times 0.125 \times \log_2 8 \\ &= 0.5 + 1.5 \\ &= 2 \end{aligned}$$

$$\begin{aligned} p(s) &= \{0.75, 0.0625, 0.0625, 0.0625, 0.0625\} \\ H & \quad = 0.75 \times \log_2(\frac{4}{3}) + 4 \times 0.0625 \times \log_2 16 \\ &= 0.3 + 1 \\ &= 1.3 \end{aligned}$$

Note that the more uneven the distribution, the lower the Entropy.

Why is the logarithm of the inverse probability the right measure for self information of a message? Although we will relate the self information and entropy to message length more formally in Section 3 lets try to get some intuition here. First, for a set of $n = 2^i$ equal probability messages, the probability of each is $1/n$. We also know that if all are the same length, then $\log_2 n$ bits are required to encode each message. Well this is exactly the self information since $i(S_i) = \log_2 \frac{1}{p_i} = \log_2 n$. Another property of information we would like, is that the information given by two independent messages should be the sum of the information given by each. In particular if messages A and B are independent, the probability of sending one after the other is $p(A)p(B)$ and the information contained is them is

$$i(AB) = \lg \frac{1}{p(A)p(B)} = \lg \frac{1}{p(A)} + \lg \frac{1}{p(A)} = i(A) + i(B) \ .$$

The logarithm is the "simplest" function that has this property.

2.2 The Entropy of the English Language

We might be interested in how much information the English Language contains. This could be used as a bound on how much we can compress English, and could also allow us to compare the density (information content) of different languages.

One way to measure the information content is in terms of the average number of bits per character. Table 1 shows a few ways to measure the information of English in terms of bits-

Date	bpc	scheme	authors
May 1977	3.94	LZ77	Ziv, Lempel
1984	3.32	LZMW	Miller and Wegman
1987	3.30	LZH	Brent
1987	3.24	MTF	Moffat
1987	3.18	LZB	Bell
.	2.71	GZIP	.
1988	2.48	PPMC	Moffat
.	2.47	SAKDC	Williams
Oct 1994	2.34	PPM*	Cleary, Teahan, Witten
1995	2.29	BW	Burrows, Wheeler
1997	1.99	BOA	Sutton
1999	1.89	RK	Taylor

Table 2: Lossless compression ratios for text compression on Calgary Corpus

per-character. If we assume equal probabilities for all characters, a separate code for each character, and that there are 96 printable characters (the number on a standard keyboard) then each character would take $\lceil \log 96 \rceil = 7$ bits. The entropy assuming even probabilities is $\log 96 = 6.6$ bits/char. If we give the characters a probability distribution (based on a corpus of English text) the entropy is reduced to about 4.5 bits/char. If we assume a separate code for each character (for which the Huffman code is optimal) the number is slightly larger 4.7 bits/char.

Note that so far we have not taken any advantage of relationships among adjacent or nearby characters. If you break text into blocks of 8 characters, measure the entropy of those blocks (based on measuring their frequency in an English corpus) you get an entropy of about 19 bits. When we divide this by the fact we are coding 8 characters at a time, the entropy (bits) per character is 2.4. If we group larger and larger blocks people have estimated that the entropy would approach 1.3 (or lower). It is impossible to actually measure this because there are too many possible strings to run statistics on, and no corpus large enough.

This value 1.3 bits/char is an estimate of the information content of the English language. Assuming it is approximately correct, this bounds how much we can expect to compress English text if we want lossless compression. Table 1 also shows the compression rate of various compressors. All these, however, are general purpose and not designed specifically for the English language. The last one, BOA, is the current state-of-the-art for general-purpose compressors. To reach the 1.3 bits/char the compressor would surely have to "know" about English grammar, standard idioms, etc..

A more complete set of compression ratios for the Calgary corpus for a variety of compressors is shown in Table 2. The Calgary corpus is a standard benchmark for measuring compression ratios and mostly consists of English text. In particular it consists of 2 books, 5 papers, 1 bibliography, 1 collection of news articles, 3 programs, 1 terminal session, 2 object files, 1 geophysical data, and 1 bit-map b/w image. The table shows how the state of the art has improved over the years.

2.3 Conditional Entropy and Markov Chains

Often probabilities of events (messages) are dependent on the context in which they occur, and by using the context it is often possible to improve our probabilities, and as we will see, reduce the entropy. The context might be the previous characters in text (see PPM in Section 4.5), or the neighboring pixels in an image (see JBIG in Section 4.3).

The *conditional probability* of an event e based on a context c is written as $p(e|c)$. The overall (unconditional) probability of an event e is related by $p(e) = \sum_{c \in C} p(c)p(e|c)$, where C is the set of all possible contexts. Based on conditional probabilities we can define the notion of conditional self-information as $i(e|c) = \log_2 \frac{1}{p(e|c)}$ of an event e in the context c. This need not be the same as the unconditional self-information. For example, a message stating that it is going to rain in LA with no other information tells us more than a message stating that it is going to rain in the context that it is currently January.

As with the unconditional case, we can define the average conditional self-information, and we call this the conditional-entropy of a source of messages. We have to derive this average by averaging both over the contexts and over the messages. For a message set S and context set C, the *conditional entropy* is

$$H(S|C) = \sum_{c \in C} p(c) \sum_{s \in S} p(s|c) \log_2 \frac{1}{p(s|c)} .$$

It is not hard to show that if the probability distribution of S is independent of the context C then $H(S|C) = H(S)$, and otherwise $H(S|C) < H(S)$. In other words, knowing the context can only reduce the entropy.

Shannon actually originally defined Entropy in terms of information sources. An *information sources* generates an infinite sequence of messages $X_k, k \in \{-\infty, \ldots, \infty\}$ from a fixed message set S. If the probability of each message is independent of the previous messages then the system is called an *independent and identically distributed* (iid) source. The entropy of such a source is called the *unconditional* or *first order* entropy and is as defined in Section 2.1. In this chapter by default we will use the term entropy to mean first-order entropy.

Another kind of source of messages is a Markov process, or more precisely a *discrete time Markov chain*. A sequence follows an order k Markov model if the probability of each message (or event) only depends on the k previous messages, in particular

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

where x_i is the i^{th} message generated by the source. The values that can be taken on by $\{x_{n-1}, \ldots, x_{n-k}\}$ are called the states of the system. The entropy of a Markov process is defined by the conditional entropy, which is based on the conditional probabilities $p(x_n|x_{n-1}, \ldots, x_{n-k})$.

Figure 1 shows an example of an first-order Markov Model. This Markov model represents the probabilities that the source generates a black (b) or white (w) pixel. Each arc represents a conditional probability of generating a particular pixel. For example $p(w|b)$ is the conditional probability of generating a white pixel given that the previous one was black. Each node represents one of the states, which in a first-order Markov model is just the previously

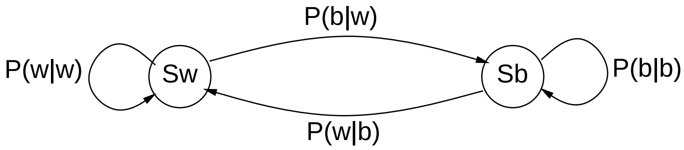


Figure 1: A two state first-order Markov Model

generated message. Lets consider the particular probabilities $p(b|w) = .01$, $p(w|w) = .99$, $p(b|b) = .7$, $p(w|b) = .3$. It is not hard to solve for $p(b) = 1/31$ and $p(w) = 30/31$ (do this as an exercise). These probabilities give the conditional entropy

$$30/31(.01\log(1/.01) + .99\log(1/.99)) + 1/31(.7\log(1/.7) + .3\log(1/.3)) \approx .107$$

This gives the expected number of bits of information contained in each pixel generated by the source. Note that the first-order entropy of the source is

$$30/31\log(31/30) + 1/31\log(1/30) \approx .206$$

which is almost twice as large.

Shannon also defined a general notion source entropy for an arbitrary source. Let A^n denote the set of all strings of length n from an alphabet A, then the n^{th} *order normalized entropy* is defined as

$$H_n = \frac{1}{n}\sum_{X\in A^n} p(X)\log\frac{1}{p(X)} \; . \qquad (1)$$

This is normalized since we divide it by n—it represents the per-character information. The *source entropy* is then defined as

$$H = \lim_{n\to\infty} H_n \; .$$

In general it is extremely hard to determine the source entropy of an arbitrary source process just by looking at the output of the process. This is because to calculate accurate probabilities even for a relatively simple process could require looking at extremely long sequences.

3 Probability Coding

As mentioned in the introduction, coding is the job of taking probabilities for messages and generating bit strings based on these probabilities. How the probabilities are generated is part of the model component of the algorithm, which is discussed in Section 4.

In practice we typically use probabilities for parts of a larger message rather than for the complete message, *e.g.*, each character or word in a text. To be consistent with the terminology in the previous section, we will consider each of these components a message on its own, and we will use the term *message sequence* for the larger message made up of these components. In general each little message can be of a different type and come from its own probability distribution. For example, when sending an image we might send a message specifying a color followed by messages specifying a frequency component of that color. Even the messages specifying the color might come from different probability distributions since the probability of particular colors might depend on the context.

We distinguish between algorithms that assign a unique code (bit-string) for each message, and ones that "blend" the codes together from more than one message in a row. In the first class we will consider Huffman codes, which are a type of prefix code. In the later category we consider arithmetic codes. The arithmetic codes can achieve better compression, but can require the encoder to delay sending messages since the messages need to be combined before they can be sent.

3.1 Prefix Codes

A *code* C for a message set S is a mapping from each message to a bit string. Each bit string is called a *codeword*, and we will denote codes using the syntax $C = \{(s_1, w_1), (s_2, w_2), \cdots, (s_m, w_m)\}$. Typically in computer science we deal with fixed-length codes, such as the ASCII code which maps every printable character and some control characters into 7 bits. For compression, however, we would like codewords that can vary in length based on the probability of the message. Such *variable length* codes have the potential problem that if we are sending one codeword after the other it can be hard or impossible to tell where one codeword finishes and the next starts. For example, given the code $\{(a, \texttt{1}), (b, \texttt{01}), (c, \texttt{101}), (d, \texttt{011})\}$, the bit-sequence `1011` could either be decoded as `aba`, `ca`, or `ad`. To avoid this ambiguity we could add a special stop symbol to the end of each codeword (*e.g.*, a `2` in a 3-valued alphabet), or send a length before each symbol. These solutions, however, require sending extra data. A more efficient solution is to design codes in which we can always uniquely decipher a bit sequence into its code words. We will call such codes *uniquely decodable* codes.

A prefix code is a special kind of uniquely decodable code in which no bit-string is a prefix of another one, for example $\{(a, \texttt{1}), (b, \texttt{01}), (c, \texttt{000}), (d, \texttt{001})\}$. All prefix codes are uniquely decodable since once we get a match, there is no longer code that can also match.

Exercise 3.1.1 *Come up with an example of a uniquely decodable code that is not a prefix code.*

Prefix codes actually have an advantage over other uniquely decodable codes in that we can decipher each message without having to see the start of the next message. This is important when sending messages of different types (*e.g.*, from different probability distributions). In fact in certain applications one message can specify the type of the next message, so it might be necessary to fully decode the current message before the next one can be interpreted.

A prefix codes can be viewed as a binary tree as follows

- Each message is a leaf in the tree
- The code for each message is given by following a path from the root to the leaf, and appending a 0 each time a left branch is taken, and a 1 each time a right branch is taken.

We will call this tree a *prefix-code tree*. Such a tree can also be useful in decoding prefix codes. As the bits come in, the decoder can follow a path down to the tree until it reaches a leaf, at which point it outputs the message and returns to the root for the next bit (or possibly the root of a different tree for a different message type).

In general prefix codes do not have to be restricted to binary alphabets. We could have a prefix code in which the bits have 3 possible values, in which case the corresponding tree would be ternary. In this chapter we only consider binary codes.

Given a probability distribution on a set of messages and associated variable length code C, we define the *average length* of the code as

$$l_a(C) = \sum_{(s,w)\in C} p(s)l(w)$$

where $l(w)$ is the length of the codeword w. We say that a prefix code C is an *optimal* prefix code if $l_a(C)$ is minimized (*i.e.*, there is no other prefix code for the given probability distribution that has a lower average length).

3.1.1 Relationship to Entropy

It turns out that we can relate the average length of prefix codes to the entropy of a set of messages, as we will now show. We will make use of the Kraft-McMillan inequality

Lemma 3.1.1 Kraft-McMillan Inequality. *For any uniquely decodable code C,*

$$\sum_{(s,w)\in C} 2^{-l(w)} \leq 1 \ ,$$

where $l(w)$ it the length of the codeword w. Also, for any set of lengths L such that

$$\sum_{l\in L} 2^{-l} \leq 1 \ ,$$

there is a prefix code C of the same size such that $l(w_i) = l_i$ $(i = 1, \ldots |L|)$.

The proof of this is left as a homework assignment. Using this we show the following

Lemma 3.1.2 *For any message set S with a probability distribution and associated uniquely decodable code C,*

$$H(S) \leq l_a(C)$$

Proof: In the following equations for a message $s \in S$, $l(s)$ refers to the length of the associated code in C.

$$\begin{aligned} H(S) - l_a(C) &= \sum_{s\in S} p(s)\log_2 \frac{1}{p(s)} - \sum_{s\in S} p(s)l(s) \\ &= \sum_{s\in S} p(s)\left(\log_2 \frac{1}{p(s)} - l(s)\right) \end{aligned}$$

$$
\begin{aligned}
&= \sum_{s\in S} p(s)\left(\log_2 \frac{1}{p(s)} - \log_2 2^{l(s)}\right)\\
&= \sum_{s\in S} p(s)\log_2 \frac{2^{-l(s)}}{p(s)}\\
&\leq \log_2(\sum_{s\in S} 2^{-l(s)})\\
&\leq 0
\end{aligned}
$$

The second to last line is based on Jensen's inequality which states that if a function $f(x)$ is concave then $\sum_i p_i f(x_i) \leq f(\sum_i p_i x_i)$, where the p_i are positive probabilities. The logarithm function is concave. The last line uses the Kraft-McMillan inequality. $\square$

This theorem says that entropy is a lower bound on the average code length. We now also show an upper bound based on entropy for optimal prefix codes.

Lemma 3.1.3 *For any message set S with a probability distribution and associated optimal prefix code C,*

$$l_a(C) \leq H(S) + 1 \ .$$

Proof: Take each message $s \in S$ and assign it a length $l(s) = \lceil \log \frac{1}{p(s)} \rceil$. We have

$$
\begin{aligned}
\sum_{s\in S} 2^{-l(s)} &= \sum_{s\in S} 2^{-\lceil \log \frac{1}{p(s)} \rceil}\\
&\leq \sum_{s\in S} 2^{-\log \frac{1}{p(s)}}\\
&= \sum_{s\in S} p(s)\\
&= 1
\end{aligned}
$$

Therefore by the Kraft-McMillan inequality there is a prefix code C' with codewords of length $l(s)$. Now

$$
\begin{aligned}
l_a(C') &= \sum_{(s,w)\in C'} p(s)l(w)\\
&= \sum_{(s,w)\in C'} p(s)\lceil \log \frac{1}{p(s)} \rceil\\
&\leq \sum_{(s,w)\in C'} p(s)(1 + \log \frac{1}{p(s)})\\
&= 1 + \sum_{(s,w)\in C'} p(s) \log \frac{1}{p(s)}\\
&= 1 + H(S)
\end{aligned}
$$

By the definition of optimal prefix codes, $l_a(C) \leq l_a(C')$. $\Box$

Another property of optimal prefix codes is that larger probabilities always have longer codes, as shown by the following theorem. This theorem will be useful later.

Theorem 3.1.1 *If C is an optimal prefix code for the probabilities $\{p_1, p_2, \ldots, p_n\}$ then $p_i > p_j$ implies that $l(c_i) \leq l(c_j)$.*

Proof: Assume $l(c_i) > l(c_j)$. Now consider the code gotten by switching c_i and c_j. If l_a is the average length of our original code, this new code will have length

$$\begin{aligned} l'_a &= l_a + p_j(l(c_i) - l(c_j)) + p_i(l(c_j) - l(c_i)) && (2) \\ &= l_a + (p_j - p_i)(l(c_i) - l(c_j)) && (3) \end{aligned}$$

Given our assumptions the $(p_j - p_i)(l(c_i) - l(c_j))$ is negative which contradicts the assumption that l_a is an optimal prefix code. $\Box$

3.2 Huffman Codes

Huffman codes are optimal prefix codes generated from a set of probabilities by a particular algorithm, the Huffman Coding Algorithm. David Huffman developed the algorithm as a student in a class on information theory at MIT in 1950. The algorithm is now probably the most prevalently used component of compression algorithms, used as the back end of GZIP, JPEG and many other utilities.

The Huffman algorithm is very simple and is most easily described in terms of how it generates the prefix-code tree.

- Start with a forest of trees, one for each message. Each tree contains a single vertex with weight $w_i = p_i$
- Repeat until only a single tree remains
  - Select two trees with the lowest weight roots (w_1 and w_2).
  - Combine them into a single tree by adding a new root with weight $w_1 + w_2$, and making the two trees its children. It does not matter which is the left or right child, but our convention will be to put the lower weight root on the left if $w_1 \neq w_2$.

For a code of size n this algorithm will require $n - 1$ steps since every complete binary tree with n leaves has $n - 1$ internal nodes, and each step creates one internal node. If we use a priority queue with $O(\log n)$ time insertions and find-mins (*e.g.*, a heap) the algorithm will run in $O(n \log n)$ time.

The key property of Huffman codes is that they generate optimal prefix codes. We show this in the following theorem, originally given by Huffman.

Lemma 3.2.1 *The Huffman algorithm generates an optimal prefix code.*

Proof: The proof will be on induction of the number of messages in the code. In particular we will show that if the Huffman code generates an optimal prefix code for all probability distributions of n messages, then it generates an optimal prefix code for all distributions of $n+1$ messages. The base case is trivial since the prefix code for 1 message is unique (*i.e.*, the null message) and therefore optimal.

We first argue that for any set of messages S there is an optimal code for which the two minimum probability messages are siblings (have the same parent in their prefix tree). By lemma 3.1.1 we know that the two minimum probabilities are on the lowest level of the tree (any complete binary tree has at least two leaves on its lowest level). Also, we can switch any leaves on the lowest level without affecting the average length of the code since all these codes have the same length. We therefore can just switch the two lowest probabilities so they are siblings.

Now for induction we consider a set of message probabilities S of size $n+1$ and the corresponding tree T built by the Huffman algorithm. Call the two lowest probability nodes in the tree x and y, which must be siblings in T because of the design of the algorithm. Consider the tree T' gotten by replacing x and y with their parent, call it z, with probability $p_z = p_x + p_y$ (this is effectively what the Huffman algorithm does). Lets say the depth of z is d, then

$$\begin{aligned} l_a(T) &= l_a(T') + p_x(d+1) + p_y(d+1) - p_z d && (4) \\ &= l_a(T') + p_x + p_y \ . && (5) \end{aligned}$$

To see that T is optimal, note that there is an optimal tree in which x and y are siblings, and that wherever we place these siblings they are going to add a constant $p_x + p_y$ to the average length of any prefix tree on S with the pair x and y replaced with their parent z. By the induction hypothesis $l_a(T')$ is minimized, since T' is of size n and built by the Huffman algorithm, and therefore $l_a(T)$ is minimized and T is optimal. $\square$

Since Huffman coding is optimal we know that for any probability distribution S and associated Huffman code C

$$H(S) \leq l_a(C) \leq H(S) + 1 \ .$$

3.2.1 Combining Messages

Even though Huffman codes are optimal relative to other prefix codes, prefix codes can be quite inefficient relative to the entropy. In particular $H(S)$ could be much less than 1 and so the extra 1 in $H(S)+1$ could be very significant.

One way to reduce the per-message overhead is to group messages. This is particularly easy if a sequence of messages are all from the same probability distribution. Consider a distribution of six possible messages. We could generate probabilities for all 36 pairs by multiplying the probabilities of each message (there will be at most 21 unique probabilities). A Huffman code can now be generated for this new probability distribution and used to code two messages at a time. Note that this technique is not taking advantage of conditional probabilities since it directly multiplies the probabilities. In general by grouping k messages the overhead of Huffman coding can be reduced from 1 bit per message to $1/k$ bits per message. The problem with this technique is that in practice messages are often not from

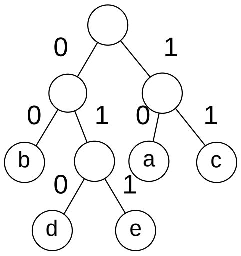


Figure 2: Binary tree for Huffman code 2

the same distribution and merging messages from different distributions can be expensive because of all the possible probability combinations that might have to be generated.

3.2.2 Minimum Variance Huffman Codes

The Huffman coding algorithm has some flexibility when two equal frequencies are found. The choice made in such situations will change the final code including possibly the code length of each message. Since all Huffman codes are optimal, however, it cannot change the average length. For example, consider the following message probabilities, and codes.

symbol	probability	code 1	code 2
a	0.2	01	10
b	0.4	1	00
c	0.2	000	11
d	0.1	0010	010
e	0.1	0011	011

Both codings produce an average of 2.2 bits per symbol, even though the lengths are quite different in the two codes. Given this choice, is there any reason to pick one code over the other?

For some applications it can be helpful to reduce the variance in the code length. The variance is defined as

$$\sum_{c \in C} p(c)(l(c) - l_a(C))^2$$

With lower variance it can be easier to maintain a constant character transmission rate, or reduce the size of buffers. In the above example, code 1 clearly has a much higher variance than code 2. It turns out that a simple modification to the Huffman algorithm can be used to generate a code that has minimum variance. In particular when choosing the two nodes to merge and there is a choice based on weight, always pick the node that was created earliest in the algorithm. Leaf nodes are assumed to be created before all internal nodes. In the example above, after `d` and `e` are joined, the pair will have the same probability as `c` and `a` (.2), but it was created afterwards, so we join `c` and `a`. Similarly we select `b` instead of `ac` to join with `de` since it was created earlier. This will give code 2 above.

3.3 Arithmetic Coding

Arithmetic coding is a technique for coding that allows the information from the messages in a message sequence to be combined to share the same bits. The technique allows the total number of bits sent to asymptotically approach the sum of the self information of the individual messages (recall that the self information of a message is defined as $\log_2 \frac{1}{p_i}$).

To see the significance of this, consider sending a thousand messages each having probability .999. Using a Huffman code, each message has to take at least 1 bit, requiring 1000 bits to be sent. On the other hand the self information of each message is $\log_2 \frac{1}{p_i} = .00144$ bits, so the sum of this self-information over 1000 messages is only 1.4 bits. It turns out that arithmetic coding will send all the messages using only 3 bits, a factor of hundreds fewer than a Huffman coder. Of course this is an extreme case, and when all the probabilities are small, the gain will be less significant. Arithmetic coders are therefore most useful when there are large probabilities in the probability distribution.

The main idea of arithmetic coding is to represent each possible sequence of n messages by a separate interval on the number line between 0 and 1, *e.g.* the interval from .2 to .5. For a sequence of messages with probabilities $p_1, \ldots, p_n$, the algorithm will assign the sequence to an interval of size $\prod_{i=1}^{n} p_i$, by starting with an interval of size 1 (from 0 to 1) and narrowing the interval by a factor of p_i on each message i. We can bound the number of bits required to uniquely identify an interval of size s, and use this to relate the length of the representation to the self information of the messages.

In the following discussion we assume the decoder knows when a message sequence is complete either by knowing the length of the message sequence or by including a special end-of-file message. This was also implicitly assumed when sending a sequence of messages with Huffman codes since the decoder still needs to know when a message sequence is over.

We will denote the probability distributions of a message set as $\{p(1), \ldots, p(m)\}$, and we define the *accumulated probability* for the probability distribution as

$$f(j) = \sum_{i=1}^{j-1} p(i) \qquad (j = 1, \ldots, m). \tag{6}$$

So, for example, the probabilities $\{.2, .5, .3\}$ correspond to the accumulated probabilities $\{0, .2, .7\}$. Since we will often be talking about sequences of messages, each possibly from a different probability distribution, we will denote the probability distribution of the i^{th} message as $\{p_i(1), \ldots, p_i(m_i)\}$, and the accumulated probabilities as $\{f_i(1), \ldots, f_i(m_i)\}$. For a particular sequence of message values, we denote the index of the i^{th} message value as v_i. We will use the shorthand p_i for $p_i(v_i)$ and f_i for $f_i(v_i)$.

Arithmetic coding assigns an interval to a sequence of messages using the following recurrences

$$\begin{aligned} l_i &= \begin{cases} f_i & i = 1 \\ l_{i-1} + f_i * s_{i-1} & 1 < i \leq n \end{cases} \\ s_i &= \begin{cases} p_i & i = 1 \\ s_{i-1} * p_i & 1 < i \leq n \end{cases} \end{aligned} \tag{7}$$

where l_n is the lower bound of the interval and s_n is the size of the interval, *i.e.* the interval is given by $[l_n, l_n + s_n)$. We assume the interval is inclusive of the lower bound, but exclusive

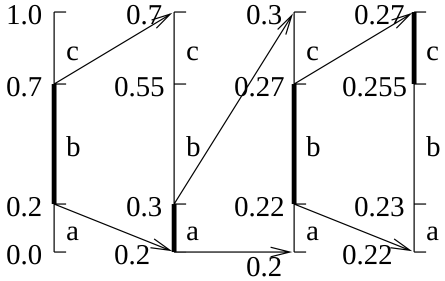


Figure 3: An example of generating an arithmetic code assuming all messages are from the same probability distribution $a = .2$, $b = .5$ and $c = .3$. The interval given by the message sequence *babc* is $[.255, .27)$.

of the upper bound. The recurrence narrows the interval on each step to some part of the previous interval. Since the interval starts in the range [0,1), it always stays within this range. An example of generating an interval for a short message sequences is illustrated in Figure 3. An important property of the intervals generated by Equation 7 is that all unique message sequences of length n will have nonoverlapping intervals. Specifying an interval therefore uniquely determines the message sequence. In fact, any number within an interval uniquely determines the message sequence. The job of decoding is basically the same as encoding but instead of using the message value to narrow the interval, we use the interval to select the message value, and then narrow it. We can therefore "send" a message sequence by specifying a number within the corresponding interval.

The question remains of how to efficiently send a sequence of bits that represents the interval, or a number within the interval. Real numbers between 0 and 1 can be represented in binary fractional notation as $.b_1 b_2 b_3 \ldots$. For example $.75 = .11, 9/16 = .1001$ and $1/3 = .01\overline{01}$, where $\overline{w}$ means that the sequence w is repeated infinitely. We might therefore think that it is adequate to represent each interval by selecting the number within the interval which has the fewest bits in binary fractional notation, and use that as the code. For example, if we had the intervals $[0, .33)$, $[.33, 67)$, and $[.67, 1)$ we would represent these with $.01(1/4)$, $.1(1/2)$, and $.11(3/4)$. It is not hard to show that for an interval of size s we need at most $-\lceil \log_2 s \rceil$ bits to represent such a number. The problem is that these codes are not a set of prefix codes. If you sent me 1 in the above example, I would not know whether to wait for another 1 or interpret it immediately as the interval $[.33, 67)$.

To avoid this problem we interpret every binary fractional codeword as an interval itself. In particular as the interval of all possible completions. For example, the codeword .010 would represent the interval $[1/4, 3/8)$ since the smallest possible completion is $.010\overline{0} = 1/4$ and the largest possible completion is $.010\overline{1} = 3/8 - \epsilon$. Since we now have several kinds of intervals running around, we will use the following terms to distinguish them. We will call the current interval of the message sequence (i.e $[l_i, l_i + s_i)$) the *sequence interval*, the interval corresponding to the probability of the i^{th} message (*i.e.*, $[f_i, f_i + p_i)$) the *message interval*, and the interval of a codeword the *code interval*.

An important property of code intervals is that there is a direct correspondence between whether intervals overlap and whether they form prefix codes, as the following Lemma shows.

Lemma 3.3.1 *For a code C, if no two intervals represented by its binary codewords $w \in C$ overlap then the code is a prefix code.*

Proof: Assume codeword a is a prefix of codeword b, then b is a possible completion of a and therefore its interval must be fully included in the interval of a. This is a contradiction. $\square$

To find a prefix code, therefore, instead of using any number in the interval to be coded, we select a codeword who's interval is fully included within the interval. Returning to the previous example of the intervals $[0, .33)$, $[.33, 67)$, and $[.67, 1)$, the codewords $.00[0, .25)$, $.100[.5, .625)$, and $.11[.75, 1)$ are adequate. In general for an interval of size s we can always find a codeword of length $-\lceil \log_2 s \rceil + 1$, as shown by the following lemma.

Lemma 3.3.2 *For any l and an s such that $l, s \geq 0$ and $l + s < 1$, the interval represented by taking the binary fractional representation of $l + s/2$ and truncating it to $-\lceil \log_2 s \rceil + 1$ bits is contained in the interval $[l, l + s)$.*

Proof: A binary fractional representation with l digits represents an interval of size less than 2^{-l} since the difference between the minimum and maximum completions are all 1s starting at the $l + 1^{th}$ location. This has a value $2^{-l} - \epsilon$. The interval size of a $-\lceil \log_2 s \rceil + 1$ bit representation is therefore less than $s/2$. Since we truncate $l + s/2$ downwards the upper bound of the interval represented by the bits is less than $l + s$. Truncating the representation of a number to $-\lceil \log_2 s \rceil + 1$ bits can have the effect of reducing it by at most $s/2$. Therefore the lower bound of truncating $l + s/2$ is at least l. The interval is therefore contained in $[l, l + s)$. $\square$

We will call the algorithm made up of generating an interval by Equation 7 and then using the truncation method of Lemma 3.3.2, the *RealArithCode* algorithm.

Theorem 3.3.1 *For a sequence of n messages, with self informations $s_1, \ldots, s_n$ the length of the arithmetic code generated by RealArithCode is bounded by $2 + \sum_{i=1}^{n} s_i$, and the code will not be a prefix of any other sequence of n messages.*

Proof: Equation 7 will generate a sequence interval of size $s = \prod_{i=1}^{n} p_i$. Now by Lemma 3.3.2 we know an interval of size s can be represented by $1 + \lceil -\log s \rceil$ bits, so we have

$$\begin{aligned}
1 + \lceil -\log s \rceil &= 1 + \lceil -\log_2 (\prod_{i=1}^{n} p_i) \rceil \\
&= 1 + \lceil \sum_{i=1}^{n} -\log_2 p_i \rceil \\
&= 1 + \lceil \sum_{i=1}^{n} s_i \rceil \\
&< 2 + \sum_{i=1}^{n} s_i
\end{aligned}$$

The claim that the code is not a prefix of other messages is taken directly from Lemma 3.3.1. □

The decoder for RealArithCode needs to read the input bits on demand so that it can determine when the input string is complete. In particular it loops for n iterations, where n is the number of messages in the sequence. On each iteration it reads enough input bits to narrow the code interval to within one of the possible message intervals, narrows the sequence interval based on that message, and outputs that message. When complete, the decoder will have read exactly all the characters generated by the coder. We give a more detailed description of decoding along with the integer implementation described below.

From a practical point of view there are a few problems with the arithmetic coding algorithm we described so far. First, the algorithm needs arbitrary precision arithmetic to manipulate l and s. Manipulating these numbers can become expensive as the intervals get very small and the number of significant bits get large. Another problem is that as described the encoder cannot output any bits until it has coded the full message. It is actually possible to interleave the generation of the interval with the generation of its bit representation by opportunistically outputting a 0 or 1 whenever the interval falls within the lower or upper half. This technique, however, still does not guarantee that bits are output regularly. In particular if the interval keeps reducing in size but still straddles .5, then the algorithm cannot output anything. In the worst case the algorithm might still have to wait until the whole sequence is received before outputting any bits. To avoid this problem many implementations of arithmetic coding break message sequences into fixed size blocks and use arithmetic coding on each block separately. This approach also has the advantage that since the group size is fixed, the encoder need not send the number of messages, except perhaps for the last group which could be smaller than the block size.

3.3.1 Integer Implementation

It turns out that if we are willing to give up a little bit in the efficiency of the coding, we can used fixed precision integers for arithmetic coding. This implementation does not give precise arithmetic codes, because of roundoff errors, but if we make sure that both the coder and decoder are always rounding in the same way the decoder will always be able to precisely interpret the message.

For this algorithm we assume the probabilities are given as counts

$$c(1), c(2), \ldots, c(m) \ ,$$

and the cumulative count are defined as before ($f(i) = \sum_{j=1}^{i-1} c(j)$). The total count will be denoted as

$$T = \sum_{j=1}^{m} c(j) \ .$$

Using counts avoids the need for fractional or real representations of the probabilities. Instead of using intervals between 0 and 1, we will use intervals between $[0..(R-1)]$ where $R = 2^k$ (*i.e.*, is a power of 2). There is the additional restriction that $R > 4T$. This will guarantee that no region will become too small to represent. The larger R is, the closer the algorithm will come to real arithmetic coding. As in the non-integer arithmetic coding, each message can come from its own probability distribution (have its own counts and accumulative counts), and we denote the i^{th} message using subscripts as before.

The coding algorithm is given in Figure 4. The current sequence interval is specified by the integers l (lower) and u (upper), and the corresponding interval is $[l, u+1)$. The size of the interval s is therefore $u - l + 1$. The main idea of this algorithm is to always keep the size greater than $R/4$ by expanding the interval whenever it gets too small. This is what the inner **while** loop does. In this loop whenever the sequence interval falls completely within the top half of the region (from $R/2$ to R) we know that the next bit is going to be a 1 since intervals can only shrink. We can therefore output a 1 and expand the top half to fill the region. Similarly if the sequence interval falls completely within the bottom half we can output a 0 and expand the bottom half of the region to fill the full region.

The third case is when the interval falls within the middle half of the region(from $R/4$ to $3R/4$). In this case the algorithm cannot output a bit since it does not know whether the bit will be a 0 or 1. It, however, can expand the middle region and keep track that is has expanded by incrementing a count m. Now when the algorithm does expand around the top (bottom), it outputs a 1 (0) followed by m 0s (1s). To see why this is the right thing to do, consider expanding around the middle m times and then around the top. The first expansion around the middle locates the interval between 1/4 and 3/4 of the initial region, and the second between 3/8 and 5/8. After m expansions the interval is narrowed to the region $(1/2 - 1/2^{m+1}, 1/2 + 1/2^{m+1})$. Now when we expand around the top we narrow the interval to $(1/2, 1/2 + 1/2^{m+1})$. All intervals contained in this range will start with a 1 followed by m $0s$.

Another interesting aspect of the algorithm is how it finishes. As in the case of real-number arithmetic coding, to make it possible to decode, we want to make sure that the code (bit pattern) for any one message sequence is not a prefix of the code for another message sequence. As before, the way we do this is to make sure the code interval is fully contained in the sequence interval. When the integer arithmetic coding algorithm (Figure 4) exits the `for` loop, we know the sequence interval $[l, u]$ completely covers either the second quarter (from $R/4$ to $R/2$) or the third quarter (from $R/2$ to $3R/4$) since otherwise one of the expansion rules would have been applied. The algorithm therefore simply determines which of these two regions the sequence interval covers and outputs code bits that narrow the code interval to one of these two quarters—a 01 for the second quarter, since all completions of 01 are in the second quarter, and a 10 for the third quarter. After outputting the first of these two bits the algorithm must also output m bits corresponding to previous expansions around the middle.

The reason that R needs to be at least $4T$ is that the sequence interval can become as small as $R/4 + 1$ without falling completely within any of the three halves. To be able to resolve the counts $C(i)$, T has to be at least as large as this interval.

An example: Here we consider an example of encoding a sequence of messages each from the same probability distribution, given by the following counts.

$$c(1) = 1, \quad c(2) = 10, \quad c(3) = 20$$

```
function IntArithCode(file, k, n)
    R = 2^k
    l = 0
    u = R − 1
    m = 0
    for i = 1 to n
        s = u − l + 1
        u = l + ⌊s · f_i(v_i + 1)/T⌋ − 1
        l = l + ⌊s · f_i(v_i)/T⌋
        while true
            if (l ≥ R/2)              // interval in top half
                Output(1)
                u = 2u − R + 1    l = 2l − R
                for j = 1 to m Output(0)
                m = 0
            else if (u < R/2)         // interval in bottom half
                Output(0)
                u = 2u + 1    l = 2l
                for j = 1 to m Output(1)
                m = 0
            else if (l ≥ R/4 and u < 3R/4)  // interval in middle half
                u = 2u − R/2 + 1    l = 2l − R/2
                m = m + 1
            else continue    // exit while loop
        end while
    end for
    if (l ≥ R/4)    // output final bits
        Output(1)
        for j = 1 to m Output(0)
        Output(0)
    else
        Output(0)
        for j = 1 to m Output(1)
        Output(1)
```

Figure 4: Algorithm for Integer Arithmetic Coding.

i	v_i	$f(v_i)$	$f(v_i+1)$	l	u	s	m	expand rule	output
start				0	255	256			
1	3	11	31	90	255	166	0		
2	2	1	11	95	147	53	0		
+				62	167	106	1	(middle half)	
3	1	0	1	62	64	3	1		
+				124	129	6	0	(bottom half)	01
+				120	131	12	1	(middle half)	
+				112	135	24	2	(middle half)	
+				96	143	48	4	(middle half)	
+				64	159	96	5	(middle half)	
+				0	191	192	6	(middle half)	
4	2	1	11	6	67	62	6		
+				12	135	124	0	(bottom half)	0111111
end							0	(final out)	01

Figure 5: Example of integer arithmetic coding. The rows represent the steps of the algorithm. Each row starting with a number represents the application of a contraction based on the next message, and each row with a + represents the application of one of the expansion rules.

The cumulative counts are

$$f(1) = 0, \quad f(2) = 1, \quad f(3) = 11$$

and $T = 31$. We will chose $k = 8$, so that $R = 256$. This satisfies the requirement that $R > 4T$. Now consider coding the message sequence 3, 1, 2, 3. Figure 5 illustrates the steps taken in coding this message sequence. The full code that is output is `01011111101` which is of length 11. The sum of the self-information of the messages is

$$-(\log_2(20/31) + \log_2(10/31) + \log_2(1/31) + \log_2(10/31)) = 8.85$$

.

Note that this is not within the bound given by Theorem 3.3.1. This is because we are not generating an exact arithmetic code and we are loosing some coding efficiency.

We now consider how to decode a message sent using the integer arithmetic coding algorithm. The code is given in Figure 6. The idea is to keep separate lower and upper bounds for the code interval (l_b and u_b) and the sequence interval (l and u). The algorithm reads one bit at a time and reduces the code interval by half for each bit that is read (the bottom half when the bit is a 0 and the top half when it is a 1). Whenever the code interval falls within an interval for the next message, the message is output and the sequence interval is reduced by the message interval. This reduction is followed by the same set of expansions around the top, bottom and middle halves as followed by the encoder. The sequence intervals therefore follow the exact same set of lower and upper bounds as when they were coded.

This property guarantees that all rounding happens in the same way for both the coder and decoder, and is critical for the correctness of the algorithm. It should be noted that reduction and expansion of the code interval is always exact since these are always changed by powers of 2.

4 Applications of Probability Coding

To use a coding algorithm we need a model from which to generate probabilities. Some simple models are to count character for text or pixel values for images and use the counts as probabilities. Such counts, however, would only give a compression ratio of about $4.7/8 = .59$ for English text as compared to the best compression algorithms that give ratios of close to .2. In this section we give some examples of more sophisticated models that are used in real-world applications. All these techniques take advantage of the "context" in some way. This can either be done by transforming the data before coding (*e.g.*, run-length coding, move-to-front coding, and residual coding), or directly using conditional probabilities based on a context (JBIG and PPM).

An issue to consider about a model is whether it is static or dynamic. A model can be static over all message sequences. For example one could predetermine the frequency of characters and text and "hardcode" those probabilities into the encoder and decoder. Alternatively, the model can be static over a single message sequence. The encoder executes one pass over the sequence to determine the probabilities, and then a second pass to use those probabilities in the code. In this case the encoder needs to send the probabilities to the decoder. This is the approach taken by most vector quantizers. Finally, the model can be dynamic over the message sequence. In this case the encoder updates its probabilities as it encodes messages. To make it possible for the decoder to determine the probability based on previous messages, it is important that for each message, the encoder codes it using the old probability and then updates the probability based on the message. The advantages of this approach are that the coder need not send additional probabilities, and that it can adapt to the sequence as it changes. This approach is taken by PPM.

4.1 Run-length Coding

Probably the simplest coding scheme that takes advantage of the context is run-length coding. Although there are many variants, the basic idea is to identify strings of adjacent messages of equal value and replace them with a single occurrence along with a count. For example, the message sequence `acccbbaaabb` could be transformed to (`a`,1), (`c`,3), (`b`,2), (`a`,3), (`b`,2). Once transformed, a probability coder (*e.g.*, Huffman coder) can be used to code both the message values and the counts. It is typically important to probability code the run-lengths since short lengths (*e.g.*, 1 and 2) are likely to be much more common than long lengths (*e.g.*, 1356).

An example of a real-world use of run-length coding is for the ITU-T T4 (Group 3) standard for Facsimile (fax) machines[2]. At the time of writing (1999), this was the standard

[2]ITU-T is part of the International Telecommunications Union (ITU, `http://www.itu.ch/`).

```
function IntArithDecode(file, k, n)
    R = 2^k
    l = 0    u = R − 1
    l_b = 0    u_b = R − 1
    j = 1
    while j ≤ n do
        s = u − l + 1
        i = 0
        do
            i = i + 1
            u' = l + ⌊s · f_i(j + 1)/T_j⌋ − 1
            l' = l + ⌊s · f_i(j)/T_j⌋
        while i ≤ m_j and not((l_b ≥ l') and (u_b ≤ u'))
        if i > m_j then
            b = ReadBit(file)
            s_b = u_b − l_b + 1
            l_b = l_b + b(s_b/2)
            u_b = l_b + s_b/2 − 1
        else
            Output(j)
            u = u'    l = l'
            j = j + 1
            while true
                if (l ≥ R/2)                  // interval in top half
                    u = 2u − R + 1    l = 2l − R
                    u_b = 2u_b − R + 1    l_b = 2l_b − R
                else if (u < R/2)             // interval in bottom half
                    u = 2u + 1    l = 2l
                    u_b = 2u_b + 1    l_b = 2l_b
                else if (l ≥ R/4 and u < 3R/4)  // interval in middle half
                    u = 2u − R/2 + 1    l = 2l − R/2
                    u_b = 2u_b − R/2 + 1    l_b = 2l_b − R/2
                else continue    // exit inner while loop
        end if
    end while
```

Figure 6: Integer Arithmetic Decoding

run-length	white codeword	black codeword
0	00110101	0000110111
1	000111	010
2	0111	11
3	1000	10
4	1011	011
..		
20	0001000	00001101000
..		
64+	11011	0000001111
128+	10010	000011001000

Table 3: ITU-T T4 Group 3 Run-length Huffman codes.

for all home and business fax machines used over regular phone lines. Fax machines transmit black-and-white images. Each pixel is called a *pel* and the horizontal resolution is fixed at 8.05 pels/mm. The vertical resolution varies depending on the mode. The T4 standard uses run-length encoding to code each sequence of black and white pixels. Since there are only two message values black and white, only the run-lengths need to be transmitted. The T4 standard specifies the start color by placing a dummy white pixel at the front of each row so that the first run is always assumed to be a white run. For example, the sequence `bbbbwwbbbbb` would be transmitted as 1,4,2,5. The T4 standard uses static Huffman codes to encode the run-lengths, and uses a separate codes for the black and white pixels. To account for runs of more than 64, it has separate codes to specify multiples of 64. For example, a length of 150, would consist of the code for 128 followed by the code for 22. A small subset of the codes are given in Table 4.1. These Huffman codes are based on the probability of each run-length measured over a large number of documents. The full T4 standard also allows for coding based on the previous line.

4.2 Move-To-Front Coding

Another simple coding schemes that takes advantage of the context is move-to-front coding. This is used as a sub-step in several other algorithms including the Burrows-Wheeler algorithm discussed later. The idea of move-to-front coding is to preprocess the message sequence by converting it into a sequence of integers, which hopefully is biases toward integers with low values. The algorithm then uses some form of probability coding to code these values. In practice the conversion and coding are interleaved, but we will describe them as separate passes. The algorithm assumes that each message comes from the same alphabet, and starts with a total order on the alphabet (*e.g.*, $[a, b, c, d, \ldots]$). For each message, the first pass of the algorithm outputs the position of the character in the current order of the alphabet, and then updates the order so that the character is at the head. For example, coding the character c with an order $[a, b, c, d, \ldots]$ would output a 3 and change the order to

$[c, a, b, d, \ldots]$. This is repeated for the full message sequence. The second pass converts the sequence of integers into a bit sequence using Huffman or Arithmetic coding.

The hope is that equal characters often appear close to each other in the message sequence so that the integers will be biased to have low values. This will give a skewed probability distribution and good compression.

4.3 Residual Coding: JPEG-LS

Residual compression is another general compression technique used as a sub-step in several algorithms. As with move-to-front coding, it preprocesses the data so that the message values have a better skew in their probability distribution, and then codes this distribution using a standard probability coder. The approach can be applied to message values that have some meaningful total order (*i.e.*, in which being close in the order implies similarity), and is most commonly used for integers values. The idea of residual coding is that the encoder tries to guess the next message value based on the previous context and then outputs the difference between the actual and guessed value. This is called the *residual*. The hope is that this residual is biased toward low values so that it can be effectively compressed. Assuming the decoder has already decoded the previous context, it can make the same guess as the coder and then use the residual it receives to correct the guess. By not specifying the residual to its full accuracy, residual coding can also be used for lossy compression

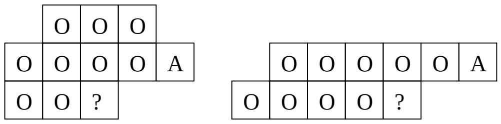
Residual coding is used in JPEG lossless (JPEG LS), which is used to compress both grey-scale and color images.[3] Here we discuss how it is used on gray scale images. Color images can simply be compressed by compressing each of the three color planes separately. The algorithm compresses images in raster order—the pixels are processed starting at the top-most row of an image from left to right and then the next row, continuing down to the bottom. When guessing a pixel the encoder and decoder therefore have as their disposal the pixels to the left in the current row and all the pixels above it in the previous rows. The JPEG LS algorithm just uses 4 other pixels as a context for the guess—the pixel to the left (W), above and to the left (NW), above (N), and above and to the right (NE). The guess works in two stages. The first stage makes the following guess for each pixel value.

$$G = \begin{cases} \min(W, N) & \max(N, W) \leq NW \\ \max(W, N) & \min(N, W) < NW \\ N + W - NW & \text{otherwise} \end{cases} \tag{8}$$

This might look like a magical equation, but it is based on the idea of taking an average of nearby pixels while taking account of edges. The first and second clauses capture horizontal and vertical edges. For example if $N > W$ and $N \leq NW$ this indicates a horizontal edge and W is used as the guess. The last clause captures diagonal edges.

Given an initial guess G a second pass adjusts that guess based on local gradients. It uses the three gradients between the pairs of pixels (NW, W), (NW, N), and (N, NE). Based on the value of the gradients (the difference between the two adjacent pixels) each is classified into one of 9 groups. This gives a total of 729 contexts, of which only 365 are needed because

[3]This algorithm is based on the LOCO-I (LOw COmplexity LOssless COmpression for Images) algorithm and the official standard number is ISO-14495-1/ITU-T.87.



(a) (b)

Figure 7: JBIG contexts: (a) three-line template, and (b) two-line template. X is the current pixel and A is the "roaming pixel".

of symmetry. Each context stores its own adjustment value which is used to adjust the guess. Each context also stores information about the quality of previous guesses in that context. This can be used to predict variance and can help the probability coder. Once the algorithm has the final guess for the pixel, it determines the residual and codes it.

4.4 Context Coding: JBIG

The next two techniques we discuss both use conditional probabilities directly for compression. In this section we discuss using context-based conditional probabilities for Bilevel (black-and-white) images, and in particular the JBIG1 standard. In the next section we discuss using a context in text compression. JBIG stands for the Joint Bilevel Image Processing Group. It is part of the same standardization effort that is responsible for the JPEG standard. The algorithm we describe here is JBIG1, which is a lossless compressor for bilevel images. JBIG1 typically compresses from 20-80% better than ITU Groups III and IV fax encoding outlined in Section 4.1.

JBIG is similar to JPEG LS in that it uses a local context of pixels to code the current pixel. Unlike JPEG LS, however, JBIG uses conditional probabilities directly. JBIG also allows for progressive compression—an image can be sent as a set of layers of increasing resolution. Each layer can use the previous layer to aid compression. We first outline how the initial layer is compressed, and then how each following layer is compressed.

The first layer is transmitted in raster order, and the compression uses a context of 10 pixels above and to the right of the current pixel. The standard allows for two different templates for the context as shown in Figure 7. Furthermore, the pixel marked A is a roaming pixel and can be chosen to be any fixed distance to the right of where it is marked in the figure. This roaming pixel is useful for getting good compression on images with repeated vertical lines. The encoder decides on which of the two templates to use and on where to place A based on how well they compress. This information is specified at the head of the compressed message sequence. Since each pixel can only have two values, there are 2^{10} possible contexts. The algorithm dynamically generates the conditional probabilities for a black or white pixel for each of the contexts, and uses these probabilities in a modified arithmetic coder—the coder is optimized to avoid multiplications and divisions. The decoder can decode the pixels since it can build the probability table in the same way as the encoder.

The higher-resolution layers are also transmitted in raster order, but now in addition to using a context of previous pixels in the current layer, the compression algorithm can use pixels from the previous layer. Figure 8 shows the context templates. The context consists of 6 pixels from the current layer, and 4 pixels from the lower resolution layer. Furthermore

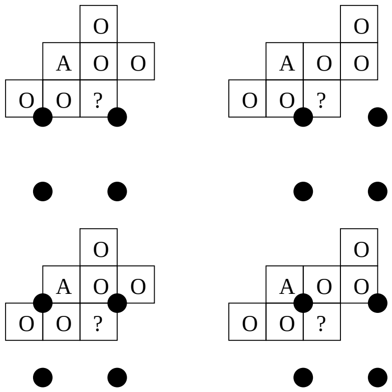


Figure 8: JBIG contexts for progressive transmission. The dark circles are the low resolution pixels, the 0s are the high-resolution pixels, the A is a roaming pixel, and the ? is the pixel we want to code/decode. The four context configurations are for the four possible configurations of the high-resolution pixel relative to the low resolution pixel.

2 additional bits are needed to specify which of the four configurations the coded pixel is in relative to the previous layer. This gives a total of 12 bits and 4096 contexts. The algorithm generates probabilities in the same way as for the first layer, but now with some more contexts. The JBIG standard also specifies how to generate lower resolution layers from higher resolution layers, but this won't be discussed here.

The approach used by JBIG is not well suited for coding grey-scale images directly since the number of possible contexts go up as m^p, where m is the number of grey-scale pixel values, and p is the number of pixels. For 8-bit grey-scale images and a context of size 10, the number of possible contexts is 2^{30}, which is far too many. The algorithm can, however, be applied to grey-scale images indirectly by compressing each bit-position in the grey scale separately. This still does not work well for grey-scale levels with more than 2 or 3 bits.

4.5 Context Coding: PPM

Over the past decade, variants of this algorithm have consistently given either the best or close to the best compression ratios (PPMC, PPM*, BOA and RK from Table 2 all use ideas from PPM). They are, however, are not very fast.

The main idea of PPM (Prediction by Partial Matching) is to take advantage of the previous K characters to generate a conditional probability of the current character. The simplest way to do this would be to keep a dictionary for every possible string s of k characters, and for each string have counts for every character x that follows s. The conditional probability of x in the context s is then $C(x|s)/C(s)$, where $C(x|s)$ is the number of times x follows s and $C(s)$ is the number of times s appears. The probability distributions can then be used by a Huffman or Arithmetic coder to generate a bit sequence. For example, we might have a dictionary with `qu` appearing 100 times and `e` appearing 45 times after `qu`. The conditional probability of the `e` is then .45 and the coder should use about 1 bit to encode it. Note that the probability distribution will change from character to character since each context has its own distribution. In terms of decoding, as long as the context

Order 0		Order 1		Order 2	
Context	Counts	Context	Counts	Context	Counts
empty	a = 4	a	c = 3	ac	b = 1
	b = 2				c = 2
	c = 5	b	a = 2		
				ba	c = 1
		c	a = 1		
			b = 2	ca	a = 1
			c = 2		
				cb	a = 2
				cc	a = 1
					b = 1

Figure 9: An example of the PPM table for $k = 2$ on the string `accbaccacba`.

precedes the character being coded, the decoder will know the context and therefore know which probability distribution to use. Because the probabilities tend to be high, arithmetic codes work much better than Huffman codes for this approach.

There are two problems with the basic dictionary method described in the previous paragraph. First, the dictionaries can become very large. There is no solution to this problem other than to keep k small, typically 3 or 4. A second problem is what happens if the count is zero. We cannot use zero probabilities in any of the coding methods (they would imply infinitely long strings). One way to get around this is to assume a probability of not having seen a sequence before and evenly distribute this probability among the possible following characters that have not been seen. Unfortunately this gives a completely even distribution, when in reality we might know that `a` is more likely than `b`, even without knowing its context.

The PPM algorithm has a clever way to deal with the case when a context has not been seen before, and is based on the idea of partial matching. The algorithm builds the dictionary on the fly starting with an empty dictionary, and every time the algorithm comes across a string it has not seen before it tries to match a string of one shorter length. This is repeated for shorter and shorter lengths until a match is found. For each length $0, 1, \ldots, k$ the algorithm keeps statistics of patterns it has seen before and counts of the following characters. In practice this can all be implemented in a single trie. In the case of the length-0 contexts the counts are just counts of each character seen assuming no context.

An example table is given in Figure 4.5 for a string `accbaccacba`. Now consider following this string with a `c`. Since the algorithm has the context `ba` followed by `c` in its dictionary, it can output the `c` based on its probability in this context. Although we might think the probability should be 1, since `c` is the only character that has ever followed `ba`, we need to give some probability of no match, which we will call the "escape" probability. We will get back to how this probability is set shortly. If instead of `c` the next character to code

Order 0		Order 1		Order 2	
Context	Counts	Context	Counts	Context	Counts
empty	a = 4	a	c = 3	ac	b = 1
	b = 2		$ = 1		c = 2
	c = 5				$ = 2
	$ = 3	b	a = 2		
			$ = 1	ba	c = 1
					$ = 1
		c	a = 1		
			b = 2	ca	a = 1
			c = 2		$ = 1
			$ = 3		
				cb	a = 2
					$ = 1
				cc	a = 1
					b = 1
					$ = 2

Figure 10: An example of the PPMC table for $k = 2$ on the string `accbaccacba`. This assumes the "virtual" count of each escape symbol ($) is the number of different characters that have appeared in the context.

is an `a`, then the algorithm does not find a match for a length 2 context so it looks for a match of length 1, in this case the context is the previous `a`. Since `a` has never followed by another `a`, the algorithm still does not find a match, and looks for a match with a zero length context. In this case it finds the `a` and uses the appropriate probability for `a` (4/11). What if the algorithm needs to code a `d`? In this case the algorithm does not even find the character in the zero-length context, so it assigns the character a probability assuming all unseen characters have even likelihood.

Although it is easy for the encoder to know when to go to a shorter context, how is the decoder supposed to know in which sized context to interpret the bits it is receiving. To make this possible, the encoder must notify the decoder of the size of the context. The PPM algorithm does this by assuming the context is of size k and then sending an "escape" character whenever moving down a size. In the example of coding an `a` given above, the encoder would send two escapes followed by the `a` since the context was reduced from 2 to 0. The decoder then knows to use the probability distribution for zero length contexts to decode the following bits.

The escape can just be viewed as a special character and given a probability within each context as if it was any other kind of character. The question is how to assign this probability. Different variants of PPM have different rules. PPMC uses the following scheme. It sets the count for the escape character to be the number of different characters seen following the given context. Figure 10 shows an example of the counts using this scheme. In this example, the probability of no match for a context of `ac` is $2/(1+2+2) = .4$ while the probability for a `b` in that context is .2. There seems to be no theoretical justification for this choice, but empirically it works well.

There is one more trick that PPM uses. This is that when switching down a context, the algorithm can use the fact that it switched down to exclude the possibility of certain characters from the shorter context. This effectively increases the probability of the other characters and decreases the code length. For example, if the algorithm were to code an `a`, it would send two escapes, but then could exclude the `c` from the counts in the zero length context. This is because there is no way that two escapes would be followed by a `c` since the `c` would have been coded in a length 2 context. The algorithm could then give the `a` a probability of 4/6 instead of 4/11 (.58 bits instead of 1.46 bits!).

5 The Lempel-Ziv Algorithms

The Lempel-Ziv algorithms compress by building a dictionary of previously seen strings. Unlike PPM which uses the dictionary to predict the probability of each character, and codes each character separately based on the context, the Lempel-Ziv algorithms code groups of characters of varying lengths. The original algorithms also did not use probabilities—strings were either in the dictionary or not and all strings in the dictionary were give equal probability. Some of the newer variants, such as `gzip`, do take some advantage of probabilities.

At the highest level the algorithms can be described as follows. Given a position in a file, look through the preceeding part of the file to find the longest match to the string starting at the current position, and output some code that refers to that match. Now move the finger past the match. The two main variants of the algorithm were described by Ziv and Lempel in two separate papers in 1977 and 1978, and are often refered to as LZ77 and LZ78. The algorithms differ in how far back they search and how they find matches. The LZ77 algorithm is based on the idea of a sliding window. The algorithm only looks for matches in a window a fixed distance back from the current position. Gzip, ZIP, and V.42bis (a standard modem protocal) are all based on LZ77. The LZ78 algorithm is based on a more conservative approach to adding strings to the dictionary. Unix compress, and the Gif format are both based on LZ78.

In the following discussion of the algorithms we will use the term *cursor* to mean the position an algorithm is currently trying to encode from.

5.1 Lempel-Ziv 77 (Sliding Windows)

The LZ77 algorithm and its variants use a sliding window that moves along with the cursor. The window can be divided into two parts, the part before the cursor, called the dictionary, and the part starting at the cursor, called the lookahead buffer. The size of these two parts are parameters of the program and are fixed during execution of the algorithm. The basic algorithm is very simple, and loops executing the following steps

Step	Input String															Output Code
1	$\boxed{a}$	$\underline{a}$	$\underline{c}$	$\underline{a}$	a	c	a	b	c	a	b	a	b	a	c	(0, 0, a)
2	**a**	$\boxed{a}$	$\underline{c}$	$\underline{a}$	$\underline{a}$	c	a	b	c	a	b	a	b	a	c	(1, 1, c)
3	**a**	**a**	**c**	$\boxed{a}$	$\underline{a}$	$\underline{c}$	$\underline{a}$	b	c	a	b	a	b	a	c	(3, 4, b)
4	a	a	**c**	**a**	**a**	**c**	**a**	**b**	$\boxed{c}$	$\underline{a}$	$\underline{b}$	$\underline{a}$	b	a	c	(3, 3, a)
5	a	a	c	a	a	c	**a**	**b**	**c**	**a**	**b**	**a**	$\boxed{c}$	$\underline{a}$	$\underline{a}$	(4, 2, a)

Figure 11: An example of LZ77 with a dictionary of size 6 and a lookahead buffer of size 4. The cursor position is boxed, the dictionary is bold faced, and the lookahed buffer is underlined. The last step does not find the longer match (10,3,1) since it is outside of the window.

1. Find the longest match of a string starting at the cursor and completely contained in the lookahead buffer to a string starting in the dictionary.

2. Output a triple (p, n, c) containing the position p of the occurence in the window, the length n of the match and the next character c past the match.

3. Move the cursor $n + 1$ characters forward.

The position p can be given relative to the cursor with 0 meaning no match, 1 meaning a match starting at the previous character, etc.. Figure 11 shows an example of the algorithm on the string `aacaacabcababac`.

To decode the message we consider a single step. Inductively we assume that the decoder has correctly constructed the string up to the current cursor, and we want to show that given the triple (p, n, c) it can reconstruct the string up to the next cursor position. To do this the decoder can look the string up by going back p positions and taking the next n characters, and then following this with the character c. The one tricky case is when $n > p$, as in step 3 of the example in Figure 11. The problem is that the string to copy overlaps the lookahead buffer, which the decoder has not filled yet. In this case the decoder can reconstruct the message by taking p characters before the cursor and repeating them enough times after the cursor to fill in n positions. If, for example, the code was `(2,7,d)` and the two characters before the cursor were `ab`, the algorithm would place `abababa` and then the `d` after the cursor.

There have been many improvements on the basic algorithm. Here we will describe several improvements that are used by `gzip`.

Two formats: This improvement, often called the *LZSS Variant*, does not include the next character in the triple. Instead it uses two formats, either a pair with a position and length, or just a character. An extra bit is typically used to distinguish the formats. The algorithm tries to find a match and if it finds a match that is at least of length 3, it uses the offset, length format, otherwise it uses the single character format. It turns out that this improvement makes a huge difference for files that do not compress well since we no longer have to waste the position and length fields.

Huffman coding the components: Gzip uses separate huffman codes for the offset, the length and the character. Each uses addaptive Huffman codes.

Non greedy: The LZ77 algorithm is greedy in the sense that it always tries to find a match starting at the first character in the lookahead buffer without caring how this will affect later matches. For some strings it can save space to send out a single character at the current cursor position and then match on the next position, even if there is a match at the current position. For example, consider coding the string

$$\mathbf{d}\quad \mathbf{b}\quad \mathbf{c}\quad \mathbf{a}\quad \mathbf{b}\quad \mathbf{c}\quad \mathbf{d}\quad \boxed{\text{a}}\quad \underline{\text{b}}\quad \underline{\text{c}}\quad \underline{\text{a}}\quad \underline{\text{b}}$$

In this case LZCC would code it as (1,3,3), (0,a), (0,b). The last two letters are coded as singletons since the match is not at least three characters long. This same buffer could instead be coded as (0,a), (1,6,4) if the coder was not greedy. In theory one could imagine trying to optimize coding by trying all possible combinations of matches in the lookahead buffer, but this could be costly. As a tradeoff that seems to work well in practice, Gzip only looks ahead 1 character, and only chooses to code starting at the next character if the match is longer than the match at the current character.

Hash Table: To quickly access the dictionary Gzip uses a hash table with every string of length 3 used as the hash keys. These keys index into the position(s) in which they occur in the file. When trying to find a match the algorithm goes through all of the hash entries which match on the first three characters and looks for the longest total match. To avoid long searches when the dictionary window has many strings with the same three characters, the algorithm only searches a bucket to a fixed length. Within each bucket, the positions are stored in an order based on the position. This makes it easy to select the more recent match when the two longest matches are equal length. Using the more recent match better skews the probability distribution for the offsets and therefore decreases the average length of the Huffman codes.

5.2 Lempel-Ziv-Welch

In this section we will describe the LZW (Lempel-Ziv-Welch) variant of LZ78 since it is the one that is most commonly used in practice. The algorithm starts with a dictionary containing one entry for each character. If it is encoding bytes, the dictionary will start with 256 entries. As the algorithm progresses it will add new strings to the dictionary such that each string is only added if a prefix one character shorter is already in the dictionary. For example, `John` is only added if `Joh` had previously appeared in the message sequence.

We will use the following interface to the dictionary. We assume that each entry of the dictionary is given a code number, where these numbers are typically given out incrementally starting at 256 (the first 256 are reserved for the characters).

```
function LZW_Encode(File)
    C = ReadChar(File)
    while C ≠ EOF do
        x = ReadChar(File)
        while StringInDict?(C, x) do
            C = GetCode(C, x)
            x = ReadChar(file)
        Output(C)
        AddDict(C, x)
        C = x
```

```
function LZW_Decode(File)
    C = ReadCode(File)
    W = GetString(C)
    Output(W)
    while C ≠ EOF do
        C' = ReadCode(File)
        if CodeInDict?(C') then
            W = GetString(C')
            AddDict(C, W[0])
        else
            C' = AddDict(C, W[0])
            W = GetString(C')
        Output(W)
        C = C'
```

Figure 12: Code for LZW encoding and decoding.

$C' = \text{AddDict}(C, x)$	Creates a new dictionary entry by extending an existing dictionary entry given by the code C with the one character x. Returns the new code.
Flag = StringInDict?(C, x)	Returns true if the string gotten by extending the entry of the code C with the character x is in the dictionary.
$C' = \text{GetCode}(C, x)$	Return the code of the string gotten by extending the entry of the code C with the character x.
$W = \text{GetString}(C)$	Returns the string W corresponding to code C.
Flag = CodeInDict?(C)	Returns true if the Code C is in the dictionary and false otherwise.

The encoding algorithm is described in Figure 12, and Tables 4 and 5 give two examples of encoding and coding. The coder and decoder build the dictionary completely symmetrically such that the two have the same dictionary at the same point in the text. In general the decoder always has an entry for a code before it receives the code. This is because the encoder adds the entry to the dictionary after sending the data that created the entry. The one exception is when there are many characters in a row. This exception is shown by the second example and handled by the **else** clause in LZW_decode.

A problem with the algorithm is that the dictionary can get too large. There are several choices of what to do. Here are some of them.

1. Throw dictionary away when reaching a certain size (GIF)

2. Throw dictionary away when not effective (Unix Compress)

3. Throw Least Recently Used entry away when reaches a certain size (BTLZ - British Telecom Standard)

	C	x	InDict?(C,x)	AddDict(C,x)	Output(C)
init	a				
	a	b	false	256 (a,b)	a
	b	c	false	257 (b,c)	b
	c	a	false	258 (c,a)	c
	a	b	true (256)		
+	256	c	false	259 (256,c)	256
	c	a	true (258)		
+	258	EOF	false	-	258

(a) Encoding

	C	C'	W	InDict?(C')	AddDict(C,W[0])	Output(W)
Init	a		a			a
	a	b	b	true	256 (a,b)	b
	b	c	c	true	257 (b,c)	c
	c	256	ab	true	258 (c,a)	ab
	256	258	ca	true	259 (256,c)	ca

(b) Decoding

Table 4: LZW Encoding and Decoding `abcabca`. The rows with a + for encoding are iterations of the inner **while** loop.

	C	x	InDict(C,x)	AddDict(C,x)	Output(C)
init	a				
	a	a	false	256 (a,a)	a
+	a	a	true (256)		
	256	a	false	257 (256,a)	256
+	a	a	true (256)		
+	256	a	true (257)		
	257	EOF	false	-	257

(a) Encoding

	C	C'	W	InDict?(C')	AddDict(C,W[0])	Output(W)
Init	a		a			a
	a	256	aa	false	256 (a,a)	aa
	256	257	aaa	false	257 (256,a)	aaa

(b) Decoding

Table 5: LZW Encoding and Decoding `aaaaaa`

Implementing the Dictionary: One of the biggest advantages of the LZ78 algorithms and reason for its success is that the dictionary operations can run very quickly. Our goal is to implement the 3 dictionary operations. The basic idea is to store the dictionary as a partially filled k-ary tree such that the root is the empty string, and any path down the tree to a node from the root specifies the match. The path need not go to a leaf since because of the prefix property of the LZ78 dictionary, all paths to internal nodes must belong to strings in the dictionary. We can use the codes as pointers to nodes of the tree (possibly indirectly through an array). To implement the GetCode function we start at the given node and follow a path from that node to the root. This requires that every child has a pointer to its parent. To implement the FindCode operation we go from the code and search to see if there is a child with character c. For the AddCode operation we add a child with character c to the node belonging to C. If we assume k is constant the FindCode and AddCode operations will take constant time since they only require going down one level of the tree. The GetCode operation requires $|W|$ time to follow the tree up to the root, but this operation is only used by the decoder, and always outputs W after decoding it. The whole algorithm for both coding and decoding require time that is linear in the message size.

To discuss one more level of detail, lets consider how to store the pointers. The parent pointers are trivial to keep since each node only needs a single pointer. The children pointers are a bit more difficult to do efficiently. One choice is to store an array of length k for each node. Each entry is initialized to empty and then searches can be done with a single arrary reference, but we need k pointers per node (k is often 256 in practice) and the memory is prohibitive. Another choice is to use a linked list (or possibly balanced tree) to store the children. This has much better space but requires more time to find a child (although technically still constant time since k is "constant"). A compromise that can be made in practice is to use a linked list until the number of children in a node rises above some threshold k' and then switch to an array. This would require copying the linked list into the array when switching.

Yet another technique is to use a hash table instead of child pointers. The string being searched for can be hashed directly to the appropriate code. This is what the GIF does.

6 Other Lossless Compression

6.1 Burrows Wheeler

The Burrows Wheeler algorithm is a relatively recent algorithm. An implementation of the algorithm called `bzip`, is currently one of the best overall compression algorithms for text. It gets compression ratios that are within 10% of the best algorithms such as PPM, but runs significantly faster.

Rather than describing the algorithm immediately, lets try to go through a thought process that leads to the algorithm. Recall that the basic idea of PPM was to try to find as long a context as possible that matched the current context and use that to effectively predict the next character. A problem with PPM is in selecting k. If we set k too large we will usually not find matches and end up sending too many escape characters. On the other hand if we set it too low, we would not be taking advantage of enough context. We could

	a	ccbaccacba	ccbaccacba$_4$	a$_1$	cbaccacbaa$_1$	c$_1$
a	c	cbaccacba	cbaccacbaa$_1$	c$_1$	ccbaccacba$_4$	a$_1$
ac	c	baccacba	baccacbaac$_1$	c$_2$	cacbaaccba$_2$	c$_3$
acc	b	accacba	accacbaacc$_2$	b$_1$	baaccbacca$_3$	c$_5$
accb	a	ccacba	ccacbaaccb$_1$	a$_2$	accbaccacb$_2$	a$_4$
accba	c	cacba	cacbaaccba$_2$	c$_3$	ccacbaaccb$_1$	a$_2$
accbac	c	acba	acbaaccbac$_3$	c$_4$	baccacbaac$_1$	c$_2$
accbacc	a	cba	cbaaccbacc$_4$	a$_3$	acbaaccbac$_3$	c$_4$
accbacca	c	ba	baaccbacca$_3$	c$_5$	aaccbaccac$_5$	b$_2$
accbaccac	b	a	aaccbaccac$_5$	b$_2$	accacbaacc$_2$	b$_1$
accbaccacb	a		accbaccacb$_2$	a$_4$	cbaaccbacc$_4$	a$_3$
	(a)		(b)		(c)	

Figure 13: Sorting the characters a$_1$c$_1$c$_2$b$_1$a$_2$c$_3$c$_4$a$_3$c$_5$b$_2$a$_4$ based on context: (a) each character in its context, (b) end context moved to front, and (c) characters sorted by their context using reverse lexicographic ordering. We use subscripts to distinguish different occurences of the same character.

have the system automatically select k based on which does the best encoding, but this is expensive. Also within a single text there might be some very long contexts that could help predict, while most helpful contexts are short. Using a fixed k we would probably end up ignoring the long contexts.

Lets see if we can come up with a way to take advantage of the context that somehow automatically adapts. Ideally we would like the method also to be a bit faster. Consider taking the string we want to compress and looking at the full context for each character—*i.e.*, all previous characters from the start of the string up to the character. In fact, to make the contexts the same length, which will be convenient later, we add to the head of each context the part of the string following the character making each context $n-1$ characters. Examples of the context for each character of the string accbaccacba are given in Figure 6.1. Now lets sort these contexts based on reverse lexical order, such that the last character of the context is the most significant (see Figure 6.1c). Note that now characters with the similar contexts (preceeding characters) are near each other. In fact, the longer the match (the more preceeding characters that match identically) the closer they will be to each other. This is similar to PPM in that it prefers longer matches when "grouping", but will group things with shorter matches when the longer match does not exist. The difference is that there is no fixed limit k on the length of a match—a match of length 100 has priority over a match of 99.

In practice the sorting based on the context is executed in blocks, rather than for the full message sequence. This is because the full message sequence and additional data structures required for sorting it, might not fit in memory. The process of sorting the characters by their context is often refered to as a *block-sorting transform*. In the dicussion below we will refer to the sequence of characters generated by a block-sorting transform as the *context-sorted sequence* (*e.g.*, c$_1$a$_1$c$_3$c$_5$a$_4$a$_2$c$_2$c$_4$b$_2$b$_1$a$_3$ in Figure 6.1). Given the correlation between nearyby characters in a context-sorted sequence, we should be able to code them quite efficiently by using, for example, a move-to-front coder (Section 4.2). For long strings with somewhat larger character sets this technique should compress the string significantly since the same character is likely to appear in similar contexts. Experimentally, in fact, the technique compresses about as well as PPM even though it has no magic number k or magic way to select the escape probabilities.

The problem remains, however, of how to reconstruct the original sequence from context-sorted sequence. The way to do this is the ingenious contribution made by Burrows and Wheeler. You might try to recreate it before reading on. The order of the most-significant characters in the sorted contexts plays an important role in decoding. In the example of Figure 6.1, these are $\mathtt{a}_1\mathtt{a}_4\mathtt{a}_2\mathtt{a}_3\mathtt{b}_2\mathtt{b}_1\mathtt{c}_1\mathtt{c}_3\mathtt{c}_5\mathtt{c}_2\mathtt{c}_4$. The characters are sorted, but equal valued characters do not necessarily appear in the same order as in the input sequence. The following lemma is critical in the algorithm for efficiently reconstruct the sequence.

Lemma 6.1.1 *For the Block-Sorting transform, as long as there are at least two distinct characters in the input, equal valued characters appear in the same order in the most-significant characters of the sorted contexts as in the output (the context sorted sequence).*

Proof: Since the contexts are sorted in reverse lexicographic order, sets of contexts whose most-significant character are equal will be ordered by the remaining context—*i.e.*, the string of all previous characters. Now consider the contexts of the context-sorted sequence. If we drop the least-significant character of these contexts, then they are exactly the same as the remaining context above, and therefore will be sorted into the same ordering. The only time that dropping the least-significant character can make a difference is when all other characters are equal. This can only happen when all characters in the input are equal. □

Based on Lemma 6.1.1, it is not hard to reconstruct the sequence from the context-sorted sequence as long as we are also given the index of the first character to output (the first character in the original input sequence). The algorithm is given by the following code.

function BW_Decode(In,FirstIndex,n)
  S = MoveToFrontDecode(In,n)
  R = Rank(S)
  j = FirstIndex
  for $i = 1$ **to** $n - 1$
    $Out[i] = S[j]$
    $j = R[j]$

For an ordered sequence S, the Rank(S) function returns an sequence of integers specifying for each character $c \in S$ how many characters are either less than c or equal to c and appear before c in S. Another way of saying this is that it specifies the position of the character if it where sorted using a stable sort.

To show how this algorithms works, we consider an example in which the MoveToFront decoder returns S = `ssnasmaisssaai`, and in which FirstIndex = 4 (the first `a`). The example is shown in Figure 14(a). We can generate the most significant characters of the contexts simply by sorting S. The result of the sort is shown in Figure 14(b) along with the

S	
s	
s	
n	
a	$\Leftarrow$
s	
m	
a	
i	
s	
s	
s	
a	
a	
i	

(a)

Sort(S)	S	Rank(S)	
a_1	s_1	9	
a_2	s_2	10	
a_3	n_1	8	
a_4	a_1	1	$\Leftarrow$
i_1	s_3	11	
i_2	m_1	7	
m_1	a_2	2	
n_1	i_1	5	
s_1	s_4	12	
s_2	s_5	13	
s_3	s_6	14	
s_4	a_3	3	
s_5	a_4	4	
s_6	i_2	6	

(b)

	Out	
a_4	a_1	$\Leftarrow$
a_1	s_1	
s_1	s_4	
s_4	a_3	
a_3	n_1	
n_1	i_1	
i_1	s_3	
s_3	s_6	
s_6	i_2	
i_2	m_1	
m_1	a_2	
a_2	s_2	
s_2	s_5	
s_5	a_4	

(c)

Figure 14: Burrows-Wheeler Decoding Example. The decoded message sequence is `assanissimassa`.

rank R. Because of Lemma 6.1.1, we know that equal valued characters will have the same order in this sorted sequence and in S. This is indicated by the subscripts in the figure. Now each row of Figure 14(b) tells us for each character what the next character is. We can therefore simply rebuild the initial sequence by starting at the first character and adding characters one by one, as done by BW_Decode and as illustrated in Figure 14(c).