Exercise on Huffman Codes

Jason Eisner, Spring 1993

This was one of several optional small computational projects assigned to undergraduate mathematics students at Cambridge University in 1993. I’m releasing my code and writeup in 2003 in case they are helpful to anyone—someone working in this area wrote to me asking for them.

Terminology: The project sheet is not quite clear about what to call the components of a segmented message. I have adopted the following terminology instead.

source letter A basic component of the source text: one of the numbers 0, 1, . . . , m − 1. (In this exercise, m = 27, and the source letters are the letters of the Roman alphabet.)

source word A sequence of n source letters. We regard the letters as the digits of an n-digit, base-m number, and represent the word by this number.

code letter In this exercise, one of the bits 0 or 1.

codeword A sequence of bits corresponding to a source word.

bitstring Any sequence of bits—but usually the concatenation of one or more codewords.

dictionary A tree that specifies some map from codewords to source words. If D is a dictionary, with child dictionaries D0 and D1, then

\[ D(b_1b_2b_3\ldots b_k) = D_{b_1}(b_2b_3\ldots b_k). \]

codearray A data structure: an array that holds the main dictionary tree and all its subtrees (also dictionaries). In particular, the leaf corresponding to source word i is stored at the ith element.
Results: I constructed codes for words of length 1, 2, and (heroically) 3 in the same session. Here is the transcript, followed by a few quick calculations:

> ; Build a Huffman code dictionary for the large sample text.
(setf code1 (build-code-from-file "~jrs23/huffman" 27 1))
(Finished reading 467535 messages.)
#<Vector T 53 1694546>

> ; Build another code from the same text, but with 2-letter segments.
(setf code2 (build-code-from-file "~jrs23/huffman" 27 2))
(Finished reading 233768 messages.)
#<Vector T 1457 18829EE>

(setf code3 (build-code-from-file "~jrs23/huffman" 27 3)) ; 3-letter segments
(Finished reading 155845 messages.)
#<Vector T 39365 18C435E>

>(describe-code code1)
Message 0 Sample prob 0.18910 (0 0 0)
Message 1 Sample prob 0.06526 (0 1 0 1)
Message 2 Sample prob 0.01265 (0 1 1 1 1)
Message 3 Sample prob 0.01871 (0 0 1 0 1 0)
Message 4 Sample prob 0.03563 (0 1 0 0 0)
Message 5 Sample prob 0.10116 (1 1 0)
Message 6 Sample prob 0.01642 (0 1 1 1 0 1)
Message 7 Sample prob 0.01736 (0 1 1 1 0 0)
Message 8 Sample prob 0.04875 (1 0 1 1)
Message 9 Sample prob 0.05817 (1 0 0 0)
Message 10 Sample prob 0.00137 (0 0 1 0 0 1 1)
Message 11 Sample prob 0.00760 (0 0 1 0 0 1 1)
Message 12 Sample prob 0.03456 (0 1 0 0 1)
Message 13 Sample prob 0.02116 (1 1 0 1 1)
Message 14 Sample prob 0.05591 (1 0 0 1)
Message 15 Sample prob 0.06167 (0 1 1 0)
Message 16 Sample prob 0.01331 (0 1 1 1 1 0)
Message 17 Sample prob 0.00079 (0 0 1 0 0 1 0 1 0)
Message 18 Sample prob 0.04478 (1 1 1 1)
Message 19 Sample prob 0.05263 (1 1 0)
Message 20 Sample prob 0.07310 (0 0 1 1)
Message 21 Sample prob 0.02388 (1 1 1 0)
Message 22 Sample prob 0.00742 (0 0 1 0 1 0 0)
Message 23 Sample prob 0.01922 (0 0 1 0 0 0)
Message 24 Sample prob 0.00135 (0 0 1 0 0 1 0 0)
Message 25 Sample prob 0.01765 (0 0 1 0 1 1)
Message 26 Sample prob 0.00041 (0 0 1 0 0 1 0 1 1)

Sorted by probability:

Message 0 Sample prob 0.18910 (0 0 0)
Message 1 Sample prob 0.10116 (1 1 0)
Message 2 Sample prob 0.07310 (0 0 1 1)
Message 3 Sample prob 0.06526 (0 1 0 1)
Message 4 Sample prob 0.06167 (0 1 1 0)
Message 5 Sample prob 0.05817 (1 0 0 0)
Message 6 Sample prob 0.05591 (1 0 0 1)
<table>
<thead>
<tr>
<th>Message</th>
<th>Sample prob</th>
<th>(codes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>0.05263</td>
<td>(1 0 1 0)</td>
</tr>
<tr>
<td>8</td>
<td>0.04875</td>
<td>(1 0 1 1)</td>
</tr>
<tr>
<td>18</td>
<td>0.04478</td>
<td>(1 1 1 1)</td>
</tr>
<tr>
<td>4</td>
<td>0.03563</td>
<td>(0 1 0 0 0)</td>
</tr>
<tr>
<td>12</td>
<td>0.03466</td>
<td>(0 1 0 0 1)</td>
</tr>
<tr>
<td>21</td>
<td>0.02388</td>
<td>(1 1 1 0 0)</td>
</tr>
<tr>
<td>13</td>
<td>0.02116</td>
<td>(1 1 1 0 1)</td>
</tr>
<tr>
<td>23</td>
<td>0.01922</td>
<td>(0 0 1 0 0 0)</td>
</tr>
<tr>
<td>3</td>
<td>0.01871</td>
<td>(0 0 1 0 1 0)</td>
</tr>
<tr>
<td>25</td>
<td>0.01765</td>
<td>(0 0 1 0 1 1)</td>
</tr>
<tr>
<td>7</td>
<td>0.01736</td>
<td>(0 1 1 1 0 0)</td>
</tr>
<tr>
<td>6</td>
<td>0.01642</td>
<td>(0 1 1 1 0 1)</td>
</tr>
<tr>
<td>16</td>
<td>0.01331</td>
<td>(0 1 1 1 1 0)</td>
</tr>
<tr>
<td>2</td>
<td>0.01265</td>
<td>(0 1 1 1 1 1)</td>
</tr>
<tr>
<td>11</td>
<td>0.00750</td>
<td>(0 0 1 0 0 1 1)</td>
</tr>
<tr>
<td>22</td>
<td>0.00742</td>
<td>(0 0 1 0 0 1 0 0)</td>
</tr>
<tr>
<td>10</td>
<td>0.00137</td>
<td>(0 0 1 0 0 1 0 1 1)</td>
</tr>
<tr>
<td>24</td>
<td>0.00135</td>
<td>(0 0 1 0 0 1 0 1 0 0)</td>
</tr>
<tr>
<td>17</td>
<td>0.00079</td>
<td>(0 0 1 0 0 1 0 1 0 1 0)</td>
</tr>
<tr>
<td>26</td>
<td>0.00041</td>
<td>(0 0 1 0 0 1 0 1 0 1 1)</td>
</tr>
</tbody>
</table>

Sample text consisted of 467535 messages.
Text would be encoded into 1931572 bits (average codeword length 4.13140).
Source entropy: 4.08893.

(describe-code code2 :brief? t)

Sample text consisted of 233768 messages.
Text would be encoded into 1746788 bits (average codeword length 7.47231).
Source entropy: 7.43955.

(describe-code code3 :brief? t)

Sample text consisted of 155845 messages.
Text would be encoded into 1583011 bits (average codeword length 10.15760).
Source entropy: 10.12947.

(log 27 2);; bits/letter for FIXED-length code (if 27 were a 2-power)
4.754887502163469

(/ 4.13140 4.75488);; compression ratio achieved by Huffman encoding
0.8688757655293089

(/ 7.47231 4.75488 2);;; ... by encoding pairs of letters
0.7857516908944074

(/ 10.15760 4.75488 3);;; ... by encoding triples of letters
0.712082464050968

(stringify '(0 5 20 1 15 9 14 19 8 18 4 12 21 13 23 3 25 7 6 16 2 11 22 10 24 17 26))
"etaoinshrdlumwcygfbpvkjxqz"

;; the string just printed gives the frequency ordering of the letters.

Note the following features of the output:

- More frequent source words have shorter codewords, as required.
- The average codeword length is barely above the source entropy in each case. The code is a very good one.
For longer segments, the source entropy per letter drops. This enables us to get even better compression.

After the space, the letters have the famous English frequency ordering “ETAOINSHRDLU…”

**Encoding and decoding text:** I have written a few routines to actually use the Huffman code, since it seemed a waste to stop at finding the expected codeword length. The transcript below tests the 1- and 2-letter codes that were created from the long sample text. “*” refers to the result of the previous line.

```
> (setf lazy "the quick brown fox jumps over the lazy dog"
  cues "ETAOINSHRDLU..."
  hero "claude shannon the mathematician is my hero")
"claude shannon the mathematician is my hero"
> (numerify lazy)
(20 8 5 0 17 21 9 3 11 0 2 18 15 23 14 0 6 15 24 0 10 21 13 16 19 0 15 22 5 18 0
20 8 5 0 12 1 26 25 0 4 15 7)
> (encode-letters code1 * 27 1)
(0 0 1 1 0 1 1 1 0 0 0 0 0 0 0 1 0 0 1 0 1 0 1 0 1 0 1 1 0 0 1 0 0 1 0 0 1 0 0 0 0 1 1
1 0 1 0 1 1 0 0 0 1 0 1 0 1 0 0 0 0 0 0 1 0 0 1 0 0 1 1 0 0 1 0 0 0 0 1 1 1 0
1 1 1 1 0 1 0 1 0 0 0 0 1 1 0 0 1 0 0 1 0 0 1 0 1 1 1 0 0 0 0 1 1 1 0
1 1 1 1 0 0 0 0 1 0 1 0 1 0 1 0 0 1 0 1 1 0 0 1 0 1 0 1 1 0 0 0 0 1 0
0 0 0 1 1 0 0 1 1 1 0
> (decode-into-letters code1 * 27 1)
(20 8 5 0 17 21 9 3 11 0 2 18 15 23 14 0 6 15 24 0 10 21 13 16 19 0 15 22 5 18 0
20 8 5 0 12 1 26 25 0 4 15 7)
> (stringify *)
"the quick brown fox jumps over the lazy dog"
> (numerify hero)
(3 12 1 21 4 5 0 19 8 1 14 14 15 14 0 20 8 5 0 13 1 20 8 5 13 1 20 9 3 9 1 14 0
9 19 0 13 25 0 8 5 18 15)
> (encode-letters code2 * 27 2)
(1 1 0 1 0 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 0 1 1 0 1 1 0 0 1 0 1 0 1 1 1
1 0 0 0 0 0 1 1 0 1 1 0 0 0 0 1 0 1 1 1 1 1 0 0 1 1 1 1 0 0 1 0 1 1 0
0 1 1 1 1 1 1 0 1 0 0 0 1 1 1 1 0 1 0 0 1 0 1 0 0 1 1 1 0 1 1 0 0 1 0
0 1 1 0 1 0 0 0 1 0 1 1 0 0 1 1 1 1 0 1 1 0 1 1 1 1 0 1 1 0 1 0 1 0
0
> (decode-into-letters code2 * 27 2)
(3 12 1 21 4 5 0 19 8 1 14 14 15 14 0 20 8 5 0 13 1 20 8 5 13 1 20 9 3 9 1 14 0
9 19 0 13 25 0 8 5 18 15 0)
> (stringify *)
"claude shannon the mathematician is my hero"
```

Note the extra space that got added to Claude Shannon’s string. The word-reading routine padded the input in order to come out with an integral number of 2-letter words.¹

¹If we were compressing real files, we would have to store the file length so that we could chop off any such superfluous spaces upon decompression.
It is instructive to compare the lengths of the bitstrings:

\[
\begin{align*}
&> (\text{length (encode-letters code1 (numerify lazy) 27 1)}) \\
&\quad \quad 212 \\
&> (\text{length (encode-letters code1 (numerify cues) 27 1)}) \\
&\quad \quad 473 \\
&> (\text{length (encode-letters code1 (numerify hero) 27 1)}) \\
&\quad \quad 174 \\
&> (\text{length (encode-letters code2 (numerify lazy) 27 2)}) \\
&\quad \quad 180 \\
&> (\text{length (encode-letters code2 (numerify cues) 27 2)}) \\
&\quad \quad 572 \\
&> (\text{length (encode-letters code2 (numerify hero) 27 2)}) \\
&\quad \quad 161
\end{align*}
\]

The sentence about Claude Shannon is typical English, and code1 compresses it the most. The lazy dog sentence is not too unusual, but contains all the letters of the alphabet. The anomalous cues string is compressed least—if anything it is expanded, to 11 bits per letter.

When we use code2, which compresses pairs of letters, these effects are exaggerated. The English sentences, lazy and hero, turn into even shorter bitstrings. But “qqqq...” is expanded even more: neither “qq” nor “q ” appear at all in the text file, and both have 26-bit codewords!

**Shannon-Fano code:** In this code, a source word with probability \( p_i \) has a codeword of \( \lceil -\log p_i \rceil \) bits. The expected codeword length is therefore

\[
\sum_i p_i \lceil -\log p_i \rceil.
\]

For the unsegmented sample text, the Shannon-Fano code would have an average codeword length of

\[
> (\text{Shannon-Fano code1}) \\
4.63484017239351
\]

bits per letter. This compares unfavorably with 4.13140 for the Huffman code.

Note that since \( a \leq \lceil a \rceil < a + 1 \), the expected Shannon-Fano codeword length must fall between \( h \) and \( h + 1 \), where \( h = 4.08893 \) is the entropy of the source.

**Segmenting the Shannon-Fano code:** Consider segmenting a Bernoulli source. A word of \( n \) letters has the form \( i_1i_2i_3...i_n \), with probability \( p_{i_1}p_{i_2}p_{i_3}...p_{i_n} \) (by independence of the letters). The average Shannon-Fano codeword therefore has length

\[
\sum_{i_1} \sum_{i_2} \cdots \sum_{i_n} \lceil -\log p_{i_1}p_{i_2}...p_{i_n} \rceil p_{i_1}p_{i_2}...p_{i_n}
\]
\[
\sum_{i_1} \sum_{i_2} \cdots \sum_{i_n} [\log p_{i_1} - \log p_{i_2} - \cdots - \log p_{i_n}] p_{i_1} p_{i_2} \cdots p_{i_n}.
\]

But
\[
[a] + [b] - 1 \leq [a + b] \leq [a] + [b],
\]
so using induction, the average codeword length is at most
\[
\sum_{i_1} \sum_{i_2} \cdots \sum_{i_n} (-\log p_{i_1} + [-\log p_{i_2}] + \cdots + [\log p_{i_n}]) p_{i_1} p_{i_2} \cdots p_{i_n}
= n \cdot \sum_i [-\log p_i] p_i
\]
and at least
\[
n \cdot \sum_i [-\log p_i] p_i - n.
\]

So if the unsegmented code uses \( k \) bits per source letter on average, then the segmented version uses between \( nk \) and \( nk - n \) for a source word of \( n \) letters.

In short, segmentation can’t hurt, but it will save us at most one bit per source letter.

**Another way to see the same thing:** As remarked earlier, the expected codeword length for the Shannon-Fano code is between \( h \) and \( h + 1 \).

If we drop the ceiling operator (“\( \lceil \rceil \)” ) from the calculation above, we see that segmentation multiplies the per-word source entropy of a Bernoulli source by exactly \( n \).

So under these conditions, the Shannon-Fano code goes from an average of \( h \) to \( h + 1 \) bits per letter (unsegmented) to \( h \) to \( h + (1/n) \) bits per letter (segmented).

**Segmenting the Huffman code:** The Shannon-Fano code is a prefix-free code, and the Huffman code is an *optimal* prefix-free code—its average codeword length is minimal. So the Huffman codeword length falls between the entropy and the Shannon-Fano codeword length.

In particular, on a Bernoulli source, the unsegmented Huffman code always takes between \( h \) and \( h + 1 \) bits per letter (on average); an \( n \)-segmented version will do no worse than the unsegmented one, and always takes between \( h \) and \( h + (1/n) \) bits per letter.

For all these Bernoulli sources, segmentation is just a way to avoid the “fractional bit” problem. For example, if the letter “e” has probability 0.1, as it does in English text, then it deserves about 2.4 bits. The Shannon-Fano
code, as well as the Huffman code constructed gives it 3. Lumping several letters together is a way to cut this waste.

Non-Bernoulli sources: For a non-Bernoulli source, segmentation can make an arbitrarily large improvement. Consider an equiprobable source alphabet of $2^k$ letters. Without segmentation, the source entropy is $k$; and in both the Huffman and Shannon-Fano codes, every codeword is $k$ bits long. But suppose the source just recites the alphabet over and over again: “0, 1, 2, ..., $(2^k - 2), (2^k - 1), 0, 1, 2, ...”” Then taking words of length $2^k$ yields an entropy of 0.

It is interesting to consider what the Huffman code looks like in this case. If we believe that we have a single source word, of probability 1, then there will be a single codeword, of length 0. We can reproduce the source with no information at all (other than the dictionary itself).

On the other hand, we may worry that the source will eventually grow up and say something other than the alphabet. In this case, we will want to construct codewords for all the words we have never seen—in case they do appear. The codeword for the word of probability 1 will then be “0,” and all the other codewords will start with “1.”

The real issue here is whether or not we have privileged knowledge of the source. If we know the true probability of each source word, then we are justified in ignoring words of probability 0. We may have such privileged knowledge if the source is a Geiger counter, and we certainly have it if we are building a one-time code to compress a given text.

In my code, I have assumed that we are merely sampling the source, and that the sample probabilities may disagree with the true probabilities. So I do build codewords for source words that do not appear in the sample (like “qq” in 2-segmented English).

It is possible to carry this view further. For example, if the source word $W$ occurs $a$ times in a sample text of $b$ words, we might not guess its true probability to be $a/b$. A maximum-likelihood estimate would be $(a + 1)/(b + 2)$, for example! For various reasons, I have not taken this radical approach in my program—nor is it likely to make a difference for the large sample text provided.

Why isn’t English text Bernoulli? Successive letters in English are hardly independent. I’ll itemize a few of the obvious dependencies:

- Orthography (everyone’s favourite example): The probability of “u” following “q” is about 1, not 0.024 (its unconditional probability).
• Pronunciation: A sequence of three consonants is almost always followed by a vowel or a space, whereas a sequence of two vowels is almost always followed by a consonant.

• Words versus non-words: “elbo” is almost always followed by “w,” although “w” has a low unconditional probability.

How Bernoulli isn’t it? We can measure the non-independence of adjacent messages by considering the following question: How much information about a letter do we get by knowing its predecessor?

Let \( p_{ij} \) be the probability of the word \( ij \). If \( i \) is a randomly selected letter, then the next letter is \( j \) with probability \( p_{ij}/p_i \). If we know what \( i \) is, then our uncertainty about \( j \) is only

\[
\sum_j -\frac{p_{ij}}{p_i} \log \frac{p_{ij}}{p_i}
\]

bits.

Taking a weighted average over all \( i \), we get

\[
\sum_i p_i \sum_j (\frac{-p_{ij}}{p_i} \log \frac{p_{ij}}{p_i})
\]

\[
= \sum_i (\sum_j -p_{ij} \log p_{ij} - \sum_j -p_{ij} \log p_i)
\]

\[
= \sum_i (\sum_j -p_{ij} \log p_{ij} - (-p_i \log p_i))
\]

\[
= \sum_i \sum_j -p_{ij} \log p_{ij} - \sum_i -p_i \log p_i
\]

\[
= \text{segmented entropy} - \text{unsegmented entropy}
\]

for the average number of bits of information carried by a letter if we know its predecessor.

If we don’t know a letter’s predecessor, the average number of bits of information it carries is just the unsegmented entropy. So the information gain from knowing the predecessor is twice the unsegmented entropy, minus the segmented entropy.

In the case of our English text sample, this is

\[
2 \cdot 4.08893 - 7.43955 = 0.73831
\]
bits. Informally, we can say that knowing a letter halves the possible choices
for its successor, or makes it twice as easy to guess its successor. By contrast,
in a Bernoulli text, this information gain would be zero.

**A Huffman code for Bernoulli-English:** It is easy enough to create
a set of word probabilities that mirror those of an English-like Bernoulli
source. All we need to do is to multiply the letter probabilities (or letter
counts) gathered from the sample text, as if they were independent.

We can construct a Huffman code for the resulting source words. I have
done so for words of length 2 and 3. Here’s what the codes look like:

```lisp
> (setf code2B (build-Bernoulli-code code1 27 2))
(Finished pretending to read 218588976225 messages.)
#<Vector T 1457 1538686>
> (describe-code code2B :brief? t)
Sample text consisted of 218588976225 messages.
Text would be encoded into 1793970407197 bits (average codeword length 8.20705).
Source entropy: 8.17787.
> (setf code3B (build-Bernoulli-code code1 27 3))
(Finished pretending to read 1021979969935375 messages.)
#<Vector T 39365 184FA06>
> (describe-code code3B :brief? t)
Sample text consisted of 1021979969935375 messages.
Text would be encoded into 125658839778562382 bits (average codeword length 12.29563).
Source entropy: 12.26680.
```

Note that the source entropy per word is exactly double or triple the
unsegmented source entropy, 4.08893. So the information per letter remains
unchanged under segmenting, just as claimed for a Bernoulli source.

The number of bits per letter drops from 4.13140 (1-letter segments)
to 4.10353 (2-letter Bernoulli) and 4.09667 (3-letter Bernoulli). The com-
pression is closer to the ideal—simply because with longer codewords, the
fractional bit problem is less severe.

**Computational complexity:** How fast is the algorithm to build a
Huffman code? Once all the message probabilities are known—reading all
the messages takes time proportional to the number of letters in the sam-
ple text—the algorithm must merge the messages together. Each merge
decreases the number of messages by 1; so if there are $k$ messages to start
with, the program needs to do $k - 1$ merges.

How long does a merge take? In this implementation, the program must
search through $O(k)$ candidate messages to find two of minimal probability.
Once these are found, merging them and updating the codearray takes only
$O(1)$ time.

So the current implementation runs in time $O(k^2)$. With cleverer data
structures, one can improve this. Using a priority queue implemented as
a binary heap, one can dequeue two messages and enqueue their merge in $O(\log k)$ time. So the overall algorithm can be made to take time $O(k \log k)$.

Note that once a Huffman code is available, we can encode and decode texts in time linear on the length of the text.