27.1 A lower bound on data compression

How much can we compress a file without loss? We present a lower bound for *any* compression algorithm under the following assumptions:

> the file contains m characters
> there are c different characters possible
> character i appears exactly $f(i)$ times in the file

Note that $\sum_{i=1}^{c} f(i) = m$ by construction.

Let F be the set of all possible files satisfying the above criteria. Our approach is simply to count the number $|F|$ of possible files and note that at least $\log_2 |F|$ bits are needed to distinguish among all these possibilities.

We need two results, both from Math 55. The first gives a formula for $|F|$:

$$|F| = \frac{m!}{f(1)! \cdot f(2)! \cdots f(c)!}$$

Here is a sketch of the proof of this formula. There are $m!$ permutations of m characters, but many are the same because there are only c different characters. In particular, the $f(1)$ appearances of character 1 are the same, so all $f(1)!$ orderings of these locations are identical. Thus we need to divide $m!$ by $f(1)!$. The same argument leads us to divide by all other $f(i)!$.

Now we have an exact formula for $|F|$, but it is hard to interpret, so we replace it by a simpler approximation. We need a second result from Math 55, namely Stirling's formula for approximating $n!$:

$$n! \approx \sqrt{2\pi} n^{n+.5} e^{-n}$$

This is a good approximation in the sense that the ratio $n!/[\sqrt{2\pi} n^{n+.5} e^{-n}]$ approaches 1 quickly as n grows. (In Math 55 we motivated this formula by the approximation $\log n! = \sum_{i=2}^{n} \log i \approx \int_1^n \log x dx$.) We will use Stirling's formula in the form

$$\log_2 n! \approx \log_2 \sqrt{2\pi} + (n + .5) \log_2 n - n \log_2 e$$

Stirling's formula is accurate for large arguments, so we will be interested in approximating $\log_2 |F|$ for large m. Furthermore, we will actually estimate $\frac{\log_2 |F|}{m}$, which can be interpreted as the *average number of bits per character* to send a long file. Here goes:

$$\begin{aligned}
\frac{\log_2 |F|}{m} &= \frac{\log_2(m!/(f(1)! \cdots f(c)!))}{m} \\
&= \frac{\log_2 m! - \sum_{i=1}^{c} \log_2 f(i)!}{m} \\
&\approx \frac{1}{m} \cdot [\log_2 \sqrt{2\pi} + (m + .5) \log_2 m - m \log_2 e \\
&\quad - \sum_{i=1}^{c} (\log_2 \sqrt{2\pi} + (f(i) + .5) \log_2 f(i) - f(i) \log_2 e)] \\
&= \frac{1}{m} \cdot [m \log_2 m - \sum_{i=1}^{c} f(i) \log_2 f(i) \\
&\quad + (1 - c) \log_2 \sqrt{2\pi} + .5 \log_2 m - .5 \sum_{i=1}^{c} \log_2 f(i)]
\end{aligned}$$

$$\begin{aligned} &= \log_2 m - \sum_{i=1}^{c} \frac{f(i)}{m} \log_2 f(i) \\ &\quad + \frac{(1-c)\log_2 \sqrt{2\pi}}{m} + \frac{.5 \log_2 m}{m} - \frac{.5 \sum_{i=1}^{c} \log_2 f(i)}{m} \end{aligned}$$

As m gets large, the three fractions on the last line above all go to zero: the first term looks like $O(1/m)$, and the last two terms look like $O(\frac{\log_2 m}{m})$. This lets us simplify to get

$$\begin{aligned} \frac{\log_2 |F|}{m} &\approx \log_2 m - \sum_{i=1}^{c} \frac{f(i)}{m} \log_2 f(i) \\ &= (\sum_{i=1}^{c} \frac{f(i)}{m}) \log_2 m - \sum_{i=1}^{c} \frac{f(i)}{m} \log_2 f(i) \\ &= -\sum_{i=1}^{c} \frac{f(i)}{m} \log_2 \frac{f(i)}{m} \end{aligned}$$

To simplify notation slightly, we define $p_i = f(i)/m$, which we may describe as *the fraction of characters in the file equaling the i-th character*, or *the probability that any given character in the file is character i.* This finally yields

$$\frac{\log_2 |F|}{m} \approx -\sum_{i=1}^{c} p_i \log_2 p_i$$

as an (approximate) lower bound on the number of bits per character that any encoding scheme would need to represent F. This important quantity is called the *entropy* H (Shannon, 1949).

$$H = -\sum_{i=1}^{c} p_i \log_2 p_i$$

Said another way, the number of bits needed to encode file F is at least Hm, where m is the length of F. This result is also due to Shannon (1949), who invented the field of *information theory.*

Let's apply this to the example of a file consisting of $c = 2^k$ characters, each appearing equally often. Thus $p_i = 2^{-k}$ for $i = 1$ to 2^k, and we may easily evaluate $H = k$. In other words, there is no better scheme than the obvious one of representing each character by a k-bit number between 0 and $2^k - 1$. You can confirm that Huffman's algorithm would yield this encoding.

If we apply this to the example of the last section, a file F of 100K characters from a through h with the probabilities given in a table, we can compute that a lower bound on the number of bits to represent F is $Hm = 222K$. Since Huffman coding yielded 224K bits, we see that Huffman got very close to the lower bound.

How much more space can Huffman coding take to encode a file than Shannon's lower bound Hm? A theorem of Gallagher (1978) shows that at worst Huffman will take $m \cdot (p_{max} + .086)$ bits more than Hm, where p_{max} is the largest of any p_i. But it often does much better, as illustrated by the example in the last paragraph.

27.2 Other Data Compression Schemes

Earlier we claimed the Huffman coding was "optimal", but this was under several assumptions:

1. The compression is lossless, i.e. uncompressing the compressed file yields exactly the original file. When lossy compression is permitted, as for video, other algorithms can achieve much greater compression, and this is a very active area of research because people want to be able to send video and audio over the Web.

2. We know all the frequencies $f(i)$ with which each character appears. How do we get this information? We could make two passes over the data, the first to compute the $f(i)$, and the second to encode the file. But this can be much more expensive than passing over the data once for large files residing on disk or tape. One way to do just one pass over the data is to assume that the fractions $f(i)/m$ of each character in the file are similar to files you've compressed before. For example you could assume all Java programs (or English text, or PowerPoint files, or ...) have about the same fractions of characters appearing. A second cleverer way is to estimate the fractions $f(i)/m$ on the fly as you process the file. One can make Huffman coding adaptive this way.

3. We know the set of characters (the alphabet) appearing in the file. This may seem obvious, but there is a lot of freedom of choice. For example, the alphabet could be the characters on a keyboard, or they could be the key words and variables names appearing in a program. To see what difference this can make, suppose we have a file consisting of n strings *aaaa* and n strings *bbbb* concatenated in some order. If we choose the alphabet $\{a, b\}$ then $8n$ bits are needed to encode the file. But if we choose the alphabet $\{aaaa, bbbb\}$ then only $2n$ bits are needed.

Picking the correct alphabet turns out to be crucial in practical compression algorithms. Both the UNIX compress and GNU gzip algorithms use a greedy algorithm due to Lempel and Ziv to compute a good alphabet in one pass while compressing. Here is how it works.

If s and t are two bit strings, we will use the notation $s + +t$ to mean the bit string gotten by concatenating s and t.

We let F be the file we want to compress, and think of it just as a string of bits, that is 0's and 1's. We will build an alphabet A of common bit strings encountered in F, and use it to compress F. Given A, we will break F into shorter bit strings like

$$F = A(1){+}{+}0{+}{+}A(2){+}{+}1{+}{+}\cdots{+}{+}A(7){+}{+}0{+}{+}\cdots{+}{+}A(5){+}{+}1{+}{+}\cdots{+}{+}A(i){+}{+}j{+}{+}\cdots$$

and encode this by

$$1 + +0 + +2 + +1 + + \cdots + +7 + +0 + + \cdots + +5 + +1 + + \cdots + +i + +j + + \cdots$$

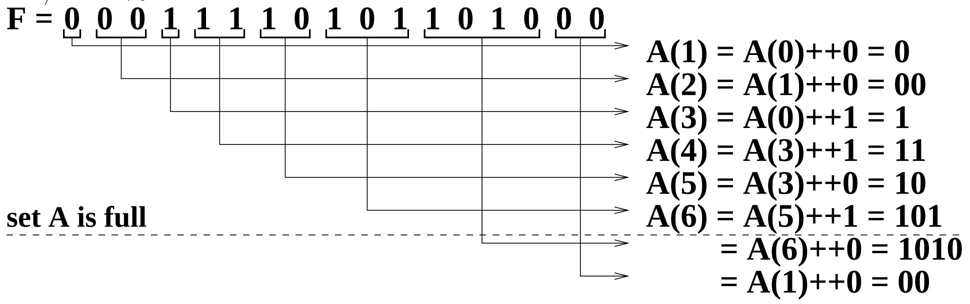
The indices i of $A(i)$ are in turn encoded as fixed length binary integers, and the bits j are just bits. Given the fixed length (say r) of the binary integers, we decode by taking every group of $r + 1$ bits of a compressed file, using the first r bits to look up a string in A, and concatenating the last bit. So when storing (or sending) an encoded file, a header containing A is also stored (or sent).

Here is the algorithm for encoding, including building A. Typically a fixed size is available for A, and once it fills up, the algorithm stops looking for new characters.

```
$A = \{\emptyset\}$ ... start with an alphabet containing only an empty string
$i = 0$ ... points to next place in file $F$ to start encoding
repeat
    find $A(k)$ in the current alphabet that matches as many leading bits $F_i F_{i+1} F_{i+2} \cdots$ as possible
        ... initially only $A(0)$ = empty string matches
        ... Let $b$ be the number of bits in $A(k)$
    if $A$ is not full, add $A(k) + +F_{i+b}$ to $A$
        ... $F_{i+b}$ is the first bit unmatched by $A(k)$
    output $k + +F_{i+b}$
    $i = i + b + 1$
until $i > length(F)$
```

Note that A is built "greedily", based on the beginning of the file. Thus there are no optimality guarantees for this algorithm. It can perform badly if the nature of the file changes substantially after A is filled up.

Here is an example of the algorithm running, where the alphabet A fills up after 6 characters are inserted. In this small example no compression is obtained, but if A were large, and the same long bit strings appeared frequently, compression would be substantial. The gzip manpage claims that source code and English text is typically compressed 60%-70%. For example, gzip compressed the latex source file of this section of the notes from 33142 bytes to 12300 bytes, or 63%.



Encoded F = (0,0),(1,0),(0,1), (3,1),(3,0), (5,1),(6,0),(1,0)
= 0000 0010 0001 0111 0110 1011 1100 0010