1 Low probability of lossless compression

One way to measure compressibility is to compute the ratio of the number of output symbols to the number of input symbols. The lower the ratio is, the better the compression. There are a few subtleties here.

First, if the output alphabet is bigger than the input alphabet, then there may be an illusory compression. For example, if the input alphabet is binary and the output alphabet has 4 symbols and we output one such quarternary symbol for every 2 bits of input, then by the above definition, we have achieved compression ratio 0.5. But this is pointless - if we count the symbols output in terms of the input alphabet, we see that we need 2 bits to encode each quarternary symbol, so there is no compression at all.

Second, the input must be recoverable from the output. It is very easy to define the best compressor according to the above definition - build an automaton that reads finite strings, and outputs the empty string. This will achieve 0 compressibility, the best possible. Unfortunately, it is impossible to recover the input from the output, so there is no decompressor corresponding to this proposed compressor.

We will eventually define a notion of a lossless compressor which handles these issues. For now, let us consider a minimal setup to discuss lossless compression.

Let $\Sigma = \{0, 1\}$ be our finite alphabet, both for input and output. Let $f : \Sigma^* \to \Sigma^*$ be the compressor. Now, we abstract the requirement of a decompressor by saying that the compressor f must be a 1-1 function. This implies that if a compressed string w has an expansion $f^{-1}(w)$, then the expansion will be unique. This handles our second objection in an abstract manner - we say that since the function is 1-1, multiple strings cannot be mapped to the empty string as in the second issue.

In practical terms, the algorithms for, and the computational complexities of the compressor and the decompressor are important, but the abstract setting that there *exists* a compressor and a decompressor will serve for now. Note that we do not require that the function is onto - some strings may not be compressed versions of any other string. These may be thought of as "corrupted archives" - they have no valid preimage.

This setting is broad, and it is clear that any lossless compression scheme has to be such a 1-1 function. This leads us to the first result in the course - that lossless compression is impossible with high probability.

We consider the standard enumeration of the strings in Σ^*, and the relation $a < b$ means that a occurs before b in the standard enumeration of strings.

Lemma 1.1. *Let $f : \Sigma^* \to \Sigma^*$ be a 1-1 function. Then the following holds for all numbers c and all large enough numbers n.*

$$\textit{Probability}\{w \mid w < n, |w| - |f(w)| > c\} \leq \frac{1}{2^{c-1}}.$$

The lemma says, for instance, that asymptotically at most half of the first n strings in the standard order can be compressed by at least 2 bits, at most a quarter of them can be compressed by at least 3 bits, and so on. Consequently, strings which can be compressed to half their length are going to be very rare. No matter how clever the scheme, as long as it is lossless, it doesn't compress many strings.

Proof. Consider the set

$$S = \{w \mid w < n, |f(w)| < |w| - c\}$$

Let M be the length of the longest word in $[n]$. Then the number of strings in $[n]$ is at most 2^{M-1}.

Since f is 1-1, we have that

$$\text{cardinality}(f(S)) = \text{cardinality}(\{f(w) \mid w \in S\}) \leq 2^{M-c}.$$

Thus the probability of S is

$$\frac{\textit{cardinality}(S)}{n} \leq \frac{2^{M-c}}{2^{M-1}} = \frac{1}{2^{c-1}}.$$

□

This implies that only very few strings can be highly compressed. But here is the important observation - most lossless compressors in practice compress all "useful strings". Most of the n-length strings are going to look fairly random - they are not encodings of any useful data like the weather readings at a location. The above lemma says that most strings cannot be compressed. Useful strings are not very many - is it possible that the few strings that a lossless compressor compresses, encompass all the useful strings we care about?

Happily, this seems to be the case, and this is why we study a class of algorithms that seem to fail with high probability. For many of the popular compressors, the low probability set of compressible strings happens to be a superset of the low probability set of "useful data". Note that our notion of utility of data seems independent of any particular compressor, but the success probability set is dependent on which particular compressor we choose.

2 General Approaches

There are, broadly, two approaches to defining a lossless compressor. One is via non-adaptive schemes - they are based on some assumed property of strings, and do not vary with the particular strings we consider. Morse code or ASCII encoding are good examples of this - no matter what the string, the encoding of a letter remains the same.

The more successful algorithms are adaptive - they may be tailored to exploit some reguarity in the particular string being compressed.

3 Ad-hoc compression

We will first see some non-adaptive schemes, which are simple to understand and set the stage for more sophisticated algorithms to follow. Some of these may also be used as a sub-stage in more sophisticated adaptive schemes.

3.1 Run-Length Encoding

One of the simplest schemes for compression in an invertible manner is to replace sequences of the same character in the text ("runs") by a pair (character,count). Suppose our alphabet consists of lowercase English letters, digits and spaces. If we do not separate the character from its count, then we will not know whether a digit is a count or a character in the text. For example, the encoding of `aa2` would be `a22` which could be a representation of a sequence of 22 `a`s. We could pick a special character, say ';' which never appears in the text, and use it to indicate that the next character is part of a count. For example, the previous example could be encoded as `a2;2`.

There are several variants of this simple scheme. For example, if we are encoding ASCII text in ASCII RLE form, if the sequence of the same character is less than 256 long, we can represent the count in a single byte, rather than the worst-case 3 bytes it would consume for writing such a number as separate ASCII digits. We will use a separator between a character and its count (slightly different from the previous scheme). For example, (keeping in mind that the ASCII code for 'a' is 97, that of '2' is 50 and that of ';' is 59), `aaaa2` can be encoded as `0x61 0x4B 0x4 0x42` representing `a;42`. There is no ambiguity since every count is forced to be within a single ASCII byte.

RLE is effective when the alphabet is small and the characters in the data depend on their predecessors.

Exercises

1. Suppose you have the binary alphabet, and the bits are drawn uniformly at random. What is the expected length of a run? What happens to the expected run length for larger alphabets?

3.2 Differential Coding

This compression scheme is suited especially for sorted data. Suppose a sequence of very large values, but whose successive entries are fairly close to each other. An example can be the successive barometer readings during a day - a "falling barometer reading" indicates a chance of rain or storm and a "rising barometer reading" indicates fair weather. Typically readings are taken every few hours. The adjacent readings usually are very close in range, but the values of the barometer readings can be numbers around 1000 (millibars). Then we could encode a sequence of readings like `1000, 976, 974, ...` by the sequence of first reading followed by successive differences as `1000, -24, -2, ....` This encoding is called *differential coding.*

A variant of differential coding can be used for lexicographically sorted data. This is called *front coding.* In this case, we have a sequence of words whose adjacent entries usually share common prefixes. In this case, the encoding is defined as follows. The first word is encoded as such. Subsequent words are encoded as n.suffix where n is the length of the prefix it has in common with its predecessor, and suffix is the remainder of the word. An example is shown below.

word	encoding
abrupt	abrupt
abrupted	6.ed
abruptedly	8.ly
abruption	6.ion
abruptly	6.ly
abscede	2.scede

Exercise

1. Suppose a sequence of words is lexicographically sorted. Show that the longest common prefix of a word with its ancestors is obtained when comparing with its immediately preceding word.
2. (Verify. ?) Suppose we are given the encoded sequence of words, and some entry in the front encoding is n,suffix. Show that it is possible to reconstruct the word by reconstructing at most n words in the encoding, independent of the position of the target word in the sequence. (*Hint: Go through the list of prefix lengths. Show that the minimum of the lengths is the common prefix our target word has in common with the first word. Try to extend this observation.*)

3.3 Move-to-Front Coding

This is a scheme to encode a sequence of words. The property it exploits is that a word which occurs in a "natural" sentence is likely to occur again soon. This scheme is very similar to implementations of the Least Recently Used schemes for Page replacement in operating systems. These schemes rely on the heuristic that patterns in the recent past will tend to occur in the near future.

The encoding scheme is as follows. We keep track of the following data structures.

1. The sentence and the next word to be encoded.
2. An ordered list of words, sorted in such a way that a word seen more recently precedes all others seen earlier. Thus the next word will go to the front of the list.
3. The output, which is a sequence of words, and digits indicating positions of the words in the list.

Suppose word w occurs in a sentence for the first time at position i. Then we write w followed by a space and then i on the output stream. Further, we add w to the front of the recently-used list.

If the next word w is already present in the list, we write only its index in the list on the output stream. After this, w is moved to the front of the list.

The encoded version of the string is just the output stream. An example is given below.

```
sentence : to be or not to be

list : Step 1 (to)               output: 1 to
       Step 2 (be,to)            output: 1 to 2 be
       Step 3 (or, be, to)       output: 1 to 2 be 3 or
       Step 4 (not, or, be, to)  output: 1 to 2 be 3 or 4 not
       Step 5 (to, not, or, be)  output: 1 to 2 be 3 or 4 not 4
       Step 6 (be, to, not, or)  output: 1 to 2 be 3 or 4 not 4 4
```

Exercises

1. Write a decompressor algorithm for Move-to-Front coding.
2. A sentence is called a palindrome if the sequence of words in it is the same as the reverse of the sequence of words in it. For example, "bye the bye" is a palindromic sentence. Show that for a palindromic sentence, the order of the words in the final MTF list will be the same as the order of the words in the sentence.
3. Construct a sentence with some words repeated, such that the final MTF list reverses the sequence of words in the sentence.
4. ***** The naïve way of manipulating the list leads to an algorithm with quadratic time complexity. Design an an algorithm with $O(n \log n)$ complexity where n is the number of words in the sentence.

4 Defining the computational complexity of a decompressor

The definition of the time complexity of a decompressor is subtle. In the theory of algorithms, running time of an algorithm is defined in terms of the size of the input. However, this is a problem for a decompressor. If the compression is very high, then the compressed word is very short in terms of the expansion. This might mean that the decompression is deemed "exponential time" merely because the compression has been very successful. This would rule out very good compression schemes as having "slow decompression".

5 Issues in lossless compression

The issues we consider when we consider lossless compression can be the following.

First, does the compressor compress all "useful data"? How do we prove that it does, if this is the case? Note that this is the most difficult issue we will consider in this course, and a full treatment of this issue will use convergence concepts from real analysis.

Second, how fast are the compressor and the decompressor? A general rule-of-thumb for compression is that decompression should be at least as fast as the compression. This may be questionable, however - we can imagine scenarios where a server compresses data to be sent to several clients, and the clients each decompress the received data, where the decompression can be slow but the compression must be quick.

Third, how much memory do the comressor and the decompressor take? Are they finite memory schemes?

Fourth, the issue of universality - can we form compression algorithms which are optimal over a class of compressors? The study of universal compression schemes and their properties will form the second part of the course.