Data Compression

- Introduction
- Run-Length Encoding
- Huffman Encoding
- Lempel-Ziv Encoding

The goal of these notes is to introduce the ideas behind several different data compression algorithms, in order to give you an intuition how these algorithms work.

Not every detail necessary to implement these algorithms is explained here - but once you have the right intuitions, the details should be relatively straightforward.
Efficient Representation of Strings

We can represent any collection of data as a string.

Review:

• A string is a finite sequence of characters drawn from a finite alphabet.

  (In this context, we are not talking about C strings, but instead a more general notion.)

• The alphabet is the set of all possible characters.

• In practice, we will usually treat the data as a string of 8-bit bytes. The alphabet, therefore, is the set of values that can be represented as bytes, which can be conveniently mapped to the integers 0..255.

• In the examples, however, we will treat the data as a string of 3-bit bytes; the alphabet, therefore, is the set of values that can be mapped to 0..7. We will call these A through H.

We can represent any collection of bytes by a string in this alphabet, and therefore our compression techniques will always use this alphabet.

In some cases, there may be an advantage to using another alphabet, but we won’t talk about such application-specific alphabets.
Compression Strategies

We will explore three different strategies, based on three different observations:

• Sometimes data can be very repetitious, consisting of many repetitions of the same character (or possibly fixed-length groups of characters). In this case, the data has a very regular structure - and that structure can be used to describe the data, perhaps quite succinctly.

• Sometimes particular characters may be very unusual, so reserving bits for them is wasteful. As an extreme example, imagine that out of the 256 possible characters in our alphabet, only 8 actually appear in the file - we could represent all of those 8 characters with only 3 bits per character.

• Sometimes the data can contain many repetitions of the same text or similar texts. We can encode these repetitions by extending the alphabet to contain them.
Run-Length Encoding

- The philosophy behind run-length encoding is similar to that behind sparse matrices, but taken to an extreme. We will assume that every character we see is repeated several times in succession.

**Algorithm:**

- The output consists of pairs of characters: a count character, followed by the name of the character. The count is the number of times the character is repeated:

  AAAAABBBBBAAAC becomes 7A3B4A1C

This algorithm suffers if characters are not repeated:

  ABABABAB becomes 1A1B1A1B1A1B

When this happens, the result is twice as long as the original. To (somewhat) fix this, we can extend the algorithm to deal with sequences of non-repeated characters by using a special delimiter to denote them. For example, we can use the count of 1 (or 0) to indicate that a string of non-repeated characters follows; the sequence will also be terminated with a 1.

  AABABAAA becomes 2A1BAB13A

When a 1 appears in a sequence of non-repeated characters, we escape it!

We can still achieve inefficiency, if we need to encode a lot of 1’s...
Run-Length Encoding (ctd)

As described, this compression scheme is not very powerful - for example, strings like "ABABABABAB" are not compressed at all (and actually get larger), even though they clearly have a structure we could take advantage of, if our algorithm was more clever.

Properties of run-length encoding:

- Can be extended in 2 (or more) dimensions to represent areas (or volumes) filled with a particular character.

This can be very useful for printing graphics, FAX machines, etc. (Think of a mostly white page with a few black marks on it.)

File systems often use a very simple based on run-length encoding - disk blocks that would be filled completely with zeros are not even allocated.

- The algorithm is easy to implement (in hardware, if necessary) and runs very quickly.

- If part of the data is lost or corrupted, all or nearly all of the rest of the data can be reconstructed. Not true of many compression techniques!

- If the data is not suitable to run-length encoding (and it’s easy to construct data that isn’t), then this encoding can be larger than the original.
Huffman Codes

Huffman codes are based on the idea that characters that appear more frequently in the data can have shorter representations than characters that appear less frequently - and if the frequent characters appear frequently enough, then this can save space.

Interpreting ordinary fixed-length codes, bit by bit, can be thought of as traversing a decision tree, trying to find the leaf that corresponds to a particular character. Imagine that we have the following 3-bit codes:

A = 000, B = 001, C = 010, D = 011
E = 100, F = 101, G = 110, H = 111

Deciding which letter a sequence of bits represents can be done with a decision tree (where 0 represents the left child, and 1 the right):

```
   0
  / \
 /   \ 
A     B
|     |
C     D
|     |
E     F
|     |
G     H
```

Each character has a unique representation as a sequence of bits. The sequences are of different lengths - more frequently-appearing characters are given a shorter representation.
Variable Length Codes

Now, imagine that in the data we are encoding, the letters A and B are much more frequent than any other characters. Perhaps it would be worthwhile to give it a shorter representation, even though this would mean giving at least some of the other characters a longer representation. For example:

```
A = 00, B = 01, C = 1000, D = 1001,
E = 1010, F = 1011, G = 110, H = 111
```

Note that the different characters in this code are represented by different numbers of bits. This makes representing them somewhat more difficult (and requires explicit fiddling with bits, which is rarely elegant) and means that we have to read through the output bit by bit, instead of character by character.

Important note: these encodings are prefix codes: no code is the prefix of another. When we traverse the decision tree, when we get to a leaf we can always stop!
The Huffman Algorithm

So, how can we decide what decision tree to build in order to minimize the number of decisions required in order to decode a given piece of data?

The Huffman algorithm provides a way to find the optimal encoding length for each character.

**Algorithm:**

1. Count the number of times each character occurs in the data.

2. For each character in the alphabet, construct a one-node tree. The node contains the character and the count of the number of times that character occurs.

   Place all of these one-node trees into a priority queue (aka min-heap), where the key of the priority queue is the occurrence count.

3. Loop until the priority queue only contains one tree:
   - Dequeue the two trees with the smallest counts.
   - Create a new tree by joining these two trees. The count for the root of the new tree is the sum of the counts of its children.
   - Enqueue the new tree, according to its count.

When this loop terminates, the resulting tree will have all the characters as its leaves, and each at the depth that minimizes the total cost of encoding the file.
Performance of Huffman Encoding

Imagine that we have a file consisting entirely of many repetitions of one 8-bit character. The Huffman algorithm will construct a tree that has that one character at depth 1 (and every other character at some deeper levels). Therefore, instead of requiring 8 bits to encode each character, now we will only need 1.

This is a nice improvement, but only an 8-to-1 compression ratio. (In this same case, the RLE method would do much better.)

However, as long as we insist that each character in the data be represented by one token in the output, we cannot achieve a better compression ratio - the most we can hope for is to shrink the input by a factor equal to the number of bits per character.

A sketch of the proof of the optimality of Huffman codes is given in Weiss; Lewis and Denenberg or Cormen, Leiserson and Rivest (and others) give more complete proofs.

To do better, we must allow each token in the output to represent an arbitrary number of characters in the input.
The Lempel-Ziv method allows sequences of characters to be represented by a single token.

But now we are faced with a difficult problem - which sequences of characters should we choose to encode with a special token?

The Lempel-Ziv algorithm provides a way of choosing that is not necessarily optimal, but usually works very well. The Lempel-Ziv algorithm greedily assumes that everything it has seen before, it will see again, and so it adds to its table every new sequence it sees.
The Lempel-Ziv Algorithm

1. Create a trie. Add to the trie all the single-character strings consisting of one character from the alphabet. The encodings for each of these strings is simply the code of that character.

2. Let P = \textbf{nil}, and N = the first character in the input.

3. Loop, until the input data is exhausted:

   Traverse the trie by reading characters from the input and following the corresponding edge in the trie. When a leaf of the trie is reached, perform the following steps:
   
   • Add the code for the current leaf to the output.
   • Let C be the path from the root to the current leaf.
   • If P is not \textbf{nil}, then add a leaf for character N to the trie, at the end of P. (If such a leaf already existed, then we would not have stopped after traversing P - we would have continued on to it.)
     
   • Let P = C.
   • Let N be the next character in the input.

\textbf{Note:} since the trie starts out with $2^8$ leaves (for 8-bit characters) and immediately starts to grow, the coding for each leaf will (eventually) require more than 8 bits.

There are some very clever and elegant ways to deal with this, but we will not explore them here.
An Interesting Property of Lempel-Ziv

When it is time to decode the output produced by the Lempel-Ziv algorithm, how do we know what the table of character sequences (in our case, the trie) looked like at each point in time? (We can’t just store the trie along with the data, because the trie keeps changing!)

It turns out that we don’t need to store the trie, or anything else about the table of sequences. The decoder can reconstruct this table explicitly as it performs the decoding, in the same manner as the encoder does!

This works because we add the code for a new pattern to the table only after that pattern has appeared in the output. Therefore, after reading in each token from the compressed data, the decoder can know exactly what the encoder would have added to the table next.

This is a very nice property of the Lempel-Ziv codes.