DEFLATE Dissected

If you find the deflate “specification” confusing, maybe even frustrating, then hopefully this page *will* help you. It probably won't at the moment, because I myself am trying to figure this thing out. Even with several tutorials and example source code I'm having difficulty understanding particulars of this algorithm.

The most complicated part of decoding (inflating) deflated data is building the Huffman tables for literals/lengths and distances. All of the docs I've read are light on steps and examples while heavy on assumption. This is where I'm currently stuck.

Glossary ...

canonical Huffman code: As in coming from a standard Huffman tree which follows certain rules so that only the lengths of the codes are necessary to generate the entire tree.

Misc.

Code 284 has 5 extra bits, but only supports lengths 227 – 257 instead of 227 – 258 (31 possibilities). In addition, code 285 has *no* extra bits and represents length 258 directly. Why is this? [RFC 1951, pg. 10]

How can the literal/length tree be generated from lengths alone?

Building Huffman Tables

This document will attempt to explain how a Huffman table is built, specifically to help figure out the deflate algorithm. First, each symbol is given a frequency or *weight*. The weightier the symbol, the smaller the resulting code. Now when I talk about symbols, I'm talking about the elements of the alphabet you are working with. A code is the bits used to walk the Huffman tree to find the symbol. You know this already, though, don't you? You're here, because like me you're not understanding how the tree itself is generated.

For the purposes of simplifying my examples, I'm going to use a very small alphabet: ABCD. The alphabet is the set of all the symbols you'll possibly have. Let's give each of these symbols a frequency:

5 A
2 B
7 C
9 D

Nodes in a Huffman tree only have two possibilities: 0 or 1. This is why each bit in the stream you are reading will represent some footwork down that tree, either left or right (0 or 1). The nodes are constructed bottom up and a leaf is a resulting symbol.

First, you'll always want to sort your codes by the weights you've assigned:

2 B
5 A
7 C
9 D

Next, take the two items with the *lowest* weights and combine them into a node. For our current example, this will be A and B. Canonical Huffman codes imply that there is a standard way of determining how nodes are arranged and indeed, most canonical implementations prefer the lesser weight on the left (bit 0). If a leaf and a node have the same weight, then the leaf appears on the left. So ...

```
      _7:*_
     /     \
   2:B    5:A
```

The resulting node has a weight that is the sum of both children, 2 + 5 = 7 in this case. This node gets put in the same list as all the leaves (e.g. symbols):

7 [AB]
7 C
9 D

Now you can combine the leaves and the nodes in the same way until all you have is one entry left in your list and thus the root of your tree. The root is never given a bit, for obvious reasons. You know where the tree starts. What you end up with for this example is:

```
  _23:*_
 /      \
9:D  _14:*_
    /      \
   7:C   _7:*_
        /     \
       2:B   5:A
```

The complicated part, for me, is now using the idea of canonical Huffman codes to generate a tree based on code *lengths*. A code's length is how many bits are required to represent it. For the above example, only 3 bits are necessary. If you take a left branch for every zero (0) and a right branch for every one (1) then the symbol 'A' is represented by the code 111, 'D' by simply 0, etc.

Coming back to code lengths. The code for 'A' is 111 so it has a code length of 3. Let's look at the code lengths for each of these:

5 A 3 (111)
2 B 3 (110)
7 C 2 (10)
9 D 1 (0)

This alphabet with its weights cannot be made into a canonical form. With canonical codes, the symbols must be ordered so that those represented by a single code and be found again. Here's what I mean, say we know the alphabet has code lengths 3, 3, 2, and 1. This would almost make sense except B comes before A, so assuming the alphabet is descending in code length we'd come up with the following table:

```
  _*_
 /   \
D   _*_
   /   \
  C   _*_
     /   \
    A     B
```

A 3 (110)
B 3 (111)
C 2 (10)
D 1 (0)

That is, B and A are reversed. Notice, however, we were able to come up with *almost* the same table. Of course, now you're wondering *how* I would know to do it this way. The trick to creating a weight from a code length in this example is to make the weight *larger* the smaller the length *and* for equal weights to be incremental.

Let's do this. First, we know what our alphabet is and the code lengths for each symbol:

A 3
B 3
C 2
D 1

Next, fill up some bits (set to all ones) shift to the right based on the code length:

A 3 (1111b >> 3 = 1)
B 3 (1111b >> 3 = 1)
C 2 (1111b >> 2 = 3)
D 1 (1111b >> 1 = 7)

Ah ha! Now, let's rebuild the table based on these weights:

1 A
1 B
3 C
7 D

First we combine A & B, since they're the lowest (notice we could've taken the frequency generation one step further to give B a '2' weight to make it simpler to sort).

```
     _2:*_
    /     \
   1:A   1:B
```

Continuing on with the combining lower-weights we will end up with a familiar table ...

```
      _14:*_
     /      \
    9:D    _5:*_
          /     \
         3:C   _2:*_
              /     \
             1:A   1:B
```

My canonical method presented here is fairly cheesy and produces ugly tables. See how the 5:* is on the right? Actually this whole thing is fucked up, but maybe it's starting to help me make sense of canonical Huffman in deflate.