Huffman Coding

The idea behind Huffman coding is to find a way to compress the storage of data using variable length codes. Our standard model of storing data uses fixed length codes. For example, each character in a text file is stored using 8 bits. There are certain advantages to this system. When reading a file, we know to ALWAYS read 8 bits at a time to read a single character. But as you might imagine, this coding scheme is inefficient. The reason for this is that some characters are more frequently used than other characters. Let's say that the character 'e' is used 10 times more frequently than the character 'q'. It would then be advantageous for us to use a 7 bit code for e and a 9 bit code for q instead because that could shorten our overall message length.

Huffman coding finds the optimal way to take advantage of varying character frequencies in a particular file. On average, using Huffman coding on standard files can shrink them anywhere from 10% to 30% depending to the character distribution. (The more skewed the distribution, the better Huffman coding will do.)

The idea behind the coding is to give less frequent characters and groups of characters longer codes. Also, the coding is constructed in such a way that no two constructed codes are prefixes of each other. This property about the code is crucial with respect to easily deciphering the code.

Building a Huffman Tree

The easiest way to see how this algorithm works is to work through an example. Let's assume that after scanning a file we find the following character frequencies:

<u>Character</u>	<u>Frequency</u>
'a'	12
'b'	2
'c'	7
'd'	13
'e'	14
'f'	85

Now, create a binary tree for each character that also stores the frequency with which it occurs.

The algorithm is as follows: Find the two binary trees in the list that store minimum frequencies at their nodes. Connect these two nodes at a newly created common node that will store NO character but will store the sum of the frequencies of all the nodes connected below it. So our picture looks like follows:

```
     9            12 'a'       13 'd'       14 'e'       85 'f'
   /    \
 2 'b'  7 'c'
```

Now, repeat this process until only one tree is left:

```
        21
       /  \
     9    12 'a'  13 'd'      14 'e'      85 'f'
    /  \
  2 'b'  7 'c'


        21                27
       /  \              /    \
     9    12 'a'  13 'd'   14 'e'        85 'f'
    /  \
  2 'b'  7 'c'

                 48
              /          \
        21                27
       /  \              /    \
     9    12 'a'  13 'd'   14 'e'        85 'f'
    /  \
  2 'b'  7 'c'

                          133
                         /     \
                 48             85 'f'
              /          \
        21                27
       /  \              /    \
     9    12 'a'  13 'd'   14 'e'
    /  \
  2 'b'  7 'c'
```

Once the tree is built, each leaf node corresponds to a letter with a code. To determine the code for a particular node, walk

a standard search path from the root to the leaf node in question. For each step to the left, append a 0 to the code and for each step right append a 1. Thus for the tree above we get the following codes:

Letter	**Code**
'a'	**001**
'b'	**0000**
'c'	**0001**
'd'	**010**
'e'	**011**
'f'	**1**

Why are we guaranteed that one code is NOT the prefix of another?

Find a set of valid Huffman codes for a file with the given character frequencies:

Character	**Frequency**
'a'	**15**
'b'	**7**
'c'	**5**
'd'	**23**
'e'	**17**
'f'	**19**

Calculating Bits Saved

All we need to do for this calculation is figure out how many bits are originally used to store the data and subtract from that how many bits are used to store the data using the Huffman code.

In the first example given, since we have six characters, let's assume each is stored with a three bit code. Since there are 133 such characters, the total number of bits used is 3*133 = 399.

Now, using the Huffman coding frequencies we can calculate the new total number of bits used:

Letter	**Code**	**Frequency**	**Total Bits**
'a'	001	12	36
'b'	0000	2	8
'c'	0001	7	28
'd'	010	13	39
'e'	011	14	42
'f'	1	85	85
Total			238

Thus, we saved 399 - 238 = 161 bits, or nearly 40% storage space. Of course there is a small detail we haven't taken into account here. What is that?

Huffman Coding is an Optimal Prefix Code

Of all prefix codes for a file, Huffman coding produces an optimal one. In all of our examples from class on Monday, we found that Huffman coding saved us a fair percentage of storage space. But, we can show that no other prefix code can do better than Huffman coding.

First, we will show the following:

Let x and y be two characters with the least frequencies in a file. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.

Here is how we will prove this:

Assume that a tree T stores an optimal prefix code. Let and characters a and b be sibling nodes stored at the maximum depth of the tree. We will show that we can create T' with x and y as siblings at the lowest depth of the tree such that the number of bits used for the coding with T' is the same as with T. (Let f(a) denote the frequency of the character a. Without loss of generality, assume f(x) ≤ f(y) and f(a) ≤ f(b). It also follows that f(x) ≤ f(a) and f(y) ≤ f(b). Let h be the height of the tree T. Let x have a depth of d_x in T and y have a depth of d_x in T.)

Create T' as follows: swap the nodes storing a and x, and then swap the nodes storing b and y. Now, we have that the depth of x and y in T' is h, the depth of a is d_x and the depth of b is d_y in T'.

Now, let's calculate the change in the number of bits used for the coding with tree T' with the coding in tree T. (Note: Since all other codes remain unchanged, we only need to analyze the total number of bits it takes to code a, b, x and y.)

# bits for tree T (for a,b,x and y) = $hf(a) + hf(b) + d_x f(x)\ \ d_y f(y)$

# bits for tree T' (for a, b, x, and y) = $d_x f(a) + d_y f(b) + hf(x) + hf(y)$.

Difference =

$hf(a) + hf(b) + d_x f(x)\ \ d_y f(y) - (d_x f(a) + d_y f(b) + hf(x) + hf(y)) =$

$hf(a) + hf(b) + d_x f(x)\ \ d_y f(y) - d_x f(a) - d_y f{*}b) - hf(x) - hf(y) =$

$h(f(a) - f(x)) + h(f(b)-f(y)) + d_x(f(x) - f(a)) + d_y(f(y) - f(b)) =$

$h(f(a) - f(x)) + h(f(b)-f(y)) - d_x(f(a) - f(x)) - d_y(f(b) - f(y)) =$

$(h - d_x)(f(a) - f(x)) + (h - d_y)(f(b) - f(y))$

Notice that all four of the terms above must be non-negative since we know that $h \geq d_x$, $h \geq d_y$, $f(a) \geq f(x)$, and $f(b) \geq f(y)$. Thus, it follows that this difference must be 0. Thus, the number of bits to used in a code where x and y (the two characters with lowest frequency) are siblings at maximum depth of the coding tree is optimal.

In layman's terms, give me what you think is an optimal coding tree, and I can create a new one from it with the two nodes corresponding to low frequencies at the bottom of the tree.

To complete the proof, you'll notice that by construction, Huffman coding ALWAYS makes sure that the nodes with the lowest frequencies are at the bottom of the coding tree, all the way through the construction. (You can't find any pair of nodes for which this isn't true.) Technically, to carry out the proof, you'd use induction, but we'll skip that for now...