

CS 199: Homework 1

Due by 10:00 a.m. on Monday, Feb 5

For these two problems,

1. A flag is worth just how many words? In this problem, you are asked to devise a scheme for describing flags, so that they do not need to be stored bit-by-bit. Take a look at

http://en.wikipedia.org/wiki/Gallery_of_sovereign-state_flags

to see what types of pictures we might be talking about.

Let's assume that all flags are 1000 pixels wide by 600 pixels high. Assume that we use "true-color", which means a *color depth* of 24-bits: (one byte, or eight bits, for each of R, G, B values at a pixel).

How many *bytes* does a .bmp (bitmap) representation take? (No compression, just storing the colors for every single pixel.)

As an example of a possible loss-less compression mechanism for encoding flags, let's make a simplifying assumption - suppose that all flags were made of up equal-width stripes, and that the only differences between flags was whether the stripes ran vertically or horizontally, the number of stripes, and the color of each stripe. Then, the flag of Indonesia could be indicated by $(0, 10, x, y)$, where 0 indicates horizontal stripes, 10 is binary for 2, indicating there are two stripes, and x and y are their RGB color values, 24 bits each. Thus, this representation of the Indonesian flag takes up only 51 bits, or a little over 6 bytes. Similarly, the flag of any of the many countries with three vertical stripes would have encoding $(1, 11, x, y, z)$ where x, y , and z are again 3 bytes each (8 bits for R, G, and B), and we have used only a tad more than 9 bytes.

The more astute observer will recognize that we don't need to list the number of stripes for this encoding, because the number of other bits tells us how many stripes there are.

However, as you can see from the above website, flags can be much more complicated. Using any ideas you'd like, come up with a scheme for representing flags, and an explanation for how to read the representation and construct the flag that is represented.

Details:

Notice first that you can always fit any flag in to your scheme, by saying "if the flag doesn't fit into the above scheme, then simply represent it as a bitmap". Notice also that some flags do have complicated graphics that may not be easily described by simple geometric regions. For these, it is a perfectly acceptable solution to simply indicate that there is a graphic element, where it is located, how big it is, and then give a bitmap description of the graphic. For example, without the crest in the middle, the flag of Bolivia might have a simple representation $(0, x, y, z)$ for suitable color choices x, y , and z . But the crest, which we estimate to be 100 by 100 pixels, will have to be represented as a bitmap separately. Our final representation for Bolivia might be:

$(0, x, y, z) (a, b, 100, 100, w)$

where $(0, x, y, z)$ are instructions to draw the three stripes, (a, b) are the coordinates of the lower left corner of the crest, the 100's indicate that the crest is 100 by 100 pixels, and w is a listing of all colors in the 100 x 100 graphic (30,000 bytes). (We might further compress w with run-length coding.)

Completely describe your scheme, and give an example of how it works for five completely different flags from the above page. For each, give a rough idea of the number of bytes in the flag's representation.

Prepare a short presentation in powerpoint or some other presentation software, and be prepared to explain your scheme, to justify your decisions, and to answer questions about it.

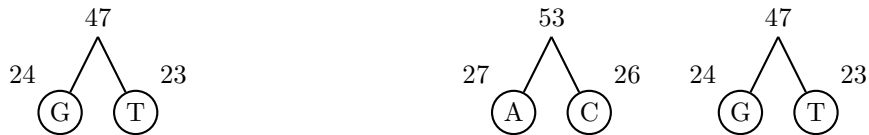
Hint: in the Bolivia example above, notice that the representation became easier when we thought of first drawing the stripes, then drawing the graphic on top of the stripes. This idea of overlaying one image on another can be used to our advantage. If we had instead drawn the graphic first, then the description of how to draw the containing stripe would have been enormously messy. For some flags, It may help to think in terms of layers, and the order that you might draw things.

2. **Compressing the Human Genome:** One of the questions on Web Quiz 2 discussed representing the human genome as a long string of the bases A,C,G,T. We realized that it was inefficient to use the ASCII encoding, and we could do better (that is, use fewer bits, saving space) using a smaller fixed-length encoding. In this question, we will consider using the Huffman encoding scheme to compress the genome.

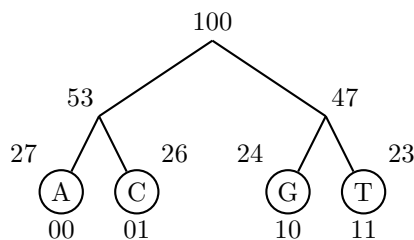
Recall that the Huffman algorithm to construct the tree was to find the two least frequent characters (suppose they occurred n_1 and n_2 times respectively), draw edges from each of them to a new *parent* symbol, and now treat the parent as a new symbol that occurred $n_1 + n_2$ times. We repeated this process until all the symbols were combined together into a single tree. Suppose there are k symbols we have to represent, let n_i be the number of times that the i th symbol occurs in the text, and let l_i be the number of bits used to represent the i th symbol. Then, the total number of bits used to represent all the copies of the i th symbol is $n_i \times l_i$, and so the total number of bits used to represent the entire text will be $(n_1 \times l_1) + (n_2 \times l_2) + \dots + (n_k \times l_k)$.

To find the optimal tree, instead of using the number of times n_i that the i th symbol occurs (called the *frequency* of the symbol), we could use its *relative frequency* f_i , which is n_i/n , where n is the total number of symbols in the text. (In our problem, n is the total number of bases in the genome, and f_1 is the fraction of the bases that are of type '1'.) The two symbols with the smallest frequency will be the two symbols with the smallest relative frequency (since we just divide everything by n), and it is easy to see that running the algorithm with the relative frequencies would give us the same tree as if we were to use the frequencies. The *average* number of bits used to represent a single text symbol will be $(f_1 \times l_1) + (f_2 \times l_2) + (f_3 \times l_3) + \dots + (f_k \times l_k)$. The total number of bits used will be n times the average number of bits per symbol. (You should check that the total number of bits using this formula is the same as the answer from the slightly different formula we used in the previous paragraph.)

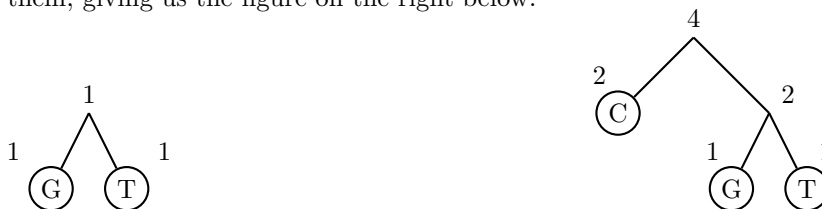
Suppose the relative frequencies of the bases are 27% A, 26% C, 24% G, and 23% T. To construct the Huffman code, we would note that G and T have the lowest frequencies, so we would connect them together to form a 'new symbol' with relative frequency 47%. (See the left side of the figure below.)



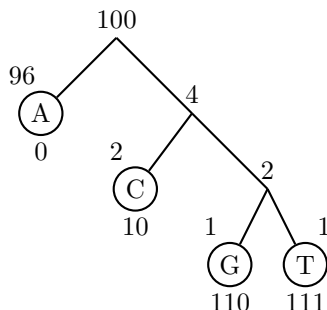
Next, A and C have the two smallest frequencies (from among those remaining), so we connect them, and form a new symbol with relative frequency 53%. (See the right side of the figure above). Now that we only have two 'symbols' left, we connect them to form a tree, as below. Now that we have a complete tree, we can assign labels to the nodes, so that A is 00, C is 01, and so on.



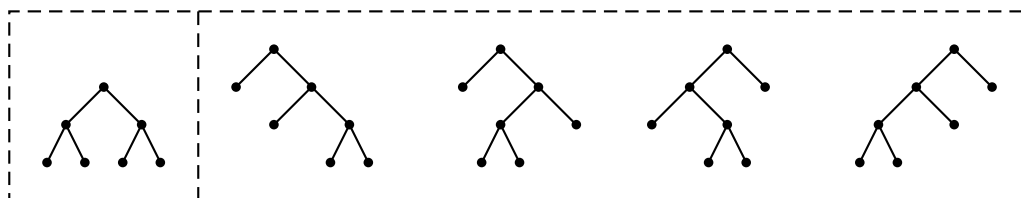
Now suppose the relative frequencies are 96% A, 2% C, 1% G and 1% T. If we were to construct the Huffman tree, we would start by combining the symbols G and T, since they have the lowest relative frequencies. This would give us the figure on the left below, with the new combined ‘symbol’ having relative frequency $1+1 = 2$. Now, the two lowest remaining frequencies are for the new symbol and C, so we connect them, giving us the figure on the right below.



Now, we have only two ‘symbols’ left; we connect them to form the tree below. We assign labels to the nodes, with 0 indicating that we move left, and 1 that we move right. So A gets the label 0, C gets the label 10, and so on.



- (a) First, notice that there are only two really different Huffman-type trees that have 4 ‘leaves’ (that is, that have 4 symbols to represent). The first tree looks like the one we got when all the symbols have nearly the same frequency (in this case, the tree looks perfectly balanced, and all the symbols have two-bit representations). The second tree looks like the one we got when one symbol is much more common than all the others (in this case, the tree looks unbalanced). Actually, there are three other trees that are essentially identical to the second type of tree; see the figure below.



Each of the last 4 trees corresponds to one symbol having a 1-bit encoding, another symbol having a 2-bit encoding, and the last two symbols each having a 3-bit encoding. (This is why we say that they are basically the same, at least as far as Huffman codes go.)

- (b) With the first example we considered, Huffman's procedure gave us a tree of the first type, which means that for these frequency values, the optimal encoding is the same as a fixed-length encoding where every symbol is represented using 2 bits. With the second example, we got a tree of the second type, which means that for these frequency values, the optimal encoding is *variable-length* (that is, not all symbols are represented using the same number of bits.) We want to determine for which frequency values variable-length encodings are better.
- (c) Suppose 30% of the bases are of type C, and 20 % of type G. (Now, if we also tell you the percentage of type T, you can determine the percentage of type A.) For each of the following relative frequencies of T, is it better to use a fixed-length or a variable-length encoding?
- 5%
 - 10%
 - 15%
 - 20%

Now suppose 35% of the bases are of type C, and 15% of type G. For each of the following relative frequencies of T, is it better to use a fixed-length or a variable-length encoding?

- 10%
 - 12.5%
 - 15%
 - 17.5%
 - 20%
- (d) Without actually running the Huffman algorithm, is there a simple way to look at the relative frequencies and decide when it is better to use a variable-length code? What is it?
(*Hint: Look at part (a), and the explanation of average number of bits per symbol.*)