Intro to Compression and Huffman Coding

Huffman Encoding and Data Compression. In the early 1980s, personal computers had hard disks that were no larger than 10MB; today, even the puniest of disks are measured in hundreds of gigabytes. Even though hard drives are getting bigger, the files we want to store (hilarious pictures of cats, (increasingly high-definition) videos, music and so on) seem to keep pace with that growth which makes even today's gargantuan disks seem too small to hold everything.

One technique to use our storage more optimally is to compress the files. By taking advantage of redundancy or patterns, we may be able to "abbreviate" the contents in such a way to take up less space yet maintain the ability to reconstruct a full version of the original when needed. Such compression could be useful when trying to cram more things on a disk or to shorten the time needed to copy/send a file over a network.

There are compression algorithms that you may already have heard of. Some compression formats, such as JPEG, MPEG, or MP3, are specifically designed to handle a particular type of data file. They tend to take advantage of known features of that type of data (such as the propensity for pixels in an image to be same or similar in color to their neighbors) to compress it. Other tools such as **compress**, **zip**, or **pack** can be used to compress any sort of file. These algorithms have no *a priori* expectations and usually rely on studying the particular data file's contents to find redundancy and patterns that allow for compression.

Some of the compression algorithms (e.g. JPEG, MPEG) are ***lossy*** - decompressing the compressed result doesn't recreate a perfect copy of the original. Such an algorithm compresses by "summarizing" the data. The summary retains the general structure while discarding the more minute details. For sound, video, and images, this imprecision may be acceptable because the bulk of the data is maintained and a few missed pixels or milliseconds of video is no big deal. For text data, though, a lossy algorithm usually isn't appropriate. An example of a lossy algorithm for compressing text would be to remove all the vowels. Compressing the previous sentence by this scheme results in:

```
        n xmpl f lssy lgrthm fr cmprssng txt wld b t rmv ll th vwls.
```

This shrinks the original 87 characters down to just 61 and requires only 70% of the original space. To decompress, we could try matching the consonant patterns to English words with vowels inserted, but we cannot reliably reconstruct the original in this manner. Is the compressed word "fr" an abbreviation for the word "four" or the word "fir" or "far"? An intelligent reader can usually figure it out by context, but, alas, a brainless computer can't be sure and would not be able to faithfully reproduce the original. For files containing text, we usually want a *lossless* scheme so that there is no ambiguity when recreating the original meaning and intent.

An Overview. The standard ASCII character encoding uses the same amount of space (one *byte* or eight *bits*, where each bit is either a 0 or a 1) to store each character. Common characters like `'e'` don't get any special treatment; they require the same 8 bits that are used for much rarer characters such as `'z'` or `'\'`. A file of 1000 characters encoded using the ASCII scheme will take 8000 bits; no more, no less, whether it be a file of 1000 spaces or a file containing 4 instances of 250 different characters. A *fixed-length* encoding like ASCII is convenient because the boundaries between characters are easily determined and the pattern used for each character is completely fixed (i.e. `'a'` is always 97 [the decimal equivalent of the 8 bit binary string]).

In practice, it is not the case that all 256 characters in the ASCII set occur with equal frequency. In an English text document, it might be the case that only 90 or so distinct characters are used at all (meaning 166 characters in the ASCII table never even appear) and within those 90 there are likely to be significant differences in the character counts.

The Huffman encoding scheme takes advantage of the disparity between frequencies and uses less storage for the frequently occurring characters at the expense of having to use more storage for each of the more rare characters. Huffman is an example of a *variable-length* encoding - some characters may only require 2 or 3 bits and other characters may require 7, 10, or 12 bits. The savings from not having to use a full 8 bits for the most common characters makes up for having to use more than 8 bits for the rare characters and the overall effect is that the file almost always requires less space.

ASCII Encoding. The example we're going to use throughout this handout is encoding the particular string `"happy hip hop"`. Using the standard ASCII encoding, this 13-character string requires 13*8 = 104 bits total. The table below shows the relevant subset of the standard ASCII table.

char	ASCII	bit pattern (binary)
h	104	01101000
a	97	01100001
p	112	01110000
y	121	01111001
i	105	01101001
o	111	01101111
space	32	00100000

`"happy hip hop"` would be encoded in ASCII as `104 97 112 112 121 32 104 105 112 32 104 111 112`. Although not easily readable by humans, it would be written as the following stream of bits (each byte is boxed to show the boundaries):

01101000	01100001	01110000	01110000	01111001	00100000	01101000
01101001	01110000	00100000	01101000	01101111	01110000	

To decode such a string (i.e. translate the binary encoding back to the original characters), we merely need to break the encoded stream of bits up into 8-bit bytes, and then convert each byte using the fixed ASCII encoding. The first 8 bits are `01101000`, which is the pattern for number 104, and position 104 in the ASCII table is assigned to lowercase `'h'`. A file encoded in ASCII does not require any additional information to be decoded since the mapping from binary to characters is the same for all files and computers.

A more compact encoding. The first thing you might notice about ASCII encoding is that using 8 bits per character can be excessively generous. Although it allows for the possibility of representing 256 different characters, we only have seven distinct characters in the phrase we're trying to encode, and thus could distinguish among these patterns with fewer bits.

We could set up a special coding table just for this phrase using 3 bits for each character. Creating such an encoding is trivial: we create a list of the unique characters, and then go through and assign each a distinct encoded number from 0 to `N`-1. For example, here is one possible 3-bit encoding (of the 7! possible permutations):

char	number	bit pattern
h	0	000
a	1	001
p	2	010
y	3	011
i	4	100
o	5	101
space	6	110

Using this table, `"happy hip hop"` is encoded as `0 1 2 2 3 6 0 4 2 6 0 5 2`. As computers store integers in binary, this would result in:

000	001	010	010	011	110	000	100	010	110	000	101	010

Using three bits per character, the encoded string requires 39 bits instead of the original 104 bits, compressing to 38% of its original size.

However, to decode this binary representation, one would need to know the special mapping used, since using `000` for `'h'` is not standard practice and in fact, in this scheme, each compressed string uses its own special-purpose mapping that is not necessarily like any other. Some sort of header or auxiliary file would have to be attached or included with the encoded representation that provides the mapping information. That header would take up some additional space that would cut into our compression savings. For a large enough file, though, the savings from trimming down the per-character cost would likely outweigh the expense of the additional table storage.

***A* variable-length *encoding*.** What if we drop the requirement that all characters take up the same number of bits? By using fewer bits to encode characters like `'p'`, `'h'`, and space that occur frequently and more to encode characters like `'y'` and `'o'` that occur less frequently, we may be able to compress even further. We'll later show how we generated the table below, but for now just take our word for it that is represents an optimal Huffman encoding for the string `"happy hip hop"`:

char	bit pattern
h	01
a	000
p	10
y	1111
i	001
o	1110
space	110

Each character has a unique bit pattern encoding, but not all characters use the same number of bits. The string `"happy hip hop"` encoded using the above variable-length code table is:

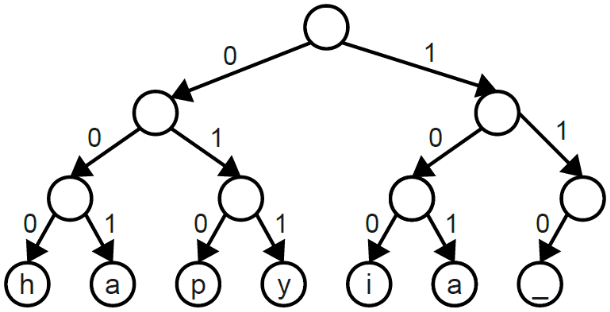
01	000	10	10	1111	110	01	001	10	110	01	1110	10

The encoded phrase requires a total of 34 bits, shaving a few more bits from the fixed-length version. What is tricky about a variable-length code is that we no longer can easily determine the boundaries between characters in the encoded stream of bits when decoding. Every character's bit pattern is boxed above to help you visualize the encoding, but without this aid, you might wonder how you will know whether the first character is encoded with the two bits `01` or the three bits `010` or perhaps just the first bit `0`? If you look at the encoding in the table above, you will see that only one of these options is possible. There is no character that encodes to the single bit `0` and no character that encodes to the sequence `010` or `0100`.

There is, however, a character that encodes to `01` and that is `'h'`. One of the important features of the table produced by Huffman coding is the *prefix property:* no character's encoding is a prefix of any other (i.e. if `'h'` is encoded with `01` then no other character's encoding will start with `01` and no character is encoded to just `0`). With this guarantee, there is no ambiguity in determining where the character boundaries are. We start reading from the beginning, gathering bits in a sequence until we find a match. That indicates the end of a character and we move on to decoding the next character.

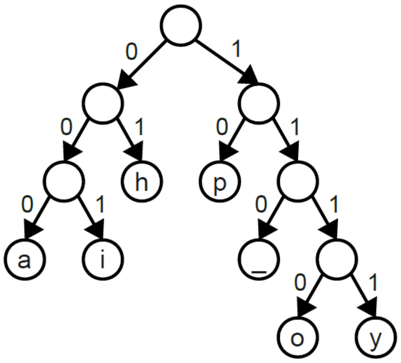
Like the special-purpose fixed-length encoding, a Huffman encoded file will need to provide the information about the table used so we will be able to decode the file. Each file's table will be unique since it is explicitly constructed to be optimal for that file's contents.

Encoding seen as a tree. One way to visualize any particular encoding is to diagram it as a binary tree. Each character is stored at a leaf node. Any particular character encoding is obtained by tracing the path from the root to its node. Each left-going edge represents a 0, each right-going edge a 1. For example, this tree diagrams the compact fixed-length encoding we developed previously:



In the above tree, the encoding for `'y'` can be determined by tracing the path from the root to the `'y'` node. Going left then right then right again represents a `011` encoding.

A similar, much larger tree could be constructed for the entire ASCII set, it would be 8 levels deep and at the bottom would be 256 leaf nodes, one for each character. The node for the character `'a'` (97 or `01100001` in binary) would be at the end of the left-right-right-left-left-left-left-right path from the root. We're starting to see why they're called *binary* trees! Let's diagram a tree for the variable-length Huffman encoding scheme:



The path to 'h' is just left right or 01, the path to 'y' is right-right-right-right or 1111. Notice that the prefix property of the Huffman encoding is visually represented by the fact that characters only occupy leaf nodes (i.e. those nodes which are not a prefix of any further nodes).

The tree shown above for "happy hip hop" is, in fact, an optimal tree - there are no other tree encodings by character that use fewer than 34 bits to encode this string. There are other trees that use exactly 34 bits; for example you can simply swap any sibling nodes in the above tree and get a different but equally optimal encoding.

The Huffman tree doesn't appear as *balanced* as the fixed-length encoding tree. At this point you should know that, in general, an unbalanced tree is bad thing (badly unbalanced trees lose the advantages of trees). However, when a tree represents a character encoding, that lopsidedness is actually a good thing. The shorter paths represent those frequently occurring characters that are being encoded with fewer bits and the longer paths are used for more rare characters. Our plan is to shrink the total number of bits required by shortening the encoding for some characters at the expense of lengthening others. If all characters occurred with equal frequency, we would have a balanced tree where all paths were roughly equal. In such a situation we can't achieve much compression since there are no real repetitions or patterns to be exploited.

Decoding using the tree. A particularly compelling reason to diagram an encoding as a tree is the ease with which it supports decoding. Let's use the fixed-length tree to decode the stream of bits 011101010010011. Start at the beginning of the bits and at the root of the tree. The first bit is 0, so trace one step to the left, the next bit is 1, so follow right from there, the following bit is 1, so take another right. We have now landed at a leaf, which indicates that we have just completed reading the bit pattern for a single character. Looking at the leaf's label, we learn we just read a 'y'.??? Now we pick up where we left off in the bits and start tracing again from the root. Tracing 101 leads us to 'i'. Continuing through the remaining bits and we decode the string "yippy".

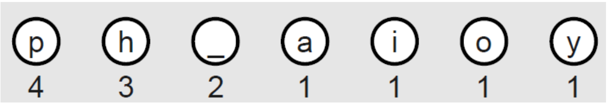
The same path-following strategy works equally well on the Huffman tree. Decoding the stream of bits 111100110101111 will first trace four steps down to the left to hit the 'y' leaf, then a left-left-right path to the 'i' leaf and so on. Again the decoded string is "yippy". Even though the encoded characters don't start and end at evenly spaced boundaries in the Huffman-encoded bits, we have no trouble determining where each character ends because we can easily detect when the path hits a leaf node in the encoding tree.

Generating an optimal tree. The pertinent question now: how is that special tree constructed? We need an algorithm for constructing the optimal tree giving a minimal per-character encoding for a particular file. The algorithm we will use here was invented by then MIT student David Huffman in 1952 and is still used today by most modern compression algorithms: Unix's pack, in conjunction with other compression algorithms in the PNG, JPEG, MPEG, and MP3 codecs, by Window's ZIP file format, among many others.

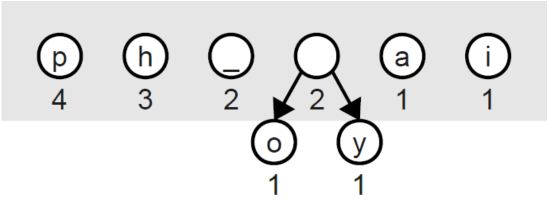
To begin generating the Huffman tree, each character gets a weight equal to the number of times it occurs in the file. In the "happy hip hop" example, the character 'p' has weight 4,'h' has weight 3, the space has weight 2, and the other characters have weight 1. Our first task is to calculate these weights, which we can do with a simple pass through the file to get the frequency counts. For each character, we create an unattached tree node containing the character value and its corresponding weight. You can think of each node as a tree with just one entry. The idea is to combine all these separate trees into an optimal tree by wiring them together from the bottom upwards. The general approach is as follows:

1. Create a priority queue of singleton trees (a "forest"), one for each character present in the input source, with weight equal to the character frequency.
2. From the collection, pluck out the two trees with the smallest weights and remove them. Combine them into a new tree whose root has a weight equal to the sum of the weights of the two trees and with the two trees as its left and right subtrees.
3. Add the new combined tree back into the collection.
4. Repeat steps 2 and 3 until there is only one tree left.
5. The remaining node is the root of the optimal encoding tree.

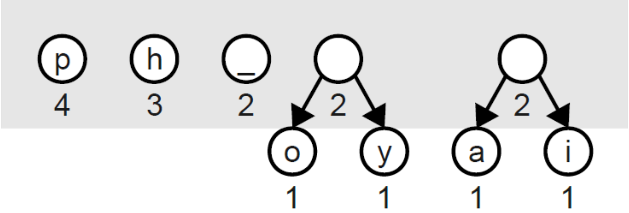
Sounds simple, doesn't it? Let's walk through building the optimal tree for our example string `"happy hip hop"`. We start with this collection of singletons, with the weight of each node labeled underneath:



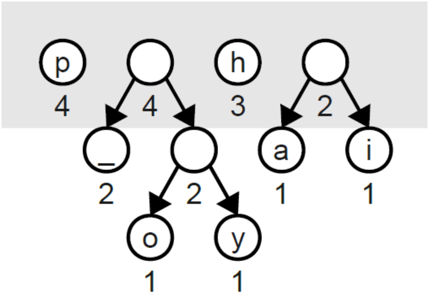
We start by choosing the two smallest nodes. There are four nodes with the minimal weight of one; it doesn't matter which two we pick (let the priority queue handle this decision). We choose `'o'` and `'y'` and combine them into a new tree whose root is the sum of the weights chosen. We replace those two nodes with the combined tree. The nodes remaining in the collection are shown in the light gray box at each stage:



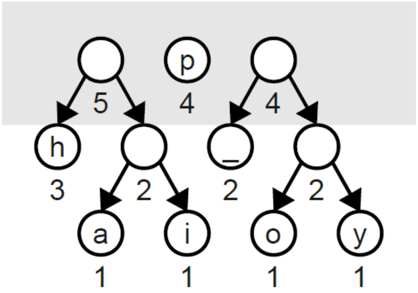
Now we repeat that step; this time there is no choice for the minimal nodes, it must be `'a'` and `'i'`. We take those out and combine them into a weight 2 tree. Note how the collection of nodes shrinks by one each iteration (we remove two nodes and add a new one back in).



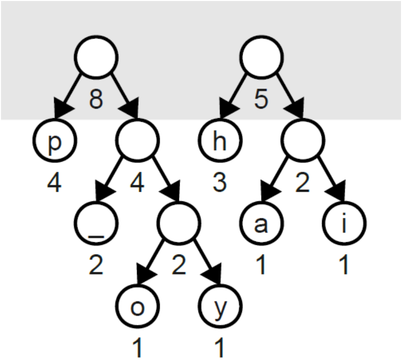
Again, we pull out the two smallest nodes and build a tree of weight 4:



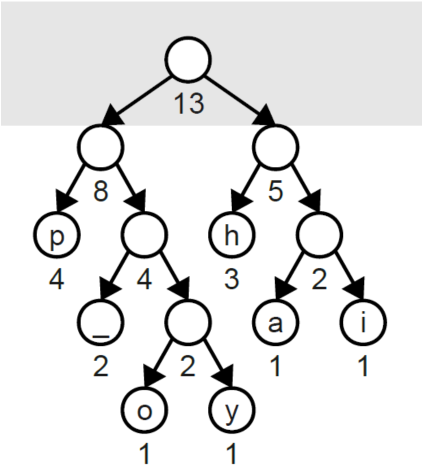
Note when we build a combined node, it doesn't represent a character like the leaf nodes do. These *interior nodes* are used along the paths that eventually lead to valid encodings, but the prefix itself does not encode a character. One more iteration combines the weight 3 and 2 trees into a combined tree of weight 5:



Combining the two 4s gets a tree of weight 8:



And finally, we combine the last two to get our final tree:



The root node of the final tree will always have a weight equal to the number of characters in the input file (which might be a good thing to check when testing). Note that this tree is different from the tree on page 4, and has slightly different bit patterns, but both trees are optimal and the total number of bits required to encode `"happy hip hop"` is the same for either tree. When we have choices among equally weighted nodes (such as in the first step choosing among the four characters with weight 1) picking a different two will result in a different, but still optimal, encoding. Similarly when combining two subtrees, it is as equally valid to put one of the trees on the left and the other on the right as it is to reverse them.

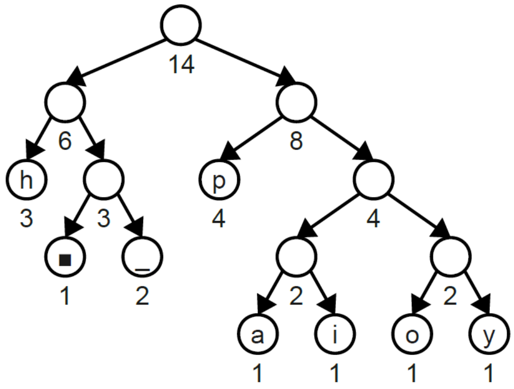
Remember that it is essential that you use the same tree to do both encoding and decoding of your files. Since each Huffman tree creates a unique encoding of a particular file, you need to ensure that your decoding algorithm generates the exact same tree, so that you can get back the file.

Practical Considerations: The Pseudo-EOF. The preceding discussion of Huffman coding is correct from a theoretical perspective, but there are a few real-world details we need to address before moving on. One important concern is what happens when we try to store a Huffman-encoded sequence on disk (in a file). Each file on your computer is typically stored as a number of bytes (groups of eight bits); files are usually measured in "megabytes" and "gigabytes" rather than "megabits" or "gigabits." As a result, if you try to write a Huffman-encoded string of bits into a file, if you don't have exactly a multiple of eight bits in your encoding, the operating system will typically pad the remainder with random bits to get to 8.

For example, suppose that we want to encode the string "ahoy" using the above Huffman tree. This results in the following sequence of bits: `1101001100111`. By my cipherin' this is exactly thirteen bits, which means that, when stored on disk, the sequence would be padded with three extra random bits. Suppose that those bits are `111`. In that case, the bit sequence would be written to disk as: `1101001100111111`. If we were to then load this back from disk and decode it into a sequence of characters, we would get string "ahoyi," which is not the same string that we started with!

Even worse, if those random bits end up being `000`, then the stored bit sequence would be `11101001100111000`. The problem is that as we decode this, we read the first thirteen bits back as "ahoy" but encounter an error when reading the last three bits because `000` is not a character in our encoding scheme.

To fix this problem, we have to have some way of knowing when we've finished reading back all of the bits that encode our sequence. One way of doing this is to transform our original input string by putting some special marker at the end. This marker won't appear anywhere else in the string and serves purely as an indicator that there is nothing left to read. For example, we might actually represent the string `"happy hip hop"` as `"happy hip hop■"`, where ■ marks the end of the input. When we build up our Huffman encoding tree for this string, we will proceed exactly as before, but would add in an extra node for the ■ marker. Here is one possible encoding tree for the characters in this new string:



Now, if we want to encode `"happy hip hop■"`, we get the following bit string: `001100101011110110011011001100111010010`. This does not come out to a multiple of eight bits (specifically, it's 39 bits), which means that it will be padded with extra bits when stored on disk. However, this is of no concern to us - because we have written the ■ marker to the end of the string, as we're decoding we can tell when to stop reading bits. For example, here is how we might decode the above string:

```
00    H
1100  A
10    P
10    P
1111  Y
011
00    H
1101  I
10    P
011
00    H
1110  O
10    P
010   ■
0     Extra bits ignored; we knew to stop when seeing ■
```

This ■ character is called a *pseudo-end-of-file character* or *pseudo-EOF* character, since it marks where the logical end of the bit stream is, even if the file containing that bit stream contains some extra garbage bits at the end. When you actually implement the Huffman encoding program, you will have to make sure to insert a pseudo-EOF character into your encoding tree and will have to take appropriate steps to ensure that you stop decoding bits when you reach it.

> ***Optional nerd stuff:*** *There is a difference between PSEUDO_EOF and a "real" EOF. The PSEUDO_EOF is a fake value that our program is using to signal the end of compressed data in a file. A real EOF is not ■ or null or -1. It is not a character or integer value at all; it is something decided internally by the operating system. The real file system knows where the end of a file is because there is a master table of data about all the files on the disk, and that table stores every file's length in bytes. The OS doesn't insert any special character at the end of each file; it just knows that you have hit the end-of-file once you have read a certain number of bytes equal to that file's length.*

Practical Considerations: The Encoding Table. There is one last issue we have not discussed yet. Suppose that I want to compress a message and send it to you. Using Huffman coding, I can convert the message (plus the pseudo-EOF) into a string of bits and send it to you. However, you cannot decompress the message, because you don't have the encoding tree that I used to send the message.

There are many ways to resolve this. We could agree on an encoding tree in advance, but this only works if we already know the distribution of the letters in advance. This might be true if we were always compressing normal English text, but in general is not possible to do.

A second, better and more flexible option, is to either prefix the bit sequence with a header containing enough information to reconstruct the Huffman encoding tree or supply the encoding tree in a separate file. There are many options you have for reading and writing the encoding table. You could store the table at the head of the file in a long, human-readable string using the ASCII characters '`0`' and '`1`', one character entry per each line, like this:

```
h = 01
a = 000
p = 10
y = 1111
...
```

Reading this back in would allow you to recreate the tree path by path. You could have a line for every character in the ASCII set; characters that are unused would have an empty bit pattern. Or you could conserve space by only listing those characters that appear in the encoding. In such a case you must record a number that tells how many entries are in the table or put some sort of sentinel or marker at the end so you know when you have processed them all.

As an alternative to storing sequences of ASCII '`0`' and '`1`' characters for the bit patterns, you could store just the character frequency counts and rebuild the tree again from those counts in order to decompress. Again we might include the counts for all characters (including those that are zero) or optimize to only record those that are non-zero. Here is how we might encode the non-zero character counts for the `"happy hip hop"` string (the 7 at the front says there are 7 entries to follow – 6 alphabetic characters and the space character):

```
7 h3 a1 p4 y1  2 i1 o1
```

It would probably be easier, while developing a Huffman compression program, to store the encoding tree in a (human-readable) file, separate from the compressed (encoded) text. If you were selling this software, however, you probably wouldn't want use the extra space.

Greedy algorithms. Huffman's algorithm is an example of a *greedy* algorithm. In general, greedy algorithms make "local" minimal/maximal choices in attempt to result in a global minimum/maximum. At each step, the algorithm makes the near choice that appears to lead toward the goal in the long-term. In Huffman's case, the insight for its strategy is that combining the two smallest nodes makes both of those character encodings one bit longer (because of the added parent node above them) but given these are the most rare characters, it is a better choice to assign them longer bit patterns than the more frequent characters. The Huffman strategy does, in fact, lead to an overall optimal character encoding.

Even when a greedy strategy may not result in the overall best result, it still can be used to approximate when the true optimal solution requires an exhaustive or expensive traversal. In a time or space-constrained situation, we might be willing to accept the quick and easy-to-determine greedy solution as an approximation.

Borrowed with minor modifications from
The Huffman Encoding and Data Compression handout, Stanford CS106B
https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1174/handouts/210%20Huffman%20Encoding.pdf