Huffman Coding

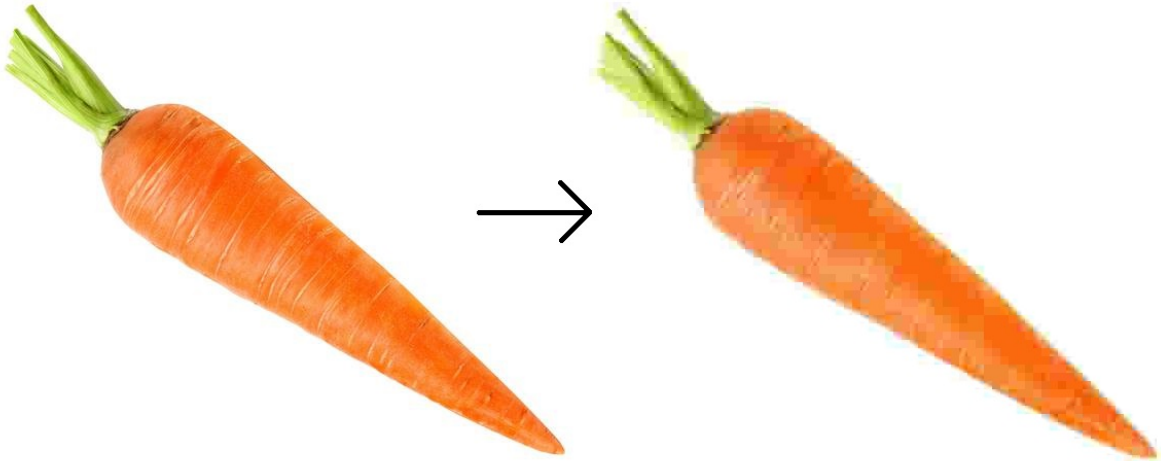
What is Huffman Coding and its history

Problem

When you put English text into a computer, it is saving every individual character as eight bits of data. Eight ones and zeros. A modern phone or computer might store a few quadrillion of them. But every time your phone complains that the storage is full, you're running up against the same question that computer scientists have been working on: How do we reduce the bits used to store things?

How computer compress texts

Our computer compresses images and videos differently to text - they use lossy compression for images and videos. This means even if the images and videos are blurry and pixelated, you can still see its content. For example, the below image is compressed heavily but it still represents a carrot:



As you can see, it doesn't matter if you lose a little bit of detail.

But text has to be loss-lessly compressed: if you lose a little bit of detail, your original word will change into a completely different word. So the first thing we need to know is how text is stored on disk before it's compressed. On a modern computer, each English character takes up exactly eight ones and zeros on disk - eight bits, or one byte. And there are 256 possible combinations of those ones and zeros, so you can have 256 possible characters. That's enough for the English alphabet, numbers, some punctuation,... If you want to know how long a string of text is, you just count the bits and you divide by eight. But, there is 1 problem: Computers don't have the luxury of spaces. So how can we solve this? A good plans will be discussed below.

A (good?) approach

First, we assign the most common characters to the smaller arrangements of bits. Example: the space bar is generally
used most often, so we give it the code "0". Then the second most used characters is lowercase "e", so give it the code "1". Then give lowercase "t" a code "00" and etc...

But, this approach has a problem. If the computer running through the text, it has no way to know whether "00" is a "t", or double space bar. Similarly, "000" is a lowercase "n", or a "t" followed by a space, or triple space? Because there are no gaps here, the computer can only see a constant stream of 1 and 0. But, in 1952 a mathematician called David Huffman (hence the name Huffman Coding) invented Huffman Coding to solve this problem.

Main data structures and algorithms of Huffman Coding

First, we talk about the algorithms first and data structure we used later.

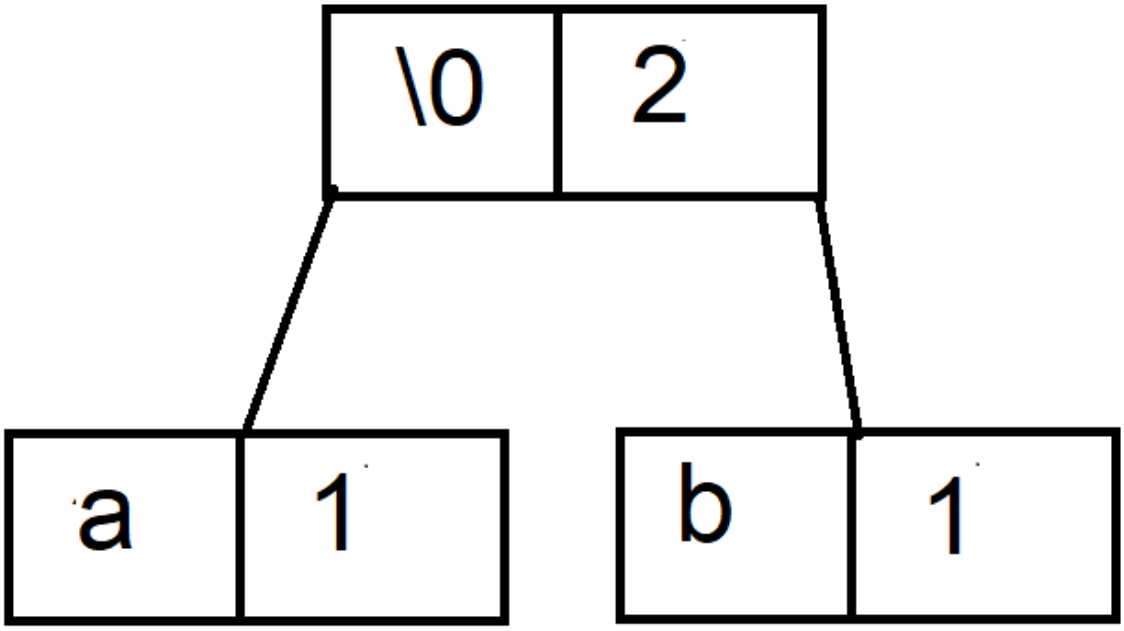
Algorithms

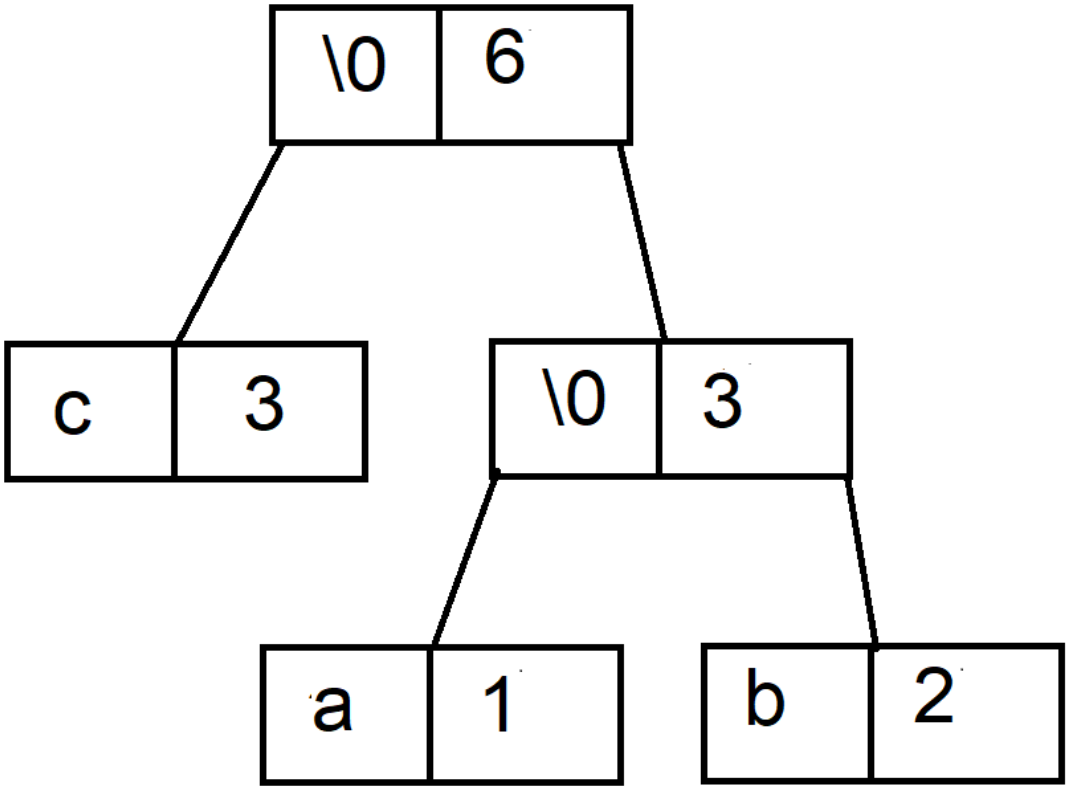
For example, we want to compress the line "abbcccdddd". When uncompressed, that is characters taking up to 80 bits.

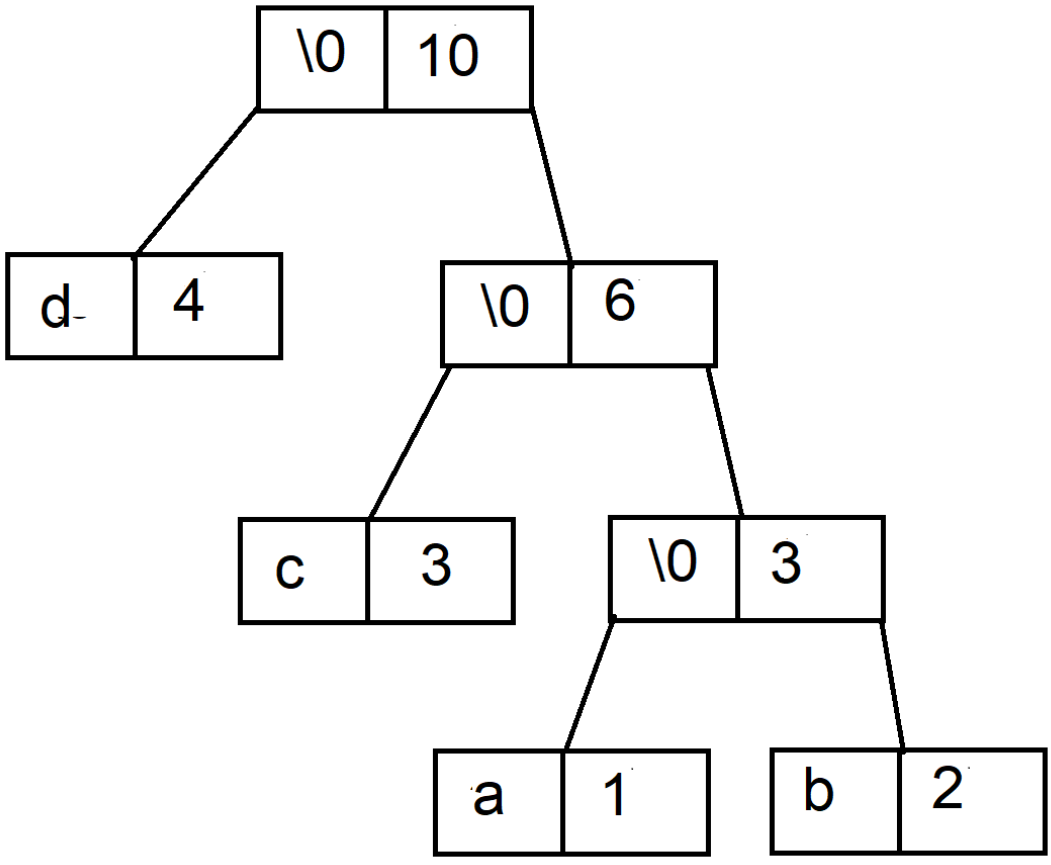
First, we count how many times each character is used and put it in a list in order.

Character	a	b	c	d
Frequency	1	2	3	4

Now take the two least used characters. Those two are going to be the bottom branches on the “Huffman tree”. Write them down, with how often they're used next to them. That’s called their frequency. Then connect them together, one level up,
with the sum of their frequencies. Now add that new sum back into your list, wherever it sits, higher up, and repeat until there are only 1 node left.







You now have a Huffman tree and it tells you how to convert your text into 1 and 0. From the root, each time you take the left hand side, write a 0; and each time you take the right hand side, write a 1 until you meat a leaf node. The written string will be the code for that node's character. So "d" will be "0", "c" will be "10", "a" will be "110" and "b" will be "111". Some of the letters will take up more than 8 bits, but that’s fine, because they’re not used very often. You do have to also

store this tree to provide a translation table between your new symbols and the uncompressed text. We compressed it down from 80 to only 19 bits.

To uncompress the resulting stream of bits, it works the other way: just read across, take the left turn every time you see a 0 and the right turn every time you see a 1. When you reach a leave node, write down that leave node's character.

Data structure

We create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root). Then, extract two nodes with the minimum frequency from the min heap. We create a new internal node with the frequency equals to the sums of the two extracted nodes. The first extracted node will be the new node's left child and the second extracted node will be the new node's second child. Then add this new node to the min heap. We repeat until there are only 1 node in the min heap.

Time complexity and space complexity evaluation

Assuming an encoded text string of length n and an alphabet of k symbols.

Time complexity

The time complexity of the Huffman algorithm is $O(nlogn)$. Using a heap to store the weight of each tree, each iteration requires $O(logn)$ time to determine the cheapest weight and insert the new weight. There are $O(n)$ iterations, one for each item.

Space complexity evaluation

Space complexity is $O(k)$ for the tree and $O(n)$ for the decoded text.

Application

Prefix codes nevertheless remain in wide use because of their simplicity, high speed, and lack of patent coverage. They are often used as a "back-end" to other compression methods. Deflate (PKZIP's algorithm) and multimedia codecs such as JPEG and MP3 have a front-end model and quantization followed by the use of prefix codes; these are often called "Huffman codes" even though most applications use pre-defined variable-length codes rather than codes designed using Huffman's algorithm.

References

http://www.huffmancoding.com/my-uncle/scientific-american

https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

https://www.youtube.com/channel/UCBa659QWEk1AI4Tg--mrJ2A