Honey, I Shrunk the File

Step 1

In 1951, David A. Huffman and his classmates in an Information Theory course were given the choice of a term paper or a final exam. The topic of the term paper: Find the most efficient method of representing numbers, letters or other symbols using a binary code. After months of working on the problem without any success, just as he was about to give up on the enticing offer of skipping an engineering course's final exam, Huffman finally solved the problem, resulting in the birth of **Huffman Coding**, an algorithm that is still widely used today.

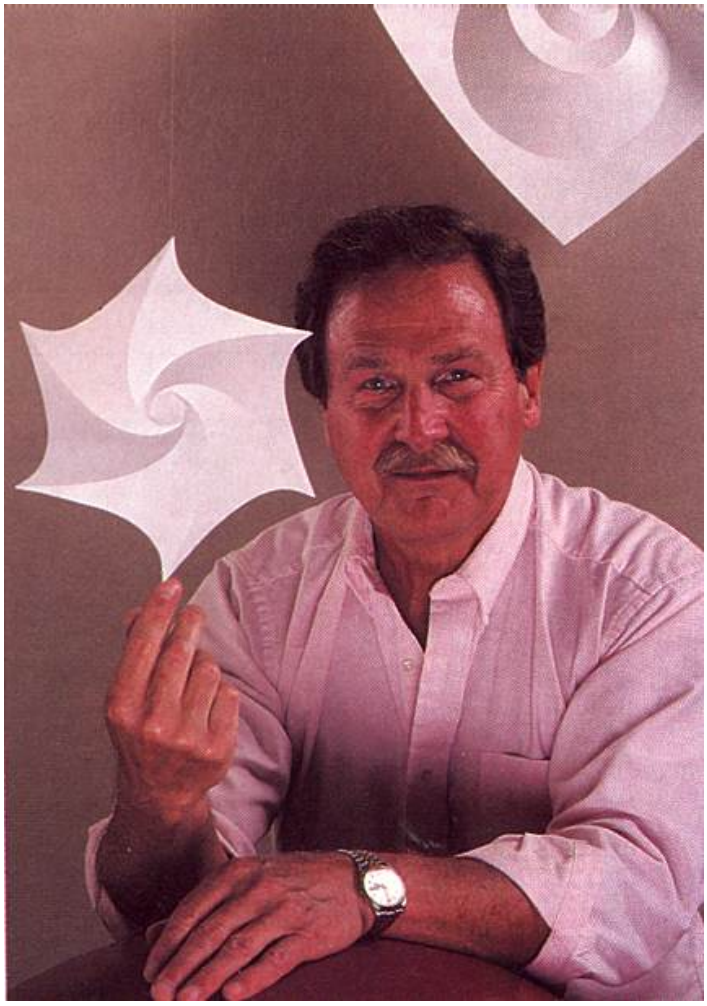


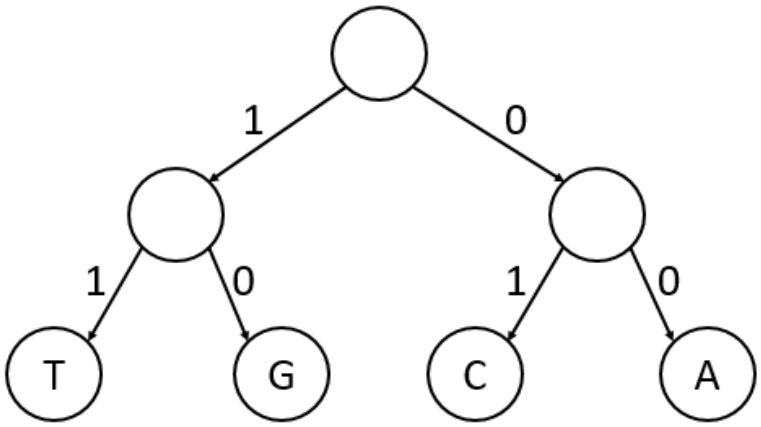
Figure: David Huffman expresses mathematical theorems in intricate paper sculptures (Photo: Matthew Mulbry)

Step 2

Say we have a set of *N* possible items and we want to represent them uniquely. We could construct a **balanced binary tree** of height $\lceil \log_2(n) \rceil$. Each item would be represented by a unique sequence of no more than $\lceil \log_2(n) \rceil$ bits (1 for right child, 0 for left child) that start at the root of the tree and continue to the leaf where that item is stored. Since data items are only in leaves, this gives a **prefix code**: no symbol is represented by a sequence that is a prefix of a sequence representing another symbol.

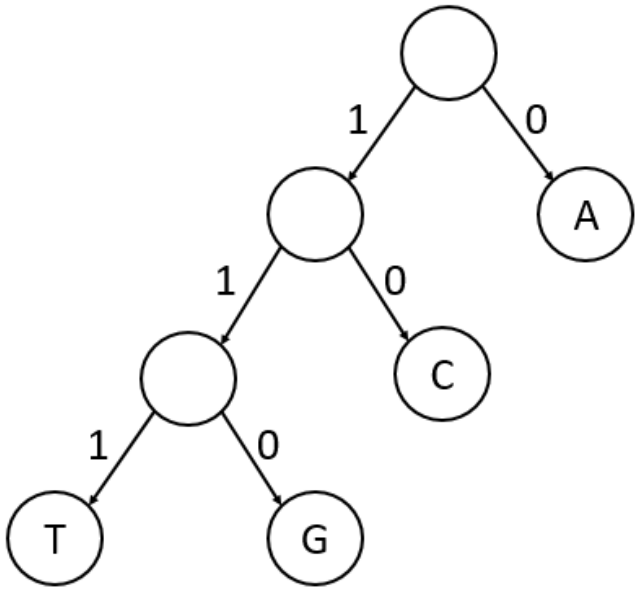
To be able to successfully encode and decode a message, our code *must* be a prefix code. For example, say (**A**, **C**, **G**, **T**) maps to (**0**, **1**, **10**, **11**) and I ask you to decode the string **110**. This mapping is *not* a prefix code: **C** (**1**) is a prefix of **G** (**10**) and **T** (**11**). Because of ambiguity arising from this fact, we are unable to exactly decode the message: **110 = CCA**, **110 = CG**, and **110 = TA**, so we fail to recover the original message.

Thus, it is clear that our code *must* be a prefix code, but in addition to the term "prefix code," we also mentioned that we could theoretically construct a *balanced* binary tree. However, do we want our encoding tree to necessarily be perfectly balanced? We saw in the previous examples that a perfectly balanced binary tree (**00**, **01**, **10**, **11**) may not be optimum if the symbol frequencies vary. Below is the tree representing our initial symbol mapping:



Step 3

When we used the unequal-length mapping (**0**, **10**, **110**, **111**), even though some (less frequent) symbols were encoded with more than $\log_2(n)\rceil$ bits, because the **more frequent** symbols were encoded with **less than** $\lceil\log_2(n)\rceil$ **bits**, we achieved significantly better compression. How do I know what the "best possible case" is for compressing a given message? In general, the entropy (in "bits") of a message is $\sum_i p_i \log \frac{1}{p_i}$ over all possible symbols *i*, where p_i is the frequency of symbol *i*. This entropy is the **absolute lower-bound** of how many bits we can use to store the message in memory. Below is the tree representing the improved symbol mapping, which happens to be the encoding tree that the Huffman algorithm would produce from our message:



Huffman's algorithm can be broken down into three stages: **Tree Construction**, **Message Encoding**, and **Message Decoding**.

Step 4

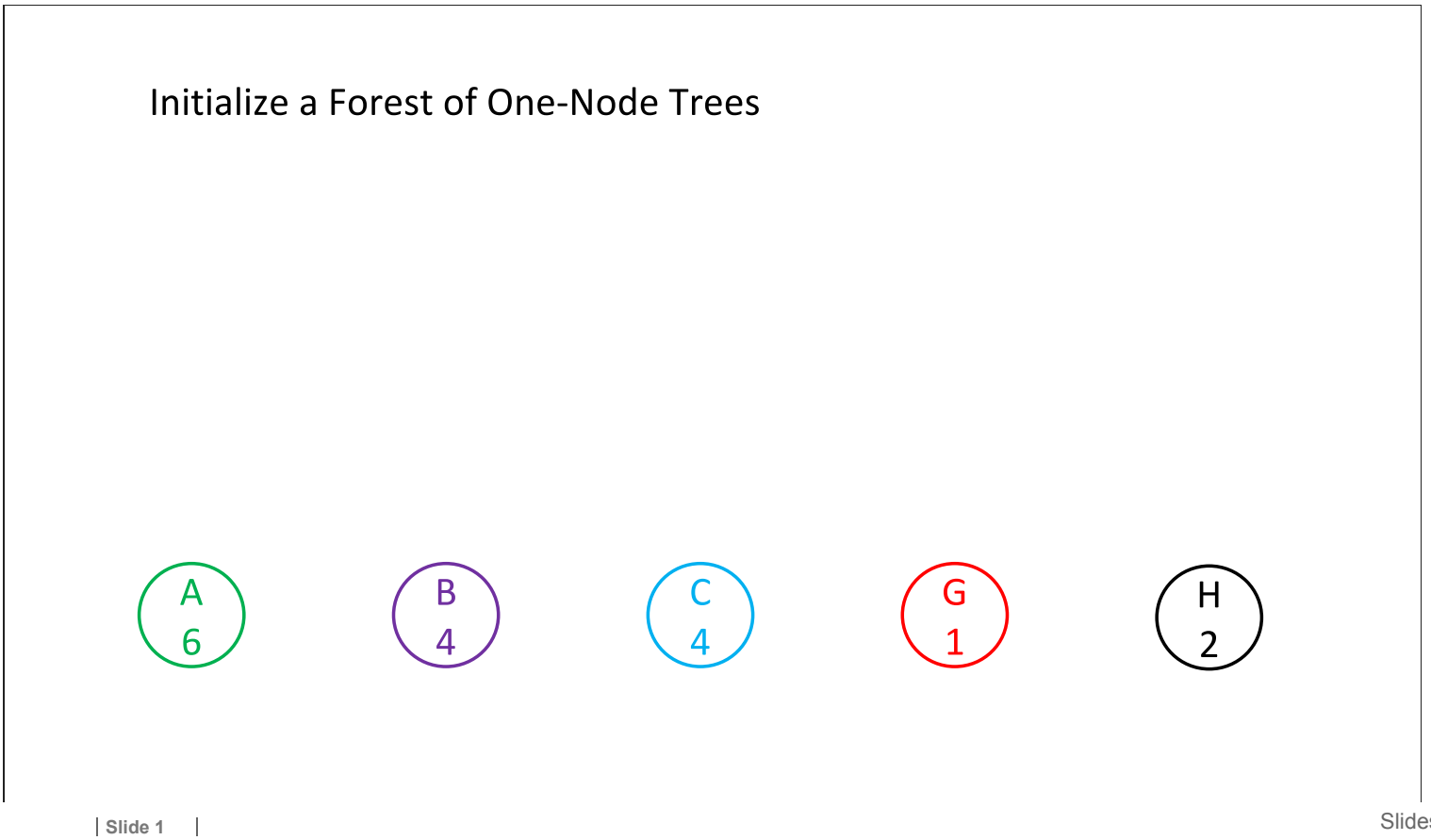
EXERCISE BREAK: Which of the following statements about Huffman compression are true?

Step 5

Huffman's Algorithm - Tree Construction: The algorithm creates the Huffman Tree in a **bottom-up** approach:

1. Compute the frequencies of all symbols that appear in the input
2. Start with a "forest" of single-node trees, one for each symbol that appeared in the input (ignore symbols with count 0, since they don't appear at all in the input)
3. While there is more than 1 tree in the forest:
    1. Remove the two trees (T1 and T2) from the forest that have the lowest count contained in their roots
    2. Create a new node that will be the root of a new tree. This new tree will have T1 and T2 as left and right subtrees. The count in the root of this new tree will be the sum of the counts in the roots of T1 and T2. Label the edge from this new root to T1 as "1" and label the edge from this new root to T2 as "0"
    3. Insert this new tree in the forest, and go back to the While statement
4. Return the one tree in the forest as the Huffman tree

Let's create a Huffman Tree from the string **AAAAABBAHHBCBGCCC**. In the visualization below, use the arrows in the bottom left to step through the slides.



Step 6

Code Challenge: Counting Symbol Frequencies in a Message

In this challenge, you will be given a randomly-generated `string` object. Your task will be to create an array of integers (named `counts`) of length 256, where `counts[i]` is the number of times the *i*-th ASCII symbol appeared in the input `string`.

Sample Input:

```
AAAAABBAHHBCBGCCC
```

Sample Output:

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 6 4 4 0 0 0 1 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
```

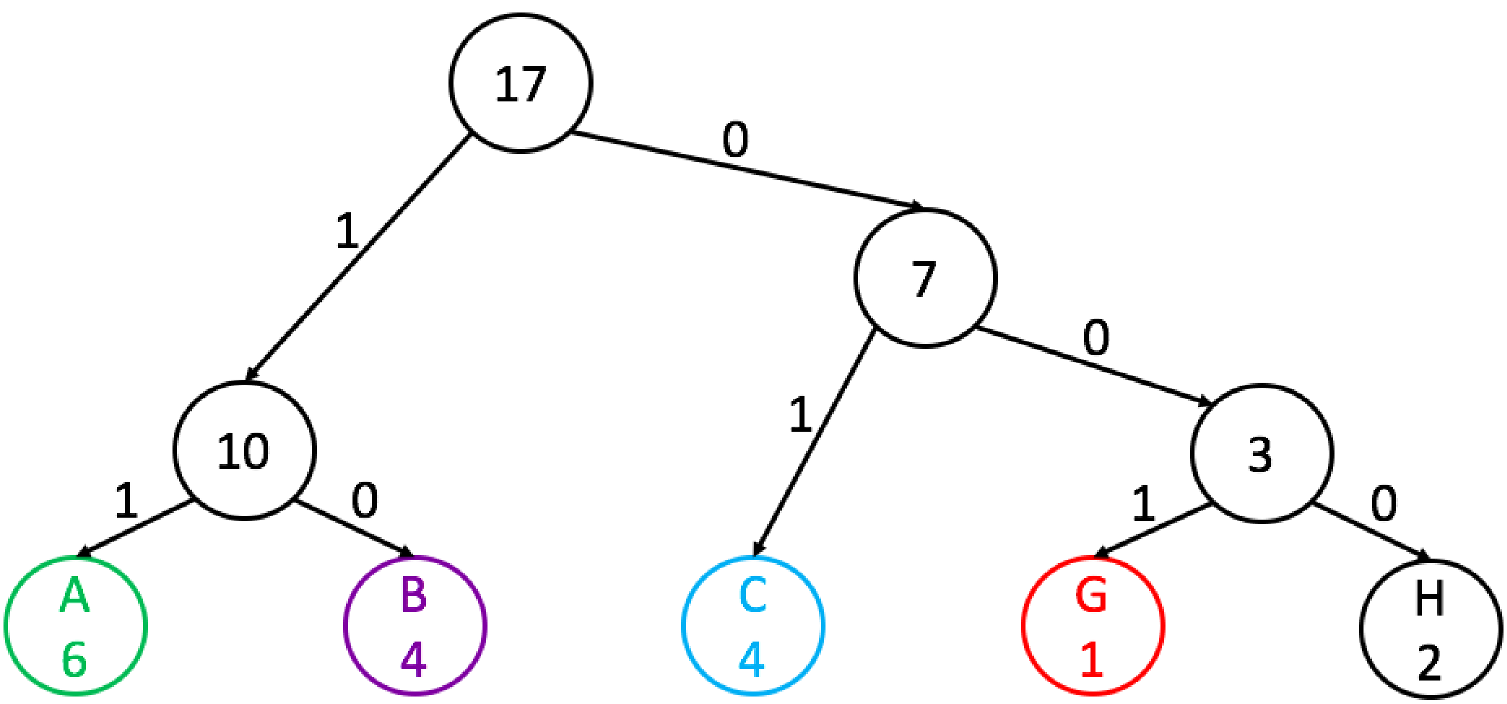
To solve this problem please visit https://stepik.org/lesson/26175/step/6

Step 7

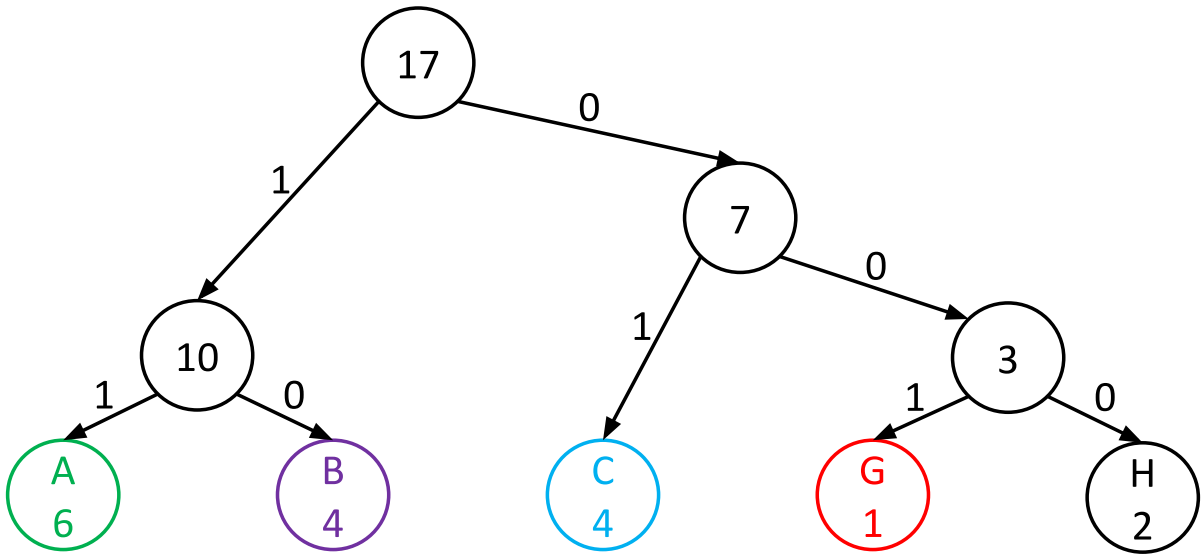
Huffman's Algorithm - Message Encoding:

Once we have a Huffman Tree, encoding the message is easy: For each symbol in the message, output the sequence of bits given by 1's and 0's on the path from the root of the tree to that symbol. A typical approach is to keep a pointer to each symbol's node, and upon encountering a symbol in the message, go to that symbol's node (which is a **leaf** in the tree) and follow the path to the root with recursive calls, and output the sequence of bits as the recursive calls pop from the runtime stack. We push and pop bits to/from a stack because a leaf's encoding is the path from the *root* to the *leaf*, but in this traversal we described, we follow the path from *leaf* to *root*, meaning each leaf's encoding will be *reversed*! So using a stack to push/pop helps us reverse the path labels in order to get the true encoding.

Recall from the previous Step that the completed Huffman Tree created from our message is as follows:



Let's encode our message (**AAAAABBAHHBCBGCCC**) using the Huffman Tree we created. In the visualization below, use the arrows in the bottom left to step through the slides.



Original Message: AAAAABBAHHBCBGCCC
Encoded Message:

| Slide 1 |

Slides

Our original message contained 17 symbols, and thus takes 17 * 8 = 136 bits to represent in ASCII. Our encoded message, however, takes only 37 bits to represent!

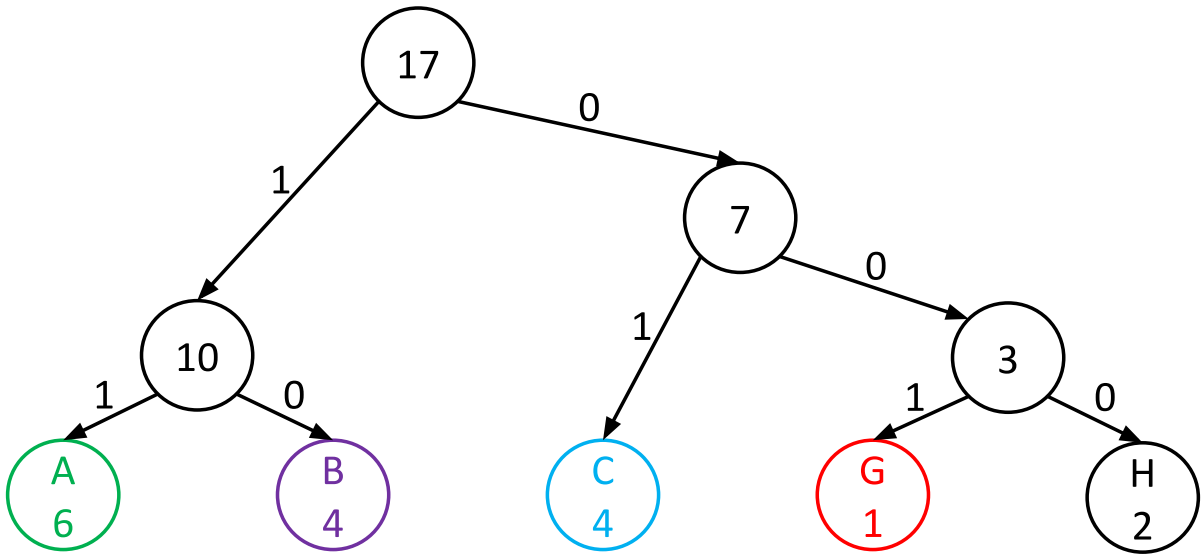
Step 8

Huffman's Algorithm - Message Decoding:

If we have the Huffman Tree, decoding a message is easy:

- Start with the first bit in the coded message, and start with the root of the code tree
- As you read each bit, move to the left or right child of the current node, matching the bit just read with the label on the edge
- When you reach a leaf, output the symbol stored in the leaf, return to the root, and continue

Let's decode the encoded message we just created (**11111111111010110000001001100010101010101**) using the same Huffman Tree. In the visualization below, use the arrows in the bottom left to step through the slides.



Encoded Message: 11111111111010110000001001100010101010101
Original Message:

| Slide 1 |

Slides

NOTE: Multiple Huffman Trees can be constructed from the same message (e.g. we could simply swap 1's and 0's in our tree). To be able to decode a message, we must ensure we're using the **same Huffman Tree** that was used to encode the message! Therefore, the tree, or enough information to reconstruct the tree exactly, must be included before the beginning of the coded message (the header I mentioned earlier) or in some other way transmitted to the receiver.

Thus, in practice, the first step of the **Message Encoding** stage is to write to the compressed file some header with information about symbol frequencies (so that the recipient can rebuild the Huffman Tree), and the first step of the **Message Decoding** stage is to read in the header of the file and actually rebuild the Huffman Tree.

Step 9

Huffman Compression is just one approach to file compression, but many other methods exist as well. Even with Huffman Compression, clever techniques exist to further improve its compression abilities. For example, biologically, different segments of an organism's genome may have different symbol frequencies. Say, for example, that the first half of a genome is enriched in C's and G's, whereas the second half of the genome is enriched in A's and T's. If we were to split the genome into two pieces, Huffman Compress each piece independently of the other, and then merge these two encoded messages (including some indicator in the merged encoded message that tells us when one segment has finished and the other has started), we will be able to attain even better compression than if we were to Huffman Compress the entire genome in one go.

Also, note that Huffman Compression is a **lossless** compression algorithm, meaning we can compress a file, and when we decompress the resulting file, it will be completely identical to the original. However, we would be able to achieve better compression if we removed this lossless requirement. Such algorithms, in which the original file can *not* be extracted from the compressed file, are called **lossy** compression algorithms. For example, when it comes to music, the lossless source file of a typical song 4 minutes in length will be approximately 25 MB. However, if we were to perform lossy compression on the file by encoding it using the MP3 audio compression algorithm, we can compress the file to somewhere in the range of 1 to 5 MB depending on what bitrate at which we choose to compress the file.

We live in an age of data, so file compression is vital to practically every part of our daily life. With better file compression, we could:

- Store more songs and movies on our mobile devices
- Speed up our internet browsing
- Archive important files in a feasible fashion
- Do infinitely more cool things!