# Huffman Coding

Dhruba Das

Spring Term 2024

## Introduction

In this assignment, we deal with Huffman coding; we deal with both and the encoding and decoding aspects of this. To give a brief overview, in this assignment, we achieve the encoding by encoding frequent characters with few bits and infrequent characters with more bits AND to keep it simple, we use lists of ones and zeroes to represent sequence of bits.

## The freq functions

The *freq* functions are responsible for the generation of a frequency table for the characters in the provided string; it recursively counts the occurrences of each character in the string. A frequency is represented using a tuple which looks like this: {character, frequency}, where character is the ASCII code for that particular char. and frequency is how many times it appears in the provided string. There is one base case, which returns the list of frequencies, given an empty list, and the list of frequencies (also called *freq*):

```
def freq(sample) do freq(sample, %{}) end
def freq([char | rest], freq) do
    case Map.get(freq, char) do
      nil ->
        freq(rest, Map.put(freq, char, 1))
      f ->
        freq(rest, Map.put(freq, char, f+1))
    end
end
```

## Huffman Tree

The next step is to create a Huffman tree, which is a binary tree where each leaf node represents a character from the input alphabet, and each internal node represents a merged subtree.

We have the list given to us by the *freq* function, which is then given as input to the **huffmantree** function. This function first sorts the list of frequency tuples in ascending order of frequency. This is crucial, because in Huffman coding, nodes with lower frequencies are typically placed higher in the tree. It then calls the **huffman** function with the sorted list of frequency tuples:

```
def huffman_tree(freq) do
    sorted = Enum.sort(freq, fn({_, f1}, {_, f2}) -> f1 < f2 end)
    huffman(sorted)
end
```

The **huffman** function has just one base case: if the list contains only one tuple, it means there's only one character in the input sample. In this case, the function simply returns the tree containing that single character.

However, if the list contains more than one tuple, the function takes the first two tuples (which represent the characters with the lowest frequencies) and combines them into a new node. The frequency of this node is the sum of the frequencies of the two characters. The function then recursively calls itself with the new node inserted into the list of tuples while maintaining the sorted order:

```
def huffman([ {tree1, f1}, {tree2, f2} | rest]) do
    node = { {tree1, tree2}, f1 + f2}
    huffman(insert(node, rest))
end
```

## Encoding the tree

There's two functions which are involved in the encoding of the Huffman tree: *encodetable* and *encode*.

The *encodetable* function is responsible for creating an encoding table based on the provided Huffman tree. It has one base case: when the input Huffman tree is a leaf node representing a character, it adds an entry to the encoding table, mapping the character to its path (a list of 0s and 1s representing the path from the root of the Huffman tree to the leaf node). When the input Huffman tree is an internal node, it recursively calls itself for the left and right branches: for the left branch, it appends 0 to the path AND for the right branch, it appends 1 to the path. It then combines the results from the left and right branches to build the complete encoding table:

```
def encode_table(tree) do
    encode_table(tree, [], %{})
end
def encode_table({zero, one}, path, table) do
```

```
    table = encode_table(zero, [0|path], table)
    encode_table(one, [1|path], table)
end
```

We then pass the encoding table to the *encode* function, which is responsible for encoding a given string using the provided encoding table. It has one base case: if the input string is empty, it returns an empty list, as there's nothing to encode. Otherwise, if the input string is not empty, it processes each character of the string by looking it up the character in the encoding table; if the character is not found in the encoding table (meaning it's not in the Huffman tree), it prints an error message indicating that the character could not be encoded. If the character is found in the encoding table, it appends the corresponding sequence to the encoded result. It then recursively continues encoding the remaining characters of the input string:

```
def encode([char|rest], table) do
    case Map.get(table, char) do
      nil ->
        IO.puts("error could not encode #{[char]}", [char])
        encode(rest, table)
      seq ->
        seq ++ encode(rest, table)
    end
end
```

## Decoding

There are also two steps/functions which are responsible for the decoding: **decodetable** and **naive**.

The **decodetable** function is responsible for creating a decoding table from the encoding table. It uses *Enum.reduce* to iterate over each entry in the encoding table; for each entry, it pattern matches on a tuple containing a character and its corresponding sequence. It uses *Map.put* to add an entry to the accumulator and the function continues this process for each entry in the encoding table until all entries have been processed.

The **decode** function recursively decodes an encoded sequence using the decoding table obtained from the **decodetable** function. It has one base case: if the input encoded sequence is an empty list, the function simply returns an empty list, indicating that the decoding process is complete. Otherwise, it uses *Enum.split* to split the input encoded sequence into two parts: the first $n$ elements and the rest of the sequence. It then attempts to retrieve a character associated with the extracted sequence rom the decoding table using *Map.get*. If the character is found, it appends the character to the result and recursively calls *naive* with the remaining encoded sequence,

resetting the splitting parameter $n$ to 1. If the character is not found, it increments $n$ by 1 and recursively calls itself with the updated $n$ and the same decoding table. This process continues until the entire encoded sequence is decoded or until there's no match found in the decoding table for any sequence in the encoded sequence:

```
def naive(encoded, n, dtable) do
  {seq, rest} = Enum.split(encoded, n)
  case Map.get(dtable, seq) do
    nil ->
      naive(encoded, n+1, dtable)
    char ->
      [char | naive(rest, 1, dtable)]
  end
end
```

## Tests and Performance

Tests on the program were run using the **kallocain.txt** sample text provided to us in canvas. I received the following output from the terminal:

```
chars 318997
bytes 333816
compressed: 178600
```

The time taken for the entire test, i.e encoding and decoding, was approximately 145 milliseconds, with 135 milliseconds being the least amount of time taken AND 155 milliseconds being the most amount of time taken.