Huffman

Sam Khosravi

March 2022

1 Introduction

In this task we were supposed to implement the Huffman algorithm. The idea of huffman is to assign different lengths of codes to each input character based on the frequency of which they occur. The character that is used the most frequently will be given the smallest code, while the least frequent character will be given a larger code. The larger the code, the deeper in the tree we will have to traverse to get the character. The code is basically the adress to the character in the tree.

2 Huffman Table

To be able to even create a tree, we first need to create the table which contains the frequencies of all our character. We need to run through the sample list and count all characters and how many times they occur. This is done by freq/1, which will call on freq/2. The basecase is reached when the sample text has been traversed. The function freq/2 is opened up one char at a time, and this char will be sent together with the list of frequencies to another method addfreq/2.

```
def freq(sample) do
  freq(sample, [])
  end

def freq([], freq) do
  freq
end

def freq([char | rest], freq) do
  freq(rest, addfreq(char, freq))
end
```

The frequencies will be added to the table in form of a tuple which contains the character in ASCII number and also its frequency. The first part is the basecase, where if we have traversed the list and there is no character that matches we will only have one character, and because of that we will return the character with the frequency 1. However, if we find a character in the list while going through it, we will increase our frequency. If our character does not match the one after it, we will add it to our list to not lose the not matching character and then tailrecursively call on addfreq to check the if the next character matches our char and so on.

```
def addfreq(char, []) do
  [{char, 1}]
end

def addfreq(char, [{char, n}|tail]) do
  [{char, n+1}|tail]
end
def addfreq(char, [head|tail]) do
  [head|addfreq(char, tail)]
end
```

To add to this, I used my insertion sort method with a tweak from the first exercise, and used it to sort the frequency list before sending it to the huffman tree function.

```
def tree(sample) do
  freq = freq(sample)
  huff_tree(isort(freq))
  end
```

3 Huffman Tree

A Huffman Tree is a tree where we have characters on the leafs. The less frequently a character appears, the longer its branch will be and vice versa. Each leaf is here represented by a tuple that contains two nodes that are a single character each.

The frequency we will send in is sorted, which lets us pick out the first two nodes and their frequencies, adding them together (will be used at the end to see the full bite size), and thencalling a format/2 function with the added frequencies. This function will give us the tree in a understandable form.

```
def huff_tree([yeye]) do yeye end

def huff_tree([{char1, freq1},{char2, freq2}| tail]) do
  huff_tree(format({{char1, char2}, freq1+freq2}, tail))
end
```

The way format works is that it will accept a tuple with a character and its frequency. If the frequency of the first character is lower than the frequency of the second character, we will append them to our list in that order. However, if the middle function is not satisfied, that means that the frequency of the second character has a lower frequency, and should be at the left side of the node. This goes on for the entirety of the list.

```
def format({char1, freq1},[]) do [{char1, freq1}] end
def format({char1, freq1},[{char2,freq2}|tail]) when freq1 < freq2 do
  [{char1, freq1}, {char2, freq2} | tail]
end
def format({char1, freq1}, [{char2, freq2}|tail]) do
  [{char2, freq2} | format({char1, freq1}, tail)]
end
```

4 Encoding Table

Now that the Huffman tree is done, we can try to extract the adresses of each character or their codes. These basically tell us how to reach the character. The encoding table will collect all characters in a tuple together with the path/code to get to them.

The input will be split up and we will send the tree part to another function encodetable/2. The idea here is that we will go down to the lower left node, and from there work our way into each node and concatenate the lists into one comeplete list. At the end the list is reveresed to give the correct format and this is done with the reverse function from the first most task in the course.

```
def encode_table({left,right}) do
  encode_table(left, [])
end

def encode_table({left,right}, list) do
  encode_table(left, [1|list]) ++ encode_table(right, [0|list])
end

def encode_table(tree, list) do
[{tree, reverse(list)}]
end
```

5 Huffman encoding

Here we basically want a list of all the sequences made for each character. We have a function encode/2, which will take in the encoding table made before and also our tree. The goal here is to match the character in the encoding table with the one in the tree, and from there take sequence of that character. This is done in the traverse/2 function which will be explained after the encode/2 function.

Encode/2 reaches its base case when our encode table is empty. Otherwise we will take the head character of our encoding table and call on traverse/2 with it and the tree. We then tail recursively call on encode/2 until we have gone through the whole encoding table.

```
    def encode([], tree) do
    []
  end

  def encode([bokstav|tail], tree) do
    List.flatten([traverse(bokstav, tree) |encode(tail, tree)])
  end
```

If we find the character we are looking for we will return its path which is the sequence we need to get to it in the tree. If we don't find it, we call on traverse again with the tail instead and continue looking. We don't need any basecase here as we know that what we are looking for will be in the tree.

```
  def traverse(bokstav, [{bokstav, path}|_]) do path end

  def traverse(bokstav,[{intebostav, _} | tail]) do
    traverse(bokstav,tail)
  end
```

When I ran the program, I kept on getting into a infinite loop, and I found out that the problem was that I had encoded my table in the wrong way, which led to the traverse/2 function never finding a match. This helped me go back and fix the issue and kind of worked as a unintentional debugging tool.

6 Huffman decoding

As we us the same table for decoding as we did for encoding, I simply used the same code for both encode/1 and decode/1. The code for decode/2 was given, and would call on decode/3, which was partially given and left for us to fill in the blanks.

This function takes in the sequence, a number and the table. We split the sequence on n. This will help us to check what part of the sequence is what letter. The code of the letter is set to code and the remaining numbers in the sequence are set to the rest part. We then use the code in our List.keyfind method, which works in the way that it will recieve the list of tuples and returns the first element at the position where the code is matching. If we find the match, we will return the character and then the rest of the sequence. Otherwise we will call on the function and just increment the value of n to take in one more code index from the sequence.

```
  def decode_char(seq, n, table) do
    {code, rest} = Enum.split(seq, n)
```

```
case List.keyfind(table, code, 1) do
  {bokstav, _} ->
    {bokstav, rest};
  nil ->
    decode_char(seq, n+1, table)
  end
end
```

7 Performance

The fastest time was creating the encoding table, and is so fast that I get it to 0. This makes sense because we will take the values we have and make them into a tree structure. The encoding and creating of the Huffman tree were pretty equal in the time they took. To encode we would need to traverse the whole tree for a character untill we found its match, and then we would do this for every other character too. This will will ofcourse take a lot longer to do than for example just creating the encoding table. The Huffman tree creating was slightly faster than the Huffman encoding. This as we will go through the list of tuples we are given and compare their frequencies and add them to a list depeneding on the result of the comparison. This continues for the complete list. Last the decoding took the absolutely most time to do. This as we first and foremost have code given that is not part of this report and given in the lab instructions. The given function would call on the decode char function we have and recursively add everything to a list which is returned at the end. In our code we would split the sequence until we found a valid one, if we don't find we will recursively call on our decoding function with an incremented value, and the way we look might not be very efficient as we will use the keyfind method, and it will be called many times until we find the correct part of the sequence, which I guess is not very efficient.

Huffman Tree	**Encode Table**	**Encode**	**Decode**
89 ms	0 ms	110 ms	0.9 s

Table 1: Benchmarking of the different functions for kallocain.txt

8 Conclusion

I liked this task because I learnt a new algorithm, and it feels very satisfying to gain a skill in one course that I know will be useful in another course, instead of just getting general programming knowledge which is good but not as satisfying as something concrete like learning a new algorithm. It was interesting to look up the Huffman Algorithm on youtube and see how it was implemented. When it comes to coding I felt like it was pretty straight forward in the beginning, but I struggled with making the tree a bit, but my biggest struggle was the encoding part. To begin with it took a lot of time to even get started writing any code at all, and when I had written the code I got errors on later functions the used the encoding table that was implemented incorrectly. I feel like not getting any code from the task was what made it harder for me, as I kept on creating new functions that would aid each other until I started slimming them down. Lastly I have come to the conclusion that the decode function could have been implemented more efficiently. The keyfind function might be the reason for the decoding being so slow, especially because we call on it many times