15-150: Functional Programming- Project 1

Kaustubh Iyer (kkiyer)

Part 1: Description of the Problem:

Data has gone out of control. The vast amounts of data being recorded and stored are growing exponentially, and it is important that the storage keeps up with this growth in amount of data. Compression is a powerful tool that helps to curb the amount of data and allow it to grow at its pace without having to worry about storage space.
Many compression algorithms have sprung up over the past few years. In this project I will be implementing one called "Huffman Coding". Huffman coding is a lossless data compression algorithm, based on the frequency of occurrence of characters of a string. I will be using this algorithm with my own implementation to be able to compress simple strings.

Part 2: Description of the Solution:

Hoffman's algorithm uses the frequency of occurrences of characters in a string to store them. It works by finding the number of occurrence of each separate character in the string and holding them.
It then takes the lowest two frequencies, and stores them on the left and right of a node (of a tree), the node containing their sum. It then removes these two characters from the frequency list and adds in the node instead. This process is repeated till there is only one node left on the list. Now the letters of the string are looked through, and each letter is replaced by its location on the tree generated.
This happens in the following way: the character on the left of the node is labelled 0 and the one on the right is labeled 1. To find the encoding of the character we traverse down the tree to the leaf of its location, appending the 0's and 1's as we go.
Decoding the same compressed string would need just the leaves of the tree, and we could follow the 0's and 1's down the tree till we reached a leaf and record the character.

Part 3: Usage Instructions:

To use the file, simply go to its directory on a machine with SML installed and write the command "sml "huffman.sml"". This shoud start up sml and load all the functions in the file. Now you can call encode on a string of your choice. Make sure to store the resulting tree as well as the encoded value. Now whenever you want the sequence decoded, just input the encoding as well as the tree you got with it. The decoded string will be returned to you as the output.

Part 4: Description of the Implementation:

Data structures used:

Lists, and
datatype htree =
    Hnode of htree*(string*string*int)*htree
    | Hleaf of (string*int)
Encoding is done with the following steps:

Generating a list of frequencies:

The first part of my code generates a list of tuples containing each unique character as a hleaf, along with the frequency of its occurence in the input. This is achieved by the functions: getFreq, isin and addin. isin checks if a character is in the frequency list, addin adds 1 to the frequency of a requested character, and getFreq goes through each character, and calls addin if the character is already in the list, or appends it if it isn't.

Sorting this list of frequencies:

Now that we have the frequencies in a list, I have them sorted using mergesort. I use this because it is the fastest sort that avoids risks and also because the order of elements with the same frequency doesn't matter (sort is not in place). I adapted the code for the sort from the sml file msort.sml that was given to us on piazza, and taught in lecture on ordinary lists. I applied this sort to all possible trees, not just the leaves I 'expect' to see at this point. This is because I'll need the merge sort in the next step as well, even for larger trees in the treelist.

Make a huffman tree:

Now using the function makeHuff, I take the first two sorted elements of the list of trees and makes a single node of them, inserting this node into the list again. It then sorts the list again and repeats the process, until I'm left with a single tree, which is my huffman tree. The huffman tree is structured in such a way that the leaves contain a character and its frequency and nodes contain a 4-tuple of left tree, left string, right string and right tree.

Creating an encoding table from the huffman tree:

This part uses the huffman tree and the sorted list of leaves from 4.3 to make a list of tuples with (character,encoding) stored in them. This is done by the function maketable, which calls findit that traverses down the huffman tree for each character and builds up a string of 1's and 0's depending on its location on the tree (process explained in Part 2). 'findit' uses a helper 'member' that checks if one character is present in a string or not.

Converting huffman tree to the given tree format:

Now we convert the huffman tree to the tree format given. This is done with a simple function turntree that just converts a hnode into a node, stripping it of the strings in it, and a hleaf into a leaf, getting rid of the frequencies.

Iterating through the string and replacing chars with their respective encoding:

The main encode function combines the above parts, returning a tuple of the tree along with the encoding. The encoding is made through the function encode' which runs through the string and uses the helper findCode to find the encoding for each character from the list.

To Decode:

Decode calls a helper to do all its work. The helper, decodeHelper, accumulates a string by getting each character from the string by calling yet another helper. So each character is picked up by decodeHelper' and decodeHelper crosses the call between characters, accumulating the decoded string.

Function Recurrences and Complexity Analysis:

Set 1: Making a Frequency Table

Recurrence for : addin

n is the length of the frequency list

Worst case is when the last character in the freqlist is the one to be added to

$$W_{addin}(n) = \begin{cases} k_0 & \text{if } n = 1 \\ k_1 + W_{addin}(n-1) & o.w \end{cases}$$

$$W_{addin}(n) = k_0 + k_1 * n$$

$$W_{addin}(n) \in O(n)$$

$$S_{addin}(n) = \begin{cases} k_0 & \text{if } n = 1 \\ k_1 + S_{addin}(n-1) & o.w \end{cases}$$

$$S_{addin}(n) = k_0 + k_1 * n$$

$$S_{addin}(n) \in O(n)$$

Recurrence for : isin

n is the length of the frequency list:

Worst case is when isin = false:

$$W_{isin}(n) = \begin{cases} k_0 & \text{if } n = 0 \\ k_1 + W_{isin}(n-1) & o.w \end{cases}$$

$$W_{isin}(n) = k_0 + k_1 * n$$

$$W_{isin}(n) \in O(n)$$

$$S_{isin}(n) = \begin{cases} k_0 & \text{if } n = 0 \\ k_1 + S_{isin}(n-1) & o.w \end{cases}$$

$$S_{isin}(n) = k_0 + k_1 * n$$

$$S_{isin}(n) \in O(n)$$

Recurrence for : getFreq

s is the length of the input string, (n is capped at 127 so we treat it as a constant:

$$W_{getFreq}(s) = \begin{cases} k_0 & \text{if } s = 0 \\ k_1 + W_{isin}(n) + W_{getFreq}(s-1) + W_{addin}(n) & o.w \end{cases}$$

$$W_{getFreq}(s) = k_0 + 2 * k_1 * n + k_2 * s$$

$$W_{getFreq}(s) \in O(s)$$

$$S_{getFreq}(s) = \begin{cases} k_0 & \text{if } s = 0 \\ k_1 + S_{isin}(n) + S_{getFreq}(s-1) + S_{addin}(n) & o.w \end{cases}$$

$$S_{getFreq}(s) = k_0 + 2 * k_1 * n + k_2 * s$$

$$S_{getFreq}(s) \in O(s)$$

Set 2: Sorting the list of frequencies

Recurrence for : merge

n is the length of the frequency list

$$W_{merge}(n) = \begin{cases} k_0 & \text{if } n = 1 \\ k_1 + W_{merge}(n-1) & o.w \end{cases}$$

$$W_{merge}(n) = k_0 + k_1 * n$$

$$W_{merge}(n) \in O(n)$$

$$S_{merge}(n) = \begin{cases} k_0 & \text{if } n = 1 \\ k_1 + S_{merge}(n-1) & o.w \end{cases}$$

$$S_{merge}(n) = k_0 + k_1 * n$$

$$S_{merge}(n) \in O(n)$$

Recurrence for : split

n is the length of the frequency list:

$$W_{split}(n) = \begin{cases} k_0 & \text{if } n = 0 \\ k_1 + W_{split}(n-2) & o.w \end{cases}$$

$$W_{split}(n) = k_0 + k_1 * n/2$$

$$W_{split}(n) \in O(n)$$

$$S_{split}(n) = \begin{cases} k_0 & \text{if } n = 0 \\ k_1 + S_{split}(n-2) & o.w \end{cases}$$

$$S_{split}(n) = k_0 + k_1 * n/2$$

$$S_{split}(n) \in O(n)$$

Recurrence for work of : msort

$$W_{msort}(n) = \begin{cases} k_0 & \text{if } n = 0 \\ k_1 & \text{if } n = 1 \\ k_1 + W_{split}(n) + 2 * W_{msort}(n/2) + W_{merge}(n) & o.w \end{cases}$$

$$W_{msort}(n) = k_3 * n * (logn) + (k_2 - k_3) * n + (k - k_2)$$

$$W_{msort}(n) \in O(n * (logn))$$

$$S_{msort}(n) = \begin{cases} k_0 & \text{if } n = 0 \\ k_1 & \text{if } n = 1 \\ k_1 + W_{split}(n) + Max(W_{msort}(n/2), W_{msort}(n/2)) + W_{merge}(n) & o.w \end{cases}$$

$$S_{msort}(n) = k_2 * (logn) - k_2 + k_3 * n$$

$$W_{msort}(n) \in O(n)$$

Set 3: Building a Huffman Tree

Recurrence for : makeHuff

$$W_{cons} = k_1$$

$$W_{\hat{}} = k_2$$

$$W_{makehuff}(n) = \begin{cases} k_0 & \text{if } n = 1 \\ k_0 + k_1 + k_2 + W_{makehuff}(n-1) + W_{msort}(n) & \text{if there's a node involved} \\ k_0 + k_1 + k_2 + W_{makehuff}(n-1) + W_{msort}(n) & o.w \end{cases}$$

$$W_{makehuff}(n) = n * k_0 + n * (k_1 + k_2) + n * k * (n * logn)$$

$$W_{makehuff} \in O(n^2 * logn)$$

$$S_{cons} = k_1$$

$$S_{\hat{}} = k_2$$

$$S_{makehuff}(n) = \begin{cases} k_0 & \text{if } n = 1 \\ k_0 + k_1 + k_2 + S_{makehuff}(n-1) + S_{msort}(n) & \text{if there's a node involved} \\ k_0 + k_1 + k_2 + S_{makehuff}(n-1) + S_{msort}(n) & o.w \end{cases}$$

$$S_{makehuff}(n) = n * k_0 + n * (k_1 + k_2) + n * k * n$$

$$S_{makehuff} \in O(n^2)$$

Set 4: Making a (char,encoding) table

Recurrence for : member

Here, n <=the number of unique characters, So ultimately this is a constant time function.

$$W_{sub}(n) = k_1$$

$$W_{extract}(n) = k * n$$

$$W_{member}(n) = \begin{cases} k_0 & \text{if } n = 0 \\ k_0 + 2 * k_1 + W_{member}(n-1) * k * n & o.w \end{cases}$$

$$W_{member}(n) = n * k_0 + n * k_1 + n^2 * k$$

$$W_{member} \in O(n^2)$$

$$S_{sub}(n) = k_1$$

$$S_{extract}(n) = k * n$$

$$S_{member}(n) = \begin{cases} k_0 & \text{if } n = 0 \\ k_0 + 2 * k_1 + S_{member}(n-1) * k * n & o.w \end{cases}$$

$$S_{member}(n) = n * k_0 + n * k_1 + n^2 * k$$

$$S_{member} \in O(n^2)$$

Recurrence for : findit

N here is the height of the htree, which is worst case the number of unique characters, which is capped at 127.

$$W_{\hat{}}(n) = k_1$$

$$W_{findit}(n) = \begin{cases} k_0 & \text{if } n = 0 \\ k_0 + W_{member}(n) + k_1 + W_{findit}(n-1) & o.w \end{cases}$$

$$W_{findit}(n) = n * k_0 + n * k_1 + n^2 * k + n^2$$

$$W_{findit} in O(n^2)$$

$$S_{\hat{}}(n) = k_1$$

$$S_{findit}(n) = \begin{cases} k_0 & \text{if } n = 0 \\ k_0 + S_{member}(n) + k_1 + S_{findit}(n-1) & o.w \end{cases}$$

$$S_{findit}(n) = n * k_0 + n * k_1 + n^2 * k + n^2$$

$$S_{findit} \in O(n^2)$$

Recurrence for : maketable

N here is the length of the list of frequencies, which is the number of unique characters, which is capped at 127.

$$W_{cons}(n) = k_1$$

$$W_{maketable}(n) = \begin{cases} k_0 & \text{if } n = 0 \\ k_0 + W_{findit}(n) + k_1 + W_{maketable}(n-1) & o.w \end{cases}$$

$$W_{maketable}(n) = n * k_0 + n * k_1 + n^2 * k + n$$

$$W_{findit} \in O(n^2)$$

$$S_{cons}(n) = k_1$$

$$S_{maketable}(n) = \begin{cases} k_0 & \text{if } n = 0 \\ k_0 + S_{findit}(n) + k_1 + S_{maketable}(n-1) & o.w \end{cases}$$

$$S_{maketable}(n) = n * k_0 + n * k_1 + n^2 * k + n$$

$$S_{findit} \in O(n^2)$$

Set 5: Converting a htree to a tree

Recurrence for : turntree

Here we use 'd' for the tree's height, and calculate work and span of it. In the Worst case, d is perfectly balanced. Hence, n = log d, n is the length of the frequency list.

$$W_{turntree}(d) = \begin{cases} k_0 & \text{if } d = 0 \\ k_1 + 2 * W_{turntree}(d-1) & o.w \end{cases}$$

$$W_{turntree}(d) = 2^d * k_1 + k_0 = n * k_1 + k_0$$

$$W_{turntree} \in O(n)$$

$$S_{turntree}(d) = \begin{cases} k_0 & \text{if } d = 0 \\ k_1 + S_{turntree}(d-1) & o.w \end{cases}$$

$$S_{turntree}(d) = d * k_1 + k_0 = log(n) * k_1 + k_0$$

$$S_{turntree} \in O(log(n))$$

Set 6: Replacing characters with encoding

Recurrence for : findcode

n is the length of the frequency list.

$$W_{findcode}(n) = \begin{cases} k_0 & \text{if } n = 0 \\ k_1 + W_{findcode}(n-1) & o.w \end{cases}$$

$$W_{findcode}(n) = k_0 + n * k_1$$

$$W_{findcode} \in O(n)$$

$$S_{findcode}(n) = \begin{cases} k_0 & \text{if } d = 0 \\ k_1 + S_{findcode}(n-1) & o.w \end{cases}$$

$$S_{findcode}(n) = n * k_1 + k_0$$

$$S_{findcode} \in O(n)$$

Recurrence for : encode'

s is the length of the input string.

$$W_{encode'}(s) = \begin{cases} k_0 & \text{if } s = 0 \\ k_1 + W_{findcode}(n) + W_{extract}(s-1) + W_{\hat{}} + W_{encode'}(s-1) & o.w \end{cases}$$

Since n is a constant, (capped at 127), we assume it's a constant

$$W_{encode'}(s) = k_0 + s * k + s^2 k_2$$

$$W_{encode'} \in O(s^2)$$

$$S_{encode'}(s) = \begin{cases} k_0 & \text{if } s = 0 \\ k_1 + S_{findcode}(n) + S_{extract}(s-1) + S_{\hat{}} + S_{encode'}(s-1) & o.w \end{cases}$$

Since n is a constant, (capped at 127), we assume it's a constant

$$S_{encode'}(s) = k_0 + s * k + s^2 k_2$$

$$S_{encode'} \in O(s^2)$$

Recurrence of encode

Let s be the input string. We calculate the recurrence on the length of s.

$$W_{encode}(s) = k_1 + W_{getFreq}(s) + W_{msort}(n) + W_{makeHuff}(n) + W_{maketable}(n) + W_{turntree}(n) + W_{encode'}(s)$$

Since n is a constant, (capped at 127), we assume it's a constant Replacing the complexities of all the functions,

$$W_{encode}(s) = k_1 + k * s + k * s^2$$
$$W_{encode} \in O(s^2)$$

$$S_{encode}(s) = k_1 + S_{getFreq}(s) + S_{msort}(n) + S_{makeHuff}(n) + S_{maketable}(n) + S_{turntree}(n) + S_{encode'}(s)$$

Since n is a constant, (capped at 127), we assume it's a constant

$$S_{encode}(s) = k_1 + k * s + k * s^2$$
$$S_{encode} \in O(s^2)$$

Recurrence of decode

Recurrence for : decodeHelp'

x is the length of the encoding of a single character, and b is the length of the whole binary string.

$$W_{decodeHelp'}(x) = \begin{cases} k_0 & \text{if } x = 0 \\ k_1 + W_{decodeHelp'}(x-1) + W_{extract}(b) & o.w \end{cases}$$

$$W_{decodeHelp'}(x) = k_0 + x * k_1 + x * b$$
$$W_{decodeHelp'} \in O(b)$$

$$S_{decodeHelp'}(x) = \begin{cases} k_0 & \text{if } x = 0 \\ k_1 + S_{decodeHelp'}(x-1) + S_{extract}(b) & o.w \end{cases}$$

$$S_{decodeHelp'}(x) = k_0 + x * k_1 + x * b$$
$$S_{decodeHelp'} \in O(b)$$

Recurrence for : decodeHelp

b is the length of the whole binary string.

$$W_{decodeHelp}(b) = \begin{cases} k_0 & \text{if } b = 0 \\ k_1 + W_{decodeHelp'}(b) + W_{decodeHelp}(b-1) & o.w \end{cases}$$

$$W_{decodeHelp}(b) = k_0 + b * k_1 + b^2 * k$$
$$W_{decodeHelp} \in O(b^2)$$

$$S_{decodeHelp}(b) = \begin{cases} k_0 & \text{if } b = 0 \\ k_1 + S_{decodeHelp'}(b) + S_{decodeHelp}(b-1) & o.w \end{cases}$$

$$S_{decodeHelp}(b) = k_0 + b * k_1 + b^2 * k$$
$$S_{decodeHelp} \in O(b^2)$$

The recurrence for decode is equivalent to the recurrence of decodeHelp, since all decode does is call decodeHelp Therefore,

$$W_{decode} \in O(b^2)$$
$$S_{decode} \in O(b^2)$$

Proof:

Proof 1:

Lemma 1. ∀ *characters 'c' in the encode string, decodeHelp'(turntree t, findit(c,t),"") = c*

Proof. Proof by structural induction on hNode(tl,tr)

- **Base case:** t = hNode(hLeaf a,_,hLeaf a')

  To show: decodeHelp'(turntree t, findit(c,t),"") = c where c=a or c=a'
  Case 1: $c = a$

$$\begin{aligned} findit(a, hNode(hLeafa, \_, hLeafa')) &= "0" \text{(From the implementation of findit)} \\ turntreet &= Node(Leafa, Leafa') \\ decodeHelp'(Node(Leafa, Leafa'), "0", "") &= decodeHelp'(Leafa, "", "") = a \\ &= RHS \end{aligned}$$

  Case 2: $c = a'$

$$\begin{aligned} findit(a', hNode(hLeafa, \_, hLeafa')) &= "1" \text{(From the implementation of findit)} \\ turntreet &= Node(Leafa, Leafa') \\ decodeHelp'(Node(Leafa, Leafa'), "1", "") &= decodeHelp'(Leafa', "", "") = a' \\ &= RHS \end{aligned}$$

- **Inductive case:** ∀ characters 'c' $\in t = hNode(tl, \_, tr)$,

  To show:

$$decodeHelp'(turntree(t), findit(c, t), "") = c$$

  IH: ∀ characters 'c' $\in tl$ or tr,

$$decodeHelp'(turntree(tl), findit(c, tl), "") = c \qquad \text{if } c \in tl$$

$$decodeHelp'(turntree(tr), findit(c, tr), "") = c \qquad \text{if } c \in tr$$

$$turntree(t) = Node(turntree(tl), turntail(tr)) \text{ (Def. of turntree)}$$

  Now for the function findit, there are two cases- c is in tl and c is in tr
  Case 1: $c \in tl$

$$findit(a, hNode(tl, \_, tr)) = "0"\hat{}\, findit(c, tl) \text{(From the implementation of findit)}$$

  For the sake of space, we call turntree 'tt'

$$\begin{aligned} &decodeHelp'(Node(tt(tl), tt(tr)), "0"\hat{}\, findit(c, tl), "") \\ &\qquad = decodeHelp'(tt(tl), "0"\hat{}\, findit(c, tl), "") \\ &\qquad = a(FromtheIH) \\ &\qquad = RHS \end{aligned}$$

Case 2: $c \in tr$

$$findit(a, hNode(tl, \_, tr)) = "1"\text{\^{}}findit(c, tl) \text{(From the implementation of findit)}$$

For the sake of space, we call turntree 'tt'

$$\begin{aligned} &decodeHelp'(Node(tt(tl), tt(tr)), "1"\text{\^{}}findit(c, tl), "") \\ &\qquad = decodeHelp'(tt(tr), "1"\text{\^{}}findit(c, tl), "") \\ &\qquad = a \\ &\qquad = RHS \end{aligned}$$

Proving both these cases proves that for any character, the property holds.Hence Proved.

Q. E. D

□

Proof 2:

Lemma 2. $findcode(c, tab) = findit(c, t)$ *where tab is make_table(t)*

Proof. Proof by argument
We know that maketable calls findit on each char c of the tree t, and adds that into tab as (c,findit(c)). Therefore, tab contains findit(c) for every c.
findcode simply performs a linear search and finds the character's encoding out of tab.
Hence, findcode(c,tab) is equivalent to findit(c,t)

Q. E. D

□

Proof 3:

Lemma 3.

$$encode'(s) = findit\_h(c1, t)\text{\^{}}...\text{\^{}}findit\_h(cn, t), \;\; where\; [c_1, c_2, ..., c_n] = explode(s)$$

Proof. Proof by induction on $s'\text{\^{}}c_{n+1}$:

- **Base case:** s = c ^ c'

  To show: $encode'(s) = findit(c)\text{\^{}}findit(c')$

$$\begin{aligned} encode'(s) &= encode'(c\text{\^{}}c') \\ &= findcode(c, tab)\text{\^{}}encode'(c') \text{ (From the interpretation of encode')} \\ &= findit(c, t)\text{\^{}}findit(c', t)\text{\^{}}\text{""} \text{ (From lemma 2)} \\ &= RHS \end{aligned}$$

- **Inductive case:** $s = s' \hat{} c_{n+1}$

  To show:

$$encode'(s \hat{} c_{n+1}) = findit(c1, t)\hat{} ... \hat{} findit(c_{n+1}, t)$$

IH: $encode'(s) = findit(c1, t)\hat{} ... \hat{} findit(c_{n+1}, t)$

$$\begin{aligned} encode'(s \hat{} c_{n+1}) &= encode(s)\hat{} findcode(c_n + 1) \\ &= encode(s)\hat{} findit(c_n + 1, t) \text{ (Lemma 2)} \\ &= findit(c1, t)\hat{} ... \hat{} findit(c_{n+1}, t)\hat{} findit(c_n + 1, t) \textbf{ (IH)} \end{aligned}$$

Q. E. D

□

Proof 4:

Theorem 1.

$$decode(decodeHelp'(c_1)\hat{} ... \hat{} decodeHelp(c_n)) = s, \text{ where } [c_1, c_2, ..., c_n] = explode(s)$$

Proof. We prove this directly by arguing about decode

Decode works by calling the decodeHelper() function with some accumulators. decodeHelper calls deccodeHelper', which traverses down on convert(t) and finds a character. Therefore, if encode gives a concatenation of findit of each character, and we know from lemma 2 that decodeHelper'(findit(c,t)) = c, then if decodeHelper' is applied on every code one by one, we get the original string back.

Q. E. D

□