Correctness of Huffman Coding

Huffman Code uses a greedy approach to generate prefix code T that minimizes the expected length $B(T)$ to encode a string. Every code in Huffman Algorithm is unique and not prefix of others. This prefix property is evident by the fact code words are the leaves of binary tree. To put it simply Huffman Algorithm generate optimum prefix codes. The cost of any encoding tree T is

$$\boldsymbol{B(T)} = \sum_{x \in C} \boldsymbol{p(x) d_T(x)}$$

Here $\boldsymbol{B(T)}$ is the number of bits required to encode a file. $\boldsymbol{p(x)}$ is the frequency of each character x in alphabet C coming in file and $\boldsymbol{d_T(x)}$ is the depth of character x in the Huffman Tree. According to Huffman Coding we get the two characters with smallest frequencies and combine them and so on. We propose that this method will give us $\boldsymbol{B(T)}$ which will be smallest than the $\boldsymbol{B(T)}$ given by any other methods and we will prove that there is no other way to get smaller $\boldsymbol{B(T)}$ than Huffman Coding.

To prove the correctness we will show that any coding tree constructed by some other method can be converted into Huffman Coding Tree without increasing its cost $\boldsymbol{B(T)}$.

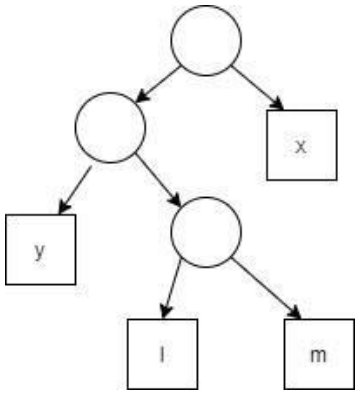
The Claim 1

Consider two characters $\boldsymbol{x}$ and $\boldsymbol{y}$ with smallest frequencies then there is an optimal code tree where these two characters are siblings at the maximum depth in tree.

Proof:

To prove this we will consider a tree **T** which is not a Huffman Coding Tree. Where characters $\boldsymbol{x}$ and $\boldsymbol{y}$ are with smallest frequencies but are not at deepest level of three and characters $\boldsymbol{l}$ and $\boldsymbol{m}$ are at the maximum depth.

T



Since $\boldsymbol{l}$ and $\boldsymbol{m}$ are at deepest level so we know that

$$\boldsymbol{d(m) \geq d(x) \quad \& \quad d(l) \geq d(y)}$$

Here d represents depth of tree so depth of m is greater or equal to depth of x and depth of l is greater or equal to depth of y. Now we can assume that

$$p(m) - p(x) \geq 0$$

&

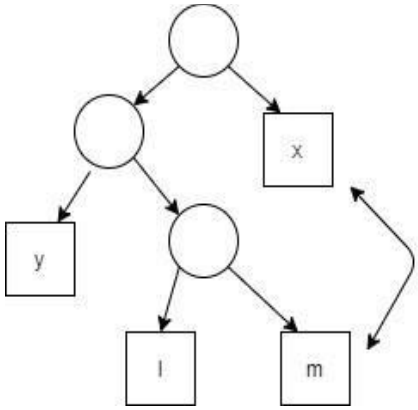
$$d(m) - d(x) \geq 0$$

Thus we can say that

$$(p(m) - p(x)) \cdot (d(m) - d(x)) \geq 0 \quad \text{..... EQU 1}$$

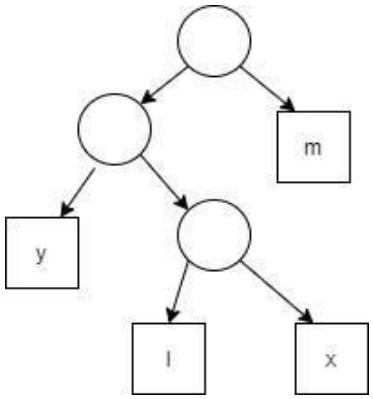
Now if we **swap** the position of x and m

T



We will get a new tree T'

T'



If we calculate the cost of $B(T')$ then

Adding and removing the cost of particular character before and after swap.

$$B(T') = B(T) - p(x)\,d(x) + p(x)\,d(m) - p(m)\,d(m) + p(m)\,d(x)$$

By taking common

$$B(T') = B(T) + p(x)\,[d(x) - d(m)] - p(m)\,[d(m) - d(x)]$$ Now

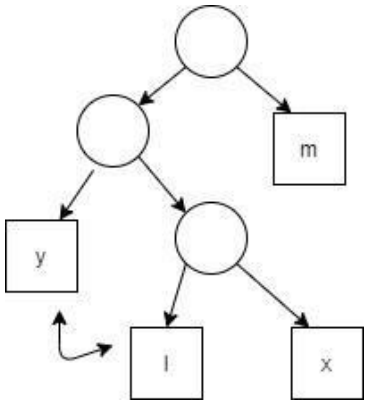
we can see that our cost is less than the previous tree.

$$B(T') = B(T) - (p(m) - p(x)) \cdot (d(m) - d(x)) \leq B(T)$$ Because from EQU 1 $(p(m) - p(x)) \cdot (d(m) - d(x)) \geq 0$

Now we clearly see that the cost of new tree T' will be less than the cost of T which means that our Tree T' **is an optimal tree.**

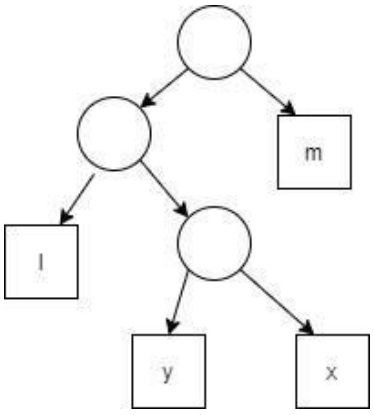
This cost is reduced by swapping one element we can do same for the y and l

T'



We will get a new Tree T'' by swapping l with y

T''



If we calculate $B(T'')$ by same method we will get

$$B(T'') = B(T') - (p(l) - p(y)) \cdot (d(l) - d(y)) \leq B(T')$$

So $B(T'')$ is again less than $B(T')$ which shows that out tree T'' is very much optimal.

Our cost has also been reduced a lot while we did nothing too much complex just simple swapping and good thing is that we have no need to do this in Huffman coding as it always gives Optimal Coding Tree.

Conclusion:

By looking at the diagram of tree T'' it is proved that our Tree is optimal and x and y are smallest frequency siblings at the maximum depth of T''.

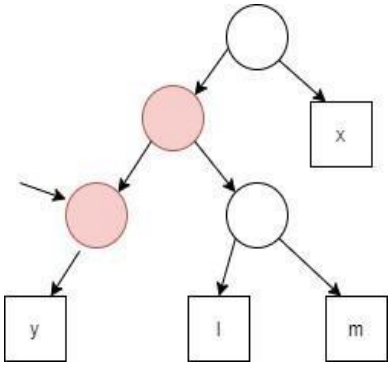
The Claim 2

The Tree for optimal prefix code must be **full** which means that each nodes have exactly two children.

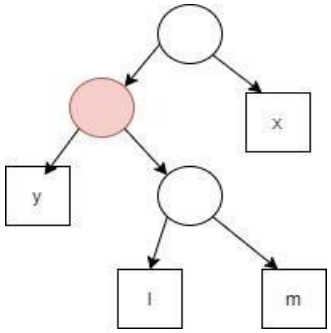
Proof:

It is very easy to prove as the answer lies within our encoding method. When we do Huffman encoding we always take two nodes with smallest frequencies and make a new node their parent therefore each node will have exactly two children.

Moreover consider a tree where an internal node does not have exactly two children.



What we can do is we can simply replace it with its unique children as it will not affect our tree.



Conclusion:

We got a full binary tree as it can be seen above. The best thing about Huffman coding is that we would not need to do this replacement as it will always generate full binary tree.

The Claim 3:

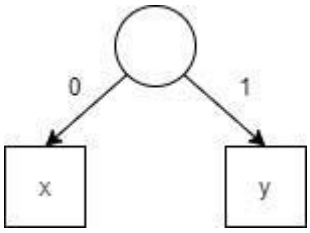
Huffman algorithm Gives Optimal Prefix Code Tree.

Proof:

We need to prove that and the first step that is to combine two smallest frequencies which Huffman algorithm uses is proper to perform. The first step of Huffman algorithm is a **greedy** approach as we choose the smallest frequencies at start and hope for the globally optimal solution which is having Optimal Prefix Code Tree.

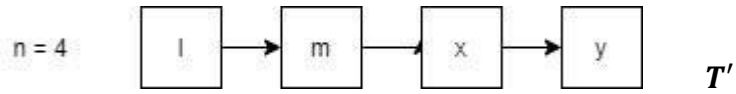
We will prove this by **Induction** on **n** where n is total number of characters.

- For the **base case** if **n = 2** then there are two nodes there tree will be



Which is already optimal as we cannot get better prefix codes than **0** and **1.**

- What we want to show that it is true for exactly **n** characters.

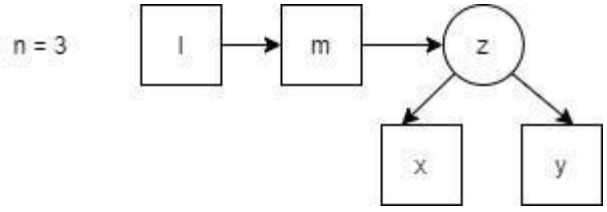


T'

If we have n characters then using previous **claim 1** which states that Consider two characters x and y with smallest frequencies then there is an optimal code tree where these two characters are siblings at the maximum depth in tree.

- Now Remove x and y and replace them with a new character z.

T



Frequency **z** will be the combination of characters x and y frequencies.

$$p(z) = p(x) + p(y)$$

Thus **n – 1** character remains. Consider tree with n - 1 characters as T.

- Replace **z** again with x and y and consider this tree of **n** characters as T'. Here we have undone the previous step.

- The **Cost** of Tree T' will be

We are removing **z** node from **n -1** and replacing it with x and y. Depth is $d(z) + 1$ because x and y are the child of **z** so are at one level deep.

$$B(T') = B(T) - p(z)d(z) + p(x)[d(z) + 1] + p(y)[d(z) + 1]$$

$$B(T') = B(T) - (p(x) + p(y))d(z) + (p(x) + p(y))[d(z) + 1]$$

$$B(T') = B(T) - (p(x) + p(y))[d(z) + 1 - d(z)] \qquad B(T') = B(T) - p(x) + p(y)$$

- The cost of final Tree T' changes but as we can see that there is no depth in equation of $B(T')$ so the change doesn't depend upon the structure of tree T (Tree for n – 1 characters)

- So to minimize this cost of final tree ***T'*** we need to build tree ***T*** on **n-1** characters optimally.
- By induction this is exactly what our Huffman algorithm does so the final tree is optimal.

Conclusion:

By induction we proved that our final tree will be optimal.

TIME COMPLEXITY ANALYSIS

Huffman Encoding PseudoCode

```
Procedure HuffmanEncoding (PQ)        //PQ is the priority Queue with Letters
1.      S = PQ.size                   //  and Frequencies set it will be custom built
2.      while S is not equal to 1 do
3.      N = new Node ()
4.      N.left = PQ.pop
5.      N.right = PQ.pop
6.      N.frequency = N.left.frequency + N.right.frequency
7.      PQ.Insert (N)
8.      S = PQ.size
9.      end while
10.     return  PQ.Top
```

To analyze the running time of Huffman's algorithm, first: it is noted that algorithm implemented using priority queue.

1. In this step, there is a need of calculations of the frequencies for each character. For this step we need to read the entire data source once which will take time of **O(n)**.

2. Using priority queue, an encoded binary tree is constructed. Insertion and deletion operations are being implemented.

- Insertion requires **O (lg n)** as it requires to traverse the tree (**max height from root to the leaf node**). Hence, it requires height of the tree cost, which is **lg(n)**.
- Similarly, in deletion operation it is required to traverse the tree. Hence, cost of deletion is the height of tree as well which is **lg(n)**.

Overall, each iteration on priority queue requires time **O (lg n)**.

So, by this tree corresponding to a prefix code, the number of bits required to encode a file can easily be computed. For each character say **x** in the alphabet **C,** consider attribute **x.freq** denote the frequency of **x** in the file and let $\mathbf{d_T(x)}$ denote the depth of x's leaf in the tree. Note that is also the length of the codeword for character **x**. Thus, the number of bits required to encode a file is thus;

$\mathbf{B(T)=\sum_{x\in C} x.freq.\ d_T(x)}$ which is defined as the cost of the tree T, where **x** belongs **C**.

Keeping in view, all this underlying model its time complexity can be calculated as follows from the above algorithm.

Step 1:

- **S** is initialized with Priority Queue, taking **O (lg n)** cost in its operations (**insertion and deletion**).

Step 2:

- In the lines **2** to **9**, the **while** loop executes **n-1** times. Plus, as each priority queue operation requires **O (lg n)**, the loop contributes **O (n lg n)** to the running time. There are **n** iterations, one for each data point so the overall running time complexity **O (n log n)**.

SUMMARIZATION:

In our algorithm the first file for compression take **O (n log n)** running time but for the subsequent files there will no need of calculating frequencies also not a need to maintain the priority queue. Thus, each character will be read from the data source and converted into binary sequencies. This will take **linear time** to code the data.

Space Complexity:

The conventional algorithm requires space to maintain the priority queue for each data file. In our case the space is fixed and stored priority queue does not change.