"Make large enough for 7 chars"

Can easily decode: 

011010100

sit

Number of bits in terms of tree?

\[ \#\text{bits} = \text{tree "cost"} = \sum_{c \in A} d_c \times f_c \]

Cheaper tree?

\[ \Rightarrow \text{Can use variable length encoding - code lengths differ} \]

Yes!

nl

\[ \Rightarrow \]

nl

nl

nl

nl

new cost = 173

\[ \Rightarrow \text{Notice: not a full binary tree} \]
How to use variable length encoding to get cheaper tree?

Assign less freq chars longer codes
e more " " shorter "

Not clear how.

What should be shortest/longest?
What if there is a tie?
Less obvious problem

let's try: e:0 sp:1 i:00

Problem: Decode 000 000 000
        e i e
        000 000 000

ambiguous encoding!

We want: unambiguous encoding - unique decoding for every encoded string.
characters at leaves only

How to ensure unambiguous encoding? - characters at leaves only

New goal: Find cheapest tree that yields unambiguous encoding.

Huffman's tree - start with single-node tree for each char, cost = freq
(continuously merge 2 cheapest trees.)
Huffman's Tree

Notice:
- unambiguous
- less freq. chars have longer codes
- more " " " " shorter

Decode: 100001
      i t
Huffman's Algorithm (term project as grad student at MIT, 1954)

Given: A of size n, $f_c$ for all $c \in A$
1. For every char $c$, create a single-node tree with cost $f_c$
2. Do $n-1$ times:
3. * Merge 2 trees with lowest costs.
4. * Cost of new tree is sum of costs of 2 subtrees.

How many merges? Every merge reduces # of trees by 1. Start with $n$ trees, want 1 final tree.

\[
\begin{align*}
&n \quad \text{n}
&n-1 \quad \text{n-1 merges}
&n-2 \quad \\
&\vdots
&1
\end{align*}
\]
Run Time first?

<table>
<thead>
<tr>
<th>char</th>
<th>freq</th>
<th>code</th>
<th># bits</th>
<th>Huff code</th>
<th># bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>10</td>
<td>000</td>
<td>30</td>
<td>001</td>
<td>30</td>
</tr>
<tr>
<td>e</td>
<td>15</td>
<td>001</td>
<td>45</td>
<td>01</td>
<td>30</td>
</tr>
<tr>
<td>i</td>
<td>12</td>
<td>010</td>
<td>36</td>
<td>10</td>
<td>24</td>
</tr>
<tr>
<td>s</td>
<td>3</td>
<td>011</td>
<td>9</td>
<td>00001</td>
<td>15</td>
</tr>
<tr>
<td>t</td>
<td>4</td>
<td>100</td>
<td>12</td>
<td>0001</td>
<td>16</td>
</tr>
<tr>
<td>sp</td>
<td>13</td>
<td>101</td>
<td>39</td>
<td>11</td>
<td>26</td>
</tr>
<tr>
<td>nl</td>
<td>1</td>
<td>110</td>
<td>3</td>
<td>00000</td>
<td>5</td>
</tr>
</tbody>
</table>

\[ \text{Total} = 174 \]

\[ \text{Total Huffman code} = 146 \]

On Alice in Wonderland: 15% reduction

Run Time: For \( n \) char types, implemented with

\[ \text{min Binary Heap: } O(n \log n) \]

Step 1. \( O(n) \)

Step 2. \( n-1 \) times.

Step 3. \( 2 \times \text{deleteMin()} + 1 \text{ insert} = 2 \times O(\log n) + O(\log n) \)

\[ \Rightarrow \text{Total: } O(n \log n) \]

Optimality: Huffman proved day before term project due

1. Final tree will be full - every non-leaf node has exactly
   2 children

2. unambiguous encoding - all chars at leaves

3. more frequent chars have longer codes

less " " shorter " 
Drawbacks:

1. Must scan entire file first to find frequencies
   - Avoid by using statistics (of English language) to estimate frequencies.
2. Does not take into account frequencies of strings, e.g. "the"