Huffman's Alg. (term project as grad student at MIT, 1954)

Given A of size n, \( f_c \) for \( \forall c \in A \).

1. For every char \( c \) with \( \text{freq} \ f_c \), create a single-node tree labeled with \( c, f_c \).

2. Do \( n-1 \) times:
   - Merge 2 trees with the lowest costs.
   - (Cost of new tree is sum of costs of 2 subtrees)

Start with \( n \) trees,

want 1 final tree.

Optimality (Won't do formal proof).
-Proved day before project was due.

1) unambiguous encoding - all characters at leaves.
2) cheapest tree -
   - full binary tree- always merging either 2 leaves , 2 full binary trees, or 1 leaf and 1 full tree
   - min cost - less freq chars have longer lengths more """" shorter """"
Runtime: For n char types.

Maintain trees in min binary heap ordered by cost.

Step 1: BuildHeap() : O(n).
Step 2: 2(n-1) deleteMin()’s: ⇒ O(nlogn) deleteMin is O(logn).
        n-1 insert()’s: => O(nlogn) insert is.

Total: O(nlogn).

One drawback: Have to read the entire file first to get a count of chars, freqs.

How to avoid this? Use statistics of English language to get an estimate of chars, freqs.
<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>1</td>
<td>26</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
<td>110</td>
<td>3</td>
<td>00000</td>
<td>5</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td><strong>174</strong></td>
<td></td>
<td><strong>146</strong></td>
</tr>
</tbody>
</table>
Lempel-Ziv

Drawbacks of Huffman's?
- must read entire file
- encodes only single characters (not sequence of characters)

ex: AABBAAAA

Huffman: A = 0, B = 1
L = 0001000

Better: AAA = 0, B = 1 → 010

Note w/ Huffman's we did: char → binary
In practice char → int (ASCII) → binary

Lempel-Ziv makes this assumption: chars as ASCII int

A = 65, B = 66, ...

Eventually, ints are stored as binary

Lempel-Ziv encodes strings instead of chars

Idea: Use a sliding window to encode string while reading the string
- Read up to the longest string that already has a code
- When see new string (not already in table), encode it and add to table

Use "sliding window" so this can be done while reading the string

ABBA

- already has a code, so extend window size to 3

A already has code

Extend window to AB, BB, BA (add new codes)

AB already has code

Extend to ABB (add new code)

Keep str: longest string so far that already has code

c: next char

ex: A B B A B B B A B B A

<table>
<thead>
<tr>
<th>newstr</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td>256</td>
</tr>
<tr>
<td>BB</td>
<td>257</td>
</tr>
<tr>
<td>BA</td>
<td>258</td>
</tr>
<tr>
<td>ABB</td>
<td>259</td>
</tr>
</tbody>
</table>

output str when str+c not found

BBA 260

AABB 261
Lempel-Ziv (msg, dictionary)

str = msg.get_next_char()

while (there are still chars in msg):
    c = msg.get_next_char()
    if (dictionary.contains(str + c))
        str = str + c // build a longer string
        // (extend window length)
    else
        output str.code // longest string with a code
        dictionary.add(str + c)
        str = c // slide the window

output str.code

Question: How to quickly check if a string already has a code? => Make the table a hash table (dictionary).

How good is LZ?

No compression: 8 bits/11 chars = 88 bits
Compressed w/ LZ: (8 bits) x 4 + (9 bits) x 3 = 59 bits
\[ \Rightarrow 33\% \text{ reduction.} \]

Effectiveness: on Alice in Wonderland
Huffman's: 15\%
Lempel-Ziv: 50\%

RunTime: n-length of msg, \(1A1 = \text{table size}\)
\(O(n)\) if contains is \(O(1)\)
\(O(1A1n)\) if contains searches table (unlikely)

Another example:

<table>
<thead>
<tr>
<th>A B R A C A D A B R A (C = 67, \ D = 68, \ R = 114)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{newstr}</td>
</tr>
<tr>
<td>---------------</td>
</tr>
<tr>
<td>AB</td>
</tr>
<tr>
<td>BR</td>
</tr>
<tr>
<td>RA</td>
</tr>
<tr>
<td>AC</td>
</tr>
<tr>
<td>CA</td>
</tr>
<tr>
<td>AD</td>
</tr>
</tbody>
</table>

encoding: 65 66 114 65 67 65  DA 262  ABR 263

68 256 258

No compression: 8(11) = 88 bits
compressed: 8(7) + 2(9) = 74 bits