UNIT 7B
Data Representation: Compression

Fixed-Width Encoding

• In a fixed-width encoding scheme, each character is given a binary code with the same number of bits.
  – Example:
    Standard ASCII is a fixed width encoding scheme, where each character is encoded with 7 bits.
    This gives us $2^7 = 128$ different codes for characters.
Fixed-Width Encoding

- Given a character set with \( n \) characters, what is the minimum number of bits needed for a fixed-width encoding of these characters?
  - Since a fixed width of \( k \) bits gives us \( n \) unique codes to use for characters, where \( n = 2^k \).
  - So given \( n \) characters, the number of bits needed is given by \( k = \lceil \log_2 n \rceil \). (We use the ceiling function since \( \log_2 n \) may not be an integer.)
  - Example: To encode just the alphabet A-Z using a fixed-width encoding, we would need \( \lceil \log_2 26 \rceil = 5 \) bits: e.g. A => 00000, B => 00001, C => 00010, ..., Z => 11001.

Using Fixed-Width Encoding

- If we have a fixed-width encoding scheme using \( n \) bits for a character set and we want to transmit or store a file with \( m \) characters, we would need \( mn \) bits to store the entire file.
- Can we do better?
  - If we assign fewer bits to more frequent characters, and more bits to less frequent characters, then the overall length of the message might be shorter.
Huffman Coding

- We can use an encoding scheme named after David A. Huffman to compress our text without losing any information.
- Based on the idea that some characters occur more frequently than others.
- Huffman codes are not fixed-width.

The Hawaiian Alphabet

- The Hawaiian alphabet consists of 13 characters.
  - ‘ ’ is the okina which sometimes occurs between vowels (e.g. KAMA’ AINA)
- The table to the right shows each character along with its relative frequency in Hawaiian words.

<table>
<thead>
<tr>
<th>Character</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>’</td>
<td>0.068</td>
</tr>
<tr>
<td>A</td>
<td>0.262</td>
</tr>
<tr>
<td>E</td>
<td>0.072</td>
</tr>
<tr>
<td>H</td>
<td>0.045</td>
</tr>
<tr>
<td>I</td>
<td>0.084</td>
</tr>
<tr>
<td>K</td>
<td>0.106</td>
</tr>
<tr>
<td>L</td>
<td>0.044</td>
</tr>
<tr>
<td>M</td>
<td>0.032</td>
</tr>
<tr>
<td>N</td>
<td>0.083</td>
</tr>
<tr>
<td>O</td>
<td>0.106</td>
</tr>
<tr>
<td>P</td>
<td>0.030</td>
</tr>
<tr>
<td>U</td>
<td>0.059</td>
</tr>
<tr>
<td>W</td>
<td>0.009</td>
</tr>
</tbody>
</table>
The Huffman Tree

- We use a tree structure to develop the unique binary code for each letter.
- Start with each letter/frequency as its own node:

```
<table>
<thead>
<tr>
<th>Letter</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.262</td>
</tr>
<tr>
<td>E</td>
<td>0.072</td>
</tr>
<tr>
<td>H</td>
<td>0.045</td>
</tr>
<tr>
<td>I</td>
<td>0.084</td>
</tr>
<tr>
<td>K</td>
<td>0.106</td>
</tr>
<tr>
<td>L</td>
<td>0.044</td>
</tr>
<tr>
<td>M</td>
<td>0.032</td>
</tr>
<tr>
<td>N</td>
<td>0.083</td>
</tr>
<tr>
<td>O</td>
<td>0.106</td>
</tr>
<tr>
<td>P</td>
<td>0.030</td>
</tr>
<tr>
<td>U</td>
<td>0.059</td>
</tr>
<tr>
<td>W</td>
<td>0.009</td>
</tr>
</tbody>
</table>
```

The Huffman Tree

- Combine lowest two frequency nodes into a tree with a new parent with the sum of their frequencies.
The Huffman Tree

- Combine lowest two frequency nodes (including the new node we just created) into a tree with a new parent with the sum of their frequencies.
The Huffman Tree

- Combine lowest two frequency nodes (including the new node we just created) into a tree with a new parent with the sum of their frequencies...
• Repeat until you have one tree with all nodes linked in.
Label all left branches with 0 and all right branches with 1.

The binary code for each character is obtained by following the path from the root to the character.
Examples:
H => 0001
A => 10
P => 110011

Fixed Width vs. Huffman Coding

```
A     A     10
0001  001  1101

H     H     0001
0011  000  0001

I     I     1111
0100  001  1001

K     K     001
0101  011  1101

L     L     0000
0110  000  0000

M     M     11000
0111  010  0110

N     N     1110
1000  111  1110

O     O     010
1001  010  0110

P     P     110011
1010  110  10011

U     U     0110
1011  001  0011

W     W     110010
1100  110  110010
```

ALOHA

Fixed Width:
001011010010011001
20 bits

Huffman Code:
100000010000111
15 bits
Priority Queues

NOTE: For this unit, you will need RubyLabs set up and you will need to include BitLab (see p. 167)

• A priority queue (PQ) is like an array that is sorted.
  
  ```ruby
  pq = PriorityQueue.new
  => []
  ```

• To add element into the priority queue in its correct position, we use the `<<` operator:
  
  ```ruby
  pq << "peach"
pq << "apple"
pq << "banana"
=> ["apple", "banana", "peach"]
  ```

Priority Queues (cont’d)

• To remove the first element from the priority queue, we will use the `shift` method:
  
  ```ruby
  fruit1 = pq.shift
  => "apple"
pq
  => ["banana", "peach"]
  ```

  ```ruby
  fruit2 = pq.shift
  => "banana"
pq
  => ["peach"]
  ```
Tree Nodes

- We can store all of the node data into a 2-dimensional array:
  ```python
table = [ ["'", 0.068], ["A", 0.262],
            ["E", 0.072], ["H", 0.045], ["I", 0.084],
            ["K", 0.106], ["L", 0.044], ["M", 0.032],
            ["N", 0.083], ["O", 0.106], ["P", 0.030],
            ["U", 0.059], ["W", 0.009] ]
```
- A tree node consists of two values, the character and its frequency. Making one of the tree nodes:
  ```python
char = table[2].first  # "E"
freq = table[2].last   # 0.072
node = Node.new(char, freq)
```

Building a PQ of Single Nodes

```python
def make_pq(table):
    pq = PriorityQueue.new
    for item in table:
        char = item.first
        freq = item.last
        node = Node.new(char, freq)
        pq << node
    return pq
```

Remember: each item in the table is a 2-element array with a character and a frequency.
Building our Priority Queue

\[
pq = \text{make\_pq}(\text{table})
\]
\[
=> [( W: 0.009 ), ( P: 0.030 ),
     ( M: 0.032 ), ( L: 0.044 ),
     ( H: 0.045 ), ( U: 0.059 ),
     ( ': 0.068 ), ( E: 0.072 ),
     ( N: 0.083 ), ( I: 0.084 ),
     ( K: 0.106 ), ( O: 0.106 ),
     ( A: 0.262 )]
\]

This is our priority queue showing the 13 nodes in sorted order based on frequency.

Building a Huffman Tree

(Slightly different than book version fig 7.9)

\[
def \text{build\_tree}(pq)
    while pq.length > 1
        node1 = pq.shift
        node2 = pq.shift
        pq << Node.combine(node1, node2)
    end
    return pq.first
end
\]

Creates a new node with node1 as its left child and node2 as its right child.
Building our Huffman Tree

tree = build_tree(pq)
=> ( 1.000 ( 0.428 ( 0.195 ( 0.089
    ( L: 0.044 ) ( H: 0.045 ) ) ( K: 0.106 ) )
    ( 0.233 ( O: 0.106 ) ( 0.127 ( U: 0.059 )
      ( ' : 0.068 ) ) ) ) ( 0.192 ( A: 0.262 )
    ( 0.310 ( 0.143 ( 0.071 ( M: 0.032 )
      ( 0.039 ( W: 0.009 ) ( P: 0.030 ) )
      ( E: 0.072 ) ) ( 0.167 ( N: 0.083 )
      ( I: 0.084 ) ) ) )

) )

This is just our Huffman tree expressed using recursively nested parenthetical components:
( root ( left ) ( right ) )

Assigning Codes, Encoding & Decoding

ht = assign_codes(tree)

ht[“W”]
=> 110010

ht[“A”]
=> 10

msg = encode(“ALOHA”, tree)
=> 10000010000110

decode(msg, tree)
=> “ALOHA”

from BitLab takes a Huffman tree and returns a hash table that maps each letter to its binary code

Note the [ ] syntax. This returns the code associated with the character from the hash table.

from BitLab encode and decode functions