Text Compression

- **Input**: String $S$
- **Output**: String $S'$
  - Shorter
  - $S$ can be reconstructed from $S'$
Text Compression: Examples

Encodings

ASCII: 8 bits/character
Unicode: 16 bits/character

<table>
<thead>
<tr>
<th>Symbol</th>
<th>ASCII</th>
<th>Fixed length</th>
<th>Var. length</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>01100001</td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>b</td>
<td>01100010</td>
<td>001</td>
<td>011</td>
</tr>
<tr>
<td>c</td>
<td>01100011</td>
<td>010</td>
<td>011</td>
</tr>
<tr>
<td>d</td>
<td>01100100</td>
<td>011</td>
<td>001</td>
</tr>
<tr>
<td>e</td>
<td>01100101</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

“abcde” in the different formats

ASCII: 01100001011000100110001101100100…
Fixed: 000001010011100
Var: 0001101001100

CompSci 100E
Huffman coding: *go go gophers*

ASCII(7bits) 3 bits Huffman

<table>
<thead>
<tr>
<th>Character</th>
<th>ASCII Code</th>
<th>Huffman Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>103</td>
<td>1100111 000 00</td>
</tr>
<tr>
<td>o</td>
<td>111</td>
<td>1101111 001 01</td>
</tr>
<tr>
<td>p</td>
<td>112</td>
<td>1110000 010 1100</td>
</tr>
<tr>
<td>h</td>
<td>104</td>
<td>1101000 011 1101</td>
</tr>
<tr>
<td>e</td>
<td>101</td>
<td>1100101 100 1110</td>
</tr>
<tr>
<td>r</td>
<td>114</td>
<td>1110010 101 1111</td>
</tr>
<tr>
<td>s</td>
<td>115</td>
<td>1110011 110 101</td>
</tr>
<tr>
<td>sp.</td>
<td>32</td>
<td>1000000 111 101</td>
</tr>
</tbody>
</table>

- Encoding uses tree:
  - 0 left/1 right
  - How many bits? 37!!
  - Savings? Worth it?
Huffman Coding

- D.A Huffman in early 1950’s
- Before compressing data, analyze the input stream
- Represent data using variable length codes
- Variable length codes though *Prefix* codes
  - Each letter is assigned a codeword
  - Codeword is for a given letter is produced by traversing the Huffman tree
  - **Property:** *No codeword produced is the prefix of another*
  - Letters appearing frequently have short codewords, while those that appear rarely have longer ones
- Huffman coding is *optimal* per-character coding method
Building a Huffman tree

- **Begin with a forest of single-node trees (leaves)**
  - Each node/tree/leaf is weighted with character count
  - Node stores two values: character and count
  - There are $n$ nodes in forest, $n$ is size of alphabet?

- **Repeat until there is only one node left: root of tree**
  - Remove two minimally weighted trees from forest
  - Create new tree with minimal trees as children,
    - New tree root's weight: sum of children (character ignored)

- **Does this process terminate? How do we get minimal trees?**
  - Remove minimal trees, hmmm......
Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”
Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”
Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”
Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”
Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”
Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”
Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”
Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”
Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”
Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”
Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”
Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”
Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”
Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”
Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”
Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”
Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”
Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”
Encoding

1. Count occurrence of all occurring character \( O(N) \)
2. Build priority queue \( O(A) \)
3. Build Huffman tree \( O(A \log A) \)
4. Create Table of codes from tree \( O(A \log A) \)
5. Write Huffman tree and coded data to file \( O(N) \)
Properties of Huffman coding

- Want to minimize weighted path length $L(T)$ of tree $T$
  \[ L(T) = \sum_{i \in \text{Leaf}(T)} d_i w_i \]
  - $w_i$ is the weight or count of each codeword $i$
  - $d_i$ is the leaf corresponding to codeword $i$

- How do we calculate character (codeword) frequencies?

- Huffman coding creates pretty full bushy trees?
  - When would it produce a “bad” tree?

- How do we produce coded compressed data from input efficiently?
Writing code out to file

- How do we go from characters to encodings?
  - Build Huffman tree
  - Root-to-leaf path generates encoding

- Need way of writing bits out to file
  - Platform dependent?
  - Complicated to write bits and read in same ordering

- See `BitInputStream` and `BitOutputStream` classes
  - Depend on each other, bit ordering preserved

- How do we know bits come from compressed file?
  - Store a *magic* number
Decoding a message

011000000100001001101

CompSci 100E
Decoding a message

1100000100001001101

CompSci 100E
Decoding a message

100000100001001101

CompSci 100E
Decoding a message

00000100001001101

![Decoding a message diagram]
Decoding a message

0000100001001101

G

CompSci 100E
Decoding a message

```
000100001001101
```
Decoding a message
Decoding a message

0100001001101
Decoding a message

100001001101
Decoding a message

00001001101
Decoding a message

0001001101
Decoding a message

001001101

GO
Decoding a message

01001101
Decoding a message

1001101

CompSci 100E
Decoding a message

001101

Compsci 100E 34.41

G O O
Decoding a message

01101

GOO
Decoding a message

1101

G O O
Decoding a message

101

GOO
Decoding a message
Decoding a message
Decoding a message

01100000100001001101

GOOD
Decoding

1. Read in tree data \( O(\quad) \)

2. Decode bit string with tree \( O(\quad) \)
Huffman coding: *go go gophers*

<table>
<thead>
<tr>
<th>ASCII</th>
<th>3 bits</th>
<th>Huffman</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>103</td>
<td>1100111</td>
</tr>
<tr>
<td>o</td>
<td>111</td>
<td>1101111</td>
</tr>
<tr>
<td>p</td>
<td>112</td>
<td>1110000</td>
</tr>
<tr>
<td>h</td>
<td>104</td>
<td>1101000</td>
</tr>
<tr>
<td>e</td>
<td>101</td>
<td>1100101</td>
</tr>
<tr>
<td>r</td>
<td>114</td>
<td>1110010</td>
</tr>
<tr>
<td>s</td>
<td>115</td>
<td>1110011</td>
</tr>
<tr>
<td>sp.</td>
<td>32</td>
<td>1000000</td>
</tr>
</tbody>
</table>

- **choose two smallest weights**
  - combine nodes + weights
  - Repeat
  - Priority queue?
- **Encoding uses tree:**
  - 0 left/1 right
  - How many bits?
Huffman Tree 2

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”

- E.g. “A SIMPLE” ⇔ “10101101001000101001110011100000”
Huffman Tree 2

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”

- E.g. “A SIMPLE” ⇔ “1010110100100100101001110011100000”
Huffman Tree 2

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”

E.g. “A SIMPLE” ⇔ “101011010010010011001110011100000”
“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”

E.g. “A SIMPLE” ⇔ “10101101001000101001110011100000”
Huffman Tree 2

- "A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS"
- E.g. "A SIMPLE" ↔ "1010110100100101001110011100000"

![Huffman Tree Diagram]
Huffman Tree 2

“"A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS"

E.g. “A SIMPLE” ⇔ “10101101001000101001110011100000”
"A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS"

E.g. “A SIMPLE” ⇔ “10101101001000101001110011100000”
Huffman Tree 2

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”

E.g. “A SIMPLE” ⇔ “10101101001000101001110011100000”
Huffman Tree 2

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”

E.g. “A SIMPLE” ⇔ “1010110100100101001110011100000”

E.g. “A SIMPLE” ⇔ “1010110100100101001110011100000”
Other methods

- Adaptive Huffman coding
- Lempel-Ziv algorithms
  - Build the coding table on the fly while reading document
  - Coding table changes dynamically
  - Protocol between encoder and decoder so that everyone is always using the right coding scheme
  - Works well in practice (compress, gzip, etc.)
- More complicated methods
  - Burrows-Wheeler (bunzip2)
  - PPM statistical methods