Text Compression

- Input: String $S$
- Output: String $S'$
  - Shorter
  - $S$ can be reconstructed from $S'$

Text Compression: Examples

```
<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>11</td>
</tr>
<tr>
<td>c</td>
<td>01100011</td>
<td>010</td>
<td>01</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>10</td>
</tr>
</tbody>
</table>
```

Unicode adds 8 more 0's to left of ASCII

Huffman coding: go go gophers

```
<table>
<thead>
<tr>
<th>Symbol</th>
<th>ASCII</th>
<th>Fixed length</th>
<th>Var. length</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>110100011</td>
<td>000</td>
<td>00</td>
</tr>
<tr>
<td>o</td>
<td>11111111</td>
<td>001</td>
<td>01</td>
</tr>
<tr>
<td>p</td>
<td>11110000</td>
<td>010</td>
<td>1100</td>
</tr>
<tr>
<td>h</td>
<td>11010000</td>
<td>011</td>
<td>1101</td>
</tr>
<tr>
<td>e</td>
<td>11011101</td>
<td>100</td>
<td>1110</td>
</tr>
<tr>
<td>r</td>
<td>11110001</td>
<td>101</td>
<td>1111</td>
</tr>
<tr>
<td>s</td>
<td>11111100</td>
<td>110</td>
<td>101</td>
</tr>
<tr>
<td>sp.</td>
<td>10000001</td>
<td>111</td>
<td>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”

```
I E N S M A B O T G D L R U P F C
```

Building a tree

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

```
I E N S M A B O T G D L R U P F C
```

```
I
F
C
```
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 \( O(N) \)
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

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```

```
Lenna
```

```
01100001100011000100001101
```

```
417
```
Decoding a message

000100001001101

Decoding a message

0100001001101

Decoding a message

0100001001101

Decoding a message

100001001101
Decoding a message

0001001101

GO

Decoding a message

001001101

GO

Decoding a message

01001101

GO

Decoding a message

01001101

GO
Decoding a message

1001101

GO

001101

GOO

Decoding a message

01101

GOO

1101

GOO
Decoding a message

101

GOOD

Decoding a message

01

GOOD

Decoding a message

1

GOOD

Decoding a message

01100000100001001101

GOOD
Decoding

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

2. Decode bit string with tree $O(\quad)$

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

E.g. “A SIMPLE” $\leftrightarrow$ “101011010001010011101100000”
“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” ⇔ “101011010001010011100000”

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