Text Compression

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

Text Compression: Examples

- "abcde" in the different formats
- 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>00</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>

Huffman coding: *go go gophers*

- ASCII (7 bits) 3 bits Huffman
  - g 103 1100111 000 00
  - o 111 1101111 001 01
  - p 112 1110000 010 1100
  - h 104 1101000 011 1101
  - e 101 1100101 100 1110
  - r 114 1110010 101 1111
  - s 115 1110011 110 101
  - sp. 32 1000000 111 101

- 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”

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

<table>
<thead>
<tr>
<th>Binary</th>
<th>Huffman Code</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>01100000100001001101</td>
<td>1100000100001001101</td>
<td>T U C L R D G O T B A M S</td>
</tr>
</tbody>
</table>

Decoding a message

<table>
<thead>
<tr>
<th>Binary</th>
<th>Huffman Code</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100000100001001101</td>
<td>1100000100001001101</td>
<td>T U C L R D G O T B A M S</td>
</tr>
</tbody>
</table>
Decoding a message

```
100000100001001101
```

Decoding a message

```
000010001001101101
```

Decoding a message

```
000010001001101
```

Decoding a message

```
00010001001101101
```

```
G
```
Decoding a message

00100001001101

100001001101

100001001101

000001001101
Decoding a message

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

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

GOOD

GOOD

GOOD

GOOD
Huffman coding: *go go gophers*

ASCII | 3 bits | Huffman
--- | --- | ---
g | 103 | 1100111 | 000 |
σ | 111 | 1101111 | 001 |
p | 112 | 1110000 | 010 |
h | 104 | 1101000 | 011 |
e | 101 | 1100101 | 100 |
r | 114 | 1110010 | 101 |
s | 115 | 1110011 | 110 |
sp. | 32 | 1000000 | 111 |

- 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" ↔ "10101101001001000101001110011100000"

---

**Huffman Tree 2**

- "A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS"
  - E.g. "A SIMPLE" ↔ "10101101001001000101001110011100000"
“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”

- E.g. "A SIMPLE" $\mapsto$ “10101101000101101100000”
A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS

E.g. “A SIMPLE” ↔ “10101101001000101001110011100000”

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