Last Time

- Greedy Algorithms
Huffman Coding
Problem: You want to transmit information. How do you do it?
Transmitting "information".
Text only. Telegraph.
How much space does it take to transmit a letter down a wire.
There are 27 letters (26 letters plus a space).
Assume we have a representation of "bits".
(Maybe dots and dashes, 1s and 0s, it doesn’t matter.)
Transmitting "information" Idea #1. Represent your alphabet as a 27-bit sequence.

__ = 00000 00000 00000 00000 00000 01
A = 00000 00000 00000 00000 00000 10
B = 00000 00000 00000 00000 00000 00001 00
C = 00000 00000 00000 00000 00000 00010 00

... 
Z = 10000 00000 00000 00000 00000 00
Transmitting “information” Idea #1. Represent your alphabet as a 27-bit sequence.

\[
\begin{align*}
_ & = 00000 \ 00000 \ 00000 \ 00000 \ 00000 \ 00001 \ 01 \\
A & = 00000 \ 00000 \ 00000 \ 00000 \ 00000 \ 00000 \ 10 \\
B & = 00000 \ 00000 \ 00000 \ 00000 \ 00000 \ 00001 \ 00 \\
C & = 00000 \ 00000 \ 00000 \ 00000 \ 00000 \ 00000 \ 0010 \ 00 \\
\cdots \\
Z & = 10000 \ 00000 \ 00000 \ 00000 \ 00000 \ 00000 \ 00
\end{align*}
\]

How much space will this take to transmit an \( n \) character string?
Transmitting “information” Idea #2. Represent your alphabet as a 4-bit sequence.

\[
\begin{align*}
_ & = 0 = 00000 \\
A & = 1 = 00001 \\
B & = 2 = 00010 \\
C & = 3 = 00011 \\
\cdots \\
Z & = 26 = 11010
\end{align*}
\]
Transmitting “information” Idea #2. Represent your alphabet as a 4-bit sequence.

\[ \begin{align*}
_0 &= 0 = 0000 \\
A &= 1 = 0001 \\
B &= 2 = 0010 \\
C &= 3 = 0011 \\
&\quad \ldots \\
Z &= 26 = 11010
\end{align*} \]

How much space will this take to transmit an \( n \) character string?
Transmitting “information”

Generalize Idea #1.
If we have a $k$ element dictionary, how many bits will it take to encode each element?
Transmitting “information”

Generalize Idea #1.
If we have a $k$ element dictionary, how many bits will it take to encode each element?

$k$-bits.
How many bits per string of length $n$?
Transmitting “information”

Generalize Idea #1.
If we have a $k$ element dictionary, how many bits will it take to encode each element?

$k$-bits.

How many bits per string of length $n$?

$nk$
Coding Information

Transmitting “information”

Generalize Idea #2.
If we have a $k$ element dictionary, how many bits will it take to encode each element?
Transmitting “information”

Generalize Idea #2.
If we have a $k$ element dictionary, how many bits will it take to encode each element?

$$\left\lceil \log(k) \right\rceil$$

How many bits per string of length $n$?
Transmitting “information”

Generalize Idea #2.
If we have a $k$ element dictionary, how many bits will it take to encode each element?

$$\lceil \log(k) \rceil$$

How many bits per string of length $n$?

$$n \cdot \lceil \log(k) \rceil$$
Transmitting “information”.

Morse Code doesn’t do this though.
How does Morse Code differ from these representations?
<table>
<thead>
<tr>
<th>A</th>
<th>.--</th>
<th>M</th>
<th>--</th>
<th>Y</th>
<th>--.--</th>
<th>6</th>
<th>----.----.----.</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>--...</td>
<td>N</td>
<td>--</td>
<td>Z</td>
<td>--..</td>
<td>7</td>
<td>----.</td>
</tr>
<tr>
<td>C</td>
<td>--.--</td>
<td>O</td>
<td>--</td>
<td>Ä</td>
<td>--.--</td>
<td>8</td>
<td>-----.----.</td>
</tr>
<tr>
<td>D</td>
<td>--</td>
<td>P</td>
<td>--.</td>
<td>Ö</td>
<td>----.</td>
<td>9</td>
<td>-----.----.</td>
</tr>
<tr>
<td>E</td>
<td>.</td>
<td>Q</td>
<td>--.</td>
<td>Ü</td>
<td>--.--</td>
<td>.</td>
<td>-----.</td>
</tr>
<tr>
<td>F</td>
<td>--.--</td>
<td>R</td>
<td>--</td>
<td>Ch</td>
<td>----.</td>
<td>,</td>
<td>-----.</td>
</tr>
<tr>
<td>G</td>
<td>--</td>
<td>S</td>
<td>...</td>
<td>0</td>
<td>----.----.----.</td>
<td>?</td>
<td>-----.----.</td>
</tr>
<tr>
<td>H</td>
<td>----.</td>
<td>T</td>
<td>--</td>
<td>1</td>
<td>----.</td>
<td>!</td>
<td>-----.</td>
</tr>
<tr>
<td>I</td>
<td>..</td>
<td>U</td>
<td>--</td>
<td>2</td>
<td>----.</td>
<td>:</td>
<td>-----.</td>
</tr>
<tr>
<td>J</td>
<td>----.</td>
<td>V</td>
<td>--</td>
<td>3</td>
<td>----.</td>
<td>“</td>
<td>-----.</td>
</tr>
<tr>
<td>K</td>
<td>--</td>
<td>W</td>
<td>--</td>
<td>4</td>
<td>-----.----.</td>
<td>'</td>
<td>-----.----.</td>
</tr>
<tr>
<td>L</td>
<td>--...</td>
<td>X</td>
<td>--.</td>
<td>5</td>
<td>-----.----.</td>
<td>=</td>
<td>-----.</td>
</tr>
</tbody>
</table>

Morse code
Transmitting “information”.

Morse Code doesn’t do this though. 

**variable lengthed encoding**

- It uses smaller bit representations for some tokens.

Which tokens are shorter which are longer? 
Why does it help to use shorter encodings of some tokens?
Transmitting “information”.

Morse Code doesn’t do this though.  

**variable lengthed encoding**

- It uses smaller bit representations for some tokens.

  Which tokens are shorter which are longer?  
  Why does it help to use shorter encodings of some tokens?  
  Aside: Is morse code a binary coding?

  \[
  \begin{align*}
  E &= . \\
  N &= -. \\
  R &= .- \\
  \ldots &= ? \\
  \ldots &= ?
  \end{align*}
  \]
The E, N vs. R ambiguity is a problem.

With words and language letters we can use other information to determine what the intended word was.

E.g. L E T T E R vs. L E T T E N

What is the problem?

\[
\begin{align*}
E &= . \\
N &= -. \\
R &= .-.
\end{align*}
\]
Ambiguity in coding

The E, N vs. R ambiguity is a problem.

With words and language letters we can use other information to determine what the intended word was.

E.g. L E T T E R vs. L E T T E E N

What is the problem?

\[
\begin{align*}
E &= . \\
N &= -. \\
R &= .-.
\end{align*}
\]

How can we be sure a coding does not have this ambiguity?
Guaranteeing no ambiguity

Prefix Codes
(also called Prefix-free Codes)
No codeword can be a prefix of any other codeword.

In Math:

\[ c_i \neq c_j \neq s \text{ for all } c_i, c_j \in C \]

where \( C \) is the set of code words, and \( s \) is an arbitrary bit string.
Example Prefix Codes

For example:

D = 000
R = 001
E = 10
A = 11

001101100010001
For example:

D = 000
R = 001
E = 10
A = 11

001101100010001
001 10 11 000 10 001

READER
Recap

We have a mechanism for constructing non-ambiguous variable length codes.
We think we should use shorter codes for more common tokens.

1. How do we construct prefix free codes?
2. How do we decide which letters get assigned which code?
3. *** Can we be sure that this encoding is optimal? ***
How do we construct prefix free codes?

Use a binary tree.
Leaves represent tokens.
Left edges are labeled 0
Right edges are labeled 1
The codeword is the label of the edges from the root to the leaf.
Example prefix-free coding tree

\[
\begin{array}{c}
\text{a} = 000 \\
\text{b} = 001 \\
\text{c} = 010 \\
\text{d} = 011 \\
\text{e} = 10 \\
\text{f} = 11
\end{array}
\]
Example prefix-free coding tree

How can we be sure that there are no prefixes?

a = 000
b = 001
c = 010
d = 011
e = 10
f = 11
Prefix Tree Coding Length

How many bits does it take to encode using a prefix tree?

![Prefix Tree Diagram]

The tree diagram shows the encoding of the letters a, b, c, d, e, and f using a prefix tree.

- a = 000
- b = 001
- c = 010
- d = 011
- e = 10
- f = 11

The formula for computing the overall bit cost is:

\[ B(T) = \sum_{c \in C} N(c)d_T(c) \]

Where:

- \( B(T) \) is the total bit cost.
- \( N(c) \) is the number of times the letter \( c \) appears.
- \( d_T(c) \) is the depth of the leaf in the tree.

**Explanation:**

- **a** = 000
- **b** = 001
- **c** = 010
- **d** = 011
- **e** = 10
- **f** = 11

The depth of each leaf \( c \) in the tree is:

- **a** = 0 (leaf at depth 0)
- **b** = 0 (leaf at depth 0)
- **c** = 1 (leaf at depth 1)
- **d** = 1 (leaf at depth 1)
- **e** = 2 (leaf at depth 2)
- **f** = 2 (leaf at depth 2)
Constructing a Huffman Code

Goal: more frequent tokens have lower depth in the tree.

**Huffman(C)**

$n \leftarrow |C|$
make a Priority Queue from C

for $i \leftarrow 1$ to $n - 1$ do

  allocate a new node $z$
  $z.left = Minimum(C)$
  $z.right = Minimum(C)$
  $f[z] = f[x] + f[y]$
  $Insert(C, z)$

end for

return the root of C
Example Huffman Coding tree construction

\begin{itemize}
   \item a:5
   \item b:9
   \item e:16
   \item d:13
   \item c:12
   \item f:45
\end{itemize}
Example Huffman Coding tree construction

a:5  b:9  e:16  d:13  c:12  f:45
Example Huffman Coding tree construction

```
14
  0 1
 a:5 b:9 e:16 d:13 c:12 f:45
```
Example Huffman Coding tree construction

```
  14
 /   \
0   1
 /   /
 a:5 b:9 e:16 d:13 c:12 f:45
```
Example Huffman Coding tree construction

```
|   14   |
| 0    1 |
|  a:5  b:9 |

|   25   |
| 1    0 |
|  e:16 d:13 c:12 f:45 |
```
Example Huffman Coding tree construction

```
    14
   /  \
  0   1
 a:5  b:9

    25
   /  \
  1   0
 e:16 d:13 c:12 f:45
```
Example Huffman Coding tree construction

```
30
  0 1
  14 e:16 25
  0 1
 a:5 b:9 c:12 d:13 f:45
```
Example Huffman Coding tree construction
Example Huffman Coding tree construction
Example Huffman Coding tree construction
Example Huffman Coding tree construction

```
f: 45
   0
   1
   55
   1
   0
30
   0
   1
   25
   1
   0
14
   0
   1
   e: 16
   c: 12
d: 13
   0
   1
a: 5
   b: 9
```
Showing the usefulness of Huffman Coding

Huffman Coding

<table>
<thead>
<tr>
<th>Letter</th>
<th>Freq.</th>
<th>Code</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>5</td>
<td>1100</td>
<td>4</td>
</tr>
<tr>
<td>b</td>
<td>9</td>
<td>1101</td>
<td>4</td>
</tr>
<tr>
<td>c</td>
<td>12</td>
<td>100</td>
<td>3</td>
</tr>
<tr>
<td>d</td>
<td>13</td>
<td>101</td>
<td>3</td>
</tr>
<tr>
<td>e</td>
<td>16</td>
<td>111</td>
<td>3</td>
</tr>
<tr>
<td>f</td>
<td>45</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
B(T) = \sum_{c \in C} N(c)d_T(c)
\]

\[
B(T) = N(a)d_T(a) + N(b)d_T(b) + N(c)d_T(c) + N(d)d_T(d) + N(e)d_T(e) + N(f)d_T(f)
\]

\[
= 5 \cdot 4 + 9 \cdot 4 + 12 \cdot 3 + 13 \cdot 3 + 16 \cdot 3 + 45 \cdot 1
\]

\[
= 224
\]
## Average cost per token

<table>
<thead>
<tr>
<th>Approach</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$ coding</td>
<td>6 bits/token</td>
</tr>
<tr>
<td>$\log n$ coding</td>
<td>3 bits/token</td>
</tr>
<tr>
<td>Huffman</td>
<td>2.24 bits/token</td>
</tr>
</tbody>
</table>
The Huffman coding algorithm is **greedy**.

The optimization problem it is solving is – define an optimal sized prefix code for a set of letter-frequency pairs \((a_i, f_i)\)

The greedy choice at each step:

- Let the two letters (or codes) that are least frequently used be represented by the same code and differ by a single bit.
Correctness of Huffman Coding

Lemma

Let $x$ and $y$ are the two least frequent characters in $C$. Then there exists an optimal code in which the codewords for $x$ and $y$ are the same length and differ by a single bit.

Proof: Let $T$ be a tree representing an optimal prefix code. Show that either 1) $x$ and $y$ are sibling leaves at maximal depth, or 2) that $T$ can be modified so that they are. Then there exists an optimal code where $x$ and $y$ have the same length and differ only in the last bit.
Correctness of Huffman Coding

Let \( a \) and \( b \) be two nodes at maximal depth that differ only in the last bit. Assume, \( f[x] < f[a] \) and \( f[y] < f[b] \). If we swap the location of \( x \) and \( a \), let us examine the coding of the new tree \( T' \).

\[
B(T) - B(T') = \sum_{c \in C} N(c)d_T(c) - \sum_{c \in C} N(c)d'_T(c)
= (f[a] - f[x])(d_T[a] - d_T[x])
\geq 0
\]

Therefore swapping \( x \) and \( a \) does not increase the cost of the encoding. The same logic applies to swapping \( y \) and \( b \).
Correctness of Huffman Coding
Lemma

Let $x$ and $y$ be the two characters in $C$ with lowest frequency. Let $C'$ be a new alphabet with $x$ and $y$ replaced by $z$ where $f[z] = f[x] + f[y]$. Let $T'$ is a optimal prefix tree of $C'$, then $T$ – replacing $z$ with $x$ and $y$ – is an optimal prefix tree for $C$.

Proof: $B(T)$ expressed in terms of $B(T')$.
If $c \in C - \{x, y\}$, then $f[c]d_T[c] = f[c]d_{T'}[c]$.
Also, $d_T[x] = d_T[y] = d'_T[z] + 1.$

$$f[x]d_T[x] + f[y]d_T[y] = (f[x] + f[y])d[z](d_{T'}[z] + 1)$$
$$= f[z]d_{T'}[z] + (f[x] + f[y])$$
$$B(T) = B(T') + (f[x] + f[y])$$

If $T'$ is an optimal prefix tree of $C'$, then $T$ must be an optimal prefix tree of $C$. – “Cut-and-paste” method.
Huffman coding constructs optimal prefix codes.

- Relationship to Entropy \( H(x) = - \sum_i p(x_i) \log p(x_i) \)
- The “alphabet” \( C \) can be made of any enumerated type, not just letter.
Discussion Problem
Bye

Next time
  - At long last, Graphs!