Codes...

- $\Sigma$: alphabet.
- **binary code**: assigns a string of $0$s and $1$s to each character in the alphabet.
- each symbol in input = a codeword over some other alphabet.
- Useful for transmitting messages over a wire: only $0/1$.
- receiver gets a binary stream of bits...
- ... decode the message sent.
- **prefix code**: reading a prefix of the input binary string uniquely match it to a code word.
- ... continuing to decipher the rest of the stream.
- binary/prefix code is **prefix-free** if no code is a prefix of any other.
- ASCII and Unicode’s UTF-8 are both prefix-free binary codes.
Codes...

- Encoding: given frequency table: $f[1 \ldots n]$.
- $f[i]$: frequency of $i$th character.
- $\text{code}(i)$: binary string for $i$th character.
- $\text{len}(s)$: length (in bits) of binary string $s$.
- Compute tree $T$ that minimizes

$$\text{cost}(T) = \sum_{i=1}^{n} f[i] \ast \text{len}(\text{code}(i)), \quad (1)$$

**Frequency table for**

"A tale of two cities" by Dickens

<table>
<thead>
<tr>
<th>char</th>
<th>frequency</th>
<th>code</th>
<th>char</th>
<th>freq</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>'A'</td>
<td>48165</td>
<td>1110</td>
<td>'N'</td>
<td>42380</td>
<td>1100</td>
</tr>
<tr>
<td>'B'</td>
<td>8414</td>
<td>10100</td>
<td>'O'</td>
<td>46499</td>
<td>1101</td>
</tr>
<tr>
<td>'C'</td>
<td>13896</td>
<td>00100</td>
<td>'P'</td>
<td>9957</td>
<td>10100</td>
</tr>
<tr>
<td>'D'</td>
<td>28041</td>
<td>0011</td>
<td>'Q'</td>
<td>667</td>
<td>1111011001</td>
</tr>
<tr>
<td>'E'</td>
<td>74809</td>
<td>011</td>
<td>'R'</td>
<td>37187</td>
<td>0101</td>
</tr>
<tr>
<td>'F'</td>
<td>13559</td>
<td>11111</td>
<td>'S'</td>
<td>37575</td>
<td>1000</td>
</tr>
<tr>
<td>'G'</td>
<td>12530</td>
<td>11110</td>
<td>'T'</td>
<td>54024</td>
<td>000</td>
</tr>
<tr>
<td>'H'</td>
<td>38961</td>
<td>1001</td>
<td>'U'</td>
<td>16726</td>
<td>01001</td>
</tr>
<tr>
<td>'I'</td>
<td>41005</td>
<td>1011</td>
<td>'V'</td>
<td>5199</td>
<td>111101010</td>
</tr>
<tr>
<td>'J'</td>
<td>710</td>
<td>1111011010</td>
<td>'W'</td>
<td>14113</td>
<td>00101</td>
</tr>
<tr>
<td>'K'</td>
<td>4782</td>
<td>11110111</td>
<td>'X'</td>
<td>724</td>
<td>1111011011</td>
</tr>
<tr>
<td>'L'</td>
<td>22030</td>
<td>10101</td>
<td>'Y'</td>
<td>12177</td>
<td>111100</td>
</tr>
<tr>
<td>'M'</td>
<td>15298</td>
<td>01000</td>
<td>'Z'</td>
<td>215</td>
<td>1111011000</td>
</tr>
</tbody>
</table>

**Computed prefix codes...**

**The Huffman tree generating the code**

Build only on A-Z for clarity.
Mergeability of code trees

- two trees for some disjoint parts of the alphabet...
- Merge into larger tree by creating a new node and hanging the trees from this common node.
- ![Diagram of merging trees]
- ...put together two subtrees.

Building optimal prefix code trees

- take two least frequent characters in frequency table...
- ... merge them into a tree, and put the root of merged tree back into table.
- ...instead of the two old trees.
- Algorithm stops when there is a single tree.
- Intuition: infrequent characters participate in a large number of merges. Long code words.
- Algorithm is due to David Huffman (1952).
- Resulting code is best one can do.
- **Huffman coding**: building block used by numerous other compression algorithms.

Analysis...

**Lemma**

- **T**: optimal code tree (prefix free!).
- Then **T** is a full binary tree.
- ... every node of **T** has either 0 or 2 children.
- If height of **T** is **d**, then there are leafs nodes of height **d** that are sibling.

Proof...

- If there is an internal node in **T** that has one child, we can remove this node from **T**, by connecting its only child directly with its parent. The resulting code tree is clearly a better compressor, in the sense of
  \[\text{cost}(T) = \sum_{i=1}^{n} f[i] * \text{len(code}(i))\].
- **u**: leaf **u** with maximum depth **d** in **T**. Consider parent \( v = p(u) \).
- \( \Rightarrow \ v \): has two children, both leafs
Lemma
Let \( x \) and \( y \) be the two least frequent characters (breaking ties between equally frequent characters arbitrarily). There is an optimal code tree in which \( x \) and \( y \) are siblings.

Proof...
1. Claim: \( \exists \) optimal code s.t. \( x \) and \( y \) are siblings + deepest.
2. \( T \): optimal code tree with depth \( d \).
3. By lemma... \( T \) has two leaves at depth \( d \) that are siblings.
4. If not \( x \) and \( y \), but some other characters \( \alpha \) and \( \beta \).
5. \( T' \): swap \( x \) and \( \alpha \).
6. \( x \) depth inc by \( \Delta \), and depth of \( \alpha \) decreases by \( \Delta \).
7. \( \text{cost}(T') = \text{cost}(T) - (f[\alpha] - f[x]) \Delta. \)
8. \( x \): one of the two least frequent characters.
   ...but \( \alpha \) is not.
9. \( \iff f[\alpha] > f[x] \).
10. Swapping \( x \) and \( \alpha \) does not increase cost.
11. \( T' \): optimal code tree, swapping \( x \) and \( \alpha \) does not decrease cost.
12. \( T' \) is also an optimal code tree (\( f[\alpha] = f[x] \)).
13. Swapping \( y \) and \( b \) must give yet another optimal code tree.
14. Final opt code tree, \( x, y \) are max-depth siblings.

Huffman’s codes are optimal

Theorem
Huffman codes are optimal prefix-free binary codes.

Proof...
1. If message has 1 or 2 diff characters, then theorem easy.
2. \( f[1 \ldots n] \) be original input frequencies.
3. Assume \( f[1] \) and \( f[2] \) are the two smallest.
5. lemma \( \iff \exists \) opt. code tree \( T_{\text{opt}} \) for \( f[1..n] \)
6. \( T_{\text{opt}} \) has 1 and 2 as siblings.
7. Remove 1 and 2 from \( T_{\text{opt}} \).
8. \( T'_{\text{opt}} \): Remaining tree has \( 3, \ldots, n \) as leaves and “special” character \( n+1 \) (i.e., parent 1, 2 in \( T_{\text{opt}} \)).
La proof continued...

- character $n+1$: has frequency $f[n+1]$. 
  Now, $f[n+1] = f[1] + f[2]$, we have

$$\text{cost}(T_{\text{opt}}) = \sum_{i=1}^{n} f[i] \text{depth}_{T_{\text{opt}}}(i)$$

$$= \sum_{i=3}^{n+1} f[i] \text{depth}_{T_{\text{opt}}}(i) + f[1] \text{depth}_{T_{\text{opt}}}(1) + f[2] \text{depth}_{T_{\text{opt}}}(2) - f[n+1] \text{depth}_{T_{\text{opt}}}(n+1)$$

$$= \text{cost}(T'_{\text{opt}}) + (f[1] + f[2]) \text{depth}(T_{\text{opt}}) - (f[1] + f[2])(\text{depth}(T_{\text{opt}}) - 1)$$

$$= \text{cost}(T'_{\text{opt}}) + f[1] + f[2].$$

La proof continued...

- implies $\min T_{\text{opt}} \equiv \min T'_{\text{opt}}$.

- $T'_{\text{opt}}$: must be optimal coding tree for $f[3, \ldots, n+1]$.

- $T'_{H}$: Huffman tree for $f[3, \ldots, n+1]$.

- $T_H$: overall Huffman tree constructed for $f[1, \ldots, n]$.

- By construction: $T'_{H}$ formed by removing leafs 1 and 2 from $T_H$.

- By induction: Huffman tree generated for $f[3, \ldots, n+1]$ is optimal.

- $\text{cost}(T'_{\text{opt}}) = \text{cost}(T'_{H}).$


- $\implies$ Huffman tree has the same cost as the optimal tree.

What we get...

- A tale of two cities: 779,940 bytes.

- using above Huffman compression results in a compression to a file of size 439,688 bytes.

- Ignoring space to store tree.

- gzip: 301,295 bytes
  bzip2: 220,156 bytes!

- Huffman encoder can be easily written in a few hours of work!

- All later compressors use it as a black box...

Average size of code word

- input is made out of $n$ characters.

- $p_i$: fraction of input that is $i$th char (probability).

- use probabilities to build Huffman tree.

- Q: What is the length of the codewords assigned to characters as function of probabilities?

- special case...

Lemma

Let $1, \ldots, n$ be $n$ symbols, such that the probability for the $i$th symbol is $p_i$, and furthermore, there is an integer $l_i \geq 0$, such that $p_i = 1/2^l$. Then, in the Huffman coding for this input, the code for $i$ is of length $l_i$. 
Proof

- Proof by induction of the Huffman algorithm.
- Let $n = 2$: claim holds since there are only two characters with probability $1/2$.
- Let $i$ and $j$ be the two characters with lowest probability.
- Must be that $p_i = p_j$ (otherwise, $\sum_k p_k$ can not be equal to one).
- Huffman’s tree merges these two letters, into a single “character” that have probability $2p_i$.
- New “character” has encoding of length $l_i - 1$, by induction (on remaining $n - 1$ symbols).
- Resulting tree encodes $i$ and $j$ by code words of length $(l_i - 1) + 1 = l_i$.

\[ \sum_i p_i \log \frac{1}{p_i} \]
\[ H(X) = \sum_i \Pr[X = i] \log \frac{1}{\Pr[X = i]} \]
which is the \textit{entropy} of $X$.