CS 151
Huffman Codes and Data Compression
• lab 10 due on Friday
• Final is next Thursday (Dec. 20) 2-4pm in King 123 (our classroom)
• I cannot let people take the exam early unless they have 3 finals within 24 hours. (These are rules I cannot mess with, sorry!)
• Plan for this week: review via a variety of fun applications. Course evals Fri.
Data Compression Problem

Motivation:

• computers store data as sequence of bits (0 or 1)
• need encoding schemes that take text in richer alphabets, convert to bits
• simplest way to do this is to use a fixed number of bits per symbol
  • ex: there are about 100 different ASCII characters
  • we would need $\lceil \log_2(100) \rceil = 7$ bits to distinguish them ($2^7 = 128 > 100$)
    • ‘a’ is 97 in ASCII, i.e. 64+32+1 so 1100001
  • in general, for C chars, you need $\lceil \log_2(C) \rceil$ bits for a fixed-len encoding
• but not all characters appear with the same frequency in most texts
  • in English, e, t, a, o, i, and n get used much more freq than q, j, x, z
  • seems wasteful to use same length encoding.
  • would be better to use short encoding for more frequent characters
  • i.e., maybe we can decrease the average number of bits per character

So the problem is: how should we encode chars to minimize average bits per char? That is, how can we compress data as much as possible?
Way back when...

- before computers, the telephone, and the radio there was this thing called...
- the telegraph
- telegraphs could only transmit pulses down a wire
- if you wanted to send a message, you needed to encode it as seq of pulses
- Samuel Morse developed Morse code
  - each letter is a sequence of dots (short pulses) and dashes (long pulses)
  - (can think of dots as 0’s, dashes as 1’s, and so it’s a binary encoding)
  - Morse encoded e to 0 (a single dot), t to 1 (a single dash), a to 01,...
  - so he made more frequent letters (in Eng.) have shorter encodings
  - but what does 0101 correspond to? Could be eta, aa, etet, aet...
  - the code was ambiguous, because some codes are prefixes of others.
  - to fix this, ppl would pause between characters, effectively encoding
    words over 3-letter alphabet of 0, 1, and “pause.”
  - so, to use only 0 and 1 would need to extend the encoding somehow
A prefix code for a set $S$ of characters is a function $\gamma$ that maps each letter $x \in S$ to some sequence of 0’s and 1’s, in such a way that for distinct $x, y \in S$ the sequence $\gamma(x)$ is not a prefix of the sequence $\gamma(y)$.

Alternative definition: a prefix code is a Trie with bases 0 and 1 such that only the leaves represent characters.

Ex.

Note that we could make this tree smaller by moving the $a$ up a level:
A prefix code for a set $S$ of characters is a function $\gamma$ that maps each letter $x \in S$ to some sequence of 0’s and 1’s, in such a way that for distinct $x, y \in S$ the sequence $\gamma(x)$ is not a prefix of the sequence $\gamma(y)$.

Alternative definition: a prefix code is a Trie with bases 0 and 1 such that only the leaves represent characters.

Ex.

With a prefix code, a given sequence (e.g. 001000001) is a unique encoding.
Optimal Prefix Codes

Recall that our goal is to encode more frequent letters with shorter sequences. Suppose that for each character \( x \in S \) there is a frequency \( f_x \) that represents the number of times that \( x \) occurs in the text we want to encode.

Then the length of our encoding is \( \sum_{x \in S} f_x \cdot |\gamma(x)| = \sum_{x \in S} f_x \cdot (\text{depth}(x)) \)

Ex. Suppose \( f_a = 32, f_b = 25, f_c = 20, f_d = 18, f_3 = 5 \)

Then total num bits is: \( 32 \cdot 2 + 25 \cdot 2 + 20 \cdot 3 + 18 \cdot 2 + 5 \cdot 3 = 225 \).

For a fixed-length 3-bit encoding: \( (32 + 25 + 20 + 18 + 5)\cdot 3 = 300 > 225 \).

But if \( \gamma_2(a) = 11, \gamma_2(b) = 10, \gamma_2(c) = 01, \gamma_2(d) = 001, \gamma_2(e) = 000 \)

Then total num bits is: \( 32 \cdot 2 + 25 \cdot 2 + 20 \cdot 2 + 18 \cdot 3 + 5 \cdot 3 = 223 \), even better!

input: an alphabet \( S \) and letter frequencies \( f_x \) goal: determine prefix code that mins the length of the code \( \sum_{x \in S} f_x \cdot (\text{depth}(x)) \)
Optimal Prefix Codes

Claim: Every node in an optimal prefix code tree has either 0 or 2 children. (such a tree is known as a full binary tree.)

Proof.
• Suppose not, i.e. that the opt tree T has a node u with exactly one child v.
• Convert T into a new tree by replacing node u with node v (shift v up)
• This change decreases the depth (i.e. number of bits needed to encode) any leaf in the subtree rooted at u, and does not affect other leaves.
• Thus prefix code corresponding to T’ has smaller length than that of T
• (this contradicts T being optimal, and thus no such node u can exist.)

Claim: If u, v are leaves of optimal prefix code tree and depth(u)<depth(v), then $f_{\gamma}(u) \geq f_{\gamma}(v)$.

Proof.
• Suppose not, i.e. u’s character is less frequent than v’s character.
• Consider a prefix code that is the same as the optimal, but swaps u & v.
• Then the length of the code is strictly better, because depth(u) gets multiplied by the more frequent letter and depth(v) gets the less frequent one.
Claim: Every node in an optimal prefix code tree has either 0 or 2 children. (such a tree is known as a full binary tree.)

Claim: If $u,v$ are leaves of optimal prefix code tree and $\text{depth}(u) < \text{depth}(v)$, then $f_{\gamma}(u) \geq f_{\gamma}(v)$.

So, when we construct optimal prefix code trees, we should...

- make them full (give nodes 0 or 2 children)
- place lower frequency characters further down the tree
Huffman’s Algorithm

An example of a “greedy algorithm” — an algorithm that makes short-sighted decisions with the hopes that they lead to long-term optimal solutions.

idea:
• construct a prefix code by repeatedly merging trees
• initially each char c is its own single-node tree with weight = c’s frequency
• while there is more than one tree:
  • select the two currently minimum-weight trees L and R
  • merge them into a single binary tree with L on left, R on right
  • set weight of new tree = weight(L)+weight(R) = sum of tree’s char’s freq

So on each step we deal with low-frequency characters.

When we merge two of them, we are placing them one level deeper in the tree.
(But we only do so when they are the worst off, so it’s okay.)

Ex. $f_a = 32, f_b = 25, f_c = 20, f_d = 18, f_3 = 5$

Implementation Details. Use a PQ.