Week 5 - Friday

COMP 4500

Last time

- What did we talk about last time?
- Minimum spanning trees
- Clustering

Questions?

Assignment 3

Logical warmup

- Consider six straight silver chains made up of five links each
- What if you want to make one large circular chain?
- The jeweler will charge $1 for every link that he must cut open and then weld close
- What is the cheapest price possible to make the six chains into one chain?

Three-sentence Summary of Data Compression

Data Compression

Data compression

- Disk space is finite
- Internet bandwidth is limited
- It is **really** useful to compress data so that we can use a smaller number of bits to represent a larger number of bits
- Iron clad law of compression:
  - You cannot always compress a given set of bits into a smaller number of bits
  - If you could, you could compress anything, eventually, into a single 1 bit or 0 bit

Common compression

- **Lossless** compression transforms one set of bits into another (hopefully smaller) set of bits in a completely reversible way
- No bits are lost
- Examples of lossless compression:
  - Zip files
  - PNG image files
  - FLAC format for audio

Lossy compression

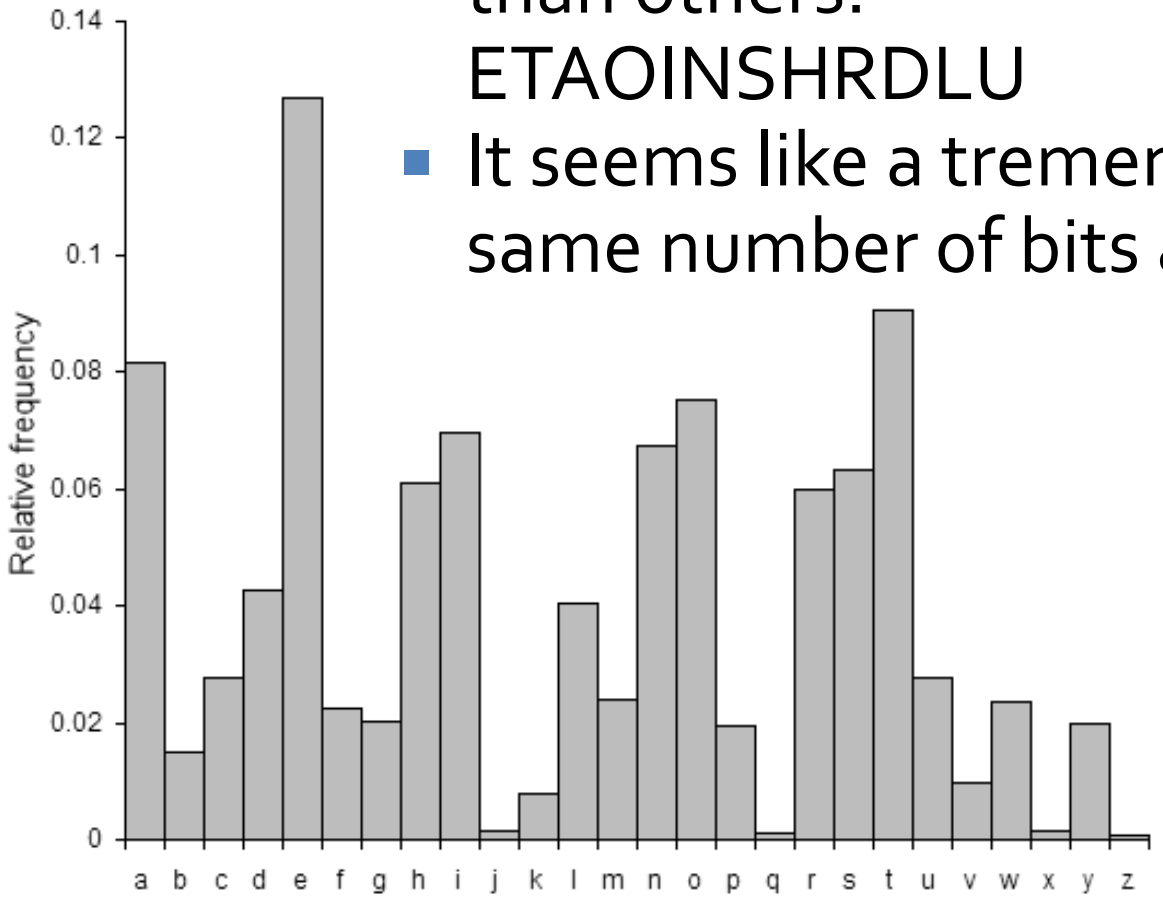
- **Lossy** compression transforms one set of bits into a (usually smaller) set of bits in a way that loses information
  - The original bits may not be reconstructible
- Lossy compression is mostly useful for media files for which the human senses don't notice the lost data
- Examples of lossy compression:
  - MPEG encoding used for DVDs and streaming video
  - JPEG image files
  - MP3 format for audio

Encoding symbols with bits

- Right now, we're only going to talk about the narrow problem of encoding symbols with bits in a lossless way
- How many letters are there in English?
- How many bits would it take to represent all of those letters?
- This is a question we think about in many CS classes because of the contortions we have to go through to store characters

Cleverly encoding symbols with bits

- Note that some English letters are used more frequently than others:
  ETAOINSHRDLU
- It seems like a tremendous waste to encode E with the same number of bits as Z



Variable length encoding

- If we use a smaller number of bits for frequent letters and a larger number of bits for rare letters, we might be able to make a much smaller document
  - Much depends on the frequency distribution
- We also have to pick our encodings carefully:
  - $a \rightarrow 0$
  - $b \rightarrow 01$
  - $c \rightarrow 11$
  - $d \rightarrow 011$
- This encoding doesn't work since 011011 could map to *acac* or *dd* or *acd* or *dac*.

Prefix codes

- We want to make an encoding such that the encoding of one letter is not a prefix of the coding of another letter
  - Such an encoding is called a prefix code
- If you have a prefix code, you can scan bits from left to right and output a letter as soon as it matches
- Example prefix code:
  - $a \rightarrow 11$
  - $b \rightarrow 01$
  - $c \rightarrow 001$
  - $d \rightarrow 10$
  - $e \rightarrow 000$

Optimal prefix codes

- If each letter ***x*** has a frequency f_x, with ***n*** letters total, nf_x gives the number of occurrences of ***x*** in a document
- Let ***code***(***x***) be the encoding of a letter ***x*** and ***S*** is the alphabet
- Total length of an encoding is:

$$\sum_{x \in S} nf_x |code(x)| = n \sum_{x \in S} f_x |code(x)|$$

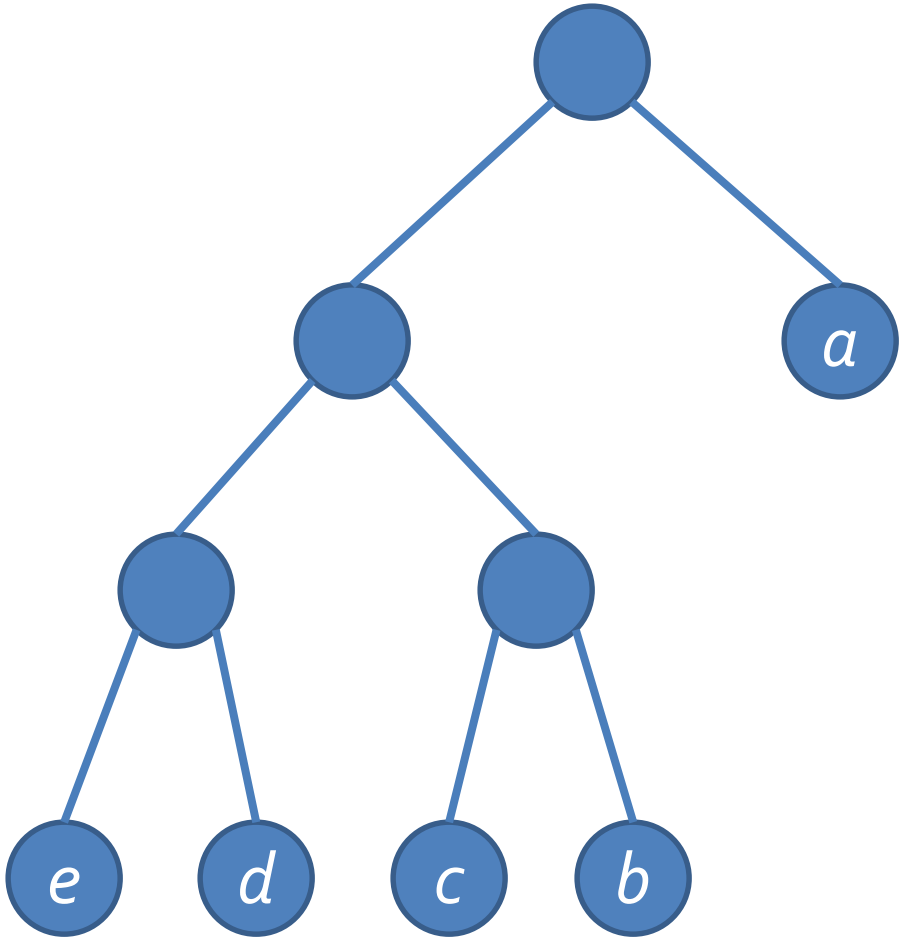
- An **optimal prefix code** minimizes average encoding length:

$$\sum_{x \in S} f_x |code(x)|$$

Algorithm design

- A key idea is that we can represent letters as leaves in a binary tree
  - Each left turn is a 0
  - Each right turn is a 1
- No letter will be the prefix of another
- Why?
- If a letter was the prefix of another, it would be on the path to the other letter, but every letter is a leaf

Prefix code tree example



$a \rightarrow 1$

$b \rightarrow 011$

$c \rightarrow 010$

$d \rightarrow 001$

$e \rightarrow 000$

Full binary trees

- Recall that a binary tree is a rooted tree in which each node has 0, 1, or 2 children
- A **full binary tree** is one in which every node that isn't a leaf has two children

An optimal prefix code is represented by a full binary tree

- Proof by contradiction:
  - Let ***T*** be a binary tree for an optimal prefix code. Suppose it contains a node ***u*** with exactly one child ***v*** (and is thus not full). We can convert ***T*** into ***T'*** by replacing ***u*** with ***v***.
  - Case 1: ***u*** is the root of ***T***
    - Delete node ***u*** and use ***v*** as the root
  - Case 2: ***u*** is not the root of ***T***
    - Let ***w*** be the parent of ***u***. Delete node ***u*** and make ***v*** the child of ***w*** that ***u*** was.
  - In both cases letters in leaves below ***u*** need one fewer bit, and other leaves are not affected. Since ***T'*** uses fewer bits for some letters, ***T*** is not optimal. Contradiction. ■

How can we figure out the tree structure?

- We know that the binary tree will be full, but there are many full binary trees with ***n*** leaves
- Imagine that we had a full binary tree ***T**** that was an optimal prefix tree
- We know that the low frequency letters should appear at the deepest levels of the tree
- For letters ***y*** and ***z***, and corresponding nodes *node*(***y***) and *node*(***z***), if *depth*(*node*(***y***)) < *depth*(*node*(***z***)) then $f_y \geq f_z$.

We don't have the structure of *T**

- If we did, we could label it by putting the highest frequency letters in the highest levels of the tree and then going down, level by level
- Instead, we work backwards
- The lowest frequency letter must be at the deepest leaf in the tree, call it ***v***
- Since this is a full binary tree, ***v*** must have a sibling ***w***

Algorithm description

- Take the two lowest frequency letters ***y*** and ***z***.
- Since they are neighbors in a full tree, we can stick them together and treat them like a meta-letter ***yz*** with the sum of their frequencies.
- Recursively repeat until everything is merged together.

Algorithm

- If S has two letters then
  - Encode one with 0 and the other with 1
- Else
  - Let y and z be the two lowest-frequency letters
  - Form a new alphabet S' by deleting y and z and replacing them with a new letter w of frequency $f_y + f_z$
  - Recursively construct a prefix code for S' with tree T'
  - Define a prefix code for S as follows:
    - Start with T'
    - Take the leaf labeled w and add two children below it labeled y and z

Proof: ABL(T') = ABL(T) - f_w

- The depth of each letter ***x*** other than ***y*** or ***z*** is the same in ***T*** and ***T'***. The depths of ***y*** and ***z*** are each one more than the depth of ***w*** in ***T'***. Recall that $f_w = f_y + f_z$.

$$\mathrm{ABL}(T) = \sum_{x \in S} f_x \cdot \mathrm{depth}_T(x)$$

$$= f_y \cdot \mathrm{depth}_T(y) + f_z \cdot \mathrm{depth}_T(z) + \sum_{x \neq y,z} f_x \cdot \mathrm{depth}_T(x)$$

$$= (f_y + f_z) \cdot (1 + \mathrm{depth}_{T'}(w)) + \sum_{x \neq y,z} f_x \cdot \mathrm{depth}_{T'}(x)$$

$$= f_w + f_w \cdot \mathrm{depth}_{T'}(w) + \sum_{x \neq y,z} f_x \cdot \mathrm{depth}_{T'}(x)$$

$$= f_w + \sum_{x \in S'} f_x \cdot \mathrm{depth}_{T'}(x)$$

$$= f_w + \mathrm{ABL}(T')$$

Our algorithm produces minimum average bits per letter of any prefix code

- Proof by contradiction:
  - Suppose that our tree T is not optimal even though it was recursively built from an optimal tree T'. Then there is some other binary tree Z such that ABL(Z) < ABL(T). However, we know that there is such a tree Z where the leaves for y and z are siblings.
  - If we remove the leaves for y and z from Z and label their parent w, we have a tree Z that defines a prefix code for S'. Since we followed the same construction, the proof from the previous slide holds for Z and Z' and ABL(Z') = ABL(Z) – f_w.
  - But since ABL(Z) < ABL(T) and ABL(T) = ABL(T') – f_w, it must be the case that ABL(Z') < ABL(T'), even though T' was optimal. Contradiction. ■

Running time of the algorithm

- We recurse ***k*** - 1 times over smaller and smaller alphabet sizes starting with ***k***
- $\sum_{i=1}^{k-1} i = \frac{k(k-1)}{2}$ which is $O(k^2)$
- However, we could use a priority queue which can extract the minimum in log ***k*** time twice at each step and add an element also in log ***k*** time, giving $O(k \log k)$ time

Exam 1 Post Mortem

Quiz

Upcoming

Next time…

- Finish exam post mortem
- Divide and conquer
- Merge sort

Reminders

- **Work on Assignment 3**
  - **Due next Friday**
- Read section 5.1
- Extra credit opportunities (0.5% each):
  - **Rublein research talk: 2/9 12:30-1:30 p.m. in Point 140**
  - **Rublein teaching demo: 2/9 3-4 p.m. in Point 140**
  - Phadke research talk: 2/12 3-4 p.m. in Point 139
  - Phadke teaching demo: 2/13 10-10:55 a.m. in Towers 112
  - Hristov teaching demo: 2/19 11:30-12:25 a.m. in Point 113
  - Hristov research talk: 2/19 4:30-5:30 p.m. in Point 139