CS 4349 Lecture—September 27th, 2017

Main topics for #lecture include #greedy_algorithms and #Huffman_codes.

Prelude

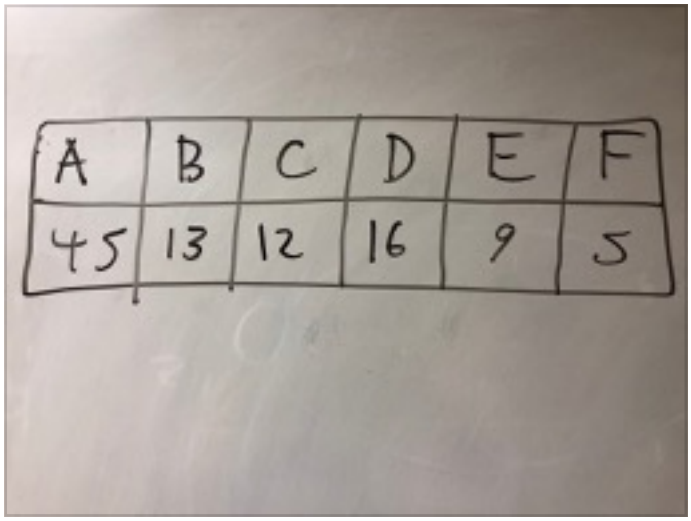
- Homework 4 due today.
- Homework 5 due Wednesday, October 11th.
- The Midterm Exam is Wednesday, October 4th from 7:00pm to 9:00pm.
  - It covers asymptotic and recurrences, divide-and-conquer, and dynamic programming. The final will be cumulative.
  - It will be four or five questions long in a similar style to the homework.
  - You will not need to provide proofs for your algorithm unless we explicitly tell you otherwise.
  - You may bring a single sheet of 8.5'' X 11'' paper with notes on both sides, handwritten or typed is fine. Otherwise, the exam is closed book.
  - We'll provide answer sheets and scratch paper.
- Monday, October 2nd, we'll do a review session instead of the normal class. Feel free to ask about homework problems or anything you see in the textbook or Jeff Erickson's lecture notes.

Huffman Codes

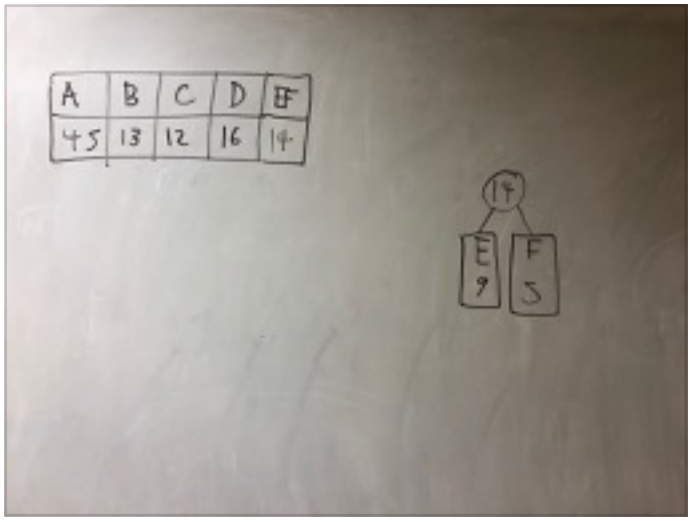
- A *binary code* assigns a string of 0s and 1s to every character in the alphabet.
- A binary code is *prefix-free* if no code is the prefix of another.
  - 7-bit ASCII is prefix free, because every code is the same length. UTF-8 is prefix free even though some codes are longer than others.
  - Morse code is a binary code, think of a dot as a 0 and a dash as a 1. Morse code is *not* prefix-free, because the code of E (0) is a prefix of the code for S (000).
  - If you're using prefix-free codes, you can just read through the concatenation of several code words and know when you've reached the end of each individual character's code.
- You can visualize any prefix-free binary code as a binary tree with the characters stored at the leaves. The code word for a character is given by the path from the root to corresponding leaf: go left for 0 or right for 1.
- This *is not* a binary search tree. The characters can be in any order.
- Prefix-free codes have this feature that you can assign shorter codes to more common characters. For example, UTF-8 seems to assume that the English alphabet's characters are the most common so they get 8-bit codes. However, you can go longer if you need to

expand to other alphabets.

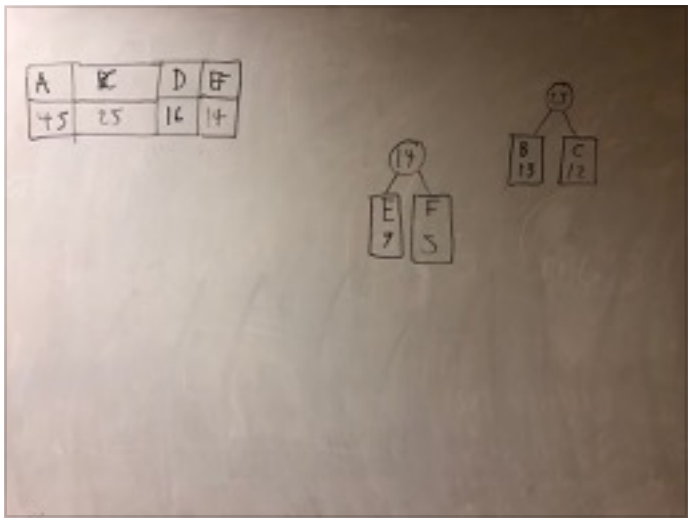
- We want to encode an alphabet so that some encoded message is as short as possible.
- Formally, we are given an array f[1 .. n] of frequency counts where f[i] is the number of times character i appears.
- Our goal is to compute a prefix-free binary code and its corresponding binary tree to minimize \sum_{i=1}^n f[i] * depth(i).
- We'll focus on building the tree since its easier to illustrate.
- One early idea for constructing the tree was given by two researchers named Fano and Shannon. They would split the frequency table into two subarrays as evenly as possible making each subarray one of the two subtrees of the root and then recursively construct the rest of the codes.
- It made some intuitive sense, but it wasn't optimal, so Fano proposed the still open problem of constructing optimal prefix-free codes in an information theory class of his.
- One of his students, David Huffman, proposed the following greedy strategy:
  - Merge the two least frequent characters and recurse.
- And it turns out that strategy is optimal! Let's discuss it in some more detail and then prove its optimality.
- Let's say you have the following frequency table:

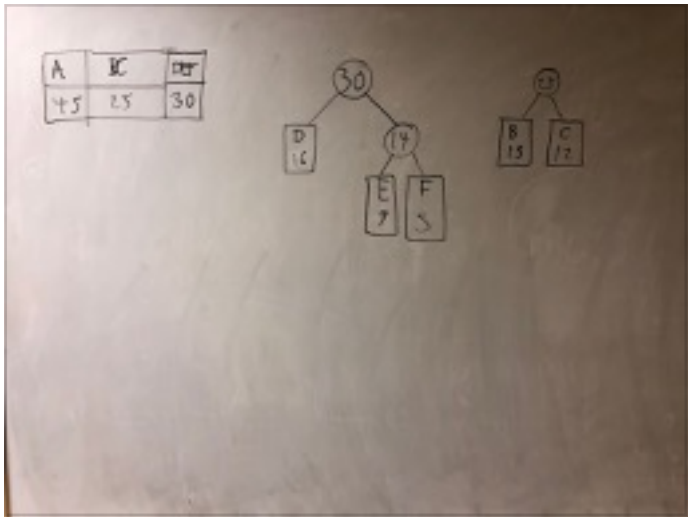


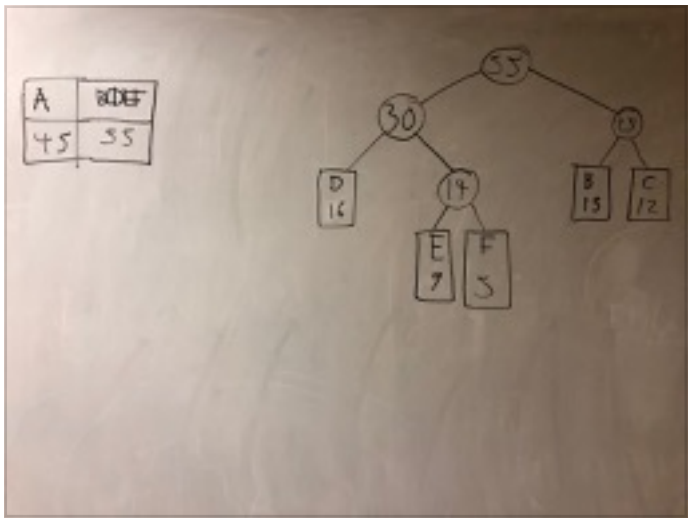
- The least frequent characters are E and F, so we merge them into a single character EF. EF becomes an internal node of the tree with children E and F.

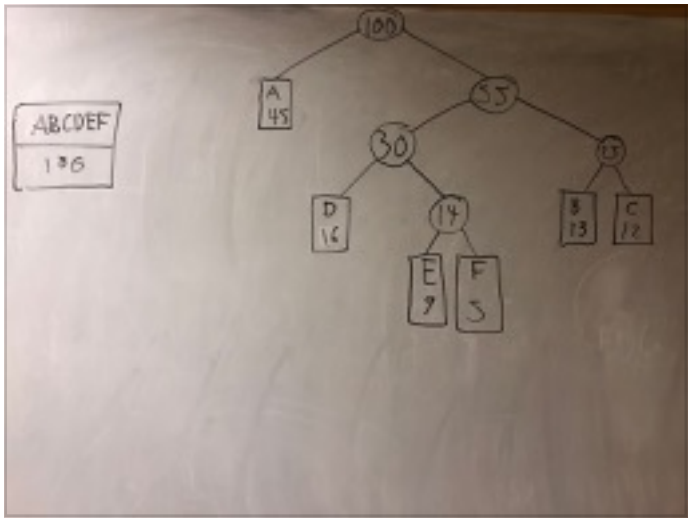


- Then we recurse!

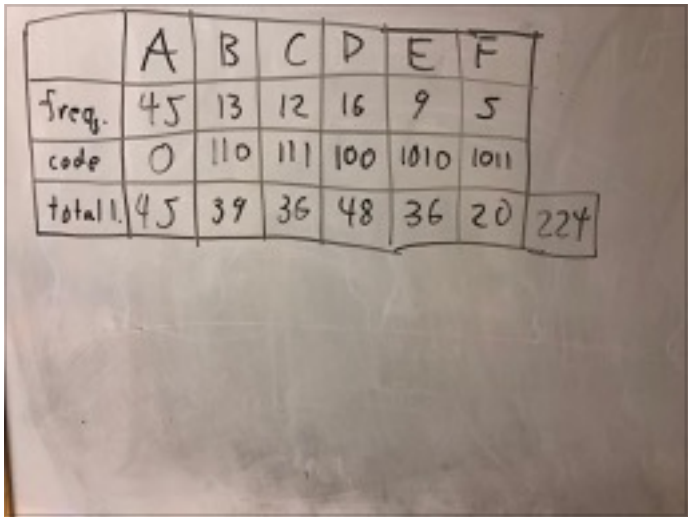








- So if the message began with CAFE, we would encode it as 111 0 1011 1010.
- Here's a table showing frequency, the code word, and the total number of bits needed to encode that character through the whole message. The entire message takes 224 bits to encode, and we cannot do better with any other prefix-free code.



- So, why is it optimal?
- Claim: Let x and y be least and second least frequent characters, respectively. There is an optimal code tree in which x and y are siblings.
  - Let T be an optimal code tree with depth d. T must be full as we can just shortcut any

node that has one child. There are two leaves at depth d. Let a be one of them.
    - Let T' be the code tree obtained by swapping x and a. The depth of x increases by some value Delta, but the depth of a decreases by Delta. So cost(T') = cost(T) - (f[a] - f[x]) Delta. Since x was at least tied for least frequent character, the cost could have only decreased.
    - Let b be the sibling of what is now x and swap y and b. y is no more frequent than b, so again the cost could have only decreased.
- Claim: Huffman codes are optimal prefix-free binary codes.
    - If n ≤ 2 then the claim is trivially true.
    - Let f[1 .. n] be the input frequencies, but assume f[1] and f[2] have the smallest frequencies. For simplicity, let f[n+1] = f[1] + f[2].
    - Some optimal code tree has characters 1 and 2 as siblings.
    - Let T be a code tree for f[1 .. n] where 1 and 2 are siblings, and let T' = T \ {1, 2}. T' is a code tree for f[3 .. n+1].
    - cost(T) =
        - sum_{i=1}^n f[i] * depth(i)
        - sum_{i=3}^{n+1} depth(i) + f[1] * depth(1) + f[2] * depth(2) - f[n + 1] * depth(n+1)
        - cost(T') + f[1] * depth(1) + f[2] * depth(2) - f[n+1] * depth(n+1)
        - cost(T') + (f[1] + f[2]) * depth(T) - f[n+1] * (depth(T) - 1)
        - cost(T') + f[1] + f[2]
    - So given that 1 and 2 are siblings in T, we just want to optimize T', which our algorithm does by induction.
- We can build the code tree using a min-heap which stores nodes and their frequencies. You can store nodes or extract nodes of minimum frequency. We'll use three arrays of length 2n - 1: L[i] is the left child of node i. R[i] is the right child. P[i] is the parent. The root is the node with index 2n - 1.
- BuildHuffman(f[1 .. n]):
    - for i ← 1 to n
        - L[i] ← 0; R[i] ← 0 leaves have no children
        - Insert(i, f[i]) store node i with frequency f[i]
    - for i ← n to 2n - 1
        - x ← ExtractMin()
        - y ← ExtractMin()
        - f[i] ← f[x] + f[y]
        - L[i] ← x; R[i] ← y
        - P[x] ← i; P[y] ← i
        - Insert(i, f[i])
    - P[2n - 1] ← 0
- If we use a balanced binary tree as the min-heap, then it takes O(log n) time to do each

heap operation. There are O(n) operations total, so the total time to build the tree is O(n log n).

- Here's some algorithms to encode and decode messages. We'll use A[1 ..] as the array of characters and B[1 ..] as the array of bits encoding them.
- HuffmanEncode(A[1 .. k]):
  - m ← 1
  - for i ← 1 to k
    - HuffmanEncodeOne(A[i]) write a single code word
- HuffmanEncodeOne(x):
  - if x < 2n - 1
    - HuffmanEncodeOne(P[x])
    - if x = L[P[x]]
      - B[m] ← 0
    - else
      - B[m] ← 1
    - m ← m + 1
- HuffmanDecode(B[1 .. m]):
  - k ← 1
  - v ← 2n - 1
  - for i ← 1 to m
    - if B[i] = 0
      - v ← L[v]
    - else
      - v ← R[v]
    - if L[v] = 0
      - A[k] ← v
      - k ← k + 1
      - v ← 2n - 1