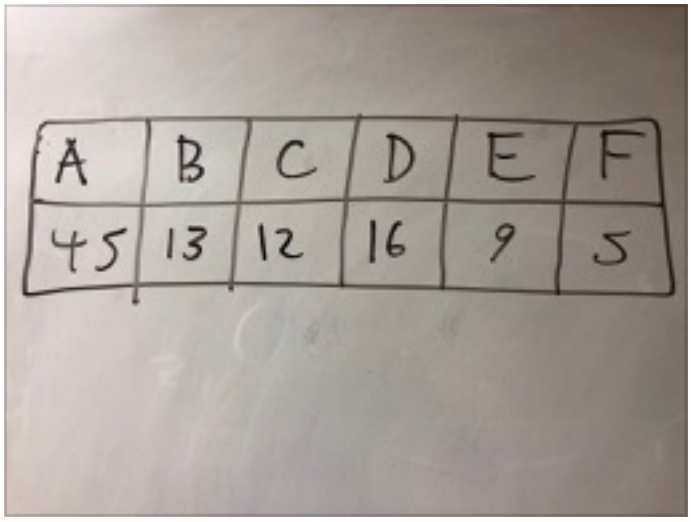
CS 4349.003.19F Lecture 11—September 25, 2019

Main topics for #lecture include #greedy_algorithms including #example/Huffman_codes.

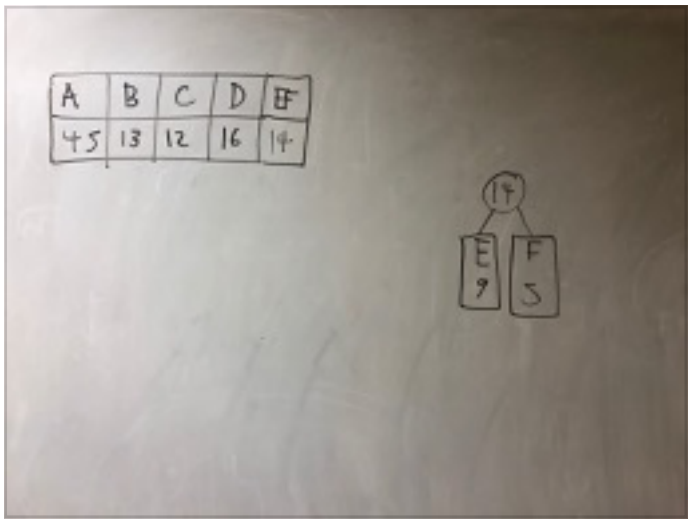
Huffman Codes

- Monday, we looked at two examples of greedy algorithms: storing files on magnetic tape in increasing order of size, and finding a maximal conflict-free schedule of classes by taking the class that ends earliest and recursing.
- Correct greedy algorithms are much rarer than correct dynamic programming algorithms, so you should always be suspicious if you think one will work. In fact, there are so few good examples of greedy algorithms that the confict-free scheduling example appears in almost every introductory lecture on the topic.
- Today, we'll see the other example that appears in most introductory lectures.
- 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 for 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. However, a non-prefix free code could be confusing. What if I'm typing Morse code very very slowly. You wouldn't know if I'm still writing an S or if I wrote a couple E's.
- 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 works under the assumption 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.
- Now, suppose somebody has written a long message in some alphabet. We want to find some prefix-free binary code for that alphabet's characters so that the encoded message uses as few bits 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.

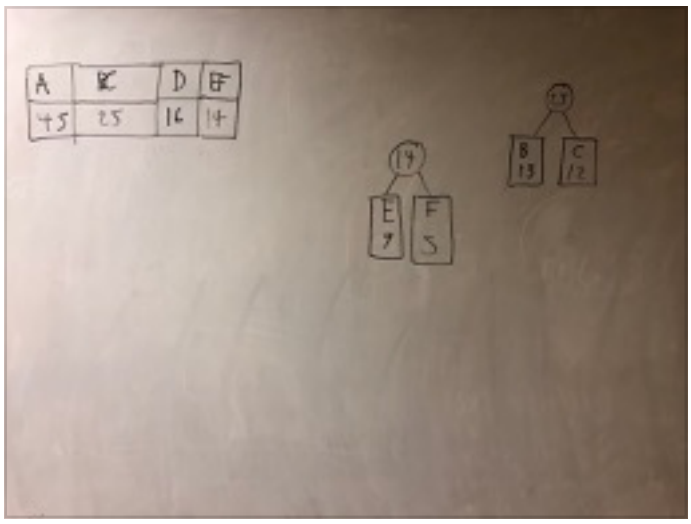
- We'll focus on building the tree since it's easier to illustrate.
- So, our goal is to compute a binary tree that minimizes \sum_{i=1}^n f[i] * depth(i).
- The difficult thing about this problem is that it's not even clear what a good backtracking approach would be.
- Our goal is to create a binary tree. Binary trees consist of a root with 0 to 2 children, each of which is the root of a smaller binary tree.
- So the natural strategy for a backtracking approach would be to guess a root and the way it splits up the characters and then recursively build binary subtrees.
- But there's no order to the nodes. So we'd essentially be trying to guess an entire *subset* of nodes to include in one side or the other. There's just way too many options to make even a reasonable backtracking algorithm with this approach.
- However, there's another way to think about binary trees recursively. First off, we should observe that the optimal tree is a full binary tree, meaning every node has 0 or 2 children.
    - If you have a node with one child, you can shorten the codes for everything in the subtree by connecting directly to the grandchild instead.
- If the tree is full, then every node at maximum depth is a leaf, so every leaf at maximum depth has a leaf sibling.
- An alternative recursive structure for full binary trees is a pair of leaf nodes connected as children to the leaf of a smaller full binary tree.
- So one backtracking strategy would be to guess which two characters share a common parent.
- Suppose we knew two leaves that should be siblings. Every path to *either* sibling goes through their parent node. In a way, the parent node is kind of like a character that appears when either of the siblings appear in the message. I'll formalize this idea a bit later.
- So, if we know two sibling leaves, we should *merge those characters* by treating the parent as a single character that appears with the sum of their frequencies, and then recurse on the smaller alphabet.
- In 1951, a Ph.D. student named David Huffman proposed the following greedy strategy based on this idea:
    - Set the two least frequently used characters as siblings.
    - To compute the rest of the tree, merge those 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:

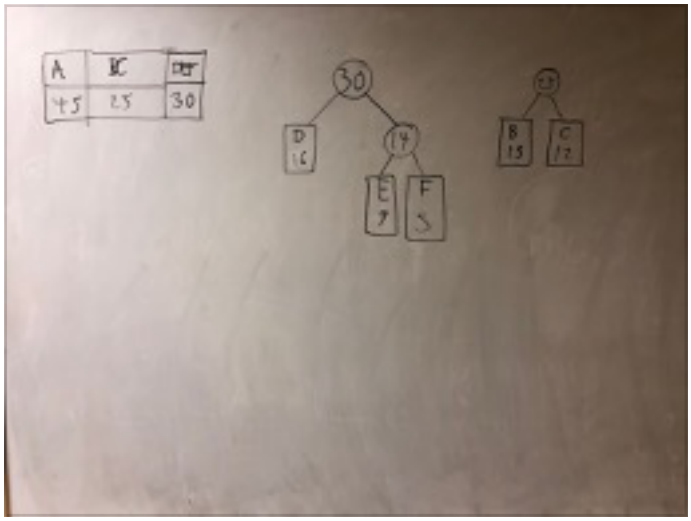


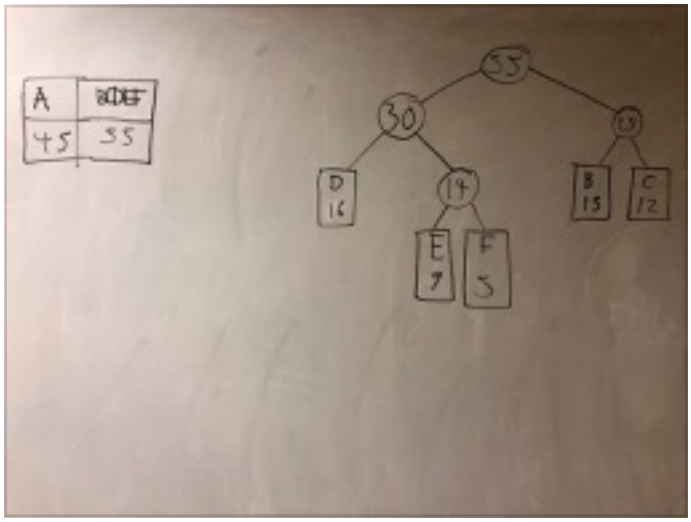
- The least frequent characters are E and F, so we'll make them sibling leaves and treat their parent node as a merged single character EF. EF becomes an internal node of the tree with children E and F.

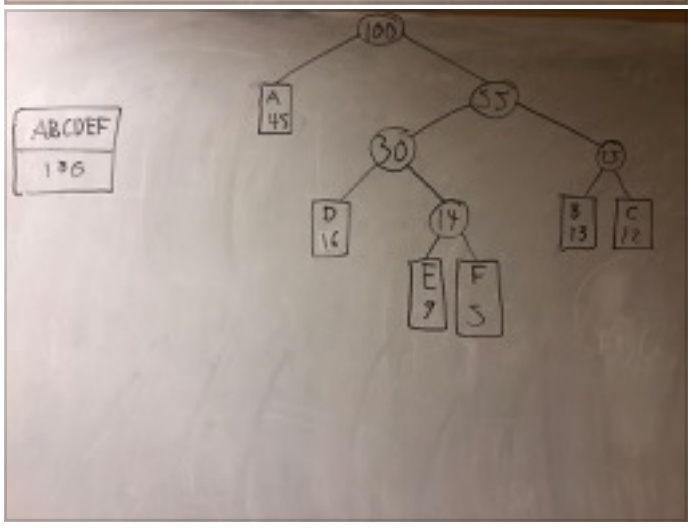


- Then we recurse!









- So if the message began with CAFE, we would encode it as 111 0 1011 1010.
- But this is a greedy algorithm. So to prove our code's optimality, we need an exchange argument.
- Lemma: Let x and y be the two least frequent characters (breaking ties arbitrarily). There is an optimal code tree in which x and y are siblings.
  - I'll actually show that x and y are siblings *and* they have the largest depth of any leaf. We'll assume x is a least frequently appearing character.
  - Let T be an optimal code tree with depth d.

- There are a pair of leaves a and b at depth d. Suppose x is neither a nor b.
- Swap leaves a and x to get tree T'.
- cost(T')
  - = cost(T) + f[x](depth(a) - depth(x)) - f[a](depth(a)-depth(x))
  - = cost(T) + (f[x] - f[a])(depth(a) - depth(x)).
- But f[x] - f[a] is non-positive and depth(a) - depth(x) is non-negative, so cost(T') ≤ cost(T) + 0 = cost(T).
- So there's an optimal tree T' where x has maximum depth.
- But now suppose x's (new) neighbor b is not equal to y.
- Now we swap y and b to make T''. The exact same algebra shows the cost(T'') ≤ cost(T') ≤ cost(T).
- But now we have an optimal tree T'' with x and y as neighbors (of maximum depth too).

- Theorem: Huffman codes are optimal prefix-free binary codes.
  - If n = 1 or n = 2 then the theorem is trivially true because you have no choices.
  - Otherwise, let f[1 .. n] be the input frequencies, but assume f[1] and f[2] have the smallest frequencies to keep the algebra clean(ish).
  - Some optimal code tree has characters 1 and 2 as siblings.
  - Let T be any code tree for f[1 .. n] where 1 and 2 are siblings, and let T' = T \ {1, 2}.
  - For simplicity, we'll treat the parent of characters 1 and 2 as character n + 1. We'll use f[n+1] := f[1] + f[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(1) - f[n+1] * (depth(1) - 1)
    - = cost(T') + f[1] + f[2] + (f[1] + f[2] - f[n+1]) * (depth(1) - 1)
    - = cost(T') + f[1] + f[2]
  - Since f[1] and f[2] are fixed, cost(T) is minimized when cost(T') is minimized. And by the induction hypothesis, cost(T') is minimized by recursively building Huffman codes for T'.
- We can build the code tree using a priority queue which stores nodes and their frequencies.
- The priority queue needs to support two operations: Insert(e, k) which inserts element e with key k and ExtractMin() which retunrs the element with minimum key from the priority queue while also removing it from the queue.
- We'll use three arrays of length 2n - 1: L[1 .. 2n - 1], R[1 .. 2n - 1], and P[1 .. 2n - 1] where L[i] is the left child, R[i] is the right child, and P[i] is the parent of node i. I'll use 0 as an element to represent no child or parent. The root is the node with index 2n - 1. **[second loop**

should go from n + 1 to 2n - 1]

```
BUILDHUFFMAN(f[1..n]):
  for i ← 1 to n
      L[i] ← 0;  R[i] ← 0
      INSERT(i, f[i])
  for i ← n to 2n − 1
      x ← EXTRACTMIN( )      ⟨⟨find two rarest symbols⟩⟩
      y ← EXTRACTMIN( )
      f[i] ← f[x] + f[y]     ⟨⟨merge into a new symbol⟩⟩
      INSERT(i, f[i])
      L[i] ← x;  P[x] ← i    ⟨⟨update tree pointers⟩⟩
      R[i] ← y;  P[y] ← i
  P[2n − 1] ← 0
```

- If we use a min-heap as the priority queue, then it takes O(log n) time to do each queue operation. There are O(n) queue 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. Both algorithms take O(m) time where m is the length of the *encoded* message.

```
HUFFMANENCODE(A[1..k]):
  m ← 1
  for i ← 1 to k
      HUFFMANENCODEONE(A[i])

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
```