CS 6363.005.19S Lecture 10—February 14, 2019

Main topics are #dynamic_programming and #greedy_algorithms with and #example/maximum_independent_set_in_trees, #example/class_scheduling, and #example/Huffman_codes.

Prelude

- Homework 2 is due Tuesday, February 19th.

Dynamic Programming on Trees

- Let's finish up dynamic programming on trees before we move on to another topic.
- Recall, and *independent set* of a tree is a subset of nodes with no edges between them. We want to compute the size of the *maximum independent set* in a tree T with n vertices.
- Let MIS(v) denote the size of the maximum independent set in the subtree rooted at v. Let w (downarrow) v mean "w is a child of v". We derived the following recurrence for MIS(v). **[First recursive call should be on w]**

$$MIS(v) = \max\left\{\sum_{w\downarrow v} MIS(v),\ 1 + \sum_{w\downarrow v}\sum_{x\downarrow w} MIS(x)\right\}$$

- There's another way we could have solved this problem so that we don't have to worry about grandchildren.
- If we take the root into our independent set, we cannot include the roots of the children subtrees. So we could ask for the maximum independent subsets of the children subtrees that *don't* include their roots.
- Similarly, we could ask for maximum independent sets that *must* include roots.
- So let MISyes(v) denote the size of the maximum independent subset of the subtree rooted at v that *includes* v.
- And MISno(v) is defined the same except we exclude v.
- Now we just want to know max{MISyes(r), MISno(r)}, and we can use the following recurrence.

$$MISyes(v) = 1 + \sum_{w\downarrow v} MISno(w)$$

$$MISno(v) = \sum_{w\downarrow v} \max\{MISyes(w), MISno(w)\}$$

- Time to memoize.
  - Subproblems: Each recursive subproblem takes a node.
  - Memoization: Again, we don't make a new array. Instead, we'll use the tree itself by

storing each MISyes(v) and MISno(v) in new fields v.MISyes and v.MISno.

- Dependencies: Each entry MISyes(v) or MISno(V) depends upon the children of v.
- Evaluation order: So we'll use a standard *post-order* traversal of the tree.
- Space and time: We're storing two numbers per vertex, so we use **O(n)** space. Time is more subtle. The time taken to compute MISyes(v) and MISno(v) for each vertex varies depending on how many children it has. So let's turn the analysis around. Each vertex contributes a constant amount of time to the evaluation of its parent's MISyes and MISno. There's at most one parent per vertex, so the algorithm will run in **O(n)** time.

- And here's the algorithm.**[Do MIS(w)'s separately in their own loop to make the postorder clear]**
- Nearly all the details are the same as before, but we get a simpler dynamic programming algorithm.

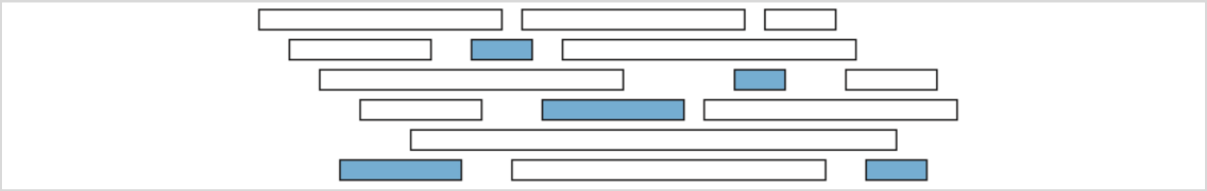
```
MIS(v):
  v.MISno ← 0
  v.MISyes ← 1
  for each child w of v
      v.MISno ← v.MISno + MIS(w)
      v.MISyes ← v.MISyes + w.MISno
  return max{v.MISyes, v.MISno}
```

Class Scheduling

- In dynamic programming algorithms, we consider a sequence of decisions, like do I include this vertex in my independent set? For each choice we can make, we tentatively make the choice and have the Recursion Fairy deal with the consequences. Then we go with the best choice.
- But rarely, and I do mean *rarely*, you can figure out the best choice *before* you do the recursive call. In other words, you figure out some best choice, *commit to it* and then have the Recursion Fairy deal with the consequences.
- Again, these situations are rare, and algorithms designed in this way always require a very careful proof of correctness. It's too easy to get these things wrong!
- I'll go over one or two examples in class today and Tuesday. And to further emphasize how rare these situations are, I'll use the same two examples that *everybody* uses when introducing greedy algorithms. More examples will come as we look at graphs.
- By the way, greedy algorithms are not a subject of the first midterm, but they will be featured in Homework 3.
- For the first example, let's suppose you're picking classes for next semester, and your number one goal is to graduate as quickly as possible, so you want to take as many courses as possible. Don't worry, these classes you're considering don't require any actual work; you just need to be present for the lecture. By some fluke all classes next semester

are scheduled on Mondays only, but you're not allowed to take classes scheduled for conflicting times.

- Formally, let S[1 .. n] be the start time of all n courses offered and F[1 .. n] be their end times; more concretely S[i] < F[i] for all i.
- You want to find a *maximal conflict-free schedule* which is a maximum size subset X of {1, 2, …, n} such that for each i,j in X either S[i] > F[j] or S[j] > F[i].
- Another way to think about it is you have this set of overlapping intervals representing the time span for each course. Find the largest subset of intervals that don't overlap.



- If we were designing a backtracking or dynamic programming algorithm, we'd consider some first interval and try both possibilities of including the class or not, letting the Recursion Fairy deal with the consequences.
- But today, let's just pick a good class to take, commit to it, and recurse.
- We'll greedily take the class that finishes first.
- We can write this iteratively by scanning through the class list ordered by finishing time, and every time we see a new class that doesn't conflict with the last one we chose, taking it. This procedure returns the indices of the classes sorted by finishing time.

```
GREEDYSCHEDULE(S[1..n], F[1..n]):
  sort F and permute S to match
  count ← 1
  X[count] ← 1
  for i ← 2 to n
      if S[i] > F[X[count]]
          count ← count + 1
          X[count] ← i
  return X[1..count]
```

- We need to sort by finishing time, which we can do in O(n log n) time using a merge sort. We also have that O(n) time for loop, so the whole thing takes O(n log n) time.

Exchange Arguments

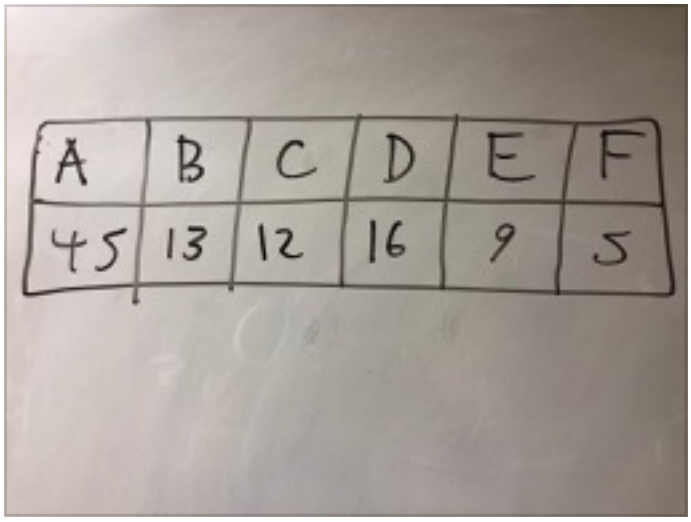
- To prove that this greedy algorithm works, we'll use something called an exchange argument.
- In short, we consider some optimal schedule. If it doesn't include our first choice of class, then we'll exchange our first choice for one of the classes in the optimal solution without decreasing its quality.
- Lemma: At least one maximal conflict-free schedule includes the class that finishes first.
  - Let f finish first, and consider any maximal conflict-free schedule X.
  - If X includes f, we're done.

    - Otherwise, let g be the first class to finish in X.
    - f finishes before g, so f does not conflict with any classes in X \ {g}.
    - So remove g and replace it with f. The new schedule is just as large and it agrees with our first choice.
- Next, we use induction just as we would with any other recursive strategy.
- Theorem: The greedy schedule is optimal.
    - If there were no classes, then the empty schedule would be optimal.
    - So let's assume the greedy schedule is optimal for n' < n classes.
    - We just proved there is an optimal schedule that uses the same first choice as greedy. Given that choice, we cannot take conflicting classes, so our goal is to find a maximal conflict-free scheduled from the non-conflicting classes. Which the greedy algorithm does by our induction hypothesis.
- So again, we followed this proof structure:
    1. Start with *some* optimal schedule. Maybe it doesn't include our first choice.
    2. Exchange our first choice for something chosen by the optimal schedule while staying optimal.
    3. Use induction to argue that the remaining choices are correct.

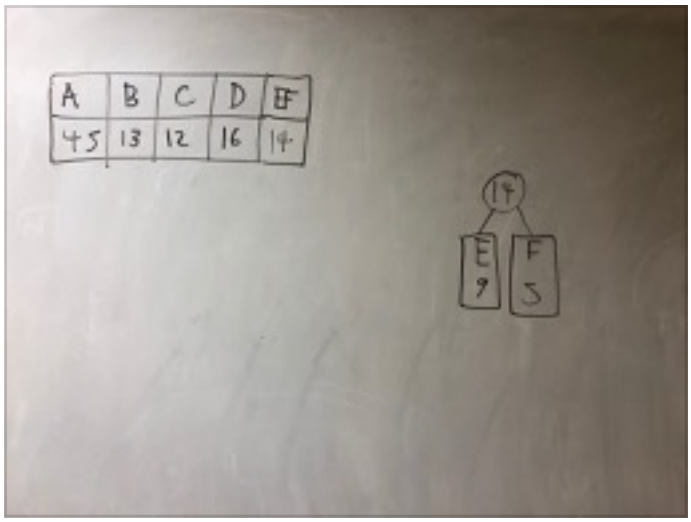
Huffman Codes

- Let's start with the other example where greedy algorithms work. We'll finish with this example next Tuesday.
- 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.
- 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.

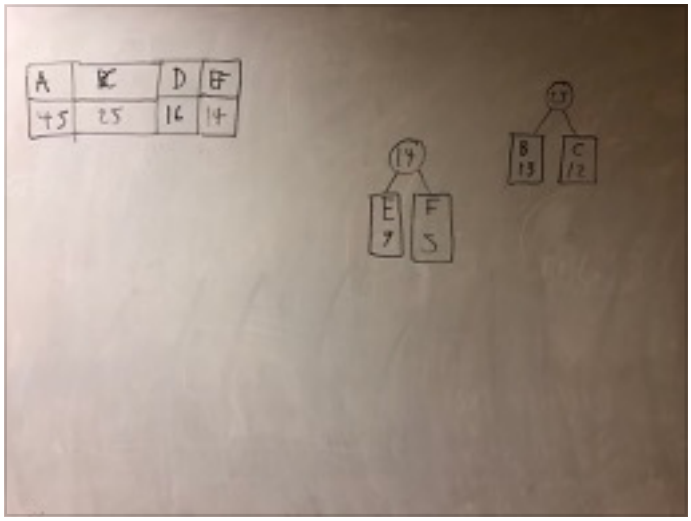
- We want to encode an alphabet so that a given encoded message is as short as possible.
- Formally, we are given an array f[1 .. n] of frequency counts where f[i] is the number of times character i appears.
- Our goal is to compute a prefix-free binary code and its corresponding binary tree to minimize \sum_{i=1}^n f[i] * depth(i).
- We'll focus on building the tree since it's easier to illustrate.
- At this point, you may think we've already solved this problem. However, Tuesday's problem involved building a *binary search tree*. The order of the elements within the tree had to be preserved. Here, the order does not matter, only the total cost. Also, the current problem requires all characters to appear at the leaves of the tree.
- This lack of order actually makes it much harder to do backtracking.
- For example, we earlier tried to decide what the root should be and divide things into left and right subtrees. Two researchers named Fano and Shannon offered a greedy algorithm for the problem based on this approach. There's no clear way to divide the characters into two pieces, so they would split the frequency table into two subarrays as evenly as possible, construct code trees for the two subarrays independently, and then attach them to a common root.
- It made intuitive sense, but it wasn't optimal, so Fano proposed the then open problem of constructing optimal prefix-free codes in an information theory class of his.
- 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.
- And 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, *merge the characters* by treating the parent as a single character that appears with the sum of their frequencies, and then recurse on the smaller alphabet.
- One of Fano's students in the class, David Huffman, proposed the following greedy strategy based on this idea:
  - 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:

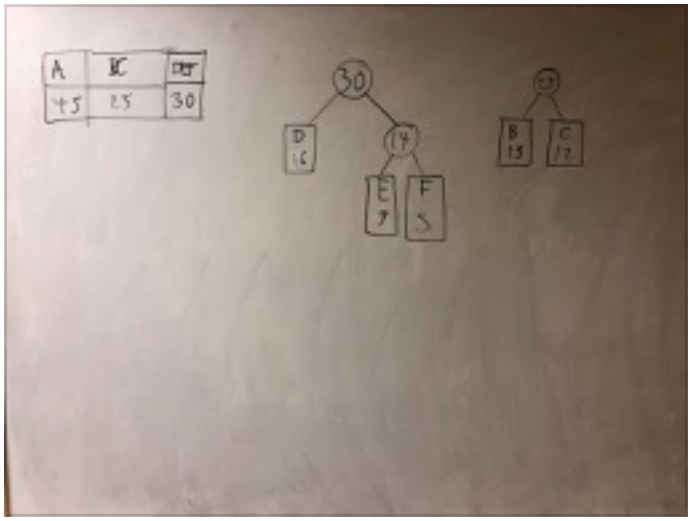


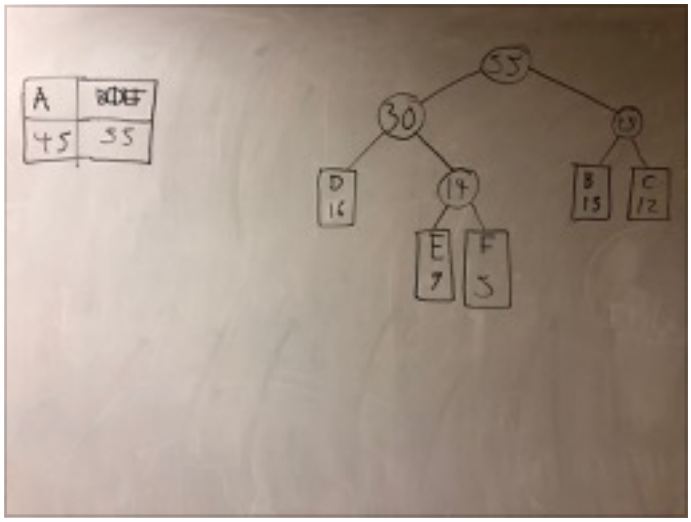
- The least frequent characters are E and F, so we merge them into a single character EF. EF becomes an internal node of the tree with children E and F.

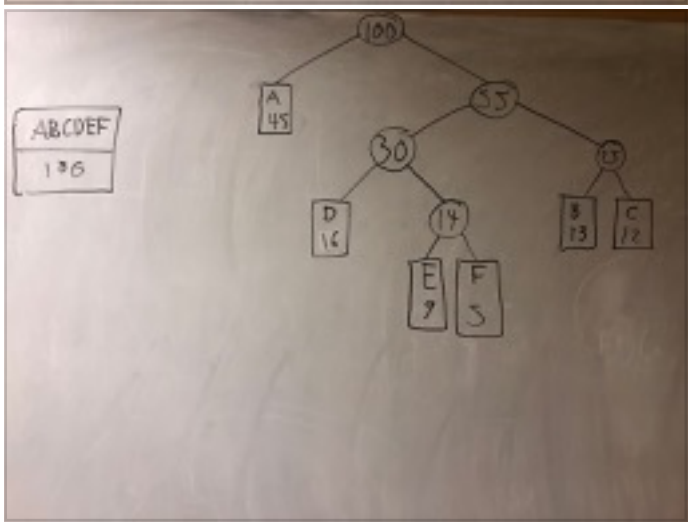


- Then we recurse!

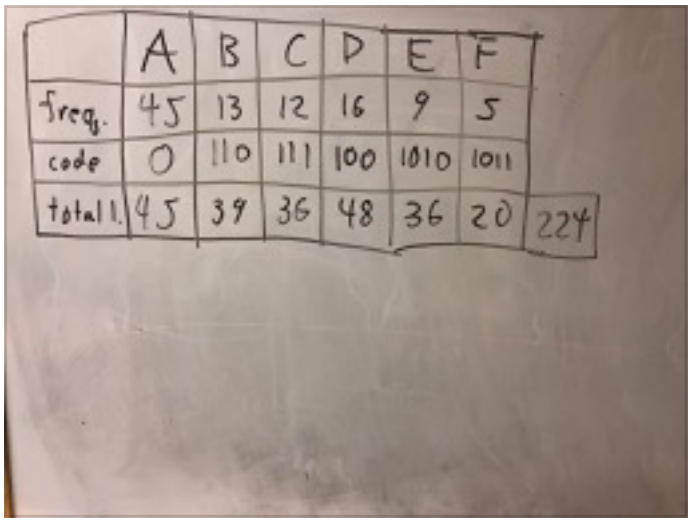








- So if the message began with CAFE, we would encode it as 111 0 1011 1010.
- Here's a table showing frequency, the code word, and the total number of bits needed to encode that character through the whole message. The entire message takes 224 bits to encode, and we cannot do better with any other prefix-free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



	A	B	C	D	E	F	
freq.	45	13	12	16	9	5	
code	0	110	111	100	1010	1011	
total	45	39	36	48	36	20	224

- But to prove 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.
  - Let T be an optimal code tree.
  - Suppose x and y are not siblings.
  - Without loss of generality, assume the depth of y is at least the depth of x.
  - Let a be the other sibling of y, so depth(a) ≥ depth(x).
  - Exchange a for x to get tree T'.
  - cost(T') = cost(T) - (f[a] - f[x])(depth(a) - depth(x)).
  - But f[a] - f[x] and depth(a) - depth(x) are both non-negative, so (f[a] - f[x])(depth(a) - depth(x)) ≥ 0 meaning cost(T') ≤ cost(T).
- Theorem: Huffman codes are optimal prefix-free binary codes.
  - If n = 1 then the theorem is trivially true.
  - For larger n, assume the theorem is true for alphabets of size n' < n.
  - Let f[1 .. n] be the input frequencies, but assume f[1] and f[2] have the smallest frequencies.
  - 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(2) - f[n+1] * depth(n+1)
    - cost(T') + (f[1] + f[2]) * depth(1) - f[n+1] * (depth(1) - 1)
    - cost(T') + (f[1] + f[2]) * depth(1) - (f[1] + f[2]) * (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. 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. **[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.

```
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
```