Greedy Algorithms - Part 2

▼ What's the definition of the Huffman Coding problem?

- Lets say we have an alphabet with A[1…n] characters, and we know the characters' frequencies F[1…n] of each character.

Example:

Example:

A	a	b	c	d	e	f	g	h	i	j
F	9	2	5	6	12	3	4	7	8	1

- We want to give binary codes to each character (a variable-length code).
- To encode the message, we just replace the character with the binary, then to decode we replace the binary with the character.
- Larger frequency items have shorter binary codes, and smaller frequency items have longer binary codes.

▼ What's the prefix property of Huffman Coding?

- No character's code is a prefix of any other character's code. The binary must be unique.
- Example: `10` = a, and `1011` is an e.
    - We don't know which thing this is.

▼ How are Huffman encodings built?

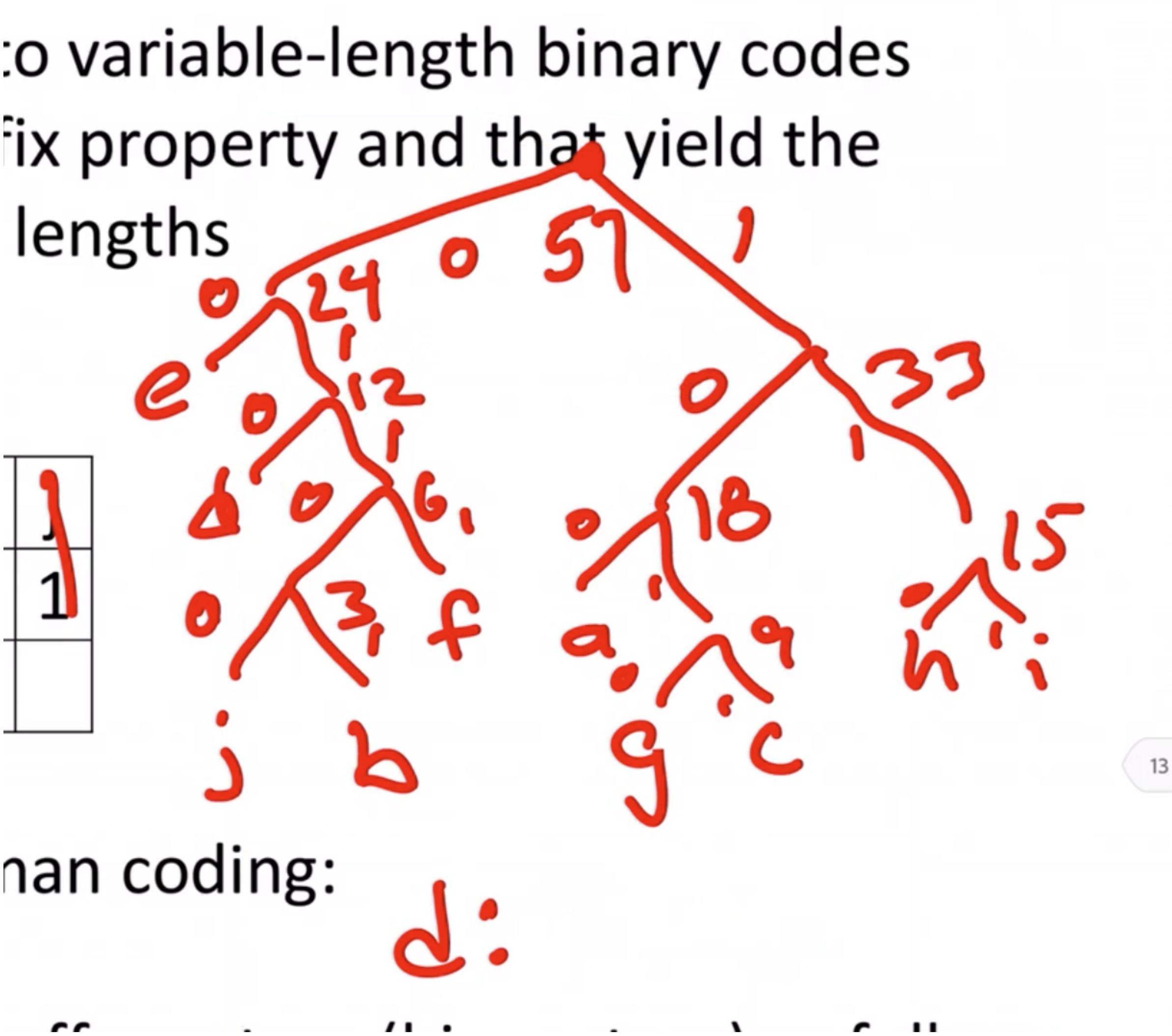
- They're built as a tree, a Huffman tree.

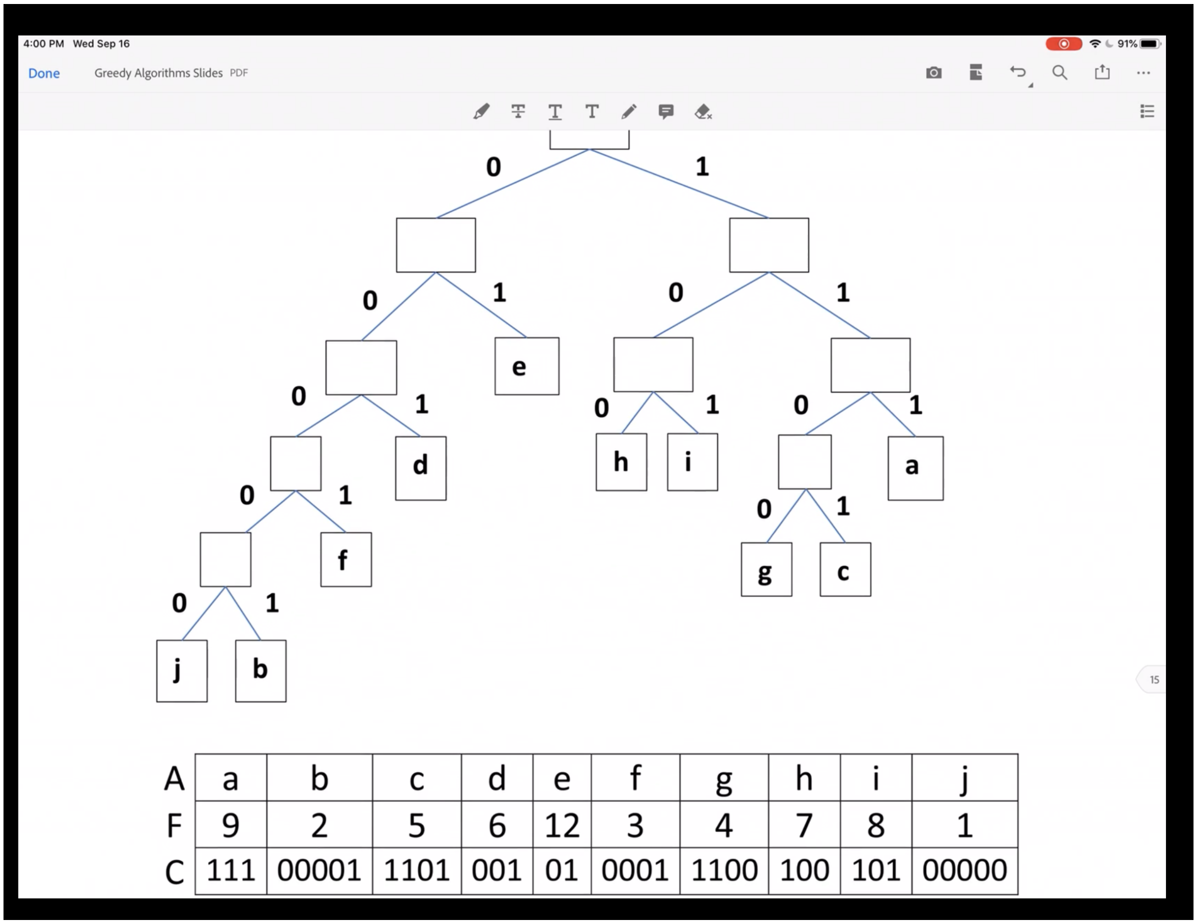
(1) First construct a Huffman tree (binary tree) as follows:

```
H = new Heap( );                // min-ordered heap
for j = 1 to n
      H.insert (new Leaf(F[j], A[j]));        // F[j] is key
while (H.size( ) >= 2) {
      left = H.removeMin( );
      right = H.removeMin( );
      sum = left.key + right.key;
      H.insert (new Node(sum, left, right));  // sum is key
}
root = H.removeMin( );
```

- We create a left node, a right node, and then we insert that node into the tree. The *sum* of this is key.
- We combine the smallest things (building this as a min heap) and then continue putting this back into the heap to build the tree.

- How do we get the encodings for each character from a Huffman Tree?
    - We label each branch of the tree with a zero or a one, and then we just traverse the tree down to each letter we're looking for.


:o variable-length binary codes
fix property and that yield the
lengths
57
24
33
12
18
15
e
d
f
a
h
i
j
b
g
c
nan coding:
d:
13



- What's the definition of the task scheduling problem?
  - We have a list of tasks {1...n} , but the tasks take the same unit of time to perform.
  - Each of these tasks have a deadline D[1...n], and we have a penalty P[1...n] that way pay if we don't get them done in time
  - Our goal is to minimize the total penalty cost for this problem.
  - If the task is late, we incur a cost.
- First greedy algorithm for task scheduling
  - Let's try scheduling tasks in order of increasing deadline.
  - Break ties in order of descending P[k]

  Example:

	1	5	3	4	2	6
D	1	2	2	3	3	5
P	5	7	3	10	8	4

S	0	1	2	3	4	5
F	1	2	3	4	5	6

- Which tasks are late here? Tasks 3, 4, 2, and 6 are late here.
- The total penalty for this is 25.
- This is not the optimal solution and is an incorrect algorithm.

▼ Second greedy algorithm

- Let's schedule things in order of descending penalty, scheduling the highest-penalty job first.

Example:

	4	2	5	1	6	3
D	3	3	2	1	5	2
P	10	8	7	5	4	3

S	0	1	2	3	4	5
F	1	2	3	4	5	6

- Tasks 5, 1, and 3 are late.
- The total penalty is 15.
- This is not the optimal solution and is therefore an incorrect algorithm.

▼ Optimal greedy solution

- Let's pick up jobs in the order of highest penalty first, but here's the difference: Let's place the task in the latest interval we can before its deadline hits, and then lets place tasks that don't fit into that as *late as we possibly can.*

Example:

	4	2	5	1	6	3
D	3	3	2	1	5	2
P	10	8	7	5	4	3

S	2	1	0	5	4	3
F	3	2	1	6	5	4

- This always creates the optimal solution.