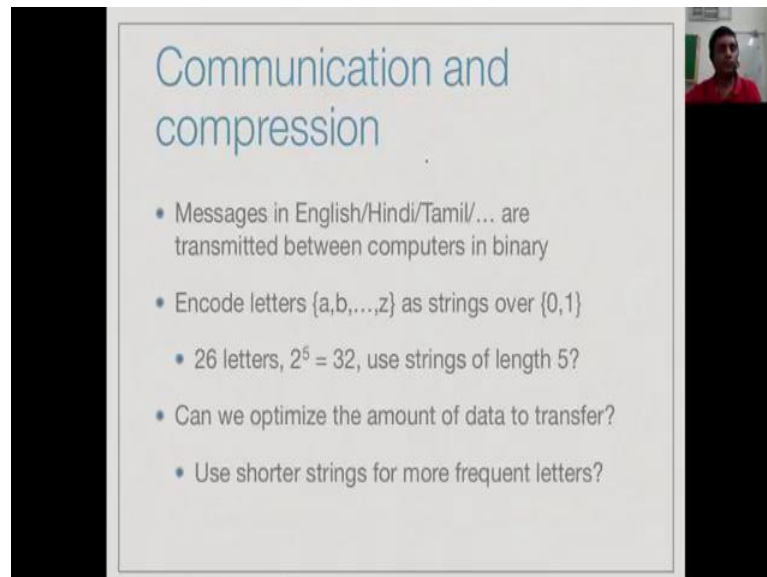


**Design and Analysis of Algorithms, Chennai Mathematical Institute**  
**Prof. Madhavan Mukund**  
**Department of Computer Science and Engineering,**

**Week - 06**  
**Module - 05**  
**Lecture - 43**  
**Greedy Algorithms: Huffman Codes**

For the last example of a greedy algorithm in this course, we will look at a problem communication theory, we will look at the problem of Huffman Codes.

(Refer Slide Time: 00:10)



The slide is titled "Communication and compression" in a light blue font. It contains a list of five bullet points in a dark blue font. The first bullet point states that messages in English/Hindi/Tamil are transmitted between computers in binary. The second bullet point asks to encode letters {a,b,...,z} as strings over {0,1}. The third bullet point notes that with 26 letters,  $2^5 = 32$ , strings of length 5 are used. The fourth bullet point asks if we can optimize the amount of data to transfer. The fifth bullet point suggests using shorter strings for more frequent letters. A small video inset in the top right corner shows a man in a red shirt.

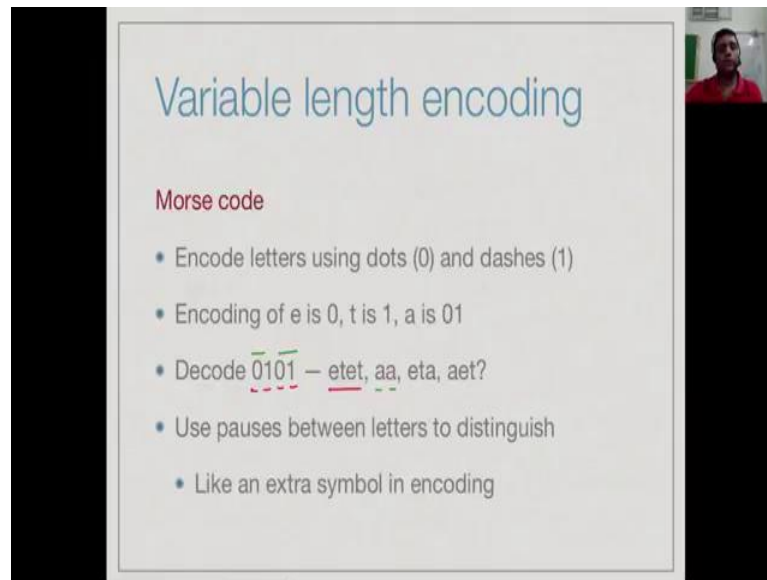
- Messages in English/Hindi/Tamil/... are transmitted between computers in binary
- Encode letters {a,b,...,z} as strings over {0,1}
  - 26 letters,  $2^5 = 32$ , use strings of length 5?
- Can we optimize the amount of data to transfer?
  - Use shorter strings for more frequent letters?

So, when we communicate, we have to transmit information from one place to another place. So, we might be working in some language like English, Hindi or whatever, but if we were using computers for example, to transmit our data, we know that they must send this information in binary strings. So, our typical goal is to take an alphabet, and then encoded it over strings of 0 and 1, so that at the other end, we can decoded and recover the message.

So, if you have say the 26 lower case letters a to z, then it is easy to see that we need to if you want to encode each letter as a fixed sequence of 0's and 1's by fixed length, then we will need to use 5 bits for letter, because if you use only 4 bits, we can only get 16 different combinations, with 5 bits we can get 32 different combinations. So, now a natural question is, can we do something clever about using different length encoding for

different letters, so that more frequent letters get send with shorter inputs. So, can we optimize the amount of data we actually transfer in order to send the message from one place to another?

(Refer Slide Time: 01:24)



Variable length encoding

Morse code

- Encode letters using dots (0) and dashes (1)
- Encoding of e is 0, t is 1, a is 01
- Decode 0101 — etet, aa, eta, aet?
- Use pauses between letters to distinguish
- Like an extra symbol in encoding

So, this brings us to the idea having a variable length encoding, where we use different strings of different length for different letters in the alphabet. So, one of the most famous examples of the variable length encoding is the classical Morse code, which is developed by Samuel Morse from the telegraph who is invented. So, this was done using a mechanical device by clicking on a contact and it will produce long and short clicks. So, the short clicks are called dots and the long clicks called dashes, we can as well think of them as representing the bits 0 and 1.

So, in the Morse code encoding, different letters do have different encodings and in English e is the most frequent letter and t is another variant frequent letter. So, Morse assigned them codes of a dot, that is 0 for e and a dash, that is 1 for t, then a Morse took other frequent letters, such as a and gave them two letter encodings. So, a is encoded as dot dash, where 0 and 1.

Now, the problem with Morse's encoding is that it is ambiguous, when you come to decoding. So, for instance, if we look at the word, the sequence 0 1, then we do not know whether we should interpret each of these as a one letter code and get e t e t, all for instance we should think of this as 2 two letter of codes and get a a and so on. So,

depending on whether we stop it is 0 or extends 0 to 0 1, we can get many different interpretations.

Now, in practice in Morse code, what we use to happen is that the operator gives a slide pause indicating the end of the letter. So, therefore, effectively Morse code is not a binary code, but it is a three letter code 0 1 and pause, now we are using of course, digital computers, we do not want to go to three letters. So, we want to efficiently do these using just two letters.

(Refer Slide Time: 03:20)

**Variable length encoding**

**Prefix code**

- Encoding  $E(\cdot)$ ,  $E(x)$  is not a prefix of  $E(y)$  for any  $x, y$
- In Morse code  $E(e) = 0$  is a prefix of  $E(a) = 01$
- Example:  $\{a, b, c, d, e\}$
- Decode 0010010011101

x	a	b	c	d	e
$E(x)$	11	01	001	10	000

Decoding example: 0010010011101  
 c e c a b

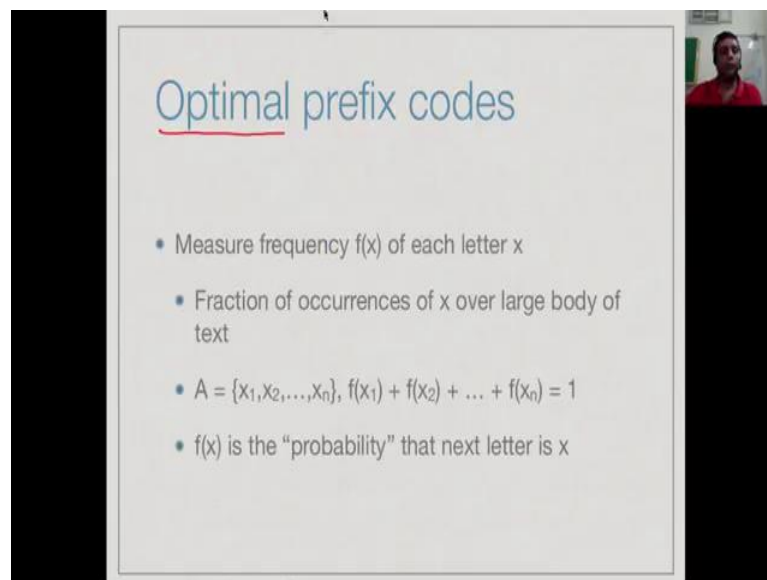
So, in order to make a variable length code an unambiguous decodable, we need what is called a prefix quantity. When we read through a sequence of 0's and 1's, we should be an ambiguously clear, whether we have read a letter or there is more to read. We should be like the earlier case, where we have read 0 and we do know, whether we stop at 0 and call it an e in the Morse code setting or we want to call it an a which is 0 1.

So, we are going to use this capital letter E to denote the encoding the function. So, E of x is the encoding of the letter. So, here is an example, so we have five letters a, b, c, d, e and now, you can check that none of these encodings can be extended to anything. I do not have, if I see 1 1, it must be an header, no other code which starts with 1 1. If I see 0 0 it is not a code, but 0 0 1 is a code, so 0 1 cannot be extend and so on. So, each of these cannot be extended to be the code of any other letter. So, now when we get along

sequence like this, there is no doubt, the first point when I completed a letter is 0 0 1 and this is a c.

The next point when I complete the letter is three 0's and it is an e, then I read another c and then an a and then a b. So, if we have the prefix code property, that is no letter is encoded to a string which is the prefix of the encoding of some other letter, then we have unambiguous decoding possible and this is very important.

(Refer Slide Time: 04:50)



The slide is titled "Optimal prefix codes" in blue text, with "Optimal" underlined in red. It contains a list of four bullet points:

- Measure frequency  $f(x)$  of each letter  $x$
- Fraction of occurrences of  $x$  over large body of text
- $A = \{x_1, x_2, \dots, x_n\}, f(x_1) + f(x_2) + \dots + f(x_n) = 1$
- $f(x)$  is the "probability" that next letter is  $x$

A small video inset in the top right corner shows a man in a red shirt speaking.

So, our goal is to find optimal prefix codes. So, we need to talk about what we mean by optimality. So, remember we said that our goal is to assign shorter codes to more frequent letters. So, somehow we have to determine, what are more frequent and less frequent letters? So, people have measure the frequency of the occurrence of each letter and different languages, so this is a very language specific thing.

So, this optimality is something which is optimal for English, may not work of French or any other Spanish or something. So, you take a large body of text in a particular language and you count the number of a's b's c's d's and e's, and then you just look at the fraction, out of the total number of letters across all the steps, how many are e's, how many b's, how many are c's. So, this is a kind of statistical estimate of the average frequency of this.



So, this frequency would be a fraction, what fraction of the letter over a large body of text will be e's, what fraction will be c's and these fraction will add up to one, because every letter would be one of them. So, the fraction of a is plus, the fraction of b is plus, fraction of c is and so on is going to added to 1 and because of this, we can also think of this statistical estimate or a kind of probability.

Let if I give you a random letter, if I look at the piece of text and pickup a random letter from the text, what is the probability that x, well it is just be the frequency of x across all the text f of x. So, these will added to 1.

(Refer Slide Time: 06:16)

**Optimal prefix codes ...**

- Message  $M$  consists of  $n$  symbols  $f(x)$  of  $n$  are  $x$   
 $n \cdot f(x)$
- For each letter  $x$ ,  $n \cdot f(x)$  occurrences of  $x$  in  $M$
- Each  $x$  is encoded by  $E(x)$  with length  $|E(x)|$   $x \mapsto E(x)$   
010
- Total length of encoded message:  $n \cdot f(x) \cdot |E(x)|$ 
  - Sum over all  $x$ ,  $n \cdot f(x) \cdot |E(x)|$   $\sum_{x \in A} x \cdot f(x) \cdot |E(x)|$
  - Average number of bits per letter  $\sum_{x \in A} f(x) \cdot |E(x)|$
  - Sum over all  $x$ ,  $f(x) \cdot |E(x)|$

So, now, we have a message, it consists of some  $n$  symbols. So, we have  $M_1, M_2$  up to  $M_n$ , so these are  $n$  symbols. Now, we know that if I take a particular letter  $x$ , then  $f(x)$  fraction of these are  $x$ , so in other words, if I take  $n$  and I multiply by a fraction is say, if I fix is say one third, then one third of  $n$ . So,  $n$  by 3 of these symbols will actually be the letter  $x$  and now, each of these  $x$  is going to be represented by its encoding.

So, supposing it is 0 1 0 then each  $x$  is going to represent by 3 bits, so then  $n$  into  $f(x)$  is the number of times  $f(x)$  and this into the length of this encoding is going to give me the total number of bits taken to encode all the  $x$ 's in this message. Now, if I do this for every letter, so if I take the summation over every  $y$  or every  $x$  in my alphabet of  $n$  times the frequency of the letter times the encoding length of that letter.

So, this tells me how many bits I need to encode that particular letter, add up all the letters, I get the total length of the encoded message. And if I do not include this n, it is no to said n it is not a part of the summation, it is an independent thing, it is a fraction of any n. If I just look at the total weighted average of two links of the encodings, then this is if you study probability theory, what is called the expected length of the encoding. So, this is the average number of bits, I use for a letter.

(Refer Slide Time: 08:06)

**Optimal prefix codes ..**

- Suppose we have these frequencies for our example

x	a	b	c	d	e
E(x)	11	01	001	10	000
f(x)	0.32	0.25	0.20	0.18	0.05

- Average number of bits per letter is

$$0.32 \cdot 2 + 0.25 \cdot 2 + 0.20 \cdot 3 + 0.18 \cdot 2 + 0.05 \cdot 3$$

Handwritten calculation:  $\sum_{x \in A} f(x) \cdot |E(x)|$

Result: **2.25**

- Fixed length encoding uses 3 bits per letter
- 25% saving using variable length code

So, let us work out how this, so suppose we take our earlier example of 5 letters, now we insert some fictitious information about frequencies. So, this all these five values are fraction between 0 and 1, you can added to 1. Then if I do this summation over x of f of x times the length of E of x, then I have 0.32 times 2, because the encoding of A is two letters, then I have 0.25 times 2, because the encoding E is two letters and so on.

So, I have these five terms a, b, c, d, e and then I added it up and I get 2.25, so it says that I need an average 2.25 bits per letter. Of course, I do not use 2.25 bits per letter, but what it says this for instance, if I have a 100 letters, I would expect to see 225 bits in the output encoding. Now, a very specific kind of prefix code is the fixed length code, where just by the fact that every code to the fixed length, I know exactly where each one th.

So, supposing I use 3 bits in this case, if I want to fix length code of this, then there are five letters, I cannot do it with 2 bits, because I only get four different combinations. So, I need 3 bits, if I use some 3 bit code, then every 3 bits will be one letter. So, in the fixed

length encoding in this, I will use 3 bits for letters, so therefore clearly the number of bits per letter is 3, because I am using 3 for every one of them. And so by going to a variable length code encoding which takes in to upon the frequency and actually savings it is to a sending 300 bits for 100 character, it send into 225. So, we have 25 percent saving, so this is what we are trying to get at.

So, in this example in the previous thing we have two frequencies 0.2 and 0.18, the 0.18 means d is less frequency than c, but somehow we assigned c to be a longer code. So, this violated our basic principle that shorter code should be assigned to more frequent letters. So, if you see a pair of letters which are where one is more frequent than other, I expected to get a shorter code and I am not done so.

(Refer Slide Time: 10:18)

Optimal prefix codes ..

- A better encoding

x	a	b	c	d	e
E(x)	11	10	01	001	000
f(x)	0.32	0.25	0.20	0.18	0.05

- Average number of bits per letter is
  - $0.32 \cdot 2 + 0.25 \cdot 2 + 0.20 \cdot 2 + 0.18 \cdot 3 + 0.05 \cdot 3$
  - 2.23 ABL
- Given a set of letters A with frequencies, produce a prefix code that is as efficient as possible
- Minimize ABL(A) — Average Bits per Letter

So, if I invert that, so supposing I now assign a three letter code to d and a two letter code to c, then these two terms change the other terms, though the encoding may have differ, the length do not change. So, then I get instead of 2.25, I get on to 2.23, so what this say is that looking at different encodings I could get different A B L values, this average bits per letter. So, now, our goal is to find an assignment capital E which minimizes this quantity. So, in our coding the average efficient is possible.

(Refer Slide Time: 10:56)

### Codes as trees

- Encoding can be viewed as a binary tree
- Path to a node is a binary string—left is 0, right is 1
- Label each node by the letter it encodes
- Prefix code: only leaves encode letters

x	a	b	c	d	e
E(x)	11	01	001	10	000

So, to get to this, it is useful to think of these encodings as binary trees, so in a binary tree I can interpret directions as 0 and 1, so typically left is 0 and right is 1. So, now, if I read of a path in a binary tree, it will also be a binary sequence. So, this path is 1 1 1 on the right for example and this path programs the 0 1 and this path is 0 0 0. Now, if I read of a path and then I find the letter of that label, then it is as good as saying that path labels that letter is encoded by that path.

So, here is because 'e' has the encoding 0 0 0 in the binary tree, I will follow the path 0 0 0 and label that corresponding letter at that vertex by 'e'. Now, because it is a prefix code, if 0 0 0 labels 'e', they will not be any code which says 0 0 0 and something more, they will not be another label below it. So, these labels will not extend to other label, I will not find an 'f' below 'd', because otherwise it is not a prefix code, I do 1 0 and I get a 't', but I do 1 0 something else, then I get an 'f'. So, the code for 'f' extends the code for 'd', this is not enough. So, in a prefix code this cannot happen, so therefore, all the labels are actually at leaf nodes, there are nodes which have more successes.

(Refer Slide Time: 12:23)

### Codes as trees

- Encoding can be viewed as a binary tree
- Path to a node is a binary string—left is 0, right is 1
- Label each node by the letter it encodes
- Prefix code: only leaves encode letters

x	a	b	c	d	e
E(x)	11	10	01	001	000

So, here is an encoding for the other scheme that we had, where we exchanged the values are c and d. So, now c has a two letter code and d has a three letter code, this is indicated by the in depth now, the depth this is path, the length of the path is the depth of the node. So, c in are earlier thing was a depth 3, and now it is a depth 2 and the d was depth 2 and now it is a depth 3. So, we want to basically put thing which have higher frequencies, higher up in the tree at lesser than.

(Refer Slide Time: 12:59)

### Codes as trees ...

- **Full tree:** Every node has 0 or 2 children

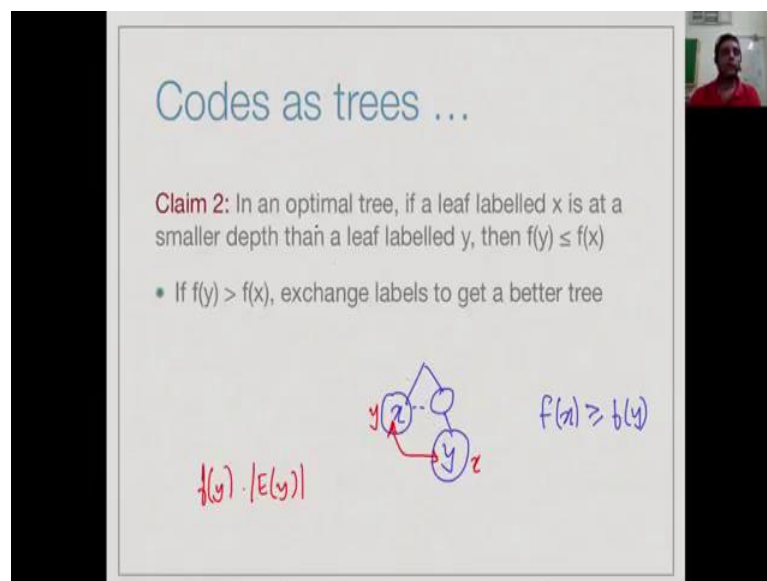
**Claim 1:** Any optimal prefix code generates a full tree

- If any node has only one child, we can promote its child and create a shorter tree

So, having encoded look at are encoding the binary tree, we will now make a couple of observation, three observation of bottom which will be useful prove to develop an optimal algorithm and prove it is optimal. So, the first thing is that in such a tree, if it is optimal, every node will either have no children will we a leaf or it will have two children. So, this is what we called a Full.

So, every optimal tree is full, now is easy to see this, because the supposing the claim, we other optimal tree in which somewhere in between, we had a node which had only one child. Then, this child can effectively will be promoted, we can remove this node completely and we can string the tree along this direction a nothing will change, except that the depth of the node is below become less. So, in fact, we will possibly get a shorter average bit length then we had, therefore by having a Singleton, either only a left child or right child, we cannot the optimal. So, the every node must either 0 or two children.

(Refer Slide Time: 14:11)



Codes as trees ...

**Claim 2:** In an optimal tree, if a leaf labelled  $x$  is at a smaller depth than a leaf labelled  $y$ , then  $f(y) \leq f(x)$

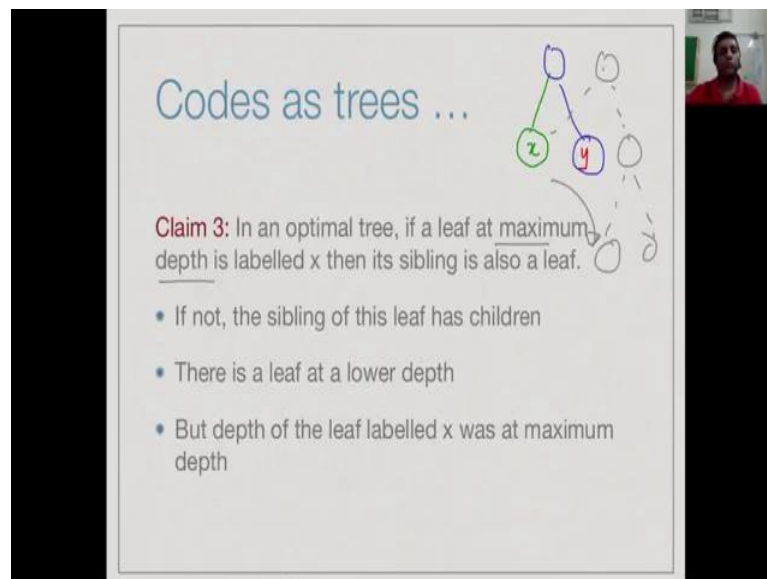
- If  $f(y) > f(x)$ , exchange labels to get a better tree

The diagram shows a tree structure with two nodes,  $x$  and  $y$ , where  $x$  is at a shallower depth than  $y$ . Handwritten notes include  $f(x) > f(y)$  and  $f(y) \cdot l(y)$ , indicating the condition for an exchange to improve the tree.

The next property is exactly what we saw the earlier thing, which is that, if I have two nodes  $x$  and  $y$ , such that,  $x$  is higher than  $y$ , so  $x$  is at some level and  $y$  is different level. Then,  $x$  has a shorter encoding then  $y$ , this must mean that  $f$  of  $x$  is at least as much  $f$  of  $y$ , in other word, when I go down the tree my frequency cannot increase, because if  $f$  of  $y$  would bigger than  $f$  of  $x$ , that if I more wise an  $x$  is a mentax. Then, I will just exchange is 2, I would find a better encoding by putting  $y$  here and  $x$  here.

Because, now if I do  $f$  of  $y$  time the length of  $y$  and the depth  $y$  in this tree, then it to reduce, because the depth of  $y$  is reduce and this is the higher multiply. So, therefore, if I had a higher thing below, then I could exchange the letter and get better tree and that does not happen in then optimal tree, so then the optimal tree, if I go down the tree, I only find lower frequency letters.

(Refer Slide Time: 15:09)



Codes as trees ...

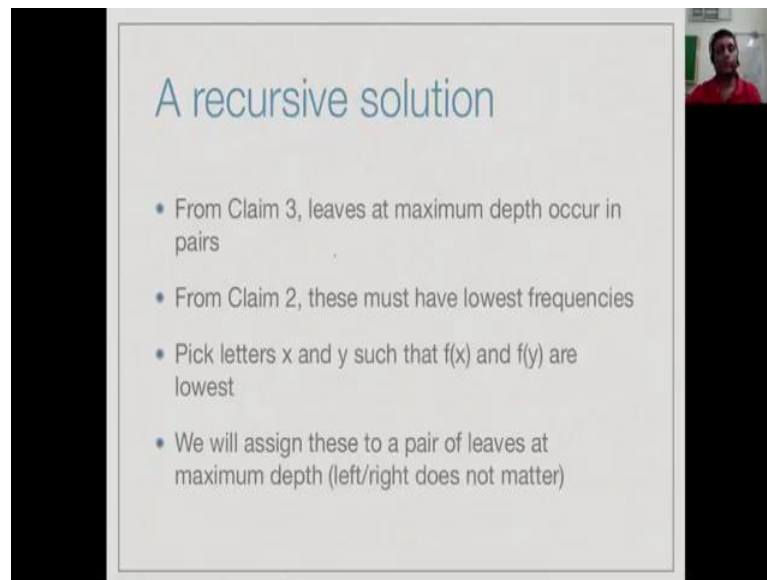
**Claim 3:** In an optimal tree, if a leaf at maximum depth is labelled  $x$  then its sibling is also a leaf.

- If not, the sibling of this leaf has children
- There is a leaf at a lower depth
- But depth of the leaf labelled  $x$  was at maximum depth

The final property is to do with leaves at the lowest level, so supposing I have the leaf at the lowest leaf, so this is some leaf of a lowest level. So, I know because it is an optimal tree, it cannot be a isolated child, it must have a sibling for go and come down, it must have a sibling. Now, there are two possibilities, the two possibilities are this is a leaf or the other possibility is that, this is not a leaf, the claim if that if it is not a leaf, then it must have children.

So, then there are leaves here which have at the lower level, then  $x$ , but  $x$  is assumed to be a maximum depth  $d$ . So, maximum depth leaf cannot have sibling, which is not a leaf, because it sibling it would have a children, which have to higher depth. So, therefore, if I have a maximum depth leaf in my optimal tree, then we need occur is a pair with another maximum depth leaf.

(Refer Slide Time: 16:19)



### A recursive solution

- From Claim 3, leaves at maximum depth occur in pairs
- From Claim 2, these must have lowest frequencies
- Pick letters  $x$  and  $y$  such that  $f(x)$  and  $f(y)$  are lowest
- We will assign these to a pair of leaves at maximum depth (left/right does not matter)

And so this is a conclusion in that leaves of maximum depth occurred in pairs and then we know that because frequencies keep of dropping as we go down increasing in depth, these pairs must have a lowest frequency among the lowest frequencies. So, in order to develop the solution, we will use recursion, so what we will do is, we will say, let us look in the overall table that we start with and pick two letters, which have the lowest frequency.

So, we can assign them the longest codes, so they can be put at the lowest level and then we know that the lowest level leaves at the pairs. So, let us assume that these will be paired out, so we will assign these lowest frequency letters  $x$  and  $y$ , to a pair of leaves maximum depth, left and right does not matter, because so the depth that matters.