Information Theory and Coding
Prof. S. N. Merchant
Department of Electrical Engineering
Indian Institute of Technology, Bombay

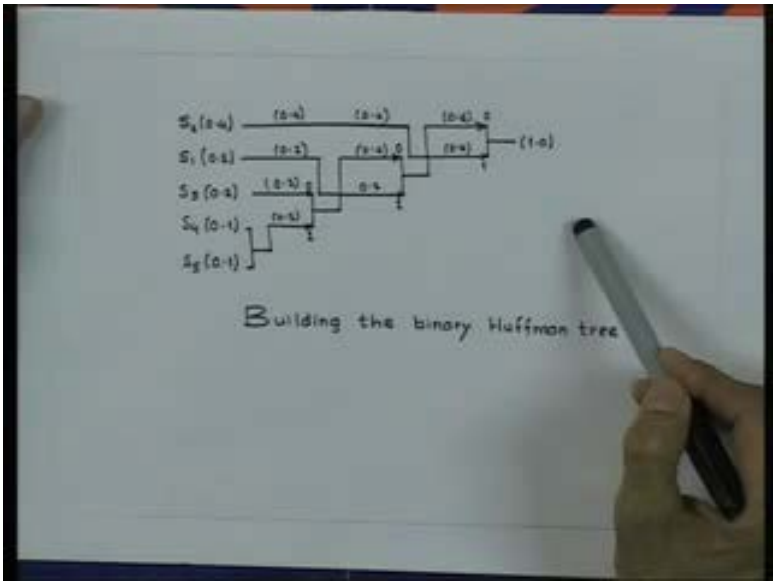
Lecture - 12
Huffman Coding and Proof of its Optimality

In the previous class, we have had a look at the construction of the Huffman code. In today's class, we will have a look at an alternative way of building the same Huffman code. Huffman code by virtue of being a prefix code can be represented by a binary tree wherein the source numbers are represented by the external node or leaves. To get a Huffman code for a particular symbol, we traverse from the root to the leaves corresponding to the source symbol. We keep on adding 0 to the code word every time we traverse on the upper branch of the tree and add a 1 to the code word every time, we traverse on the lower branch of the tree.

So, let us look at the procedure of building a Huffman code based on the tree concept. Now, we start the building of this Huffman tree from the leaf node. Now, we know that the code word corresponding to the two symbols, which have the smallest probability, are identical except for the last bit; what this means that the path from the root node to the leaves, corresponding to this symbols will be the same except for the last step. This implies that the leaf corresponding to these symbols, which have the smallest probability, will be off springs of the same intermediate node. So, what we can do is connect this 2 leaves corresponding to the 2 symbols with smallest probability into another single node.

Treat this single node as a new symbol in the reduced alphabet. Now, the probability of this new symbol in the reduced alphabet will be the probability of the off springs. So, it is the sum of the probabilities of the off springs. Now, for the nodes corresponding to the reduced alphabet, apply the same rule to generate a parent node for the nodes corresponding to the 2 symbols in the reduced alphabet with lowest probability. So, continuing in this manner, end up in a single node, which is known as the root node. Once we get the tree and then to get a code for a symbol, traverse the tree from the root to each node to each leaf node assigning a 0 to an upper branch and a 1 to the lower branch. Let us try to look at this procedure with the help of an example, which we had considered earlier in the class.

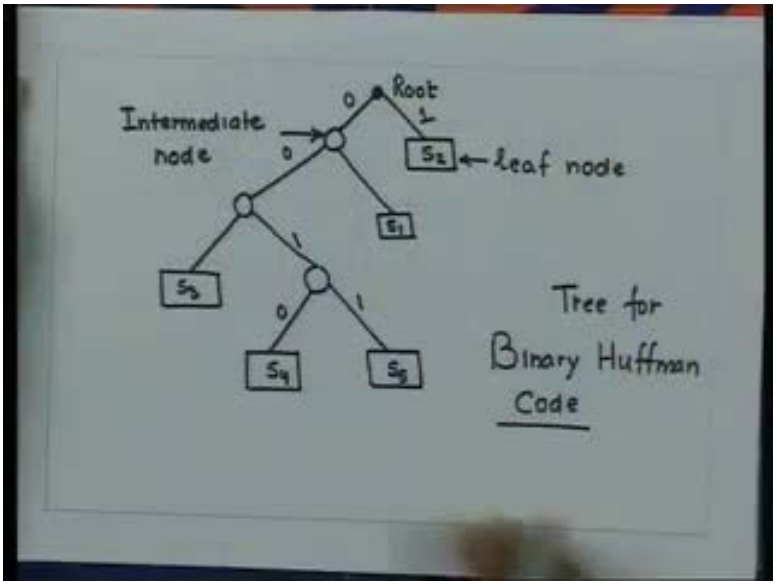
(Refer Slide Time: 04:26)



Now, I have the source consisting of five symbols. These symbols correspond to my node. Initially, I sort these nodes in the decreasing order of probability. Doing so will ease the construction of this tree. So, take these two nodes, which correspond to this two source symbols S 4 and S 5. They are having the lowest probability. Combine these two nodes and get a parent node. This parent and this node are treated as a new symbol in this new alphabet, which I get. The probability of the symbol is the sum of the probabilities of the off springs. So, 0.1 plus 0.1, I get 0.2. Now, I get new symbols in the new reduced alphabet corresponding to the new node.

Again, combine the two nodes, which correspond to the least probabilities. So, combining 0.2 and 0.2, I get another node with the probability of the new probability of the new symbol to be equal to 0.4. So, I place it on a top of this node. I get a reduced alphabet with three nodes. Finally, I again combine this and this node and get a node with the probability of 0.6. Finally, you combine these two nodes to get a probability of 1. So, this is the procedure for building the binary Huffman tree. If you look at this procedure, this figure is similar to what we had designed in the last class using the concept of sorting in the list. Now, this same binary Huffman tree, which I got here can be redrawn in a more formal way as follows.

(Refer Slide Time: 06:52)



I have a root. From the root, I have one intermediate node and another node, which corresponds to a symbol. So, this is the more formal representation of the same binary Huffman tree. So, it is not difficult to convert this to this form, which is the more formal representation of a tree. Now, this is known as a root node. This is the leaf node or they are also known as external node. This corresponds to an intermediate node. This intermediate node is a parent for this two leaves. Then, this intermediate node is a parent for this node and an external node or the leaf node.

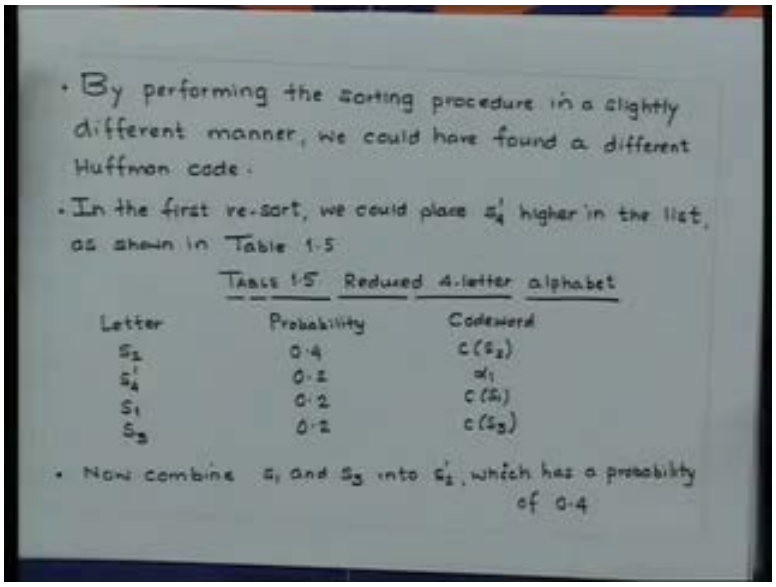
So, this is a grandparent of the leaf node S 4 and S 5. So, this is the formal representation of a binary Huffman; tree for binary Huffman code. At the moment, we are restricting our discussion to a binary alphabet. But this can be easily being extended to a code alphabet of size other than binary. Now, the next question that comes to our mind is that after I have got this Huffman code for this source. Now, for Huffman code for the source can be obtained from this binary tree also. Whenever I traverse on the left hand side, it is 0. Whenever I travel on the right hand side, it is 1.

So, if I want a binary code for S 4, it is 0 0 1 and 0. Similarly, the binary code for S 5 would be 0 0 1 1. So, the next question is that this Huffman code, which we have got for the source is it unique? One can, I design another Huffman code for the same source. Te answer is yes, a trivial extension or a trivial different version of the Huffman code, which

we have designed for this would be to get another Huffman code by inverting all the bits in the code word.

This we obtain from the Huffman code or by exchanging 2 code words of the same length. This is a trivial way of getting another Huffman code. But a little non trivial procedure would be to obtain a Huffman code by carrying out some kind of sorting in a slightly different manner of the symbols in the list for the alphabet which occurs at every step of the Huffman code. To understand this, let us again go back to our original example which we had taken in the last class.

(Refer Slide Time: 11:31)

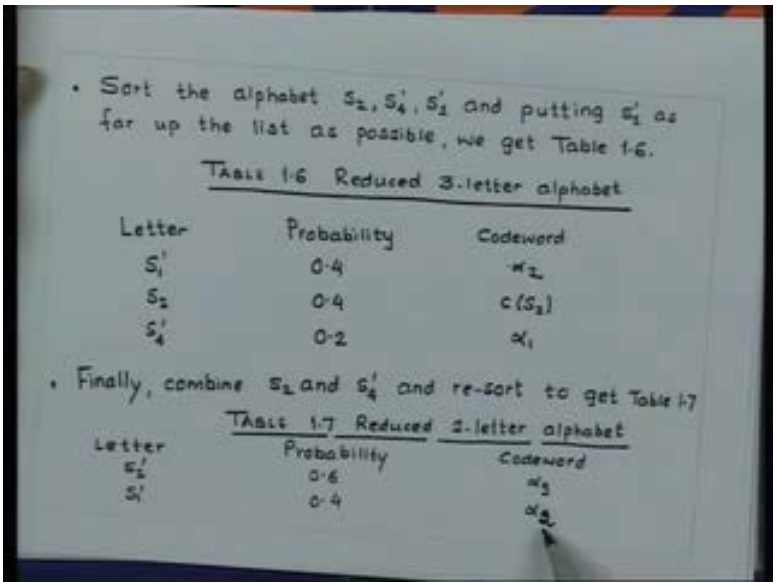


In the last class, we at the first step of the reduction, we had got four new symbols for the four letter reduced alpha S 2, S 4 dash, S 1 and S 3. In that case, we had placed S 4 dash at the bottom of the list. Now, what we do is basically instead of placing S 4 dash at the bottom of the list, we will try to place S 4 dash at the top of the list. So, this is what I mean by resorting of the list consisting of the symbols obtained for the reduced alphabet at every stage of the Huffman procedure. So, by performing the sorting procedure in a slightly different manner, we could have found a different Huffman code.

We try to do this with an example which we had discussed in the last class. So, in the first resort, which we get in the last class which we had seen, what we do? This time we place S 4 dash, which has been obtained as a new symbol by combining S 4 and S 5. We

then place S 4 dash on the top of the list. So, when I place S 4 on the top of the list, I get our new table, which is for a reduced 4 letter alphabet. Now, this is the topmost position, where I can place in the list because the probability of S 4 dash, S 1 and S 3 are all 0.2, 0.2, and 0.2. So, it does not make any difference where I place this alphabet. Sorry, this symbol S 4 dash. Now, you combine S 1 and S 3 into S 1 dash, which has a probability of 0.4. So, we carry out the same procedure which we had studied in the last class. So, you sort the alphabet S 2, S 4 dash, S 1 dash.

(Refer Slide Time: 14:07)

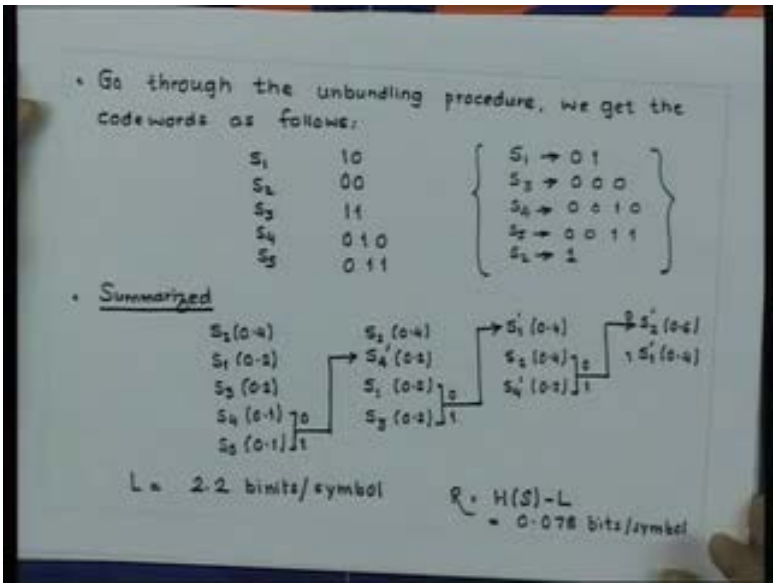


Now, you put S 1 dash as far up the list as possible. So, this is the strategy which we follow whenever we get a new symbol. We will try to place that new symbol in the reduced list at the top of the list. We will try our best to put it as far up the list as possible. So, when I combine S 2, S 4 dash and S 1 dash, when I have this alphabet, I place it like this; S 1 dash which has 0.4. In the previous class, we had placed S 1 dash below S 2. But I placed S 1 dash on the top of the list. I place it higher than S 2.

Again, we give the code words. Finally, we combine S 2 and S 4 dash to get a new symbol that is S 2 dash. Then finally, I get a reduced 2 letter alphabet and the code words for S 2 dash is alpha 3, S 1 dash alpha 2. For this reduced 2 letter alphabet, the obvious choice for the optimum allotment of the code word is 0 and 1. I allot alpha 3 as 0 and alpha 2 as 1. Once I have allotted 0 to alpha 3 and 1 to alpha 2, then I can reverse my process that is unbundling procedure and get the code word for my original symbols. So,

if we do that, what we get is the code word listed out here from S 1 to S 5. This is the code which I get when I follow the procedure.

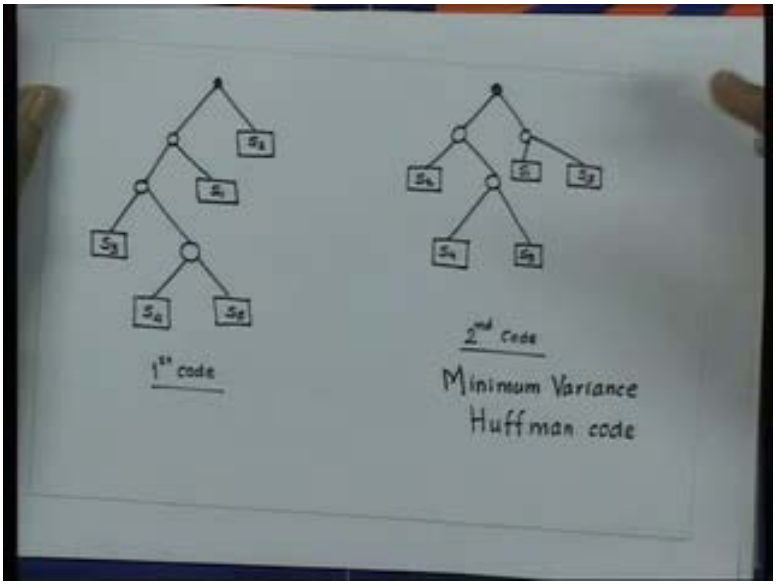
(Refer Slide Time: 15:54)



Now, this is the code which we had obtained in the last class. Now, the code which we have obtained today basically can be summarized in this figure. It is very easy to understand this figure. You have the sorting in the descending order. Combine these. Place in the top of the list in the reduced alphabet. So, I get this. Again, combine this and try to place it at the top of the list in the reduced alphabet. By placing at top S 1 dash, combine this. I get this.

Then, allot 0 1 and unbundle the follow, the unbundling procedure. Now, if we calculate the average code length for this code, which we have got, we will find that the average length turn out to be 2.2 binits per symbol. Their redundancy is defined as entropy minus average length for the code turn out to be 0.078 bits per symbol. So, in the code which we have today and the code which we had designed in the last class, both have the same average length. Both have the same redundancy. But there is a little difference between these 2 codes. To look at the difference, let us look at the binary tree for both of the codes.

(Refer Slide Time: 17:34)



So, on the left hand side, I have a binary tree representation of the first code. That is the code, which we designed in the last class. This is the code, which we designed today. So, if we call this first code and if you call this as second code, then you can calculate the variance for this code. You can calculate the variance for this code though the average lengths are same. You will find that the variance for this code, second code turns out to be lower than the variance of the first code. Whenever you design using the procedure which we have discussed, then that procedure of Huffman code designed is known as the code which we get from there is known as minimum variance Huffman code.

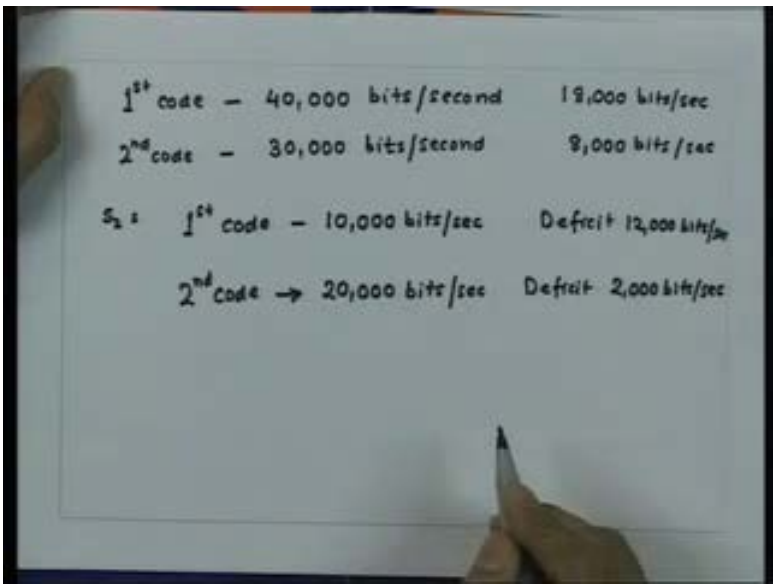
Now, which one of these 2 codes is a better one to understand? Let us look at the practical implication of minimum variance Huffman code. Now, in many applications although you might be using a variable length code, the available transmission rate is generally fixed. So, let us see what happened when this kind of a constraint. Let us take the same example of we have the source consisting of 5 symbols for which we have already discussed the 2 designs of the Huffman code in the form of first code and second code.

Now, let us say that we are interested in transmitting the source symbols on this code. Let us presume that this code generated 10,000 symbols per second. Now, we know that the average length of the both the codes which we have designed the first code and the second code is 2.2 bits per symbol or 2.2 bits per symbol. So, we can say that we would

require the transmission capacity of 22,000 bits per second. So, what it means that the transmission channels expects to receive 22,000 bits second from the source. Now, since we are using the variable length coding, the bit generation rate will not be constant.

The bit generation rate will vary around 22,000 bits per second. So, usually the output of such a source code is generally fed into a buffer. The purpose of the buffer is to smooth out the variation in the bit generation rate. However, the buffer has to be of finite size. Therefore, the greater the variance in the code word the more difficult the buffer design problem becomes. Now, let us look at the same source. Suppose, the source was to generate a string of S 4 and S 5 for several seconds, so if I have a sequence consisting of S 4 and S 5 existing for few seconds, then in that case, my first code would generate 40,000 bits per second.

(Refer Slide Time: 21:17)

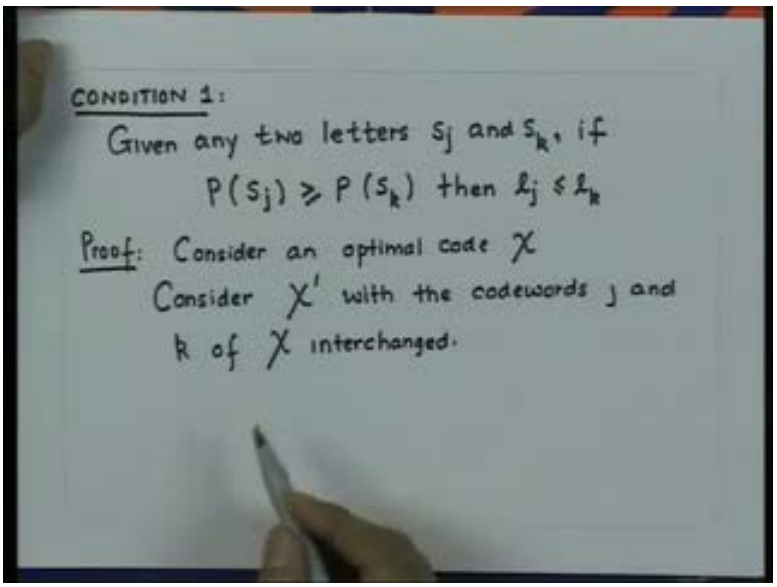


This is because the code word length for both S 4 and S 5 was 4 in the first code. We are generating symbols at the rate of 10,000 symbols per second. So, the channel is expecting 22,000 bits per second. So, the buffer has to code 18,000 bits per second. Now, if we had used the second code for transmission of the source symbol, then in that case since the code word length for both S 4 and S 5, using the second code is triple. Then, what it implies? The second code would be generating 30,000 bits per second. So, in this case, the buffer has to code 8,000 bits per second if the sequence process for some time.

Now, if the same source generates a string of say S 2's, in that case what will happen? The first code will generate 10,000 bits per second. Since the channel is expecting 22,000 bits per second, so we will have to make somehow, we will have to make up for the deficient of 12,000 bits per second. But if we use the second code and if we have the sequence of S 2's, then on the channel, we will have 22,000 bits per second. Since, the channel is expecting 22,000 bits per second, somehow the channel has to make up for the deficient 2,000 bits per second. So, looking at this example, it is reasonable to say that we should use the second code instead of the first code.

So, in a practical scenario, it is very important to go for minimum variance codes. So, when you are designing Huffman code, we should go for minimum variance Huffman code. The way to design minimum variance Huffman code is to see that we always put the combined letter as high in the list as possible. Now, after having load at the construction of the Huffman code, let us try to prove the optimality of Huffman code. Now, to prove the optimality of Huffman code, we will try to write down the necessary conditions that an optimal code has to satisfy. Then, show that satisfying this condition necessary need to designing the Huffman code. So, let us look into the necessary conditions, which are required for an optimal variable length binary code.

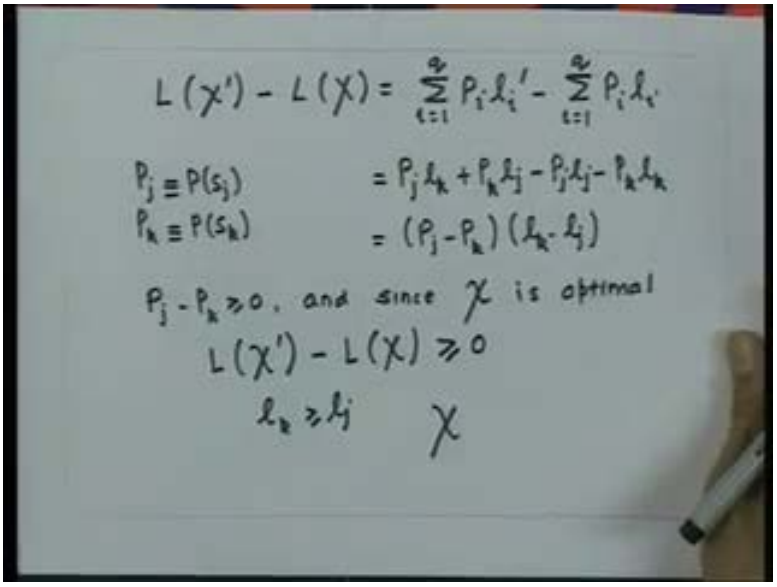
(Refer Slide Time: 25:10)



So, the first condition which an optimal variable and binary code has to satisfy is given. Given any two letters are S j and S k. If probability of S j is greater than equal to

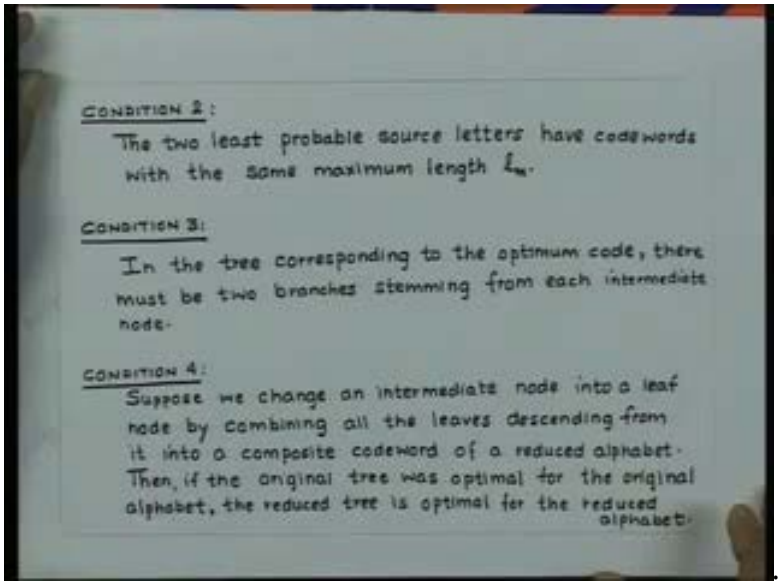
probability P of S k, then the code word length associated with the symbol S j that is l j will be less than equal to the code word length associated with the symbol S k. We have justified this in our earlier class too. But let us provide a quantitative proof for this. So, to prove this, consider an optimal code. Now, consider another code X prime. It is obtained by swapping the code word j and k. So, with code words j and k of X interchanged, if we do this, let us calculate the average length for the code X prime.

(Refer Slide Time: 27:34)



So, the average length for X prime and difference between the average length of original code X would be given by P i l i, i equal to 1 to q. So, when I take the difference, all the terms will cancel except for the two terms corresponding to the code words for the symbols S j and S k, so what I get here is P j l k plus P k l j minus P j l j minus P k l k. This can be simplified as where P j is probability of S j. P k is probability of S k. Now, we say that P j minus P k is greater than or equal to 0. Since, code X is optimal, what it implies that average length X dash minus average length of X should be always greater than equal to 0. Now, if condition has to be satisfied and if this is there, it implies that l k should be greater than equal to l j. This means that your code X satisfies the condition 1.

(Refer Slide Time: 30:11)



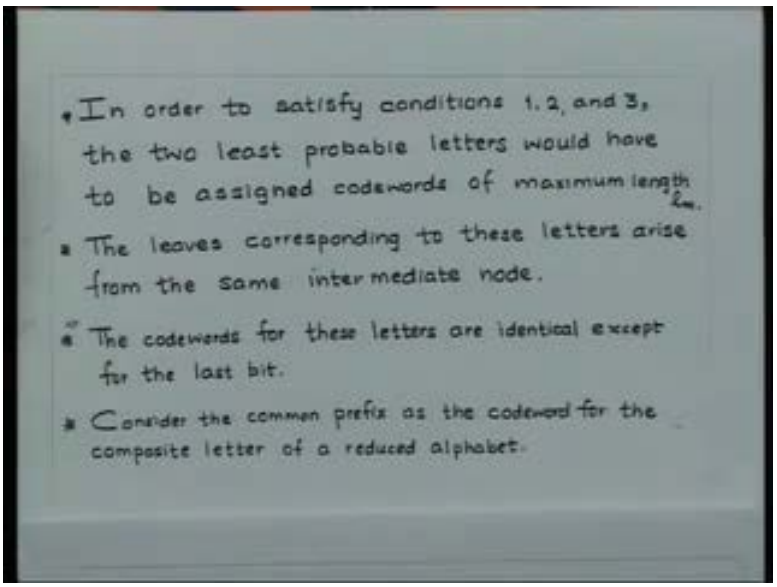
Now, the second condition which an optimal code should satisfy is that the 2 least probable source letters have code words with the same maximum length l m. Now, we have already provided the justification for this in the last class. We showed that if this condition was violated, then I would get a code with an average code length. This is less than the optimum. It contradicts our assumptions about the original optimum code. Next, the third condition which an optimal code should satisfy is; in the tree corresponding to the optimum code, there must be 2 branches stemming from each intermediate node.

Now, if this condition was not proved, it means that if there we any intermediate node with only one branch coming from this node, then we can always remove that branch or compress that without affecting the decipherability of the code. In the process, we are reducing its average length. So, this condition also has to be satisfied by an optimal code. Finally, the condition number 4 is that suppose we change an intermediate node into a leaf node by combining all the leaves descending from it into composite code word of a reduced alphabet. Then, if the original tree was optimal for the original alphabet, the reduced tree is optimal for the reduced alphabet.

Now, if this condition was not satisfied, then what it means that we could find a code with a smaller average code length for the reduced alphabet. Then, simply expand the composite code word again into a new code tree that would have a shorter average length than our original optimum tree. Now, this would contradict our statement about the

optimality of the original tree. Now, we will try to prove this little more quantitatively in the later part of the class today. Now, once we know that these conditions are to be satisfied for an optimal code. Let us see how to construct a code based on these conditions.

(Refer Slide Time: 33:30)

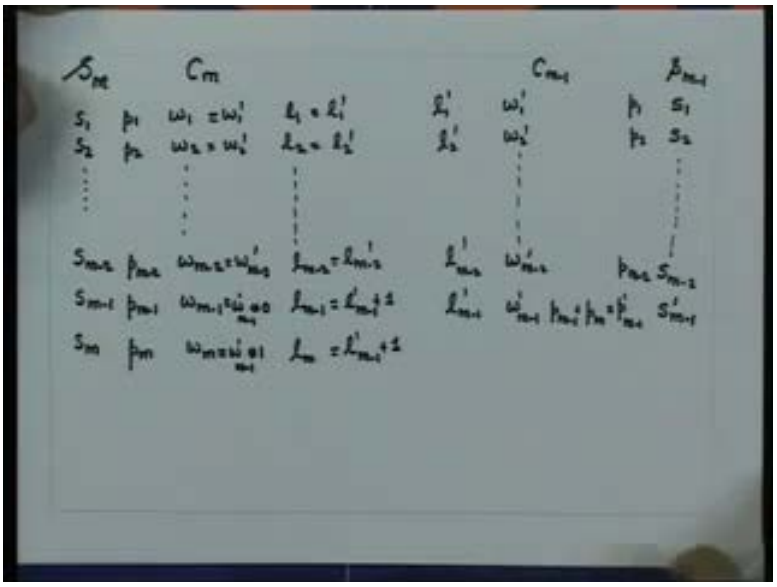


So, in order to satisfy conditions 1, 2 and 3, the 2 least probable letters would have to be assigned code words of maximum length let us say, l m. Now, furthermore, the leaves corresponding to these letters arrive from the same intermediate node. So, what this implies is that the code words for these letters are identical except for the last bit. So, what we can do is consider common prefix as the code word for the composite letter of a reduced alphabet. Now, since the code form the reduced alphabet, it needs to be optimum.

For the code of the original alphabet to be optimum, we follow this procedure again. So, to satisfy the necessary conditions, the procedure needs to be iterated until we have a reduced alphabet of size 2 for which the solution is obvious. But this is exactly the Huffman procedure. Therefore, we can say that the necessary conditions cited which are satisfied by the Huffman's procedure are also sufficient conditions. So, in the Huffman procedure, in order to obtain the optimum code for the original source alphabet, the optimum code is obtained for the reduced alphabet in an iterative fashion. Now, this is based on the fact that the code for the reduced alphabet needs to be optimum for the code

of the original alphabet to be optimum. So, let us try to prove this quantitatively. Let me assume that at a particular step in the Huffman procedure, we have an alphabet consisting of m symbols.

(Refer Slide Time: 36:04)



So, let me say that alphabet, denote that alphabet as s m consisting of letters s 1, s 2, s 10. For this alphabet, I have the corresponding code. Let me denote as C m with the code words given by w 1, w 2. Let us assume that the associated probabilities with these letters are that. So, my problem is to design an optimum code or to find an optimum code C m for this alphabet with these probabilities. So, alphabet consists of s m. Now, in the Huffman procedure, what we do is basically we obtain a reduction of this source.

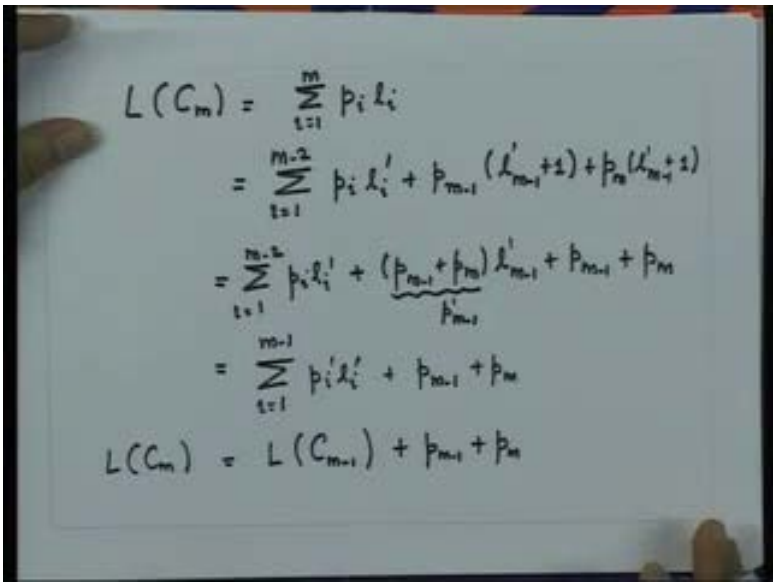
So, let us call this reduction of the source as s minus 1 with the source letters given by s 1, s 2, s minus 2 and s minus 1 dash. s dash m minus 1 is obtained by combining the source symbols or this letters in s m that is s m minus 1 s m. I am assuming that we have arranged this in the decreasing order of probabilities. The associated probabilities are this p 1, p 2, p m minus 2 with this. You can put dashes. But p 1 dash is equal to p 1. So, it makes no difference. p dash m minus 1 is equal to p minus 1 plus p m. Now, with this code, I also have the lengths associated with it. So, let me call l 1, l 2 up to l minus l m minus 2, l m minus 1 and m.

So, with this reduced alphabet, I want to have the code. The code words are given by w 1 dash, w 2 dash, w m minus 2 dash and w dash m minus 1. The associated lengths will be

l 1 dash, l 2 dash l m minus 2 dash. Now, from the way we have generated this reduced source, from this source, I can write that w 1 is equal to w 1 dash. w 2 is equal to w 2 dash. w m minus 2 is equal to w m minus 2 dash. w m minus 1 and w m. They will differ in just the last bit.

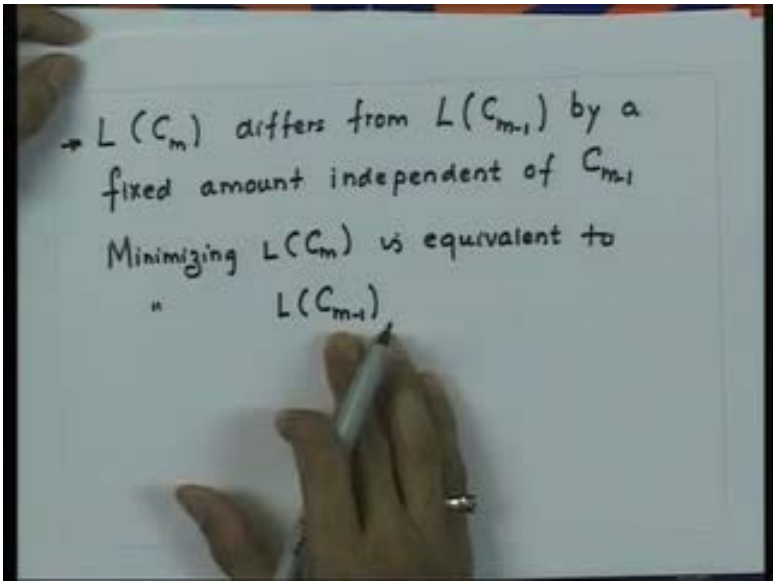
So, the common prefix will be the code word of s dash m minus 1. So, this would be given by quantity with 0. This will be given by the common prefix of both of this is the code word for this reduced symbol. Similarly, the length here would be same here. The length will be l dash m minus 1 plus 1 is the only difference of 1 bit. Similarly, for length m. Now, once we have written this, let us try to calculate the average code length for the code C m.

(Refer Slide Time: 41:37)



So, the average code length can be written as this can be summation for i equal to 1 to m minus 2. l i is same l i dash. So, I can write this expression plus m minus 1 and the length for m minus 1 symbol is l dash m minus 1 plus 1. Similarly, the length for the code word corresponding to s m is l dash m minus 1 plus 1. So, this can be simplified as that. Now, this is nothing, but p dash m minus 1. This quantity out here is p dash m minus 1. p i is equal to p i dash for i equal to 1 to m minus 2. So, I can combine these two X terms as one term. Write it down as and this. By definition, it is the average length of my code p of m minus 1.

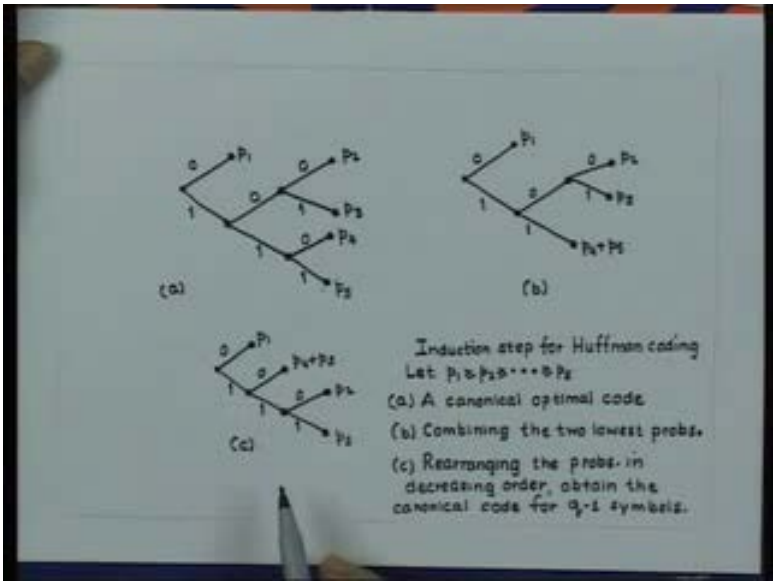
(Refer Slide Time: 44:17)



So, if we look at this expression, we can say that when l m differs from average length of l of m minus 1 by a fixed amount independent of C m minus 1. So, this is the conclusion which we can draw. So, what it means that minimizing this implies that minimizing is equivalent to minimizing. Now, we have reduced the problem to 1 with m minus. So, we have by moving to C minus 1.

We have reduced the original problem from m symbols and m probabilities to m minus 1 symbols and m minus 1 probabilities. So, for Huffman code, this procedure is repeated iteratively. So, we have proved in any case, if you want to have the optimum code for the alphabet, then you can design the optimal code for the reduced alphabet. Then, you go backwards to get the code. Now, the same thing can be depicted in form of a tree. This would look something like this.

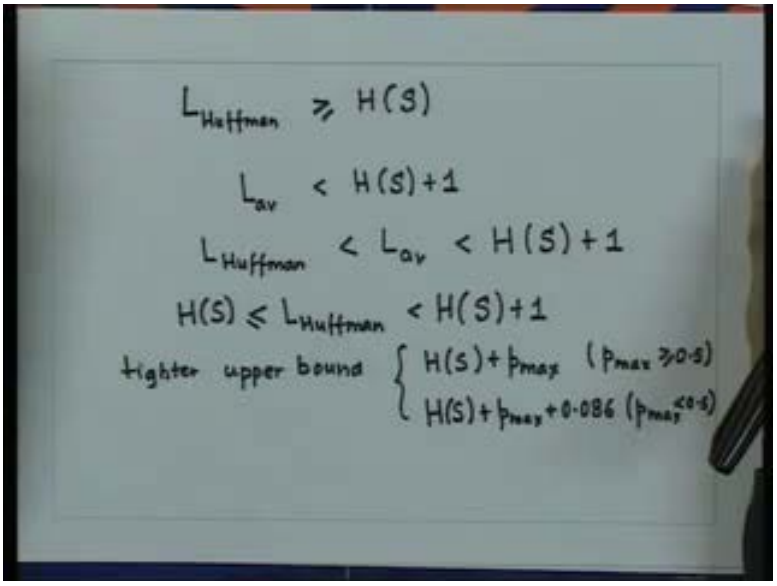
(Refer Slide Time: 46:28)



So, here we have taken, let us assume that at some stage is in the Huffman coding, we have 5 symbols corresponding to an alphabet. We presume that p 1, p 2, p 3, p 4, p 5 are being arranged in the descending order. So, this is a canonical optimal code. Now, you can combine the two probabilities p 4 and p 5 into 1 node. I get a new node here. Combine here and get the probability here. Then, rearrange the probabilities in decreasing order. I am assuming that the probability of p 4 plus p 5 is greater than p 2 or p 3. So, if I assume this, then I add it to probabilities greater than that.

I can rearrange to obtain the canonical code. For in this case, it should be 4 minus 14, q is in this case. Now, after proving the optimality of the Huffman code, the next question is that is it possible for us to calculate the bounds for this Huffman code. Now, it is little bit difficult to get the exact bounds. But we get some kind of an estimate both for the lower bound and the upper bound. The lower bound is not very difficult to find out because we know that for any code. The average length of the code has to be always greater than or equal to the entropy of the code. So, that is also valid for the Huffman code.

(Refer Slide Time: 48:38)



So, in that case, the average length of a Huffman code, if I call it as L Huffman, it will be always greater or equal to the entropy of the code. Now, to get an upper bound, we will follow a different strategy. We know that Shannon's coding strategy is a form of a uniquely decodable coding strategy. We have also proved that if we follow the Shannon's coding strategy or it is more popularly known as Shannon's coding, then the average length, which we obtain from the Shannon's coding is less than entropy of the source plus 1.

Now, for an optimal code like an optimal Huffman code, the length should be less than the length average here. So, it means that because length of a Huffman code is less than the average length of the Shannon's code which in turn implies that it is less than H S plus 1. So, we can say that for the Huffman code also the average length of the Huffman code will be bounded on the lower side by entropy and the upper side by 1 extra bit entropy plus 1.

Now, as I have said that it is possible to find out tighter bound on the Huffman code average length. But this is more involved. So, we will not discuss this in this class. But we can show that this tighter upper bound will be of the form H s plus p maximum where p maximum corresponds to the maximum probability of a source symbol in the source alphabet. This is valid when p maximum itself is greater or equal to 0.5. But if this p

maximum is less than 0.5, then the tighter upper bound becomes H s plus p max plus 0.0086. This is for p maximum less than 0.5.

Now, we have also seen that Huffman code will be always providing on the average code length, which is less than any other code. So, it is obvious that the Huffman code average length will be less than the average code length obtained from the Shannon's coding. But still it is worthwhile to have a look at the performance comparison of the Huffman and Shannon's coding strategy. We will have a look at the performance comparison in the next class.