

**An Introduction to Information Theory**  
**Prof. Adrish Banerjee**  
**Department of Electronics and Communication Engineering**  
**Indian Institute of Technology, Kanpur**

**Lecture – 04**  
**Block to Variable Length Coding-III: Huffman Coding**

Welcome to the course on an Introduction to Information Theory. So, we will continue our discussion on block to variable length code. In this lecture, we will talk about optimal block to variable length codes, which are also known as Huffman coding.

(Refer Slide Time: 00:33)

**Huffman code**

- The binary tree of an optimum binary prefix-free code for  $U$  has no unused leaves.  
Sketch of Proof:
- If the tree has unused leaves, they must be at maximum depth as the code is optimal.
- For atleast one value of  $u_i$  of  $U$ , we have following situation.

OR

- In either case, we can delete the last digit of the codeword, and still have a prefix-free code. The new code has smaller  $E[W]$  and thus original code must not be optimal.

And then we will talk about how we can encode a source. We have so far talked about a coding a single random variable, how the same concept can be extended to coding a source of length  $l$ . So, the property which I am going to show for optimal prefix-free code is as follows. So, let us first consider a binary prefix-free code. So, we are considering a codeword consists of zeros and ones. So, a binary tree corresponding to an optimal binary prefix-free code does not have any unused leaves. So, the first claim that we are making is if you have a binary prefix-free code which is optimal and if we map it to a binary tree then there should not be any unused leaves in this binary tree.

Now if there are. So, let us prove this. So, let us do by method of contradiction. So, let say if there are unused leaves then this unused leaves must be at the maximum depth, otherwise we can always bring in a codeword from higher depth to smaller depth in the

process reduce the expected codeword length. So, if at all there are unused leaves, they must be at the maximum depth; otherwise a code would not be optimal. Now, if so they are at the maximum depth then one of these two things can happen. Since, it is a binary tree, so there are two leaves. So, either one of this is assigned a codeword and the other one is just an unused leaf or we could have this as unused leaf and a codeword assigned to this.

Now, in this situation, you can see in either case, I can use this codeword here, I can move this codeword here and in the process, I am decreasing the expected codeword length by one. So, by deleting this last digit of the codeword we still have a prefix-free code and this new code has smaller expected value of  $W$ , hence my earlier claim that there are unused leaves at the maximum depth for optimal code is false. So, by method of contradiction, we have shown that this cannot happen. If you have an optimal binary prefix-free code then the binary free code corresponding to this prefix-free code cannot have any unused leaves.

(Refer Slide Time: 03:42)

**Huffman code**

- There is an optimal binary prefix-free code for  $U$  such that the two least likely codewords, say those for  $u_{K-1}$  and  $u_K$ , differ only in their last digit.

Sketch of Proof:

- Assume  $P_U(u_{K-1}) \geq P_U(u_K)$ . We have the following situation.

- If  $j \neq K$ , we switch the leaves for  $u_j$  and  $u_K$  without increasing  $E[W]$ .
- Similarly if  $i \neq K - 1$ , we switch the leaves for  $u_i$  and  $u_{K-1}$  without increasing  $E[W]$ .
- The new optimum code has its two least likely codewords differing only in their last digit.

The next result which you claim, which I am going to make is if there is an optimal binary prefix-free code such that two least likely codewords let us call them  $u_{k-1}$  and  $u_k$ . So,  $u_{k-1}$  and  $u_k$  are the two least likely codewords. So, these should differ only in their last digit. So, if you have two least likely codewords they should differ only in their last digit. So, the proof is like this. So, this is our least likely

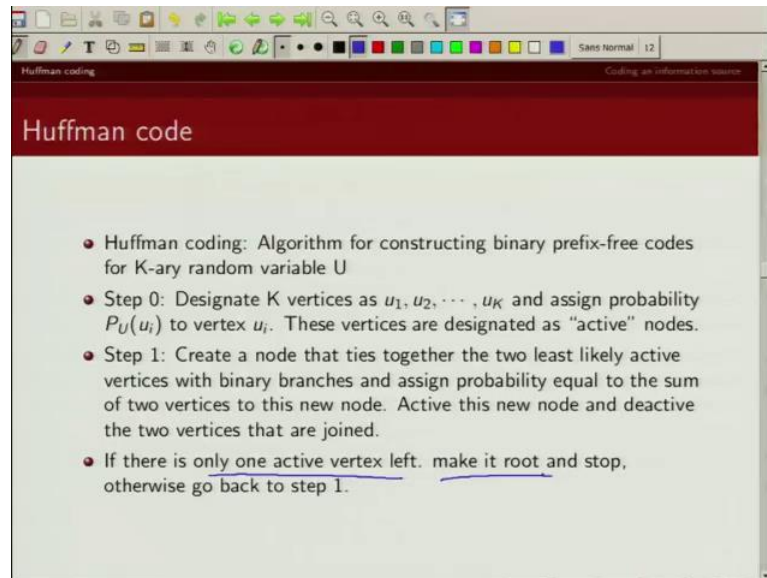
codewords and then the next least likely codeword is this. So, we have situation like this. So, we have a node at highest depth where you have two codewords  $u_i, u_j, u_i, u_j$  we can have something like this.

Now, what we can do is if let see our  $j$  is not equal to the least likely codeword then we can always replace  $u_j$  by codeword corresponding to  $u_k$ . So, if  $j$  is not equal to  $K$  we can always switch the leaves of  $u_j$  and  $u_k$ , because  $u_j$  probability of  $u_j$  is more than probability of  $u_k$ ; and we are moving  $u_j$  to a smaller depth in the process this swapping would not increase expected value of  $W$ . In fact, it can decrease; it is possible that the expected value of  $w$  can decrease. So, if this  $j$  is not  $u_j$  is not same as the leaves likely codeword  $u_k$ , I can always swap wherever my  $u_k$  was and I bring it here and I can move  $u_j$  back to where  $u_k$  was right. And this process would not increase my expected codeword length.

Similarly, if my  $u_i$  is not same as  $u_{k-1}$ , which is the second least likely codeword then I can always swap where my  $u_k$  and  $u_{k-1}$  was and bring it here and move  $u_i$  back to where  $u_{k-1}$  was. Now, since probability of  $u_i$  is more than  $u_{k-1}$  and since my  $u_i$  was already at the maximum depth, when I swap them my expected codeword length is not going to increase, in fact it can decrease.

So, hence what I have shown you is if at maximum depth, you have codewords like this  $u_i$  and  $u_j$  then I can make this  $u_i, u_j$  as  $u_k$  and  $u_{k-1}$  then without actually increasing my expected codeword length. Hence, if I have and what is my optimal code the one which will minimize my codeword length expected value of  $w$ . So, what I have shown you is I can have a situation where the two least likely codewords are the part of the leaves or the highest depth and this will result in average codeword length which is minimum. Hence, we can see that if they are part of the coming out from the same node then these two codewords will differ only in the last digit because before this the path from the root to this node is same for both these codewords. So, two least likely codewords then differ only in their last digit. So, what we have shown is the new optimum code which has the minimum expected value of  $w$  will have two least likely codewords and they will only differ in the last digit that is because from root to this node the bits are all same, they are only differing in the last digit.

(Refer Slide Time: 08:10)



So, now, let us give the algorithm using these two results we will give algorithm for constructing optimal binary prefix-free code. So, we are interested in optimal prefix-free code for a random variable U which takes K different values. So, let us designate these K different values as  $u_1, u_2, u_3, \dots, u_k$  and we assign probabilities to each of these vertices, which is equal to the probability of  $u_i$ . and we will designate these vertices as active vertices.

Step 1: We create a node that ties to two least likely active vertices please note we have said in the previous lemma that a for optimal binary prefix-free code two least likely codewords will differ in only one location - one last bit locations. So, we consider two least likely active vertices and we combine them and join them and create a new node; and a probability of that node is equal to sum of the probabilities of the two vertices. So, this new node that we are creating is probability is equal to the sum of probabilities of the two nodes that get just got joint. We are going to activate this new node and deactivate the two vertices that were joint. So, now, those two vertices got deactivated, but we have created a new active node which has probability equal to sum of the probabilities of those two earlier vertices. Now, we continue this process until we are left with only one active vertex and if that is the case will make it root vertex and we stop and that is how we are going to construct optimal binary prefix-free code which is a Huffman code.

(Refer Slide Time: 10:29)

Huffman code

- Construct binary Huffman code for the following example:  
 $P(u_1) = 0.05, P(u_2) = 0.1, P(u_3) = 0.15, P(u_4) = 0.2, P(u_5) = 0.23, P(u_6) = 0.27$

U	Z
$u_1$	0000
$u_2$	0001
$u_3$	001
$u_4$	10
$u_5$	11
$u_6$	01

So, let us take an example. We have a binary Huffman code which takes 6 different values. So,  $u_1$  has probability 0.05,  $u_2$  has probability 0.1,  $u_3$  has probability 0.15,  $u_4$  has probability 0.2,  $u_5$  has probability 0.23 and  $u_6$  has probability 0.27. So, first thing we do is we just write down a vertex call is corresponding to each of the six possible values of  $u$ , and we write the corresponding probabilities of these  $u_i$  which is given by here. Now, note we are interested in binary Huffman codes; there are two branches from each node. So, we look at two vertices which have the least probability and which are those two vertices this one corresponding to  $u_1$ , another corresponding to  $u_2$ . So, what we do is we combine these two vertices and we create a new vertex which is the sum of probabilities of this and this. So, this new vertex which is formed is its probability is you know that we are forming a probability 0.15. Now, we deactivate these nodes. So, now, the active nodes are this one, this, this, this and this.

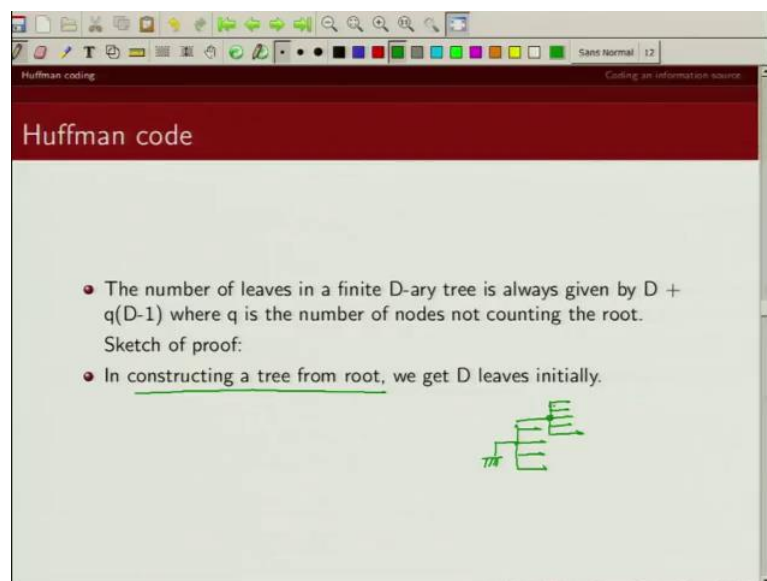
Next step, what are the two least likely active nodes. So, clearly the least likely nodes are at this point is this one and this one. So, we combine these two and create a new node, now what is the probability of this node, the probability of this node is probability of this node which is 0.15 plus probability of this node which is also 0.15. So, this probability is 0.3. Now, we deactivate these nodes. So, now, our active nodes are this one, this one, this one and this one. Now among these four active nodes which two have the least probability the 1 cause point 2  $u_4$  and  $u_5$ . So, we combine these two and create a new node and what is the probability of this node 0.2 plus 0.23. So, that is 0.43. Now, we

deactivate these nodes and we activate this node. So, now, our active nodes are this one, this one and this one. So, we have three active nodes.

Now out of these three active nodes which one has the least probability 0.27 and 0.3? So, we combine these two nodes, we create a new node whose probability sum of probability of these two, so 0.3 plus 0.27 that is 0.57. Now, we deactivate these two nodes now what are the active nodes left now, one is this node, another is this node. So, we combine these together, we get a new node this probability of this node is 1, and now we have only one active node remaining. So, we make it a root node. And now each of these two branches which are starting from each of these nodes we find one of them as code bit 0, other as bit 1.

Now, you can trace back the path from root to each of these codewords to get the corresponding codeword. So, for example, u 1 is nothing but 0, 0, 0, 0 so that is my codeword corresponding to u 1. What is u 2? 0, 0, 0, 1, 0, 0, 0, 1; What is u 3? That is 0, 0 and 1 that is u 3. What is u 4? u 4 is this one so that is 1 and then 0. What is u 5? u 5 is 1 1. And similarly, what is u 6? u 6 is this one so that is 0 and 1. So, this is the optimal binary prefix pre coding for a source which has six possible values and whose probabilities are given by this. So, this is the optimal Huffman coding for this particular source.

(Refer Slide Time: 16:33)



The image shows a screenshot of a presentation slide titled "Huffman code". The slide contains the following text:

- The number of leaves in a finite D-ary tree is always given by  $D + q(D-1)$  where  $q$  is the number of nodes not counting the root.  
Sketch of proof:
- In constructing a tree from root, we get D leaves initially.

Below the text is a small, hand-drawn tree diagram with a root node and several branches leading to leaf nodes.

Now, we will try to extend this result for a D-ary prefix-free code. So, first we will go to

prove a result which says the number of leaves in a finite D-ary tree is given by D plus q times D minus 1 where q is number of nodes now counting the root node. So, what we are saying here is the number of leaves in a D-ary tree is given by D plus q times D minus 1. So, how do we prove this result? So, let us look at. So, let us construct a tree from a root. So, if we are constructing a tree from a root. So, initially, what is going to happen; initially we are going to have D leaves starting from this root. Now, initially you will from a root, you will have D leaves. Next, if you try to extend any of the leaves, what is going to happen is you are going to create D new leaves, but in the process this leaf which was earlier a leaf now become a node. So, effectively we are adding D minus 1 leaves.

(Refer Slide Time: 18:04)

The image shows a presentation slide with a red header containing the text "Huffman code". Below the header, there is a list of three bullet points:

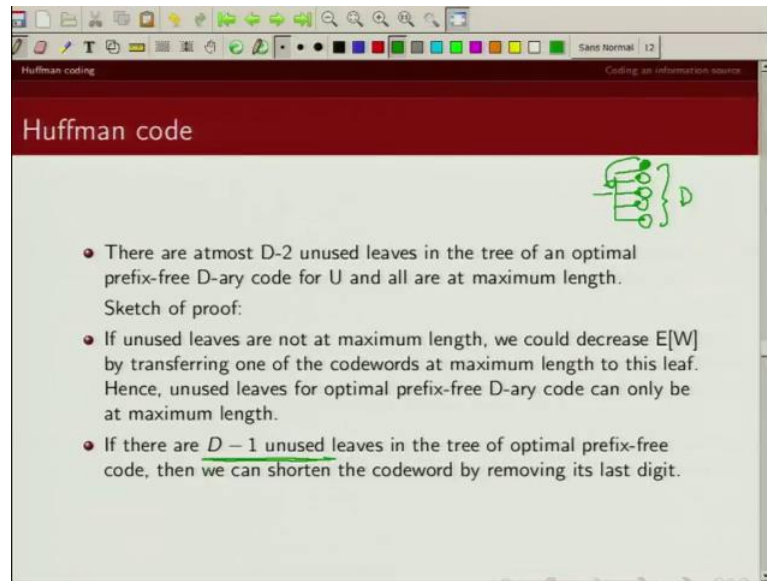
- The number of leaves in a finite D-ary tree is always given by  $D + q(D-1)$  where q is the number of nodes not counting the root.
- Sketch of proof:
- In constructing a tree from root, we get D leaves initially.
- At each subsequent step, if we extend any leaf we get D new leaves and lose one old leaf.

Below the text, the formula  $D + q(D-1)$  is handwritten in green ink and enclosed in a green rectangular box.

So, then at each subsequent step, if you are trying to extend a leaf, we are getting D new leaves; however, we are losing 1 old leaf, because that particular leaf now has become a node. So, if you do q such extensions of this tree, what we are going to get is we are going to get D plus q times D minus 1 number of leaves where q is the number of leaves not counting the root leaf, because if we start from root leaf we get D leaves. But subsequently when we are trying to extend a leaves, we create D new leaves; however, that particular leaf now becomes a node. So, effectively we are adding D minus 1 new leaves. So, this is the total number of leaves that we can have in a D-ary tree.



(Refer Slide Time: 19:12)



The slide is titled "Huffman code" and is part of a presentation on "Coding an information source". It contains the following text:

- There are at most  $D-2$  unused leaves in the tree of an optimal prefix-free  $D$ -ary code for  $U$  and all are at maximum length.  
Sketch of proof:
- If unused leaves are not at maximum length, we could decrease  $E[W]$  by transferring one of the codewords at maximum length to this leaf. Hence, unused leaves for optimal prefix-free  $D$ -ary code can only be at maximum length.
- If there are  $D - 1$  unused leaves in the tree of optimal prefix-free code, then we can shorten the codeword by removing its last digit.

A small tree diagram is drawn in green on the right side of the slide, showing a root node with three children, and one of those children having three children of its own, with a bracket indicating a total of  $D$  leaves.

So, first result we are going to show for a  $D$ -ary prefix-free code is optimal  $D$ -ary prefix-free code is there are at most  $D$  minus 2 unused leaves. For an optimal  $D$ -ary code and all of them are at the maximum length. Now, proof is very, very similar to the case for binary source. If there are unused leaves and if they are not at maximum depth then we can always decrease our average codeword length by moving leaf from maximum depth to those locations. So, if at all there are unused leaves, it has to be at the maximum depth. So, this is clear that if there are unused leaves they have to be at the maximum depth. Now, how many such unused leaves can be there?

So, note that there are maximum  $D$  leaves possible; this number of leaves is basically we can have  $D$  leaves possible. Now, if there are  $D$  minus 1 unused leaves that means only one of the leaves is been used all others are not used. If such a situation happens, we can always move this leaf to this node without increasing our average codeword length. Hence, we cannot have  $D$  minus 1 unused leaves, so at most we can have  $D$  minus 2 unused leaves and those  $D$  minus 2 unused leaves have to be at the maximum depth otherwise we can always reduce expected codeword length by moving some of the codeword to lower depth.



(Refer Slide Time: 21:28)

**Huffman code**

- The number of unused leaves in the tree of an optimal  $D$ -ary prefix-free code for a random variable  $U$  with  $K$  possible values,  $K \geq D$ , is the remainder when  $(K - D)(D - 2)$  is divided by  $D - 1$ .  
Proof:
- Let  $r$  be the number of unused leaves. Then if  $U$  has  $K$  values,  

$$r = [\text{number of leaves in } D\text{-ary tree of the code}] - K$$
- This implies  

$$r = [D + q(D - 1)] - K$$
- or  

$$D - K = -q(D - 1) + r \quad \text{where } 0 \leq r < D - 1$$
- Adding  $(K - D)(D - 1)$  to both sides of the above equation, we get  

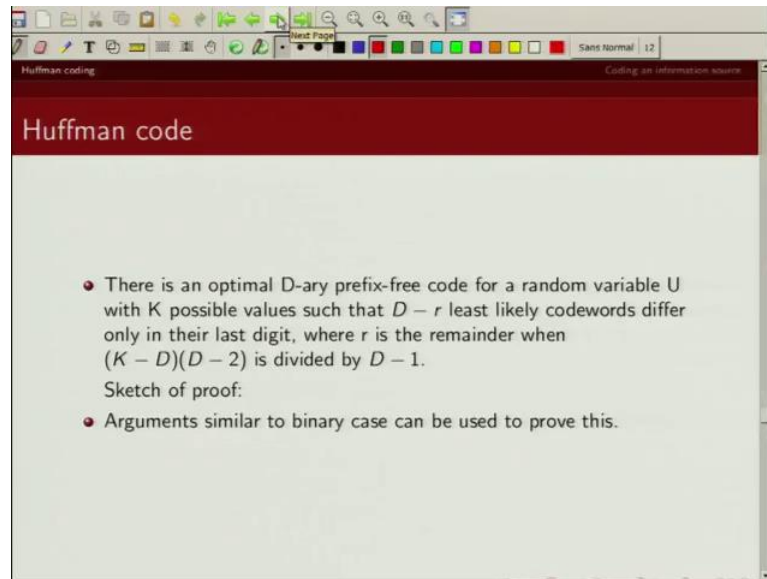
$$[(K - D)(D - 2)] = (K - D - q)(D - 1) + r \quad \text{where } 0 \leq r < D - 1$$

Now, in the previous result, we have shown that utmost there can be  $D$  minus 2 unused leaves and those unused leaves have to be at the maximum depth. In this result, we are going to calculate how many unused leaves are there in an optimal  $D$ -ary prefix-free code. So, for a random variable  $U$  that takes  $K$  different values where  $K$  is greater than  $D$  then the remainder obtained by dividing  $K$  minus  $D$  into  $D$  minus 2 by  $D$  minus 1 that number will give us number of unused leaves in our  $D$ -ary prefix-free code. And remember all of these unused leaves have to be at the maximum depth. So, number of unused leaves is given by number of leaves in a  $D$ -ary tree minus number of possibilities of this random variable  $U$ , which is  $K$ , because  $U$  takes  $K$  different values.

And what is the number of leaves in a  $D$ -ary tree that number is given by  $D$  plus  $q$  times  $D$  minus 1 where  $q$  is number of nodes minus the root nodes. So, number of unused leaves is then given by this expression. Now, I do some algebraic manipulation I can write this as this particular way. So, you can think of as number of unused leaves as if I divide  $D$  minus  $K$  by  $D$  minus 1 whatever remainder I get that is the number of unused leaves. Or I can do some more manipulation if I add  $K$  minus  $D$  into  $D$  minus 1 to both sides I can write this expression in this particular fashion. So, I can then see that if I divide  $K$  minus  $D$  into  $D$  minus 2 if I divide this by  $D$  minus 1 whatever remainder I get is my number of unused leaves. So, for an optimal  $D$ -ary prefix-free code, the numbers of unused leaves are given by if I divide  $K$  minus  $D$  into  $D$  minus 2 by  $D$  minus 1, whatever remainder I get that is the number of unused leaves and those unused leaves must be at the

largest depth.

(Refer Slide Time: 24:27)



Huffman coding

## Huffman code

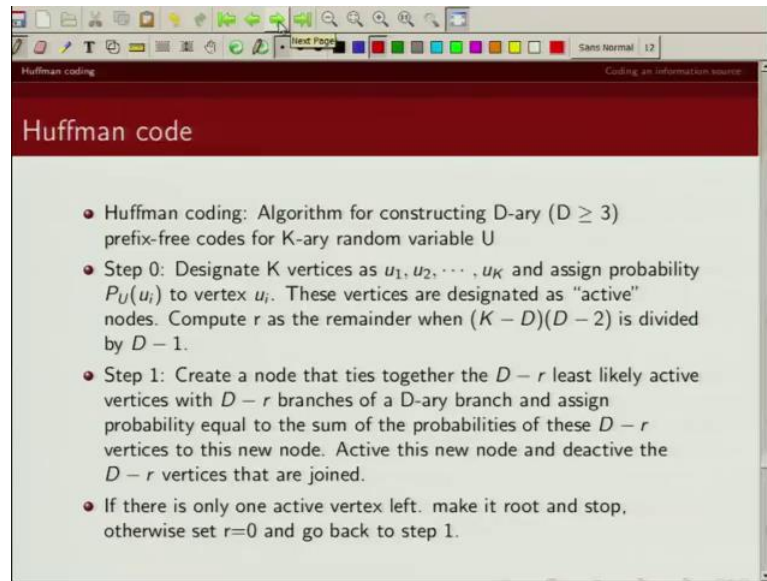
- There is an optimal  $D$ -ary prefix-free code for a random variable  $U$  with  $K$  possible values such that  $D - r$  least likely codewords differ only in their last digit, where  $r$  is the remainder when  $(K - D)(D - 2)$  is divided by  $D - 1$ .

Sketch of proof:

- Arguments similar to binary case can be used to prove this.

So, then I can write that there is an optimal  $D$  ary prefix-free code for a random variable  $U$  with  $K$  possible values such that the  $D$  minus  $r$  least likely codewords they differ only in their last digit, and this  $r$  is computed as we shown earlier by dividing  $K$  minus  $D$  into  $D$  minus 2 by  $D$  minus 1. Now, this prove that  $D$  minus  $r$  least likely codewords differ in only one position this proof is very similar to the proof that we did for binary optimal prefix-free code where we showed that there two least likely codewords differ in only one bit location. So, I am not going over this proof this is very, very similar you start off with some codeword  $u_i, u_j$  and if they are not the least likely codewords you can always swap it with least likely codewords and you can show that this process of swapping would not increase your expected codeword length. So, this proof is very, very similar to the proof that we did for optimal binary prefix-free code.

(Refer Slide Time: 25:56)



The image shows a presentation slide titled "Huffman code" with a red header. The slide contains a bulleted list of steps for constructing a D-ary prefix-free code. The steps are:

- Huffman coding: Algorithm for constructing D-ary ( $D \geq 3$ ) prefix-free codes for K-ary random variable U
- Step 0: Designate K vertices as  $u_1, u_2, \dots, u_K$  and assign probability  $P_U(u_i)$  to vertex  $u_i$ . These vertices are designated as "active" nodes. Compute  $r$  as the remainder when  $(K - D)(D - 2)$  is divided by  $D - 1$ .
- Step 1: Create a node that ties together the  $D - r$  least likely active vertices with  $D - r$  branches of a D-ary branch and assign probability equal to the sum of the probabilities of these  $D - r$  vertices to this new node. Active this new node and deactivate the  $D - r$  vertices that are joined.
- If there is only one active vertex left, make it root and stop, otherwise set  $r=0$  and go back to step 1.

So, then I can state the algorithm for optimal prefix-free code for D ary prefix-free code. So, you have U that takes K different values lets designate those vertices as  $u_1, u_2, u_3, \dots, u_{K-1}$  and let us assign probability  $P_U(u_i)$  to each of these  $u_i$ . Now, we will designate this is what is the active vertices. The first thing that we are going to do is we are going to compute how many unused leaves are there at the maximum depth that is  $r$ . And how do we compute that that is computed by dividing  $K - D$  into  $D - 2$  by  $D - 1$  and whatever remainder we get that is my number of unused leaves.

So, in the first step, what we do is we create a node that ties together  $D - r$  least likely vertices with  $D - r$  branches of a D-ary branch, node that this  $r$  of the leaves is unused and total there are  $D$  branches. So, in the first step, we are going to combine only  $D - r$  vertices  $D - r$  least likely vertices and we will assign probability to this newly created node which is nothing but sum of probabilities of these  $D - r$  vertices. We will activate these vertices and deactivate the  $D - r$  vertices that were joined.

Now, in the next step, we are going to combine because you remember the unused leaves can only be at the maximum depth. So, in the next time, when we combine we are going to combine  $D$  vertices,  $D$  active nodes together and then combine them into a new node whose probability will be sum of probabilities of these  $D$  nodes,  $D$  vertices. And we will continue this process until we all left with only one node and that would be our root node with probability one.

(Refer Slide Time: 28:38)

Huffman code

• Construct ternary Huffman code for the following example:  
 $P(u_1) = 0.05, P(u_2) = 0.1, P(u_3) = 0.15, P(u_4) = 0.2, P(u_5) = 0.23, P(u_6) = 0.27$

$D = 3$   
 $K = 6$   
 $r = \frac{(K-D)(D-2)}{D-1} = 1$

U	Z
$u_1$	200
$u_2$	201
$u_3$	21
$u_4$	22
$u_5$	0
$u_6$	1

So, let us illustrate this Huffman coding algorithm with an example. So, we are considering a ternary Huffman code. So,  $D$  in our case is 3. We are considering the same example which we use for binary Huffman code. So, in this example, my key is 6. So,  $u$  takes 6 different values  $u_1, u_2, u_3, u_4, u_5$  and  $u_6$ ; and these are the corresponding probabilities of these  $u_i$ (s). The first step is I write down all my  $u_i$ s and the corresponding probabilities. So, this is my  $u_1, u_2, u_3, u_4, u_5$  and  $u_6$  and these are my corresponding probabilities of these  $u_i$ (s).

Now, the first step is I need to compute how many unused leaves are there. So, how do I find unused leaves, I divide  $K$  minus  $D$   $D$  minus 2 I divide this by  $D$  minus 1. So, what is  $K$  minus  $D$  that is 3,  $D$  minus 2 is 1, so 3 divide by 2 remainder is 1; that means, in this example I have one unused leaf and remember that unused leaf has to be at the maximum depth. So, what do I do, in the first step, I only combine two vertices which have the least probabilities; and what are those two vertices this is  $u_1$  and  $u_2$ . So, combine these two and this is my new node which has probability equal to sum of these two nodes this is 0.15; clearly I have an unused node here.

Next, I deactivate these nodes and now my active nodes are this one, this one, this one, this one and this one. So, I now have to pick three least likely active nodes and what are those that is 0.15, 0.15 and 0.2. So, least likely nodes are this I create a new node its probability is given by sum of probabilities of these two nodes is 0.15 plus 0.15 plus 0.2,

so that is 0.5. Now I deactivate these nodes. So, at this point, my active nodes are this, this and this. So, three active nodes, I combine all of them and what I get is so 0.5, 0.23 and 0.27, I get basically this. So, I get this, I get this, and I get this. So, this one node left, I deactivate these nodes, so has only one active node left. So, I make that as my root node.

And next I will assign a bit 2 each of this three branches, I am calling it 0, 1, 2 and then you can trace back the path from root to each of these codewords and that is your codeword. For example, u 1, what is u 1, u 1 is this 2, 0, 0 that is your u 1. U 2 is 2, 0, 1 similarly you can find out the corresponding codewords for each of these u i(s). So, again I repeat the difference from the binary case here is here in the final step in the maximum depth you can have some unused leaf, so you have to first find out that. So, in the first step, you are only going to combine D minus r active vertices; and subsequently, you will be combining D vertices until you are left with only one vertex which is basically your root.

(Refer Slide Time: 33:42)

**Coding an information source**

- Parsing an information source

```

graph LR
    A[Information Source] -- "U1U2..." --> B[Source Parser]
    B -- "V1V2..." --> C[Message Encoder]
    C -- "Z1Z2..." --> D[ ]
  
```

- The source parser divides the output sequences from the information source into messages that are encoded by the message encoder.
- We consider an L-block source parser, i.e.
 
$$V_1 = [U_1, U_2, \dots, U_L]$$

$$V_2 = [U_{L+1}, U_{L+2}, \dots, U_{2L}]$$

$$\vdots$$

Now, what we have done so far is we have shown you how to encode a single random variable. So, we have a single random variable U which takes K different values, how can we optimally encoded using block to variable length coding. Now, what about if you have block of data, so I have an information sequence U 1, U 2, U 3 and I want to encode block of let say block of L bits how do I apply whatever we have studied so far to a block

of data. So, what I need to do then is I have an information source which is giving you the few is I need to partition this into blocks of data. Let us say I am partitioning this into source parts of what we does it, it divides the output sequence into messages, which will be encoded by this source encoder message encoder. So, it is creating blocks of so each block will have L bits of  $u_i(s)$ .

So, we consider an L blocks source parser. So, you can think of us  $V_1$  as  $U_1$  to  $U_L$ ,  $V_2$  as  $U_{L+1}$  to  $U_{2L}$ ,  $V_3$  as  $U_{2L+1}$  to  $U_{3L}$ , like that basically the source parser what is doing is you are getting bits of data it is partitioning into a block of L bits. Now, you are sending this block of l bits to the message encoder which is going to do block to variable length coding.

(Refer Slide Time: 35:22)

Huffman coding Coding an information source

### Coding an Information source

- Block to variable length coding theorem for a DMS. There exists a D-ary prefix-free coding of a L-block message from DMS such that the average number of D-ary code digits per source satisfies

$$\frac{E[W]}{L} < \frac{H(U)}{\log D} + \frac{1}{L}$$

where  $H(U)$  is the uncertainty of a single source letter. Conversely for every D-ary prefix-free coding of an L-block message

$$\frac{E[W]}{L} \geq \frac{H(U)}{\log D}$$

So, we can show that there exist prefix-free codes a D-ary prefix-free code for this block of L bits which are coming from a discrete memory less source which satisfies this condition that average number average codeword length per block size is upper bonded by this and lower bounded by this term. Now, we already have computed bounds on optimal codeword length for a coding of single random variable we are going to extended result for the case when we want to encode a force of length L.

(Refer Slide Time: 36:18)

**Coding an Information source**

- Proof: Since the message  $V = [U_1, U_2, \dots, U_L]$  has L i.i.d. components, we have
 
$$\begin{aligned} \underline{H(V)} &= H(U_1) + H(U_2) + \dots + H(U_L) \\ &= \underline{LH(U)} \end{aligned}$$
- For any D-ary prefix-free code for  $V$ , we have
 
$$\begin{aligned} \frac{H(V)}{\log D} &\leq E[W] < \frac{H(V)}{\log D} + 1 \\ \Rightarrow \frac{LH(U)}{\log D} &\leq E[W] < \frac{LH(U)}{\log D} + 1 \\ \Rightarrow \frac{H(U)}{\log D} &\leq \frac{E[W]}{L} < \frac{H(U)}{\log D} + \frac{1}{L} \end{aligned}$$

So, we have a source  $U$  which consists of length components of this  $U_i$ 's. This is  $L$  i.i.d. component. So, in that case, uncertainty in  $V$  can be returned as uncertainty in  $U_1$  plus uncertainty in  $U_2$  plus uncertainty in  $U_3$  up to  $U_L$ . Since they are identically distributed and they are independent  $H$  of  $U_1$  is same as  $H$  of  $U_2$  is same as  $H$  of  $U_L$  is same as  $H$  of  $U$  and since they are independent for we can write uncertainty of  $V$  as  $L$  times  $H$  of  $U$ . Now, we know if we have a prefix-free code for  $V$ , we have shown earlier that, expected codeword length is lower bounded by entropy by  $\log 2$  and its upper bounded by entropy by  $\log 2$  plus 1 this we have proved in the earlier lecture.

So, if we plug in the value of uncertainty in  $V$ , what we get here is  $L$  times  $H$  of  $U$  by  $\log$  of  $D$  this must be less than expected codeword length and expected codeword length if  $L$  times  $H$  of  $U$   $\log D$  plus 1. Now if we divide 1 by  $L$ , what we get here is this relation that average codeword length per this length  $L$  is basically lower bounded by this quantity entropy and upper bounded by this quantity. Of course, if  $L$  is very large you can see this upper and lower bound are very close. So, with this, we will conclude our discussion on block to variable length coding.

Thank you.