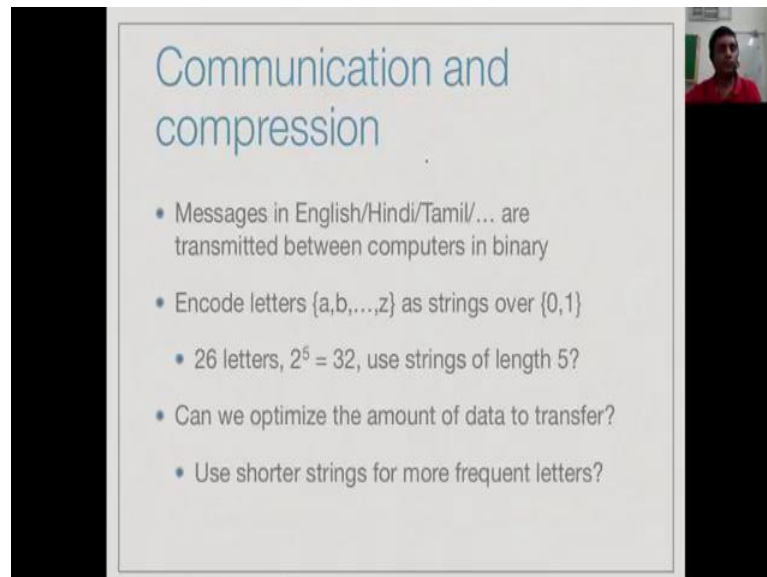


Design and Analysis of Algorithms
Prof. Madhavan Mukund
Chennai Mathematical Institute

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 blue text. It contains a list of five bullet points in black text. A small video inset of a man in a red shirt is visible in the top right corner of the slide.

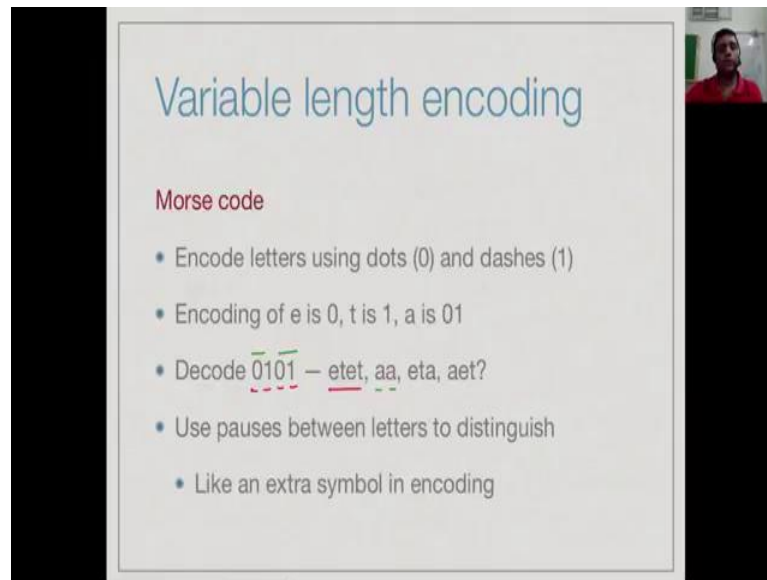
- Messages in English/Hindi/Tamil/... are transmitted between computers in binary
- Encode letters $\{a, b, \dots, z\}$ as strings over $\{0, 1\}$
 - 26 letters, $2^5 = 32$, use strings of length 5?
- Can we optimize the amount of data to transfer?
 - Use shorter strings for more frequent letters?

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.

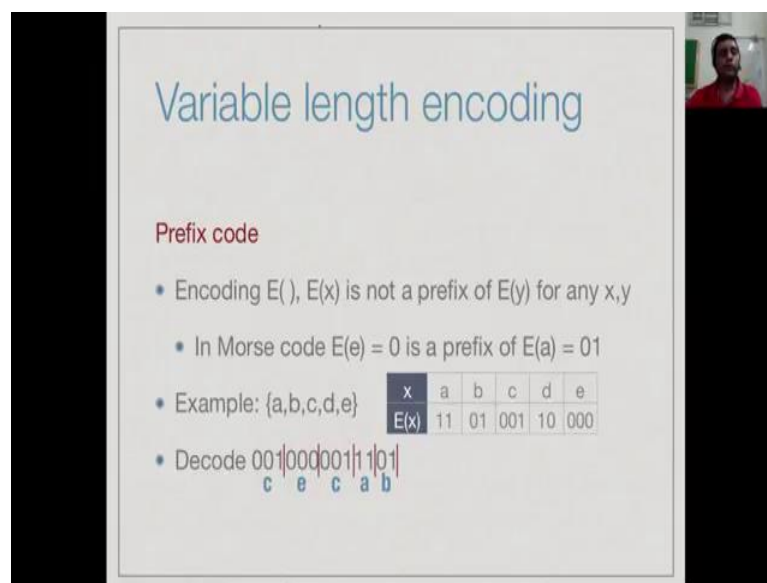
So, in the Morse code encoding, different letters do have different encodings and in English e is the most frequent letter and t is another variant frequent letter. So, Morse assigned them codes of a dot, that is 0 for e and a dash, that is 1 for t, then a Morse took other frequent letters, such as a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

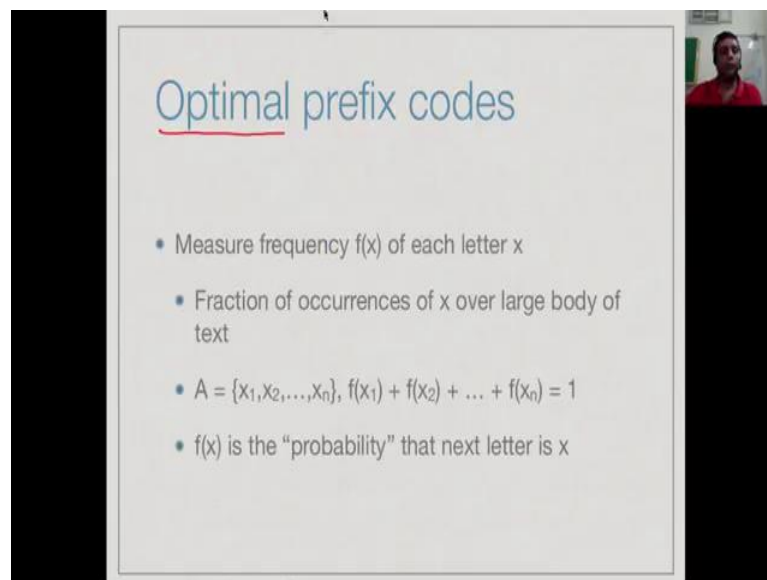
So, in order to make a variable length code an unambiguous decodable, we need what is called a prefix quantity. When we read through a sequence of 0's and 1's, we should be an ambiguously clear, whether we have read a letter or there is more to read. We should be like the earlier case, where we have 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
- For each letter x , $n \cdot f(x)$ occurrences of x in M $n \cdot f(x)$
- 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 $\sum_{x \in A} f(x) \cdot |E(x)|$

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

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

2.25

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

- 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 labeled 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

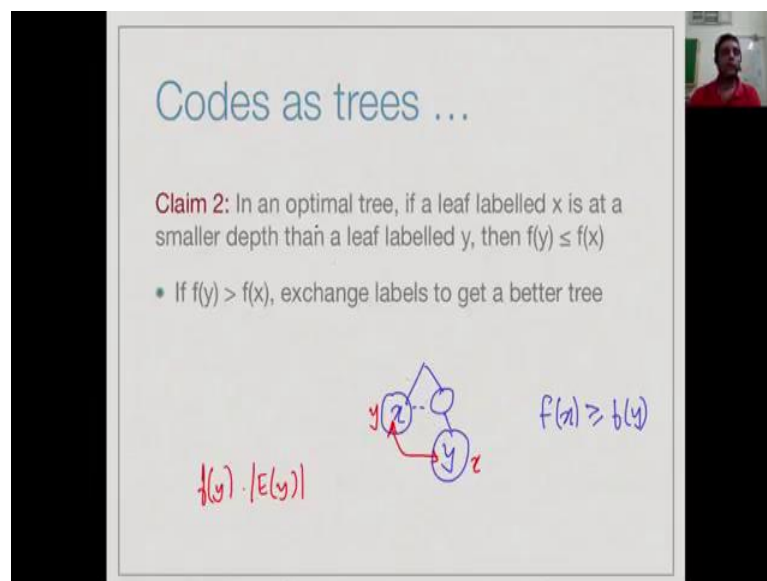
Claim 1: Any optimal prefix code generates a full tree

- 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 $f(y) \leq f(x)$

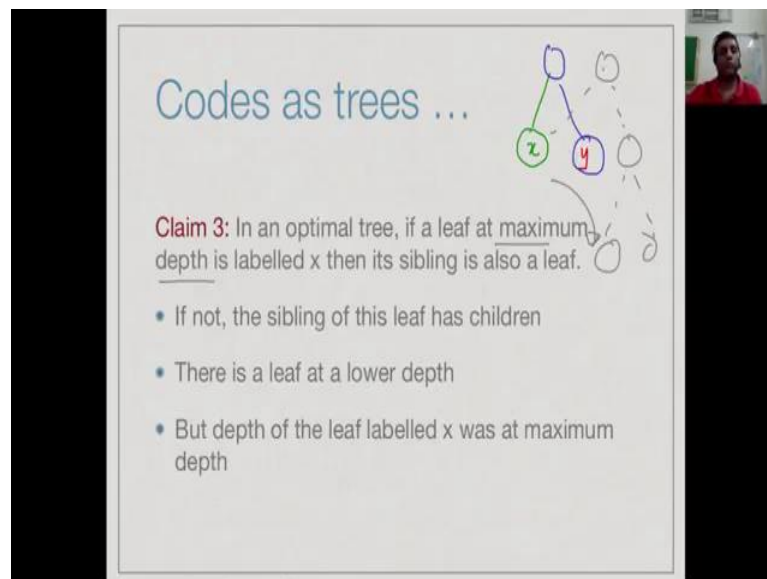
- If $f(y) > f(x)$, exchange labels to get a better tree

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 ...

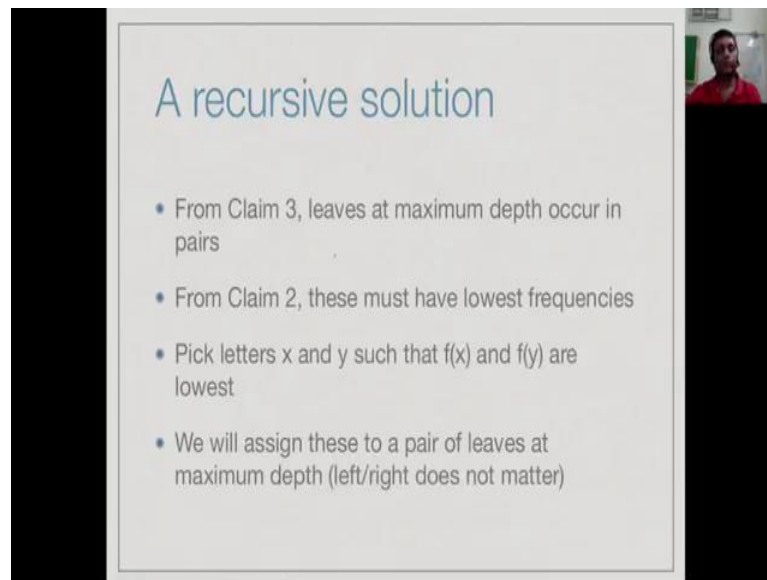
Claim 3: In an optimal tree, if a leaf at maximum depth is labelled x then its sibling is also a leaf.

- 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.

(Refer Slide Time: 17:10)

A recursive solution ...

- "Combine" x and y into a new letter " xy " with $f(xy) = f(x) + f(y)$
- New alphabet A' is original $A - \{x, y\} + \{xy\}$
- Recursively find an optimal encoding of A'
 - Base case, $|A'| = 2$, assign the two letters codes 0, 1
- Replace the leaf labelled " xy " by a node with two children labelled x and y
- Huffman's algorithm — Huffman coding



So, now, the recursive a solution will say, that how do a figure of what the rest of the tree looks like, well if I have a situation, where I have decided x and y both here. Then, I will kind of tree, this is a unit and make a new letter call x , y and give it the cumulative frequency effects plus x , y of the old two letter. So, construct the new alphabet and which I drop x and y and I add a new composite of hybrid letter x , y ; whose frequencies going to be $f x$ plus $f y$.

Now, recursion fiction, I have a k minus 1 letter alphabet, so I have recursively find and optimal encoding of that. Now, before coming to how to adapt the solution, the recursive ends when have a only two letters, for two the optimal solution is to build the tree which exactly two leaves, label 0 and 1 at the path. So, this is the basic case, if I have more than two letters I will recursively construct tree to the smaller thing and then I will come back and now, the tree that I constructed I will have some where the leaf label $x y$.

Now, $x y$ is not a letter, so what I do is, I will replace this, write new two leaves called x and y . So, I will go from the tree over a A prime to a tree over A by doings. So, this is an algorithm called by develop Huffman and this type of coding is call Huffman coding.

(Refer Slide Time: 18:36)

Huffman's algorithm

x	a	b	c	d	e
f(x)	0.32	0.25	0.20	0.18	0.05

Combine d, e as "de"

x	a	b	c	de
f(x)	0.32	0.25	0.20	0.23

Combine c, de as "cde"

x	a	b	cde
f(x)	0.32	0.25	0.43

Combine a, b as "ab"

x	ab	cde
f(x)	0.57	0.43

Two letters, base case

```
graph TD; Root(( )) --- ab((ab)); Root --- cde((cde)); ab --- a((a)); ab --- b((b)); cde --- c((c)); cde --- de((de))
```

So, let us look at this example that we had earlier, so here the two lowest frequency letters d and e. So, we merge them into the new letter d, e and this is a frequency 0.23, because it is 0.18 plus 0.05. Now, these two are a two lowest letters, so we merge them and we get a new letter c, d, e of cumulative frequency 0.43, which is some of all the frequencies are that two values.

Now, turns out that, these two are the smaller two. So, I am then the letter a, b and now, I reach my base case where we have exactly two letters. So, I can set off the trivial tree with these two letters, label 0 and 1. And now I work backwards, so the last thing that I did was to merge a and b, now I will take this a and b thing and split it into a and b.

(Refer Slide Time: 19:30)

Huffman's algorithm

x	a	b	c	d	e
f(x)	0.32	0.25	0.20	0.18	0.05

Combine d, e as "de"

x	a	b	c	de
f(x)	0.32	0.25	0.20	0.23

Combine c, de as "cde"

x	a	b	cde
f(x)	0.32	0.25	0.43

Split "ab" as a, b

x	ab	cde
f(x)	0.57	0.43

Two letters, base case

```
graph TD; Root(( )) --- a1((a)); Root --- b1((b)); a1 --- a2((a)); a1 --- b2((b)); b1 --- c((c)); b1 --- de((de)); c --- c1((c)); c --- de1((de)); de1 --- d((d)); de1 --- e((e));
```

I will split a, b as a and b, I will get this print, then the previous step was to combine c, d e into c and d, e. So, I am going to the split this c and d, e.

(Refer Slide Time: 19:40)

Huffman's algorithm

x	a	b	c	d	e
f(x)	0.32	0.25	0.20	0.18	0.05

Combine d, e as "de"

x	a	b	c	de
f(x)	0.32	0.25	0.20	0.23

Split "cde" as c, de

x	a	b	cde
f(x)	0.32	0.25	0.43

Split "ab" as a, b

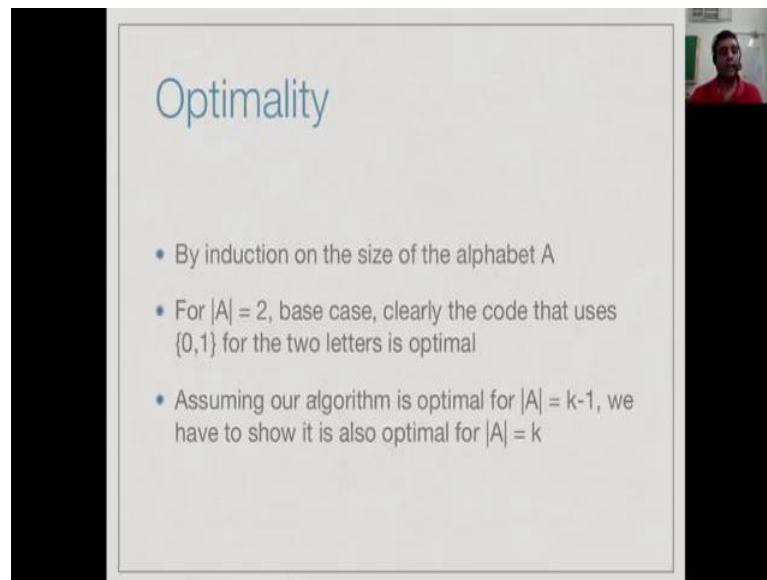
x	ab	cde
f(x)	0.57	0.43

Two letters, base case

```
graph TD; Root(( )) --- a1((a)); Root --- b1((b)); a1 --- a2((a)); a1 --- b2((b)); b1 --- c((c)); b1 --- de((de)); c --- c1((c)); c --- de1((de)); de1 --- d((d)); de1 --- e((e));
```

And finally, I am going to split this up is d and e. So, this is Huffman's algorithm and by recursively combining the two lowest frequency nodes, and then taking the composite node and splitting them back up to it is.

(Refer Slide Time: 19:56)

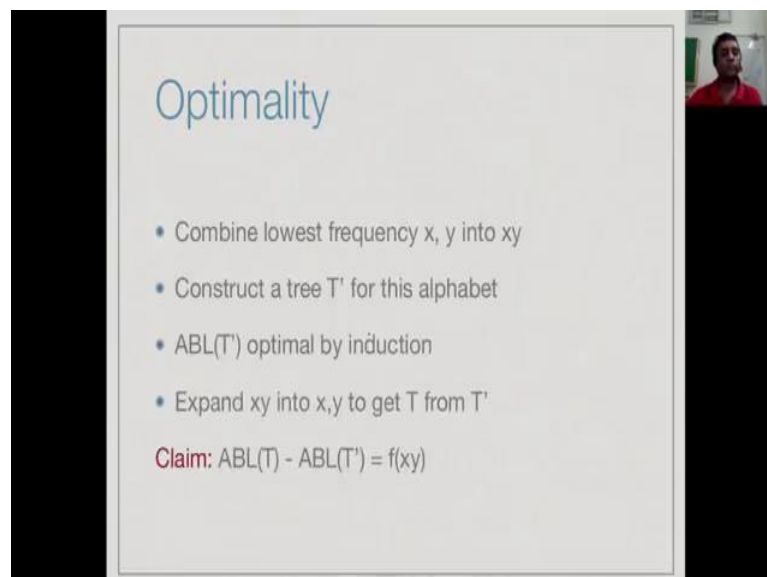


Optimality

- By induction on the size of the alphabet A
- For $|A| = 2$, base case, clearly the code that uses $\{0,1\}$ for the two letters is optimal
- Assuming our algorithm is optimal for $|A| = k-1$, we have to show it is also optimal for $|A| = k$

So, to show that this algorithm is the optimal, we go by the end of the size in the algorithm. Now, clearly when I have only two letters, I cannot do any better than assign the one of them 0 and one of them 1, so the base case is optimal. So, we will assume that this optimal for k minus 1 letters and now show that also optimal for k letters.

(Refer Slide Time: 20:17)



Optimality

- Combine lowest frequency x, y into xy
- Construct a tree T' for this alphabet
- $ABL(T')$ optimal by induction
- Expand xy into x, y to get T from T'

Claim: $ABL(T) - ABL(T') = f(xy)$

So, recall that, we went to k minus 1 by combining the two lowest frequency letters as 1, constructing an optimal tree for these smaller alphabet and then expanding the x, y get a new. So, the claim was when I go from the tree over k minus letters to the tree over k

letters, the cost is going to increase, but this cost is going to be fixed by whichever letters I choose to contract.

So, if I choose x and y to merge to go from T to T' , then the amount by which the average bits per letter is going to change is exactly the frequency of this combined letter $f(x+y)$. So, the deterministically fixed by which one I choose, then even though I do know the cost of the trees directly, I can tell you that the cost is going to be different by this some amount.

(Refer Slide Time: 21:08)

Optimality

$ABL = \sum_{z \in A} f(z) \text{depth}(z)$

Claim: $ABL(T) - ABL(T') = f(xy)$

- From T' to T , only xy , x , y change contribution to ABL
- Subtract $\text{depth}(xy)f(xy)$, add $(1 + \text{depth}(xy))(f(x) + f(y))$
- $f(xy) = f(x) + f(y)$, so $\text{depth}(xy)f(xy) = \text{depth}(xy)(f(x) + f(y))$
- Hence $ABL(T)$ is bigger than $ABL(T')$ by $f(x) + f(y) = f(xy)$

This is not very difficult to prove, so in that summation that we had, so in the tree notation remember this depth of z is the same as the length of the encoding of set, because the depth exactly reflects the length of the string used to encode. So, this is our ABL calculation. Now, for every node other than the x and y , there are exactly the same position in T and T' .

So, this summation with terms do not change, so the only changes are these three nodes. So, what I will do and going from T' to T is I will remove this contribution of the combined letter x and y . And then at a next level which is 1 plus the depth of the x and y , I will add the node $f(x)$ and I will add the node $f(y)$ and I will get the components x and y , $f(x)$ times that plus x and y times, I am subtracting this amount on the left, then I am adding this amount on the right.

So, now, actually f of $x y$ is nothing but $f x$ plus $f y$. So, this left hand side term is actually this right hand side component depth of $x y$, times of x plus y . So, I am subtracting this and adding it back, so they cancel each other, so therefore, all I am left with is one times that the x plus y which is $f x$ by $f y$ or $f x y$. So, therefore I am going from T prime to T , the crucial thing is only depend for which letters I have contract, that fix is the difference unique.

(Refer Slide Time: 21:51)

Optimality

- Suppose there is another tree S with $ABL(S) < ABL(T)$
- Can shuffle labels of max depth leaves in S , so that lowest frequency pair x and y label siblings
- Merge, x and y into xy and contract S to S'
- S' is over same alphabet as T' , T' is optimal by induction, so $ABL(T') \leq ABL(S')$
- $ABL(S) - ABL(S') = ABL(T) - ABL(T') = f(xy)$, so $ABL(T) \leq ABL(S)$ as well, contradiction!

Handwritten notes on the slide:

$$S' + f(xy) = S$$

$$T' + f(xy) = T$$

Now, let us assume that, we know how to solve all k minus 1's say alphabets efficiently and we have done is recursive thing to construct the tree for size k . Suppose, this is another strategy would produce the better tree for size k , so this another tree candidate tree S produce by some different strategy, who is average bits for letter is strictly better than the one that we construct recursive.

Now, in S , we know for sure that these two letters that we use the in recursive construction x and y . So, they occur somewhere the bottom of this tree. So, this is my tree S , these must be leaf nodes, because they have the lowest frequency is over all the letters, so must be a maximum depth, that they may not be next to each other. But, it does not matter, because since they are both and maximum depth, I can move them around, this step, I can move letters around, I can reassign the two other leaf to a sender.

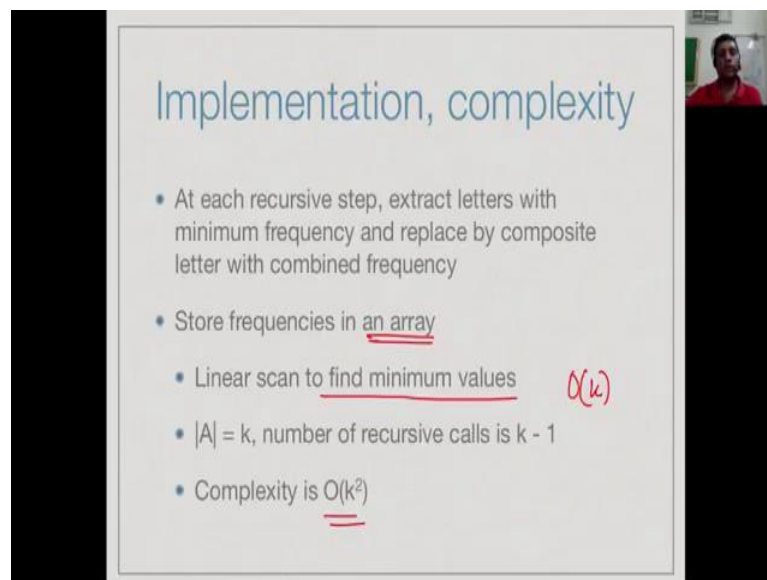
Such that, I come with the configuration I come to new S , I call it S again, where actually have x and y together. I can assume that the S , that has this optimal property, which is

better in the tree as constructed, actually as x and y a sibling leaves other maximum depth. Now, what I am going to do is this S , I am going to concretely fuse this and get an S prime.

So, this explain will be our k minus 1 letters except instead of doing this by a recursive call, I am actually taking a call concrete tree over k letter and I am actually compressing to two nodes into 1 and call in it to tree over k minus 1 data set. But, because this over k minus 1 letters and these are represent the encoding, it cannot be any better than the encoding that I recursively computed for k minus 1 letters, because I am assumed by induction by that algorithm those efficiently for k minus 1 letters.

So, so S prime is no better than T time, but as prime plus f of x y is S , T prime plus f of x y is T . So, the different T prime and T and S prime S and exactly the same. So, the S prime is a then T prime, then S cannot any better than T . So, it was a contradiction to assume that as I strictly better than T is this 2's that strategy of recursively computing T is optimal for all k .

(Refer Slide Time: 25:21)



Implementation, complexity

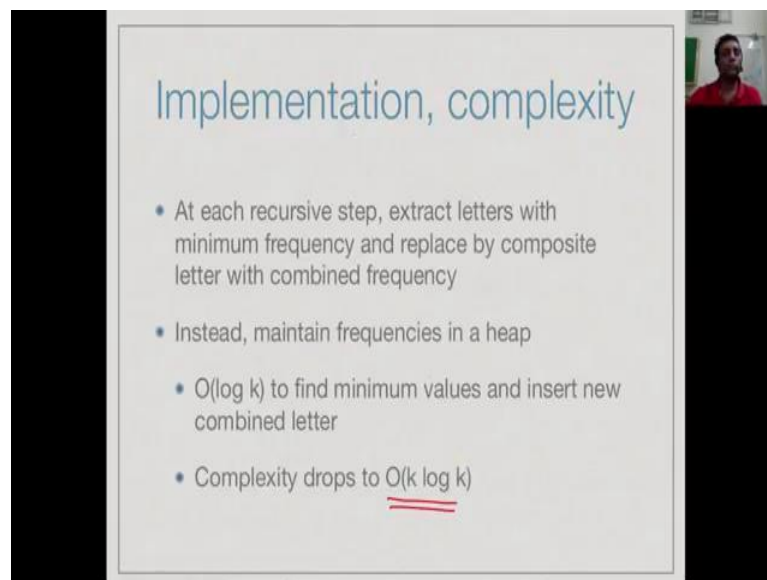
- At each recursive step, extract letters with minimum frequency and replace by composite letter with combined frequency
- Store frequencies in an array
- Linear scan to find minimum values $O(k)$
- $|A| = k$, number of recursive calls is $k - 1$
- Complexity is $O(k^2)$

The word about the implementation, so what we have to do is k minus 1 time, we have to merge this two minimum values and compute the recursive solution, bottle neck, what will make is finding the minimum values. If you use an array, then as we know scan the array instant to find the minimum values, remember that the minimum of values keep changing, I cannot short it one send for all. Because, each time I combine two letters, I

can use a new letter into my array with a new minimum value which was not there before and not a new value, which may not there before it is may or may not be the minimum.

So, each state I have to find the minimum, so it is an order case can each time, so linear scan and I do this appropriate k these times. So, I get order case two, but it should be fairly cleared into see that this bottle neck can be got around by using a heap, where there is precise what heaps are good at finding the minimum.

(Refer Slide Time: 26:13)

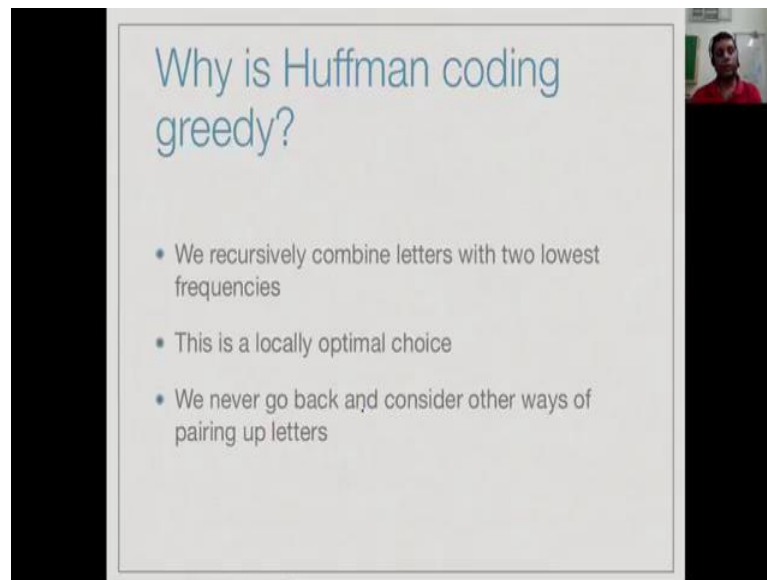


Implementation, complexity

- At each recursive step, extract letters with minimum frequency and replace by composite letter with combined frequency
- Instead, maintain frequencies in a heap
- $O(\log k)$ to find minimum values and insert new combined letter
- Complexity drops to $O(k \log k)$

So, if I maintain the frequencies is not at as a heap, then the order $\log k$ time, I can find the minimum values and then I can insert back the new composite letter also into heap in to $\log k$ time. So, each iteration takes some $\log k$, and so I am improving from k square to $k \log k$.

(Refer Slide Time: 26:33)



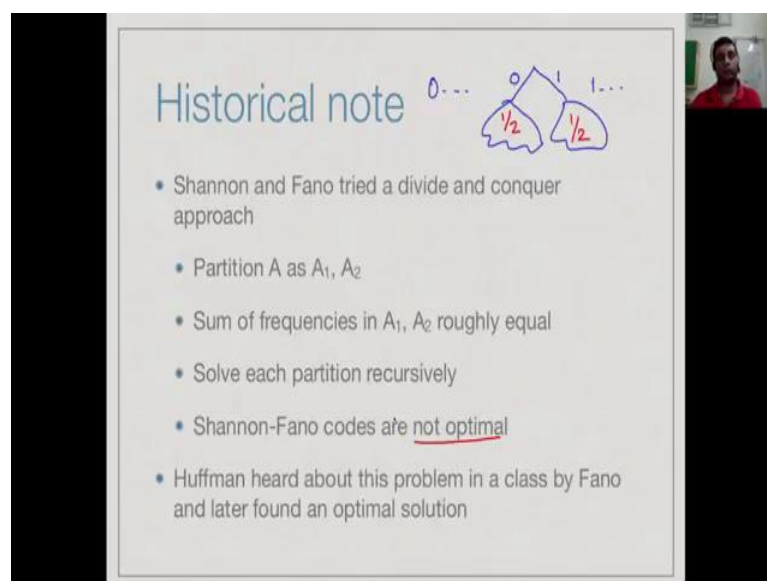
Why is Huffman coding greedy?

- We recursively combine letters with two lowest frequencies
- This is a locally optimal choice
- We never go back and consider other ways of pairing up letters

So, recall that, we mention that the beginning that this is going to be a last greedy algorithm. So, y is greedy, well because every time, we make a recursive call, we are deciding to combine two nodes into one and the two letters we are choose always once with the lowest frequencies. Now, what is to say, that we could not to better by choosing the lowest then the third rows per, but we now a try, we only try to lowest to the second rows.

So, we make a locally optimal choice and we keep going with choice, never going back to the visited, and finally we get a global solution. Now, we are prove that this global solution is actually optimal and we have to do that, because there is no other reason is expect that I making a short sided choice, at the current time take the two worst frequencies and combine them, that you are always going to get the best solution. So, this is very much the greedy is letter.

(Refer Slide Time: 27:25)



Historical note

- Shannon and Fano tried a divide and conquer approach
- Partition A as A_1, A_2
- Sum of frequencies in A_1, A_2 roughly equal
- Solve each partition recursively
- Shannon-Fano codes are not optimal
- Huffman heard about this problem in a class by Fano and later found an optimal solution

So, finally a brief historical note, so Clot Shannon is the father of information theory and when, we are looking at these problems around 1950 are, so they where face to this problem are finding and efficient. So, Shannon and Fano, proposed the divide and conquer thing, so what us it was let us look at the encoding of the alphabet. So, some of them are going to start with 0, some other going to start with 1.

Everything which is in the left sub tree of this coding tree that we construct is going to have a code of the found 0 are something, something, everything on the right is going to have something at the found once. So, it seem intuitive to them, then divide and conquer strategy is good, what you can put letters on this side, such that the occupied roughly of the frequency weight of the total alphabet. That is a frequencies of all the letters, who's encoding start with 0, their frequencies added to roughly hard and the other one also to hard.

So, split the alphabet in the two of equal weight assign some of them to start with 0's, the other to start with 1, then I recursive it is solve this t. So, a partition is A_1, A_2 , sums of the frequencies in each other sets are roughly equal, solve them recursively. Unfortunately, this is not guaranty to generate an optimal encoding, you can come up to the example, where you can do this and then end of the something which you can improve by doing some other.

So, it turned out the Huffman was a graduates student in a course of Fano, he heard about this problem and we thought about it, and after a few years he came up with this clever algorithm which we are done in it.