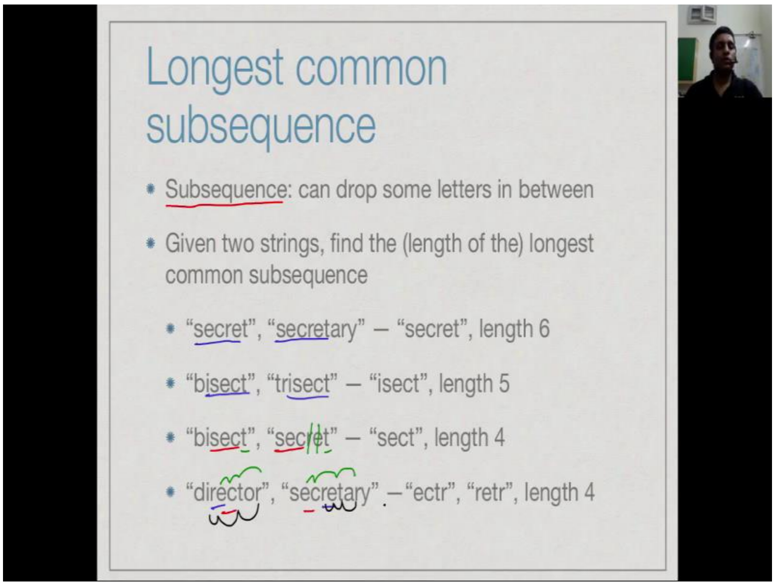
So, we have seen earlier that if we just look blindly at every position and try to scan the word starting that position, we get something which is an order m, n square. now, this solution when require as to fill in the table of size m time n, so obviously, every entries in the m times n table, we just have to look at a neighbors to fill it up. So, it is a constant time operation. So, m times n entries, we fill it m times n times. So, we have an order n 1, m n. If use dynamic programming, we have done it, but if use memoization also, you will get the same answer, all though remember, there is a recursive calls might cost you and terms of actual implementation time.

(Refer Slide Time: 13:04)



Longest common subsequence

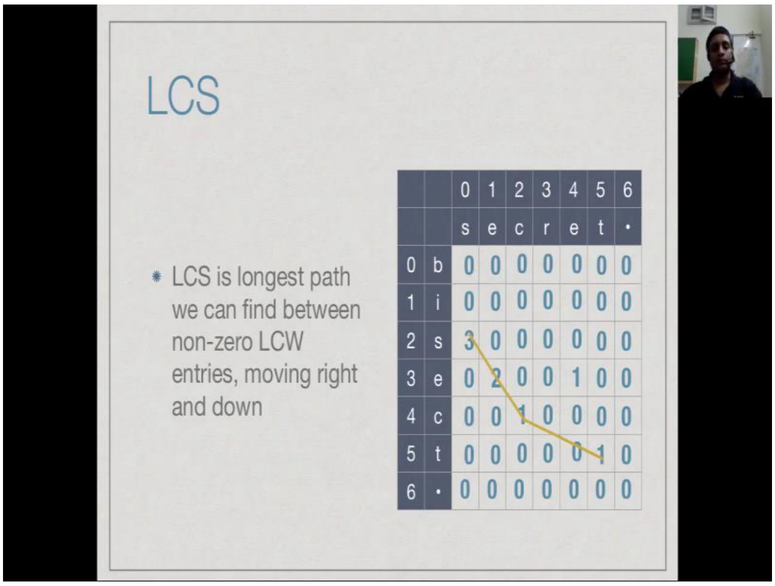
- Subsequence: can drop some letters in between
- Given two strings, find the (length of the) longest common subsequence
  - "secret", "secretary" — "secret", length 6
  - "bisect", "trisect" — "isect", length 5
  - "bisect", "secret" — "sect", length 4
  - "director", "secretary" —"ectr", "retr", length 4

So, we can now look at a slightly more general problem than longest common subword in one which is more interesting computationally. So, what if we do not look for an exact match, but we allow a self should drop some letters. So, we have a subsequence not a sub word, it is allows us to drop some letters and now, if you want to know, after dropping some letters, what is the longest match we can find.

So, now, our earlier example, some of them are the same, like in this case without dropping any letter I can could get 6, I cannot improve it, same we will bisect, I cannot improve it. But, now if I look at bisect and secret, earlier we only had a length 3 match sec, sec, but now I can extend match the length 4, because here if we add it t, here I can drop two letters and get it t. So, I can actually get a match which is length 4, likewise in these two director and secretary, earlier we had re, re and we had ec, ec.

But, now by doing this topic I can get ectr, like get ectr, similarly I can get retr, I can get retr. So, if we allow a self's to drop letter as we can get longer sequences that match and this is called longest common subsequence.

(Refer Slide Time: 14:32)



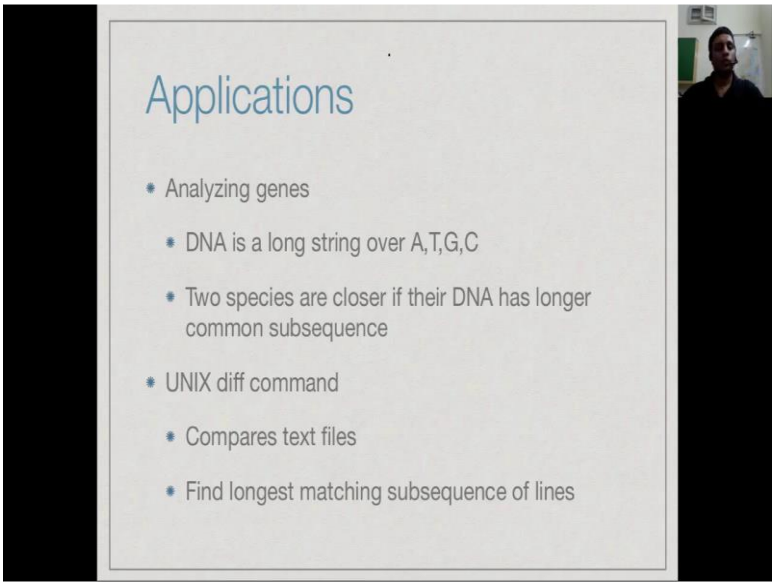
LCS

- LCS is longest path we can find between non-zero LCW entries, moving right and down

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	0	0	0	0	0	0	0
1	i	0	0	0	0	0	0	0
2	s	3	0	0	0	0	0	0
3	e	0	2	0	0	1	0	0
4	c	0	0	1	0	0	0	0
5	t	0	0	0	0	0	1	0
6	•	0	0	0	0	0	0	0

So, one we have thinking about it in terms of our earlier longest common subword is that I can now, if I match the certain segment, I can continue to match to the right. So, I can let both the indices go forward. So, I can go down and right in the grid and look for in other place where there is a match. So, earlier we had a three level match between a sec, and then we add a one level match between d. So, I can combine these two like a four level longest common subsequence, but we will do the subsequence calculation in a much more direct way there itself.

(Refer Slide Time: 15:04)



Applications

- Analyzing genes
  - DNA is a long string over A,T,G,C
  - Two species are closer if their DNA has longer common subsequence
- UNIX diff command
  - Compares text files
  - Find longest matching subsequence of lines

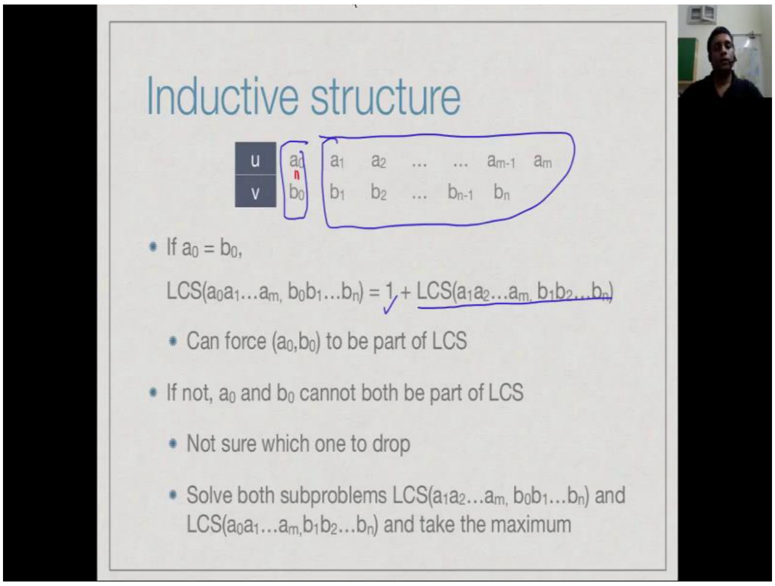
So, before we proceed it useful to look at why the longest common subsequence problem is interesting. So, one of the area is an bio informatics. So, biologist are interested in identifying, how close two species are each other in the genetic sense. So, if we look at are DNA, our DNA is basically a long string over an alphabet of size 4. So, these are 4 proteins, it form a DNA, the abbreviated are A, T, G, C. So, now, if we look at two strings of DNA and natural way to compare, how close they are each other is to ask how much, how well they aligned features drop of few things here and there.

So, it could be one species other few extra genes and other species as a few extra gene something else and if drop those genes, then everything else the same, but these genes may occur, it different places in two species. So, it is not that there is a common part, and then there is an extra part, it rather than the new genes are interspersed among the other genes.

So, we need to know, if we can drop of few genes in one and few genes in another can be over lap. So, that is the longest common subsequence problem, if you use UNIX or LINUX are any related operating system, there is a command code DIFF, which allows you to check with the two text files are the same or find the minimal difference between them.

So, the DIFF command also does longest common subsequence, it reach each line as a character and it says, what is the minimum number of lines, I can drop between these two files, 4 I can match them, and then it tells you how to insert things back. But, basically it is doing the longest common subsequence calculation to tell you, how close two text files are to each other. So, there are plenty of applications for this longest common subsequence problem.

(Refer Slide Time: 16:52)



Inductive structure

$$\begin{array}{c|ccccccc} u & a_0 & a_1 & a_2 & \ldots & \ldots & a_{m-1} & a_m \\ v & b_0 & b_1 & b_2 & \ldots & b_{n-1} & b_n & \end{array}$$

- If $a_0 = b_0$,

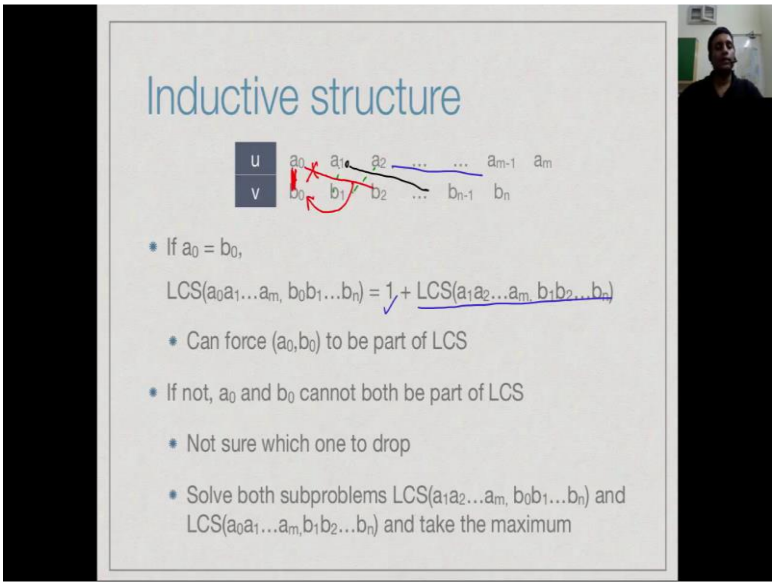
  $LCS(a_0a_1\ldots a_m, b_0b_1\ldots b_n) = 1 + \underline{LCS(a_1a_2\ldots a_m, b_1b_2\ldots b_n)}$

  - Can force (a_0,b_0) to be part of LCS
- If not, a_0 and b_0 cannot both be part of LCS
  - Not sure which one to drop
  - Solve both subproblems $LCS(a_1a_2\ldots a_m, b_0b_1\ldots b_n)$ and $LCS(a_0a_1\ldots a_m,b_1b_2\ldots b_n)$ and take the maximum

So, let us try understanding inductive structure of this longest common subsequence problem directly, not throw the longest common subword. So, the first thing to note is that if I am looking for this longest common subword between these two, supposing I find that a 0, in fact equal to b 0. Now, I claim that, I should combine them in the solution, and then look for a solution in the rest.

So, I should do something, where I say that there is one match a 0 equal to b 0, and then I must find the list. So, this requires a little of bit of an argument, because it could be that this is not the optimum. So, remember this is the bit like a greedy thing; we are saying that, because a 0 equal to b 0 match it up, and then proceed.

(Refer Slide Time: 17:44)



Inductive structure

u	a_0	a_1	a_2	…	…	a_{m-1}	a_m
v	b_0	b_1	b_2	…	b_{n-1}	b_n	

- If $a_0 = b_0$,

  $LCS(a_0a_1 \ldots a_m, b_0b_1 \ldots b_n) = 1 + LCS(a_1a_2 \ldots a_m, b_1b_2 \ldots b_n)$

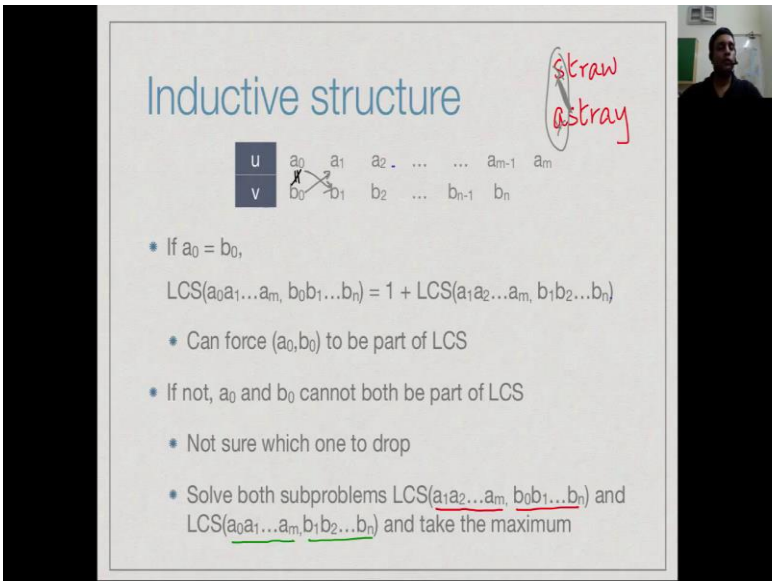
  - Can force (a_0,b_0) to be part of LCS
- If not, a_0 and b_0 cannot both be part of LCS
  - Not sure which one to drop
  - Solve both subproblems $LCS(a_1a_2 \ldots a_m, b_0b_1 \ldots b_n)$ and $LCS(a_0a_1 \ldots a_m, b_1b_2 \ldots b_n)$ and take the maximum

So, one might argue that, this is not the way, I want to go, supposing a 0, in fact, should be match to b 2, that would be the best solution. So, it is not good idea match a 0 to be b 0, but now notice that if a 0 is match to b 2, because it is the subsequence, then anything to the write, say a 1 is match to something it must be further to the right, these lines, I cannot have anything which process like this, I cannot any match with process.

Because, they must occur in the same sequence, so if a 0 match to b 2, a 1 something match into the right, a 2 must match something still for the right and so on. So, all these matches go from bottom from the top word to the bottom word without processing each other. So, now, if I take this solution and I know the a 0 and b 0, then I can actually move this arrow here and make match like this and remove this, what will this do, it disturb the original solution by no long using a 0 match a b 2, but, a 0 match to b 0.

But, in terms of the quality of the solution, the number of matches this same, earlier a 0 matches somewhere else, now it match to b 0, the length of the longest common subsequence does not change. So, turning the arguments backward, it says that therefore if a 0 equal to b 0, it is very safe to assume that a 0, b 0 forms part the solution and proceed to the rest of the problem. So, we can look at the subproblem from a 1 and b 1. So, this is the first case, the first case says, that we can look at if a 0 equal to b 0, we can look at this subproblem a 1 to b 1.

(Refer Slide Time: 19:15)



Inductive structure

u	a_0	a_1	a_2	…	…	a_{m-1}	a_m
v	b_0	b_1	b_2	…	b_{n-1}	b_n	

- If $a_0 = b_0$,

  $LCS(a_0a_1\ldots a_m, b_0b_1\ldots b_n) = 1 + LCS(a_1a_2\ldots a_m, b_1b_2\ldots b_n)$
  - Can force (a_0,b_0) to be part of LCS
- If not, a_0 and b_0 cannot both be part of LCS
  - Not sure which one to drop
  - Solve both subproblems $LCS(\underline{a_1a_2\ldots a_m}, \underline{b_0b_1\ldots b_n})$ and $LCS(\underline{a_0a_1\ldots a_m},\underline{b_1b_2\ldots b_n})$ and take the maximum

Now, if it is not equal, then one is not sure what you do, it is not s sound idea a to drop both. So, for instant supposing I have something like straw and astray, then just because the S does not match the a, does not mean that I should in both of them, I should keep that S alive to match with next S. So, both cannot be there, because they do not be each other and S match it something on the right and a cannot match as something for it.

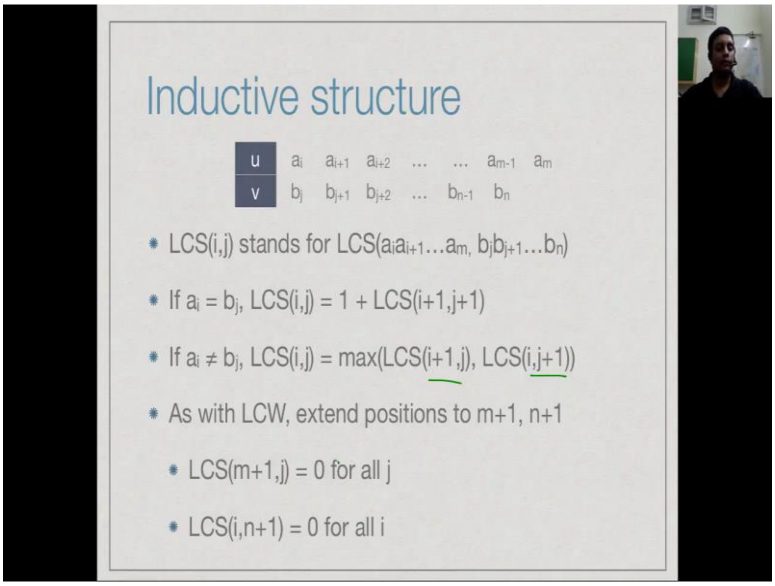
If they are both different, if they are not equal, then this must match something there and this must match something here and both cannot happen. because, they will cross, this we said not allowed, so only one of them can match something to the right on the other word, but we do not know which one. So, this is what the general principle of these inductive solutions is that you do not write to choose one or the other.

So, if I leave b 0 and I drop a 0, then I get a solution, I get a subproblem which has a 1 to a m and b 0 to b m. If on the other hand, I keep a 0 and a drop b 0, then a 0 to a m remains, but knows b 1 to b n, these are my two possible subproblems knowing that both a 0 and b 0 cannot be there. But, I should allow one of them to be there, otherwise I made how sub optimize solution.

So, dropping one of them, it is like in a job schedule case, for you say that, if this job with this something else is not there, if a 0 is there, b 0 is not there, b 0 is there, a 0 is not there, but beyond that you cannot say. So, therefore, I solve both of these problems in principle and take them maximum, the better of them.

So, this is the inductive structure, either the first letter matches which case I include, it my sequences and solve the remaining subproblem. Or, the first letter does not match in which case is generate two subproblems, one by propping each of the letters in term and I take the maximum of the true.

(Refer Slide Time: 21:18)



Inductive structure

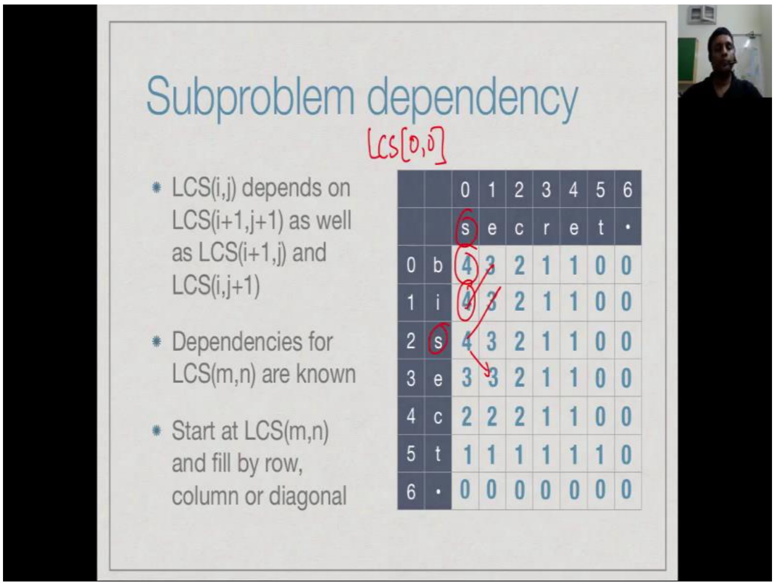
u	a_i	a_{i+1}	a_{i+2}	…	…	a_{m-1}	a_m
v	b_j	b_{j+1}	b_{j+2}	…	b_{n-1}	b_n	

- LCS(i,j) stands for LCS($a_ia_{i+1}\ldots a_m$, $b_jb_{j+1}\ldots b_n$)
- If $a_i = b_j$, LCS(i,j) = 1 + LCS(i+1,j+1)
- If $a_i \neq b_j$, LCS(i,j) = max(LCS($\underline{i+1}$,j), LCS(i,$\underline{j+1}$))
- As with LCW, extend positions to m+1, n+1
  - LCS(m+1,j) = 0 for all j
  - LCS(i,n+1) = 0 for all i

So, as usual let LCS i comma j, stand for the LCS of the problem starting at a i and b j. So, if a i equal b j as we say, LCS of i j is 1 plus LCS of i plus 1 j plus 1, this says, we will assume that a i equal to b j is included not solution. And then proceed with the rest of the input, if it is not, then I have to drop one of them. So, either I will look at LCS i plus 1 comma j for LCS i j plus 1, I will look at both and then I take the maximum of these two and there is no other thing, because the current would not does not match.

So, as with the longest common subword problem, we will extend that position is to beyond the word to indicate the word is over. So, we will go from 0 to m plus 1 and 0 to n plus 1 and when, we have least m plus 1 or n plus 1, then the LCS problem will give a 0, because they cannot be a common subsequence, since one of the words going to be empty.

(Refer Slide Time: 22:19)



Subproblem dependency

LCS[0,0]

- LCS(i,j) depends on LCS(i+1,j+1) as well as LCS(i+1,j) and LCS(i,j+1)
- Dependencies for LCS(m,n) are known
- Start at LCS(m,n) and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	4	3	2	1	1	0	0
1	i	4	3	2	1	1	0	0
2	s	4	3	2	1	1	0	0
3	e	3	3	2	1	1	0	0
4	c	2	2	2	1	1	0	0
5	t	1	1	1	1	1	1	0
6	•	0	0	0	0	0	0	0

So, the subproblem dependency in LCS is a little more complicated than in LCW, LCW we only had these dependency, that is we said that, i j depended i plus 1 j plus 1. But, now we are also dependency i plus 1 j and i j plus 1. So, we have a dependency coming to the right and from the low as well. So, we have a three way dependency as we saw all these values are going to be 0 here. So, this is the first non trivial value that we can compute, because all it see neighbors are nowhere else are around.

So, if I look at here for example, the bottom neighbors not, if I look here the neighbors not. So, LCS m comma n, 5 comma 5 in this case, the value is available to compute, because everything around it the three dependence squares around it are all populate. So, I can do that and then I can again I do row by row. Once I got this, I can do this, I will have all three values, once I do this, I can go or I can go left, I can do this and so on, I can do diagonally, so let us do it column by column.

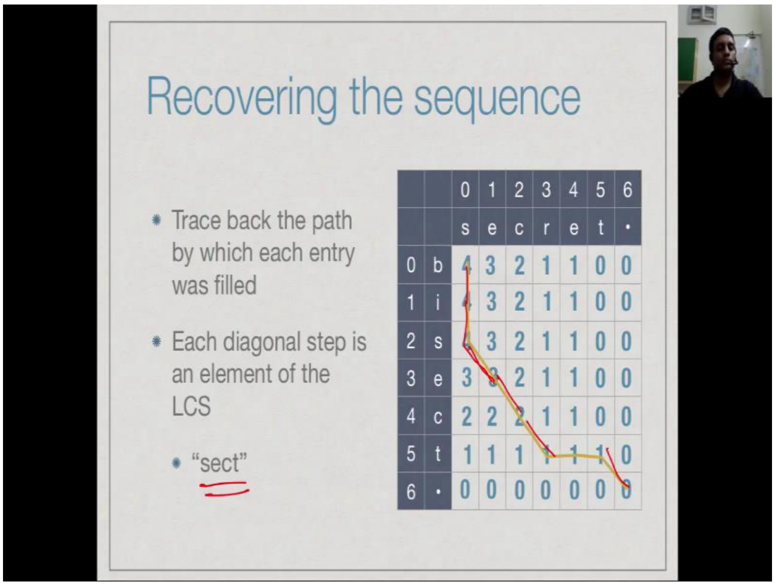
So, we start with the base case, where the LCS at the boundary 0, because you cannot have a longest common subsequence with an empty word. Then, we fill up the first column and here we get a 1, because when the two match, we have 1 plus cso, i plus 1, j plus 1. Now, we have from differences, so when it does not match like when c, so if I look at c that e they do not match, then what I am suppose to do, I am suppose to take the maximum of these two.

So, this is the maximum of these 2, I get to 1, now since I take the maximum of these 2, I get to 1, maximum of these 2, I get to 1 and so on. So, the longest common subword problem, it say if the current let it does not match, then I get a 0, here is not there, because I am allow to drop this set and go head. So, therefore, this one proper get along this one.

Similarly, the one property to get this along the previous column, because nothing is an r. now, some get an additions, when I reach c and c, it is 1 plus i plus 1 j plus 1. So, we get it 2. Likewise, when I go to the next column, when I reach e and e, it is 1 plus i plus 1 j plus 1, so I get the 3. And finally, when I reach this S and S, I get this 4 and now, this 4 propagate, because here I take the max of these 2.

So, I get this 4 max of these 2, I get 4. So, in this particular case actually LCS of 0 comma 0 is my answer, remember in LCW add look around the in the whole grid to find out, we are the maximum was in LCS, you do not want to do that. The value you get 0 comma 0 is acts with answer you are looking for...

(Refer Slide Time: 25:07)



Recovering the sequence

- Trace back the path by which each entry was filled
- Each diagonal step is an element of the LCS
  - "sect"

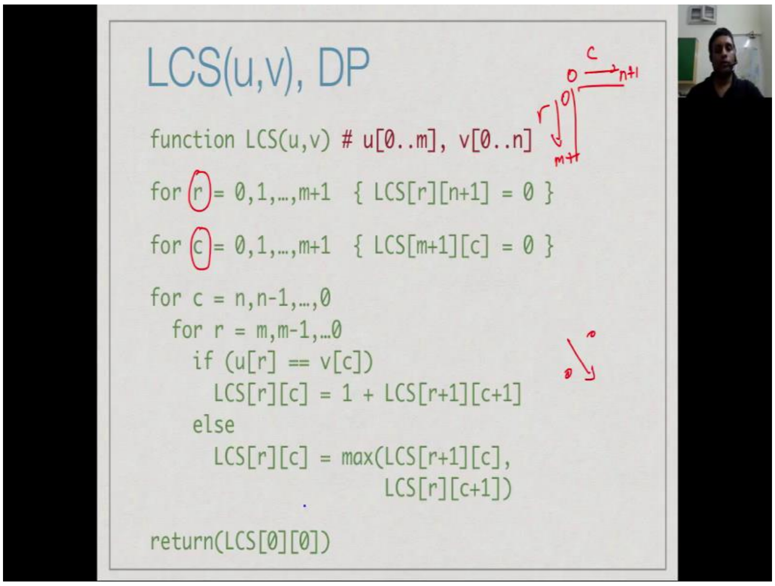
		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	4	3	2	1	1	0	0
1	i	4	3	2	1	1	0	0
2	s	4	3	2	1	1	0	0
3	e	3	3	2	1	1	0	0
4	c	2	2	2	1	1	0	0
5	t	1	1	1	1	1	1	0
6	•	0	0	0	0	0	0	0

And now as before you can trace back the path, why was each value filled, was it filled, because it to 1 plus i plus 1, j plus 1 or it goes to fill, because max with other two networks, if so which was the match. So, which was very clear this 4 came, because it was a max, because S is not equal to b. So, it came from below and this 4 also came from below S is equal to S.

So, this is came from here and so on, so you can trace out this value and everywhere where you have this diagonal, it was there, because the value is matched. So, there are value is 4 and there is exactly four diagonal steps, this one, this one, this one. And then one of the bottom, these are the four matches which constitute the longest common subsequence and you can read it of that the thing and this is forms the sequence sect.

So, as we say before provided you can compute the answer numerically, you can go back and retrace that computation and figure out the witness and it is called and which word actually which subsequence actually gives has to be sanction.

(Refer Slide Time: 26:07)



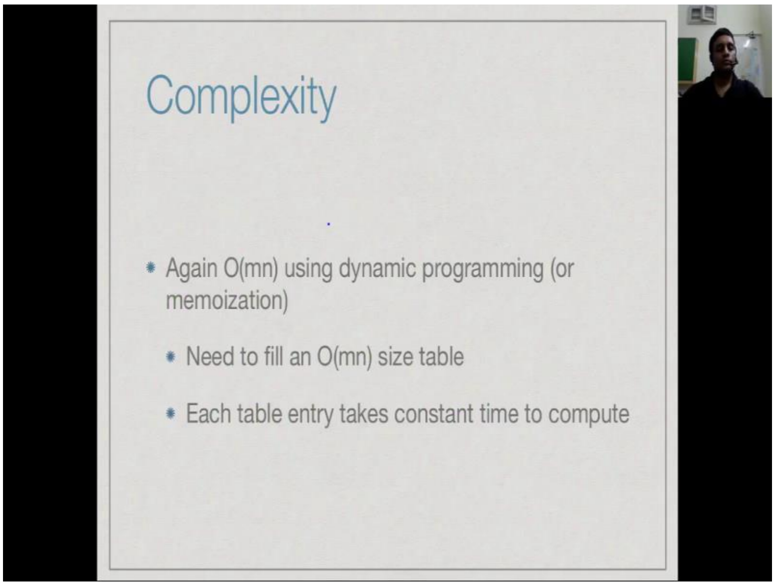
LCS(u,v), DP

```
function LCS(u,v) # u[0..m], v[0..n]

for r = 0,1,…,m+1  { LCS[r][n+1] = 0 }

for c = 0,1,…,m+1  { LCS[m+1][c] = 0 }

for c = n,n-1,…,0
  for r = m,m-1,…0
    if (u[r] == v[c])
      LCS[r][c] = 1 + LCS[r+1][c+1]
    else
      LCS[r][c] = max(LCS[r+1][c],
                      LCS[r][c+1])

return(LCS[0][0])
```

So, the code for the LCS is little bit simpler then for LCW, because we do not have to keep track of these maximum value and varied occurs, because we only need to find the value at 0 c. So, as before we use r and c to be little clearer that rows go this way and columns go this way. So, column is go from 0 to n plus 1 and rows go from 0 to m plus 1.

So, we initialize the boundary to be 0, and then we do column by column row by row from bottom to top, if the two are equal, then you add 1 plus the value of bottom. If the two are not equal, I take the maximum of these two values and finally, when this where is are filled out, I written the value, it is 0 comma 0.

(Refer Slide Time: 26:53)



Complexity

- Again O(mn) using dynamic programming (or memoization)
  - Need to fill an O(mn) size table
  - Each table entry takes constant time to compute

So, this is similar to LCW, we basically fill out in m, n size table each entry in the table is easy to compute takes only constant among look at the three neighbors. And therefore, overall using dynamic programming, we have demonstrated an order m n algorithm, we can also use memoization at the cost of recursion.