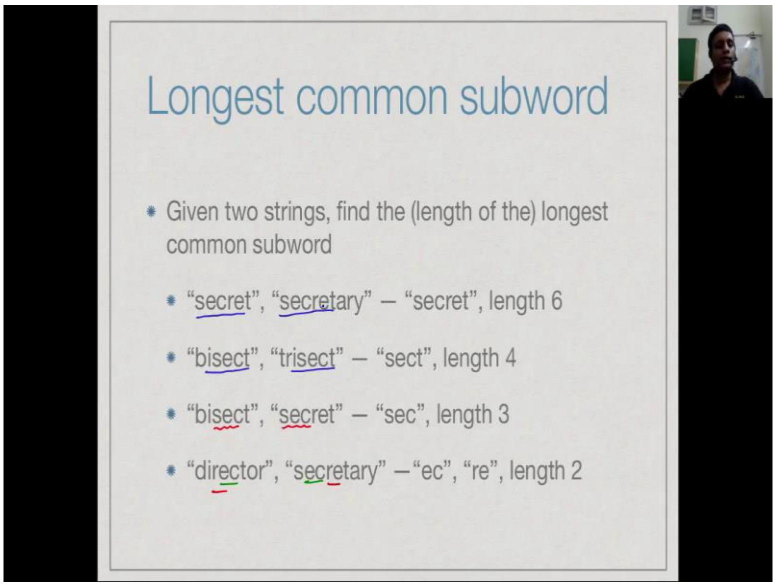
Design and Analysis of Algorithms, Chennai Mathematical Institute
Prof. Madhavan Mukund
Department of Computer Science and Engineering,

Week- 07
Module - 04
Lecture - 47

Common Subwords and Subsequences

Let us now turn to the problem of finding Common Subwords and Subsequences between two sequences.

(Refer Slide Time: 00:09)



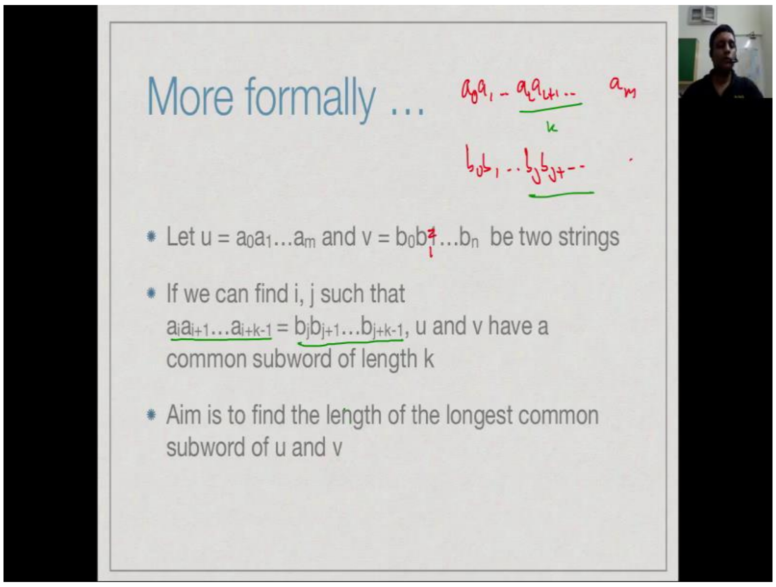
Longest common subword

- Given two strings, find the (length of the) longest common subword
  - "secret", "secretary" — "secret", length 6
  - "bisect", "trisect" — "sect", length 4
  - "bisect", "secret" — "sec", length 3
  - "director", "secretary" —"ec", "re", length 2

So, the first problem we look is what is called a longest common subword problem. So, let us suppose we are given two words for the moment, let us just assume there going in English or a language like that, we given two strings and we want to find the length of the longest common subword between them. In a subword, it is just a segment, so far instants, if I have secret and secretary and the longest common subword is just the prefix secret and it has length 6, between bisect and trisect, I have this common segment isect.

If I bisect and secret on the other hand, then the common subword is of length 3, namely sec and if I have director and secretary, then there are actually only two length subwords and there are two of them, so ec occurs in both of them and so does re. So, we may have more than one candidate for the longest common subword, but our main through goal at the moment is to compute not the subword itself, but the length, we will see how we can recover this subword quit easily from the length calculation. So, the focus at the moment is to get the length of the subword.

(Refer Slide Time: 01:28)



More formally …

$\underbrace{a_0 a_1 \ldots a_i a_{i+1} \ldots}_{k} \quad a_m$

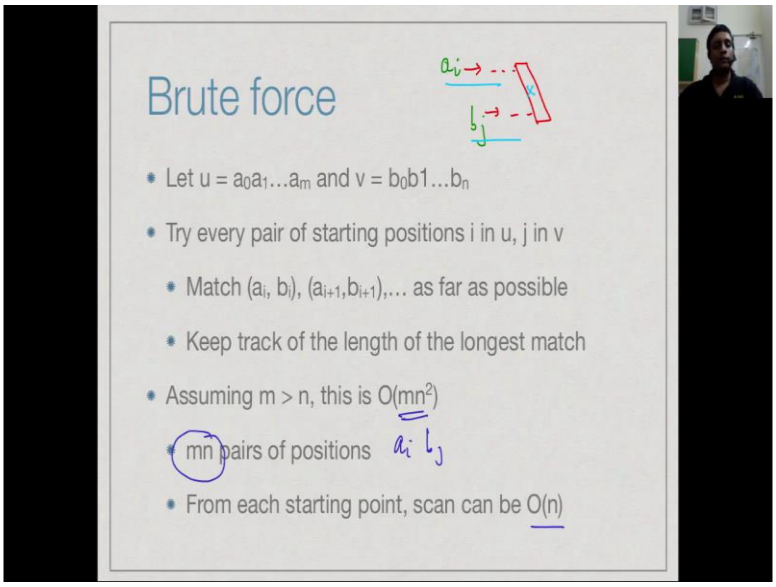
$b_0 b_1 \ldots b_j b_{j+1} \ldots$

- Let u = $a_0a_1 \ldots a_m$ and v = $b_0b_1 \ldots b_n$ be two strings
- If we can find i, j such that $a_ia_{i+1} \ldots a_{i+k-1} = b_jb_{j+1} \ldots b_{j+k-1}$, u and v have a common subword of length k
- Aim is to find the length of the longest common subword of u and v

So, let us look at this thing more formally, so we have two words a 0 to a m and b 0 to b n, and now what we want to say is that if I write out a 0, a 1, then have a position a i, a i plus 1 and so on, then a m. So, if I can find some segments starting with a i and b j, such that if I start from here and I read of k letters, then start from b j and I read of k letters. So, I read a i a i plus 1 up to k minus 1, i plus k minus 1 and b j, b j plus 1 and b j plus k minus 1.

Then these are both are k line segment which are equal, then I will say that u and v have a common subword of length k. So, there must be a matching sub segment of the word and now, the aim is to find the length of the longest such segment. So, the longest common subword of u and w.

(Refer Slide Time: 02:25)



Brute force

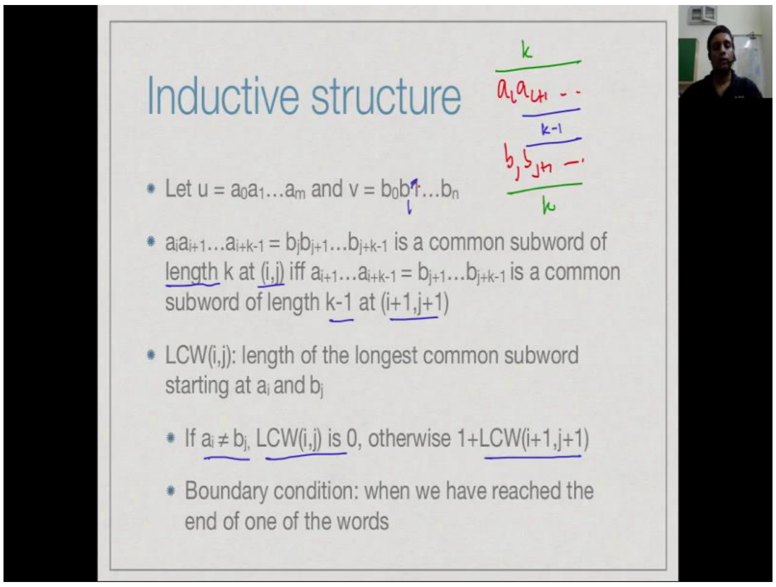
- Let u = $a_0a_1 \ldots a_m$ and v = $b_0b1 \ldots b_n$
- Try every pair of starting positions i in u, j in v
  - Match (a_i, b_j), (a_{i+1}, b_{j+1}),... as far as possible
  - Keep track of the length of the longest match
- Assuming m > n, this is $O(mn^2)$
  - mn pairs of positions
  - From each starting point, scan can be O(n)

$a_i \rightarrow$
$b_j \rightarrow$
a_i b_j

So, here is a Brute force algorithm, you just try out every position, you look at every a i and b j, so you look at every i and j, i between 0 and m and j between 0 and n. And then I just keep looking, I look at the, if these two match, I look at the next letter and go on and I keep going until I find two positions which do not match. Then I know that the longest subword starting from these two positions is only mismatch, so I match a b i a plus 1 b i plus 1 as for as possible may keep track of the longest such match, I do this for every a i and b j.

So, if m is greater than n, then first of all the number of choices of a i and b j is m times n and in general, I will go to the length of the end of the shorter word matching everything. So, I could do order n word before I reach the end of this scan, everything matches until the end of the shorter word, then I put do order n word. So, therefore, this will be O of m times n into n or O m n square. So, now, our goal is to see, whether we can make this more efficient by doing something inductive.

(Refer Slide Time: 03:41)



Inductive structure

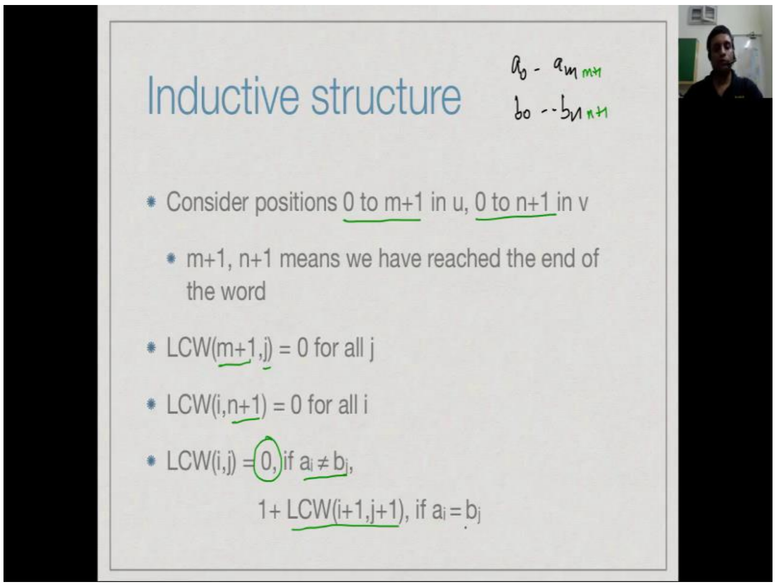
- Let $u = a_0a_1 \ldots a_m$ and $v = b_0b_1 \ldots b_n$
- $a_ia_{i+1} \ldots a_{i+k-1} = b_jb_{j+1} \ldots b_{j+k-1}$ is a common subword of length k at (i,j) iff $a_{i+1} \ldots a_{i+k-1} = b_{j+1} \ldots b_{j+k-1}$ is a common subword of length k-1 at (i+1,j+1)
- LCW(i,j): length of the longest common subword starting at a_i and b_j
  - If $a_i \neq b_j$, LCW(i,j) is 0, otherwise 1+LCW(i+1,j+1)
  - Boundary condition: when we have reached the end of one of the words

So, the first observation in the inductive observation is that if I have a i a i plus 1, if I have b j b j plus 1. So, if I have a subword staring from here of length k, then look at the next position and go forward, it is must be a subword of length k minus 1. So, there is a common subword of length k at i j, if and only if there is a long common subword of length k minus 1 and i plus 1 and j plus 1.

So, in other words I look at i j, if i j is equal, then there is possible to start a word at common subword at a i and b j. So, if a i is equal to b j, I look at the longest common subword I could have bought i plus 1 and j plus 1 and add 1 to it. On the other hand, if a i not equal to b j, then I cannot have a common subword adding to it, because the very first let it does not match. So, then I just the length of the longest common subword starting at that position 0. So, this gives us a handle on the inductive structure.

So, I look at two positions, if there not equal, I declare that there is no common subwords starting there, if they are equal, then I postpond the problem to the next position and add 1 to whatever I get in next position. In the boundary condition is when one of the words is empty, then we have no let us left, so we cannot extend one of the words any more, we cannot go forward, then we can say that there cannot be a common subword, because there are no letters to use from one of the common subwords.

(Refer Slide Time: 05:14)



Inductive structure

$a_0 - a_m \; m+1$

$b_0 \,\text{--}\, b_n \; n+1$

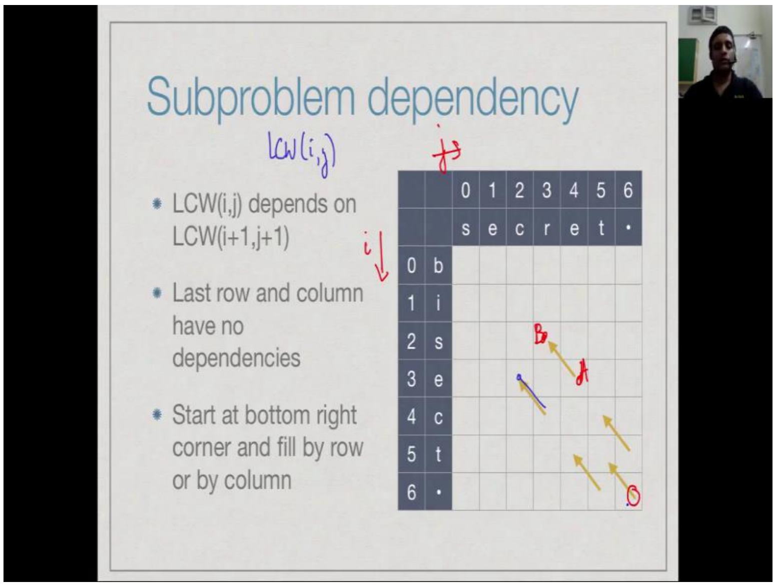
- Consider positions $\underline{0 \text{ to } m+1}$ in u, $\underline{0 \text{ to } n+1}$ in v
  - m+1, n+1 means we have reached the end of the word
- $LCW(\underline{m+1},\underline{j}) = 0$ for all j
- $LCW(i,\underline{n+1}) = 0$ for all i
- $LCW(i,j) = \boxed{0}$, if $\underline{a_i \neq b_j}$,

  $1 + \underline{LCW(i+1,j+1)}$, if $a_i = b_j$

So, for convenience, all though our words go from a 0 to a m and from b 0 to b m, we will allow a m plus oneth position and an n plus oneth position. So, we will have positions 0 to m plus 1 in u and 0 to n plus 1 in v, so m plus 1 means I have gone beyond the last position in u and exhausted all the letters. Likewise n plus 1 in b means we have reach the end of the word in v.

So, what we said is that if we are reach the end of the word in u, then no matter where we are in v, there is no commons of word, so length of the common subword is 0. Likewise, if we have reach the end of the word in v, wherever we are in u, there is no common subword length is 0. So, if we are not in these two cases, we are not in the end of the word, then we check whether a i equal to b j, if a i is not equal to b j, we declare as we saw earlier with longest common subword has length 0. Otherwise, we inductively compute the value of in next position in the both of words and add 1 to it, because we can extended that word by 1, because a i equal to b j.

(Refer Slide Time: 06:24)



Subproblem dependency

lcw(i,j)

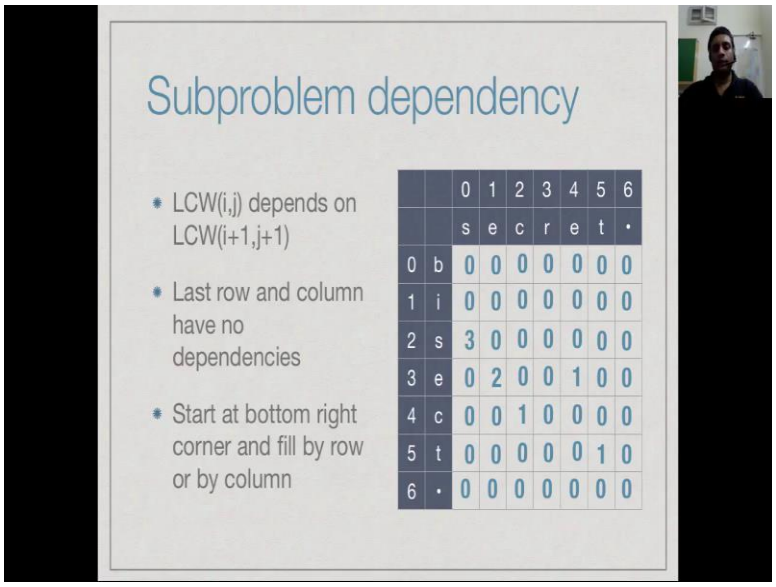
- LCW(i,j) depends on LCW(i+1,j+1)
- Last row and column have no dependencies
- Start at bottom right corner and fill by row or by column

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

So, we could write a memorized recursive function for that, but we will directly try to compute a dynamic programming solution for this problem. So, we will typically have these sub problems of the form LCW, i j; these are the basic sub problem. So, what we are saw from the earlier thing is that if I have a i equal to b j, I need to look at a i plus 1, b j plus 1, so LC, IW, LCW i j depends on LCW i plus 1 and j plus 1. We are this kind of the subproblem dependency, at every point a value needs to look to the one which is below.

So, we have in this to be clear this is our i and this is our j, so it has to look down and write. So, therefore with our convention that we draw the arrays backward that is, if this square depends on this square, then we draw an arrow, indicating the dependency in reverse. So, I need to compute the bottom of the arrow, before I get to that down arrow, that is what the arrow, I need to compute this, before I can compute this. So, now, we can start at this corner, because this depends on nothing and then start working.

(Refer Slide Time: 07:43)



Subproblem dependency

- LCW(i,j) depends on LCW(i+1,j+1)
- Last row and column have no dependencies
- Start at bottom right corner and fill by row or by column

		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

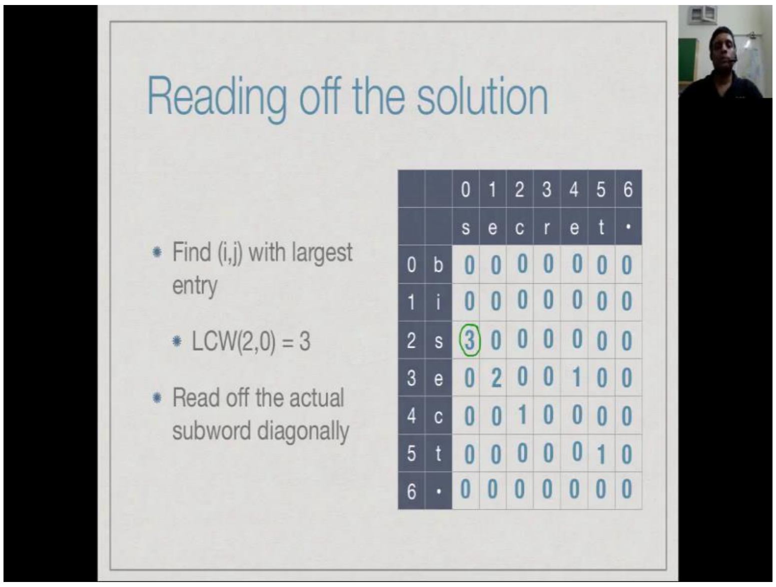
So, as we saw at the boundary, where I have exhausted one word, where either the first word or the second word is. So, we have return the word by bisect and secret this 2, tell us what the values of a i and b j. So, these are the values of a I and these are the values of b j, so these are my two words. So, initially at the boundary every longest common subword is 0, because one of the two words is empty and now, I can do it say row by row or column by column. So, let us do it by column for a change.

So, now, at this point we had that a 5 is equal to b 5, therefore we get 1 plus LCW of 6, 6 and therefore we get a 1, everywhere else c and t, e and t, then are at the t. So, everywhere else, it is just 0, because the value do not match. So, if I go to the previous or column now. Here, the only one that has anything to say is e and e, but there because the next one is 0. So, I get 1 plus 0 is 1.

If I go to the previous one, in fact the value no, there is no r in bisect, so therefore, all the values the subword is 0, if I come to the next thing, again now I have one place where c and c match. So, I say that the longest common subword starting here is 1 plus whatever happens that t and r, but t and r do not match with 1 plus 0 and so on, but now because this c is now proceeded by an e.

So, at this e I find that it is 1 plus whatever happens to the bottom, so it 1 plus 1 is 2 and likewise at s, I find that s and s match and it is 1 plus whatever is bottom right is becomes 3. So, we identify this way, we put an entry 3. So, in this particular table, it is a bit mysterious, where the answer is.

(Refer Slide Time: 09:33)



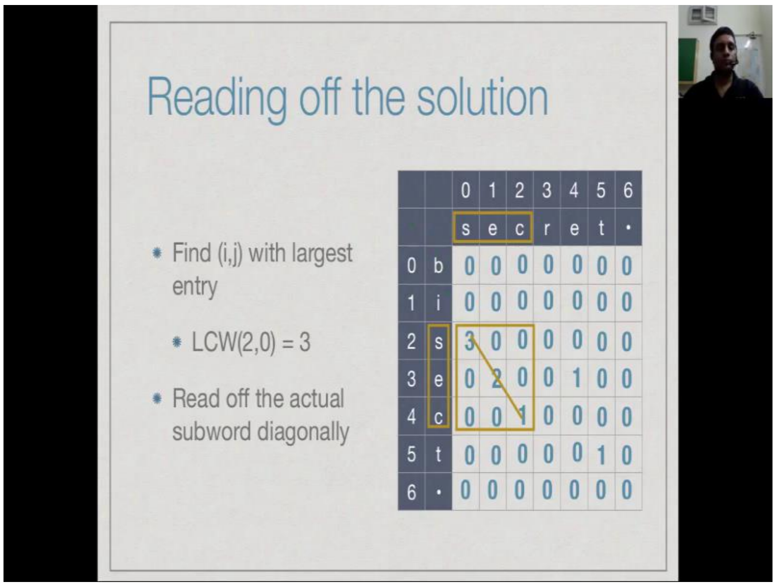
Reading off the solution

- Find (i,j) with largest entry
  - LCW(2,0) = 3
- Read off the actual subword diagonally

		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, what we have to do is find the entry with a largest value, in this case 2 comma 0, this is our largest value. So, this tells us that between these two words, the longest common subword has length 3. As I said before, we are only computing the longest common subword, so how do we actually find the subword. Well, we can actually in this case, very easily read it off, we know that there is a longest common subword here of length 3; that means, that it must be succeeded by the word length 2 and so on.

(Refer Slide Time: 10:06)



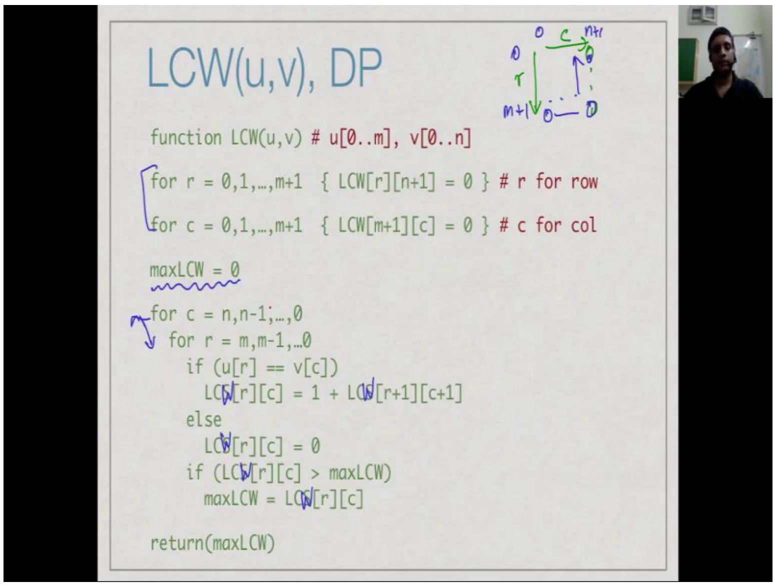
Reading off the solution

- Find (i,j) with largest entry
  - LCW(2,0) = 3
- Read off the actual subword diagonally

		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, this particular diagonal tells us what the letters in the word are and if I just project this diagonal on to both the one column, I get the word in c, c. So, in this example as we see, once we computed the length, it is very easy to read of this solution.

(Refer Slide Time: 10:22)



LCW(u,v), DP

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

for r = 0,1,…,m+1  { LCW[r][n+1] = 0 } # r for row

for c = 0,1,…,m+1  { LCW[m+1][c] = 0 } # c for col

maxLCW = 0

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

return(maxLCW)
```

So, here is some code for this particular algorithm, the dynamic programming version, just to make it less confusing, I have explicitly used the variables r and c, instead of i and j. So, r is the row, it was this way and c goes this way. So, what we say is that initially if

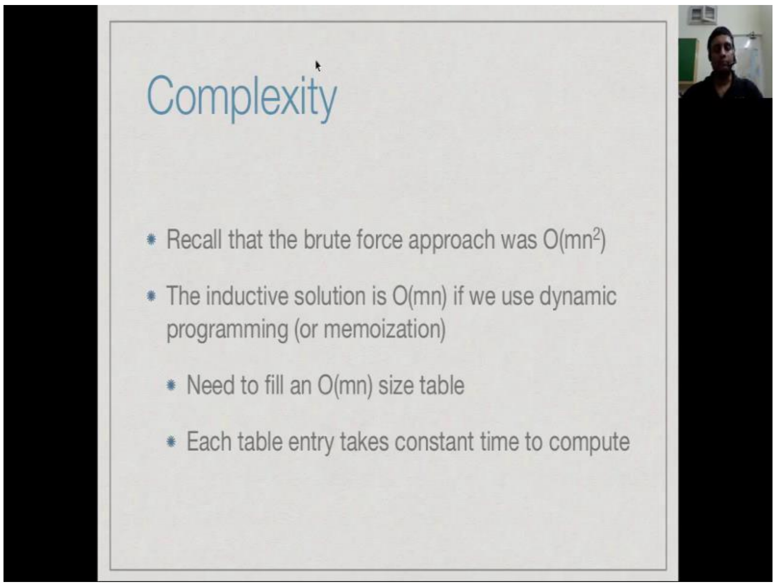
I am looking at the last column, these are all 0's and you looking at the bottom row, remember that the numbering is from 0 to n plus 1 and from 0 to m plus 1.

So, we looking the bottom row, when the row number is n plus 1, every column is 0. So, this puts the 0's. So, these two lines just populate things, now we have to find, remember the goal is to find the maximum position in the entire array in order to give the answer finally, so we just keep track of that with the value here. So, we initial assume that the maximum common subword of length 0.

Now, what we do is, we just go column by column and row by row, so we go to each column from n to 0, so we do column by column and each column, we go row by row from bottom to top, m to 0. We look at the word position r and position c is u, r equal to b, c. If so I add 1, should be LCW, so if I have u a bar is equal to d of c, then the longest common word, length will longest common subword r c 1 plus add r plus 1 c plus 1.

Otherwise, if 0, because there is no common word and if the new value of computed is bigger than the value as seen so far, then I will updated to the current value and finally, the end this returns length. So, this is a straight forward iterate of way to process this thing column by column, you could invert these two induces and it row by row, so that is an exercise.

(Refer Slide Time: 12:27)



Complexity

- Recall that the brute force approach was $O(mn^2)$
- The inductive solution is $O(mn)$ if we use dynamic programming (or memoization)
    - Need to fill an $O(mn)$ size table
    - Each table entry takes constant time to compute