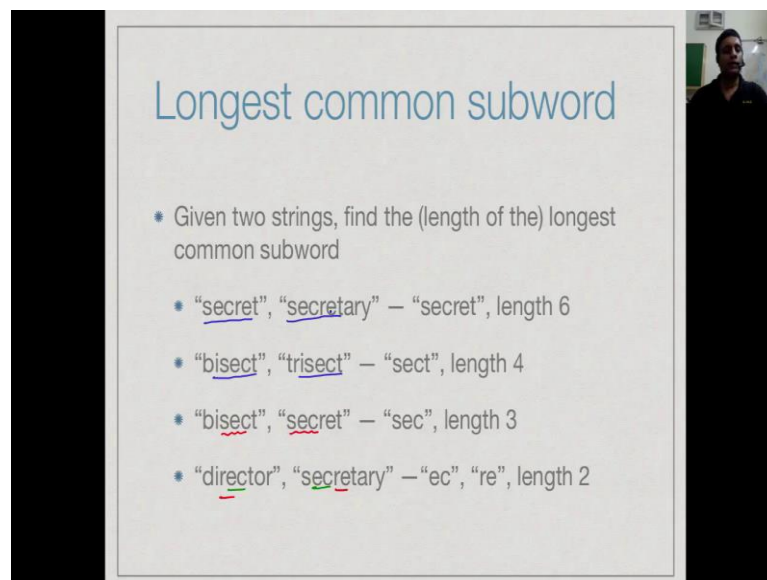


Design and Analysis of Algorithms
Prof. Madhavan Mukund
Chennai Mathematical Institute

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 quite 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 ...

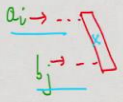
$a_0 a_1 \dots a_{i+k-1} \dots a_m$
 $b_0 b_1 \dots b_j \dots b_{j+k-1} \dots b_n$

- Let $u = a_0 a_1 \dots a_m$ and $v = b_0 b_1 \dots b_n$ be two strings
- If we can find i, j such that $a_i a_{i+1} \dots a_{i+k-1} = b_j b_{j+1} \dots b_{j+k-1}$, 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+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+1} up to $i+k-1$, $i+k-1$ and b_j, b_{j+1} and b_{j+k-1} .

Then these are both 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 $u = a_0a_1\dots a_m$ and $v = b_0b_1\dots b_n$
- Try every pair of starting positions i in u , j in v
 - Match $(a_i, b_j), (a_{i+1}, b_{j+1}), \dots$ 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 a_i, b_j
- From each starting point, scan can be $O(n)$

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_i with b_j plus 1 b_j plus 1 as far 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 $u = a_0 a_1 \dots a_m$ and $v = b_0 b_1 \dots b_n$
- $a_i a_{i+1} \dots a_{i+k-1} = b_j b_{j+1} \dots b_{j+k-1}$ is a common subword of length k at (i, j) iff $a_{i+1} \dots a_{i+k-1} = b_{j+1} \dots b_{j+k-1}$ 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 starting from here of length k , then look at the next position and go forward, it 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 postpone 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

$a_0 - a_m$
 $b_0 - b_n$

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

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 $LCW, i+1, j+1$. 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

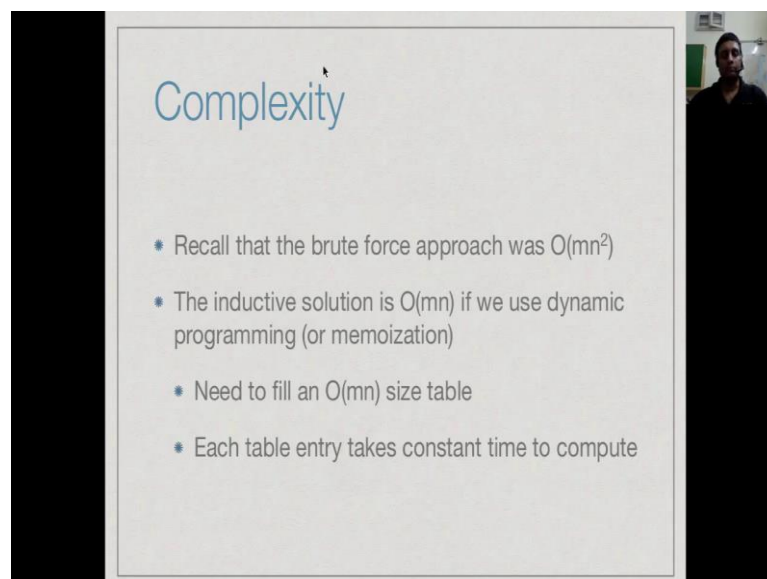
I am looking at the last column, these are all 0's and you looking at the bottom row, remember that the numbering is from 0 to n plus 1 and from 0 to m plus 1.

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

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 requires as to fill in the table of size m time n , so obviously, every entry 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^3 . If we use dynamic programming, we have done it, but if we use memoization also, you will get the same answer, although remember, there is a recursive call might cost you in terms of actual implementation time.

(Refer Slide Time: 13:04)

Longest common subsequence

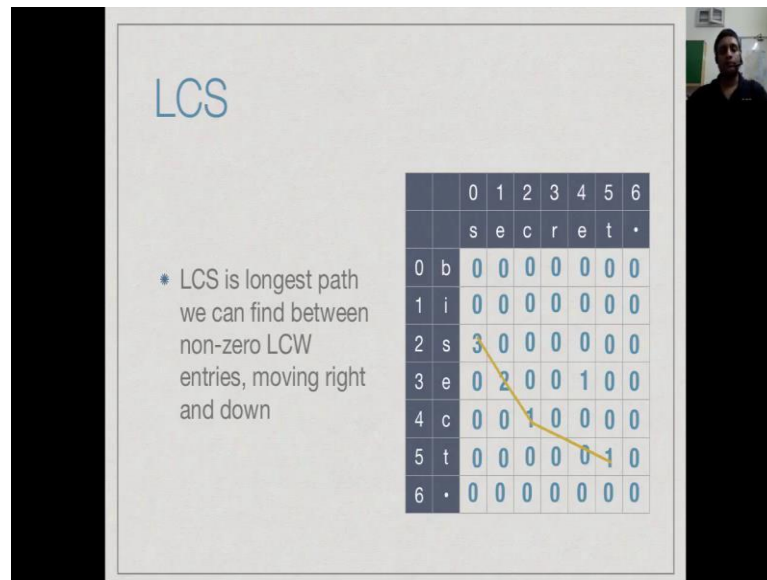
- Subsequence: can drop some letters in between
- Given two strings, find the (length of the) longest common subsequence
- "secret", "secretary" — "secret", length 6
- "bisect", "trisection" — "isect", length 5
- "bisect", "secret" — "sect", length 4
- "director", "secretary" — "ectr", "retr", length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

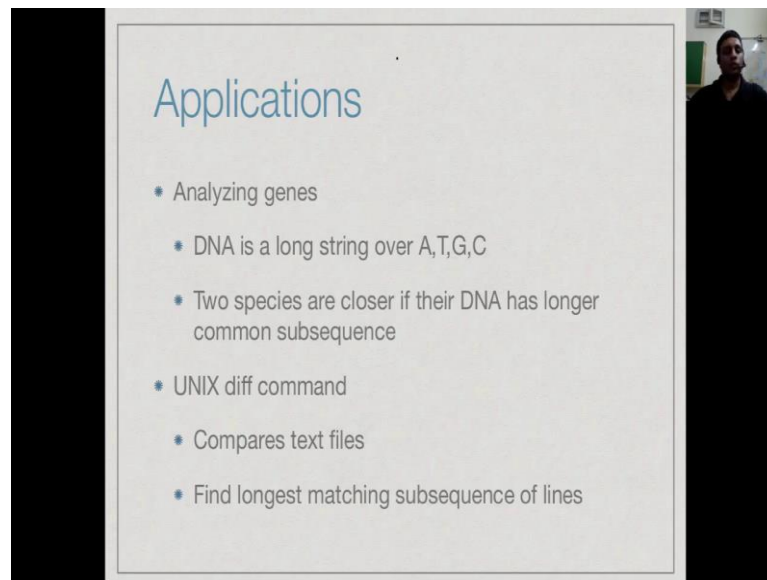
But, now by doing this topic I can get ectr, like get ectr, similarly I can get retr, I can get retr. So, if we allow a self's to drop letter as we can get longer sequences that match and this is called longest common subsequence.

(Refer Slide Time: 14:32)



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

U	a ₁	a ₁	a ₂	a _{m-1}	a _m
V	b ₀	b ₁	b ₂	b _{n-1}	b _n

- If $a_0 = b_0$,

$$\text{LCS}(a_0a_1\dots a_m, b_0b_1\dots b_n) = 1 + \text{LCS}(a_1a_2\dots a_m, b_1b_2\dots b_n)$$
- 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 $\text{LCS}(a_1a_2\dots a_m, b_0b_1\dots b_n)$ and $\text{LCS}(a_0a_1\dots a_m, b_1b_2\dots b_n)$ 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 $a_0 = b_0$,

$$\text{LCS}(a_0a_1\dots a_m, b_0b_1\dots b_n) = 1 + \text{LCS}(a_1a_2\dots a_m, b_1b_2\dots b_n)$$
- 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 $\text{LCS}(a_1a_2\dots a_m, b_0b_1\dots b_n)$ and $\text{LCS}(a_0a_1\dots a_m, b_1b_2\dots b_n)$ 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

$$\begin{array}{c|cccccccc}
 U & a_0 & a_1 & a_2 & \dots & \dots & a_{m-1} & a_m \\
 V & b_0 & b_1 & b_2 & \dots & \dots & b_{n-1} & b_n
 \end{array}$$

- If $a_0 = b_0$,

$$LCS(a_0a_1\dots a_m, b_0b_1\dots b_n) = 1 + LCS(a_1a_2\dots a_m, b_1b_2\dots b_n)$$
- 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 $LCS(a_1a_2\dots a_m, b_0b_1\dots b_n)$ and $LCS(a_0a_1\dots a_m, b_1b_2\dots b_n)$ and take the maximum

Now, if it is not equal, then one is not sure what you do, it is not a sound idea to drop both. So, for instance 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.

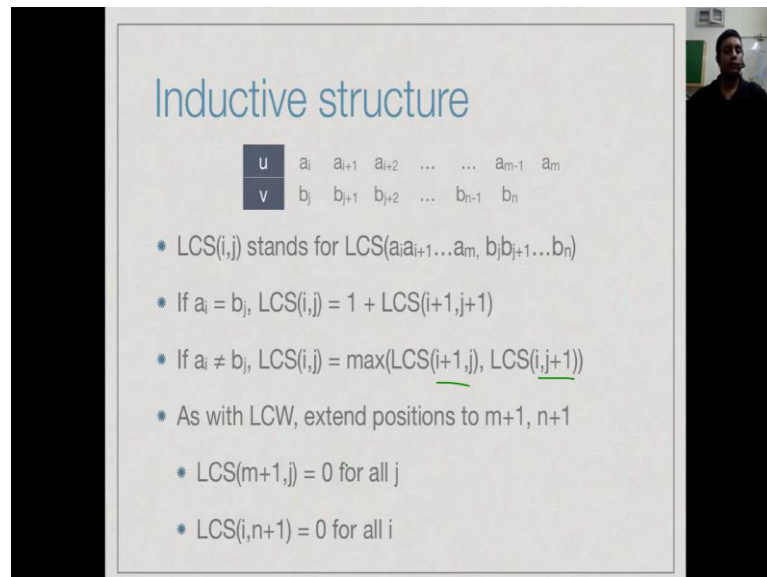
If they are both different, if they are not equal, then this must match something there and this must match something here and both cannot happen. because, they will cross, this we said not allowed, so only one of them can match something to the right on the other word, but we do not know which one. 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 I 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 $LCS(a_i a_{i+1} \dots a_m, b_j b_{j+1} \dots b_n)$
- * If $a_i = b_j$, $LCS(i,j) = 1 + LCS(i+1, j+1)$
- * If $a_i \neq b_j$, $LCS(i,j) = \max(LCS(i+1, j), LCS(i, j+1))$
- * 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	0	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 $i+1, j+1$. But, now we are also dependency $i+1, j$ and $i, j+1$. 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, $LCS(m, n, 5, 5)$ 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 $1 + c_{s[i+1], j+1}$. 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"

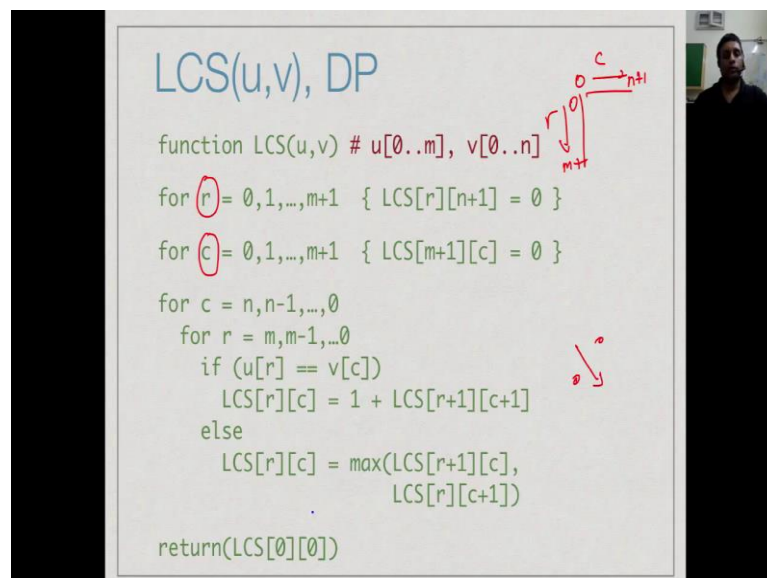
		0	1	2	3	4	5	6
		s	e	c	r	e	t	.
0	b	4	3	2	1	1	0	0
1	i	3	2	1	1	0	0	0
2	s	3	2	1	1	0	0	0
3	e	3	2	1	1	0	0	0
4	c	2	2	2	1	1	0	0
5	t	1	1	1	1	1	0	0
6	.	0	0	0	0	0	0	0

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.

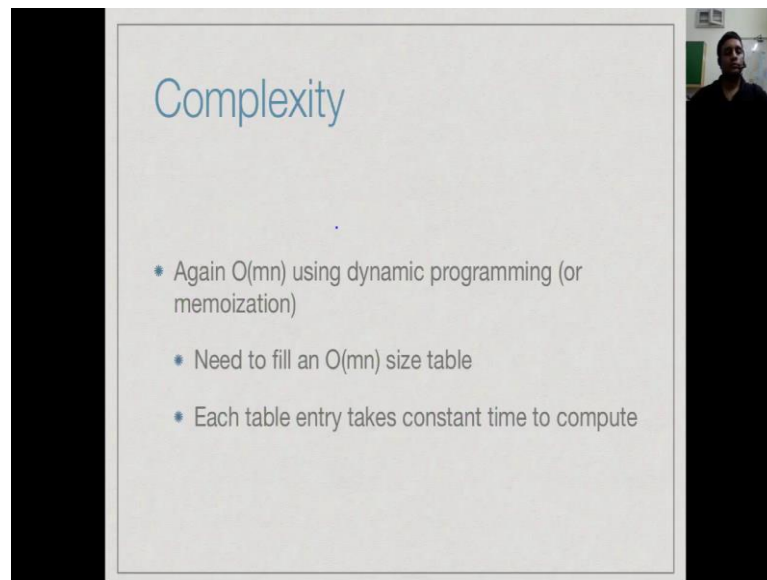
(Refer Slide Time: 26:07)



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.