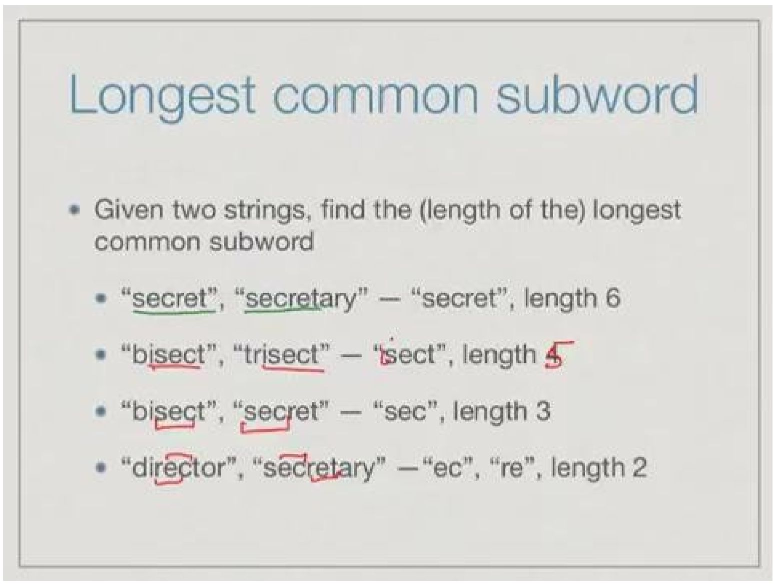
Programming, Data Structures and Algorithms in Python

Prof. Madhavan Mukund
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 08
Lecture - 03
Longest Common Subsequence

We are in the realm of Inductive Definitions, Recursive Functions and Efficient Evaluation of these using memorization and dynamic programming.

(Refer Slide Time: 00:04)



Longest common subword

- Given two strings, find the (length of the) longest common subword
  - "secret", "secretary" — "secret", length 6
  - "bisect", "trisect" — "isect", length 5
  - "bisect", "secret" — "sec", length 3
  - "director", "secretary" —"ec", "re", length 2

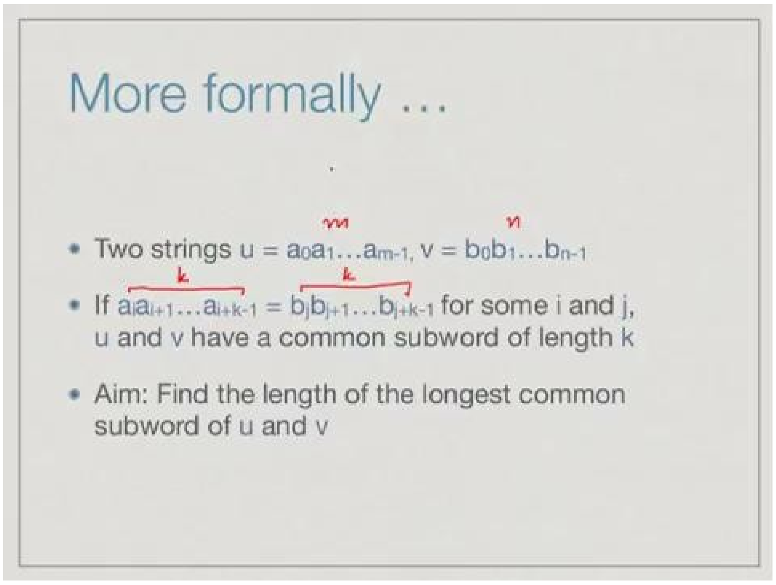
So, we are looking examples of problems where the main target is to identify the inductive structure and once you identify the inductive structure then the recursive structure of the program becomes apparent from which you can extract the dependencies and figure out what kind of memo-table you have and how you will can iteratively using dynamic programming.

This is something which comes to the practice and by looking at more examples hopefully the procedure become clearer, but the key thing to dynamic programming is to be able to understand the inductive structure. So, you need to take a problem, identify how the main problem depends on its sub parts and using this come up with the nice inductive definition which you can translate in to a recursive program. Once you have the recursive program then the memo-table and the dynamic programming almost comes out automatically from that.

This is the problem involve in words. So, what you want to do is take a pair of words and find the longest common subword. For instance, here we have secret and secretary and secret is already also inside secretary and clearly secret is the longest word in secret itself. The longest subword that is common is the word secret and it has length 6. Let we move to the next think bisect and trisect then actually this should be isect that say which has length 5, similarly if we have bisect and secret then sec.

When we say subword, of course we do not mean a word in the sense; we just mean a sequence of letters. So, s e c is the longest common subword in has length 3 and if you have two very different words like director and secretary, sometimes you might have only small things, for example, here r e and e c are, for examples of subword but there are really very long words which are common to the subword is only length 2.

(Refer Slide Time: 02:11)

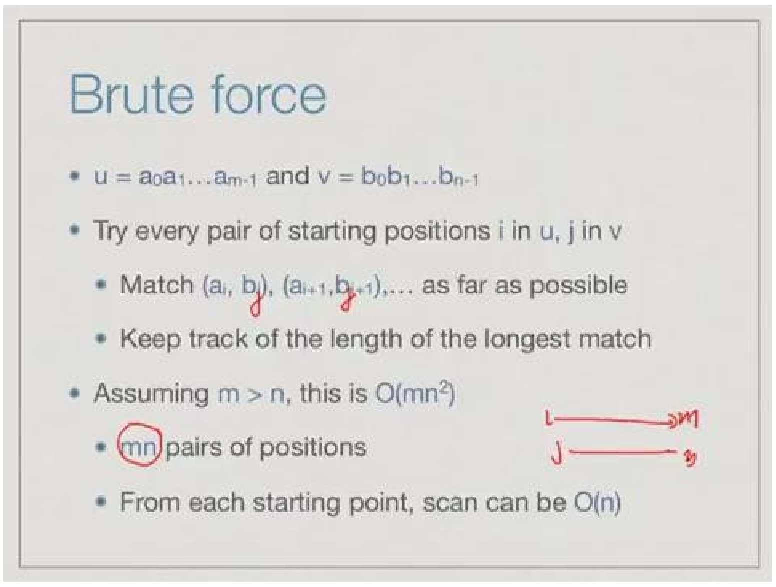


More formally ...

- Two strings $u = \overset{m}{a_0a_1 \ldots a_{m-1}}$, $v = \overset{n}{b_0b_1 \ldots b_{n-1}}$
- If $\overbrace{a_ia_{i+1} \ldots a_{i+k-1}}^{k} = \overbrace{b_jb_{j+1} \ldots b_{j+k-1}}^{k}$ for some i and j, u and v have a common subword of length k
- Aim: Find the length of the longest common subword of u and v

Here is the more formal description right. So, supposing I have two words u and v. So, u is of length m and v is of length n and the number positions using python notation and number 0 to n minus 1, 0 to n minus 1 then what I want to do is able to start at a i and go k steps. So, i to i plus k minus 1 and b j to j plus k minus 1 such that, these two segments are identical, this is a common subword and we want to find the longest such common subword, what is the k, we do not even want to subword, will find that subword will be a byproduct. You first need to just find k, what is the length of the longest common subword of u n?

(Refer Slide Time: 02:56)



Brute force

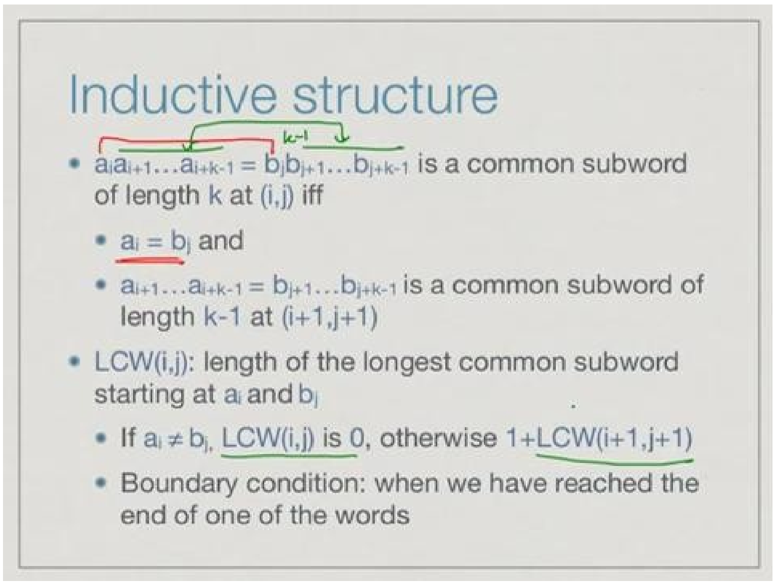
- $u = a_0a_1\ldots a_{m-1}$ and $v = b_0b_1\ldots b_{n-1}$
- Try every pair of starting positions i in u, j in v
    - Match (a_i, b_i), (a_{i+1}, b_{i+1}),... as far as possible
    - Keep track of the length of the longest match
- Assuming m > n, this is $O(mn^2)$
    - mn pairs of positions
    - From each starting point, scan can be O(n)

There is a brute force algorithm that you could use which is you just start at i and j in two word. In each word you can start a position i in u j in v and see how far you can go before you find they are not. So, you match a i and b j right. So, if a i and b j work then its fine. So, it should be b j and if a i and b j work then you go to a i plus 1 b j plus 1 and so on and whenever we find two letters which differ then the commons adverse starting at a j has ended and you so from i j I have a common subword of something.

Now, among all the i js you look for the longest one and that becomes your answer. Now, this unfortunately is effectively now an n cube algorithm. We think of m and n can be equal technically m n squared because there are m times n different choices of i and j and in general I started i j and then I have to go from i to the end right and from j to the end.

So, we have to do a scan for each i j in this scan in general adds up to an order, order n factor and so we have order m n squared or order n cube if you like.

(Refer Slide Time: 04:09)



Inductive structure

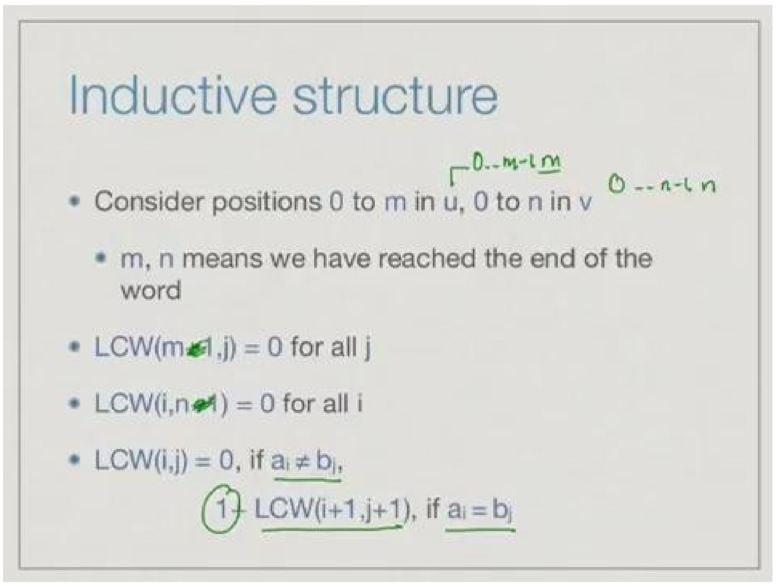
- $a_i a_{i+1} \ldots a_{i+k-1} = b_j b_{j+1} \ldots b_{j+k-1}$ is a common subword of length k at (i,j) iff
  - $a_i = b_j$ and
  - $a_{i+1} \ldots a_{i+k-1} = b_{j+1} \ldots b_{j+k-1}$ is a common subword of length k-1 at (i+1,j+1)
- LCW(i,j): length of the longest common subword starting at a_i and 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

Our goal is to find some inductive structure which makes this thing computationally more efficient. So, what is the inductive structure? Well we have already kind of seen it when can we say that there is a commons subword starting at i j of length k, the first thing is that we need this a i to be the same as b j. So, I need this condition and now if this is a commons subword of length k at i j then what remains of subword namely this segment from i plus 1 to this and j plus 1 to this must also match and they must be in turn be a k minus 1 length subword from here to there. So, we want to say that there is a k length subword starting at i j if a i is equal to b j and from i plus 1 and j plus 1 there is a k minus 1 length subword.

In other words, I can now write the following definition, I can say that the longest common the length of the longest common subword l c w starting from i j. Well, if they two or not the same if a i is not the same the same there is no common subword at all because if I start from i immediately have two different letters. So, when the length is 0 otherwise I can inductively find out what is the longest common subword to may right start i plus 1 start from j plus 1.

Find out what I can do from there and to word I can add one letter because this current letter a i is equal to b j. So, I get one plus that and the base case of the boundary condition is when one of the two words is empty right. If I have no letters left, if I have gone i j I am looking at difference combinations i and j. So, if either i or j has reached the end of the word then there is no possibility of a common subword at that point. So, when we reach the end of one of the words say answer must be 0.

(Refer Slide Time: 05:58)



Inductive structure

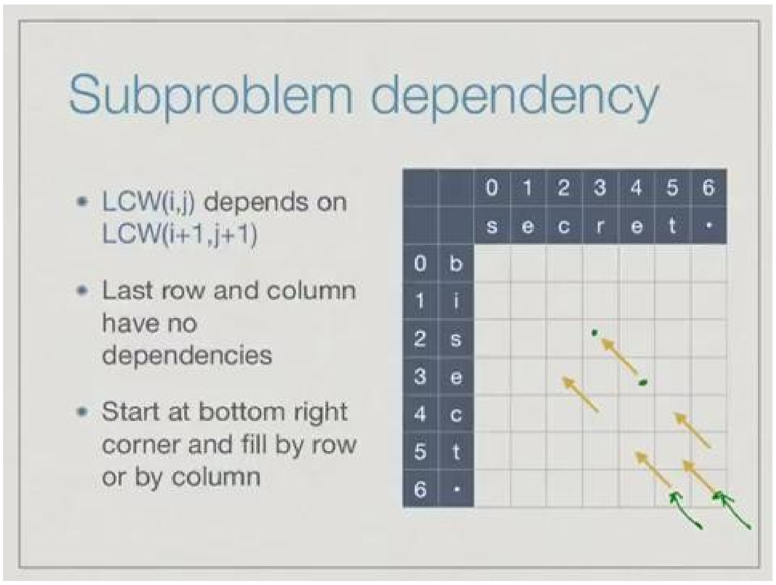
- Consider positions 0 to m in u, 0 to n in v $\overset{0..m-1\,\underline{m}}{}$ $0 -- n-1\ n$
    - m, n means we have reached the end of the word
- LCW(m$\cancel{+1}$,j) = 0 for all j
- LCW(i,n$\cancel{+1}$) = 0 for all i
- LCW(i,j) = 0, if $\underline{a_i \neq b_j}$,

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

This gives us the following definitions. So, remember that u is actually has length m. So, it has 0 to m minus 1. So, what we will do is, we will add a position of m to indicate that we have crossed the last letter. Similarly, v has 0 to n minus 1 has valid positions. So, we will use in this 0 to n. So, if i becomes m or j becomes n it means that that corresponding index has gone beyond the end of the word right. So, this should be m and this should be n. We have that if you reach m then lcw of n comma j 0 is 0 because we have gone past the length u.

Similarly, if you reach n, but lcw of i comma n is 0 because you gone past the length of v and if you are not gone past the length, if you are somewhere inside the word in a valid position then the length is going to be 0 if the two positions are not the same. If a i is not equal to b j, otherwise inductively I compute the length from i plus 1 and j plus 1 and add one to it. this is the case when a i is equal to b j because that segment i can extend by. So, this is just stating in an equation form the inductive definition that we purposely earlier.

(Refer Slide Time: 07:18)



Subproblem dependency

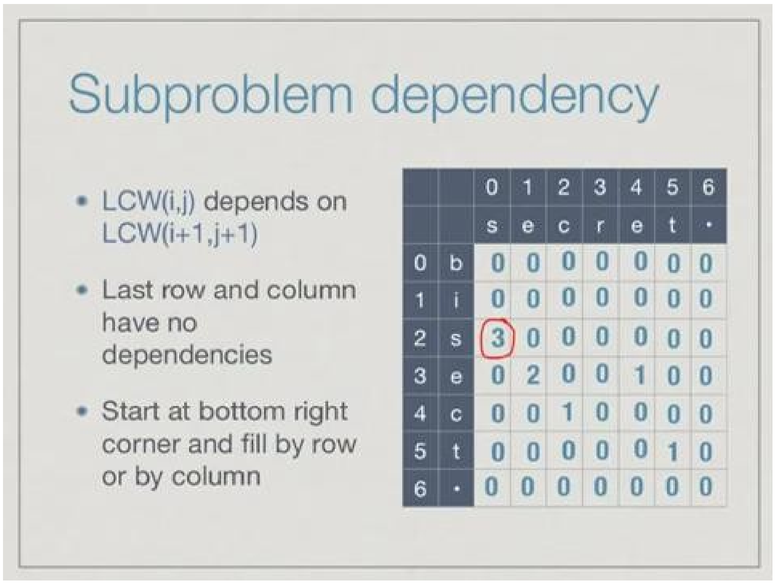
- LCW(i,j) depends on LCW(i+1,j+1)
- Last row and column have no dependencies
- Start at bottom right corner and fill by row or by column

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

So, here we saw example for bisect and secret. We have position 0 to 5 and then we have the 6 position indicating the end of the word and now remember that the way our inductive definition was phrased i j depends only on i plus 1 j plus 1. So, actually the dependencies at this ways to the arrows are indicating that and in order solve this I need to solve this first. The value at 2 comma 3 depends on the value 3 comma 4.

In order to solve this I do not need to solve anything because everything once i. So, in order to solve this I only need to so anyway. We can basically, we have this simple thing which says that the corner and the actually the right column and the bottom thing do not require anything and we know that because those are all 0s.

(Refer Slide Time: 08:08)



Subproblem dependency

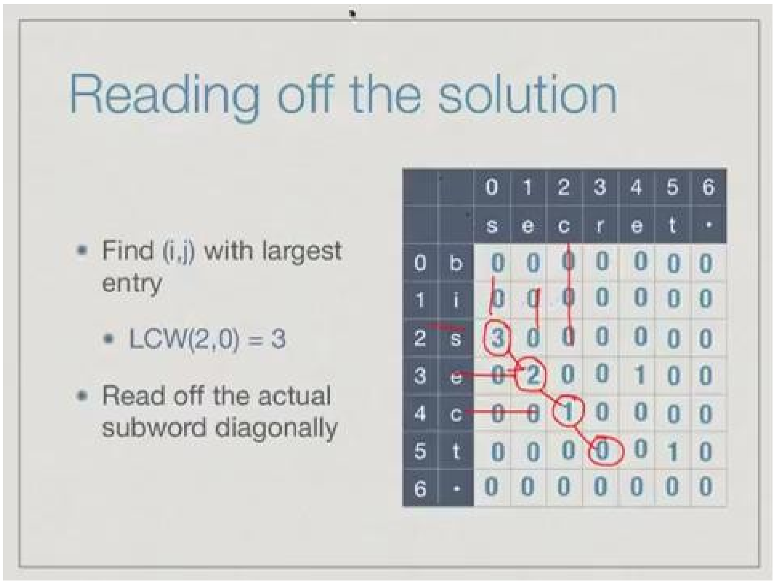
- LCW(i,j) depends on LCW(i+1,j+1)
- Last row and column have no dependencies
- Start at bottom right corner and fill by row or by column

		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

We can actually fill in those values as 0 because that is given to us by definition and now we can start, for instance, we can start with this value because its value is known. We will look at whether this t matches that t it is that. So, we take one plus the value two is bottom. So, we get 1 and then we can walk up and do the same thing at every point we will say that if c is not the same as t. So, none of these letters if you look at these letters here right none of these letters are t. So, for all of these letters I will get 0 directly because it says that a i is not equal to b j.

I do not even have to look at i plus 1 j plus 1, I directly says that 0 because is not there. So, in this way I can fill up this column. This is like our grid pack thing I can fill by column by column even though there the dependency was to the left and bottom and here the dependence is diagonally bottom right. I can fill up column by column and I can keep going and if I keep going I find an entry 3. So, the entry 3 is the largest entry that I see and that is actual answer this entry 3.

(Refer Slide Time: 09:13)



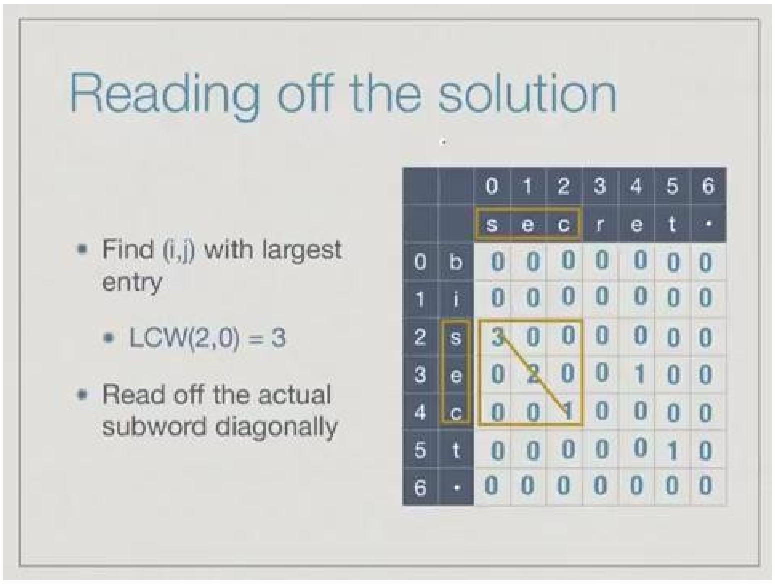
Reading off the solution

- 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

And now we said earlier that we are focusing on the length of the longest common subword not the word itself in the reason. We need to; we can afford to do that is because we can actually read of the answer once we have got the lengths. So, we ask ourselves why we did we get a 3 here. We got 3 here because we came as 1 plus 2. Since we came as 1 plus 2 it must mean that these two letters are the same. So, we got 2 here is because it is 1 plus 1. So, these two letters must also be the same. Finally, we got one here because this is 1 plus 0. So, these two letters are the same. Therefore, these three letters must be the same.

(Refer Slide Time: 09:54)



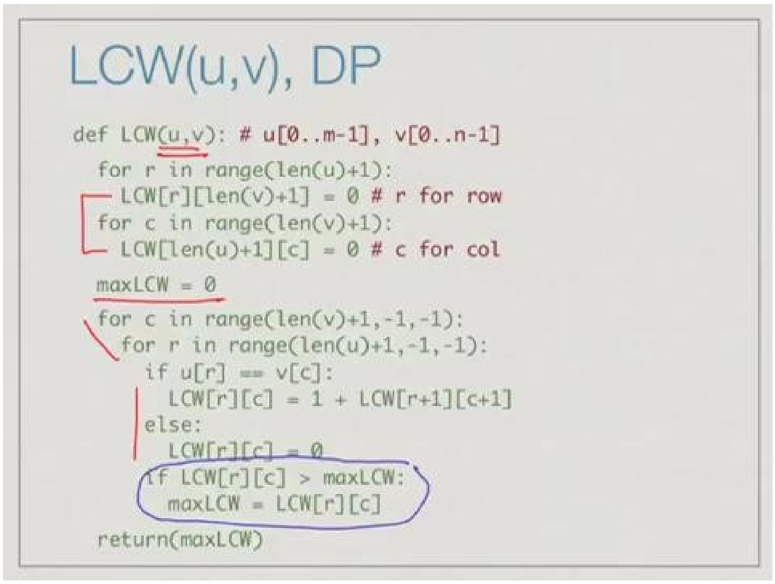
Reading off the solution

- 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

If you walk down from that magical value, the largest value and we follow the sequence then we can read of and the corresponding row or column because they are the same, your actual subword which is the longest common subword for these two.

(Refer Slide Time: 10:10)



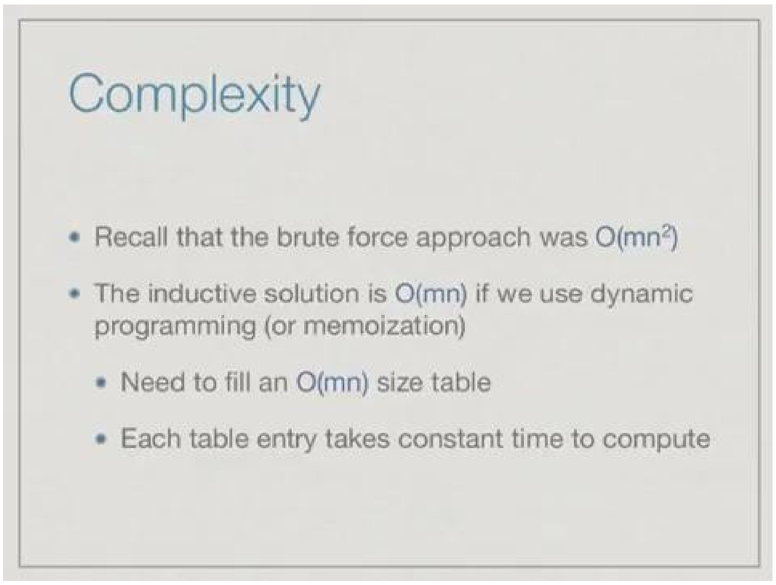
LCW(u,v), DP

```
def LCW(u,v): # u[0..m-1], v[0..n-1]
  for r in range(len(u)+1):
    LCW[r][len(v)+1] = 0 # r for row
  for c in range(len(v)+1):
    LCW[len(u)+1][c] = 0 # c for col
  maxLCW = 0
  for c in range(len(v)+1,-1,-1):
    for r in range(len(u)+1,-1,-1):
      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)
```

Here is a very simple implementation in python. So, all it says is that you start with the two words u and v, you initialize this lcw thing at the boundary at the nth row on the nth column and then, now you remember the maximum value. So, you keep that by initializing the maximum value to 0 and then you fill up in this particular case the column order.

For each column, then for each row and that column you fill up the thing using the equation, if it is equal i to 1 plus otherwise as it say 0 and if i see and a new value this is the thing where I update if i see and new entry which is bigger than the entry which is currently the maximum, I update the maximum. So, this is allows me to quickly find out what is the maximum length overall and finally, when I go through this look i would filled up the entire table and i will return the maximum value i saw over.

(Refer Slide Time: 11:08)

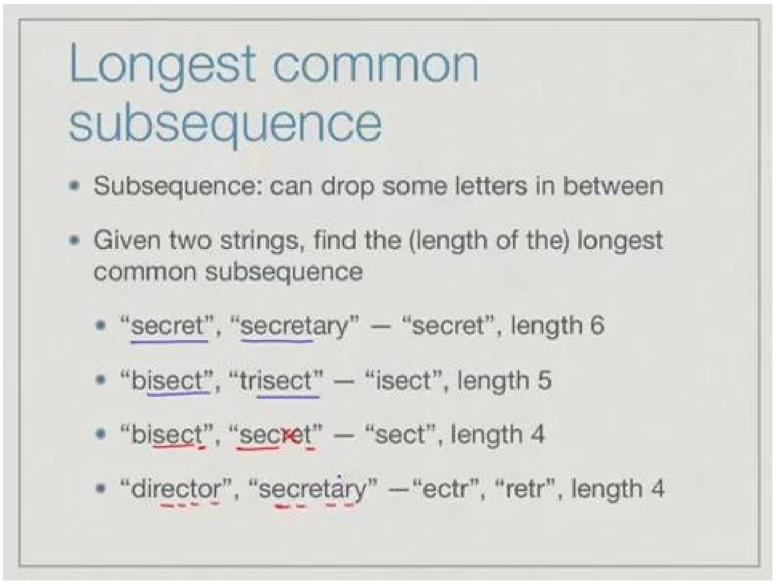


Complexity

- 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, when we did it by brute force we had an order m n square algorithm. Here we are filling up table which is of size order m by n and each entry only require us to check the ith position in the word the jth position in the world and depending on that, if necessary look up one entry i plus 1 j plus 1. It is a constant time update. We need to fill up one table of size order m n each update takes constant time. So, this algorithm brings us from m n squared in the brute force case to m n using dynamic programming.

(Refer Slide Time: 11:43)



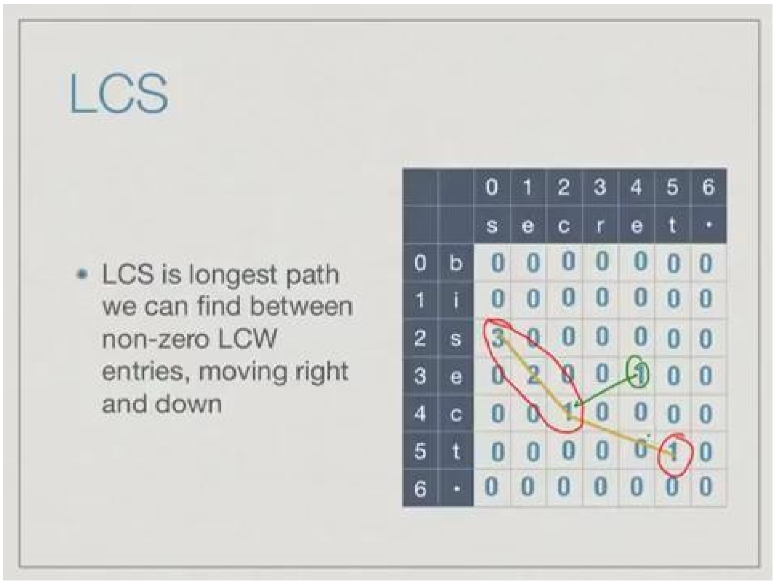
Longest common subsequence

- 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

A much more useful problem in practice than the longest common subword is what is called the longest common subsequence. So, the difference between a subword and a subsequence is that we are allowed to drop some letters in between. So, for instance, if you go back to the earlier examples of secret and secretary, there is no problem because the subword is actually the entire thing and again for bisect and trisect also it is the same thing, but if we have bisect and secret earlier if we did not allow us to skip, we could only match sec with sec, but now we can take this extra t and we can skip there 2 and match this t as say that s e c t is a subsequence in the right word, which matches the corresponding subsequence which is also a subword in the left word in the right is not a subword.

For the subsequence i have to drops some letters to get s e c t. Similarly, if I have secretary and director then I can find things like e c t r e c t r in both of them by skipping over with judiciously. So, why is this are better problem?

(Refer Slide Time: 12:54)



LCS

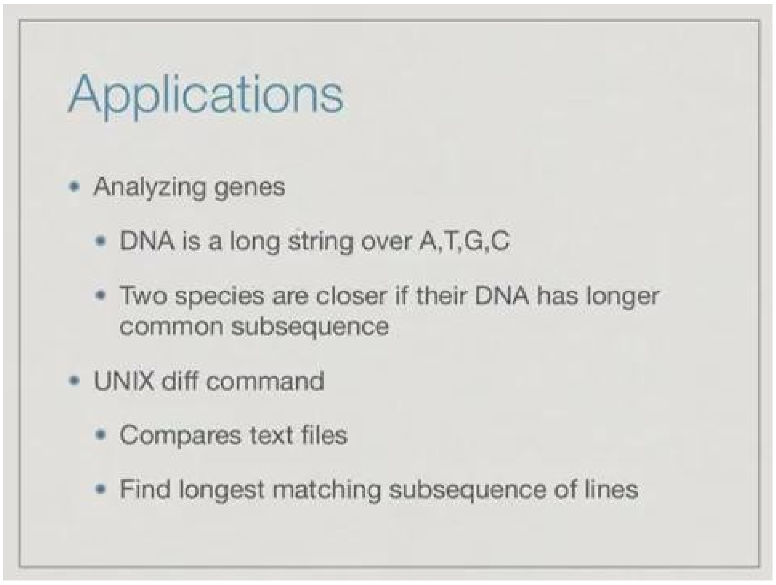
- LCS is longest path we can find between non-zero LCW entries, moving right and down

		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

Well we will see that, but effectively what skipping mean, skipping means that I get segments which are connected by gaps.. So, I get this segment then I want to continue this segments. So, I look for the next match, I skip, but the next match must come to my right. It must come to the right and below the current match because I cannot go backwards in a word and start the match again. So, I cannot, for instance go here and say that this is an extension because this requires me to go back and reuse the e that I have seen in sec to match.

The second e in secret which is not allowed, I can keep going forward which in the table corresponds to going to the right and back to the right and down. So, I am going increasing the order of index in both words and I can group together these things and this is what the longest kind of subsequences. So, we could in principle look at the longest common subword answer and look for these clever connections, but it turns out there is a much more direct way to do it in an inductive way.

(Refer Slide Time: 13:56)



Applications

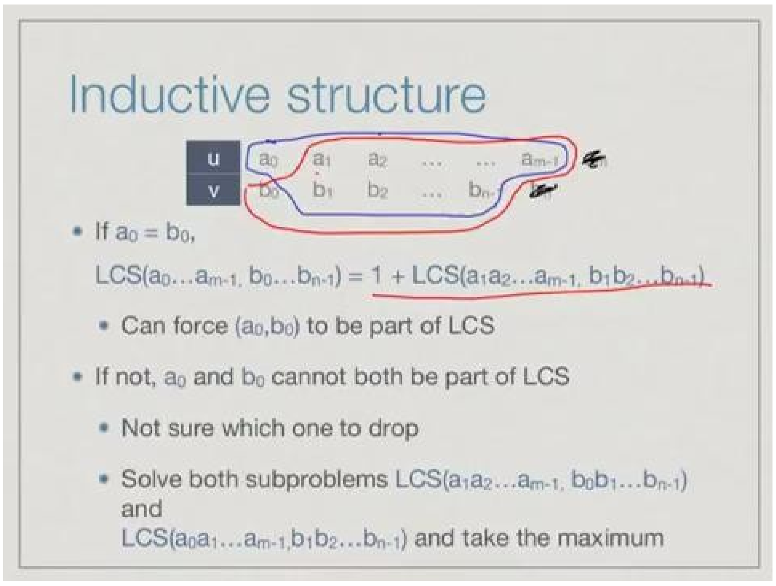
- Analyzing genes
  - DNA is a long string over A,T,G,C
  - Two species are closer if their DNA has longer common subsequence
- UNIX diff command
  - Compares text files
  - Find longest matching subsequence of lines

The motivations, well one of the big motivations for subsequence matching comes from things like genetics, for instance, when we compare the genes sequence of two organisms they are rarely equal. So, what we are looking for are large matches, where there might be some junk genes in between which you want to discard. So, you want to say that two genes sequences or two organisms are similar, if there are large overlaps over the entire genome not just looking for individuals segment along, but by just throwing away the minimal things on both sides, we can make that align as we call.

And other important example is something called a diff, which is Unix command compared to text files. So, this treats in fact, line by line two files as a word. So, each line is compared to each line in the other file if the line match they considered to be equal and this is the good way of comparing one version of the file with other version of the file. Supposing you are collaborating on a document or program with somebody else and you send it by email and they send it by.

So, they had made some changes then diff tells you quickly, what are the differences between the file you sent and the file you go back and diff essentially is doing the same thing is trying to find the longest match between the file that you sent of the file you got back and the shortest way in which you can transform one to the other by change in the few lines its. These are some typical example of this longest common subsequence problem and therefore, it is usually much more useful in practice in the longest common subword problem.

(Refer Slide Time: 15:29)



Inductive structure

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

- If $a_0 = b_0$,
  $LCS(a_0 \ldots a_{m-1}, b_0 \ldots b_{n-1}) = \underline{1 + LCS(a_1a_2 \ldots a_{m-1}, b_1b_2 \ldots b_{n-1})}$
  - 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 $LCS(a_1a_2 \ldots a_{m-1}, b_0b_1 \ldots b_{n-1})$ and $LCS(a_0a_1 \ldots a_{m-1}, b_1b_2 \ldots b_{n-1})$ and take the maximum

What is the inductive structure of the longest common subsequence problem? As before we have the words laid out. So, we can say a 0 to a m or a minus 1 a n minus 1, it does not matter how you choose it, but in the picture it says a n, but if you want to you can remove this last. So, a 0 to a n minus 1 is the first word b 0 to b n minus 1 is the second word and now there are two cases. The first case is the easy case, supposing I have these two things are equal then like before I can inductively solve the problem for a 1 and b 1 onwards and add this. I can extend that solution by saying a 0 match is b 0 and then whatever matches.

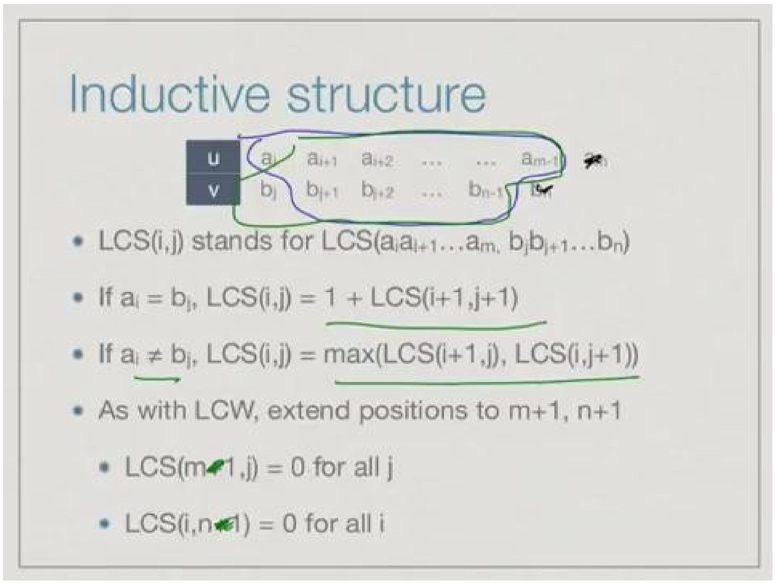
What is the subsequence, the subsequence actually is some kind of a matching, it says that you know it will say that this matches this and then this matches this, these are the same and this match is this and then this match is this and so on. Only thing is that these lines cannot over lap, they must be kept going from left to right without over lap. So, this kind of paring up equal letters, the maximum way in which again to do this is a longest common subsequence.

Now, what we are saying is that if I can actually match the first two things then I should match them and then I can go ahead and match the rest as I want, and the reason is very simple, supposing the best solution did not match these supposing you claim that the best solution actually requires meet match a 0 and b 1. Well, if I could match a 0 and b 1 I can also undo it and match a 0 and b 0 and then continue because a 0 and b 1 if they match and a 1 if match is to right. So, I can take that solution and change it to a solution where a 0 matches b 0. The first two letters are the same and might is well go with that and say it is one plus the result of optimally solved in the rest, what if they not the same this is the interesting case.

Supposing, these are not the same then what happens. Then can we just go ahead and ignore a 0 and b 0, no right. So, it could be that a 0 actually matches b 1 or it could be that b 0 matches a 1, we do not know, but we certainly know that a 0 does not match b 0. So, we have to drop one of them because we cannot make a solution a 0 matching b 0, but we do not know which one. So, what we do is we take two sub problems, we say let us assume b 0 is not part of the solution then the best solution come out of a 0 to a and minus 1 and b 1 to b n, b 0 is exclude because I cannot match it to the a 0 and whatever a 0 matches must match to the right. So, i must go ahead with it.

But maybe this is the wrong choice. The other choice should be to keep b 0 and drop a 0 and which case I do a 1 to a m minus 1 and b 0 to b m. These are two different choices which one to choose, well since we do not know we solve them both. We solve if a i a 0 is not b 0 we solved both these problems a 1 to a m minus 1 b 0 and a 0 to a m minus 1 b 1 solve of them take the maximum one whichever one is better it is the one.

(Refer Slide Time: 18:45)



Inductive structure

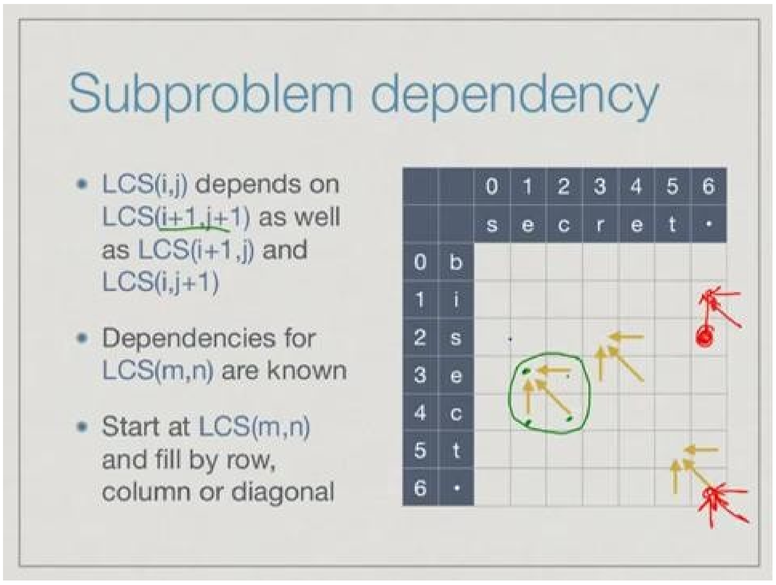
u	a_i	a_{i+1}	a_{i+2}	…	…	a_{m-1}	a_m
v	b_j	b_{j+1}	b_{j+2}	…		b_{n-1}	b_n

- LCS(i,j) stands for LCS($a_i a_{i+1} \ldots a_m$, $b_j b_{j+1} \ldots b_n$)
- If $a_i = b_j$, LCS(i,j) = $\underline{1 + \text{LCS}(i+1,j+1)}$
- If $a_i \neq b_j$, LCS(i,j) = $\underline{\max(\text{LCS}(i+1,j), \text{LCS}(i,j+1))}$
- 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

This in general will take us deeper in the words. So, we said a 0 b 0 will require solved it for a 1 and b 0 or a b a 0 and b 1. So, in general we have a i and b j right. Again since we have a i and b j then you will use the same logic if a i is equal to b j then it is one plus the rest. So, this is the good case, if a i is not equal to b j then what we do is we look at the same thing, we drop b j and solve it and symmetrically we drop a i and solve it and take the better of the two. We take max of the solution from i and the solution from j plus 1.

If we say like we had before that lcs of i j is the length of the longest common starting to i and j if a i is equal to b j it will be one plus the length start it from i plus 1 j plus plus 1. If it is not equal it will be the maximum of the two sub problems where either increment i or increment j and has with the longest common subword when we go to the last position m and n we get 0.

(Refer Slide Time: 20:01)



Subproblem dependency

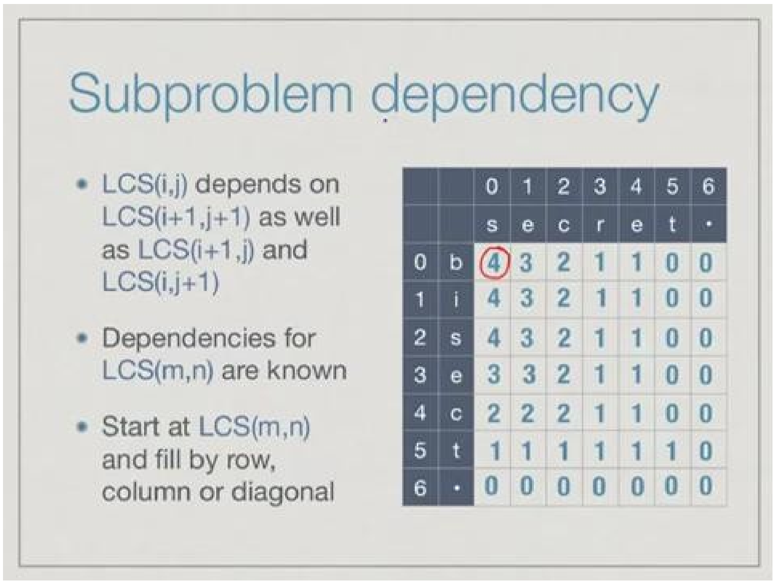
- LCS(i,j) depends on LCS(i+1,j+1) as well as LCS(i+1,j) and LCS(i,j+1)
- Dependencies for LCS(m,n) are known
- Start at LCS(m,n) and fill by row, column or diagonal

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

So, here the dependency is slightly more complicated because depending on the case, I either have to look at i plus 1 j plus 1 or I have to look at i plus 1 j or i j plus 1. So, I had for this square, I had looked at its right neighbor, right diagonal neighbor and the bottom neighbor, but once again the ones which have no dependency appear. So, earlier we had for longest common subword we had only this dependency this mean that even a square like this had no dependencies because there is nothing to its bottom right.

But now, for instance if we look at this picture, since we are looking bottom right and left, if I look at this its dependencies are in three directions; two of the directions are empty, but this direction there is dependence. So, I cannot fill up this square directly the can only square, I can fill up directly is this one because it has nothing to its right nothing in diagonally and nothing below.

(Refer Slide Time: 20:54)



Subproblem dependency

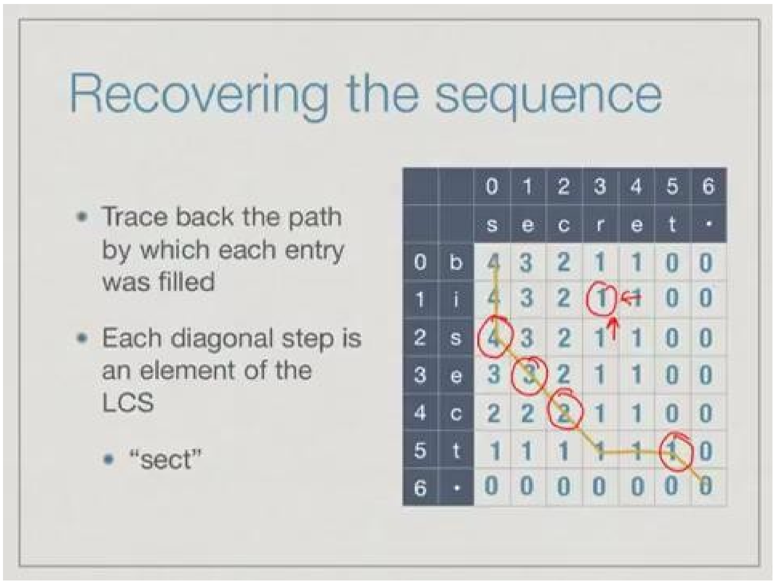
- LCS(i,j) depends on LCS(i+1,j+1) as well as LCS(i+1,j) and LCS(i,j+1)
- Dependencies for LCS(m,n) are known
- Start at LCS(m,n) and fill by row, column or diagonal

		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, I start from there and I put a 0 and as before we can go down this because now once we have this, we have everything would to its left and once we have this and because we are beyond the word where at the m, this dummy position, the row and column becomes 0, but the important thing to remember is the row and column become 0 not because they have no dependency, but because we can systematically, fill it up exactly like in the grid parts we can fill up the bottom row and the left most column there, here the right most column.

Now, once we have this we can fill up this part right and then again we can have two and three c entries we can fill up this. We have three entries we can fill up this, we have three entries we can fill up this and we can fill up this column and we can do this column and we can do this column by column and we propagate it and then finally, the value the propagates here is our longest length of the longest common subsequences, we could also do this row by row.

(Refer Slide Time: 21:44)



Recovering the sequence

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

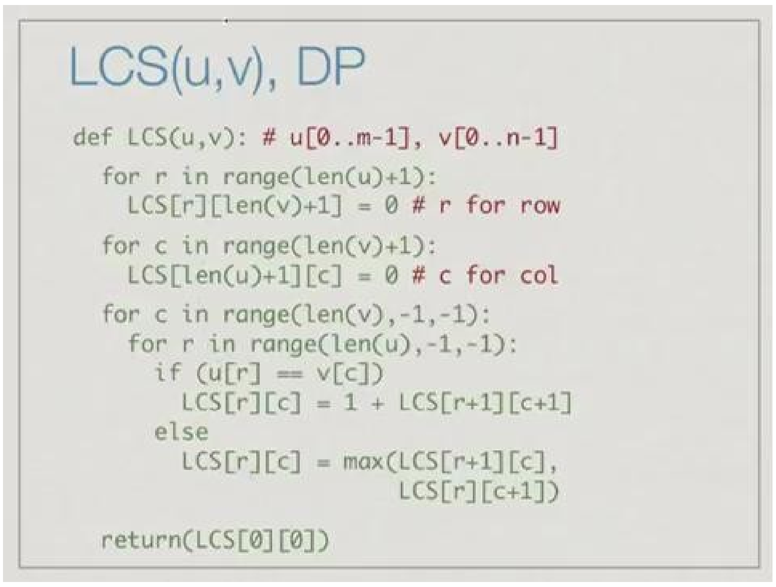
		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	4	3	2	1	1	0	0
1	i	4	3	2	1	1	0	0
2	s	4	3	2	1	1	0	0
3	e	3	3	2	1	1	0	0
4	c	2	2	2	1	1	0	0
5	t	1	1	1	1	1	1	0
6	•	0	0	0	0	0	0	0

Now, how do we trace out the actual solution when the solution grows, whenever we increment the number? So, we can ask why is this 4? So, we say that this is 4 not because we did plus 3 because s is not equal to b, we did 4 because we got the max value from here, why is this 4 again i is not equal to s. So, we got the max value from here why is this 4, Oh s is equal to x. We must have got it by 3 plus 1, why is this because e plus e. So, we must have got it from here. So, we follow the path according to the choices that we made in applying the inductive function in order to generate the value at each parts.

In other words, for each cell i j that we write we remember whether we wrote it because it was one plus that diagonal neighbor or the maximum of the left in the right in which case we record whether the left or the bottom it was the maximum. Now, in this a picture every time we take a diagonal step it means we actually had a match. So, this is the match the first one here is a match s equal to s, this is the match e equal to e, this is the match c equal to c. Now, after this point we are flat and then a at this point again we have a match. So, we get s e c and t.

So, we can read of the diagonal steps along this kind a explanation of the longest number largest number we got and each diagonals step will contribute to the final solution, now there could be more than 1, because we have not got any example in this case, but sometimes the max could be one of in both directions if I am taking max of the left the right neighbor and the bottom they could be the same. I could have the situation like this supposing I landed up here then I do not know whether I got it from here or from here. So, I might have two different extensions which lead me to a solution. So, the longest common subsequence need not be unique, but you can recover at least one by following this path

(Refer Slide Time: 23:32)

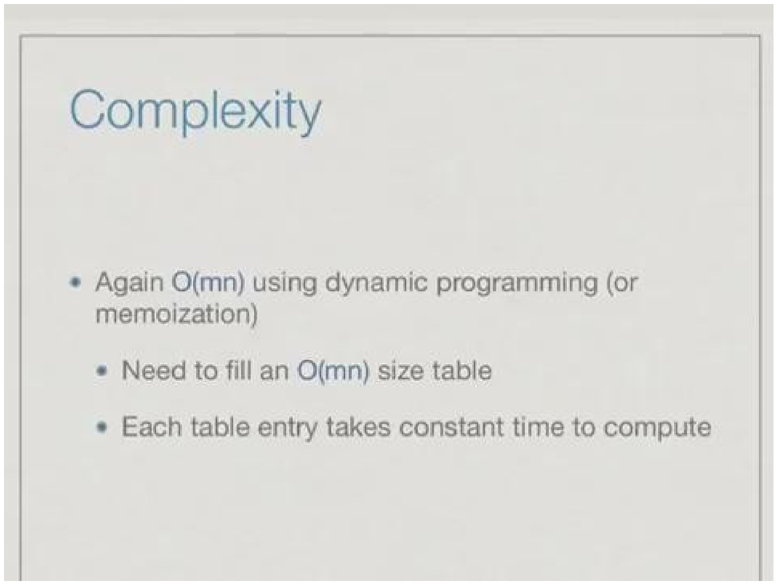


LCS(u,v), DP

```
def LCS(u,v): # u[0..m-1], v[0..n-1]
  for r in range(len(u)+1):
    LCS[r][len(v)+1] = 0 # r for row
  for c in range(len(v)+1):
    LCS[len(u)+1][c] = 0 # c for col
  for c in range(len(v),-1,-1):
    for r in range(len(u),-1,-1):
      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, here is the python code is not very different from the earlier one. We can just see we have just initialize the last row and the bottom row on the last column and then as before you walk up row by row, column by column and filling using the equation and in this case, we do not have to keep track of the maximum value and keep updating because the maximum value automatically propagates to the 0 0 value.

(Refer Slide Time: 23:56)



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

Just like the longest common subword, here once again we are filling in a table of size m times n. Each entry only requires you to look at most do three other entries. So, one to the right one to the bottom right in that the one below. So, it is a constant amount of work. So, m n entries constant amount of work per entry, this takes time m times n.