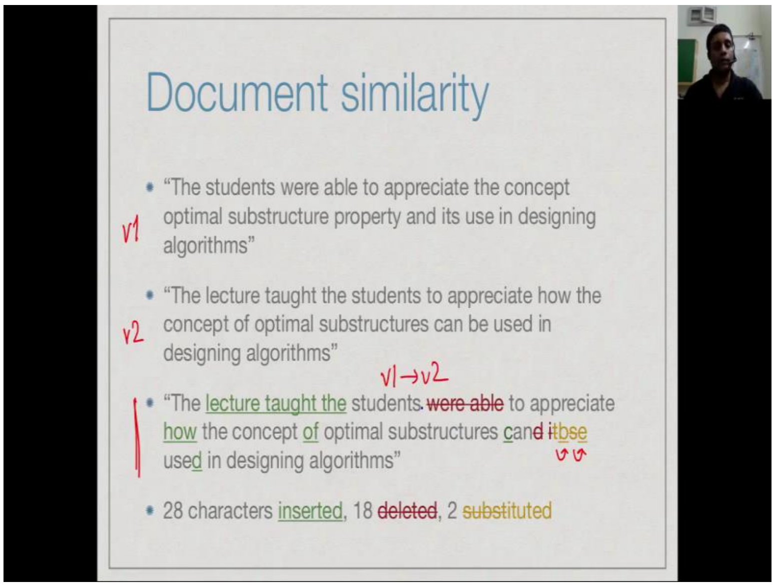
Design and Analysis of Algorithms, Chennai Mathematical Institute
Prof. Madhavan Mukund
Department of Computer Science and Engineering,

Week - 07
Module - 05
Lecture - 48
Edit Distance

Having looked at the longest common subword and subsequence problem, we now look at a closely related problem called Edit Distance.
(Refer Slide Time: 00:09)



Document similarity

- "The students were able to appreciate the concept optimal substructure property and its use in designing algorithms"

v1

- "The lecture taught the students to appreciate how the concept of optimal substructures can be used in designing algorithms"

v2

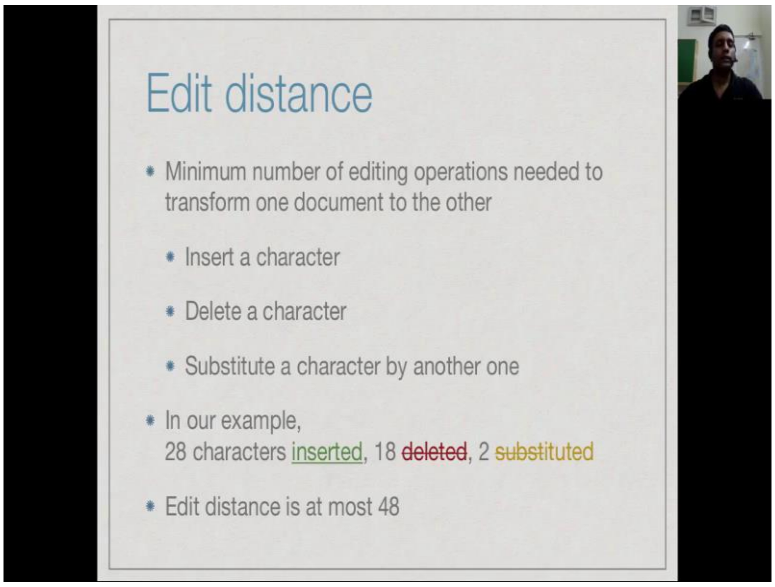
$v1 \to v2$

- "The lecture taught the students~~.~~ ~~were able~~ to appreciate how the concept of optimal substructures can~~d~~ ~~it~~bse used in designing algorithms"
- 28 characters inserted, 18 ~~deleted~~, 2 ~~subst~~ituted

So, the aim is to measure how similar two pieces of texture, it is so called document similarity problem. So, let us look at the following two sentences; the first sentence says the students who were able to appreciate the concept optimal substructure property and its use in designing algorithms. The second sentence says the lecture taught the students to appreciate how the concept of optimal substructures can be used in designing algorithms. So, the third sentence here indicates how one might obtain one of these two from the other.

If you have used certain document preparation systems which allow you to track changes, it will typically indicate the changes between one version of a document and other version like this. So, here the green indicates letters which have been, so if we called as version 1 and this version 2, then in going from version one1 to version 2, what we have done is you are introduced the letters in green. So, you have inserted these characters, you have deleted the letters marked in red with a line through them and in this yellow we have replace. So, we have replaced the t by v and an s by n. So, we have 28 characters, we can count that there are 28 characters which have been inserted including spaces and other, 18 characters have been deleted and 2 characters have been substituted.

(Refer Slide Time: 01:41)



Edit distance

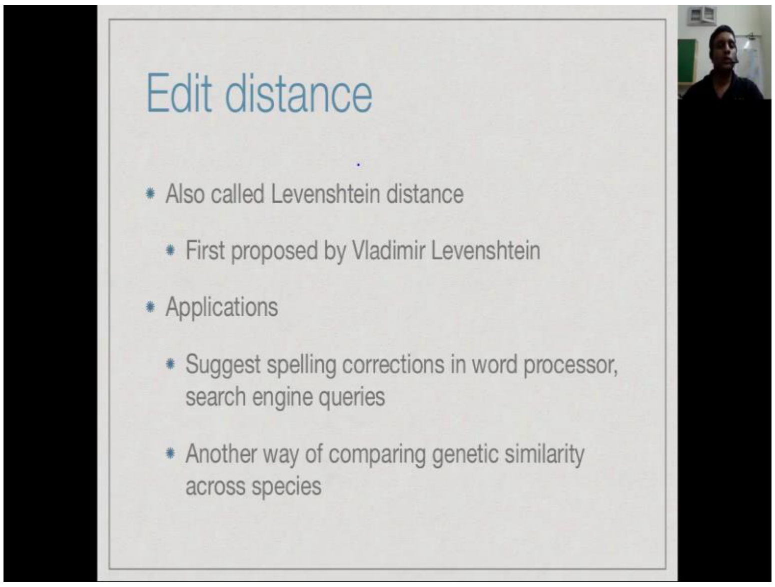
- Minimum number of editing operations needed to transform one document to the other
  - Insert a character
  - Delete a character
  - Substitute a character by another one
- In our example, 28 characters inserted, 18 ~~deleted~~, 2 ~~substituted~~
- Edit distance is at most 48

So, now, this can be a basis of measuring how close two documents are close to each other. So, the edit distance with the minimum number of edit operations required to transform one document to another, so we have to define what we mean by editing operations. So, let us just start with very basic operation, either you can insert a character or you can delete a character or you can replace one character by another one in one step.

So, replacing of course could be a think taught of us deleting and inserting, so that could be a two step operations, I delete the character and then I insert the character I want. But, I am going to allow changing one letter a by b or s by t as a single operation. So, in our example we claimed that 28 characters were inserted, 18 were deleted and 2 were substituted. So, the total number of changes we made is 48.

If the edit distance is suppose be the minimum number of changes, it cannot be more then 48, because they already shown that it is possible to do in 48 characters, possible to do it in more clever way less than 48. So, the edit distance is at most 48 for the two sentence that we shown earlier.

(Refer Slide Time: 02:44)



Edit distance

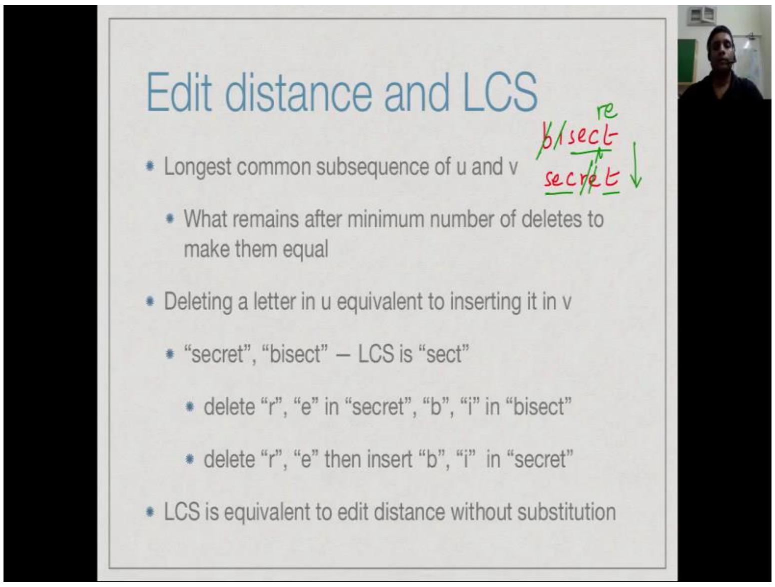
- Also called Levenshtein distance
  - First proposed by Vladimir Levenshtein
- Applications
  - Suggest spelling corrections in word processor, search engine queries
  - Another way of comparing genetic similarity across species

So, this distance is also called the Levenshtein distance, because it was first proposed by the Soviet, now Russian scientist called Vladimir Levenshtein and this like the longest common subsequence problem, it is extremely useful in practice. So, the first thing is to suggest spelling corrections. Now, if somebody types something that is a wrong, then it is spelling corrective will have to suggest the correct word from the dictionary to replace it. So, which word should the spelling corrective choose?

So, one criterion for choosing is to identify among all the words in the dictionary that are possible which one is closest to the one that has been typed. So, this can be measured in terms of this edit distance and then that many you will ((Refer Time: 03:29)) did you mean typist. This also happens when you type queries and search engines. So, if you type something to Google, Google will sometimes change your query to a word which is meaningful, because it recognizes that you mistyped a name or a concept.

We also said that the longest common subsequence problem that we saw earlier is useful in Genetics, in Bioinformatics and in the same way edit distance also, if you want to compare the genetic information in two different species, then it is natural to become to compare them in terms of the content of the DNA and DNA have just long sticks. So, you want to find out how easy or difficult it is to transform one piece of DNA to another and depending on that we can tell whether two species are closed to each other or not.

(Refer Slide Time: 04:15)



Edit distance and LCS

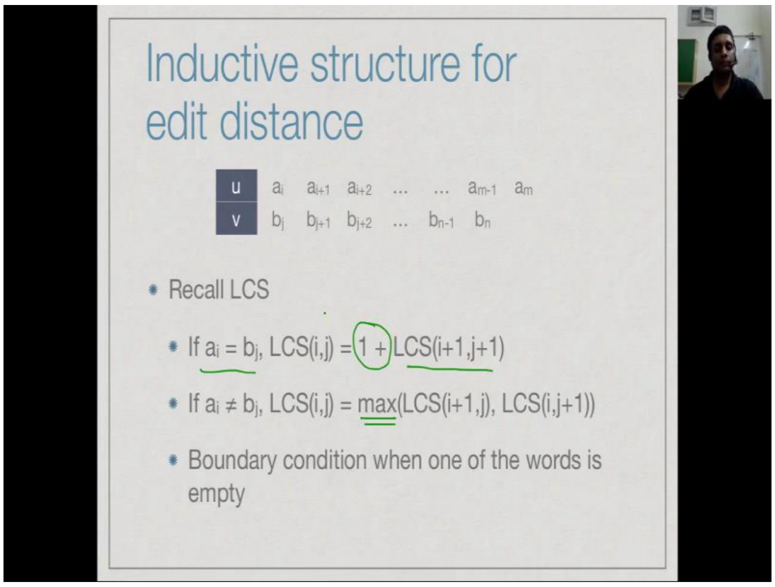
- Longest common subsequence of u and v
  - What remains after minimum number of deletes to make them equal
- Deleting a letter in u equivalent to inserting it in v
  - "secret", "bisect" – LCS is "sect"
    - delete "r", "e" in "secret", "b", "i" in "bisect"
    - delete "r", "e" then insert "b", "i" in "secret"
- LCS is equivalent to edit distance without substitution

re
bisect
secret

So, if you go back to the longest common subsequence problem, so one way of thinking a longest common subsequence is supposing I just delete all the parts which are not part of the LCS, so I had things like bisect and secret. So, now, in this if I deleted the b and i from here and the r and e from here, then I am left with exactly the longest common subsequence. The other way of thinking about it is let identity, I have start with one word and I am transforming it to this word, so I first delete the b and i and then I insert here the r e.

So, I do not operate on both words and delete from both, I operate only on the first word. So, I delete the words, I delete the letters I do not want, because I not have a second word and I insert the letter, so I do want which are there in the second not in the first and in this way I transform the first word to the second word and this is equivalent to deleting from both and come it to a common subsequence. So, this tells us that LCS is somehow equivalent to computing the edit distance, if I only allow delete and insert. So, the interesting thing about edit distance is that it also allows a substitution of one character and other. So, it might give us a slightly different metric from longest common subsequence.

(Refer Slide Time: 05:42)



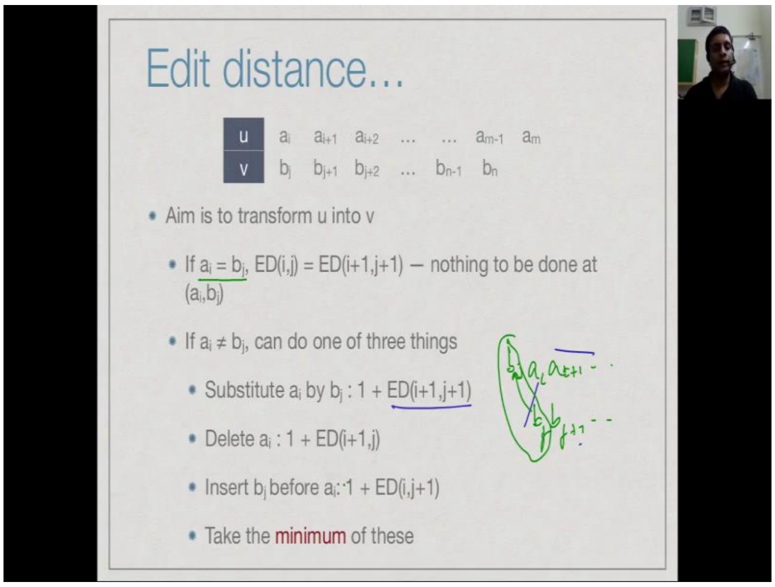
Inductive structure for edit distance

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	

- Recall LCS
  - 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))
  - Boundary condition when one of the words is empty

So, going back to longest common subsequence, it says that if two letters match at the beginning of a word, then it is useful to assume that the common subsequence includes that. So, I have this one which says that a i is equal to b j, so I include this in my subsequence and then I solve the rest of the problem. Otherwise, I take the max of the two sub problems that I get by dropping a i or b j.

(Refer Slide Time: 06:13)



Edit distance…

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	

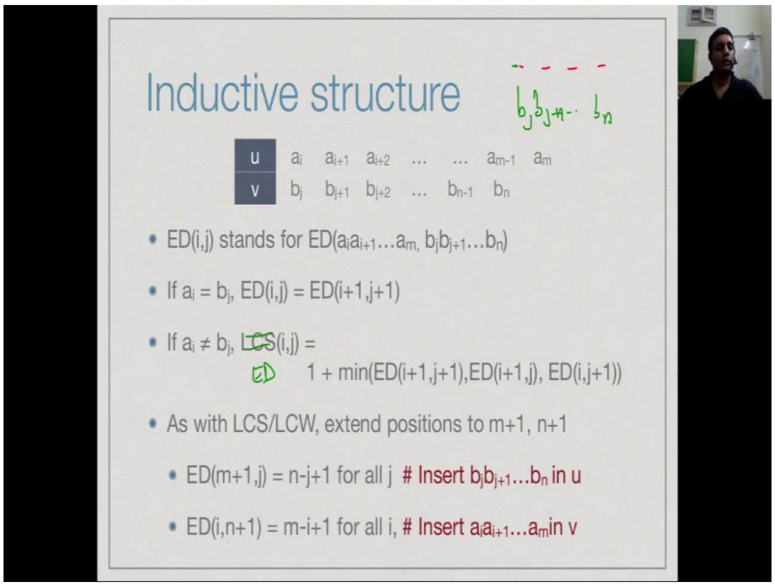
- Aim is to transform u into v
  - If $a_i = b_j$, ED(i,j) = ED(i+1,j+1) — nothing to be done at (a_i,b_j)
  - If $a_i \neq b_j$, can do one of three things
    - Substitute a_i by b_j : 1 + ED(i+1,j+1)
    - Delete a_i : 1 + ED(i+1,j)
    - Insert b_j before a_i : 1 + ED(i,j+1)
    - Take the minimum of these

Now, in the edit distance problem we have a similar criterion, if a i is equal to b j, then I have nothing to do. So, the edit distance of word starting at i and j is the same as the edit distance starting at i plus 1 j plus 1. Recall that we are now doing the opposite of longest common subsequence, we are now trying to minimize the number of changes in order to maximize the number of matches. So, if we see that something is already matched, then no changes needed, so we just move ahead to the next position.

Now, if they do not match now three things are possible, so I have a i, a i plus 1 and so on and I have b j, b j plus 1 and so on. So, in the first thing I can do is I can directly make this into b j, in which case I have now made the first letters exactly equal and so I just need to look at i plus 1 j plus 1. The other thing which I can do is to remove this a i all together, I just remove this a i. So, now, this disturbance is gone, but I am still left to the rest to the problem, so now, I have to now see if i plus 1 onwards matches j onwards.

And finally, the last thing which I could have done is to introduce that this b j at the beginning, so now, this means that these two things now match up. So, now, I have to look at a i onwards and b j plus 1 onwards and whichever of these three requires the minimum work overall is the one i 1. So, I want to take 1 plus in the solution i plus 1 j plus 1 or 1 plus solution i plus 1 j or 1 plus solution for i j plus 1 and take the minimum of these three.

(Refer Slide Time: 07:59)



Inductive structure

$b_j b_{j+1} \dots b_n$

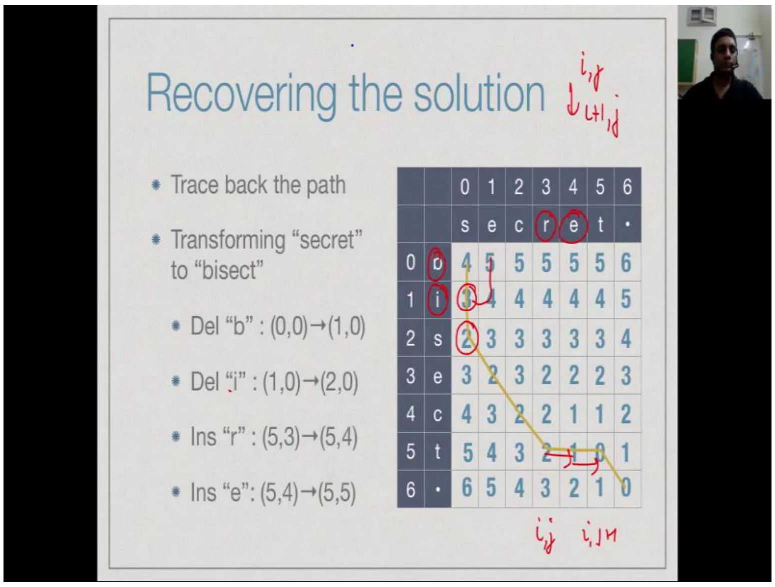
u	a_i	a_{i+1}	a_{i+2}	...	...	a_{m-1}	a_m
v	b_j	b_{j+1}	b_{j+2}	...	b_{n-1}	b_n	

- ED(i,j) stands for ED($a_i a_{i+1} \dots a_m$, $b_j b_{j+1} \dots b_n$)
- If $a_i = b_j$, ED(i,j) = ED(i+1,j+1)
- If $a_i \neq b_j$, ~~LCS~~(i,j) =
  ED 1 + min(ED(i+1,j+1),ED(i+1,j), ED(i,j+1))
- As with LCS/LCW, extend positions to m+1, n+1
  - ED(m+1,j) = n-j+1 for all j # Insert $b_j b_{j+1} \dots b_n$ in u
  - ED(i,n+1) = m-i+1 for all i, # Insert $a_i a_{i+1} \dots a_m$ in v

So, this gives us the final inductive structure that we want. So, ED the Edit Distance for the word starting at i and j, if a i is equal to b j it is ED of i plus 1 j plus 1. If it is not equal, then it is 1 plus the minimum of the edit distance of these three different sub problems and as usual we extent this to m plus 1 n plus 1, but the interpretation is now different. If one of the words is empty, then the number of changes I need to make is to transform everything else or insert everything from that word you use.

So, if I am looking at b j, b j plus 1 to b n and beside I have nothing, then what is the edit distance. Well, I have to basically insert this many letters. How many letters are there? N n minus j plus 1, so the edit distance when u is empty and v has z position j is n minus j plus 1. Likely, likewise b is empty and u is not, it is n minus i plus 1, we have to insert all those like this. So, this is a slight difference, it is not zero, but it is the cost of using up the letters which are there, which are missing in this word.

(Refer Slide Time: 09:13)



Recovering the solution

- Trace back the path
- Transforming "secret" to "bisect"
  - Del "b" : (0,0)→(1,0)
  - Del "i" : (1,0)→(2,0)
  - Ins "r" : (5,3)→(5,4)
  - Ins "e": (5,4)→(5,5)

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

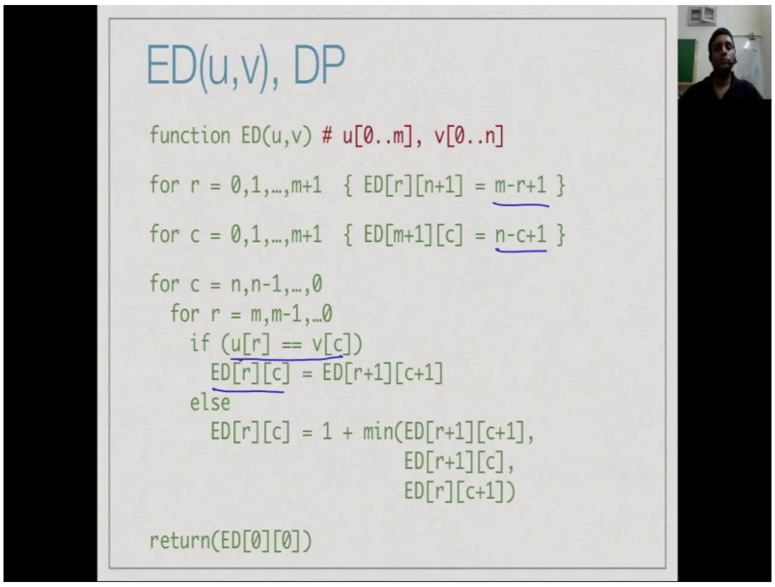
So, the inductive substructure in edit distance is exactly the same as in LCS. Every position depends on i plus 1 j plus 1 and though i plus 1 j and i j plus 1. So, we have the same three neighbors depended C that we had as we did before and us usual we can start at the bottom right corner and work backwards row by row or corner by corner. So, in this case remember that the boundary condition is not zero, but it is n minus j plus 1, so as I go up, the number increases.

So, this is my boundary and now I can just apply my recursive thing which says that t matches, for edit distance at t is zero, because I have nothing to do and there is nothing to beyond them. Everywhere else, because I do not match, the edit distance is just going to be the minimum of the remaining three. So, it is minimum to the remaining three plus 1, so in this case if I pick, look at this for example, the minimum of 0, 1 and 2 is 0 plus 1 is 1.

The minimum of 4, 2 and 3 is 2, plus 1 is 3 and so on, so it is the minimum of those plus 1 to make the current. So, I keep doing this and I go left, eventually I will find that the minimum edit distance between bisect and secret is actually 4, I need to make four changes, just using the inductive definition that we have done earlier. And as before now the question is how do I recover the solution from this, so I follow the path. So, why did I get a 4 there, because it is the minimum or it is three ((Refer Time: 10:53)) not equal to s. What is the minimum of these three? It is 3, so I went back.

Similarly, why is this 3 here, because this is the minimum of these, so I went down? So, every time I go down, it amounts to saying that I go from i j to i plus 1 j and if you are remember in our case, i plus 1 j and i i is corresponds to deleting. Similarly, at this place I go a right. This means I go from i j, i j plus 1, this corresponds to inserting. So, what I can read of from here is that when I am going to delete this v, I delete this i, then I come here, I insert this r and I insert this e. And these are exactly the four changes that I need to make in order to transform bisect into secret.

(Refer Slide Time: 11:41)

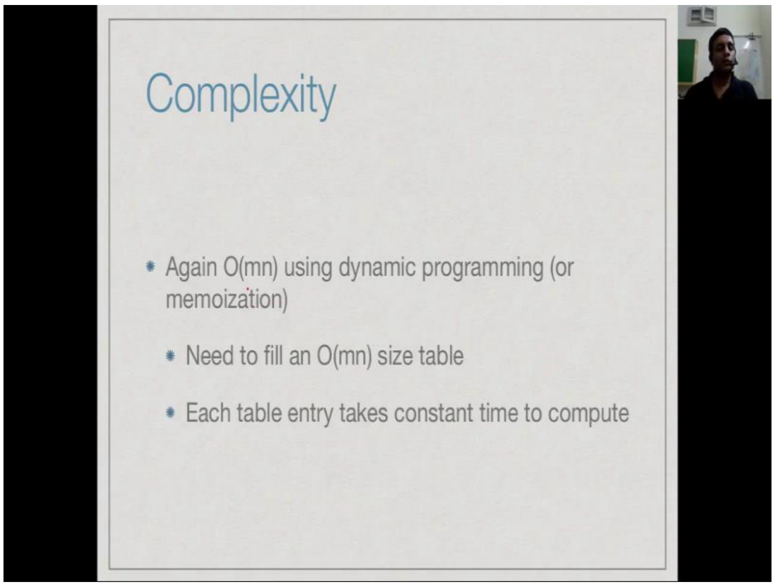


ED(u,v), DP

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

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

for c = 0,1,…,m+1  { ED[m+1][c] = n-c+1 }

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

return(ED[0][0])
```

So, the pseudo code for every distance is very similar to the longest common subsequence. The only major change done is in the initialization, the values at the boundary are not zero, but whatever is required to complete the editing. Then, as usual I start at the bottom right end and I do columns in right to left rows from bottom to top, if the current value if the two values that I am looking at u of r and v of c are it is same, then I just postponed the edit distance to the next thing, I have nothing to do here. Otherwise, I take the minimum of the sub problems and add 1 for the current changes and finally, I have returned the value 0 and 0.

(Refer Slide Time: 12:24)

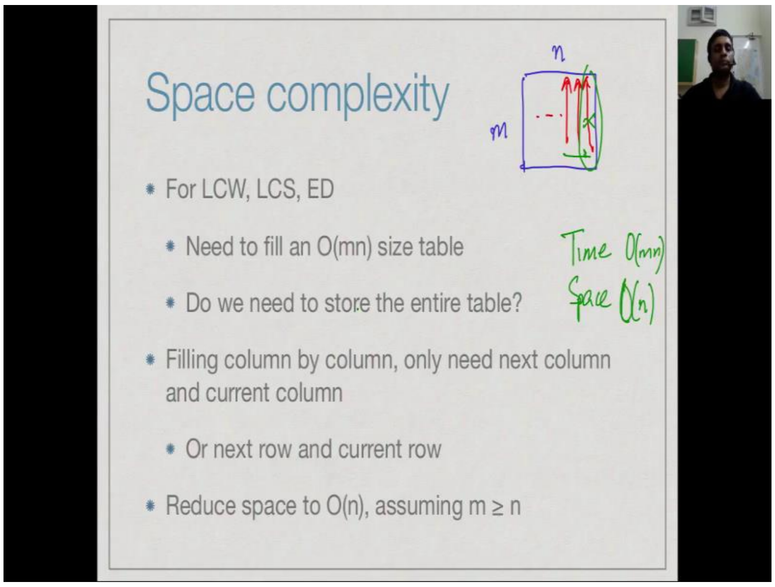


Complexity

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

The complexity is order m times n, it is exactly the same as the previous one, because after fill up this table of size m times n and it takes a constant time to compute. Now, in all these three problems there is another issue which we have not addressed. So, let us deal with it now.

(Refer Slide Time: 12:41)



Space complexity

- For LCW, LCS, ED
  - Need to fill an O(mn) size table
  - Do we need to store the entire table?
- Filling column by column, only need next column and current column
  - Or next row and current row
- Reduce space to O(n), assuming m ≥ n

n
m
Time O(mn)
Space O(n)

So, this is a space complexity. So, we have this table that we have to compute which we have a green is m times n, but notice that the way we computed it for example, was to compute this column, then this column, then this column and so on. So, at any given point, when I am computing this second column for example, I only read the first column. When I need the third column, this column is now not needed anymore, because the third column depends only on the second column from the right.

So, if I need, if I fill column by column I only need the next column, the column on my right and the current column or if I need row by row, the next row and the current row. So, in other words while I just keeping two columns or two rows, I can completely compute this thing back to 0 0. So, therefore, there was actually no need to rule m times n size table as storage. I still have to compute n times, but the time complexity remains m times n. So, time is m times n what we are saying that this space can be reduced if, left us say that n is the smaller of the two, this space can be reduced topper n by just keeping two columns.

And now you might ask how do I recover the solution, because in the solution I trace back the path on the whole thing, well you can actually keep track incrementally as you are going, you can build up the solution associated to each entry incrementally, so even that can be done in order n space. So, you do not need this table at all even to compute back the witness for the given numerical solution. So, in all these problems where we have a very limited neighborhood dependency in our dynamic programming, you can actually often make do much less space, then it seems to require in terms of the actual table as given to you.