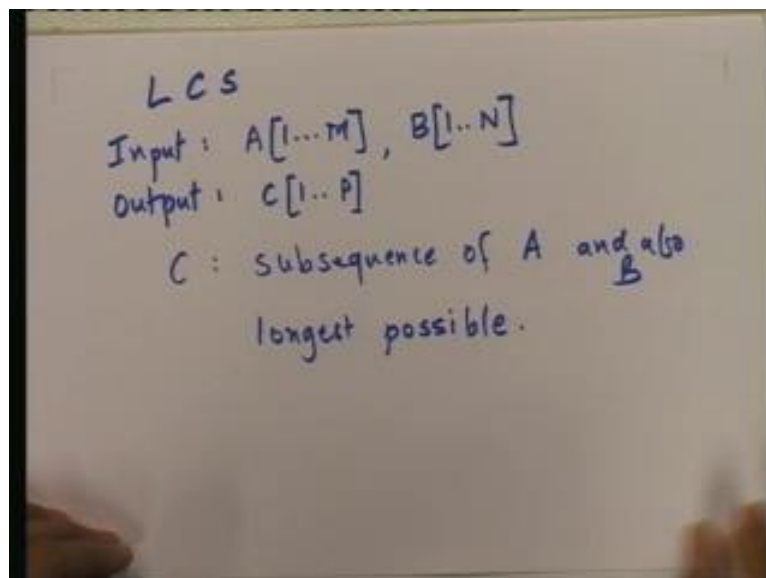


**Design & Analysis of Algorithms**  
**Prof. Abhiram Ranade**  
**Department of Computer Science Engineering**  
**Indian Institute Technology, Bombay**

**Lecture - 19**  
**Longest Common Subsequence**

Welcome to another lecture on Design and Analysis of Algorithms. Today's lecture will also be on dynamic programming. The topic for today is the Longest Common Subsequence. Let me, start by defining the problem.

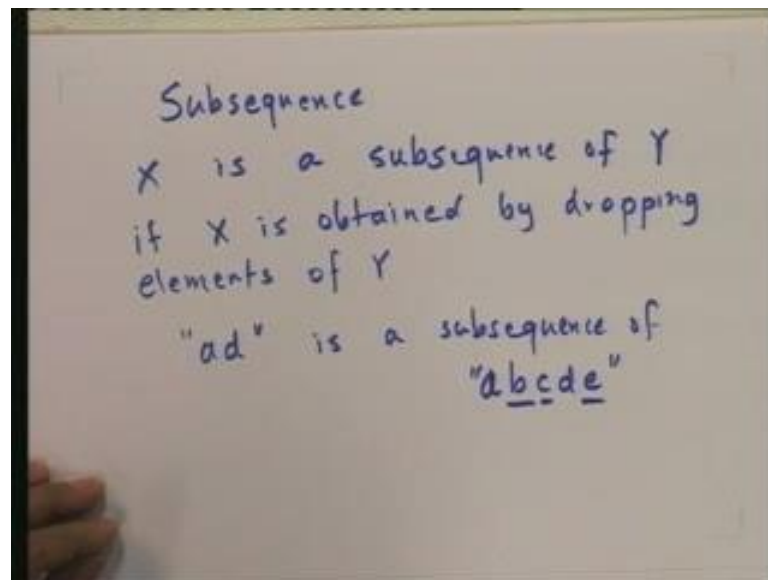
(Refer Slide Time: 01:09)



So, the longest common subsequence will be abbreviated as LCS. The input to this problem consists of two strings. So, there is one string A of length M, and another string B of length N. I might string occasionally, but I do mean sequence, it does not really matter. The sequence will be a sequence of characters typically. So, I will use string and sequence interchangeably again.

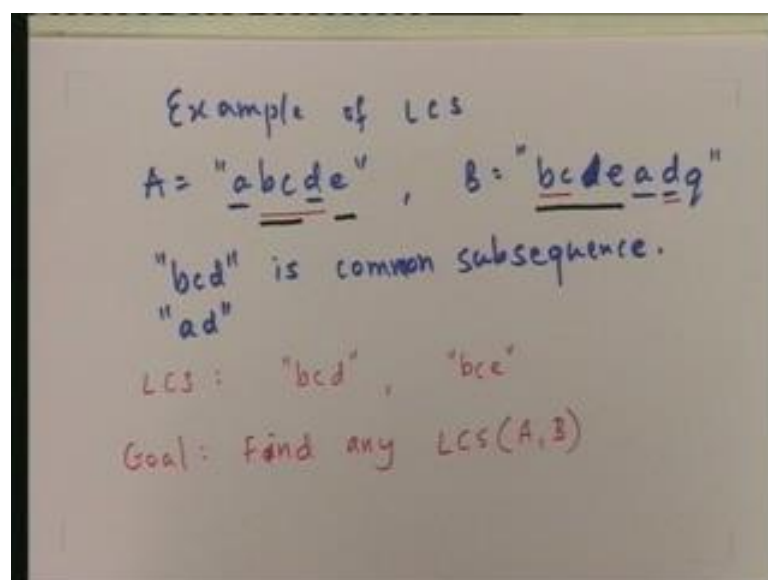
The output is also going to be a string, let me call it C. And say it will have some length P, which is as yet unknown. And we require that C have the following property. So, C should be a subsequence of A as well as B. That is what a common subsequence is. I will define subsequence more formally in a minute. So, C should be a subsequence. And we want C to be the longest possible subsequence. So, this is your problem. Let me define what a subsequence is.

(Refer Slide Time: 02:34)



So, we will see that X is a subsequence of Y, if X is obtained by dropping elements of Y. And of course, the remaining elements, which hidden drop should be kept in the same origin. So, for example, a d is a subsequence of say a b c d e. So, these are the elements that we dropped, and as we said common subsequences or just sequences of both. So, let us take an example of LCS.

(Refer Slide Time: 03:42)



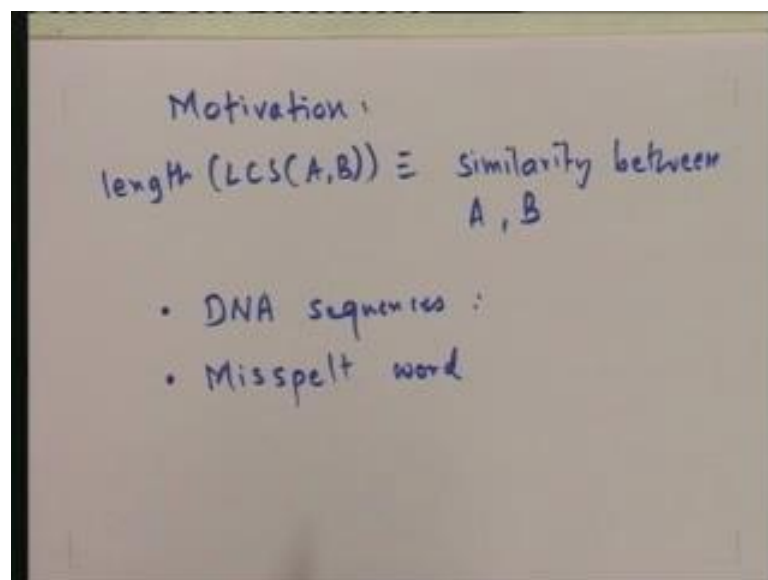
So, let us say A is a sequence of characters a b c d e. And let us say B is a sequence of characters say b c d or let me say, let me write it as b c e a d and some q. They do not

have to give the same length. So, clearly b c d is a subsequence of both. And therefore, it is a common subsequence. a d also is a common subsequence. So, a d for example, appears over here and over here. b c d. For example, appears over here and over here. So, now we know what common subsequences are we would like to define, but the longest common subsequences.

So, LCS of A and B for example, is b c d. You can look through the sequences and you will figure out, the only common sequences of length 3 are very few. And in fact, there is no common sequence of length 4. And so therefore, this is A long common subsequence. In fact, you can see that b c e is also a common subsequences. So, let me just underlying that, so say b c e appears over here. And over here it appears over here. So, the goal now is to find one of these common subsequence.

So, goal in this LCS problem find any longest common subsequence of A and B. We do not care which one, but we want one of those. When we speak of LCS, we will mean either every LCS or one of the LCS's. And what we mean should generally, we clear by context otherwise will explicitly ((Refer Time: 06:21)).

((Refer Slide Time: 06:26))

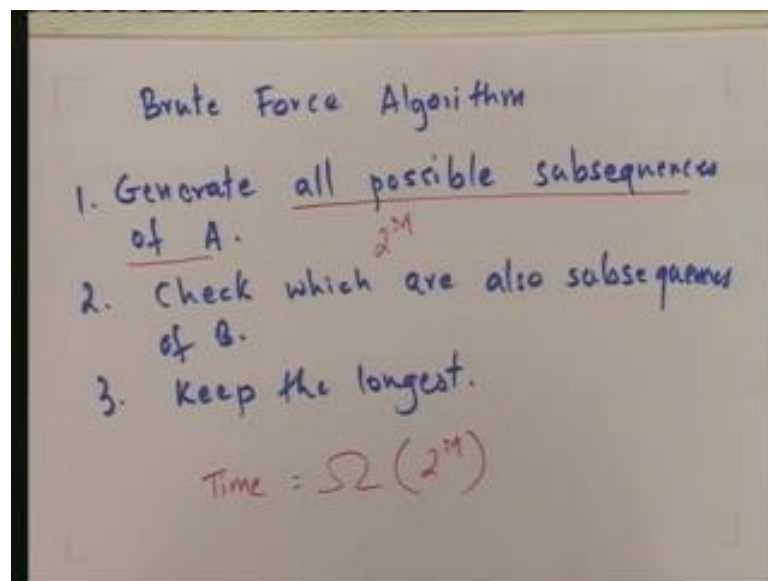


Let me give you a ((Refer Time: 06:26)) motivation about, why we care about this LCS problem. The length of the LCS of say A and B can be thought of as representing the similarity between A and B. Clearly, the longer this LCS's the more similar A and B ((Refer Time: 06:58)). You might ask, why do we care about strings similarity. The

answers to that are fairly well known and obvious probably. So, say we have two DNA sequences. And we want to know are they very similar.

If they are very similar maybe there is a common evaluation in those sequences. And so therefore, some measure of similarity is useful, suppose we have a miss pelt word. We would like to know the word, which is most similar to ((Refer Time: 07:38)) from our given dictionary. Again, we need some measure of what similarity is. So even here, the notion of an longest common subsequence is useful. As it turns out, LCS is not the only measure of similarity, there are others as well. But, interestingly enough the algorithms we develop a LCS will be somewhat similar. And you can probably extend them for other measures of similarity as well. Let me, start by considering a Brute force algorithm for LCS.

(Refer Slide Time: 08:14)

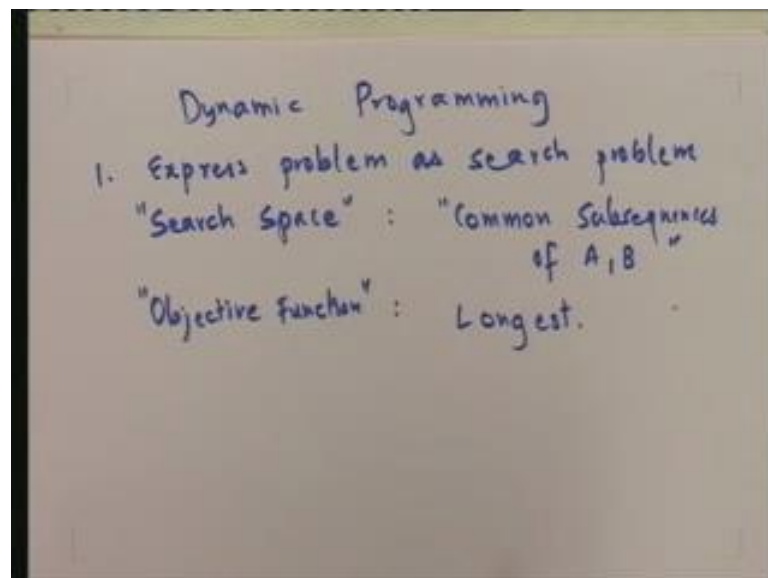


This is probably a good idea before embarking on trying to discover a complicated algorithm. It at least tells you something. If, not you have something to fall back up on. So, what could be the brute force algorithm. So, the idea could be to say let us generate all possible subsequences of A. This will be the first step. Then we will check, which are also subsequences of B. And then, we retain the longest. As we said, there could be several longest, but we do not care we will just keep one of those. And that could be the answer that you could print.

Is this algorithm good enough? Well, this is algorithm good, well certainly will written an answer. But the question is will it be fast. Now, it is not going to be very fast, because a number of all possible sequences of A is quite large. In fact, you know that if sequence as length M. Then there are  $2^M$  possible subsequences. So, therefore, the time for this algorithm is going to be at least  $2^M$ . In fact, if you want to write down the sequences the subsequences, you will also multiply this by the M itself.

But, any in case that is at least Exponential M. So, that is not what you want, we would like something faster. And in fact, we have said earlier, that we are going to apply dynamic programming for this. And our hope is the dynamic programming will give a something better. So, let me review, what the dynamic programming ideas are.

(Refer Slide Time: 10:16)



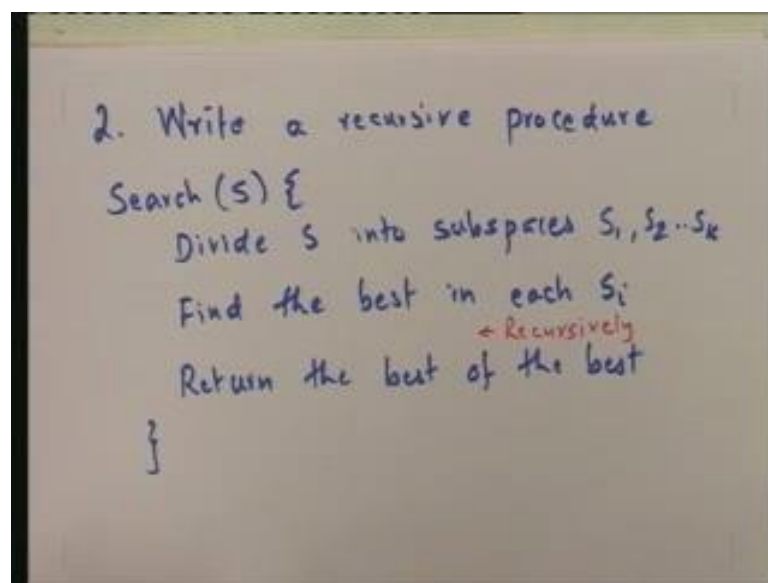
So, the first idea was that we should try to express the problem as a search problem. By that I mean, that you should clearly identify what the search space is. So, what is it, what is a possible candidate that you are looking for? So, in our case, the search space consists of common subsequences of A and B. So, we are going to search through the common subsequences. And so, in that sense is first step is over well not quite. So, there is also one more thing, which is we want to determine the objective function.

So, what is the objective function? So, let us say of all the elements in the sub space, which is the one that you are going to pick. So, in our case, that easily stated it is a longest one. So, we have a space consisting of all subsequences of A and B. And we

want from it. That string which is the longest. Of course, we are not going to generate this space explicitly. We are not going to ever write down anywhere all the common subsequences that will take too much time. This is just for the purposes of thinking.

So, let me write the term. So, this is only help in thinking. We are not actually going to generate the subspace any of these spaces. Then, what we are going to do is. We are going to write the recursive procedure. Listed explicitly what a search space is having defined explicitly, what a search spaces.

(Refer Slide Time: 12:28)



2. Write a recursive procedure

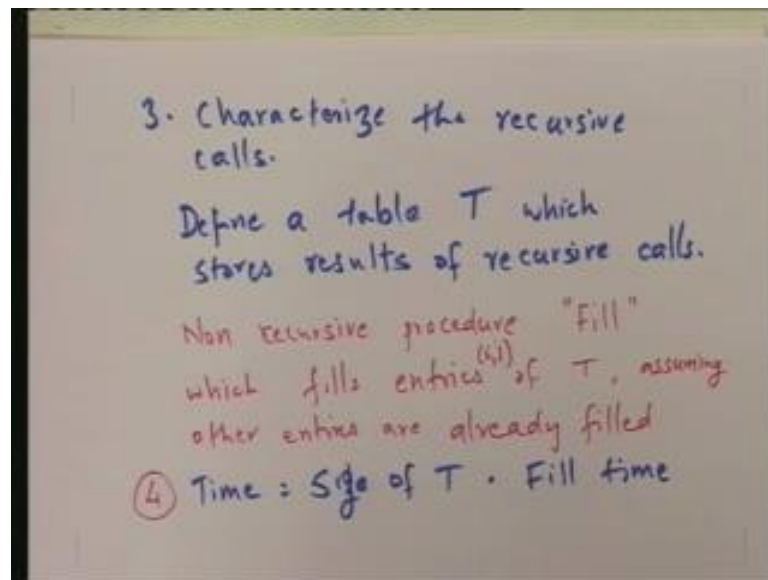
```
Search(S) {  
  Divide S into subspaces  $S_1, S_2 \dots S_k$   
  Find the best in each  $S_i$   
  Return the best of the best  
}
```

The handwritten text is on a piece of paper. The word 'Recursively' is written in red ink next to the line 'Find the best in each  $S_i$ '.

The next step is to write a recursive procedure, which looks something like this. So, it is going to be a search. And it will take as argument the space has which we want to search. And now, the procedure typically will divide  $S$  into subspaces. Say, let me call those  $S_1, S_2$  say some  $S_k$ . So, there are  $k$  subspaces. And the idea is that will find the best in each  $S_i$ . So, by best we mean in the one in the element, which is which maximize as our objective function, which is which optimizes our object function.

So, in our case, it is going to be the longest subsequent belonging to each of these  $S_i$ 's. And then, we will return the best of the best. By this best I mean, the best computed above here. Simple enough, now the important point is going to be, how do we find the best in each  $S_i$ ? This is the recursion is going to come in. So, this if we can express recursively will at least have a recursive procedure. Of course, this is not your dynamic programming stops.

(Refer Slide Time: 14:13)

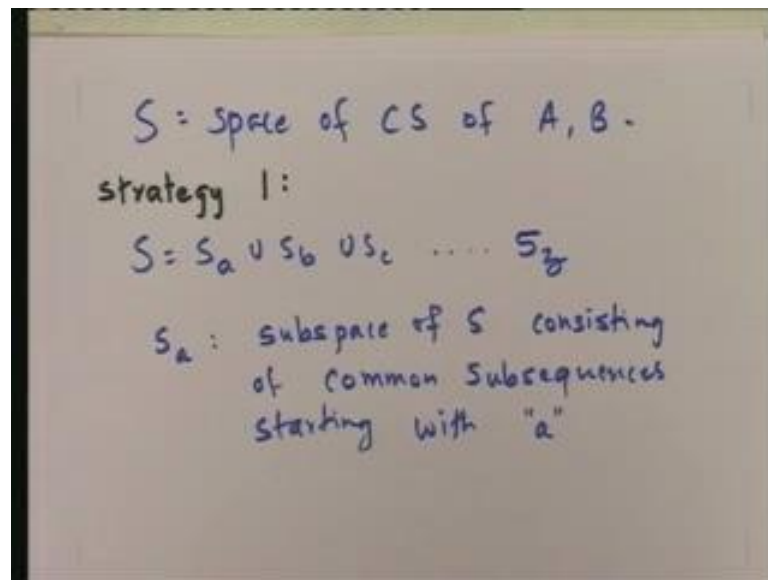


So, the next step is that we are going to characterize the calls the recursive calls. So, we are going to ask what are the arguments to the recursive calls? In fact, to all the recursive calls, which are made during the life time of that recursive procedure. So, we are going to define a table then. So, let me call it  $T$ , which will store the results evaluated for each recursive calls. Now, here comes an importance which, now we are going to ask that we should write non recursive procedure.

And this, I am going to call fill, which fills entries of  $T$ , assuming other entries are already filled. In particular, it fills say entry  $i, j$  of  $T$  assuming all other entries are filled. The key time is going to be that, the key point is going to be that as a result of this the total time will be small. In fact, right now I can list what the total time is going to be. So, total time is going to be size of the table times filling time. So, if we can do this with a small enough table.

And if we can show, that each entry requires will be a small amount of time to be filled. Then will have a fast algorithm. And your hope is that this strategy will give as a fast algorithm for the longest common subsequent problem. So, let us see, how to do this in the case of LCS. The first step is to define the search space, which we already did. The next step ((Refer Time: 17:04)) is to write this procedure and this requires just divide  $S$  into subspaces  $S_1$  to  $S_k$ . So, just do that one.

(Refer Slide Time: 17:17)

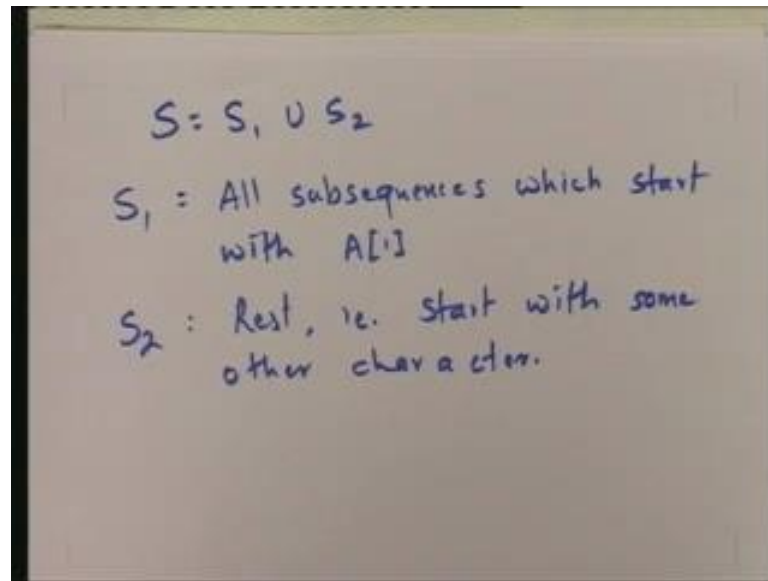


Remind you,  $S$  is the space of the common sequences of  $A$  and  $B$ . So, this is what we are going to search. And now, we go to partitioning. So, let me write down one strategy. There can be several strategies. And I am just going to write down 1. So, one idea might be, that I will define I will partition  $S$  as  $S_{\text{sub } a} \cup S_{\text{sub } b} \cup S_{\text{sub } c}$  all the way till  $S_{\text{sub } z}$ . The idea is that  $S_{\text{sub } a}$  is subspace of  $S$  consisting of common subsequences starting with letter  $a$ .

$S_{\text{sub } b}$  will be elements of  $S$ , which start to the letter  $B$ .  $S_{\text{sub } c}$  will be the elements of  $S$ , which start to the letter  $C$  and so on. So, here is one partition. Now, dynamic programming will require as the idea of dynamic programming will require as to find to write a recursive procedure, which recursively search as  $S_a, S_b, S_c$  and so on. And then, pick the best of the best of those answers. This is the 1 strategy. This will give us, better than exponential time solution. But, this is not the only possible strategy. So, let me give you another strategy as well.



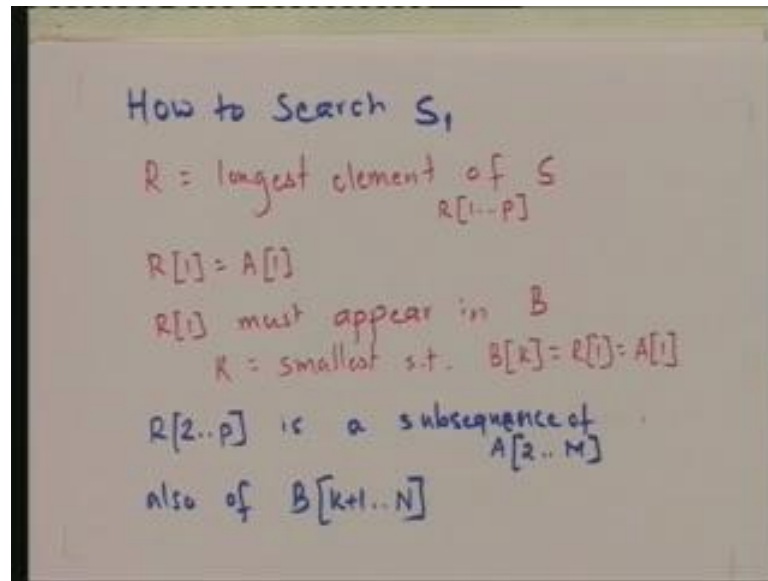
(Refer Slide Time: 19:13)



In this strategy, we are going to take  $S$  and we are going to partitioning it, only in two subspaces. So, let me tell you, what  $S_1$  is going to be. So,  $S_1$  is going to be all subsequences, which start with the first letter of  $A$ . So, remember  $A$  is already known to us.  $A$  and  $B$  are known to us. And so we can ask questions about what  $A[1]$  is. So,  $S_1$  is going to be all subsequences, which start with  $A[1]$ .  $S_2$  is the rest, that is start with something else, some other character.

Now, this is the strategy we are going to develop further. But will see that the idea of this development can also be used for the previous strategy. So now we need to define a recursive strategy, recursive procedure for searching  $S_1$ .

(Refer Slide Time: 20:21)



So, our question is how to search  $S_1$ . So, let us let me remind you, what is  $S_1$ ,  $S_1$  consists of all subsequences, which start with  $A_1$ . So, when we are looking for something it is usually a good idea to write down explicitly what is it, that we are looking for. So, let us say,  $R$  equal to longest element of  $S_1$ . So, let us say  $R$  as  $P$  as length  $P$  of course,  $P$  is not known to us, but let me just define notation  $P$ .

What is to be known about  $R$ . Well we know that  $R[1]$  the first letter of  $R$  is the same as the first letter of  $A$ . Not comes from the definition of  $S_1$ . We just said that,  $S_1$  only consists of subsequences, which start with  $A_1$ . Now,  $R$  is a subsequence of  $A$  as well as a subsequence of  $P$ . So, which means that  $R[1]$  must appear in  $B$  as well. So, without loss of generality we can assume. Let us, say that  $k$  is the smallest, such that  $B[k]$  is equal to  $R[1]$  or actually it is equal to  $A[1]$ .

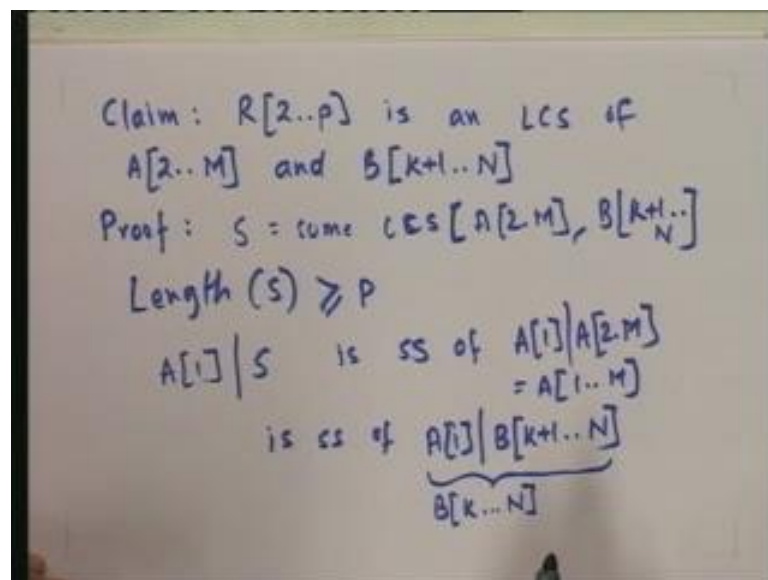
Now, it may turn out, that even does not at all appear in  $B$ . That is possible, in that case this search this space  $S_1$  will be null. So, that is going to be a better of a degenerate case. So, for now, for a minute I am going to assume that this does not happen. And that  $k$  is well defined. So, there is some  $k$ . So, now once we know what this  $k$  is, what can we say about, can we give something more, can we say something more about  $R[1]$ . Well, here is what we can say.

So, we know now that  $R[2..P]$  is a subsequence of  $A$ . Well not of  $A$ , because  $A[1]$  is already taken up by  $R[1]$ . So, the rest of  $A$ . So,  $2$  to  $M$  and also of  $B$  from  $k+1$  to  $N$ .

onwards. So, it is going to be B of k plus 1 to N. So, this just comes from the definition of R. So, R has to be a common subsequence of A as well as B. The first element of R appears at A 1 by definition of S 1. It appears at k, at B k in B. And the rest of it also must be a subsequence. Otherwise, it would not be a common subsequence.

And therefore, R must be a subsequence of this as well as the subsequence of this. So, now having characterized what R is our question is how do we find R. Usually, turns out that if you characterize something adequately it helps some finding. And so, in fact, we are almost ready to find. And in fact, we can claim that since R has to be a subsequence of both perhaps it is a longest such subsequence. So, let me write it down as a formal claim.

(Refer Slide Time: 24:13)

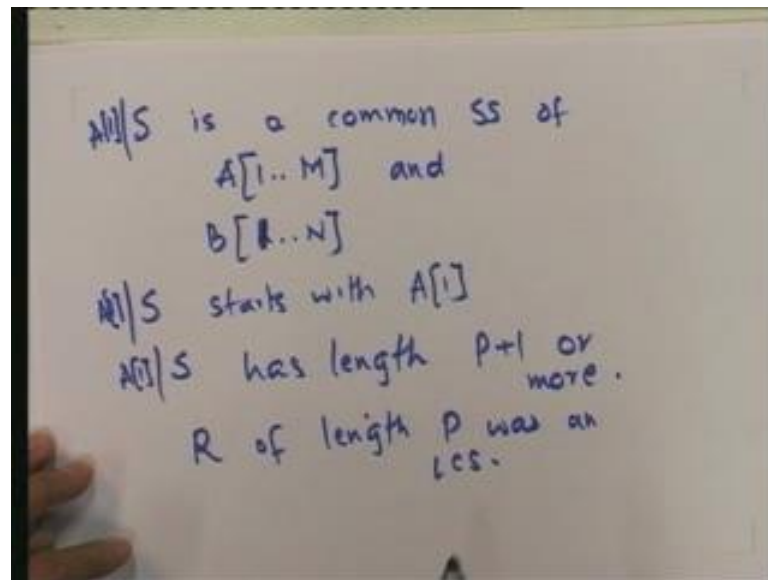


So, my claim is R 2 through P is an LCS of A 2 through M and B k plus 1 through N. Where, k is as we defined a minute ago. k is the index of the first occurrence of A 1 inside P. Turns out that this claim is true. And the proof is not terribly hard either the proof is by contradiction. So, let us assume that R 2 P is not an LCS of this. So, let us say, there is some S is some LCS of A 2 to M and B k plus 1 to N. And in fact, for the sake of contradiction we have to argue, we have to assume that S is longer than R 2 to P.

So, that is length of S is greater than the length of this, the length of this is P minus 1. So, let us say, it is greater than R equal to P. Now, the interesting thing is that suppose I consider the sequence A 1 concatenated with S. Can we say anything interesting about this? Well, S is a subsequence of A to M. So, this must be a subsequence of A 1

concatenated with  $A_2$  to  $M$ . But, this is simply  $A_1$  to  $M$ . Similarly, this is also a subsequence of  $A_1$  concatenated with  $B_{k+1}$  to  $N$ . But what is this? Well, this is nothing but even is also  $B_k$ . And therefore, this is  $B_k$  forward by  $B_{k+1}$  to  $N$ . And therefore, this whole thing is in fact,  $B_k$  to  $N$ . So, what have we found then?

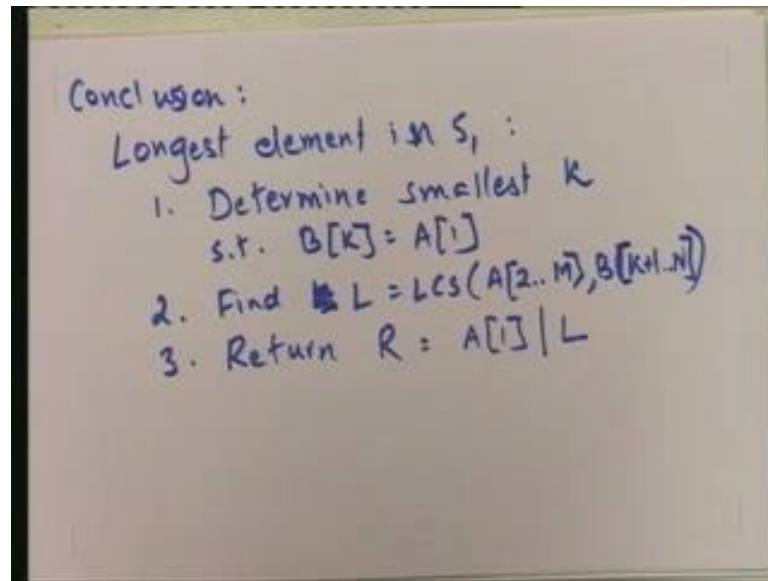
(Refer Slide Time: 27:07)



So, we have found, we have discovered that this  $S$  is a common subsequence of  $A_1$  to  $M$ . And  $B_k$  to  $N$ . But, this is itself is a subsequence of  $B_1$  to  $N$ . So, I could just the right this as  $1$  to  $N$ . And also,  $S$  starts with let me put it here.  $A$  starts with  $A_1$ . Now, what is the length of  $S$ .  $S$  has length at least  $P$ . So, this really up to be  $A_1$  concatenated with  $S$  is a common subsequence of this. And in fact,  $A_1$  concatenate with  $S$  starts with  $A_1$ . So,  $A_1$  concatenated with  $S$  has length at least  $P$  plus  $1$ .

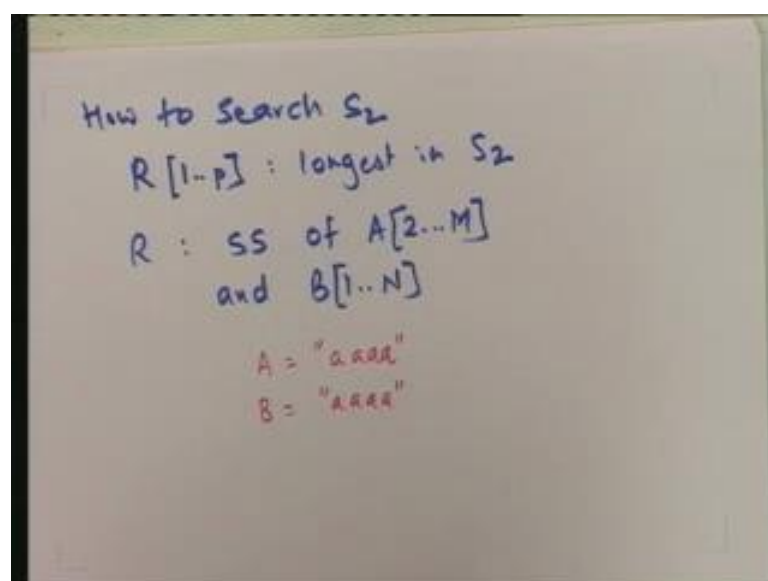
But now, we have a contradiction. Because, we said that  $R$  of length  $P$  was an LCS. So, then that means, there cannot be longer sequence, which is common to both. But, which we have just proved has to be there. And therefore, our original claim must have been true. So, what was our original claim, that  $R \geq P$  is an LCS of  $A$  to  $M$  and  $B_{k+1}$  to  $N$ . So, what is the conclusion from this? So, let me write it that down. So, the conclusion is that we know now, how to search  $S_1$ .

(Refer Slide Time: 29:09)



So, longest element in  $S_1$  can be found as follows. So, 1 determine smallest  $k$  such that  $B$  of  $k$  equal to  $A$  of 1. Second find capital  $L$  equals  $\text{LCS}$  of  $A$  of 2 to  $M$ ,  $B$  of  $K$  plus 1 to  $N$ . Why did we do this well this? Well this ((Refer Time: 30:04)) immediately from our claim. So, ((Refer Time: 30:09))  $R \in P$  we claimed is an  $\text{LCS}$  of this. So, we do not know what this is. So, we just find it. So, we just use recursion for it. And finally, return  $R$  equal to  $A$  of 1 concatenated with this  $L$ . So, we claim that this we showed, that this has to be along a common sequence of both  $A$  and  $B$ . Such that it lies in  $S_1$ , and so that is what we have constructed. So, we have constructed a procedure for searching  $S_1$ .

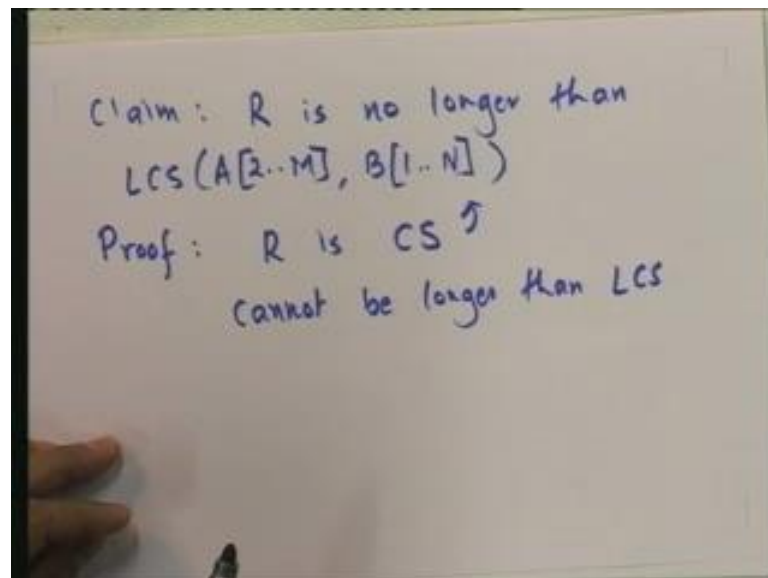
(Refer Slide Time: 30:53)



So, the next thing is to find a procedure for searching  $S_2$ . Well, as before let us write down what we are searching for. So, let us say, we are searching for some  $R$  again. So, say again of lengths  $P$ , which is longest on  $S_2$ . Can we characterize it further. Well, we know that  $R$  is a subsequence of  $A_2$  to  $M$ . Why  $2$  to  $m$ . Because we know that  $A_1$  does not appear in it. So, there is no question of  $R$  being a subsequence of a whole thing. So, it has to be  $2$  to  $M$ . And  $R$  is also subsequence of  $B_1$  to  $N$ .

So, note that over here, we have used a defining property of  $S_2$ . So, this is how you have characterized  $R$ . So, at this point it might be tempting to claim or at least conjecture that perhaps  $R$  should be the longest common subsequence of both of these. That is the nice conjecture. But, unfortunately it is not true. I am not going to explain it into much detail. But, let me just give you, a hint of why it is not true and then you can prove it for yourself. So, just consider  $A$  as a sequence of letters  $a a a a$ . And  $B$  say as also the same. Then, work out what  $X_1$  and  $X_2$  are going to be, and then you see why this is not true. So,  $R$  is not the longest common subsequence of both of these. However, something equally useful is true. So, let me write that down. So, what is true is the following.

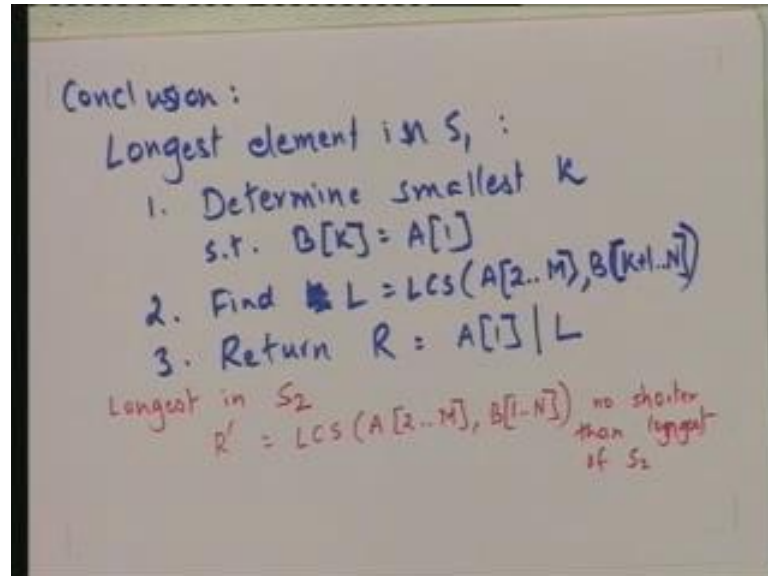
(Refer Slide Time: 32:54)



So, I claim that  $R$  as defined here is no longer than LCS of  $A_2$  through  $M$  and  $B_1$  through  $N$ . I will tell you, why this is useful in a minute. Let me, prove it first. The proof is actually a single line. We know that  $R$  is a common sequence of both of these. So well, so it cannot be longer than LCS of these two sequences, so done. So, let me now

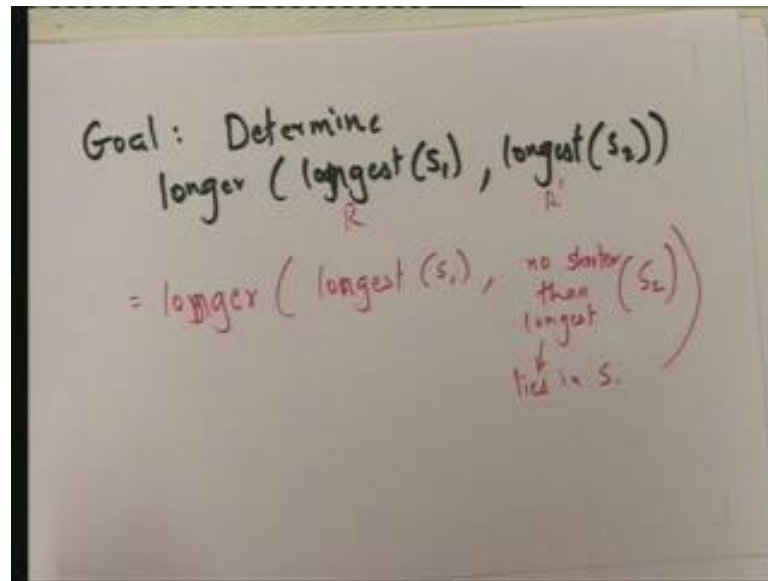
compare to this conclusion that we have written earlier. So, we said that we want to here is a procedure for looking for the longest element in  $S_1$ .

(Refer Slide Time: 33:56)



So, we also want to procedure for looking for the longest element in  $S_2$ . So, let me write that down over here. So, what will that procedure look like? So, I claim, that I am going to define  $R$  prime equal to LCS of  $A$  of 2 through  $M$ ,  $B$  of 1 through  $N$ . So, this will be generated by recursive call. And this is no smaller, no shorter than longest of  $S_2$ . So, what we have, we have a procedure to determine the longest element and  $S_1$ , and an element, which is no shorter than the longest of  $S_2$ . So, is this useful, so what was our goal. So, let us go back to our goal.

(Refer Slide Time: 35:04)

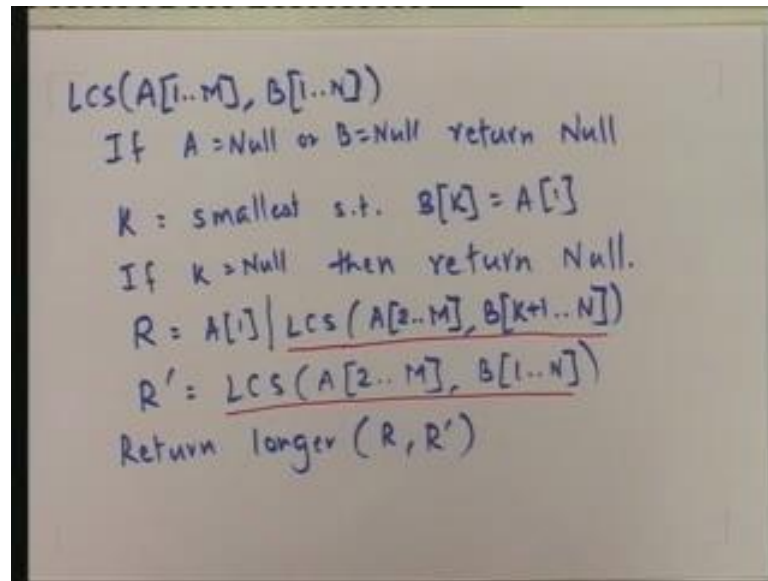


Our goal was to determine the longest of the longest the longest of  $S_1$  and the longest of  $S_2$ . And of course, so long as and we would happy. So, long as it is longer itself in  $S$ . So, this we know, this is our, this is what we just said or and this is  $R$ . What if I put  $R$  prime over here. So, this is no shorter. And so if I take the larger of this it will certainly be no shorter than all of this. And it will also lie in our original space. And therefore, this is also, so the longest of  $S$  and so if I substitute  $R$  prime over here. That is fine.

And therefore, this is nothing but longer of the longest of  $S_1$ . And no shorter than longest of  $S_2$  provided this also lies in  $S$ . So, if I do this. I will still be happy. And therefore, this conclusion that we have written ((Refer Time: 36:27)) is good in the sense that, if I compute  $R$  prime in this way. I just have to return the longer of  $R$  prime and  $R$ , and that will give may the best answer for the original space itself. So, now we are ready to write down our general search procedure. So, let me write that down.



(Refer Slide Time: 37:08)



```
LCS(A[1..M], B[1..N])
If A = Null or B = Null return Null
K = smallest s.t. B[K] = A[1]
If K = Null then return Null.
R = A[1] | LCS(A[2..M], B[K+1..N])
R' = LCS(A[2..M], B[1..N])
Return longer(R, R')
```

So, recursive procedure, which I am going to call LCS. It will take as arguments A of 1 through M, and B of 1 through N. And now, since we are going to write it as a procedure, let us be careful about boundary conditions. So, if A equal to null or B equal to null. That is as equal to saying, say these are strings of 0 length. So, say first index is bigger than the second index return null. So, clearly the longest common sequence of 2 null sequence or 1 null and something else is null sequence.

Now, now we have to worry about the main case. So, let us start by doing that. So, we are first going to see how S 1 is going to be searched. So, k is equal to smallest. Such that B of k is equal to A of 1. Remember, we are searching S 1 and that is what we did. If k equal to null as here or k is undefined that is my short form for that is k equal to null is a short for saying that. Then, again will return null. Otherwise this will ((Refer Time: 38:37)). So, we are going to do all this.

So, we are going to write or I will just write it jointly. So, we are going to compute R equals all of this. So, let me write that down. Otherwise, will compute R equals A of 1 concatenated with LCS of A of 2 through M. And B of k plus 1 through N, then they will compute R prime. Now, this time we are searching S 2. So, that as LCS of A of 2 through M and B of 1 through N. And then we will return the longer of R and R prime. So, we will get R and R prime explicitly will compute their lengths.

And will return the longer of the two. Now, I should point out some technicalities. So, in this call we wrote down A of 2 through M. The moment we write down A of 2 through M, we are implicitly assuming that M is bigger than or equal to 2, which may not be true M might be equal to 1. In this case, I am going to assume that an expression like this will be evaluated as null. So, instead of passing, so this expression will be passed as null if M is in fact, smaller than 2.

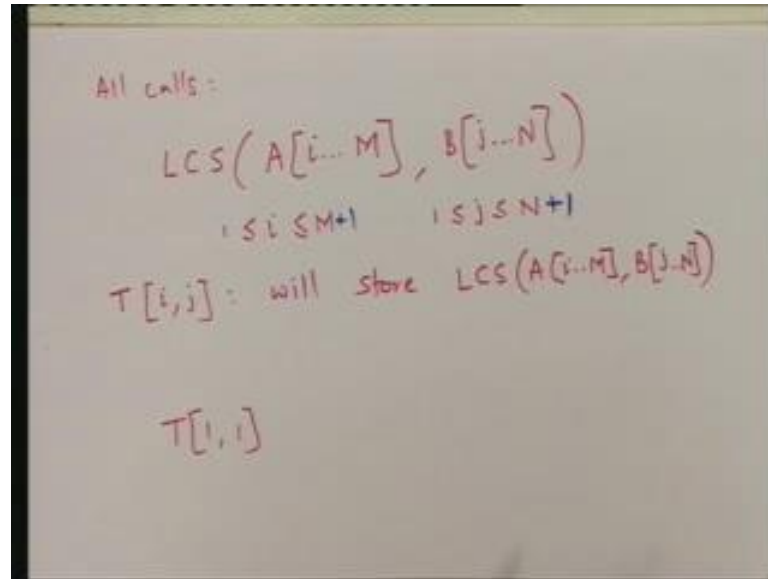
So, what will happen then, when this comes over here in the recursive call and null will be return. But, which is exactly what we want over here. And so therefore, this interpretation is fine. So, I do not have to write that is special case, whether M is greater than or equal to 2. It is just handle by saying that, this just represents null strings or null sequence. So, this is our basic recursive procedure. Now, one possibility might be to write down the recurrence to estimate the time taken for this.

And I will let you do that. And you will see, that the time taken is actually quite large. It will be exponential. However, that is not where we stop dynamic programming require first go further. And that is that next step is in fact, the one which is going to improve the time for us. So, what this dynamic programming require. So, dynamic programming as such to characterize what recursive calls happen in this. So, let us look at that. So, first the recursive call is this.

After that, there are two recursive calls. So, there is a recursive call over here. And then there is a recursive call over here. So, is there any nice property, which we know for this recursive calls. Well, if I look at A the argument pass for first string it is always going to be some part of A. And in fact, the second index is always going to be M. Similarly, if I look at the second argument to the recursive call it is always going to be B. The second argument is always going to be N.

The terminal index in that array is always going to be N. In the first index could be anything, it could be k plus 1. And if you recurs again, this property is going to stay they say.... So, we are, so when we recurs the key idea is that when we recurs our calls are going to be something like this.

(Refer Slide Time: 42:39)

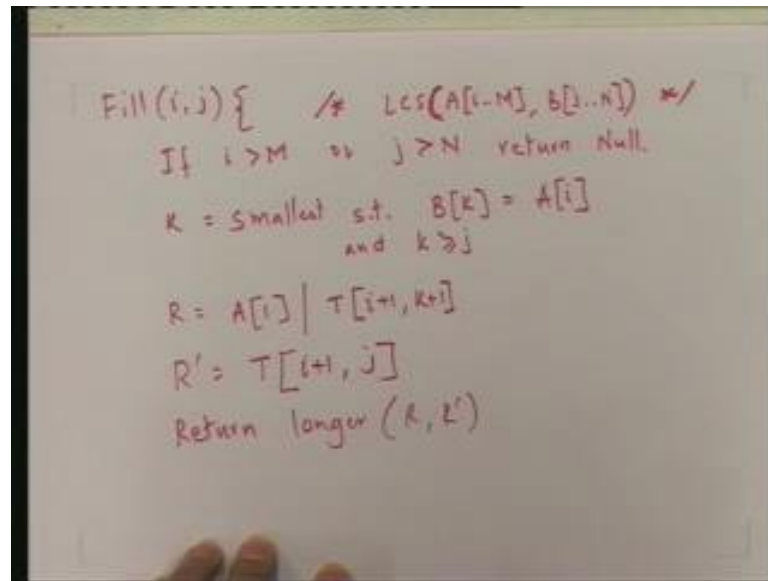


So, all calls, which are going to be made are going to have the form LCS of A of  $i$  to  $M$ , B of  $j$  to  $N$ . Where,  $i$  and  $j$  could be anything. So, this is an important observation. And this is an important part of the whole dynamic programming strategy. So, once we know that all calls are of these kinds. So, let me actually note that down (Refer Time: 43:10)) going to be a this kind. So, let me write this term say, where  $1$  is less than or equal to  $i$  is less than or equal to  $M$ . And  $1$  is less than or equal to  $j$  is less than or equal to  $N$ .

Dynamic programming says let us define a table let us called  $T$ . Such that its entry will save the solution answers to these calls. So,  $T[i, j]$  will store LCS of A  $i$  to  $M$ , B  $j$  to  $N$ . So, actually I am going to use not  $i$  ranging over just  $1$  through  $M$ , but over  $1$  through  $M$  plus  $1$ . And  $j$  will arrange over  $1$  through  $N$  plus  $1$ . And the reason for that will become clear in a minute. It will serve as a certain kind of ((Refer Time: 44:18)). So, this will be a two dimensional table with  $M$  plus  $1$  rows, and  $M$  plus  $1$  columns.

And now our goal is simply to fill in the entries of this table. So, once we fill the entries of the table  $T$  of  $1, 1$  will be LCS of A  $1$  through  $M$  and B  $1$  through  $M$ . And that is the solution we are looking for. So, if we can generate the procedure for filling these entries. And it has to be a non recursive procedure. Then we will be done. So, long as we say, in which order the entries are going to be filled. So, I claim that from this procedure of ours, there is a fairly nice way of filling in the entries. So, why is that? So, let me define fill  $i, j$  as follows.

(Refer Slide Time: 45:20)



So, what is will i j going to do. It is going to fill the entry  $T_{ij}$ . But, what is  $T_{ij}$ .  $T_{ij}$  is going to be the LCS let me write that term. So, it is going to be LCS of  $A_i$  through  $M$ ,  $B_j$  through  $N$ . So, let me comment it. So, this is what the entries suppose to be. The fill is suppose to written this entry. Now, if we look at our original procedure then this is after all a string and this is another string. And we could just as will pass this string or these two subsequences. So, all I have to do is to mimic what happens in this procedure.

And see, how fill should really behave. So, the first step is, if  $A$  is equal to null or  $B$  is equal to null, then return null. So, in this case, what does being null mean. So, being null is simply if you want to saying, when does this expression not having any meaning at all. So, I will interpret that as saying if  $i$  is greater than  $n$  or  $j$  is greater than  $n$ , then it does not make sense. And then so we will return null. So, that is that take care of that. In the next step is I have to find out, what this smallest  $k$  is such that  $B_k$  equal to  $A_i$ .

Now, here the string was starting from the first index  $A$ . Here, we are starting from  $i$ . So, naturally instead of 1 over here, I should be looking at the  $i$ th element over here, because this is nothing but a starting element of this string. And therefore, this should be the element that I look at. So, our corresponding step over here is going to be  $k$  equals smallest, such that  $B$  of  $k$  equals  $A$  of  $i$ . And of course,  $k$  is greater than or equal to  $j$ .

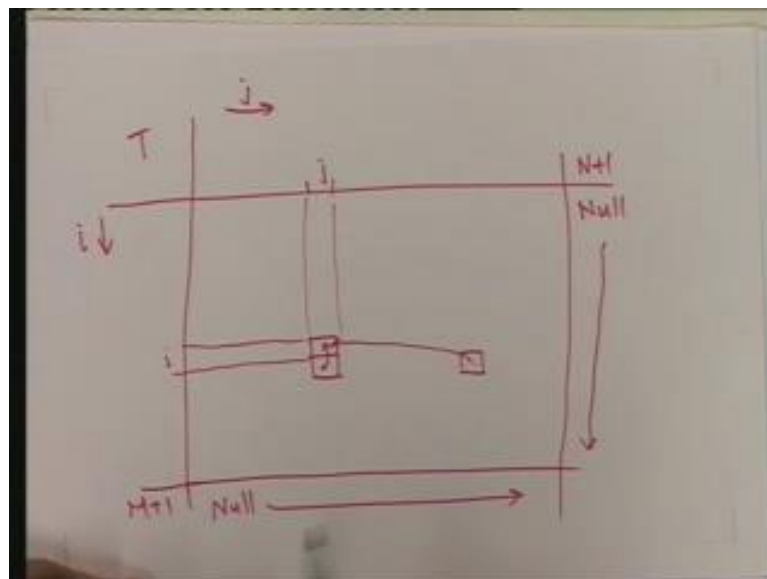
Why this because we want to search to happen from starting from  $j$  over here. Within this string we do not want this search to go before  $B$ . The next step over here ((Refer Time:

48:03)) was if  $k$  equal to null then return null. So, we could put that in. But, we do not as a terms out we do not need to. So, the next step is going to be  $R$  is equal to  $A$  of 1 concatenated with this LCS. But, remember that we do not want to recursive calls. So, we cannot write filled of all of this.

However, we are allow to assume that the table is already full. So, in which case what can we do. We can write over here  $T$  of  $A$  of 2 through  $M$ . But, ((Refer Time: 48:59))  $A$  of 2 through  $M$  has to be interpreted in light of this  $A$  of 1 through  $M$ , which in our cases  $i$  through  $M$ . So, this 2 through  $M$  really is nothing but  $i$  plus 1 through  $M$ . So. In fact, we will be writing instead of writing  $A$  of 2 through  $M$  we will be writing  $T$  of  $i$  plus 1 comma  $k$  plus 1.

The next thing is  $R$  prime and again over here instead of LCS of 2 through  $M$ . Since, we started with  $A$  which is going from  $i$  through  $M$  this will be  $i$  plus 1. And this is not going to be 1 through  $N$ , but this will be  $j$  through  $N$ . So, this will mean that  $R$  prime will be slightly different. So,  $R$  prime will have to be table entry not LCS again, but the table entry. So,  $i$  plus 1 comma  $j$ , so this is what it was. And now we are going to return the longer of  $R$  and  $R$  prime. So, now here essentially done? So, we have return the fill procedure. So, let us see, how the fill procedure is working out.

(Refer Slide Time: 50:36)



So, let us look at our table. So, this is our table. So, in our table, say  $I$  is going in this direction,  $j$  is going in this direction. In the first step is, ((Refer Time: 50:48)) if  $I$  greater

than  $M$  or  $j$  greater than  $N$  return null. So, this step is essentially says, that if this is the  $M$  plus 1th entry. So, this part of the table and this part of the table, so this is the  $N$  plus 1 entry over here or going to be all nulls, in this entire thing. Now, if we look at  $i, j$ , how to fill  $i, j$  it depends on  $i$  plus 1  $k$  plus 1 and  $i$  plus 1 and  $j$ , these are the two table and ((Refer Time: 51:20)) depends upon.

So, let me write that out. So, if I want to fill, so this is some index  $i$  and if I want to fill this. So, this is this  $j$  over here. Then this entry is going to depend upon  $i$  plus 1  $j$ . So, it is going to depend upon this entry. So, let me write that is an arrow over here. And it will also depend upon  $i$  plus 1  $k$  plus 1. Just going back to this ((Refer Time: 51:46)) it depends upon  $i$  plus 1  $k$  plus 1 as well. So,  $i$  plus 1  $k$  plus 1 could be something in this. So, it could be some entry of over here. So, it will also depend upon this.

So, if I want to fill this entry. I need to have these entries already filled. But, that is very easy. How do I do it. Well, if I start filling the entries from the bottom. Then I am essentially, then I will ensure that when it comes time to fill these entries, these entries are already filled. And then the fill procedure will work correctly. So, that is all. So, that is what I need to write down next. So, that is very simply done. So, I just want to tell you. what the code is going to look like. So, first I want to make sure that these two things these two are properly set.

(Refer Slide Time: 52:43)

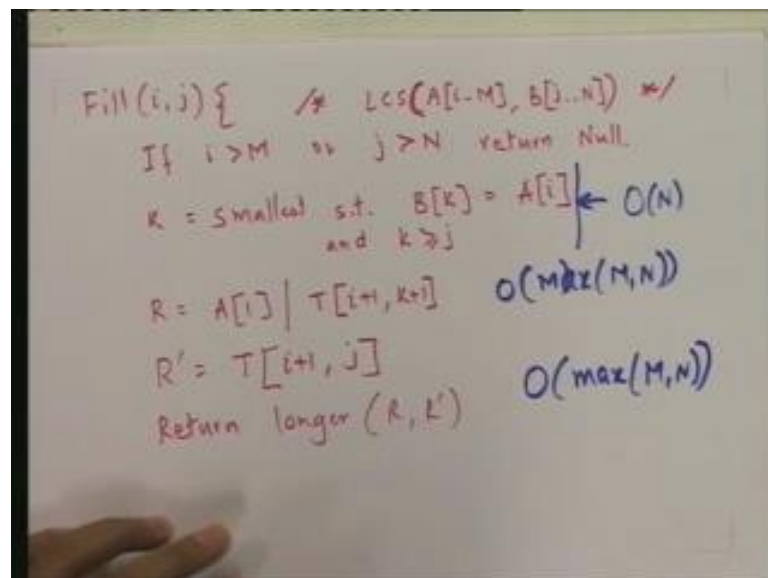
```
For i = 1 to M+1  
    T[i, N+1] = null  
For j = 1 to M+1  
    T[M+1, j] = null  
For i = M to 1  
    for j = N to 1  
        T[i, j] = Fill(i, j)  
Return T[1, 1]
```

$O(M)$   
 $O(N)$   
 $O(MN)$

So, that will be done by saying for  $i$  equal to 1 to  $M$  plus 1. What happens,  $T$  of  $i$  comma  $N$  plus 1 is equal to null. And for  $j$  equal to 1 to  $M$  plus 1.  $T$  of  $M$  plus 1 comma  $j$  is equal to null. That just set the bottom row and the right hand side column to null. Now, comes the main point. So, for  $i$  equal to  $M$  to 1, for  $j$  equals  $N$  to 1 will just fill  $T$   $i$   $j$  is equal to fill of  $i$   $j$ , and because our rows are going  $M$  to 1 backwards. Will make sure that fill of  $i$   $j$  will find the table entries already filled properly. So, that takes care of the whole thing.

So, at the end of this is simply return  $T$  of 1, 1 that is going to be your final answer. So, what remains now, is to estimate the time taken. So, this time is going to be simply  $O$  of  $M$ . This is going to be  $O$  of  $N$ . And this is going to be  $O$  of  $M$   $N$ . Because, of these loops over here and the time taken for fill  $i$   $j$ .

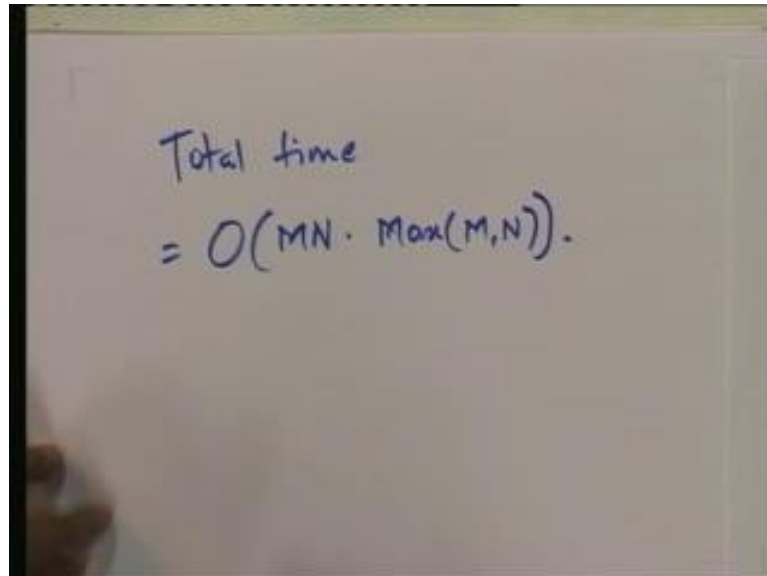
(Refer Slide Time: 54:18)



So, how long this fill  $i$   $j$  take. Well the only place, where time is taken there are two places where time is taken. So, one is this. So, this could take time proportional to the length of  $B$ . So, this could take time proportional to  $N$ . What about this. So, we are going to do the concatenation. Now, if you store the subsequences, which have been computed as arrays, then this could take time  $O$  of  $M$ . Because, we are going to take the subsequence and we have to concatenated. Or in fact, it would take time, it could be that the subsequences the entire things. So, it could be  $\max$  of  $M$  and  $N$ . And maybe the longer computing the longer will also take time this much. So, over all the time taken for this is going to be  $O$  of  $\max$  of  $M$   $N$ . So, coming back this last part fill is going to take

time  $O$  of  $((\text{Refer Time: 55:24})) \max$  of  $M N$ . And so the total time is going to be the product of these two things.

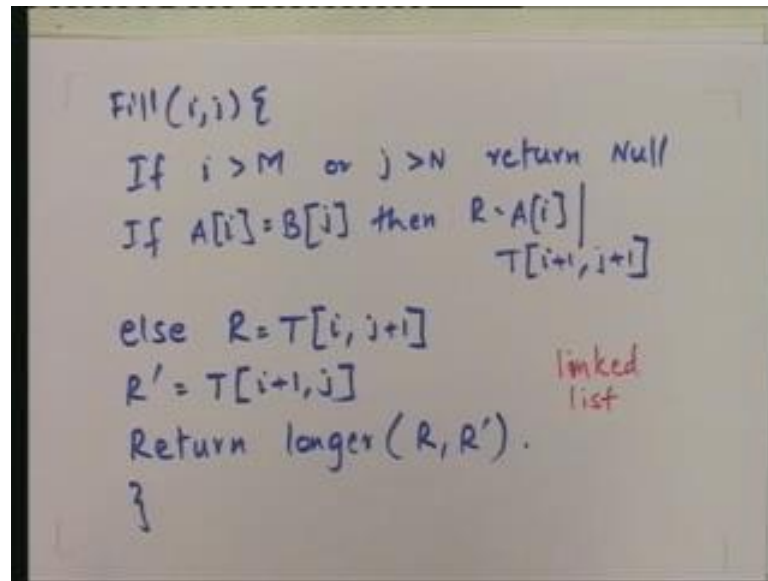
(Refer Slide Time: 55:39)

A photograph of a whiteboard with handwritten text in blue ink. The text reads: "Total time" followed by an equals sign and the expression "O(MN \* Max(M, N)).".
$$\text{Total time} = O(MN \cdot \max(M, N)).$$

Or in other words, total time is equal to  $O$  of  $M N$  times  $\max$  of  $M N$ . Now, let me just to summarize, let me just say that I will give you dynamic programming algorithm, which runs in so much time. There is a simple modification that you can make to this, using which we can reduce the times further. So, it could be it can be reduce to time  $O$  of  $M N$ . So, I am just going to write down the code for fill and we will not prove its correctness. But, the correctness will be included in the manner analogous to it will be only slightly more complicated. And so I will just do that and I will stop this lecture.



(Refer Slide Time: 56:24)



```
Fill(i, j) {  
    If  $i > M$  or  $j > N$  return Null  
    If  $A[i] = B[j]$  then  $R = A[i] |$   
                         $T[i+1, j+1]$   
  
    else  $R = T[i, j+1]$   
          $R' = T[i+1, j]$  linked  
         list  
    Return longer( $R, R'$ ).  
}
```

So, this is the code, which will give you faster result. It will require that you store the result being calculated. The table entry should be stored as a linked list. Rather than storing the entire subsequence in each entry in the table, store it as a linked list. If you do that, with the little slight trick you should be able to reduce the time taken just down to  $O$  of  $M N$  ((Refer Time: 57:58)). So, with that I will stop this lecture.

Thank you.