

Note: the slides were originally produced by Dr. Vigoda; Dr Gerandy Brito has been the professor for the OMSCS course since 2020.

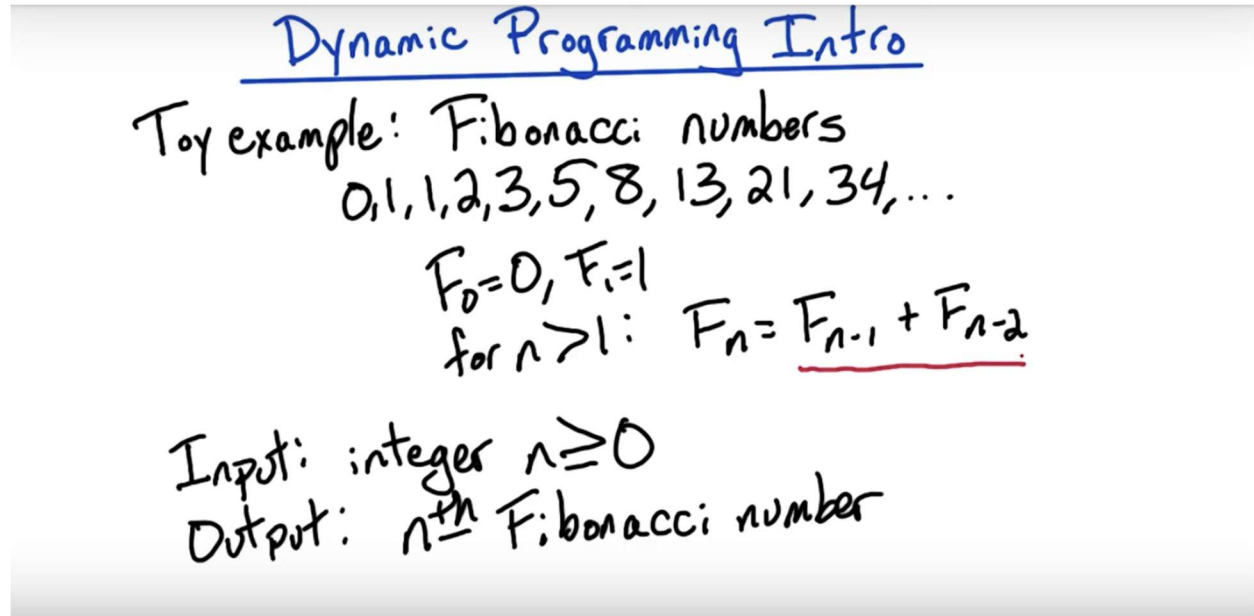
Lesson: Introduction to Graduate Algorithms (Slide 1) Overview

Welcome to Graduate Algorithm's. I'm Eric Vigoda This is a theory course and as such, it is heavily mathematical in nature. Let's take a look at the course outline. We'll start with Dynamic Programming, then we'll take a quick look at randomized algorithms, including cryptography and hashing. Next, we'll look at some more sophisticated divide and conquer examples, including the beautiful FFT algorithm. We'll dive into graph algorithms, using DFS for connectivity problems on directed graphs. And we'll look at a few applications, including the two-step problem, and in some sense, the page rank algorithm. Page Rank is the algorithm at the heart of Google's search engine. Next, we'll continue with Graph Algorithms, looking at algorithms for the max flow problem. And then, we'll look at linear programming, which is a powerful and widely used tool for optimization problems. And finally in the last section of the course, we'll dive into computational complexity. We'll explore the theory of NP-Completeness. Let's jump into the course. I hope you enjoy it.

Lesson: DP1: FIB – LIS - LCS

(Slide 1) Dynamic Programming Overview

Our first topic is dynamic programming. This is one of my favorite topics to teach because it is extremely useful. Students often have some trouble with it initially, but with enough practice it will sink in and actually, they will seem quite easy once you get the hang of it. I'll do my best to show you how I solve dynamic programming problems. And, from there the key to really mastering dynamic programming is to do lots and lots of practice problems. The homework is just a start. Do additional problems from the textbook, and from other algorithms textbooks, and from other courses you can find on the web. We'll start with the toy example, computing Fibonacci numbers, to illustrate the basic idea of dynamic programming. Then, we'll dive into a variety of example problems to get a feel for the different styles of DP algorithms;- Longest increasing subsequence (LIS), longest common subsequence (LCS) the classic Knapsack Problem, chain matrix multiplication and finally, we'll look at a few shortest path algorithms using DP.



The slide features handwritten text in blue ink on a light gray background. At the top, the title "Dynamic Programming Intro" is underlined. Below it, the text "Toy example: Fibonacci numbers" is followed by the sequence "0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...". The initial conditions are given as $F_0 = 0, F_1 = 1$. The recursive formula is $\text{for } n > 1: F_n = \underline{F_{n-1} + F_{n-2}}$, with the sum underlined. At the bottom, the input is "integer $n \geq 0$ " and the output is " n^{th} Fibonacci number".

Dynamic Programming Intro

Toy example: Fibonacci numbers
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

$F_0 = 0, F_1 = 1$
for $n > 1$: $F_n = \underline{F_{n-1} + F_{n-2}}$

Input: integer $n \geq 0$
Output: n^{th} Fibonacci number

Given an integer n , we're going to look at an algorithm for generating the n Fibonacci number. This will be a very simple algorithm, but it will illustrate the idea of dynamic programming and then, we'll look at dynamic programming in general; the techniques for designing a dynamic programming algorithm and we'll look at some more sophisticated examples. Recall the Fibonacci numbers are the following sequence, 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 and so on. There's a simple recursive formula that defines the Fibonacci numbers. The first two numbers in the sequence are zero and one, and then the n Fibonacci number is the sum of the previous two Fibonacci numbers. We're going to take as input a non-negative integer n and our goal is to output the n th Fibonacci number. We want an efficient algorithm and therefore we're aiming for a running time which is polynomial in n . Now, the Fibonacci numbers are defined by this simple recursive formula. Therefore, we might think a recursive algorithm is a natural algorithm for this problem. Let's look at that natural algorithm, that natural recursive algorithm and when then we'll analyze it.

Fibonacci #'s: Recursive Alg.

$$\text{for } n > 1: \underline{F_n = F_{n-1} + F_{n-2}}$$

Fib1(n):

input: integer $n \geq 0$

output: F_n

if $n=0$, return(0)

if $n=1$, return(1)

Return($\underbrace{Fib1(n-1)}_{T(n-1)} + \underbrace{Fib1(n-2)}_{T(n-2)}$)

Let $T(n) = \# \text{ steps for } Fib1(n)$

$$\underline{T(n) \leq O(1) + T(n-1) + T(n-2)}$$

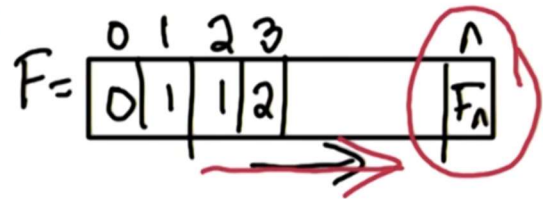
$$T(n) \geq F_n \approx \frac{\phi^n}{\sqrt{5}}$$

where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$
"golden ratio"

Look at the natural recursive algorithm for computing the Nth Fibonacci number. Recall from the previous slide, the recursive formula for the Nth Fibonacci number is the sum of the previous two Fibonacci numbers. Let's detail the recursive algorithm for computing the Nth Fibonacci number. We'll call the algorithm Fib1, a little bit foreshadowing. This will be our first attempt then we'll have a second successful attempt for efficiently computing the Fibonacci numbers. Recall the input to our algorithm is a non-negative integer n , and the output is the Nth Fibonacci number or more precisely actually the n plus first Fibonacci number if you're keeping track of the indices. First, we'll deal with the base cases. The first two Fibonacci numbers are zero and one. In general, we'll utilize the recursive definition of the Fibonacci numbers. We will recursively compute the n minus first, and n minus second Fibonacci number. And we'll return the sum of those two. This completes the definition of our algorithm. Now, let's take a look at the running time of our algorithm. To analyze the running time of our algorithm, let's create a function T of n which denotes the number of steps of our algorithm and an input of size n . We have two basic cases first. Those each take order one time. And then, we have a two recursive calls, one of size n minus one, and one of size n minus two. Those take time, T of n minus one plus T of n minus two. Putting it all together, we have the following formula. Our running time on the input of size n is order one for the two base cases and then, T of n minus one in order to compute the n minus first Fibonacci number, and T of n minus two to compute the n minus second Fibonacci number. Now, this formula may look a little bit familiar. Notice, it reminds us of the Fibonacci numbers. Fibonacci numbers, the n

minus Nth Fibonacci number is the sum of the previous two. This is the sum of the previous two plus an extra term. So, it's at least as large as the Nth Fibonacci number. Therefore, our running time on input of size n is at least the Nth Fibonacci number. Now, unfortunately Fibonacci numbers grow exponentially and then more precisely. They grow exponentially in this constant ϕ . What is ϕ ? Well, ϕ is this constant defined as one plus square-root five over two which is roughly 1.618. But ϕ is a magical constant. It's known as the golden ratio. The golden ratio has many applications. For example, if you're designing a building such as the Parthenon or if you want just a cool fact to show your kids, then look up the golden rectangle. But for us, the punchline is that the Fibonacci numbers grow exponentially in n . And the running time to compute the Nth Fibonacci number using a recursive algorithm is at least the Nth Fibonacci number. So, our running time is exponential in n . So this is a terrible algorithm for computing the Nth Fibonacci number. Let's take a look at exactly why our running time is so terrible and then, we'll get an idea for a more efficient algorithm.

for $n > 1$: $F_n = F_{n-1} + F_{n-2}$

$$T(n-1) + T(n-2)$$


6 / 49

Dynamic Programming

```
Fib2(n):  
  { F[0] = 0  
    F[1] = 1  
    for i = 2 → n:  ← O(n)  
      F[i] = F[i-1] + F[i-2]  ← O(1)  
    Return(F[n])  
  }  
O(n) total time
```

The image shows a handwritten code snippet for a dynamic programming algorithm to compute the nth Fibonacci number. The code is enclosed in a box. It starts with 'Fib2(n):' followed by a block of code. The first two lines are 'F[0] = 0' and 'F[1] = 1'. Then there is a 'for' loop 'for i = 2 → n:' with a purple arrow pointing to it from the label 'O(n)'. Inside the loop is the assignment 'F[i] = F[i-1] + F[i-2]' with a purple arrow pointing to it from the label 'O(1)'. The loop is followed by 'Return(F[n])'. Below the box, the text 'O(n) total time' is written in purple.

Now, let's detail our dynamic programming algorithm for computing the nth Fibonacci number. Here's our second attempt at an algorithm for computing the nth Fibonacci number. Recall, we're going to create an order array F which is going to store the Fibonacci numbers and we're going to start at the small indices namely. At the first two indices, we store the two base cases, zero and one, which are the first two Fibonacci numbers in the sequence. Then we have a for loop which goes from two up to n which is our goal. And using the recursive formula, the i th Fibonacci number is the sum of the previous two which are already stored in our array. Finally, we return the nth Fibonacci number which is stored in the last index in our array. This completes our algorithm. Notice the key thing is that our algorithm has no recursion in it. We have a recursive formula which defines the Fibonacci numbers but our algorithm has no recursion in it. Finally, let's analyze the running time of our algorithm. Once again, the base cases take order one time each. Then we have a for loop of size order n . The one step of the for loop takes order one time. Therefore, the total time of the for loop is order n time and the total run time of our algorithm is order n time. This completes our algorithm and gives us a glimpse of a dynamic programming algorithm.

(Slide 6) FIB2: DP Recap

Before moving on to a more sophisticated example, let's recap a few key issues. I want to stress one important point about Dynamic Programming algorithms. Our algorithm had no recursion inside of it. We used the recursive nature of the problem to design our Dynamic Programming Algorithm, but the algorithm itself has no recursion inside of it. Now, there is an alternative approach to Dynamic Programming. In this approach, you use a hash table or some other similar structure to keep track of which sub-problems that have already been solved, so that you don't recompute those problems. Now, we're not going to use this at all in our course. This approach is called memoization. Try to say that five times, I can't and therefore we're not going to use it in this class. But actually, the real reason why we're not using it, is because the goal of this unit is to learn dynamic programming. Therefore, to avoid any confusion we're going to say no recursion in our algorithms. Dynamic programming has several advantages over memoization or similar techniques. Some might say the algorithms themselves are more beautiful. Certainly, they're faster because they have less overhead from avoiding recursion. And finally, it's much simpler to analyze the running time of Dynamic Programming Algorithms, dynamic programming is widely used. At first, students often find it challenging, but what we find is that with enough practice the dynamic programming algorithms start to seem similar to each other. At that point it will click and hopefully it'll seem easy to design a dynamic programming after that. To get to that point, what do you need to do? Practice, practice, practice. Do a lot of practice problems and then it will click and you'll find dynamic programming easy. We're going to do some in class, you're going to do something during the homework and then do a lot on your own. There's a lot of practice dynamic programming problems out there in the textbook and other online resources that you can find. You do enough practice, you ace the topic.

Longest Increasing Subsequences (LIS)

Input: n numbers a_1, a_2, \dots, a_n

Goal: find length of LIS in a_1, \dots, a_n

Example:

5	7	4	-3	9	1	10	4	5	8	9	3
---	---	---	----	---	---	----	---	---	---	---	---

substring = set of consecutive elements

subsequence = subset of elements in order (can skip)

LIS = -3, 1, 4, 5, 8, 9 len = 6

Let's look now at a more sophisticated example of dynamic programming. The problem we're going to consider is the longest increasing subsequence problem. And for simplicity, we'll call it LIS. The input to the problem are n numbers which we'll denote as, A_1, A_2 , up to A_n . Our goal is to compute the length of the longest increasing subsequence in the n numbers of the input. One important note, we're only trying to find the length of the longest increasing subsequence. We're not trying to find the subsequence itself. But if we can find the length, then we're going to get to the heart of the problem and then it will be easy to transform that into an algorithm to output an actual subsequence of longest length. Let's take a look at a specific example to help illustrate the terminology in this problem. Here's an example where n equals 12. Before defining subsequence, let's recall the definition of the more common term substring. A substring is a string that occurs inside the larger string. So it's a set of consecutive elements. For instance, here's one substring -3, 9, 1, 10. Another substring is 4 itself. And another substring is 9, 1, 10, 4, 5, 8, 9, 3. These are all substrings. How many substrings are there possible? We can specify a substring by the start index and the ending index. Therefore, there is a most order n -squared substrings. Now, our problem is not defined for substrings, it's defined for subsequences. A subsequence is a string you can obtain by deleting elements of the larger string. So it's a subset of elements in order but we can skip elements. It doesn't have to be consecutive elements. Let's look at some examples subsequences. For instance, 4, -3, 1, 9 is a subsequence. Another subsequence is 1 itself. Another subsequence is 5, 7, 3. Now, we're trying to find a subsequence which is increasing. That means that each number is strictly larger

than the previous. In this case, it's not increasing because 3 is smaller than 7 but an increasing subsequence for instance, is 4, 9, 10. Another subsequence is 4, 4, 8, 9. But that is not an increasing subsequence. That's a non-decreasing subsequence, and we don't allow that in our example. Our goal is to find the longest increasing subsequence in the input array. For this example, what is the longest increasing subsequence? The longest increasing subsequence in this example is -3, 1, 4, 5, 8, 9. So, the output of our algorithm on this instance is six. Now, let's try to design a dynamic programming algorithm for this longest increasing subsequence.

DP Solution

1st step: define subproblem in words

$F[i] = i^{\text{th}}$ Fibonacci number

2nd step: state recursive relation

express $F[i]$ in terms of $F[1], \dots, F[i-1]$

$$F[i] = F[i-1] + F[i-2]$$

Now let's look at the recipe for designing a dynamic programming algorithm. The first step is to define the sub-problem in words. To see what I mean by this, let's look back at the simple example of Fibonacci numbers. In that example, our sub-problem definition was $F[i]$ is the i th Fibonacci number. The second step in our recipe, is to state a recursive relation. We want to express the solution to the i sub-problem in terms of smaller sub-problems. So for the instance of Fibonacci numbers, we want to express $F[i]$ in terms of $F[1]$ through $F[i-1]$. Because if you recall our dynamic programming algorithm for computing the Fibonacci numbers, we computed $F[1]$ up to $F[i-1]$, so those will be stored in our table, and then we can use those to compute $F[i]$. Now for the case of the Fibonacci numbers, it was straightforward to express $F[i]$ in terms of smaller sub-problems. Namely $F[i]$ is the sum of the previous two Fibonacci numbers. Now, let's figure out how to follow this recipe for the longest increasing sub-sequence problem.

DP Solution

1st step: define subproblem in words

Let $L(i) = \text{length of LIS on } a_1, a_2, \dots, a_i$

2nd step: state recursive relation

express $L(i)$ in terms of $L(1), \dots, L(i-1)$

Now, let's follow the first step in our recipe. We want to define the sub problem in words. Our first attempt is always going to be, to use the identical problem on a prefix of the input. In this case, that means we're going to look at the longest increasing sub-sequence on the first i elements of the input array. Therefore, we're going to make a new function L , which is the length of the longest increasing sub-sequence on the first i elements of the input. Now, we want to figure out how to express $L(i)$ in terms of $L(1)$ through $L(i-1)$ smaller subproblems. To do this, let's take a look back at our earlier example, and see if we can gain some intuition.

(Slide 9) LIS: Recurrence Attempt 1

DP Solution

Let $L(i)$ = length of LIS on a_1, a_2, \dots, a_i

express $L(i)$ in terms of $L(1), \dots, L(i-1)$

$A =$	5	7	4	-3	9	1	10	4	5	8	9	3
$L =$	1	2	2	2	3	3	4	4	4	5		

$i=9$

$L(9)$ using $L(1), \dots, L(8)$

Need min ending character in LIS solution

$5, 7, 4, -3, 9, 1, 10, 4, 5, 8$

$5, 7, 9, 10$

$L(9) = 5$

$-3, 1, 4, 5, 8$

Recall, our subproblem definition is $L(i)$ is the length of the longest increasing subsequence on the first i elements of the input, all right? And our goal is to express $L(i)$ in terms of the solution to smaller subproblems. So, now, let's recall our earlier example. Here is our earlier example which had n equals 12. Now, let's look at L on this example. To start with, for $L(1)$, we're looking at the longest increasing subsequence on the one element array. That's length one or 5 itself. Then, for $L(2)$, we're looking at the two element array 5, 7. The longest increasing subsequence is 5, 7 itself, which is length two. And for "5, 7, 4", "5, 7" is still the longest and we add on -3, but still length two is the longest. For 9, we can append on to the longest solution. We get the length three with 1, stays 3. When we add on 10, we can append 10 on to the longest solution. We get like 4. Now let's pay attention to the case i equals 9. What is $L(9)$? Well, we cannot append 8 on to the end of this current longest solution but in fact, there's a different solution. -3, 1, 4, 5, for which we can append 8 on to the end and we get a solution of length 5. So, the problem is, how can we compute $L(9)$ using $L(1)$ through $L(8)$? How do we know whether we can append 8 on to the end of the current solution or not? We're not maintaining the current solution but even if we maintain the current solution, how do we know that we can append 8 on to the end of it? For this solution we couldn't maintain it. We couldn't append it on to the end. But for this solution we could append 8 on to the end of it. So, suppose we kept track of the current solution or actually, what do we need to know? We need to know the ending character of the current solution and what's the key fact? We want to know the longest increasing subsequence with the minimum ending character. So, we want to know the smallest ending character, because the smallest ending character gives us the most

possibilities for appending on a new character on to the end. So in this case, in order to compute $L(9)$ using $L(1)$ through $L(8)$, we need to keep track of the longest increasing subsequence solution with the minimum ending character. And in this case, it's five and then we realize we can append 8 on to the end of it and we can increase those solution length from four. Let's go back to i equals 8 and we'll start to see the complication in this solution. Previously, our solution was 5, 7, 9, -3, 1, 4, 5 because this is also of length 4 and it ends in a smaller character. For now let's go back to i equals 7. For i equals 7, the longest increasing subsequence is 5, 7, 9, 10. But, no. when i equals 8, we need to have this sequence -3, 1, 4. Now this is only length 3. So it's a suboptimal solution but we need to maintain it in order to realize that in the next step that we can append 5 on to the end of it and then we obtain a length 4 solution for i equals 8. So, how do we maintain this suboptimal solution? The key is, that for every possible ending character, we want to maintain the length of the longest increasing solution with that ending character.

DP Solution

Let $L(i)$ = length of LIS on a_1, a_2, \dots, a_i & includes a_i

express $L(i)$ in terms of $L(1), \dots, L(i-1)$

$A = [5, 7, 4, -3, 9, 1, 10, 4, 5, 8, 9, 3]$

$L = [1, 2, 2, 2, 3, 3, 4, 4, 4, 5, \quad, \quad]$

$\uparrow \uparrow i=9$

Need: length of LIS for every ending character

5, 7, 9, 10
-3, 1, 4, 5

We need to know the length of the longest increasing subsequence for every possible ending character. If we know that for every possible ending character, then when we get a new character, in this case 5, we can try all possible ending characters and we can try to add on 5. We can see whether it is allowed to add on five on to the end of it. Now, how many possible ending characters are there? What are the possible ending characters? Well, the ending characters must be an earlier element in the input array. So in this case it is 5, 7, 4, -3 up to 4. Those are the possible ending characters. So they're not limitless. There's a finite small number. Actually, $i-1$ possible ending characters. So, this actually gives us an idea for how to modify our

subproblem formulation. We want to know the length of the longest increasing subsequence for every possible ending character. We just noticed that the possible ending characters are the earlier elements in the array. Therefore, we want to maintain the length of the longest increasing subsequence for each element of the array. This gives us an idea for the new subproblem formulation. We're going to modify the length of the longest increasing subsequence on a_1 through a_i and includes a_i , okay? So this will give us the longest increasing subsequence, which ends at the i element of the array, in this case 5. And then, we have that for $L(1)$ through $L(i-1)$ and then we can use that to decide, what is the longest sequence ending at 8 because we can see which characters allow us to append 8 on to the end of it. Let's go back and formulate this more precisely and then we'll see the recurrence.

DP Solution

1st step: Define subproblem in words
Let $L(i)$ = length of LIS in a_1, \dots, a_i which includes a_i

2nd step: state recursive relation

$i=10$

A=	5	7	4	-3	9	1	10	4	5	8	9	3
L=	1	2	1	1	3	2	4	3	4	5		

5, 7, 4, -3, 9, 1, 10, 4, 5, 8

We now have a new subproblem formulation, let $L(i)$ denote the length of the longest increasing subsequence on the first i elements of the input array a_1 through a_i and which includes a_i itself. This is the extra restriction that we added into our subproblem formulation, okay? It requires that a_i is included in the subsequences that you consider. Now, it will be straightforward to express a recurrence, which formulates $L(i)$ in terms of $L(1)$ through $L(i-1)$. Let's go back and look at our earlier example and now we'll see the straightforward recurrence that arises. Here is our earlier example. And let's look at L for this new subproblem formulation:

- In this case, $L(1)=1$ because we're looking at the length of the longest increasing subsequence on an input of size one just 5 itself and it has to include 5.
- Then for $i=2$ with length two from the subsequence 5, 7.
- Now, we start to see a difference when $i=3$. What is the length of the longest increasing subsequence in 5, 7, 4 which includes 4? That's just 4 itself. So that's length of one, okay? Recall in our earlier definition of L , it was length, it was three. this entry.
- Once again, for $i=4$, $L(4)=1$ because of -3.
- So $i=5$, now, the length is three from 5, 7, 9.
- For $i=6$, the length is two from -3, 1.
- For $i=7$, the length is four from 5, 7, 9, 10.
- Finally, let's look at the case $i=10$, which is the case that caused us problems in the earlier definition of L . Now, in this case, we want to see which subproblems allow

us to append 8 on to the end of it. Now notice that 8 can be appended on to subsequences ending at 5, 7, 4, -3. But 9, it cannot get appended on to - that won't be increasing. It cannot get appended on to 10 either. But, the others allow it to be appended on to the end. So, we're going to take the longest of those and we're going to add +1 for adding 8 on to the end of it. So whatever is the longest increasing subsequence ending at 5, we can append 8 on to the end of it. We don't have to know the subsequence itself. We just have to know it's of length four and it ends at 5 and then we can append 8 on to the end. We obtain the sequence of length five, therefore, $L(10)$ is five. Now, this highlights the recurrence for the solution of $L(i)$, in terms of smaller subproblems $L(1)$ through $L(i-1)$.

DP Solution

1st step: define subproblem in words

Let $L(i)$ = length of LIS in a_1, \dots, a_i which includes a_i

2nd step: state recursive relation

$$L(i) = 1 + \max_j \{ L(j) : a_j < a_i \text{ \& } j < i \}$$

$$L(i) = 1 + \max_{\substack{1 \leq j \leq i-1 \\ a_j < a_i}} L(j)$$

$a_j < a_i$

Now, let's formally state the recurrence for $L(i)$ in terms of smaller subproblems. $L(i)$ requires that a_i is included in the subsequence. Therefore we get 1 for including element a_i in it. And then we take the longest subsequence that we can append on to the beginning. So we're going to take the max over all earlier indices of the length of the subsequence ending at that character a_j . Now that earlier subsequence allows us to append a_i onto the end of it only if a_j is strictly smaller than a_i . So you only want to consider j 's where a_j is strictly smaller than a_i . And then we can take that earlier subsequence ending at a_j and append a_i onto the end of it and we get $L(j)$ for that earlier subsequence plus 1 for a_i . And of course, we need that j is earlier in the input array than a_i . Just in case the mathematical notation is confusing for anybody let me re-express it with slightly different notation and re-explain what it's saying in words [this is his narrative as he builds the recurrence formula]:

- So $L(i)$ is the length of the longest increasing subsequence on the first i elements, which includes a_i . So we get one for that element a_i and then we're taking, we have a_i here.
- Then we're going to take the longest increasing subsequence that we can put at the beginning. It's going to end at some element a_j .
- Okay. So we're going to try all possibilities for that j . Now, we need that index j is earlier in the input array. So we're going to try j 's between $i-1$ and 1. And we only want to try j 's where a_j is strictly smaller than a_i .

- So we're doing a max over j . j is the variable that we're varying. We're trying j 's where it's between 1 and $i-1$ in such that a_j is strictly smaller than a_i and we're taking the value $L(j)$.
- Finally, we have our sub-problem definition and we have our recursive relation that the sub-problems satisfy.

Now to be straightforward to detail our dynamic programming algorithm. So let's give this pseudocode for a dynamic programming algorithm and then analyze its running time and then we'll have completed this problem.

DP Algorithm

$L(i) = 1 + \max_j \{L(j) : a_j < a_i \text{ \& } j < i\}$

LIS(a_1, \dots, a_n):
for $i = 1 \rightarrow n$:
 $L(i) = 1$
 for $j = 1 \rightarrow i-1$:
 if $a_j < a_i$ & $L(i) < 1 + L(j)$
 then $L(i) = 1 + L(j)$
 max = 1
 for $i = 2 \rightarrow n$:
 if $L(i) > L(\text{max})$ then max = i
Return ($L(\text{max})$)

Now, let's state the pseudocode for the dynamic programming algorithm for the longest increasing subsequence problem. And I kept the recurrence from the previous slide about the $L(i)$ in terms of smaller subproblems as a sort of crib notes for ourselves.

Here's the pseudocode for our dynamic programming algorithm for the LIS problem. The input is a_1 through a_n . Our solutions are in one dimensional array L and we're going to fill the table L from bottom up. So we're going to start at index $i=1$ and go up to $i=n$. So we're going to have a single for loop for i going from 1 to n . Now our subproblem formulation requires that a_i is included in the solution for $L(i)$ plus one term. Therefore, we start by setting $L(i) = 1$. And then we're going to do the max over j . So we're going to have another for loop which varies j from 1 to $i-1$. Now we need to check that a_j is strictly smaller than a_i . If a_j is strictly smaller than a_i , then we need to check if the solution we can obtain by appending a_i onto the end of the solution ending at a_j is longer than our current best solution. So if a_j is strictly smaller than a_i , and our current best solution $L(i)$ is strictly smaller than the new solution, we can obtain by appending a_i onto the end of $L(j)$. In this case we want to update our current best solution, which is now the solution that we obtained by appending a_i onto the end of the solution that ends at a_j . Now we've given an algorithm which defines our table L . But what is

the solution that we're trying to output from this algorithm? For the case of Fibonacci numbers, it was the last entry in the table. In this case, the last entry of the table is the longest increasing subsequence ending at a_n . That's not necessarily the solution that we're trying to obtain. We're trying to find the longest increasing subsequence regardless of where it ends at. So what we need to do is try to look through L and find out maximum entry in the entire one dimensional array. So let's go ahead and do that. We'll make a variable max , which will maintain the index with the largest entry of the table. We'll start max at 1 and then we'll go through the entries at the table to see if we find a larger entry than the current max . This for loop simply finds the largest entry of the table and stores the index for that entry in the variable max . Finally, what do we return? We return the entry at index max . This completes the formulation of our dynamic programming algorithm. Now let's take a look at the running time of our algorithm.

Longest Increasing Subsequence Algorithm

LIS(a_1, a_2, \dots, a_n) :

for $i = 1 \rightarrow n$:

$L(i) = 1$

for $j = 1 \rightarrow i - 1$:

if $a_j < a_i$ & $L(i) < 1 + L(j)$:

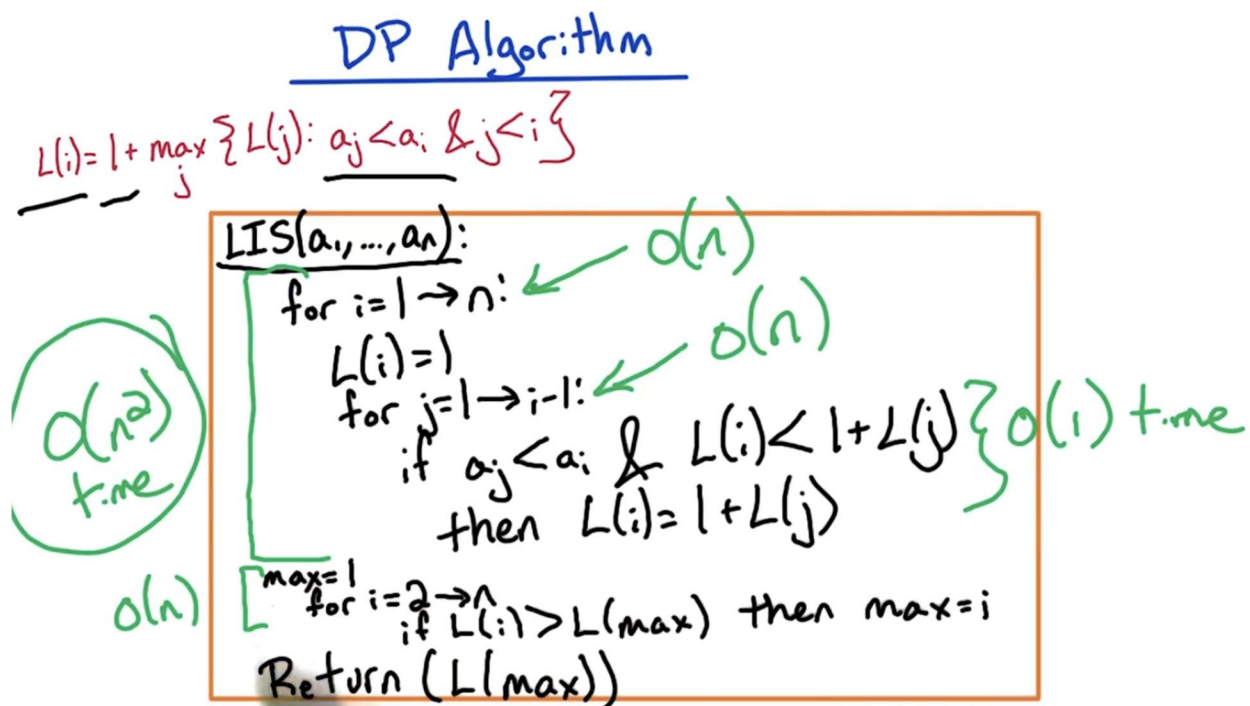
then $L(i) = 1 + L(j)$

$max = 1$

for $i = 2 \rightarrow n$:

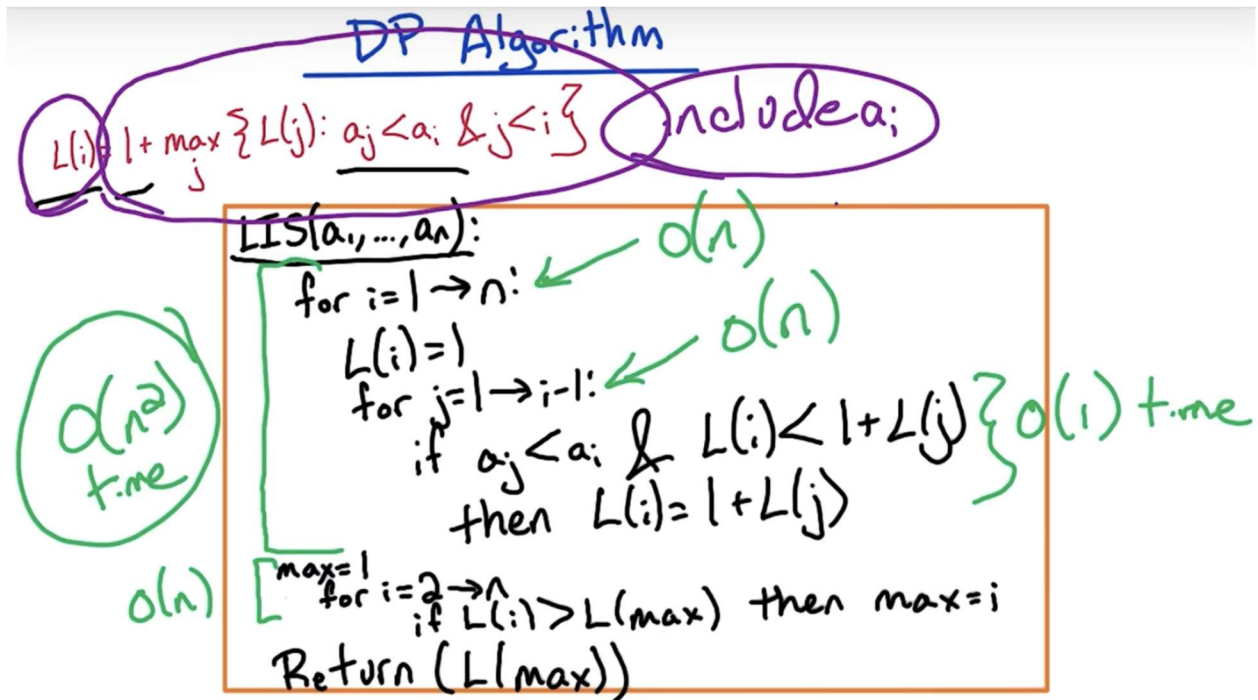
if $L(i) > L(max)$ **then** $max = i$

return $L(max)$



We have one **for loop**, which varies over order n elements, and we have another nested **for loop**, which varies over, at most, n elements. Within these two nested **for loops**, we have a **if-then** statement, which takes $O(1)$ ("order one") time. Therefore, the total time for these nested **for loops** is, $O(n^2)$ ("order n -squared") time. Finally, we have another **for loop** which finds a **max** entry in the table. This is one **for loop**, it's not nested, this is $O(n)$ ("order n ") time. Therefore, the total run time of our algorithm is $O(n^2)$ ("order n -squared") time.

(Slide 14) LIS: Recap



This completes the formulation of our dynamic programming algorithm, and the analysis of its running time.

But now, let's go back and take a look at some important aspects of our algorithm design:

- [FIRST STEP] What was the first step in our recipe for designing a dynamic programming algorithm? It was to define the sub-problem in words. In this case, define $L(i)$ in words.

Our initial attempt was to use a prefix of the input, to find the length of the longest increasing sub-sequence on the first i elements of the array.

- [SECOND STEP] Then, our second step in our recipe is to find a recurrence relation that the solutions' sub-problems satisfy.

In that case, for that definition, we were unable to do so. So what did we do? We went back and **we reformulated our sub-problem definition**. We added an extra condition into it, and then we were able to write a recurrence for the sub-problems.

Now, the intuition for why we wanted to strengthen the sub-problem definition: It goes back to, if you recall perhaps from a discrete math class way long ago, when you were trying to prove some statement by induction, how did you go about it? Well, you would first state an inductive hypothesis. Usually, the inductive hypothesis is of the same form as the statement that you're trying to prove. Then you try to prove that hypothesis using induction, but sometimes you run into difficulties. And what do you do? You go back and you alter your inductive hypothesis. Typically, what you do is you try to strengthen your inductive hypothesis by adding extra conditions in. That's exactly what we did here, we added the extra condition, that the sub-problem has to include a_i itself. Okay? And then, you strengthen the inductive hypothesis and you prove that stronger statement.

And that's what we're doing here. We're giving you an algorithm for this stronger problem. We're finding, in length, the longest increasing sub-sequence with the specific character at the end. Using that solution to that stronger problem, we can then solve the weaker problem, where we don't care, what is the ending character. So, a lot of the intuition for what we're doing in dynamic programming, comes from some ideas from induction proofs.

Longest Common Subsequence (LCS)
input: 2 strings $X = x_1 \dots x_n$ & $Y = y_1 \dots y_n$
Goal: find the length of longest string
which is a subsequence of both X & Y

Our next dynamic programming example is Longest Common Subsequence. Which we'll use the shorthand LCS. The input to the problem is two strings, which will denote as X and Y . And for simplicity, for now, we'll keep them both of the same length. Our goal is to find the length of the longest string which appears as a subsequence in both X and Y , and notice that we're trying to find the subsequence, not a substring, and we're simply trying to output the length of this string. And if we can find the length, we can use that to find an actual string which appears as a subsequence and we'll see how to do that at the end.

(1st Slide 16) LCS Example

Longest Common Subsequence (LCS)

input: 2 strings $X = x_1 \dots x_n$ & $Y = y_1 \dots y_n$

Goal: find the length of longest string
which is a subsequence of both X & Y

Example: $X = \text{BCD BCDA}$
 $Y = \text{ABECBAB}$

length of LCS = _____

Let's take a look at an example to make sure you're okay with the terminology. Here's an example with two strings of length seven. Let's do a quick check to make sure you understand the problem definition. In this example, what's the longest common subsequence? Go ahead and mark the longest common subsequence in this example and also indicate the length of it.

(2nd Slide 16)

Longest Common Subsequence (LCS)

input: 2 strings $X = x_1 \dots x_n$ & $Y = y_1 \dots y_n$

Goal: find the length of longest string
which is a subsequence of both X & Y

Example: $X = \text{BCD BCDA}$
 $Y = \text{ABECBAB}$

length of LCS = 4

The solution is the substring BCBA, BCBA, which is the length four. Why do we care about it? Well, I mean first off it's a nice example, which illustrates a slightly different flavor of the dynamic programming solution. That's the main motivation here, but also, actually, this simple problem is used in Unix diff. If you want to compare two files, you have this unique script diff, which compares two files and looks at the differences and it does utilize the longest common subsequence dynamic programming algorithm.

DP Design: Attempt 1

Step 1: Design subproblem in words
try same problem on prefix of input

For i where $0 \leq i \leq n$,
let $L(i)$ = length of LCS in $x_1 \dots x_i$
 $y_1 \dots y_i$

Step 2: Define recurrence
Express $L(i)$ in terms of $L(1), \dots, L(i-1)$

So let's dive into the first two steps in our recipe for designing a dynamic programming algorithm. Our first step, is to define the subproblem in words. Recall our subproblem definition from some of our earlier examples. For Fibonacci numbers, we set $F(i)$ to be the i -th Fibonacci number. In the longest increasing sub sequence problem, we set $L(i)$ to be the length of the longest increasing sub sequence on the first i numbers in the input. What are we going to do here? Our first attempt is always to try this same problem on a prefix of an input. So it's key. It's just identical to the original problem and it's just on a prefix. So all we've changed is from doing it on length n to length i , first i characters. So let's formalize that. So we're going to have a variable i which is the prefix length and the prefix length varies between 0 corresponding to the empty string and length n which is our original input. And we define a new function $L(i)$, which is the length of the longest common sub sequence in the first i characters of X and the first i characters of Y . Our second step, is to define a recurrence. We want to express $L(i)$ in terms of $L(1)$ through $L(i-1)$. So let's take a look at how to define the recurrence in this example, that we saw earlier.

LCS: Recurrence Attempt 1

For i where $0 \leq i \leq n$
let $L(i)$ = length of LCS in $X_1 \dots X_i$
 $Y_1 \dots Y_i$

Example: $X = \text{BCD BCD A C}$
 $Y = \text{A B E C B A B C}$

Either: $X_i = Y_i$ $L(i) = 1 + L(i-1)$

or $X_i \neq Y_i$

Let's detail the subproblem definition we proposed. We have a parameter i which corresponds to the prefix length. i is going to vary between 0 and n . i equals 0 corresponds to the empty string. i equals n corresponds to the original input. Now, we're going to define a function $L(i)$. This will be the length of the longest common subsequence in a prefix of X of length i and a prefix of Y of length i . So, X_1 through X_i and Y_1 through Y_i . Note, this is analogous to the original problem except on a prefix of the input. One other notable difference, this does not store the subsequence itself. It simply stores the length. We want our table to store our number or true/false. Now, let's take a look back at our earlier example. Now, this was our earlier example. We're trying to find a recursive relation for $L(i)$. We're trying to express L of i in terms of smaller subproblems, L_1 through L_{i-1} . Let's try to gain some insight from this example. In order to get a smaller subproblem, we're going to look at the last character. So we're going to look at X_i and Y_i . We're going to look at how X_i and Y_i are used in the solution to $L(i)$ and then, we can use the solution to the subproblem of size $i-1$. We take the optimal solution for the subproblem of size $i-1$ and then we append on the solution for X_i and Y_i . Now, there are two cases to consider: either of these last characters are different, in this case, or they're the same. We're going to consider these two cases separately. The first case is when the last characters X_i and Y_i are the same. The second case is when the last characters X_i and Y_i are different.

We're going to do this case first: $X_i = Y_i$. The last characters are the same. This turns out to

be the easy case. Now, let's modify our example so that the last characters are the same - so, I append on the character C onto the end of both strings. Now, in this case, where the last character is the same, what do we know about the longest common subsequence? Well, we know it must include and must end in this common character. Why? Well, it give me a common subsequence and suppose it does not include this last character. Well, then, I can append on this common character and I get a longer subsequence. So, therefore, the longest common subsequence must include this last character. So, in this case, where $X_i = Y_i$, what do we know about $L(i)$? We know that the longest common subsequence includes this last character. So, we get one in the length for that common character and then we can simply drop this last character and then we can take this input sequence of length $i-1$ and we can take the longest common subsequence in this input sequence of length $i-1$ and append on this common character C. What is the length of the longest common subsequence in this input? It's simply $L(i-1)$. So, we have a recursive relation. We can express $L(i)$ in terms of $L(i-1)$: $L(i) = 1 + L(i-1)$. This handles a case when $X_i = Y_i$.

Now, let's take a look at the case when X_i is not equal to Y_i .

LCS: Recurrence Problem

For i where $0 \leq i \leq n$
let ~~$L(i)$~~ = length of LCS in $X_1 \dots X_i$ $Y_1 \dots Y_j$ X

Example: $X = \text{BCD BCDA}$
 $Y = \text{ABECBA}$

$X_i \neq Y_i$: LCS does not include X_i &/or Y_i
Need $LCS(X_1 \dots X_i, Y_1 \dots Y_{i-1})$
 $i \& j$

Let's take a look now, at the case when the last characters are different. This is the situation in our original example, $X_i \neq Y_i$ - A is not equal to B. Now in this case, when X_i is not equal to Y_i , the last character of the longest common subsequence can either be A or B, or neither. Certainly, cannot be both. Now suppose it's A. For example, suppose this is the last character in the long common subsequence. And what we know about Y_i (B)? - for B has nothing left to match within X. Therefore, we know the LCS does not include B, Y_i . Similarly, if the last character is B - say it's a match with this B - then the LCS cannot include A, because it has nothing left to match within Y. The key point is that the longest common subsequence for this prefix of length i , either does not include X_i or it does not include Y_i or both. So either X_i is dropped or Y_i is dropped or both of them are dropped.

Now let's consider these three cases:

- If both of them are dropped, then we can simply take the longest common subsequence in this prefix of $L(i-1)$. So it's similar to the equal case except we don't get this +1 here.
- Now what happens if just X_i is dropped? Well, then, we have a prefix of length $i-1$ in X and a prefix of length i in Y. So we have no way of looking this up in our table.

The solution to this subproblem is not in our table because the prefixes are of different length in X and Y. And notice, even if we knew how Y_i is matched up with an X, for instance, if we knew this B was a match with this B, then we have a prefix of length 3 in X and we have a prefix of length 6 in Y. So there's still a different length.

- And similarly, if Y_i is not included, so we dropped this last character from Y then we have prefix of length 7 in X and a prefix of length 6 in Y. So the prefixes are of different length in X and Y.

And once again, the solution to this sub problem is not in our table because these prefixes are on a different length. So for this case where where Y_i is dropped, we need to look up the longest common subsequence in the prefix of length i in X with the prefix of length i-1 in Y. Now this isn't in our table presently. And, similarly when we try to solve this problem, well, then, we might chop off the last character from Y and then we get even shorter prefixes in length in Y.

So for this sub problem definition, we are unable to define a recurrence. We are unable to express $L(i)$ in terms of smaller subproblems, but we got some insight about what is valid.

What is a good subproblem definition? The difficulty here was that the prefixes are of the same length for x and y, but we need to allow them to be of different lengths. So how do we achieve that?. We're going to change from a single parameter i to a pair of parameters i and j: i will correspond to the length of the prefix X and j will correspond to the length of the prefix Y. And, our table will now be a two dimensional table.

So $L(i,j)$ will be the length of the longest common subsequence in X_1 through X_i with Y_1 through Y_j . And then, we're going to try all possibilities for i and j.

DP Design: Attempt 2

Two indices i & j & a 2-dimensional table

Subproblem definition:
For i & j where $0 \leq i \leq n$ & $0 \leq j \leq n$
let $L(i, j)$ = length of LCS in $\begin{matrix} x_1 \dots x_i \\ y_1 \dots y_j \end{matrix}$

Recurrence:

$L(i, \underline{0}) = 0$ $L(\underline{0}, j) = 0$

So let's go back and revise our subproblem definition with the insight that we just gain. The key insight was that we're going to need a prefix of x and a prefix of y of possibly different lengths.

Therefore we're going to have two indices, i and j , which are going to correspond to the length of the prefix in X and the length of the prefix in Y . And as a consequence, we're going to end up with a two-dimensional table. In all our previous examples, we had a one dimensional array or table. And now, we're going to have a two-dimensional table or array, because we have two indices, i and j .

So let's go ahead and formalize our subproblem definition. For indices i and j where once again i varies between 0 and n , and similarly j varies between 0 and n . We're going to define the function. We're going to define the function $L(i, j)$, which is the length of the longest common subsequence in the first i characters of x and the first j characters of y . The key is that X is of length i and Y is a prefix of length j .

Now let's go ahead and see if we can define the recurrence for this new subproblem definition. Let's start off with the base cases:

- Base cases are going to be when $i = 0$ or $j = 0$. Let's start with considering the case when $j = 0$. So what is $L(i, 0)$? and what is $L(0, j)$ for the case when i is 0? Well, for $L(i, 0)$, so here j is 0. That means that Y is the empty string. So what is the longest

common subsequence? Well, in both where Y is the empty string, so there is no subsequence in it. So the length of it is of length 0. Similarly, if we're taking an empty string for X then the longest common subsequence is the empty string itself, which is of length 0. So those are two base cases. Let's move on to the more interesting cases.

Recurrence: Unequal case

let $L(i,j)$ = length of LCS in $X_1 \dots X_i$
 $Y_1 \dots Y_j$

Example: $X = BCDBCD A$
 $Y = ABECB A B D$

if $X_i \neq Y_j$: either X_i &/or Y_j are not in optimal solution

if Drop X_i then $L(i,j) = L(i-1,j)$

if Drop Y_j then $L(i,j) = L(i,j-1)$

Recurrence: Unequal case

let $L(i,j)$ = length of LCS in $X_1 \dots X_i$
 $Y_1 \dots Y_j$

Example: $X = BCDBCD A$
 $Y = ABECB A B D$

if $X_i \neq Y_j$: either X_i &/or Y_j are not in optimal solution

$$L(i,j) = \max \{ L(i-1,j), L(i,j-1) \}$$

So let's go ahead and try to find the recurrence for our new subproblem definition. The length that the longest common subsequence, in the first i characters of X and the first j characters of Y . And let's go back and look at our earlier example.

Here's our earlier example, and notice that here, they're both of the same length and just to

illustrate our new subproblem definition, let's add in one more character to Y, so there are now of different length. And if you recall our approach from before, what we're going to do is, we're going to look at the last character of both sequences, and we're going to condition on whether they're the same or different. If they're the same and then we can possibly match them together, if they're different, there's no way we can match them together.

So let's do the case where they're different first. Our key insight, is that in this case with the last characters are unequal, then the longest common subsequence either ends in A the last character of X, or in the last character of Y, which in this case is D, or it ends in neither. Now if it ends in A, that means it does not end in D, it does not end in D, it does not include D in the final solution, okay? There might be other occurrences of D, but we're differentiating the multiple occurrences of this character. *This* character is not included in the final solution, or *this* character is not included in the final solution, or both of these final characters are not included in the optimal solution. So there are three cases to consider: (1) either the last character of X is not used in the optimal solution (2) the last character Y is not used in the optimal solution, or (3) the last character of both is not used in the optimal solution. If the last character of both is not used, we can get there by dropping the last character of X and then dropping the last character or Y or vice versa or dropping less character Y and then X. So there's really only two cases to consider: either we drop the last character of X, or the last character of Y, and we're going to take the best of those two.

So let's recap, so if we dropped the last character of X and X_i is not used in the optimal solution then, $L(i,j)$ is the same as $L(i-1,j)$, which has just dropped X_i from it, and j. The other scenario, is that we dropped the last character of Y and in this case $L(i,j) = L(i,j-1)$ - we just dropped Y_j from Y. How do we know which of these two is the better solution? We just take the larger of the two. So that's going to be our recurrence. Our recurrence is $L(i,j)$ is going to be the max of these two possible solutions.

Let's recap, in this case where X_i and Y_j are different, so the last characters of the two prefixes are different, then we have a recurrence. $L(i,j)$ is going to be the best of the two possible scenarios. Either we dropped the last character of X, and therefore we have this solution for $i-1$ characters of X, and j characters from Y, or we drop the last character from Y and we get the solution, $L(i,j-1)$, and which of the two are we're going to take? We're gonna take the better of the two. Which means we're going to take the one which has larger length which is the max. So this defines a recurrence for the case when X_i is not equal to Y_j . Now let's do the case where X_i equals Y_j .

Recurrence: Equal case

let $L(i,j)$ = length of LCS in $X_1 \dots X_i$
 $Y_1 \dots Y_j$

Example: $X = \text{BC DB C D A}$
 $Y = \text{A B E C B A}$

if $X_i = Y_j$: either Drop X_i , Drop Y_j or optimal solution ends at $X_i = Y_j$

$L(i,j) = L(i-1,j)$ $L(i,j) = L(i,j-1)$ $L(i,j) = 1 + L(i-1,j-1)$

So for our new subproblem definition, we have defined the recurrence for the case when the last characters of the two prefixes are different. Now, let's go look at the case where the last characters of the two prefixes are the same. So, I have modified the example so that this is the case. Here, they both end in A.

We're looking now at a case where X_i equals Y_j , So the last characters of the two prefixes are the same. And now, in this case, there are three possibilities to consider. As in the unequal case, we can either not include X_i in the optimal solution, in which case we drop X_i or we drop Y_j . Notice actually those two are the same in this scenario because X_i and Y_j are the same. So we're dropping X_i we're also dropping Y_j . I'm just going to foreshadow a little bit of the solution or recurrence. The other possibility which didn't exist in the unequal case is that the optimal solution ends in the common character X_i (equals Y_j). As in the unequal case, what we're going to do is we're going to look at these three scenarios and we're going to take the best of the three.

- Now, in the first case, where we drop X_i , notice that $L(i,j) = L(i-1,j)$, where we just dropped X_i from the solution, just like in the unequal case.
- Similarly, when we drop Y_j , $L(i,j) = L(i,j-1)$.
- Now, the interesting case which didn't exist before in the unequal case is that the optimal solution ends in this common character. And in this case, $L(i,j) = 1 + \dots$, the one is for the common character that were pending on the end and then, we're going to take the optimal solution which drops X_i and Y_j from there and we take the

optimal solution to these smaller prefixes. So this is $L(i,j) = 1 + L(i-1,j-1)$. We take the length of that optimal solution for that smaller problem, add one for this common character which we append at the end and that gives us the length of the longest common subsequence in this new subproblem.

Recurrence: Equal case

let $L(i,j)$ = length of LCS in $X_1 \dots X_i$
 $Y_1 \dots Y_j$

Example: $X = \text{BC DB C DA}$
 $Y = \text{ABECBA}$

if $X_i = Y_j$: $L(i,j) = \max\{L(i-1,j), L(i,j-1), 1 + L(i-1,j-1)\}$

$L(i,j) = 1 + L(i-1,j-1)$

So let's recap our recurrence for the case where X_i equals Y_j , the last characters are the same and the two prefixes. Here, we have three possibilities and we're going to take the best of the three, therefore, we take the max.

- We have $L(i-1,j)$, which corresponds to dropping X_i ;
- $L(i,j-1)$, which corresponds to dropping Y_j ;
- or we include X_i (Y_j) the common character at the end and we get a plus one for that, and then we take the optimal solution to this smaller subproblem $L(i-1,j-1)$,

we take the max of these three.

Some of you may have noticed that in fact, the recurrence, in this case, can be simplified even further. We only need to consider the last case, where we append on X_i (Y_j) the common character onto the end. Let me give you some brief intuition about why that's the case. Notice, if this solution doesn't include either of these characters, then we can just add them on and we get a longer. So, therefore, it wasn't optimal length. So it's got to include either X_i or Y_j .

Now, it may be the case that X_i is matched with some earlier recurrence of that character. In this case, it's Y_1 , but any solution which we obtain by matching X_i with an earlier recurrence


of that character, we could have obtained Y instead matching X_i with this latter solution. Any subsequence which occurs in that smaller prefix of Y , also is a subsequence in this larger prefix of Y , okay? It's a superset of it. Therefore, we only need to consider this case and we get this simpler recurrence for this case where they're equal, okay?

And this completes our definition of the recurrence.

Recurrence: Summary

let $L(i,j)$ = length of LCS in $X_1 \dots X_i$
 $Y_1 \dots Y_j$

For $i \geq 1, j \geq 1$:

$$L(i,j) = \begin{cases} \max\{L(i-1,j), L(i,j-1)\} & \text{if } X_i \neq Y_j \\ 1 + L(i-1,j-1) & \text{if } X_i = Y_j \end{cases}$$


Let's recap the recurrence that the $L(i,j)$ satisfies. Now we're looking at the case with the two strings are non-empty. So i has at least one and j is at least one. Now we had two cases here. We had the case X_i is not equal to Y_j , the last characters are different, and we have the case where the last characters are the same. In the case where they are the same, we had the simple recurrence. We depend on X_i, Y_j , and therefore get a plus one for that, and then we take the optimal solution to the smaller subproblem with one less character in each string. Now in the case where they are not equal, we took the best of two scenarios. We either drop the last character from X , or we drop the last character from Y . And we take the larger of these two solutions, and we also had the base cases, where one of the two strings was empty, in which case the length is 0.

Finally, now we can state our dynamic programming algorithm. The pseudocode for the algorithm. And now, notice we have a two dimensional table now. L is a two dimensional array. How are we going to fill it up? We're going to fill it up row by row. And now when we're looking at this entry $L(i,j)$, what are the entries that we're going to need for it? We're going to need either this diagonal. It's right here, $L(i-1,j-1)$, which will be there because we filled in the previous row, or we'll need the entry just above, or we'll need the entry just to the left. So we're just gonna look at these three entries, which will all be there, because we are filling it up row by row. So earlier in this row will be completed and the previous row will be completed. And now we can go ahead and state our dynamic programming algorithm.

LCS: DP Algorithm

LCS(X,Y):

```
for i=0→n, L(i,0)=0
for j=0→n, L(0,j)=0
for i=1→n
  for j=1→n
    if Xi=Yj then L(i,j)=1+L(i-1,j-1)
    else L(i,j)=max{ L(i,j-1), L(i-1,j) }
Return (L(n,n))
```

Now, we can state the pseudocode for our dynamic programming algorithm for the longest common subsequence problem.

The input are these two strings, X and Y.

Now, let's start with the base cases which are the top row and the first column, which first set $L(i,0)$ to be zero. This corresponds to setting the first column to zeros. Then, we set the first row to all zeros. These are our two base cases.

Now we can fill the interior of our table. As we said earlier, we're going to go row by row. Index i is going to correspond to the current row. Then we go along this row. This is the index j . Now we have two cases to consider, either the current last characters of X and Y are the same or they're different. So, if $X_i = Y_j$, we have one recurrence. And if X_i is not equal to Y_j , then we have a different recurrence. In the case where $X_i = Y_j$, the recurrence is $1+$, the one comes from the appending X_i and Y_j to the end, and then taking the optimal solution for the prefix of length $i-1$ with length $j-1$.

In the case where they're not equal, the recurrence is the best of the two scenarios. Either we drop the last character from Y, giving us $L(i, j-1)$, or we drop the last character from X, giving us $L(i-1, j)$ and we take the max of these two.

Finally, what is the output of our algorithm? It's the entry in the bottom right of our matrix. $L(n, n)$ is the length of the longest common subsequence for all of X with all.

(2nd Slide 25) LCS: DP Algorithm (Answer)

LCS: DP Algorithm

LCS(X, Y):

$O(n)$ { for $i = 0 \rightarrow n, L(i, 0) = 0$
for $j = 0 \rightarrow n, L(0, j) = 0$

$O(n^2)$ { for $i = 1 \rightarrow n$
for $j = 1 \rightarrow n$
if $X_i = Y_j$ then $L(i, j) = 1 + L(i-1, j-1)$
else $L(i, j) = \max \{ L(i, j-1), L(i-1, j) \}$

$O(n^2)$ total time

$O(1)$

Return ($L(n, n)$)

Finally, let's take a look at the running time of our algorithm.

- The two base cases are each $O(1)$ per step and then they are $O(n)$ sized **for loop**. Therefore, they take $O(n)$ total time.
- Then we have a **for loop** of size $O(n)$ ("order n") and a **nested for loop** of size $O(n)$ and inside it takes $O(1)$ time. So the total time is $O(n^2)$ with these nested **for loops** and the total run time of our algorithm is dominated by the $O(n^2)$, so we get $O(n^2)$ total time.

This completes the dynamic programming algorithm for the longest common subsequence problem. The interesting thing that arose in this solution was that we needed to use a two

dimensional table. This came about because we needed to consider prefixes of X of different length than prefixes of Y .

DP1: Addendum

Practice Problems: 6.1, 6.2, 6.3, 6.4, 6.11*

Approach:

1. Define subproblem in words
 - a. prefix
 - b. add constraint - include last element
2. Recurrence relation
 $T(i)$ in terms of $T(1), \dots, T(i-1)$

At this point, I suggest you try some practice dynamic programming problems. There are a lot of great practice problems in the Dasgupta book.

Let me suggest a few problems for you. Now, these are problems from the textbook Algorithms by Dasgupta, Papadimitriou, Vazirani. This problem 6.1, which is about finding a contiguous subsequence of maximum sum. Now, note a contiguous subsequence is the same as a substring. Now, whenever I sign you a homework problem, I'm always going to tell you some little blurb about what the problem is about, so you can identify it. Because, some of the online versions of this textbook have different numbering. So, you should always make sure it's the correct problem, if you have an online version.

The next problem is 6.2 where you're planning a trip and you want to figure out the hotel stops to minimize the penalty. Problem 6.3 is about yuckDonald's. In 6.4, you're given a string and you want to see if you can break up that string into a set of words.

Now, 6.11 is longest common subsequence. We already did that in this lecture, but for practice, you might try the variant where you did the longest common substring. It's useful to look at how dynamic programming solutions vary. What's the difference when you have substring versus subsequences?

Once again, at this point, you should be able to do these practice problems: 6.1, 6.2, 6.3, 6.4, 6.11 or this variant (6.11*) of 6.11 where you do substrings instead of subsequences.

Let me give you a quick summary of the approach we used for designing a dynamic programming algorithm. And you can implement this approach when you try these practice problems:

- The first step is to define the subproblem in words. How do you do that? Well, you'd take the same problem, the original problem, on a prefix of the input.

So, the subproblem should be of the same form as the original problem, but instead of being an input of length n , try it on a prefix of the input. So, an input of length i .

- The second step is to define a recurrence relation. If you have a one-dimensional table, then you want to express $T(i)$ or the i th entry of this one dimensional table in terms of smaller subproblems - $T(1)$ through $T(i-1)$. This is what we did for LCS, though in that case, it was a two dimensional table.

Now, when we did LIS, Longest Increasing Subsequence, we were unable to do this. And what we needed to do was strengthen the subproblem. So, we had to go back and redefine the subproblem. So, we've strengthened the subproblem by adding the constraint. Typically, we add the constraint that the last element, element i , is required to be included in this solution to this subproblem. So, for LIS, we said, $L(i)$ was the longest increasing subsequence in the prefix of the input of size i . But the last element A_i had to be included in that solution.

Now, one thing to keep in mind is typically, when we add in this constraint, and the final output is no longer the last element in the table, but instead we're going to have to do a max or a min. But we're going to have to go over the whole table and look for the best, the optimal solution in the table.

Now, I want to show you how I solve these practice problems. So, I'm going to show you how I approach this problem 6.1. But to get the most out of it, I suggest that you try the problem first, and then if you're having difficulty, you can watch the next slide to see how to solve it.

[DPV] 6.1: Attempt 1

Input: a_1, \dots, a_n

Goal: substring with max sum

Subproblem: For $0 \leq i \leq n$

let $S(i) = \text{max sum from substring of } a_1, \dots, a_i$

$S(i)$ in terms of $S(1), \dots, S(i-1)$

The input to this problem is a_1 through a_n . Our goal is to find the contiguous sub-sequence. Notice, that a contiguous subsequence is the same as a substring, and we want to find the substring with maximum sum. Now, the first step is to define the subproblem in words. I'm going to try the same problem on a prefix of the input. So, we have a parameter i , which is the length of the prefix. So, we have an i , and this is going to vary between zero and n . And now, I'm going to define $S(i)$, in words, first off. It's going to be the max sum that I can obtain...the same format as the original problem...but, instead of considering the whole input, and I'm only going to consider the first i numbers. So, it's a max sum from a substring of a_1 through a_i . So it's the first i characters of the input. Now, we want to try to find a recurrence relation, we want to express $S(i)$ in terms of $S(1)$ through $S(i-1)$ smaller subproblems. Let's take $S(i-1)$...is the max sum that we can obtain from a substring of a_1 through a_{i-1} . So, let's take that sum. Now, there's two possibilities that I would consider: either I add on a_i on to the end of it or not. First off, can I add a_i on to the end of this solution? Well, I have to maintain that it's a substring, that is contiguous. I don't know where this one ends, so, I don't know whether I can add on a_i onto the end of this solution. So, I need to know where this one ends. With this current definition, there's no way - I don't know how to express $S(i)$ in terms of the smaller subproblems. But I notice, that if I knew where this solution ended at, so, if it included a_{i-1} , then, I know that if it includes a_{i-1} then I can add on a_i and maintain that it's a substring. So, I have to go back and strengthen my subproblem definition.

[DPV] 6.1: Solution

Input: a_1, \dots, a_n

Goal: substring with max sum

Subproblem: For $0 \leq i \leq n$

let $S(i) = \text{max sum from substring of } a_1, \dots, a_i$
which includes a_i

$$S(i) = a_i + \max\{0, S(i-1)\}$$

Output: $\max_i S(i)$

Here's a solution to problem 6.1. Let's reformulate our subproblem using our insight we just gained. We wanted to add the extra condition that the subproblem had to include a_i .

So we define $S(i)$ as a max sum which we can obtain from a substring of a_1 through a_i with the extra restriction that a_i has to be included in that substring. Now, we will be able to re-express a recurrence for $S(i)$.

First off, let's handle the base case; that's $i=1$. What's $S(0)$? Well, we have the empty string, so that's, of course, zero.

Now, let's look at $S(i)$ for i at least one. Well, we are required to use a_i , so let's add an a_i . So by including a_i , this sum is at least a_i plus ...what do we append onto it? Well, there are two possibilities: either we just use a_i by itself or we append it on to the longest substring from a_1 through a_{i-1} and we're going to take the best of those two possibilities. So if we have a_i by itself, then we get zero for the rest, for a_1 through a_{i-1} . OR, if we take the optimal substring for a_1 through a_{i-1} , what's the sum from that? Well, it's $S(i-1)$ and we're going to take the best of these two scenarios. Now, clearly if this is negative, then we're going to just use a_i by itself, and if this is positive, then we're going to take this solution and append on a_i to it.

So this gives us the recurrence, it's easy to fill the table by going from $i=0$ up to n . What's the

final output of the algorithm? It's not necessarily $S(n)$ because that's the longest maximum sum from a substring which includes a_n . BUT, we're just looking for the maximum sum we can obtain from any substring - we don't care what the last character is. So, we need to try all possibilities for the last character or number. So we do a max over i of $S(i)$. So we take our one-dimensional table and we look for the max entry in that table and that's the output.

And what's the running time of this algorithm? Well, each entry of the table takes $O(1)$ time because we just have to look at two numbers and how many entries in the table are there, and there's $O(n)$. So the total run time is $O(n)$.