[SQUEAKING] [RUSTLING] [CLICKING]

ERIK DEMAINE: All right, welcome back to 006. Today we start a totally new section of the class. Up till now, we've mostly been showing you really cool and powerful algorithms, sorting algorithms, graph algorithms, data structures, trees, lots of good stuff that you can apply to solve tons of algorithmic problems, either by reducing to the data structures that we showed you, or reducing to graph problems that we showed you, or by modifying those algorithms a bit.

Today we're going to start a new section on algorithmic design-- how to, from scratch, come up with a polynomial time algorithm to solve a problem. And in particular, we're going to talk about a algorithmic design paradigm called dynamic programming, which is extremely powerful. It's probably the most powerful algorithmic design paradigm. Very general. Can solve lots of problems.

It's a particular type of recursive algorithm design. And in general, this class-- all of algorithms-- is about recursive algorithm design at some level, because we want to write constant-sized pieces of code that solve problems of arbitrary size. We have some problem size n and we're trying to write 100 lines of code or whatever, some constant amount that doesn't depend on the problem size. We have one algorithm that solves all instances of the problem.

And so we have to write code that is recursive or uses loops or somehow reuses the instructions that we give the computer. And you may know you can convert any algorithm based on loops into an algorithm using recursion. And we're going to take the recursive view today, in particular because it fits very well with our proof-by-induction technique, which we've used throughout this class, but also because it gives us some structure on how different subproblems relate in something called a subproblem graph, that we'll be talking about today.

And so we're going to start out with, in general, how do we design recursive algorithms? That's sort of the overall, encompassing everything. We have thought very hard to come up with a cool acronym for this paradigm which we invented called SRTBOT-- thanks, Jason. And so we'll talk-- it's not actually for sorting. It's just an acronym for sub-problems, relations, topological order, base case, original problem, and time.

But it's an acronym that will help you remember all the steps you need in order to specify a recursive algorithm. And dynamic programming is going to build on this template by adding one new idea called memoization, which is just the idea of reusing work that you've done before. And that's going to let us solve tons of problems.

And let's see. I don't-- let's get into it. So we'll start out today with SRTBOT. So here is SRTBOT down the column here. This is a recursive algorithm design paradigm. And in general, what we're going to do is take the problem that we actually want to solve and split it up into lots of possible sub problems.

And so the first part is to define what the heck are the subproblems. In general, we'll want some polynomial number of them. But it's pretty open-ended what these look like. And the hardest part, usually, in defining a recursive algorithm is figuring out what the sub problems should be. Usually they're related to the problem you want to solve.

Often the problem you want to solve-- this is actually near the last step-- the original problem you're trying to solve is often one of these sub problems. And then you use the smaller sub problems in order to build up the final, original problem. But sometimes at the end, you need to take a bunch of subproblems and combine them into your original problem.

You can think-- one analogy you can think of here is divide and conquer algorithms, which also had this kind of style. But more generally, we're going to relate different sub problem solutions with some recursive structure-- some recurrence relation. This is just a recursive algorithm that defines how to solve one problem in terms of smaller sub-problems for some notion of smaller.

And this is given by the topological order. So if we think of the subproblems as a graph and we draw an edge between-- so the vertices of the graph are sub problems. The edges are the dependencies between those subproblems. Then what we'd like is the topological ordering, the topological sort problem we talked about in the context of DFS or DAG shortest paths.

What we would like is that the subproblems and the calls-- the recursive calls between them in this recursive relation-- forms a DAG. We want it to be acyclic, otherwise you have an infinite loop in your recursive calls. If you have a cycle, you'll never terminate. And so to make sure that these dependencies between subproblems given by this recurrence relation is acyclic, one way to do that is to specify a topological order.

Or you could prove it some other way. But often it's just a for loop to say, just do it in this order. Then of course any recursive structure needs base cases. So that's a useful step not to forget. We want to solve the original problem using these sub problems. And then we analyze a running time at the end. So six easy steps. Actually, the hardest ones are these two, which are interrelated.

And what we're going to see over the next four lectures-- this is the first of four lectures on dynamic programming-- is lots of examples of applying this paradigm over and over together with the memoization idea, which we'll get to soon. Let's see an example first of an algorithm we've already seen, which is merge sort, so a divide and conquer algorithm, phrased with this structure of SRTBOT.

So for the sub problems-- so our original problem is to sort the elements of A. And some sub-problems that we solve along the way are sorting different sub-arrays of A. So for every-- well, not for every i and j, but for some i and js, we sort the items from i up to j minus 1. So I'm going to define that subproblem to be s of ij.

So this is something that I might want to solve. The original problem that I want to solve is s of 0 comma n, where n is the length of the array. So that's what I actually care about in the end. But we're going to solve that by writing it recursively in terms of sorting different sub-arrays as follows. This is the recurrence relation. I've written it very simply here.

Of course, there's a merge algorithm, which is somewhat complicated. But as we saw the two finger linear time merge algorithm, given two sorted arrays-- so this is supposed to be the sorted array version of the items i through m. m is the middle element between i and j and the sorted array of the items from m up to j.

If we merge those, that gives us the sorted array from i up to j. And that's exactly what merge sort does. So in general, this relation is just some algorithm for if you're given the solutions to some smaller subproblems, how do I solve the subproblem that I want to solve?

And so we need to make sure that this problem is bigger than the ones that we recursively call on and that we don't get an infinite cyclic loop of recursions. And here our valid topological order is to say, solve these problems in order where j minus i-- the length of the sub-array-- is increasing. And then you can check because m is strictly between i and j. As long as we're not in a base case, then we know we can-- these subarrays will be smaller than this one. And so this increasing order gives us a valid topological order on all of the problems, all the subproblems.

We have a base case, which is if we don't want to sort anything, that's the empty array, or at least in the original problem. And then running time is-- I mean, there's no better way to solve it than the recurrence that we already saw how to solve. So this is just another way to think of n log n merge sort in this labeled framework of SRTBOT.

Let's get to another problem that does not fit recursion so well. But we can make it better. So this is-- we're going to start with a very simple problem, which is computing Fibonacci numbers. It's really just a toy problem to illustrate a very powerful idea, which is memoization.

So the problem I'm interested in is I'm given a particular number, n. And I want to compute the nth Fibonacci number. And in case you forgot, the nth Fibonacci number is given by this recurrence. fn is fn minus 1 plus fn minus 2 with base case, let's say, f1 equals f2 equals 1.

And so we'd like to compute this. This seems-- this is a recurrence. So it seems very natural to write it as a recursive algorithm. So let's try to do it. We start with what are the sub problems. The obvious sub problems are just the various Fibonacci numbers, f i for i between 1 and n. So there are n of these sub problems.

Cool. Let's see. We want a relation between them. Well, maybe just to distinguish the problems from the Fibonacci numbers, let me write f of i. This is a function, an algorithm we're going to define. And it's defined to be-- the goal we're trying to get is the ith Fibonacci number given i.

And then we can write the recurrence relation on these guys, just f of i equals f of i minus 1 plus f of i minus 2. So in other words, recursively compute those Fibonacci numbers then add them together. That's an algorithm. Next is t for topological order.

Here, of course, we just want to compute these in order of increasing i from the base case is up. Another way I like to write this is as a for loop for i equals 1 to n. We will see why. But this gives an explicit order to compute these sub problems.

And base case is just the same as the Fibonacci numbers, but I guess I should write in parentheses. The original problem we want to solve is f of n. And the time-- all right, here's where things get interesting or bad. So what is the running time of this recursive algorithm? As I've stated it so far, the running time is given by a recurrence. Let's write the recurrence.

So in order to compute f of n, I recursively compute f of i minus 1 or f of n minus 1 here. And I recursively compute f of n minus 2. So that will take t of n minus 2. This first step will take t of n minus 1. And now I need to solve this recurrence.

This is not a recurrence that falls to the master method. It doesn't have a divided by. So we have to think about it a little bit. But we don't have to think about it too hard, because this recurrence is the same as this recurrence, which is the same as this recurrence. I've written it three times now. And so the solution to this is the nth Fibonacci number.

Oh, sorry. It's a little bit worse because in addition to those recursions, I also spend constant time to do the addition, maybe more than constant time. But if we just count the number of additions we do, it will be plus 1 additions. OK. But this is bigger than the nth Fibonacci number. And if you know anything about Fibonacci numbers, they grow exponentially. They're about golden ratio to the end. I'm wearing golden ratio, in case you forgot the number.

So that's bad, because golden ratio is bigger than 1. So this is exponential growth, as we know, especially in this time, exponential growth is bad. In algorithms, exponential growth is bad, because we can only solve very small problems with exponential growth. Very small n. So this is a terrible way to compute the nth Fibonacci number-- exponential bad.

OK, so don't do this. But there's a very tiny tweak to this algorithm that makes it really good, which is memoization. And this is a big idea. It is the big idea of dynamic programming. It's a funny word, probably made up by computer scientists. Instead of memorization, it's memoization, because we're going to write things down in a memo pad. It's the idea. And it's a very simple idea, which is just remember and reuse solutions to sub-problems.

So let's draw the recursion tree for this recursive algorithm as we've done it so far. So at the top, we-- let me make a little bit of space. At the top we are calling f of n. And then that calls f of n minus 1 and f of n minus 2. And it does an addition up here. And then this calls f of n minus 2. And this calls f of n minus 3.

This calls f of n minus 3. And this calls f of n minus 4. OK. And we notice that this sub problem is the same as this sub problem. So to compute f of n minus 1, I need f of minus 3. And also to compute f of n minus 2 I need f of n minus 3. So why are we computing it twice? Let's just do it once. When we solve it, let's write it in a table somewhere. And then when we need it again, we'll just reuse that value. Question?

AUDIENCE: What about the f of n minus 2?

ERIK DEMAINE: f of n minus 2 is also shared. So let me use a different symbol. f of n minus 2 is already here. So this was at the same level. But we also get shared reuse between different levels. In fact, I wouldn't even call f of n minus 3 because this whole part doesn't need to be computed a second time. If I already computed it here, it doesn't matter which one comes first. Let's say this one comes first.

Once this is done, I can write it down and reuse it over here. And then in here, we're going to call f of n minus three. So there's still another computation of f of n minus 3. When that one's done, I won't need to do this recursively. OK, so magically this is going to make this algorithm efficient with this very simple tweak. Let me write down the tweak more explicitly.

I won't write code here. But just describe it as a data structure. So we're going to maintain our good friend, the dictionary, which is abstract data type or interface. We could use different data structures to do it. But we're going to map some problems to their solutions, at least the ones that we've solved already.

And usually we can do this with just a direct access array, though you could use a hash table. Just get expected bounce. So when we write the code for our recursive function-- so in general, once we have a sort bot description, we can turn this into code. We define f of i. And it says am I in a base case? If so, return this. Otherwise, do this recursive call. That's our recursive algorithm.

But we're going to do a little more now. And first we're going to check whether this sub problem that we're trying to solve has already been solved. And if so, we return that storage solution. That's the easy case, but it might not exist.

And then we'll compute it in the usual way. So what the code then would look like to define f of i is first we check is i in our data structure. This is usually called the memo. So we say, is this sub-problem-- is i in my memo data structure? If so just return memo of i. Done. No recursion necessary.

Otherwise, check if I'm a base case. If so, done. Otherwise, recurse. So recursively call f of i minus 1 and f of i minus 2. And in this recursion, we can see that after we call f of i minus 1, in fact, it will have already computed f of i minus 2. So while this call is recursive, this one will immediately terminate because i minus 2 will already be in the memo table.

And so if you think about what happens, in fact, we'll just have recursion down the left branch of this thing. And all the right branches will be free. We can just look things up in the memo table. So what is the overall running time? For Fibonacci, this should be order n. Why is it order n? This is number of additions. Come back to that in a second.

In general, the way to analyze an algorithm like this that uses memoization is we just count how many different sub-problems are there? Because once we solve the sub-problem, we will never solve it again. That's the whole idea of a memo table. So we will solve each sub-problem at most once.

And so we just need to count, how much time does it take to solve every sub-problem? And here you can see it's constant. Either it's a base case and it takes constant time or we recursively call these things. But those are different sub-problems. So we're going to count those later.

And then the work that's actually done by this recurrence is a single addition. So in fact, it's n additions. To compute fn would be exactly n additions. So it turns out to be very nice closed form in this case. It should be exactly n sub problems to compute f of n because we started as dot at 1. And each one has one additional-- I guess not the base case. Maybe n minus 2.

OK. Definitely order n. Now, there's this one subtlety which-- let's forget about dynamic programming for a moment and go back to good old lecture one and two, talking about the word ram model of computation. A question here that usually doesn't matter in this class. Usually we assume additions take constant time. And we usually do that because it's usually true.

And in general, our model is the w bit additions-- where w is our machine word size-- takes constant time. But for this problem and this problem only, pretty much, for Fibonacci numbers, I happen to know that the Fibonacci numbers grow exponentially. So to write them down actually requires theta n bits because they are some constant to the n power. And so they're actually really big . n is probably bigger than w. Usually you think of problems that are much bigger than 64 or whatever your word size happens to be.

We do assume that w is at least log n. But n is probably bigger than w. It might be bigger or smaller. We don't know. And in general, to do an n bit addition-- these are n bit additions-- is going to take ceiling of n over w time. So in the end, we will spend this times n, because we have to do that, many of them, which is n plus n squared over w time.

So a bit of a weird running time. But it's polynomial, whereas this original recursive algorithm was exponential here. Using this one simple idea of just remembering the work we've done, suddenly this exponential time algorithm becomes polynomial. Why? Because we have few sub problems. We had n sub problems. And for each sub problem, we could write a recurrence relation that if we already knew the solutions to smaller sub problems, we could compute this bigger problem very efficiently.

This happened to be constant time or constant additions. n over w time. But as long as this is polynomial and this is polynomial, we're happy, because we have this nice formula that the time it takes is, at most, the sum over all sub problems of the relation time.

So I'm referring to sub problems, like a number of them and the time it takes to evaluate this, ignoring the recursive calls. That's important. This is the non recursive part. In the notes, I call this non-recursive work. So this formula gives us a way to bound the running time of one of these algorithms if we use memoization.

Without memoization, this is not true, Fibonacci to exponential time. But if we add memoization, we know that we only solve each sub-problem once. And so we just need to see, for each one, how much did it cost me to compute it, assuming all the recursion work is free, because that's already taken into account by the summation.

So in particular, this summation is at most the number of sub-problems times the time per sub-problem, which in this case was order n. We could try to apply that analysis to merge sort, because after all, this is also a recursive algorithm. It happens to not benefit from memoization. But we could throw in memoization. It wouldn't hurt us.

But if you think about the call graph here, which is like s of 0 m, which calls s of m-- 0 n over 2 and o of n over 2n and so on. It has the same picture, but there's actually no common substructure here. You'll never see a repeated sub-problem, because this range is completely disjoined from this range.

But you could throw in memoization and try to analyze in the same way and say, well, how many sub-problems are there? It looks like there's n choices for i and not quite n choices but it's at most n squared different choices. In fact, it's the triangular number sum of i equals 1 to n of i, different possible choices for inj.

But this is theta n squared sub-problems, which seems not so good. And then how much time are we spending per sub problem? Well, to solve s of ij, we have to merge about that many elements. We know merge takes linear time. And so this takes theta j minus i time to evaluate.

And so what we'd like to do is sum over all the sub problems of j minus i. This is the not triangular number but the tetrahedral number, I guess. And so we end up that the running time is, at most, n cubed. Great. So it's true that n log n is less than or equal to n cubed, but obviously not terribly useful. This algorithm by the way we already know how to analyze it is, indeed, n log n. And the running time turns out to be theta n log n.

So sometimes this equation is not what you want to use. But often it's good enough. And especially if you just want to get a polynomial upper bound, then you can try to optimize it later. This will give you a polynomial upper bound as long as the number of sub-problems is polynomial and the time per sub-problem is polynomial.

And indeed, n cubed is polynomial. It's not a great polynomial, but this is an alternate way to analyze merge sort. Obviously don't do this for merge sort. But it illustrates the technique. Good so far? Any questions? All right. Let me remember where we are. Cool.

So the next thing I'd like to do is show you one more algorithm that we've already seen in this class that fits very nicely into this structure-- arguably is a dynamic program-- and that is DAG shortest paths. So just to close the loop here, when I say dynamic programming, I mean recursion with memoization.

I mean, we take-- we write a recursive piece of code, which is like def f of some args, some sub-problem specification. We check is the problem in the memo table? If so, return memo of sub-problem.

And otherwise check if it's a base case and solve it if it's a base case. And otherwise, write the recurrence recurse via relation. And set the memo table of the sub-problem to be one of those things. OK, so this is the generic dynamic program. And implicitly, I'm writing Fibonacci in that way. And all of the dynamic programs have this implicit structure where I start with a memo table which is empty and I always just check if I'm in the memo table. If I am, I return it. Otherwise I compute according to this recursive relation by recursively calling f.

And that's it. So this is every DP algorithm is going to have that structure. And it's just using recursion and memoization together. OK, so now let's apply that technique to think about the DAG shortest paths problem. The problem was, I give you a DAG. I give you a source vertex, S-- single source shortest paths. Compute the shortest path weight from S to every vertex. That's the goal of the problem.

And we saw a way to solve that, which is DAG relaxation. I'm going to show you a different way, which turns out to be basically the same, but upside down, or flipped left right, depending which way you direct your edges. So what are our sub-problems? Well, here, actually, they're kind of spelled out for us. We want to compute delta and SV for all these. So that is size of these sub-problems.

That turns out to be enough for this overall problem. And the original problem we want to solve is all of the sub-problems. We solve all the sub-problems, we're done. And then we have-- I think we wrote this at some point during the DAG shortest paths lecture-- we have a recursive relation saying that the shortest way to get from s to v is the minimum of the shortest path to get to some vertex u plus the weight of the edge from u to v.

Why? Because if we look at a vertex v, unless we started there, we came from somewhere. And so we can consider all of the possible choices for the previous vertex u. And if you start at s and get to v, you must go through one of them. And so this is finding the best way among all the choices of u. What's the best way to get to u? And then take the edge from u to v for all edges uv.

And this is adjacency minus. We don't usually think of that. Usually we look at adjacency plus the outgoing edges. This is the incoming edges. And so u is an incoming-- uv is an incoming edge into v. OK, if we take that minimum-- and of course, possible there is no way to get to v. And so I'll also throw infinity into the set.

Take the min of that set. That will give me the shortest pathway in an acyclic graph from s to v. And great, this is recursive. This was a sub problem. These are sub problems which are smaller, I guess. There's no clear notion of smaller here, except we already know the clear notion of smaller is the topological order of our DAG.

Because our graph is acyclic, we know it has a topological order. We know how to compute it with DFS. And so that guarantees there's a topological order to compute these problems. And in fact, the relationship between problems is exactly the given graph, G. In order to compute the shortest pathway from s to v, I need to know the shortest pathway from s to all of the incoming vertices to v.

And so this is I guess in the call graph, this vertex calls this vertex, but direct the edge this way to say that this vertex requires-- this vertex needs to be computed before this one. And so then I can complete them in a topological order.

OK, we have a base case, which is delta of ss equals 0. And the running time is, again, we can use this formula and say, let's just sum over all the sub problems of the non recursive work in our recurrence relation and so it's computing this min. If I gave you these deltas for free and I gave you these weights, which we know from our weight data structure, how long does it take to compute this min? Well, however many things there are, however many numbers we're minning, which is the size of the incoming adjacency list plus 1 for that infinity.

And so if you compute this sum, sum of incoming edges to every vertex, that's all the edges. So this is v plus e. So in fact, this algorithm is morally the same algorithm as the one that we saw on the DAG shortest path lecture, which was compute a topological order and process vertices in that order and relax edges going out from vertices.

So here-- so in that algorithm, we would have tried to relax this edge if there was a better path to v. And the first one certainly is better than infinity. So the first one we relax indeed. The next edge, if this gave a better path from s to v, then we would relax that edge and update the way here and do the same here.

In the end, we're just computing this min in the relaxation algorithm but doing it step by step. In the relaxation algorithm, DAG relaxation, for each incoming edge to v, we update d of e if it's better. And so if you repeatedly update if you're better, that ends up computing a min. OK, so this is the same algorithm just kind of flipped backwards.

A funny thing, although we wrote down the topological order of the sub problem graph here is the topological order of g, because the sub-problem graph is g, the algorithm doesn't actually have to compute one. It's doing it automatically for free.

If you think about this algorithm, generic dp algorithm, which is check whether we're in a memo table. If so, return. Otherwise, recurse, or base case. This actually is a depth-first search through the sub-problem graph-- technically through the reverse of the sub-problem graph.

If I draw an edge-- so from small to big-- so I'm just saying, I orient the edges from my smaller sub-problems to the ones that need it-- then I'm actually depth-first searching backwards in this graph because the bigger problem calls the smaller problem. And the memo table is serving as the "have I visited this vertex already" check in DFS.

So this is actually a DFS algorithm. Plus we're doing some computation to actually solve the sub-problems we care about. So implicit in this algorithm, we are doing a DFS, and at the same time, we're doing this shortest path computation in the finishing order of that DFS traversal because all the edges are backwards. This is the same as the reverse finishing order if the graph is forwards.

So in the end, we're computing a topological order because dynamic programming includes in it depth first search. A lot of words. But it's kind of cool that this framework just solves DAG shortest paths without much work. I mean, we did a lot of work in shortest paths to prove that this relation is true. Once you know it's true, the algorithm part is pretty much free. You just write down SRTBOT and you're done.

OK. This brings us to in general-- at this point we have seen two examples of dynamic programming. I guess technically merge sort you could think of as a dynamic program, but it doesn't actually reuse anything. So it's not interesting. And indeed, that gave us a really bad bound.

We've definitely seen DAG shortest paths and Fibonacci numbers as two interesting examples. And what the next remainder of this lecture and the next three lectures are going to be about is more and more examples of dynamic programming and how you can use it to solve increasingly general problems.

So far, we've just solved an easy problem and a problem we already knew how to solve. Let's go to a new problem, which is bowling. Bowling is popular in Boston. Boston likes to play candlepin bowling, which is a bit unusual. Today we're going to play an even more unusual bowling game, one that I made up based on a bowling game that Henry [INAUDIBLE] made up in 1908. So ancient bowling, I'll call it, or I think linear bowling is what I might call it. I'll just call it bowling here.

And now I'm going to attempt to draw a bowling pin. Not bad. They might get progressively worse. So imagine n identical bowling pins. Please pretend these are identical. And I have a ball which is approximately the same size as a bowling pin. These bowling pins are pretty close together. I should have left a little gap here.

And you are a really good bowler. Now, unfortunately, these bowling pins are on a line. And you're bowling from way down at infinity. So when you bowl, you can only hit one pin or two pins or zero pins. But probably you want to hit some pins. So if you bowl straight out of pin, you will just hit that one pin. And if you bowl in the middle between two pins, you will knock down-- that's a ball, sorry-- you will knock down two pins. And this is your model of bowling, model of computation.

Now, what makes this interesting is that the pins have values. Pin i has value-- this is obviously a toy problem, though this problem-- this type of bowling does go back to 1908, it was also a toy problem in that setting. So each of these bowling pins has some number on it, let's say 1, 9, 9-- I'll do a slightly more interesting example, maybe another one here and a 2 and a 5 and a 5, something like this.

OK. Or maybe make it a little more interesting. Let's put some negative numbers on here. OK. And the model-- so you're at the carnival bowling. Each pin has different-- potentially different values. And the model is if you hit one pin, i, then you get vi points. So that's straight forward. To make it interesting, when you hit two pins, you get the product.

So if I hit two pins, it's always i and i plus 1 for some I. You get vi times vi plus 1 points. This is the game you're playing. And it doesn't really matter that this is a product. Product is just some weird function that's hard to imagine.

If you stare at this long enough, you should convince yourself that the optimal solution is probably to-- so, for each of these numbers, I could leave it singleton or pair it with its left neighbor or pair it with its right neighbor. But the pairings can't overlap because once I hit a pin, it's gone. It's knocked over. It disappears.

So because of these nine, which are a very high value, what I'd probably like to do is hit both of them together, so pair them up, because 9 times 9 is 81. That's really big, much better than hitting them individually or hitting 9 times 1 or 9 times 2.

1 and 1 is kind of funny, because it's actually better to hit them individually. That will give you two points, whereas if I'd pair them up, I only get one point. 2 and minus 5, that seems bad. Negative 10 points. My goal is to maximize score.

Do you have to hit all the pins? Let's say no, you don't have to hit all the pins. So I could skip the minus fives. But in fact, here, because they're adjacent, minus 5 times minus 5 is good. That's 25 points. So the optimal solution for this particular instance are to hit all the pins, these positive, these together, these together. If I added, for example, another pin of minus 3 here, I would choose not to hit that pin. Good question.

So you just play until you are tired. When you decide to stop playing, how can I maximize your score? There are many variations of this game. All of them-- basically any variation-- not literally every variation, but many, many variations of this problem can all be solved quickly with dynamic programming. But let's solve this particular one. OK.

So now we're really in algorithmic design mode. We need to think about SRTBOT. And in particular, we need to think about what would the sub problems be here? And at this point, we don't have a lot of help. So I should probably give you some tools. If I want to solve a problem like this, the input is a sequence of numbers. It's a sequenced data structure. Maybe it's an array of numbers, which is this v array.

And let's see. A general tool for sub-problem design which will cover most of the problems-- maybe all of the problems that we see in this class for dynamic programming. Here's a trick. If your input is a sequence, here are some good sub-problems to consider.

We could do all prefixes. So let's call the sequence x. So we could do x prefix means up to a given i for all i. We could do all the suffixes, x from i onward for all i. Or we could do substrings, which are the consecutive items from i to j. I don't write subsequence here. Subsequence means you can omit items in the middle.

So substring you have to start in some position and do all the things up to j. So these are nice, easy to express in Python notation. And these are great, because they're polynomial. If I have n things-- if the length of my sequence, x, is n, then there are n prefixes-- technically n plus 1. So let's do theta n prefixes. There are theta n suffixes. And there are theta n squared substrings because there's n-- roughly n choices for i and j separately. Sorry?

Sub-sequences. Good. Right. I didn't write sub-sequences, because in fact, there are exponentially many sub sequences. It's 2 to the n. For every item, I could choose it or not. So I don't want to parameterize-- I don't want my sub problems to be sub sequences because that's guaranteed-- well, then you're guaranteed to get an exponential number of sub-problems, which is bad.

We'd like to balance the numbers of sub-problems by polynomial. So these are three natural ways to get polynomial bounds. Now, prefixes and suffixes are obviously better because there's fewer of them, linear instead of quadratic. And usually almost every problem you encounter, prefixes and suffixes are equally good. It doesn't really matter which one you choose.

So maybe you'd like to think of-- well, we'll get to-- just choose whichever is more comfortable for you. But sometimes it's not enough. And we'll have to go to substrings. That won't be for another lecture or two. Today I claim that prefixes or suffixes are enough to solve the bowling problem.

So what we're going to do is think about-- I prefer suffixes usually, because I like to work from left to right, from the beginning to the end. So we're going to think of a suffix of the bowling pins. And so what is the sub-problem on a suffix? Well, a natural version is just to solve the original problem, bowling. How do I maximize my score if all I were given were these pins?

Suppose the pins to the left of i didn't exist. How would I maximize my score on the remaining pins? Or for this suffix, given these four pins, what would I do? And there's some weird sub problems here. If I just gave you the last pin, what would you do? Nothing. That's clearly different from what I would do globally here.

But I claim if I can solve all suffixes I can solve my original problem, because one of the suffixes is the whole sequence. So let's do it. Sort by for bowling. So here is our dynamic program. The sub-problems are suffixes. So I'll write b of i is the maximum score we could get possible with our starting-- if we started a game with pins i, i plus 1, up to n minus 1, which is a suffix of the pins.

Very important whenever you write a dynamic program to define what your sub-problems are. Don't just say how to compute them, but first say what is the goal of the sub problem. This is a common mistake to forget to state what you're trying to do. So now I have defined b of i. Now, what is the original thing I'm trying to solve?

You also put in SRTBOT-- you could put the O earlier, then it actually spells sort. So why don't I do that for fun. The original problem we're trying to solve is b of 0, because that is all of the pins. The suffix starting at 0 is everything. So if we can solve that, we're done. Next is r for relate. This is the test of, did I get the sub-problems right, is whether I can write a recurrence relation.

So let's try to do it. We want to compute b of i. So we have pin i here and then the remaining pins. And the big idea here is to just think about-- the nice thing about suffixes is if I take off something from the beginning, I still have a suffix.

Remember, my goal is to take this sub-problem, which is suffix starting at i, and reduce it to a smaller sub problem, which means a smaller suffix. So I'd like to clip off one or two items here. And then the remaining problem will be one of my sub problems. I'll be able to recursively call b of something smaller than i-- or sorry, b of something larger than i will be a smaller subsequence because we're starting later.

OK, so what could I do? Well, the idea is to just look at pin i and think, well, what could I do to pin i? I could not hit it ever with a ball. I could skip it. That's one option. What would be my score then? Well, if I skip pin i, that leaves the remaining pins, which is just a smaller suffix. So that is b of i plus 1.

I'm going to write a max out here because I'd like to maximize my score. And one of the options is, forget about pin i. Just solve the rest. Another option is I throw a ball. And I exactly hit pin i. That's one thing I could do. And it would leave exactly the same remainder. So another option is b of i plus 1 plus vi.

Why would I prefer this over this? Well, if vi is negative, I'd prefer this. But if vi is positive, I'd actually prefer this over that. So you can figure out which is better, just locally. But then there's another thing I can do, which is maybe I hit this pin in a pair with some other pin. Now, there's no pin to the left of this one. We're assuming we only have the suffix. And so the only other thing I can do is throw a ball and hit i together with i plus 1. And then I get the product.

Now, what pins remain? i plus 2 on. Still a suffix. So if I remove one or two items, of course, I still get a suffix-- in this case, b of i plus 2-- and then the number of points that I add on are vi times vi plus 1. So this is a max of three things. So how long does it take me to compute it? I claim constant time.

If I don't count the time it takes to compute these other sub problems, which are smaller because they are smaller suffixes further to the right, then I'm doing a couple of additions-- product, max. These are all nice numbers and I'll assume that they live in the w-bit word, because we're only doing constant sized products. That's good.

So this takes constant, constant non-recursive work. How many sub problems are? Well, it's suffixes, so it's a linear number of sub problems. And so the time I'm going to end up needing is number of sub problems, n, times the non-recursive work I do per sub problem, which is constant. And so this is linear time. Great.

And I didn't finish SRTBOT, so there's another t, which is to make sure that there is a topological order and that is in decreasing i order. Or I might write that as a for loop-- for i equals n, n minus 1. This is the order that I would compute my problems because the suffix starting at n is the empty suffix. The suffix starting at 0, that's the one I actually want to compute. That's the final suffix I should be computing.

And then we have a b for base case, which is that first case, b of n equals 0, because there's no pins. So I don't get any points. Sad. OK, so this is it. We just take these components, plug them into this recursive, memoized algorithm, and we have a linear time algorithm. I want to briefly mention a different way you could plug together those pieces, which is called bottom up dp, which is-- let's do it for this example.

So if I have-- let's see. Let me start with the base case, b of n equals 0. But now it's an assignment. And I'm going to do for loop from the topological order for i equals n, n minus 1 to 0. Now I'm going to do the relation, b of i equals max of b of i plus 1 and b of i plus 1 plus bi and b of i plus 2 plus di vi plus 1. Technically this only works if i is strictly less than n minus 1.

So I should have an if i is less than minus 1 for that last part because I can only do-- I can only hit two pins if there's at least two pins left. And then return b of 0. So what I just did is a transformation from this SRTBOT template into a non-recursive algorithm, a for loop algorithm, where I wrote my base case first. Then I did my topological order. Then I did my relation.

Then at the end, I did my-- not base case. The original problem. And provided you can write your topological order as some for loops. This is actually a great way to write down a dp as code. If I were going to implement this algorithm, I would write it this way, because this is super fast. No recursive calls. Just one for loop.

In fact, this is almost a trivial algorithm. It's amazing that this solves the bowling problem. It's in some sense considering every possible strategy I could for bowling these pins. What we're using is what we like to call local brute force, where when we think about pin i, we look at all of the possible things I could do to pin i, here there's really only three options of what I could do.

Now, normally, if I tried all the options for pin i and then all the options for i plus 1 and i plus 2 and so on, that would be exponential. It'd be 3 times 3 times 3. That's bad, but because I can reuse these sub problems, it turns out to only be linear time. It's almost like magic. dp-- dp is essentially an idea of using local brute force.

And by defining a small number of sub-problems up front-- and as long as I stay within those sub problems, as long as I'm always recursing into this polynomial space, I end up only doing polynomial work, even though I'm in some sense exploring exponentially many options. And it is because what I do to this pin doesn't depend too much to what I do to a pin much later.

There's a lot of intuition going on here for what-- when DP works. But we're going to see a lot more examples of that coming up. And I just want to mention the intuition for how to write a recurrence like this is to think about-- in the case of suffixes, you always want to think about the first item, or maybe the first couple of items. The case of prefixes, you always think about the last item. And for substrings, it could be any item-- maybe in the middle.

If I remove an item from the middle of a substring, I get two substrings, so I can recurse. Here or in general, what we want to do is identify some feature of the solution that if we knew that feature we would be done. We would reduce to a smaller sub problem. In this case, we just say, well, what are the possible things I could do to the first pin?

There are three options. If I knew which option it was, I would be done. I could recurse and do my addition. Now, I don't know which thing I want to do. So I just try them all and take the max. And if you're maximizing, you take the max. If you're minimizing, you take the min. Sometimes you take an or or an and. There might be some combination function.

For optimization problems where you're trying to maximize or minimize something, like shortest paths you're trying to minimize, we put them in here. So usually it's min or max. And this is extremely powerful. All you need to do-- the hard part is this inspired design part where you say, what do I need to know that would let me solve my problem?

And if you can identify that and the number of choices for what you need to know is polynomial, then you will be able to get a polynomial dynamic program. That's the intuition. You'll see a lot more examples in the next three lectures.