**PROFESSOR:** We ended up the last lecture talking about dynamic programming. And we'll spend all of today on that topic. We showed how it could be used to provide a practical solution to the shortest path problem, a solution that would allow us to solve what was, in principle, a complex problem quickly. And we talked about the properties that make it possible to solve that problem and, in general, the properties we need to make dynamic programming applicable.

So let's return to that topic right now. What we looked at is we looked at two properties. The first was optimal substructure. And you recall what this meant is that we can construct a globally optimal solution by combining solutions to local subproblems. So we've looked at a lot of problems already this term that have optimal substructure-- merge sort, for example, which exploits the fact that a list can be sorted by first sorting components of it and then combining those. So merge sort exhibited optimal substructure.

The second property we looked at was overlapping subproblems. And what this meant was that in the course of running our algorithm, we would end up solving the same problem more than once. And that is what gave us the power of using memoization, to use table lookup instead of re-computing the solution.

Notice that merge_sort does not have overlapping subproblems. There's no reason that we would expect, if we're sorting a list, that we would ever encounter the same sublist twice. And so in fact, we cannot use dynamic programming to solve sorting. Because it has one of these properties, but not both.

So how about shortest path? Does it have these properties? Well, let's first think about optimal substructure. You might begin by asking the question if I knew the shortest path from A to B, and I knew the shortest path from B to C, can I necessarily combine those to get the shortest path from A to C? No, exactly right.

Because for example, maybe there's a direct link from A to C. So we can't combine them that way. So that gives us the question-- Oh, pardon?

[INAUDIBLE]

**PROFESSOR:** --Almost got it there. So that gives us the question, what is the optimal substructure here? Well, there is something we do know about shortest path, that if I have the shortest path from one node to another. And I take any sub path of that, I will have-- so let's say I have a path that goes from-- let's say I know that the shortest path from A to C is A to B to D to E to C. If this is the shortest path from A to C, I know that this is the shortest path from B to E. Right? Because otherwise, I wouldn't have bothered going through D. If there were, for example, a link directly from B to E, then the shortest path from A to C would not have included this.

So that's the optimal substructure I have, and that's the substructure that I exploited last time in showing you the dynamic programming solution to the shortest path problem.

You also recall that we looked at the shortest path. We'd found overlapping subproblems. Because I would end up solving the same intermediate problem multiple times, figuring out how to get from one node to another. So indeed, we saw that the shortest path problem exhibited both of these properties and therefore, was amenable to solution by the dynamic programming. And in fact, we ran it and we saw that it ran quite quickly.

It's not immediately obvious, but the 0-1 knapsack problem also exhibits both of these properties. You'll recall a few lectures ago, and also in a problem set not so long ago, we looked at a recursive solution to the 0-1 knapsack problem. It's called "solve" in this code. It uses backtracking to implement the decision tree.

if we look at the decision tree we had there-- I gave you a small example here of the 0-1 knapsack problem. I've got A, B, C and D. I've got some values, and I've got some weights. We started by saying at the beginning, we looked at this node here, where we had decided not to take anything . We still had the list A, B, C, and D to consider. The total value of our items was 0, and for our example, I think we had 5 pounds of material left, or 5 units of weight.

We then considered a decision. What happens if we take A? Well, if we take A, we have B, C, and D left to consider. We now look at this node and say, all right, our value is 6 -- the value of A. But we only have 2 pounds left. We then considered going depth first, left first in this case. This is the depth- first search. We've looked at that before. All right, let's consider the next branch.

Well the next branch we'll consider C. Well we discover-- no we'd consider B rather. We can't take B, because B weighs 3. So this is a branch that we can't get to. We can elect the other decision not to take B, in which case we still have C and D left to consider. And still 6 and 2. We then proceed and said right, well can we take C? Yes we can. And that leaves us with a value of 14 and no weight left. So since we have no available weights, we might as well stop considering things. So this is a terminal node. We then back up, and said all right, let's consider the decision where we don't take C. Can we take D? And the answer is no because it weighs too much. And then we're done.

And so we see on this side of the tree, we've explored various possibilities. And this is our best option so far. And we can walk through and now we consider well, what if we don't take A? Well then we have these decisions to make. And eventually, we can look at the whole thing and make a decision.

By the way, I noticed this morning that there was an error on your handout. I chose not to correct it in the slide so that you would notice it was in your handout as well. What should node 18 be? What's the error here? Pardon?

[INAUDIBLE]

**PROFESSOR:** Right. That should be the empty one. Right? Because we considered everything. And in fact, typically all of your bottom nodes should either have 0 weights or no items left to consider. Not very deep, just a typo but has the advantage that by looking for it, you can see whether you understand how we're constructing these decision trees. So this is a very common way to deal with an optimization problem. You consider all combinations of decisions in a systematic way. And when you're done, you choose the best one. So this is using depth-first search and backtracking. And this is essentially what you did in your problem set.

Now what you saw in your problem set is that if the set of items is even moderately large, this algorithm can take a very long time to run. How long? Well, at each level of the tree we're deciding whether to keep or not keep one item. So what's the maximum depth of the tree? How many levels can this tree go? If we have n items-- Pardon?

[INAUDIBLE]

**PROFESSOR:** n levels-- exactly. So if we have n items, it can go n levels-- Good grab. All right, how many nodes do we have at each level? Well, at level 0 we have only 1 node. But what's the down at the bottom? Let's say we have an enormous amount of weight so that in fact we never run out of weight in our knapsack. We'll have quite a broad tree, right? How many nodes at level 2? At each level, right? Down here we have up to 4 nodes, up to 8 nodes, up to 16 nodes. So if we had say, 39, 40 items, at level 39 we'd have 2 to the 39th nodes-- pretty darn big tree. All right? Not very deep, but very bushy.

So we know that for any reasonable number of items and a reasonable amount of weight, this is just not going to work. And that's why when you wrote your problem set, and you were trying to solve the optimal set of courses to take, you could only run it on a small subset of the courses at MIT. Because if we gave you everything-- and we gave you a lot for a test-- you noticed it effectively just wouldn't finish.

So now we have to ask the question well, can we use dynamic programming to solve this problem? And in particular, that boils down to the question of does this solution exhibit optimal substructure and overlapping subproblems? Well, optimal substructure is easily visible in both the decision tree and the code. Right? Each parent node combines the solution-- We can look at the code since you have the subtree in your handout-- or the decision tree in your handout. And the key place in the code is where we combine decisions from lower down in the tree as we go up.

So at each parent node, we select the better of the two children. Right? If the left child is better than the right child, we select the left. Otherwise, we select the right. And we just percolate up the tree. So there's clearly optimal substructure here, that we can solve the higher nodes with solutions to the lower nodes. And we see that again in the code where it says choose better branch.

It's less obvious to answer the question whether or not there are overlapping subproblems. If we go back and look at the tree-- At first glance, it may appear that there are no overlapping subproblems. You'll notice that at each node we're solving a different problem. Right? The problem is described in some sense by this fortuple.

But if I've already taken A and C, and I have D left to consider, what should I do? Where up here I've taken A, and I have B, C, and D left to consider, what should I do? And by design of the decision tree, each node is different. Right? We're not considering the same thing over and over again. So you might look at it and say, well, we're out of luck. There are no overlapping subproblems. Not true.

So let's think hard for a second and ask in what sense are there overlapping subproblems? When are we potentially, at least, solving the same problem? Well what is the problem we're actually solving? We're solving the problem that can in some sense be stated as follows- at each node, we're saying what should we do about the nodes left to consider?

So we're solving a problem given a set of items-- what items we have at a node. Given an available weight, which items should we take? What's missing from this? It says nothing about the items you've already taken. In order to decide what to take next, we need to know how much weight we have available. But we don't need to know why we have that much weight available, which items we have previously decided to take.

That part of the for tuple turns out not to be relevant to the problem I still have to solve. Why is that important? Because there may be many different sets of items I could take. It would add up to the same total weight. Therefore, leaving me the same problem to solve.

And that's the key observation that tells us we can use dynamic programming to solve the 0-1 knapsack problem. Does that-- I should ask the positive question-- does that make sense to anybody? Raise your hand if it makes sense-- a small number of hands. Raise your hands if it doesn't make sense. OK. Can any of you to whom it doesn't make sense formulate a question? No, it's hard.

**AUDIENCE:** Total weight, or the same total value?

**PROFESSOR:** The same total weight. Because-- the question is are there many sets of items with the same total value or the same total weight? There might be many with the same total value, but that doesn't really matter. Because in choosing what to do next, as I go down the tree, I don't care what the value is above the tree. Because I'll still try and find the best solution I can to the remaining subproblem. If a value above is a million, or the value above is 3, it doesn't matter what I should do next.

**AUDIENCE:** Which makes sense. But if you use different things to get that weight, wouldn't you have a different set of items to choose from?

**PROFESSOR:** Right. So the question is it makes sense if there are many different sets of items that would have the same total weight, but what I'm going to-- but, wouldn't I have different items to choose from? Well you'll notice that I do have to look at what items I have left. But as I go through-- let's assume here that I have a list of items to take. As I go through from the front, say, I'll label each node of the tree. Each item will have either a 0 or a 1, depending upon whether I decided to take it or not.

And then what I have to consider is the remaining items. And let's say the values of the first two items-- the first four items-- were, well, let's say they were all one, just to make life simple. Well that'll be confusing, since I'm using that for taken or not taken. Let's say their values were all 2.

When I get here, and I'm deciding what to do with these items, I might-- so this tells me that I've used up 4 pounds of my allotment. Right? But if I had done this one, I would also use up 4 pounds of my allotment. And so, which of these I take or don't take is independent of how I got here. I do have to keep track of which items are still available to me. But I don't care how I've used up those 4 pounds. And obviously, there are a lot of ways to choose two that will use up 4 pounds here, right? But the solution to this part of the problem will be independent of which two I took. That make sense now?

Therefore, as long as there is a prefix of my list of items, such that multiple things add up to the same weight, I will have overlapping subproblems. Now you could imagine a situation in which no combination added to the same weight, in which case dynamic programming wouldn't buy me anything. It would still find the right answer, but it wouldn't speed anything up. But in practice, for most 0-1 knapsack problems-- they may not be this simple-- but you could expect-- and we'll see this complexity later-- that as long as your possible weights are being chosen from a relatively small set, you'll have many things add up to the same thing-- particularly as you get further down this list and have a lot longer prefix to consider.

So indeed I do have overlapping subproblems. That answer the question? All right, now. People feel better about what's going on? OK. Maybe looking at some code will make it even clearer, and maybe not. Anyway, I do appreciate the questions. Since it's sometimes hard for me to appreciate what's coming across and what's not coming across. You should get rewarded for good questions, as well as good answers.

All right, so now let's look at a dynamic programming solution. I've just taken the previously-- the example we looked at before was solve-- made it fast solve. And I've added a parameter of memo. This is the same kind of trick we used last time for our shortest path problem.

I should point out that I'm using-- this calls to mind a dirty, little secret of Python, one I have previously been hiding because it's so ugly I just hate to talk about it. But I figured at some point, honesty is the right policy. What I'd like is the first time I call fast solve, I shouldn't have to know that there's even a memo. Fast solve should have the same interface as solve. The items to consider and the available weight, and that's all you should need to know. Because the memo is part of the implementation, not part of the specification of solving the 0-1 knapsack problem. We've made that argument several times, earlier in the semester.

And so you might think that what I should therefore do is to initialize the memo to, say the empty dictionary. I'm going to use dictionaries for memos, as before. And then just check and if it's the empty one, that means it's the first-- I can know whether it's the-- it'll work fine for the first call. Well it doesn't work fine. And here's the dirty, little secret. In Python, when you have a parameter with a default value, that default value is evaluated once.

So the way the Python system works is it will process all of the def statements and evaluate the right-hand side here once. And then every time the function is called, without this optional argument, it will use the value it found when it evaluated the def statement. Now if this value is immutable, it doesn't matter. None will be none forever. 28 will be 28 forever. 3.7 will be 3.7 forever. But if this is a mutable value, for example a dict, then what will happen is it will create an object which will be initially an empty dictionary. But then every time fast solve is called without this argument, it will access the same object.

So if I call fast solve to solve one problem, and in the course of that, it builds up a big dictionary. And then I call it again to solve another problem, instead of starting with the empty dictionary, it will start with the same object it started with the first time, which is by now a dictionary filled with values. And so I will get the wrong answer. This is a subtlety, and it's a common kind of bug in Python. And I confess to having been bitten by it very recently, as in yesterday, myself. So it's worth remembering.

And there's a simple workaround, which is the default value is the immutable value. I chose none. And what I say when I enter it is if the memo is none, it means it's been invoked. And now I'll initialize it to the empty dictionary. And now every time it will get a new one, because we know what this statement says is allocate a new object of type dict, and initialize it to empty. This will happen dynamically when the thing is invoked, rather than statically at the time Python processes the diffs. It's a silly little problem. I hate to bring it up. I don't think it should work that way, but it does work that way. So we're stuck. Yeah.

[INAUDIBLE]

**PROFESSOR:** I'm sorry. You've have to speak more loudly.

[INAUDIBLE]

**PROFESSOR:** Well, why doesn't my first call-- So the question is when I go to test it, say, why don't I start by calling it with fast solve in an empty dictionary? And it's because, as we'll see when I get there, I want solve and fast solve, in this case, to have the same interface. And in fact, you can say I want them to have the same specification.

Because in fact, imagine that you wrote a program that called solve many times. And it was slow, and then you took 6.00-- So, I know why it's slow, because I've used a stupid solve. Let me use dynamic programming. I want to then be able to not name the new one fast solve, but replace the old solve by the new solve. And have all the programs that you solve still work. That will not happen if I insist that the new solve has an extra argument.

Now I could put what's called a wrapper in and call solve, and then have it call fast solve with the extra argument. And that would work too. That would be an equally good solution to address this problem. But does that make sense?

So either one would work. I chose this one but what wouldn't work in a practical engineering sense is to change the specification of solve. All right? So it's a silly little thing, but since people do get bitten by it, I figure I should tell you about it.

And I guess while I'm on the subject of silly little things, I'll tell you one more thing before we go back to this algorithm. Down here in the place where I'm testing it, you'll note that I've imported something called sys, and then said sys.setrecursionlimit to 2000. In Python, there's a maximum depth of recursion. When this is exceeded, it raises an exception-- depth exceeded. As it happens, the default value for that is some tiny number-- I forget what it is-- such that when I run this on an interesting size example, it crashes that exception.

So all I'm doing here is saying no, I know what I'm doing. It's OK to recurse to a greater depth. And I've set it to 2000 here, which is plenty for what I need. I could set it to 20,000. I could set it to 50,000. Again, if you're writing a serious program, you'll probably find that the default value for the depth of recursion is not enough, and you might want to reset it. I don't expect you to remember how to do this. I expect you to remember that it's possible. And if you need to do it, you can Google "recursion limit Python" and it will tell you how to change it. But again, I've found in testing this, the default was too small.

Ok. Let's go look at the code now for fast solve. It's got the items to consider, what's available, the available weight, and a memo. And I'm going to keep track of the number of calls. Just for pedagogical reasons, we'll want to review that later. OK, so I'll initialize the memo if it's the first time through. And then the first thing I'll do is I'm going to code what I have available essentially the way I've coded it here-- so what items are remaining by using an index. So my items are going to be a set of lists.

And I'm going to just sort of keep track of where I am in the list-- just march through exactly the way I'd marched through here. And so if the list of things to consider, the length of it-- because every time the length is the same, it will be the same list. Since I'm systematically examining a prefix. I'm not shuffling the list each time. So the length can be used to tell me what I've already looked at. So if I've looked at that sublist before, with this available weight, it will be in the memo.

And I'll just look up the solution-- result equals memo of [UNINTELLIGIBLE] to consider avail. So those will be my keys for my dictionary. The key will be a pair of essentially which items I have left to consider, which is coded by a number, and the amount of weight, another number. Or if there's nothing left to consider, or there's no weight available, then I'll return 0 and the empty tuple-- no value, didn't take anything. Otherwise, I'm now in the interesting case. If to consider subzero, the first one here, if the weight of that is greater than what I have available, then I know I can't take it. Right?

So I'll just lop it off and call fast solve recursively with the remaining list-- same old value of avail and the same old memo. Otherwise, well now I have an option of taking the first element in the list. I'll set the item to that element, and I'll consider taking it. So I'll call fast solve with to consider without the first item. But the amount of weight will be available now is what was available before minus the weight of that item. This is my left branch, if you will, where I decide to take the item. And so now I'm solving a smaller problem. The list is smaller, and I have less weight.

The next thing I'll do is consider not taking the item. So I'll call fast solve again with the remainder of the list. But avail and memo are not changed. That's the right branch. And you'll remember our decision tree the right branch avail and weight were never changed. Because I elected not to take that item. Then I'll just choose the better of the two alternatives, as before. And when I'm done, I'll update the memo with the solution to the problem I just solved.

All right? So it's exactly the same as the recursive solution you looked at a few weeks ago. Except I've added this notion of a memo to keep track of what we've already done. Well let's see how well it works. So I've got this test program. Just so things are repeatable, I'd set the C to 0 -- doesn't matter what I set it to, this just says instead of getting a random value each time, I'll get the same C so I'll get the same sequence of pseudo-random numbers-- makes it easier to debug. I'll have this global variable num calls. I'm arbitrarily setting the capacity to eight times the maximum weight.

So I'm saying the maximum value of an item is 10. The maximum weight is 10. This just says runs slowly is false, as in don't run the slow version. Run only the fast version. We'll come back to that. I set the capacity of the Knapsack to 8 times the max weight.

And then for the number of items-- and I'm just going to go through a different number of items, 4, 8 16, up to 1024-- I'm going to call build many items. Again, we've seen this program before. That just takes a number of items, a maximum value, and a maximum weight, and builds some set of items, choosing the values and weights at random from the ones we've offered it. This saves me. I wasn't going to type a set of 1024 items, I guarantee you that.

If it runs slowly, then I'm going to test it on both fast solve and solve. Notice that these are the names of the functions, not invocations of the functions. Because remember functions, like everything else in Python, are just objects. Otherwise, my tests will be only the tuple fast solve. And then for each function in tests, I'm going to set the number of calls to 0. I'm going to load up a timer, just because I'm curious how long it takes. And then I'll call the func. Notice again, getting back to what we talked about before, both fast solve and solve get called with the same arguments. I could not have done this trick if fast solve had required an extra parameter. And then I'll just see how well it did.

So let's start with this equal to true. So in that case we'll test both of them. And let's see how we do.

So it's chugging along. And we see that if I have 4 items, fast solve made 29 calls, took this much time. The good news, by the way, you'll notice that fast solve and solve gave me the same value each time. We would hope so, right? Because they're both supposed to find an optimal solution. They could find different sets of items. That would be OK. But they better add up to the same value.

But we seem to be stuck here. So you'll notice at 16, Solve made 131,000 calls, fast solve 3,500-- only took a third of a second. But we seem to be kind of stuck at-- actually, fast solve only made 1,200. At 32, fast solve made 3,500 calls. But solve seems to be stuck. Should this be a shocker that going from 16 to 32 made a big difference? Well remember here we're talking about 2 to the 16 possibilities to investigate, which is not a huge number. But now we're at 2 to the 32, which is quite a large number. And so we could wait quite a long time for solve to actually finish.

And in fact, we're not going to wait that long. We'll interrupt it.

So let's see how well fast solve works. We just call that. So we'll set this to false. It says don't run slowly, i.e. Don't call solve. So not only did we not get stuck at 32 items way up here, we didn't get stuck at 1024 items. Huge difference, right? Instead of only being able to process a set of 16, we can process a set of over 1,000. And in fact, I don't know how much higher we can go. I didn't try anything bigger. But even this-- big difference between getting stuck here and not getting stuck here. And in fact, not only did not get stuck, it was only two seconds.

So let's ask ourselves the question-- how fast is this growing? Not very fast-- more or less, what we see here-- and this is not something we can guarantee. But in this case, what we see is as we double the number of items, the number of calls kind of doubles as well. So we're actually growing pretty slowly. Well, OK that's good. But suppose we want to look at it a little bit more carefully and consider the question of what does govern the running time of fast solve. Well we know that looking things up in the dict is constant time. So that's OK. So what's going to govern it is every time we have to add an element to the memo, it's going to slow us down. Because you're gonna have to solve a problem.

Every time we can just look something up in the memo, we'll get our answer back instantaneously. So if we think about the running time, it's going to be related to how many different key value pairs might we have to compute for the memo. Well what governs that? So we know that the keys in the memo are pairs of to consider and avail. And so we know that the number of possible different keys will be the number of possible values in to consider, multiplied by the number of possible values of avail. Right?

How many possible values of to consider are there? What governs that? That's the easy question. Number of items, right? Because I'm just marching across this list, chopping off one item at a time. At most I can chop off n items, if the list is of length n.

Avail is more complicated. What does avail depend upon? It depends upon, well, the initial available weight. If the initial available weight is 0, well then it's gonna end really quickly. If it's really big, I might have to churn longer.

But it also depends-- and this is the key thing-- on the number of different weights, that sets of items can add up to. Because you'll recall the whole secret here was that we said that different sets of items could actually have the same total weight. And that's why we had this business of overlapping subproblems.

Now here, since I only had 10 possible weights, that tells me that I can't have very many different combinations. Right? That any set of say, one item has to be 0 through 10, or 0 through 9, I forget which it was. But either way, that means that all only have 10 different possible sums for sets of length 1. Whereas if my weights ranged over 100, I would have 100 different possibilities for sets of length 1. So that's an important factor in governing it.

So it's a complicated situation, but just to see what happens, try and remember how long this took for 1024. It took 2.2 seconds. I'm going to allow 20 different weights. And let's see what happens. It took roughly twice as long. So that's sort of what we would have expected. So again, we're going to see the running time is related to that.

So we can see that it's actually a fairly complicated thing. Suppose I did something really nasty and went up here where I built my items-- so here you'll see, when I built my items I chose the weight between 1 and max weight. And these were all integers-- ints. Suppose instead of that, I do this. So here after I choose my value between 1 and 10, I'm going to multiply it by some random number between 0 and 1. We've all seen random.random.

So now how many possible different weights am I going to have? A huge number, right? The number of random numbers, real floats between 0 and 1 is close to infinite, right? And so in fact, now I'm going to see that with reasonable probability, every single item will have a different weight. And so, things adding up to the same could still happen, even if they're all different. But it's not very probable.

So let's see what happens when I run it now. It's slowing down pretty quickly. Pretty much where the other one stopped right? Remember we got through 16 with Slowsolve but not through 32? Because effectively, I've now reduced the fast solve to the same as solve. It will never find anything in the memo, so it's doing exactly the same amount of work that solve did. And it will run very slowly.

OK, life is hard. But this is what you'd expect, because deep down we know the 0-1 knapsack problem is exponential, inherently exponential. And while I think dynamic programming is kind of miraculous how well it usually works, it's not miraculous in the liturgical sense, right? It's not actually performing a miracle in solving an exponential problem in linear time. It can in all things go bad. The exponential, and that's what it is here. This kind of algorithm is called pseudo-polynomial. It kind of runs in polynomial time, but not when things are really bad.

And I won't go into the details of pseudo-polynomial, but if any of you are in Course 6, you'll hear about it in 6.006 and 6.046. OK, we can adjourn. I'll see you all on Thursday. And we're not going to wait for this to finish, because it won't.