PROFESSOR: At the end of the lecture on Tuesday, a number of people asked me questions, asked Professor Grimson questions, which made it clear that I had been less than clear on at least a few things, so I want to come back and revisit a couple of the things we talked about at the end of the lecture.

You'll remember that I had drawn this decision tree, in part because it's an important concept I want you to understand, the concept of decision trees, and also to illustrate, I hope, visually, some things related to dynamic programming. So we had in that decision tree, is we had the weight vector, and I just given a very simple one [5,3,2], and we had a very simple value vector, [9,7,8]. And then the way we drew the tree, was we started at the top, and said all right, we're going to first look at item number 2, which was the third item in our list of items, of course, and say that we had five pounds left of weight that our knapsack that could hold, and currently had a value of 0. And then we made a decision, to not put that last item in the backpack, and said if we made that decision, the next item we had to consider was item 1, we still had five pounds available, and we still had a weight 0 available.

Now I, said the next item to consider is item 1, but really what I meant is, 1 and all of the items proceeding it in the list. This is my shorthand for saying the list up to and including items sub 1, kind of a normal way to think about it. And then we finish building the tree, left first step first, looking at all the no branches, 0,5,0 and then we were done, that was one branch. We then backed up, and said let's look at a yes, we'll include item number 1. Well, what happens here, if we've included that, it uses up all the available weight, and gave us the value of 9.

STUDENT: [UNINTELLIGIBLE]

PROFESSOR: Pardon?

STUDENT: -- want to be off the bottom branch.

PROFESSOR: Yup, Off by 1. Yeah, I wanted to come off this branch, because I've backtrack just 1, thank you. And then I backtrack up to this branch, and from here we got 0,2,7. And I'm not going to draw the rest of the tree for you here, because I drew it last time, and you don't need to see the whole tree. The point I wanted to make is that for every node, except the leaves, the leaves are the bottom of a tree in this case, computer scientists are weird, right, they draw trees where the root is at the top, and the leaves are at the bottom. And I don't know why, but since time immemorial that is the way computer scientists have drawn trees. That's why we're not biologists, I

guess. We don't understand these things.

But what I want you to notice is that for each node, except the leaves, the solution for that node can be computed from the solutions from it's children. So in order to look at the solution of this node, I choose one of the solutions of it's children, a or b, is the best solution if I'm here, and of course this is the better of the 2 solutions. If I look at this node, I get to choose its solution as the better of the solution for this node, and this node. All the way up to the top, where when I have to choose the best solution to the whole problem, it's either the best solution to the left node, or the best solution to the right node. This happens to be a binary decision tree. There's nothing magic about there being only two nodes, for the knapsack problem, that's just the way it works out, but there are other problems where there might be multiple decisions to make, more than a or yes or no, but it's always the case here that I have what we last time talked about as what? Optimal sub structure. As I defined it last time, it means that I can solve a problem by finding the optimal solution to smaller sub problems. Classic divide and conquer that we've seen over and over again in the term. Take a hard problem, say well, I can solve it by solving 2 smaller problems and combine their solution, and this case, the combining is choosing the best, it's a or b.

So then, I went directly from that way of thinking about the problem, to this straightforward, at the top of the slide here, also at the top of your handout, both yesterday and today, a straightforward implementation of max val, that basically just did this. And as you might have guessed, when you're doing this sort of thing, recursion is a very natural way to implement it. We then ran this, demonstrated that it got the right answer on problems that were small enough that we knew with the right answer was, ran it on a big problem, got what we hoped was the right answer, but we had no good way to check it in our heads, but noticed it took a long time to run. And then we asked ourselves, why did it take so long to run? And when we turned on the print statement, what we saw is because it was doing the same thing over and over again. Because we had a lot of the sub-problems were the same.

It was as if, when we went through this search tree, we never remembered what we got at the bottom, and we just re-computed things over and over. So that led us to look at memoization, the sort of key idea behind dynamic programming, which says let's remember the work we've done and not do it all over again. We used a dictionary to implement the memo, and that got us to the fast max val, which got called from max val 0, because I wanted to make sure I didn't change the specification of max val by introducing this memo that users shouldn't know even exists, because it's part of the implementation, not part of the problem statement. We did that, and all I did was take the original code and keep track of what I've done, and say have I computed this value before, if so, don't compute it again. And that's the key idea that you'll see over and over again as you solve problems with dynamic programming, is you say, have I already solved this problem, if so, let me look up the answer. If I haven't solved the problem, let me solve it, and store the answer away for later reference.

Very simple idea, and typically the beauty of dynamic programming as you've seen here, is not only is the idea simple, even the implementation is simple. There are a lot of complicated algorithmic ideas, dynamic programming is not one of them. Which is one of the reasons we teach it here. The other reason we teach it here, in addition to it being simple, is that it's incredibly useful. It's probably among the most useful ideas there is for solving complicated problems.

All right, now let's look at it. So here's the fast version, we looked at it last time, I'm not going to bore you by going through the details of it again, but we'll run it. This was the big example we looked at last time, where we had 30 items we could put in to choose from, so when we do it exponentially, it looks like it's 2 to the 30, which is a big number, but when we ran this, it found the answer, and it took only 1805 calls. Now I got really excited about this because, to me it's really amazing, that we've taken a problem that is apparently exponential, and solved it like that. And in fact, I could double the size of the items to choose from, and it would still run like. Eh - I'm not very good at snapping my fingers -- it would still run quickly. All right, so here's the question: have I found a way to solve an inherently exponential problem in linear time. Because what we'll see here, and we saw a little of this last time, as I double the size of the items, I only really roughly double the running time. Quite amazing. So have I done that? Well, I wish I had, because then I would be really famous, and my department head would give me a big raise, and all sorts of wonderful things would follow. But, I'm not famous, and I didn't solve that problem. What's going on?

Well this particular algorithm takes roughly, and I'll come back to the roughly question, order (n,s) time, where n is the number of items in the list and s, roughly speaking, is the size of the knapsack. We should also observe, that it takes order and s space. Because it's not free to store all these values. So at one level what I'm doing is trading time for space. It can run faster because I'm using some space to save things. So in this case, we had 30 items and the wait was 40, and, you know, this gives us 1200 which is kind of where we were. And I'm really emphasizing kind of here, because really what I'm using the available size for, is as a proxy for the number of items that can fit in the knapsack. Because the actual running time of this, and the actual space of this algorithm, is governed, interestingly enough, not by the size of the problem alone, but by the size of the solution. And I'm going to come back to that.

So how long it takes to run is related to how many items I end up being able to fit into the knapsack. If you think about it, this make sense. An entry is made in the memo whenever an item, and an available size pair is considered. As soon as the available size goes to 0, I know I can't enter any more items into the memo, right? So the number of items I have to remember is related to how many items I can fit in the knapsack. And of course, the amount of running time is exactly the number of things I have to remember, almost exactly, right? So you can see if you think about it abstractly, why the amount of work I have to do here will be proportional to the number of

items I can fit in, that is to say, the size of the solution.

This is not the way we'd like to talk about complexity. When we talk about the order, or big O, as we keep writing it, of a problem, we always prefer to talk about it in terms of the size of the problem. And that makes sense because in general we don't know the size of the solution until we've solved it. So we'd much rather define big O in terms of the inputs. What we have here is what's called a pseudo-polynomial algorithm. You remember a polynomial algorithm is an algorithm that's polynomial in the size of the inputs. Here we have an algorithm that's polynomial in the size of the solution, hence the qualifier pseudo. More formally, and again this is not crucial to get all the details on this, if we think about a numerical algorithm, a pseudo-polynomial algorithm has running time that's polynomial in the numeric value of the input. I'm using a numeric example because it's easier to talk about it that way. So you might to look at, say, an implementation of factorial, and say its running time is proportional to the numerical value of the number who's factorial. If I'm computing factorial of 8, I'll do 8 operations, Right Factorial of 10, I'll do 10 operations.

Now the key issue to think about here, is that as we look at this kind of thing, what we'll see is that, if we look at a numeric value, we know that that's exponential number in the number of digits. So that's the key thing to think about, Right That you can take a problem, and typically, when we're actually formally looking at computational complexity, big O, what we'll define the in terms of, is the size of the coding of the input. The number of bits required to represent the input in the computer. And so when we say something is exponential, we're talking about in terms of the number of bits required to represent it.

Now why am I going through all this, maybe I should use the word pseudo-theory? Only because I want you to understand that when we start talking about complexity, it can be really quite subtle. And you have to be very careful to think about what you mean, or you can be very surprised at how long something takes to run, or how much space it uses. And you have to understand the difference between, are you defining the performance in terms of the size of the problem, or the size of the solution. When you talk about the size of the problem, what do you mean by that, is it the length of an array, is it the size of the elements of the array, and it can matter. So when we ask you to tell us something about the efficiency, on for example a quiz, we want you to be very careful not to just write something like, order n squared, but to tell us what n is. For example, the number of elements in the list. But if you have a list of lists, maybe it's not just the number elements in the list, maybe it depends upon what the elements are. So just sort of a warning to try and be very careful as you think about these things, all right.

So I haven't done magic, I've given you a really fast way to solve a knapsack problem, but it's still exponential deep down in its heart, in something. All right, in recitation you'll get a chance to look at yet another kind of problem that can be solved by dynamic programming, there are many of them. Before we leave the knapsack problem though, I want to take a couple of minutes to look at a slight variation of the problem.

So let's look at this one. Suppose I told you that not only was there a limit on the total weight of the items in the knapsack, but also on the volume. OK, if I gave you a box of balloons, the fact that they didn't weight anything wouldn't mean you couldn't put, you could put lots of them in the knapsack, right? Sometimes it's the volume not the weight that matters, sometimes it's both. So how would we go about solving this problem if I told you not only was there a maximum weight, but there was a maximum volume. Well, we want to go back and attack it exactly the way we attacked it the first time, which was write some mathematical formulas. So you'll remember that when we looked at it, we said that the problem was to maximize the sum from i equals 1 to n, of p sub i, x sub i, maybe it should be 0 to n minus 1, but we won't worry about that. And we had to do it subject to the constraint that the sum from 1 to n of the weight sub i times x sub i, remember x is 0 if it was in, 1 if it wasn't, was less than or equal to the cost, as I wrote it this time, which was the maximum allowable weight. What do we do if we want to add volume, is an issue? Does this change? Does the goal change? You're answering. Answer out -- no one else can see you shake your head.

STUDENT: No.

PROFESSOR: No. The goal does not change, it's still the same goal. What changes?

STUDENT: The constraints.

PROFESSOR: Yeah, and you don't get another bar. The constraint has to change. I've added a constraint. And, what's the constraint I've added? Somebody else -- yeah?

STUDENT: You can't exceed the volume that the knapsack can hold.

PROFESSOR: Right, but can you state in this kind of formal way?

STUDENT: [INAUDIBLE]

PROFESSOR: -- sum from i equals 1 to n --

STUDENT: [INAUDIBLE]

PROFESSOR: Let's say v sub i, x sub i, is less than or equal to, we'll write k for the total allowable volume. Exactly. So the thing to notice here, is it's actually quite a simple little change we've made. I've simply added this one extra constraint, nice thing about thinking about it this way is it's easy to think about it, and what do you think I'll have to do if I want to go change the code? I'm not going to do it for you, but what would I think about doing when I change the code?

Well, let's look at the simple version first, because it's easier to look at. At the top. Well basically, all I'd have to do is go through and find every place I checked the constraint, and change it. To incorporate the new constraint. And when I went to the dynamic programming problem, what would I have to do, what would change? The memo would have to change, as well as the checks, right? Because now, I not only would have to think about how much weight did I have available, but I have to think about how much volume did I have available. So whereas before, I had a mapping from the item and the weight available, now I would have to have it from a tuple of the weight and the volume. Very small changes. That's one of the things I want you to sort of understand as we look at algorithms, that they're very general, and once you've figured out how to solve one problem, you can often solve another problem by a very straightforward reduction of this kind of thing. All right, any questions about that. Yeah?

STUDENT: I had a question about what you were talking about just before.

PROFESSOR: The pseudo-polynomial?

STUDENT: Yes.

PROFESSOR: Ok.

STUDENT: So, how do you come to a conclusion as to which you should use then, if you can determine the size based on solution, or based on input, so how do you decide?

PROFESSOR: Great question. So the question is, how do you choose an algorithm, why would I choose to use a pseudo-polynomial algorithm when I don't know how big the solution is likely to be, I think that's one way to think about it. Well, so if we think about the knapsack problem, we can look at it, and we can ask ourselves, well first of all we know that the brute force exponential solution is going to be a loser if the number of items is large. Fundamentally in this case, what I could look at is the ratio of the number of items to the size of the knapsack, say well, I've got lots items to choose from, I probably won't put them all in. But even if I did, it would still only be 30 of them, right? It's hard. Typically what we'll discover is the pseudo-polynomial algorithms are usually better, and in this case, never worse. So this will never be worse than the brute force one. if I get really unlucky, I end up checking the same number of things, but I'd have to be really, it'd have to be a very strange structure to the problem. So it's almost always the case that, if you can find a solution that uses dynamic programming, it will be better than the brute force, and certainly not, well, maybe use more space, but not use more time. But there is no magic, here, and so the question you asked is a very good question. And it's sometimes the case in real life that you don't know which is the better algorithm on the data you're actually going to be crunching. And you pays your money and you takes your chances, right? And if the data is not what you think it's going to be, you may be wrong in your choice, so you typically do have to spend some time thinking about it, what's the data going to actually look like. Very good question. Anything else?

All right, a couple of closing points before we leave this, things I would like you to remember. In dynamic programming, one of the things that's going on is we're trading time for space. Dynamic programming is not the only time we do that. We've solved a lot of problems that way, in fact, by trading time for space. Table look-up, for example, right, that if you're going to have trig tables, you may want to compute them all at once and then just look it up. So that's one thing. Two, don't be intimidated by exponential problems. There's a tendency for people to say, oh this problem's exponential, I can't solve it. Well, I solve 2 or 3 exponential problems before breakfast every day. You know things like, how to find my way to the bathroom is inherently exponential, but I manage to solve it anyway. Don't be intimidated. Even though it is apparently exponential, a lot of times you can actually solve it much, much faster.

Other issues. Three: dynamic programming is broadly useful. Whenever you're looking at a problem that seems to have a natural recursive solution, think about whether you can attack it with dynamic programming. If you've got this optimal substructure, and overlapping sub-problems, you can use dynamic programming. So it's good for knapsacks, it's good for shortest paths, it's good for change-making, it's good for a whole variety of problems. And so keep it in your toolbox, and when you have a hard problem to solve, one of the first questions you should ask yourself is, can I use dynamic programming? It's great for string-matching problems of a whole variety. It's hugely useful. And finally, I want you to keep in mind the whole concept of problem reduction. I started with this silly description of a burglar, and said : Well this is really the knapsack problem, and now I can go Google the knapsack problem and find code to solve it. Any time you can reduce something to a previously solved problem, that's good. And this is a hugely important lesson to learn. People tend not to realize that the first question you should always ask yourself, is this really just something that's well-known in disguise? Is it a shortest path problem? Is it a nearest neighbor problem? Is it what string is this most similar to problem? There are scores of well-understood problems, but only really scores, it's not thousands of them, and over time you'll build up a vocabulary of these problems, and when you see something in your domain, be it physics or biology or anything else, linguistics, the question you should ask is can I transform this into an existing problem? Ok, double line. If there are no questions I'm going to make a dramatic change in topic. We're going to temporarily get off of this more esoteric stuff, and go back to Python. And for the next, off and on for the next couple of weeks, we'll be talking about Python and program organization. And what I want to be talking about is modules of one sort, and of course that's because what we're interested in is modularity. How do we take a complex program, again, divide and conquer, I feel like a 1-trick pony, I keep repeating the same thing over and over again. Divide and conquer to make our programs modular so we can write them a little piece at a time and understand them a little piece at a time. Now I think of a module as a collection of related functions. We've already seen these, and we're going to refer to the functions using dot notation. We've been doing this all term, right, probably somewhere around lecture 2, we said import math, and then somewhere in our program we wrote something like math dot sqrt of 11, or

some other number. And the good news was we didn't have to worry about how math did square root or anything like that, we just got it and we used it. Now we have the dot notation to avoid name conflicts. Imagine, for example, that in my program I wrote something like import set, because somebody had written a module that implements mathematical sets, and somewhere else I'd written something like import table, because someone had something that implemented look-up tables of some sort, something like dictionaries, for example. And then, I wanted to ask something like membership. Is something in the set, or is something in the table? Well, what I would have written is something like table dot member. And then the element and maybe the table. And the dot notation was used to disambiguate, because I want the member operation from table, not the member one from set. This was important because the people who implemented table and set might never have met each other, and so they can hardly have been expected not to have used the same name somewhere by accident. Hence the use of the dot notation.

I now want to talk about a particular kind of module, and those are the modules that include classes or that are classes. This is a very important concept as we'll see, it's why MIT calls things like 6.00 subjects, so that they don't get confused with classes in Python, something we really need to remember here. Now they can be used in different ways, and they have been historically used in different ways. In this subject we're going to emphasize using classes in the context of what's called object-oriented programming. And if you go look up at Python books on the web, or Java books on the web, about 80% of them will include the word object-oriented in their title. Object-oriented Python programming for computer games, or who knows what else. And we're going to use this object-oriented programming, typically to create something called data abstractions. And over the next couple of days, you'll see what we mean by this in detail. A synonym for this is an abstract data type. You'll see both terms used on the web, and the literature etc., and think of them as synonyms. Now these ideas of classes, object-oriented programming, data abstraction, are about 40 years old, they're not new ideas. But they've only been really widely accepted in practice for 10 to 15 years. It was in the mid-70's, people began to write articles advocating this style of programming, and actually building programming languages, notably Smalltalk and Clue at MIT in fact, that provided linguistic support for the ideas of data abstraction and object-oriented programming. But it really wasn't until, I would say, the arrival of Java that object-oriented programming caught the popular attention. And then Java, C++ , Python of, course. And today nobody advocates a programming language that does not support it in some sort of way. So what is this all about? What is an object in object-oriented programming?

An object is a collection of data and functions. In particular functions that operate on the data, perhaps on other data as well. The key idea here is to bind together the data and the functions that operate on that data as a single thing. Now typically that's probably not the way you've been thinking about things. When you think about an int or a float or a dictionary or a list, you knew that there were functions that operated on them. But when you pass a parameter say, a list, you didn't think that you were not only passing the list, you were also passing the functions

that operate on the list. In fact you are. It often doesn't matter, but it sometimes really does. The advantage of that, is that when you pass an object to another part of the program, that part of the program also gets the ability to perform operations on the object. Now when the only types we're dealing with are the built-in types, the ones that came with the programming language, that doesn't really matter. Because, well, the programming language means everybody has access to those operations. But the key idea here is that we're going to be generating user-defined types, we'll invent new types, and as we do that we can't assume that if as we pass objects of that type around, that the programming language is giving us the appropriate operations on that type.

This combining of data and functions on that data is a very essence of object-oriented programming. That's really what defines it. And the word that's often used for that is encapsulation. Think of it as we got a capsule, like a pill or something, and in that capsule we've got data and a bunch of functions, which as we'll see are called methods. Don't worry, it doesn't matter that they're called methods, it's a historical artifact.

All right, so what's an example of this? Well, we could create a circle object, that would store a representation of the circle and also provide methods to operate on it, for example, draw the circle on the screen, return the area of the circle, inscribe it in a square, who knows what you want to do with it. As we talk about this, as people talk about this, in the context of our object-oriented programming, they typically will talk about it in terms of message pass, a message passing metaphor. I want to mention it's just a metaphor, just a way of thinking about it, it's not anything very deep here. So, the way people will talk about this, is one object can pass a message to another object, and the receiving object responds by executing one of its methods on the object. So let's think about lists. So if l is a list, I can call something like s dot sort, l dot sort. You've seen this. This says, pass the object l the message sort, and that message says find the method sort, and apply it to the object l, in this case mutating the object so that the elements are now in sorted order. If c is a circle, I might write something like c dot area. And this would say, pass to the object denoted by the variable c, the message area, which says execute a method called area, and in this case the method might return a float, rather than have a side-effect.

Now again, don't get carried away, I almost didn't talk about this whole message-passing paradigm, but it's so pervasive in the world I felt you needed to hear about it. But it's nothing very deep, and if you want to not think about messages, and just think oh, c has a method area, a circle has a method area, and c as a circle will apply it and do what it says, you won't get in any trouble at all.

Now the next concept to think about here, is the notion of an instance. So we've already thought about, we create instances of types, so when we looked at lists, and we looked at the notion of aliasing, we used the word instance, and said this is 1 object, this is another object, each of those objects is an instance of type list. So now that gets us to a class. A class is a collection of objects with characteristics in common. So you can think of class list. What is the characteristic that all objects of class list have in common, all instances of class list? it's the set of methods

that can be applied to lists. Methods like sort, append, other things. So you should think of all of the built-in types we've talked about as actually just built-in classes, like dictionaries, lists, etc. The beauty of being able to define your own class is you can now extend the language. So if, for example, you're in the business, God forbid, of writing financial software today, you might decide, I'd really like to have a class called tanking stock, or bad mortgage, or something like that or mortgage, right? Which would have a bunch of operations, like, I won't go into what they might be. But you'd like to write your program not in terms of floats and ints and lists, but in terms of mortgages, and CDOs, and all of the objects that you read about in the paper, the types you read about. And so you get to build your own special purpose programming language that helped you solve your problems in biology or finance or whatever, and we'll pick up here again on Tuesday.