The following content is provided under a Creative Commons license. Your support will help MIT OpenCourseWare continue to offer high quality educational resources for free. To make a donation or view additional materials from hundreds of MIT courses, visit MIT OpenCourseWare at ocw.mit.edu.

PROFESSOR: I want to take a few minutes at the start of today's lecture to wrap up a few more things about debugging. And then I'll move on to the main event. Just a few small points. First of all, keep in mind that the bug is probably not where you think it is. Because if a bug were there, you'd have already found it.

There are some simple things that you should always look at when you're looking for bugs. Reversed order of arguments. You might have a function that takes to floats and you just passed them in the wrong order. You may have noticed that when Professor Grimson and I used examples in class, we were pretty careful how we name our parameters. And I often end up using the same names for the actuals and the formals. And this just helps me not get confused about what order to do things in.

Spelling. These are just dumb little things to look for. Did you spell all the identifiers the way you think you did. The problem is, when you read code you see what you expect to see. And if you've typed in l when you should've typed a 1, or a 1 when you should've typed an l, you're likely to miss it. If you've made a mistake with upper and lower case. Things like that. So just be very careful looking at that.

Initialization. A very common bug is to initialize a variable. Go through a loop, and then forget to reinitialize it when it needs to be reinitialized again. So it's initialized outside the loop when it should be initialized inside the loop. Or conversely, inside the loop when it should be outside the loop. So look carefully at when variables are being initialized.

Object versus value equality. Particularly when you use double equals, are you asking whether you've got the same object or the same value? They mean very different things, as we have seen. Keep track of that.

Related to that is aliasing. And you'll see that, by the way, on the problem set that we're posting tomorrow. Where we're going to ask you to just look at some code, that there's some issues there with the fact that you alias things, maybe on purpose, maybe on accident. And by that I mean two different ways to get to the same value, the same object. Frequently that introduces bugs into the program.

A particular instance of that is deep versus shallow copy. When you decide to make a copy of something like a list, you have to think hard about are you only copying the top level of the list, but if it's, say, a list that contains a list, are you also getting a new copy of everything it contains? It doesn't matter if it's a list of integers or a list of strings or a list of tuples, but it matters a lot if it's a list of lists. Or a list of anything that could be mutable. So when you copy, what are you getting? When you copy a dictionary, for example. Think through all of that.

And again, a related problem that people run into is side-effects. You call a function and it returns a value, but maybe it, on purpose or on accident, modifies one of the actual parameters. So you could each make a list of things. Every experienced programmer over time develops a personal model of the mistakes they make. I know the kinds of things I get wrong. And so when I'm looking for a bug, I always say, oh, have you done this dumb thing you always do. So be a little bit introspective. Keep track of the mistakes you make. And if your program doesn't work, that should be your first guess.

Couple of other hints. Keep a record of what you've tried. People will look at things and they'll come in and that'll talk to a TA, and the TA will say, did you try this? And the person will say, I don't know. That leads you to end up doing the same thing over and over again. This gets back to my main theme of Tuesday, which is be systematic. Know what you've already tried, don't waste your time trying it again.

Think about reconsidering your assumptions. A huge one is, are you actually running the code you're looking at in your editor. This is a mistake I make all the time in Python. I sit there in idle, and I edit some code. I then click on the shell, rather than hitting F5, try and run it and say, it's not doing what I thought I should do. And I'll sit there staring at the output, staring at the code, not realizing that that output was not produced by that code. It's a mistake I've learned that I make. It's an easy one to make in Python. Lots of assumptions like that. You thought you knew what the built-in function sort did, method sort did. Well, does it do what you think it does? Does append do what you think it does? Go back and say, alright, I've obviously, some assumption is not right.

When you're debugging somebody else's code, debug the code, not the comments. I've often made the mistake of believing the comments somebody wrote in your code about what some function does. Sometimes it's good to believe it. But sometimes you have to read it.

Important thing. When it gets really tough, get help. Swallow your pride. Any one of my graduate students will tell you that from time to time I walk into their office and say, do you have a minute? And if they foolishly say yes, I drag them back to my office and say I'm stuck on this bug, what am I doing wrong? And it's amazing what -- well, a, they're smarter than I am which helps. But even if they weren't smarter than I am, just a fresh set of eyes. I've read through the same thing twenty times and I've missed something obvious. Someone who's never seen it before looks at and says, did you really mean to do this? Are you sure you didn't want to do for i in range of list, rather than for i in list? That sort of thing. Makes a big difference to just get that set of eyes.

And in particular, try and explain to somebody else what you think the program is doing. So I'll sit there with the student and I'll say, I'm going to try and explain to you why what I think this program is doing. And probably 80% of the time, halfway through my explanation, I say, oh, never mind. I'm embarrassed. And I send them away. Because just the act of explaining it to him or her has helped me understand it. So when you're really stuck, just get somebody. And try and explain to them why you think your program is doing what it's doing.

Another good thing to do when you're really stuck is walk away. Take a break. Come back and look at it with fresh eyes of your own. Effectively, what you're doing here is perhaps trading latency for efficiency. It may, if you go back, come back and two hours later, maybe you'll, it'll take you at least two hours to have found the bugs, because you've been away for two hours. And maybe if you'd stayed there and worked really hard you'd have found it an hour and fifty eight minutes. But is it really worth the two minutes? This is another reason, by the way, to start it long before it's due. That you actually have the leisure to walk away.

All right. What do you do when you've found the bug and you need to fix it? Remember the old saw, haste makes waste. I don't know this but I'll bet Benjamin Franklin said this. Don't rush into anything. Think about the fix, don't make the first change that comes to mind and see if it works. Ask yourself, will it fix all of the symptoms you've seen? Or if not, are the symptoms independent, will at least fix some of them? What are the ramifications of the proposed change. Will it break other things? That's a big issue. You can fix one thing, you break something else. So think through what this change might break.

Does it allow you to tidy up other things? This is important, I think, that code should not always grow. We all have a tendency to fix code by adding code. And the program just gets bigger and bigger and bigger. The more code you have, the harder it is to get it right. So, sometimes, what you need to, so you just pull back. And say, well, let me just tidy things up. That, by the way, is also a good debugging tool. Sometimes when I'm really stuck, I say, alright let me stop looking for the bug. Let me just clean up the code a little bit. Make my program prettier. And in the process of tidying it up and making it prettier, I'll often by accident find the bug. So it's a good trick to remember.

Finally, and this is very important, make sure that you can revert. There's nothing more frustrating then spending four hours debugging, and realizing at the end of four hours your program is worse than it was when you started. And you can't get back to where it was when you started.

So if you look at one of my directories, you'll find that I've been pretty anal about saving old versions. I can always pretty much get back to some place I've been. So if I've made a set of changes and I realize I've broken something that used to work, I can find a version of the code in which it used to work, and figure out what was going on there. Disk space is cheap. Don't hesitate to save your old versions. It's a good thing.

Alright, that's the end of my, maybe sermon is the right word on debugging. Polemic, I don't know. I hope it will be helpful. And I hope you'll remember some of these things as you try and get your programs to work.

I now want to get back to algorithms. And where we've sort of been for a while, and where we will stay for a while. More fundamentally, I want to get back to what I think of as the main theme of 6.00, which is taking a problem and finding some way to formulate the problem so that we can use computing to help us get an answer.

And in particular, we're going to spend the next few lectures looking at a class of problems known as optimization problems. In general, every optimization problem is the same. It has two parts. Some function that you're either attempting to maximize or minimize the value of. And these are duals. And some set of constraints that must be honored. Possibly an empty set of constraints.

So what are some of the classic optimization problems? Well one of the most well-known is the so-called shortest path problem. Probably almost every one of you has used a shortest path algorithm. For example, if you go to Mapquest, or Google Maps and ask how do I get from here to there. You give it the function, probably in this case to minimize, and it gives you a choice of functions. Minimize time, minimize distance.

And maybe you give it a set of constraints. I don't want to drive on highways. And it tries to find the shortest way, subject to those constraints, to get from Point A to Point B. And there are many, many other instances of this kind of thing. And tomorrow it's recitation, we'll spend quite a bit of time on shortest path problems.

Another classic optimization problem is the traveling salesman. Actually, I should probably, be modern, call it the traveling salesperson, the traveling salesperson problem. So the problem here, roughly, is given, a number of cities, and say the cost of traveling from city to city by airplane, what's the least cost round trip that you can find?

So you start at one place, you have to go to a number of other places. End up where you started. It's not quite the same as the shortest path, and figure out the way to do that that involves spending the least money. Or the least time, or something else.

What are some of the other classic optimization problems? There's bin packing. Filling up some container with objects of varying size and, perhaps, shape. So you've got the trunk of your car and you've got a bunch of luggage. More luggage than can actually fit in. And you're trying to figure out what order to put things in. And which ones you can put in. How to fill up that bin. Very important in shipping. People use bin packing algorithms to figure out, for example, how to load up container ships. Things of that nature. Moving vans, all sorts of things of that nature.

In biology and in natural language processing and many other things, we see a lot of sequence alignment problems. For example, aligning DNA sequences, or RNA sequences. And, one we'll spend a fair amount of time on today and Tuesday is the knapsack problem. In the old days people used to call backpacks knapsacks. So we old folks sometimes even still make that mistake. And the problem there is, you've got a bunch of things. More than will fit into the knapsack, and you're trying to figure out which things to take and which things to leave. As you plan your hiking trip. How much water should you take. How many blankets? How much food? And you're trying to optimize the value of the objects you can take subject to the constraint that the backpack is of finite size.

Now, why am I telling you all of this at this lightning speed? It's because I want you to think about it, going forward, about the issue of problem reduction. We'll come back to this. What this basically means is, you're given some problem to solve, that you've never seen before. And the first thing you do is ask is it an instance of some problem that other people have already solved?

So when the folks at Mapquest sat down to do their program, I guarantee you somebody opened an algorithms book and said, what have other people done to solve shortest path problems? I'll rely on fifty years of clever people rather than trying to invent my own. And so frequently what we try and do is, we take a new problem and map it onto an old problem so that we can use an old solution.

In order to be able to do that, it's nice to have in our heads an inventory of previously solved problems. To which we can reduce the current problem. So as we go through this semester, we'll look at, briefly or not so briefly, different previously solved problems in the hope that at some time in your future, when you have a problem to deal with, you'll say, I know that's really like shortest path, or really like graph coloring. Let me just take my problem and turn it into that problem, and use an existing solution.

So we'll start looking in detail at one problem, and that's the knapsack problem. Let's see. Where do I want to start? Oh yes, OK.

So far, we've been looking at problems that have pretty fast solutions. Most optimization problems do not have fast solutions. That is to say, when you're dealing with a large set of things, it takes a while to get the right answer. Consequently, you have to be clever about it.

Typically up till now, we've looked at things that can be done in sublinear time. Or, at worst, polynomial time. We'll now look at a problem that does not fall into that. And we'll start with what's called the continuous knapsack problem.

So here's the classic formulation. Assume that you are a burglar. And you have a backpack that holds, say, eight pounds' worth of stuff. And you've now broken into a house and you're trying to decide what to take. Well, let's assume in the continuous world, what you is you walk into the house and you see something like four pounds of gold dust. And you see three pounds of silver dust, and maybe ten pounds of raisins. And I don't actually know the periodic table entry for raisins. So I'll have to write it out.

Well, how would you solve this problem? First, let's say, what is the problem? How can we formulate it? Well, let's assume that what we want to do is, we have something we want to optimize. So we're looking for a function to maximize, in this case. What might that function be? Well, let's say it's some number of, some amount of, the cost of the value of gold. Times however many pounds of gold. Plus the cost of silver times however many - no, gold, is a g, isn't it. Pounds of silver, plus the cost of raisins times the number of pounds of raisins.

So that's the function I want to optimize. I want to maximize that function. And the constraint is what? It's that the pounds of gold plus the pounds of silver plus the pounds of raisins is no greater than eight. So I've got a function to maximize and a constraint that must be obeyed.

Now, the strategy here is pretty clear. As often is for the continuous problem. What's the strategy? I pour in the gold till I run out of gold. I pour in the silver until I run out of silver. And then I take as many raisins as will fit in and I leave. Right? I hope almost every one of you could figure out that was the right strategy. If not, you're not well suited to a life of crime.

What I just described is an instance of a greedy algorithm. In a greedy algorithm, at every step you do what maximizes your value at that step. So there's no planning ahead. You just do what's ever best. It's like when someone gets their food and they start by eating dessert. Just to make sure they get to the best part before they're full.

In this case, a greedy algorithm actually gives us the best possible solution. That's not always so. Now, you've actually all implemented a greedy algorithm. Or are in the process thereof. Where have we implemented a greedy algorithm, or have been asked to do a greedy algorithm? Well, there are not that many things you guys have been working on this semester. Yeah?

STUDENT: [INAUDIBLE]

PROFESSOR: Exactly right. So what you were doing there, it was a really good throw. But it was a really good answer you gave. So I'll forgive you the bad hands. You were asked to choose the word that gave you the maximum value. And then do it again with whatever letters you had left. Was that guaranteed to win? To give you the best possible scores? No. Suppose, for example, you had the letters this, doglets. Well, the highest scoring word might have been something like Doges, these guys used to rule Venice, but if you did that you'd been left with the letters l and t, which are kind of hard to use. So you've optimized the first step. But now you're stuck with something that's not very useful. Whereas in fact, maybe you would have been better to go with dog, dogs, and let.

So what we see here is an example of something very important and quite general. Which was that locally optimal decisions do not always lead to a global optimums. So you can't just repeatedly do the apparently local thing and expect to necessarily get to it.

Now, as it happens with the continuous knapsack problem as we've formulated it, greedy is good. But let's look for a slight variant of it, where greedy is not so good. And that's what's called the zero-one knapsack problem.

This is basically a discrete version of the knapsack problem. The formulation is that we have n items and at every step we have to either take the whole item or none of the item. In the continuous problem, the gold dust was assumed to be infinitely small. And so you could take as much of it as you wanted. Here it's as if you had gold bricks. You get to take the whole brick or no brick at all. Each item has a weight and a value, and we're trying to optimize it as before. So let's look at an example of a zero-one knapsack problem.

Again we'll go back to our burglar. So the burglar breaks into the house and finds the following items available. And you'll see in your handout a list of items and their value and how much, what they weight. Finds a watch, a nice Bose radio, a beautiful Tiffany vase, and a large velvet Elvis. And now this burglar finds, in fact, two of each of those. Person is a real velvet Elvis fan and needed two copies of this one.

Alright, and now he's trying to decide what to take. Well if the knapsack were large enough the thief would take it all and run, but let's assume that it can only hold eight pounds, as before. And therefore the thief has choices to make. Well, there are three types of thieves I want to consider: the greedy thief, the slow thief, and you.

We'll start with the greedy thief. Well, the greedy thief follows the greedy algorithm. What do you get if you follow the greedy algorithm? What's the first thing the thief does? Takes the most valuable item, which is a watch. And then what does he do after that? Takes another watch. And then? Pardon?

STUDENT: [INAUDIBLE]

PROFESSOR: And then?

STUDENT: [INAUDIBLE]

PROFESSOR: No. Not unless he wants to break the vase into little pieces and stuff it in the corners. The backpack is now full, right? There's no more room. So the greedy thief take that and leaves. But it's not an optimal solution. What should the thief have done? What's the best thing you can do? Instead of taking that one vase, the thief could take two radios. And get more value. So the greedy thief, in some sense, gets the wrong answer. But maybe isn't so dumb.

While greedy algorithms are not guaranteed to get you the right answer all the time, they're often very good to use. And what they're good about is, they're easy to implement. And they're fast to run. You can imagine coding the solution up and it's pretty easy. And when it runs, it's pretty fast. Just takes the most valuable, the next most valuable, the next most valuable, I'm done. And the thief leaves, and is gone. So that's a good thing.

On the other hand, it's often the case in the world that that's not good enough. And we're not looking for an OK solution, but we're looking for the best solution. Optimal means best. And that's what the slow thief does. So the slow thief thinks the following. Well, what I'll do is I'll put stuff in the backpack until it's full. I'll compute its value. Then I'll empty the backpack out, put another combination of stuff compute its value, try all possible ways of filling up the backpack, and then when I'm done, I'll know which was the best. And that's the one I'll do.

So he's packing and unpacking, packing and unpacking, trying all possible combinations of objects that will obey the constraint. And then choosing the winner. Well, this is like an algorithm we've seen before. It's not greedy. What is this? What category of algorithm is that? Somebody? Louder?

STUDENT: Brute force.

PROFESSOR: Brute force, exhaustive enumeration, exactly. We're exhausting all possibilities. And then choosing the winner. Well, that's what the slow thief tried. Unfortunately it took so long that before he finished the owner returned home, called the police and the thief ended up in jail. It happens. Fortunately, while sitting in jail awaiting trial, the slow thief decided to figure what was wrong. And, amazingly enough, he had studied mathematics. And had a blackboard in the cell. So he was able to work it out.

So he first said, well, let me try and figure out what I was really doing and why it took so long. So first, let's think about what was the function the slow thief was attempting to maximize. The summation, from i equals 1 to n, where n is the number of items, so I might label watch 1-0, watch 2-2, I don't care that they're both watches. They're two separate items. And then what I want to maximize is the sum of the price of item i times whether or not I took x i. So think of x as a vector of 0's and 1's. Hence the name of the problem.

If I'm going to keep that item, item i, if I'm going to take it, I give it a 1. If I'm not going to take it I give it a 0. And so I just take the value of that item times whether or not it's 0 or 1. So my goal here is to choose x such that this is maximized. Choose x such that that function is maximized, subject to a constraint. And the constraint, is it the sum from 1 to n of the weight of the item, times x i, is less than or equal to c, the maximum weight I'm allowed to put in my backpack. In this case it was eight.

So now I have a nice, tidy mathematical formulation of the problem. And that's often the first step in problem reduction. Is to go from a problem that has a bunch of words, and try and write it down as a nice, tight mathematical formulation. So now I know the formulation is to find x, the vector x, such that this constraint is obeyed and this function is maximized. That make sense to everybody? Any questions about that? Great.

So as the thief had thought, we can clearly solve this problem by generating all possible values of x and seeing which one solves this problem. But now the thief started scratching his head and said, well, how many possible values of x are there? Well, how can we think about that? Well, a nice way to think about that is to say, I've got a vector. So there's the vector with eight 0's in it. Three, four, five, six, seven, eight. Indicating I didn't take any of the items.

There's the vector, and again you'll see this in your handout, says I only took the first item. There's the one that says I only took the second item. There's the one that says I took the first and the second item. And at the last, I have all 1's. This series look like anything familiar to you? These are binary numbers, right? Eight digit binary numbers. Zero, one, two, three. What's the biggest number I can represent with eight of these? Somebody?

Well, suppose we had decimal numbers. And I said I'm giving you three decimal digits. What's the biggest number you can represent with three decimal digits? Pardon?

STUDENT: [INAUDIBLE]

PROFESSOR: Right. Which is roughly what? 10 to the?

STUDENT: [INAUDIBLE]

PROFESSOR: Right. Yes.

STUDENT: [INAUDIBLE]

PROFESSOR: Right. Exactly right. So, in this case it's 2 to the 8th. Because I have eight digits. But exactly right. More generally, it's 2 to the n. Where n is the number of possible items I have to choose from. Well, now that we've figured that out, what we're seeing is that the brute force algorithm is exponential. In the number of items we have to choose from. Not in the number that we take, but the number we have to think about taking.

Exponential growth is a scary thing. So now we can look at these two graphs, look at the top one. So there, we've compared n squared, quadratic growth, which by the way Professor Grimson told you was bad, to, really bad, which is exponential growth. And in fact, if you look at the top figure it looks as exponential or, quadratic isn't even growing at all. You see how really fast exponential growth is?

You get to fifteen items and we're up at seventy thousand already and counting. The bottom graph has exactly the same data. But what I've done is, I've use the logarithmic y-axis. Later in the term, we'll spend quite a lot of time talking about how do we visualize data. How do we make sense of data. I've done that because you can see here that the quadratic one is actually growing. It's just growing a lot more slowly. So the moral here is simple one. Exponential algorithms are typically not useful. n does not have to get very big for exponential to fail.

Now, imagine that you're trying to pack a ship and you've got ten thousand items to choose from. 2 to the 10,000 is a really big number. So what we see immediately, and the slow thief decided just before being incarcerated for years and years, was that it wasn't possible to do it that way. He threw up his hands and said, it's an unsolvable problem, I should have been greedy, there's no good way to do this.

That gets us to the smart thief. Why is this thief smart? Because she took 600. And she learned that in fact there is a good way to solve this problem. And that's what we're going to talk about next. And that's something called dynamic programming.

A lot of people think this is a really hard and fancy concept, and they teach in advanced algorithms classes. And they do, but in fact as you'll see it's really pretty simple. A word of warning. Don't try and figure out why it's called dynamic programming. It makes no sense at all. It was invented by a mathematician called Bellman. And he was at the time being paid by the Defense Department to work on something else. And he didn't want them to know what he was doing. So he made up a name that he was sure they would have no clue what it meant. Unfortunately, we now have lived with it forever, so don't think of it as actually being anything dynamic particularly. It's just a name.

It's very useful, and why we spend time on it for solving a broad range of problems that on their surface are exponentially difficult. And, in fact, getting very fast solutions to them. The key thing in dynamic programming, and we'll return to both of these, is you're looking for a situation where there are overlapping sub-problems and what's called optimal substructure. Don't expect to know what these mean yet. Hopefully by the end of the day, Tuesday, they will both make great sense to you. Let's first look at the overlapping sub-problems example.

You have on your handout, a recursive implementation of Fibonacci. Which we've seen before. What I've done is I've augmented it with this global called num calls just so I can keep track of how many times it gets called. And let's see what happens if we call fib of 5.

Well, quite a few steps. What's the thing that you notice about this output? There's something here that should tip us off that maybe we're not doing this the most efficient way possible. I called fib with 5 and then it calls it with 4. And then that call calls fib with 3. So I'm doing 4, 3, 2, 2, 1, 0. And I'm doing 1, 2. Well, what we see is I'm calling fib a lot of times with the same argument. And it makes sense. Because I start with fib of 5, and then I have to do fib of 4 and fib of 3. Well, fib of 4 is going to also have to do a fib of 3 and a fib of 2 and a fib of 1, and a fib of 0. And then the fib of 3 is going to do a fib of 2 and a fib of 1 and a fib of 0. And so I'm doing the same thing over and over again. That's because I have what are called overlapping sub-problems. I have used divide and conquer, as we seen before, to recursively break it into smaller problems. But the smaller problem of fib of 4 and the smaller problem of fib of 3 overlap with each other. And that leads to a lot of redundant computation. And I've done fib of 5, which is a small number. If we look at some other things, for example, let's get rid of this. Let's try see fib of 10. Well, there's a reason I chose 10 rather than, say, 20.

Here fib got called 177 times. Roughly speaking, the analysis of fib is actually quite complex, of a recursive fib. And I won't go through it, but what you can see is it's more or less, it is in fact, exponential. But it's not 2 to the something. It's a more complicated thing. But it grows quite quickly. And the reason it does is because of this overlapping.

On Tuesday we'll talk about a different way to implement Fibonacci, where the growth will be much less dramatic. Thank you.