JOHN GUTTAG: All right, welcome to the 60002, or if you were in 600, the second half of 600. I'm John Guttag. Let me start with a few administrative things.

What's the workload? There are problem sets. They'll all be programming problems much in the style of 60001. And the goal-- really twofold. 60001 problem sets were mostly about you learning to be a programmer. A lot of that carries over. No one learns to be a programmer in half a semester. So a lot of it is to improve your skills, but also there's a lot more, I would say, conceptual, algorithmic material in 60002, and the problem sets are designed to help cement that as well as just to give you programming experience. Finger exercises, small things. If they're taking you more than 15 minutes, let us know. They really shouldn't, and they're generally designed to help you learn a single concept, usually a programming concept.

Reading assignments in the textbooks, I've already posted the first reading assignment, and essentially they should provide you a very different take on the same material we're covering in lectures and recitations. We've tried to choose different examples for lectures and from the textbooks for the most part, so you get to see things in two slightly different ways. There'll be a final exam based upon all of the above. All right, prerequisites-- experience writing object-oriented programs in Python, preferably Python 3.5.

Familiarity with concepts of computational complexity. You'll see even in today's lecture, we'll be assuming that. Familiarity with some simple algorithms.

If you took 60001 or you took the 60001 advanced standing exam, you'll be fine. Odds are you'll be fine anyway, but that's the safest way to do it. So the programming assignments are going to be a bit easier, at least that's what students have reported in the past, because they'll be more focused on the problem to be solved than on the actual programming. The lecture content, more abstract. The lectures will be-- and maybe I'm speaking euphemistically-- a bit faster paced. So hang on to your seats. And the course is really less about programming and more about dipping your toe into the exotic world of data science.

We do want you to hone your programming skills. There'll be a few additional bits of Python. Today, for example, we'll talk about lambda abstraction.

Inevitably, some comments about software engineering, how to structure your code, more emphasis in using packages. Hopefully it will go a little bit smoother than in the last problem set in 60001.

And finally, it's the old joke about programming that somebody walks up to a taxi driver in New York City and says, "I'm lost. How do I get to Carnegie Hall?" The taxi driver turns to the person and says, "practice, practice, practice." And that's really the only way to learn to program is practice, practice, practice.

The main topic of the course is what I think of as computational models. How do we use computation to understand the world in which we live? What is a model? To me I think of it as an experimental device that can help us to either understand something that has happened, to sort of build a model that explains phenomena we see every day, or a model that will allow us to predict the future, something that hasn't happened. So you can think of, for example, a climate change model. We can build models that sort of explain how the climate has changed over the millennia, and then we can build probably a slightly different model that might predict what it will be like in the future.

So essentially what's happening is science is moving out of the wet lab and into the computer. Increasingly, I'm sure you all see this-- those of you who are science majors-- an increasing reliance on computation rather than traditional experimentation. As we'll talk about, traditional experimentation is and will remain important, but now it has to really be supplemented by computation. We'll talk about three kinds of models-- optimization models, statistical models, and simulation models.

So let's talk first about optimization models. An optimization model is a very simple thing. We start with an objective function that's either to be maximized or minimized.

So for, example, if I'm going from New York to Boston, I might want to find a route by car or plane or train that minimizes the total travel time. So my objective function would be the number of minutes spent in transit getting from a to b.

We then often have to layer on top of that objective function a set of constraints, sometimes empty, that we have to obey. So maybe the fastest way to get from New York to Boston is to take a plane, but I only have $100 to spend. So that option is off the table. So I have the constraints there on the amount of money I can spend. Or maybe I have to be in Boston before 5:00 PM and while the bus would get me there for $15, it won't get me there before 5:00. And so maybe what I'm left with is driving, something like that. So objective function, something you're either minimizing or maximizing, and a set of constraints that eliminate some solutions. And as we'll see, there's an asymmetry here. We handle these two things differently.

We use these things all the time.

I commute to work using Waze, which essentially is solving-- not very well, I believe-- an optimization problem to minimize my time from home to here. When you travel, maybe you log into various advisory programs that try and optimize things for you. They're all over the place. Today you really can't avoid using optimization algorithm as you get through life.

Pretty abstract. Let's talk about a specific optimization problem called the knapsack problem. The first time I talked about the knapsack problem I neglected to show a picture of a knapsack, and I was 10 minutes into it before I realized most of the class had no idea what a knapsack was. It's what we old people used to call a backpack, and they used to look more like that than they look today. So the knapsack problem involves-- usually it's told in terms of a burglar who breaks into a house and wants to steal a bunch of stuff but has a knapsack that will only hold a finite amount of stuff that he or she wishes to steal. And so the burglar has to solve the optimization problem of stealing the stuff with the most value while obeying the constraint that it all has to fit in the knapsack.

So we have an objective function.

I'll get the most for this when I fence it. And a constraint, it has to fit in my backpack. And you can guess which of these might be the most valuable items here.

So here is in words, written words what I just said orally.

There's more stuff than you can carry, and you have to choose which stuff to take and which to leave behind.

I should point out that there are two variants of it. There's the 0/1 knapsack problem and the continuous. The 0/1 would be illustrated by something like this. So the 0/1 knapsack problem means you either take the object or you don't. I take that whole gold bar or I take none of it. The continuous or so-called fractional knapsack problem says I can take pieces of it. So maybe if I take in my gold bar and shaved it into gold dust, I then can say, well, the whole thing won't fit in, but I can fit in a path, part of it. The continuous knapsack problem is really boring. It's easy to solve. How do you think you would solve the continuous problem?

Suppose you had over here a pile of gold and a pile of silver and a pile of raisins, and you wanted to maximize your value. Well, you'd fill up your knapsack with gold until you either ran out of gold or ran out of space. If you haven't run out of space, you'll now put silver in until you run out of space. If you still haven't run out of space, well, then you'll take as many raisins as you can fit in. But you can solve it with what's called a greedy algorithm, and we'll talk much more about this as we go forward.

Where you take the best thing first as long as you can and then you move on to the next thing. As we'll see, the 0/1 knapsack problem is much more complicated because once you make a decision, it will affect the future decisions.

Let's look at an example, and I should probably warn you, if you're hungry, this is not going to be a fun lecture. So here is my least favorite because I always want to eat more than I'm supposed to eat. So the point is typically knapsack problems are not physical knapsacks but some conceptual idea. So let's say that I'm allowed 1,500 calories of food, and these are my options. I have to go about deciding, looking at this food-- and it's interesting, again, there's things showing up on your screen that are not showing up on my screen, but they're harmless, things like how my mouse works. Anyway, so I'm trying to take some fraction of this food, and it can't add up to more than 1,500 calories.

The problem might be that once I take something that's 1,485 calories, I can't take anything else, or maybe 1,200 calories and everything else is more than 300. So once I take one thing, it constrains possible solutions. A greedy algorithm, as we'll see, is not guaranteed to give me the best answer.

Let's look at a formalization of it. So each item is represented by a pair, the value of the item and the weight of the item.

And let's assume the knapsack can accommodate items with the total weight of no more than w. I apologize for the short variable names, but they're easier to fit on a slide. Finally, we're going to have a vector l of length n representing the set of available items. This is assuming we have n items to choose from. So each element of the vector represents an item.

So those are the items we have. And then another vector v is going to indicate whether or not an item was taken. So essentially I'm going to use a binary number to represent the set of items I choose to take. For item three say, if bit three is zero I'm not taking the item. If bit three is one, then I am taking the item. So it just shows I can now very nicely represent what I've done by a single vector of zeros and ones.

Let me pause for a second. Does anyone have any questions about this setup? It's important to get this setup because what we're going to see now depends upon that setting in your head. So I've kind of used mathematics to describe the backpack problem. And that's typically the way we deal with these optimization problems. We start with some informal description, and then we translate them into a mathematical representation. So here it is. We're going to try and find a vector v that maximizes the sum of V sub i times I sub i.

Now, remember I sub i is the value of the item. V sub i is either zero or one So if I didn't take the item, I'm multiplying its value by zero. So it contributes nothing to the sum. If I did take the item, I'm multiplying its value by one. So the value of the item gets added to the sum. So that tells me the value of V. And I want to get the most valuable V I can get subject to the constraint that if I look at the item's dot weight and multiply it by V, the sum of the weights is no greater than w. So I'm playing the same trick with the values of multiplying each one by zero or one, and that's my constraint.

Make sense?

All right, so now we have the problem formalized.

How do we solve it? Well, the most obvious solution is brute force. I enumerate all possible combinations of items; that is to say, I generate all subsets of the items that are available-- I don't know why it says subjects here, but we should have said items. Let me fix that. This is called the power set. So the power set of a set includes the empty subset. It includes the set that includes everything and everything in between. So subsets of size one, subsets of size two, et cetera. So now I've generated all possible sets of items. I can now go through and sum up the weights and remove all those sets that weigh more than I'm allowed. And then from the remaining combinations, choose any one whose value is the largest.

I say choose any one because there could be ties, in which case I don't care which I choose.

So it's pretty obvious that this is going to give you a correct answer. You're considering all possibilities and choosing a winner.

Unfortunately, it's usually not very practical. What we see here is that's what the power set is if you have 100 vec. Not very practical, right, even for a fast computer generating that many possibilities is going to take a rather long time. So kind of disappointing. We look at it and say, well, we got a brute force algorithm. It will solve the problem, but it'll take too long. We can't actually do it. 100 is a pretty small number, right. We often end up solving optimization problems where n is something closer to 1,000, sometimes even a million. Clearly, brute force isn't going to work.

So that raises the next question, are we just being stupid? Is there a better algorithm that I should have showed you? I shouldn't say we. Am I just being stupid? Is there a better algorithm that would have given us the answer? The sad answer to that is no for the knapsack problem. And indeed many optimization problems are inherently exponential. What that means is there is no algorithm that provides an exact solution to this problem whose worst case running time is not exponential in the number of items.

It is an exponentially hard problem.

There is no really good solution. But that should not make you sad because while there's no perfect solution, we're going to look at a couple of really very good solutions that will make this poor woman a happier person. So let's start with the greedy algorithm. I already talked to you about greedy algorithms. So it could hardly be simpler. We say while the knapsack is not full, put the best available item into the knapsack.

When it's full, we're done.

You do need to ask a question. What does best mean? Is the best item the most valuable? Is it the least expensive in terms of, say, the fewest calories, in my case? Or is it the highest ratio of value to units? Now, maybe I think a calorie in a glass of beer is worth more than a calorie in a bar of chocolate, maybe vice versa. Which gets me to a concrete example. So you're about to sit down to a meal. You know how much you value the various different foods. For example, maybe you like donuts more than you like apples. You have a calorie budget, and here we're going to have a fairly austere budget-- it's only one meal; it's not the whole day-- of 750 calories, and we're going to have to go through menus and choose what to eat. That is as we've seen a knapsack problem. They should probably have a knapsack solver at every

McDonald's and Burger King.

So here's a menu I just made up of wine, beer, pizza, burger, fries, Coke, apples, and a donut, and the value I might place on each of these and the number of calories that actually are in each of these. And we're going to build a program that will find an optimal menu.

And if you don't like this menu, you can run the program and change the values to be whatever you like.

Well, as you saw if you took 60001, we like to start with an abstract data type, like to organize our program around data abstractions. So I've got this class food. I can initialize things. I have a getValue, a getCost, density, which is going to be the value divided by the cost, and then a string representation. So nothing here that you should not all be very familiar with.

Then I'm going to have a function called buildMenu, which will take in a list of names and a list of values of equal length and a list of calories. They're all the same length. And it will build the menu.

So it's going to be a menu of tuples-- a menu of foods, rather. And I build each food by giving it its name, its value, and its caloric content. Now I have a menu.

Now comes the fun part. Here is an implementation of a greedy algorithm. I called it a flexible greedy primarily because of this key function over here.

So you'll notice in red there's a parameter called keyfunction. That's going to be-- map the elements of items to numbers. So it will be used to sort the items. So I want to sort them from best to worst, and this function will be used to tell me what I mean by best. So maybe keyfunction will just return the value or maybe it will return the weight or maybe it will return some function of the density. But the idea here is I want to use one greedy algorithm independently of my definition of best. So I use keyfunction to define what I mean by best.

So I'm going to come in. I'm going to sort it from best to worst. And then for i in range len of items sub copy-- I'm being good. I've copied it. That's why you sorted rather than sort. I don't want to have a side effect in the parameter. In general, it's not good hygiene to do that. And so for-- I'll go through it in order from best to worst. And if the value is less than the maximum cost, if putting it in would keep me under the cost or not over the cost, I put it in, and I just do that until I can't put anything else in.

So I might skip a few because I might get to the point where there's only a few calories left, and the next best item is over that budget but maybe further down I'll find one that is not over it and put it in. That's why I can't exit as soon as I reach-- as soon as I find an item that won't fit. And then I'll just return. Does this make sense? Does anyone have any doubts about whether this algorithm actually works?

I hope not because I think it does work.

Let's ask the next question.

How efficient do we think it is?

What is the efficiency of this algorithm?

Let's see where the time goes. That's the algorithm we just looked at. So I deleted the comment, so we'd have a little more room in the slide.

Who wants to make a guess? By the way, this is the question. So please go answer the questions. We'll see how people do. But we can think about it as well together.

Well, let's see where the time goes. The first thing is at the sort. So I'm going to sort all the items. And we heard from Professor Grimson how long the sort takes. See who remembers. Python uses something called timsort, which is a variant of something called quicksort, which has the same worst-case complexity as merge sort. And so we know that is n log n where n in this case would be the len of items.

So we know we have that.

Then we have a loop. How many times do we go through this loop?

Well, we go through the loop n times, once for each item because we do end up looking at every item.

And if we know that, what's the order?

AUDIENCE: [INAUDIBLE].

JOHN GUTTAG: N log n plus n-- I guess is order n log n, right? So it's pretty efficient. And we can do this for big numbers like a million.

Log of a million times a million is not a very big number.

So it's very efficient.

Here's some code that uses greedy.

Takes in the items, the constraint, in this case will be the weight, and just calls greedy, but with the keyfunction and prints what we have.

So we're going to test greedy. I actually think I used 750 in the code, but we can use 800. It doesn't matter. And here's something we haven't seen before. So used greedy by value to allocate and calls testGreedy with food, maxUnits and Food.getValue.

Notice it's passing the function. That's why it's not-- no closed parentheses after it. Used greedy to allocate.

And then we have something pretty interesting.

What's going on with this lambda?

So here we're going to be using greedy by density to allocate-- actually, sorry, this is greedy by cost. And you'll notice what we're doing is-- we don't want to pass in the cost, right, because we really want the opposite of the cost. We want to reverse the sort because we want the cheaper items to get chosen first. The ones that have fewer calories, not the ones that have more calories. As it happens, when I define cost, I defined it in the obvious way, the total number of calories. So I could have gone and written another function to do it, but since it was so simple, I decided to do it in line.

So let's talk about lambda and then come back to it. Lambda is used to create an anonymous function, anonymous in the sense that it has no name. So you start with the keyword lambda.

You then give it a sequence of identifiers and then some expression.

What lambda does is it builds a function that evaluates that expression on those parameters and returns the result of evaluating the expression. So instead of writing def, I have inline defined a function. So if we go back to it here, you can see that what I've done is lambda x one divided by Food.getCost of x.

Notice food is the class name here. So I'm taking the function getCost from the class food, and I'm passing it the parameter x, which is going to be what? What's the type of x going to be?

I can wait you out. What is the type of x have to be for this lambda expression to make sense?

Well, go back to the class food. What's the type of the argument of getCost?

What's the name of the argument to getCost? That's an easier question.

We'll go back and we'll look at it.

What's the type of the argument to getCost?

AUDIENCE: Food.

JOHN GUTTAG: Food. Thank you. So I do have-- speaking of food, we do have a tradition in this class that people who answer questions correctly get rewarded with food. Oh, Napoli would have caught that.

So it has to be of type food because it's self in the class food.

So if we go back to here, this x has to be of type food, right.

And sure enough, when we use it, it will be. Let's now use it. I should point out that lambda can be really handy as it is here, and it's possible to write amazing, beautiful, complicated lambda expressions. And back in the good old days of 6001 people learned to do that.

And then they learned that they shouldn't.

My view on lambda expressions is if I can't fit it in a single line, I just go right def and write a function definition because it's easier to debug. But for one-liners, lambda is great.

Let's look at using greedy. So here's this function testGreedy, takes foods and the maximum number of units.

And it's going to go through and it's going to test all three greedy algorithms.

And we just saw that, and then here is the call of it. And so I picked up some names and the values. This is just the menu we saw. I'm going to build the menus, and then I'm going to call testGreedys. So let's go look at the code that does this.

So here you have it or maybe you don't, because every time I switch applications Windows decides I don't want to show you the screen anyway.

This really shouldn't be necessary.

Keep changes. Why it keeps forgetting, I don't know. Anyway, so here's the code. It's all the code we just looked at. Now let's run it.

Well, what we see here is that we use greedy by value to allocate 750 calories, and it chooses a burger, the pizza, and the wine for a total of-- a value of 284 happiness points, if you will. On the other hand, if we use greedy by cost, I get 318 happiness points and a different menu, the apple, the wine, the cola, the beer, and the donut. I've lost the pizza and the burger.

I guess this is what I signed up for when I put my preferences on.

And here's another solution with 318, apple, wine-- yeah, all right. So I actually got the same solution, but it just found them in a different order. Why did it find them in a different order? Because the sort order was different because in this case I was sorting by density.

From this, we see an important point about greedy algorithms, right, that we used the algorithm and we got different answers.

Why do we have different answers?

The problem is that a greedy algorithm makes a sequence of local optimizations, chooses the locally optimal answer at every point, and that doesn't necessarily add up to a globally optimal answer. This is often illustrated by showing an example of, say, hill climbing. So imagine you're in a terrain that looks something like this, and you want to get to the highest point you can get. So you might choose as a greedy algorithm if you can go up, go up; if you can't go up, you stop. So whenever you get a choice, you go up. And so if I start here, I could right in the middle maybe say, all right, it's not up but it's not down either. So I'll go either left or right.

And let's say I go right, so I come to here. Then I'll just make my way up to the top of the hill, making a locally optimal decision head up at each point, and I'll get here and I'll say, well, now any place I go takes me to a lower point. So I don't want to do it, right, because the greedy algorithm says never go backwards. So I'm here and I'm happy. On the other hand, if I had gone here for my first step, then my next step up would take me up, up, up, I'd get to here, and I'd stop and say, OK, no way to go but down. I don't want to go down. I'm done. And what I would find is I'm at a local maximum rather than a global maximum.

And that's the problem with greedy algorithms, that you can get stuck at a local optimal point and not get to the best one. Now, we could ask the question, can I just say don't worry about a density will always get me the best answer? Well, I've tried a different experiment. Let's say I'm feeling expansive and I'm going to allow myself 1,000 calories.

Well, here what we see is the winner will be greedy by value, happens to find a better answer, 424 instead of 413.

So there is no way to know in advance. Sometimes this definition of best might work. Sometimes that might work. Sometimes no definition of best will work, and you can't get to a good solution-- you get to a good solution. You can't get to an optimal solution with a greedy algorithm.

On Wednesday, we'll talk about how do you actually guarantee finding an optimal solution in a better way than brute force. See you then.