JOHN GUTTAG: We ended the last lecture looking at greedy algorithms. Today I want to discuss the pros and cons of greedy. Oh, I should mention-- in response to popular demand, I have put the PowerPoint up, so if you download the ZIP file, you'll find the questions, including question 1, the first question, plus the code, plus the PowerPoint. We actually do read Piazza, and sometimes, at least, pay attention. We should pay attention all the time.

So what are the pros and cons of greedy? The pro-- and it's a big pro-- is that it's really easy to implement, as you could see. Also enormously important-- it's really fast. We looked at the complexity last time-- it was m log n-- quite quick.

The downside-- and this can be either a big problem or not a big problem-- is that it doesn't actually solve the problem, in the sense that we've asked ourselves to optimize something. And we get a solution that may or may not be optimal.

Worse-- we don't even know, in this case, how close to optimal it is. Maybe it's almost optimal, but maybe it's really far away. And that's a big problem with many greedy algorithms. There are some very sophisticated greedy algorithms we won't be looking at that give you a bound on how good the approximation is, but most of them don't do that.

Last time we looked at an alternative to a greedy algorithm that was guaranteed to find the right solution. It was a brute force algorithm. The basic idea is simple-- that you enumerate all possible combinations of items, remove the combination whose total units exceed the allowable weight, and then choose the winner from those that are remaining.

Now let's talk about how to implement it. And the way I want to implement it is using something called a search tree. There are lots of different ways to implement it. In the second half of today's lecture, you'll see why I happen to choose this particular approach.

So what is a search tree? A tree is, basically, a kind of graph. And we'll hear much more about graphs next week. But this is a simple form where you have a root and then children of the

root. In this particular form, research C, you have two children.

So we start with the root. And then we look at our list of elements to be considered that we might take, and we look at the first element in that list. And then we draw a left branch, which shows the consequence of choosing to take that element, and a right branch, which shows the consequences of not taking that element. And then we consider the second element, and so on and so forth, until we get to the bottom of the tree. So by convention, the left element will mean we took it, the right direction will mean we didn't take it.

And then we apply it recursively to the non-leaf children. The leaf means we get to the end, we've considered the last element to be considered. Nothing else to think about. When we get to the code, we'll see that, in addition to the description being recursive, it's convenient to write the code that way, too. And then finally, we'll choose the node that has the highest value that meets our constraints.

So let's look at an example. My example is I have my backpack that can hold a certain number of calories if you will. And I'm choosing between, to keep it small, a beer, a pizza, and a burger-- three essential food groups.

The first thing I explore on the left is take the beer, and then I have the pizza and the burger to continue to consider. I then say, all right, let's take the pizza. Now I have just the burger. Now I taste the burger. This traversal of this generation of the tree is called left-most depth-most.

So I go all the way down to the bottom of the tree. I then back up a level and say, all right, I'm now at the bottom. Let's go back and see what happens if I make the other choice at the one level up the tree. So I went up and said, well, now let's see what happens if I make a different decision, as in we didn't take the burger. And then I work my way-- this is called backtracking-- up another level.

I now say, suppose, I didn't take the piece of pizza. Now I have the beer only and only the burger to think about, so on and so forth, until I've generated the whole tree. You'll notice it will always be the case that the leftmost leaf of this tree has got all the possible items in it, and the rightmost leaf none.

And then I just check which of these leaves meets the constraint and what are the values. And if I compute the value and the calories in each one, and if our constraint was 750 calories, then I get to choose the winner, which is-- I guess, it's the pizza and the burger. Is that right?

The most value under 750.

That's the way I go through. It's quite a straightforward algorithm. And I don't know why we draw our trees with the root at the top and the leaves at the bottom. My only conjecture is computer scientists don't spend enough time outdoors.

Now let's think of the computational complexity of this process. The time is going to be based on the total number of nodes we generate. So if we know the number of nodes that are in the tree, we then know the complexity of the algorithm, the asymptotic complexity.

Well, how many levels do we have in the tree? Just the number of items, right? Because at each level of the tree we're deciding to take or not to take an item. And so we can only do that for the number of items we have.

So if we go back, for example, and we look at the tree-- not that tree, that tree-- and we count the number of levels, it's going to be based upon the total number of items. We know that because if you look at, say, the leftmost node at the bottom, we've made three separate decisions. So counting the root, it's n plus 1. But we don't care about plus 1 when we're doing asymptotic complexity. So that tells us how many levels we have in the tree.

The next question we need to ask is, how many nodes are there at each level? And you can look at this and see-- the deeper we go, the more nodes we have at each level. In fact, if we come here, we can see that the number of nodes at level i-- depth i of the tree-- is 2 to the i.

That makes sense if you remember last time we looked at binary numbers. We're saying we're representing our choices as either 0 or 1 for what we take. If we have n items to choose from, then the number of possible choices is 2 to the n, the size of the powerset. So that will tell us the number of nodes at each level.

So if there are n items, the number of nodes in the tree is going to be the sum from 0 to n of 2 to the i because we have that many levels. And if you've studied a little math, you know that's exactly 2 to the n plus 1. Or if you do what I do, you look it up in Wikipedia and you know it's 2 to the n plus 1.

Now, there's an obvious optimization. We don't need to explore the whole tree. If we get to a point where the backpack is overstuffed, there's no point in saying, should we take this next item? Because we know we can't. I generated a bunch of leaves that were useless because the weight was too high. So you could always abort early and say, oh, no point in generating

the rest of this part of the tree because we know everything in it will be too heavy.

Adding something cannot reduce the weight. It's a nice optimization. It's one you'll see we actually do in the code. But it really doesn't change the complexity. It's not going to change the worst-cost complexity.

Exponential, as we saw this, I think, in Eric's lecture, is a big number. You don't usually like 2 to the n. Does this mean that brute force is never useful? Well, let's give it a try. We'll look at some code. Here is the implementation.

So it's maxVal, toConsider, and avail. And then we say, if toConsider is empty or avail is 0-- avail is an index, we're going to go through the list using that to tell us whether or not we still have an element to consider-- then the result will be the tuple 0 and the empty tuple. We couldn't take anything. This is the base of our recursion. Either there's nothing left to consider or there's no available weight-- the Val, as the amount of weight, is 0 or toConsider is empty.

Well, if either of those are true, then we ask whether to consider * 0, the first element to look at. Is that cost greater than availability? If it is, we don't need to explore the left branch. because it means we can't afford to put that thing in the backpack, the knapsack. There's just no room for it.

So we'll explore the right branch only. The result will be whatever the maximum value is of toConsider of the remainder of the list-- the list with the first element sliced off-- and availability unchanged. So it's a recursive implementation, saying, now we only have to consider the right branch of the tree because we knew we couldn't take this element. It just weighs too much, or costs too much, or was too fattening, in my case.

Otherwise, we now have to consider both branches. So we'll set next item to toConsider of 0, the first one, and explore the left branch. On this branch, there are two possibilities to think about, which I'm calling withVal and withToTake.

So I'm going to call maxVal of toConsider of everything except the current element and pass in an available weight of avail minus whatever-- well, let me widen this so we can see the whole code. This is not going to let me widen this window any more. Shame on it. Let me see if I can get rid of the console. Well, we'll have to do this instead.

So we're going to call maxVal with everything except the current element and give it avail

minus the cost of that next item of toConsider sub 0. Because we know that the availability, available weight has to have that cost subtracted from it. And then we'll add to withVal next item dot getValue. So that's a value if we do take it.

Then we'll explore the right branch-- what happens if we don't take it? And then we'll choose the better branch. So it's a pretty simple recursive algorithm. We just go all the way to the bottom and make the right choice at the bottom, and then percolate back up, like so many recursive algorithms.

We have a simple program to test it. I better start a console now if I'm going to run it. And we'll testGreedys on foods. Well, we'll testGreedys and then we'll testMaxVal. So I'm building the same thing we did in Monday's lecture, the same menu. And I'll run the same testGreedys we looked at last time. And we'll see whether or not we get something better when we run the truly optimal one.

Well, indeed we do. You remember that last time and, fortunately, this time too, the best we did was a value of 318. But now we see we can actually get to 353 if we use the truly optimal algorithm. So we see it ran pretty quickly and actually gave us a better answer than we got from the greedy algorithm. And it's often the case. If I have time at the end, I'll show you an optimization program you might want to run that works perfectly fine to use this kind of brute force algorithm on.

Let's go back to the PowerPoint. So I'm just going through the code again we just ran. This was the header we saw-- toConsider, as the items that correspond to nodes higher up the tree, and avail, as I said, the amount of space. And again, here's what the body of the code loooked like, I took out the comments.

One of the things you might think about in your head when you look at this code is putting the comments back in. I always find that for me a really good way to understand code that I didn't write is to try and comment it. And that helps me sort of force myself to think about what is it really doing. So you'll have both versions-- you'll have the PowerPoint version without the comments and the actual code with the comments. You can think about looking at this and then looking at the real code and making sure that you're understanding jibes.

I should point out that this doesn't actually build the search tree. We've got this local variable result, starting here, that records the best solution found so far. So it's not the picture I drew where I generate all the nodes and then I inspect them. I just keep track-- as I generate a

node, I say, how good is this? Is it better than the best I've found so far? If so, it becomes the new best.

And I can do that because every node I generate is, in some sense, a legal solution to the problem. Probably rarely is it the final optimal solution but it's at least a legal solution. And so if it's better than something we saw before, we can make it the new best. This is very common. And this is, in fact, what most people do with it when they use a search tree-- they don't actually build the tree in the pictorial way we've looked at it but play some trick like this of just keeping track of their results. Any questions about this?

All right. We did just try it on example from lecture 1. And we saw that it worked great. It gave us a better answer. It finished quickly. But we should not take too much solace from the fact that it finished quickly because 2 to the eighth is actually a pretty tiny number. Almost any algorithm is fine when I'm working on something this small.

Let's look now at what happens if we have a bigger menu. Here is some code to do a bigger menu. Since, as you will discover if you haven't already, I'm a pretty lazy person, I didn't want to write out a menu with a 100 items or even 50 items. So I wrote some code to generate the menus. And I used randomness to do that.

This is a Python library we'll be using a lot for the rest of the semester. It's used any time you want to generate things at random and do many other things. We'll come back to it a lot. Here we're just going to use a very small part of it.

To build a large menu of some numItems-- and we're going to give the maximum value and the maximum cost for each item. We'll assume the minimum is, in this case, 1. Items will start empty. And then for i in range number of items, I'm going to call this function random dot randint that takes a range of integers from 1 to, actually in this case, maxVal minus 1, or 1 to maxVal, actually, in this case.

And it just chooses one of them at random. So when you run this, you don't know what it's going to get. Random dot randint might return 1, it might return 23, it might return 54. The only thing you know is it will be an integer.

And then I'm going to build menus ranging from 5 items to 60 items-- buildLargeMenu, the number of items, with maxVal of 90 and a maxCost of 250, pleasure and calories. And then I'm going to test maxVal on each of these menus. So building menus of various sizes at

random and then just trying to find the optimal value for each of them.

Let's look at the code. Let's comment this out, we don't need to run that again. So we'll build a large menu and then we'll try it for a bunch of items and see what we get. So it's going along. Trying the menu up to 30 went pretty quickly. So even 2 to the 30 didn't take too long. But you might notice it's kind of bogging down, we got 35.

I guess, I could ask the question now-- it was one of the questions I was going to ask as a poll but maybe I won't bother-- how much patience do we have? When do you think we'll run out of patience and quit? If you're out of patience, raise your hand. Well, some of you are way more patient than I am. So we're going to quit anyway.

We were trying to do 40. It might have finished 40, 45. I've never waited long enough to get to 45. It just is too long. That raises the question, is it hopeless? And in theory, yes. As I mentioned last time, it is an inherently exponential problem.

The answer is-- in practice, no. Because there's something called dynamic programming, which was invented by a fellow at the RAND Corporation called Richard Bellman, a rather remarkable mathematician/computer scientist. He wrote a whole book on it, but I'm not sure why because it's not that complicated.

When we talk about dynamic programming, it's a kind of a funny story, at least to me. I learned it and I didn't know anything about the history of it. And I've had all sorts of theories about why it was called dynamic programming. You know how it is, how people try and fit a theory to data. And then I read a history book about it, and this was Bellman's own description of why he called it dynamic programming.

And it turned out, as you can see, he basically chose a word because it was the description that didn't mean anything. Because he was doing mathematics, and at the time he was being funded by a part of the Defense Department that didn't approve of mathematics. And he wanted to conceal that fact. And indeed at the time, the head of Defense Appropriations in the US Congress didn't much like mathematics. And he was afraid that he didn't want to have to go and testify and tell people he was doing math. So he just invented something that no one would know what it meant. And years of students spent time later trying to figure out what it actually did mean.

Anyway, what's the basic idea? To understand it I want to temporarily abandon the knapsack

problem and look at a much simpler problem-- Fibonacci numbers. You've seen this already, with cute little bunnies, I think, when you saw it. N equals 0, n equals 1-- return 1. Otherwise, fib of n minus 1 plus fib of n minus 2.

And as I think you saw when you first saw it, it takes a long time to run. Fib of 120, for example, is a very big number. It's shocking how quickly Fibonacci grows. So let's think about implementing it.

If we run Fibonacci-- well, maybe we'll just do that. So here is fib of n, let's just try running it. And again, we'll test people's patience. We'll see how long we're letting it run. I'm going to try for i in the range of 121. We'll print fib of i. Comes clumping along.

It slows down pretty quickly. And if you look at it, it's kind of surprising it's this slow because these numbers aren't that big. These are not enormous numbers. Fib of 35 is not a huge number. Yet it took a long time to compute. So you have the numbers growing pretty quickly but the computation, actually, seems to be growing faster than the results. We're at 37.

It's going to gets slower and slower, even though our numbers are not that big. The question is, what's going on? Why is it taking so long for Fibonacci to compute these results? Well, let's call it and look at the question. And to do that I want to look at the call tree.

This is for Fibonacci of 6, which is only 13, which, I think, most of us would agree was not a very big number. And let's look what's going on here. If you look at this, what in some sense seems really stupid about it? What is it doing that a rational person would not want to do if they could avoid it?

It's bad enough to do something once. But to do the same thing over and over again is really wasteful. And if we look at this, we'll see, for example, that fib 4 is being computed here, and fib 4 is being computed here. Fib 3 is being considered here, and here, and here. And do you think we'll get a different answer for fib 3 in one place when we get it in the other place? You sure hope not.

So you think, well, what should we do about this? How would we go about avoiding doing the same work over and over again? And there's kind of an obvious answer, and that answer is at the heart of dynamic programming. What's the answer?

AUDIENCE: [INAUDIBLE]

JOHN GUTTAG: Exactly. And I'm really happy that someone in the front row answered the question because I can throw it that far. You store the answer and then look it up when you need it. Because we know that we can look things up very quickly. Dictionary, despite what Eric said in his lecture, almost all the time works in constant time if you make it big enough, and it usually is in Python. We'll see later in the term how to do that trick.

So you store it and then you'd never have to compute it again. And that's the basic trick behind dynamic programming. And it's something called memoization, as in you create a memo and you store it in the memo. So we see this here. Notice that what we're doing is trading time for space. It takes some space to store the old results, but negligible related to the time we save.

So here's the trick. We're going to create a table to record what we've done. And then before computing fib of x, we'll check if the value has already been computed. If so, we just look it up and return it. Otherwise, we'll compute it-- it's the first time-- and store it in the table.

Here is a fast implementation of Fibonacci that does that. It looks like the old one, except it's got an extra argument-- memo-- which is a dictionary. The first time we call it, the memo will be empty. It tries to return the value in the memo. If it's not there, an exception will get raised, we know that. And it will branch to here, compute the result, and then store it in the memo and return it. It's the same old recursive thing we did before but with the memo.

Notice, by the way, that I'm using exceptions not as an error handling mechanism, really, but just as a flow of control. To me, this is cleaner than writing code that says, if this is in the keys, then do this, otherwise, do that. It's slightly fewer lines of code, and for me, at least, easier to read to use try-except for this sort of thing.

Let's see what happens if we run this one. Get rid of the slow fib and we'll run fastFib. Wow. We're already done with fib 120. Pretty amazing, considering last time we got stuck around 40. It really works, this memoization trick. An enormous difference.

When can you use it? It's not that memorization is a magic bullet that will solve all problems. The problems it can solve, it can help with, really, is the right thing. And by the way, as we'll see, it finds an optimal solution, not an approximation.

Problems have two things called optimal substructure, overlapping subproblems. What are these mean? We have optimal substructure when a globally optimal solution can be found by combining optimal solutions to local subproblems. So for example, when x is greater than 1 we

can solve fib x by solving fib x minus 1 and fib x minus 2 and adding those two things together. So there is optimal substructure-- you solve these two smaller problems independently of each other and then combine the solutions in a fast way.

You also have to have something called overlapping subproblems. This is why the memo worked. Finding an optimal solution has to involve solving the same problem multiple times. Even if you have optimal substructure, if you don't see the same problem more than once-- creating a memo. Well, it'll work, you can still create the memo. You'll just never find anything in it when you look things up because you're solving each problem once. So you have to be solving the same problem multiple times and you have to be able to solve it by combining solutions to smaller problems.

Now, we've seen things with optimal substructure before. In some sense, merge sort worked that way-- we were combining separate problems. Did merge sort have overlapping subproblems? No, because-- well, I guess, it might have if the list had the same element many, many times. But we would expect, mostly not.

Because each time we're solving a different problem, because we have different lists that we're now sorting and merging. So it has half of it but not the other. Dynamic programming will not help us for sorting, cannot be used to improve merge sort. Oh, well, nothing is a silver bullet.

What about the knapsack problem? Does it have these two properties? We can look at it in terms of these pictures. And it's pretty clear that it does have optimal substructure because we're taking the left branch and the right branch and choosing the winner.

But what about overlapping subproblems? Are we ever solving, in this case, the same problem-- add two nodes? Well, do any of these nodes look identical? In this case, no. We could write a dynamic programming solution to the knapsack problem-- and we will-- and run it on this example, and we'd get the right answer. We would get zero speedup. Because at each node, if you can see, the problems are different.

We have different things in the knapsack or different things to consider. Never do we have the same contents and the same things left to decide. So "maybe" was not a bad answer if that was the answer you gave to this question.

But let's look at a different menu. This menu happens to have two beers in it. Now, if we look

at what happens, do we see two nodes that are solving the same problem? The answer is what? Yes or no? I haven't drawn the whole tree here.

Well, you'll notice the answer is yes. This node and this node are solving the same problem. Why is it? Well, in this node, we took this beer and still had this one to consider. But in this node, we took that beer but it doesn't matter which beer we took. We still have a beer in the knapsack and a burger and a slice to consider. So we got there different ways, by choosing different beers, but we're in the same place. So in fact, we actually, in this case, do have the same problem to solve more than once.

Now, here I had two things that were the same. That's not really necessary. Here's another very small example. And the point I want to make here is shown by this. So here I have again drawn a search tree. And I'm showing you this because, in fact, it's exactly this tree that will be producing in our dynamic programming solution to the knapsack problem.

Each node in the tree starts with what you've taken-- initially, nothing, the empty set. What's left, the total value, and the remaining calories. There's some redundancy here, by the way. If I know what I've taken, I could already always compute the value and what's left. But this is just so it's easier to see. And I've numbered the nodes here in the order in which they're get generated.

Now, the thing that I want you to notice is, when we ask whether we're solving the same problem, we don't actually care what we've taken. We don't even care about the value. All we care is, how much room we have left in the knapsack and which items we have left to consider. Because what I take next or what I take remaining really has nothing to do with how much value I already have because I'm trying to maximize the value that's left, independent of previous things done.

Similarly, I don't care why I have a 100 calories left. Whether I used it up on beers or a burger, doesn't matter. All that matters is that I just have 100 left. So we see in a large complicated problem it could easily be a situation where different choices of what to take and what to not take would leave you in a situation where you have the same number of remaining calories. And therefore you are solving a problem you've already solved.

At each node, we're just given the remaining weight, maximize the value by choosing among the remaining items. That's all that matters. And so indeed, you will have overlapping subproblems.

As we see in this tree, for the example we just saw, the box is around a place where we're actually solving the same problem, even though we've made different decisions about what to take, A versus B. And in fact, we have different amounts of value in the knapsack-- 6 versus 7. What matters is we still have C and D to consider and we have two units left.

It's a small and easy step. I'm not going to walk you through the code because it's kind of boring to do so. How do you modify the maxVal we looked at before to use a memo? First, you have to add the third argument, which is initially going to be set to the empty dictionary. The key of the memo will be a tuple-- the items left to be considered and the available weight.

Because the items left to be considered are in a list, we can represent the items left to be considered by how long the list is. Because we'll start at the front item and just work our way to the end. And then the function works, essentially, exactly the same way fastFib worked. I'm not going to run it for you because we're running out of time. You might want to run it yourself because it is kind of fun to see how really fast it is.

But more interestingly, we can look at this table. This column is what we would get with the original recursive implementation where we didn't use a memo. And it was therefore 2 to the length of items. And as you can see, it gets really big or, as we say at the end, huge.

But the number of calls grows incredibly slowly for the dynamic programming solution. In the beginning it's worth Oh, well. But by the time we get to the last number I wrote, we're looking at 43,000 versus some really big number I don't know how to pronounce-- 18 somethings. Incredible improvement in performance. And then at the end, it's a number we couldn't fit on the slide, even in tiny font. And yet, only 703,000 calls.

How can this be? We know the problem is inherently exponential. Have we overturned the laws of the universe? Is dynamic programming a miracle in the liturgical sense? No. But the thing I want you to carry away is that computational complexity can be a very subtle notion.

The running time of fastMaxVal is governed by the number of distinct pairs that we might be able to use as keys in the memo-- toConsider and available. The number of possible values of toConsider is small. It's bounded by the length of the items. If I have a 100 items, it's 0, 1, 2, up to a 100.

The possible values of available weight is harder to characterize. But it's bounded by the

number of distinct sums of weights you can get. If I start with 750 calories left, what are the possibilities? Well, in fact, in this case, maybe we can take only 750 because we're using with units. So it's small. But it's actually smaller than that because it has to do with the combinations of ways I can add up the units I have.

I know this is complicated. It's not worth my going through the details in the lectures. It's covered in considerable detail in the assigned reading.

Quickly summarizing lectures 1 and 2, here's what I want you to take away. Many problems of practical importance can be formulated as optimization problems. Greedy algorithms often provide an adequate though often not optimal solution. Even though finding an optimal solution is, in theory, exponentially hard, dynamic programming really often yields great results. It always gives you a correct result and it's sometimes, in fact, most of the times gives it to you very quickly.

Finally, in the PowerPoint, you'll find an interesting optimization problem having to do with whether or not you should roll over problem that grades into a quiz. And it's simply a question of solving this optimization problem.