6.00 Introduction to Computer Science and Programming, Fall 2008

MIT OpenCourseWare
http://ocw.mit.edu

6.00 Introduction to Computer Science and Programming, Fall 2008
Transcript – Lecture 13

PROFESSOR: OK. I want to start where we left off. You remember last time we were looking at Fibonacci. And so we've got it up here, a nice little recursive implementation of it. And the thing I wanted to point out is, we've got this global variable number of calls. Which is there not because Fibonacci needs it but just for pedagogical reasons, so that we can keep track of how much work this thing is doing. And I've turned on a print statement which was off last time. So we can see what it's doing is it runs. So let's try it here with Fib of 6.

So, as we would hope, Fib of 6 happens to be 8. That right? That right, everybody? Should Fib of 6 be 8? I don't think so. So first thing we should do is scratch our heads and see what's going on here. Alright, let's look at it. What's happening here? This is your morning wake-up call. What is happening? Yes.

STUDENT: [INAUDIBLE]

PROFESSOR: See if I can get it all the way to the back. No I can't. That's embarrassing. Alright. So if n is less than or equal to 1, return n. Well that's not right, right? What should I be doing there? Because Fib of is? What? 1. So let's fix it. How about that, right? Or maybe, what would be even simpler than that? Maybe I should just do that.

Now let's try it. We feel better about this? We like that answer? Yes, no? What's the answer, guys? What should Fibonacci of 6 be? I think that's the right answer, right? OK.

So what do I want you to notice about this? I've computed a value which is 13. And it's taken me 25 calls. 25 recursive calls to get there. Why is it taking so many? Well, what we can see here is that I'm computing the same value over and over again. Because if we look at the recursive structure of the program, what we'll see that's going on here is, I call Fib of 5 and 4, but then Fib of 5 is also going to call Fib of 4. So I'm going to be computing it on those branches. And then it gets worse and worse as I go down.

So if I think about computing Fib of I'm going to be computing that a lot of times. Now, fortunately, Fib of 0 is short. But the other ones are not so short. And so what I see is that as I run this, I'm doing a lot of redundant computation. Computing values whose answer I should already know. That's, you'll remember last time, I talked about the notion of overlapping sub-problems. And that's what we have here. As with many recursive algorithms, I solve a bigger problem by solving a smaller instance of the original problem. But here there's overlap. The instant unlike binary

search, where each instance was separate, here the instances overlap. They share something in common. In fact, they share quite a lot in common. That's not unusual.

That will lead me to use a technique I mentioned again last time, called memoization. Effectively, what that says is, we record a value the first time it's computed, then look it up the subsequent times we need it.

So it makes sense. If I know I'm going to need something over and over again, I squirrel it away somewhere and then get it back when I need it. So let's look at an example of that.

So I'm going to have something called fast Fib. But first I'm going to have, let's look at what fast Fib does and then we'll come back to the next question. It takes the number whose Fibonacci I want plus a memo. And the memo will be a dictionary that maps me from a number to Fib of that number.

So what I'm going to do, well, let's get rid of this print statement for now. I'm going to say, if n is not in memo. Remember the way dictionary works, this is the key. Is the key of a value. Then I'll call fast Fib recursively, with n minus 1 in memo, and n minus 2 in memo. Otherwise I'll return the memo.

Well, let's look at it for a second. This is the basic idea. But do we actually believe this is going to work? And, again, I want you to look at this and think about what's going to happen here. Before we do that, or as you do that, let's look at Fib 1. The key thing to notice about Fib 1 is that it has the same specification as Fib. Because when somebody calls Fibonacci, they shouldn't worry about memos. And how I'd implemented it. That has to be under the covers. So I don't want them to have to call something with two arguments. The integer and the memo. So I'll create Fib 1, which has the same arguments as Fib. The first thing it does is it initializes the memo. And initializes it by saying, if I get I -- whoops. Aha. Let's be careful here. If I get 0 I return 1. I get 1, I return 1. So I put two things in the memo already. And then I'll call fast Fib and it returns the result it has.

So you see the basic idea. I take something with the same parameters as the original. Add this memo. Give it some initial values. And then call. So now what do we think? Is this going to work? Or is there an issue here? What do you think? Think it through. If it's not in the memo, I'll compute its value and put it in the memo. And then I'll return it. OK? If it was already there, I just look it up. That make sense to everybody?

Let's see what happens if we run it. Well, actually, let's turn the print statement on, since we're doing it with a small value here. So what we've seen is I've run it twice here. When I ran it up here, with the old Fib, and we printed the result, and I ran it with Fib 1 down here. The good news is we got 13 both times.

The even better news is that instead of 25 calls, it was only 11 calls. So it's a big improvement. Let's see what happens, just to get an idea of how big the improvement is. I'm going to take out the two print statements. And let's try it with a bigger number.

It's going to take a little bit. Well, look at this difference. It's 2,692,537 versus 59. That's a pretty darn big difference. And I won't ask you to check whether it got the right answer. At least, not in your heads.

So you can see, and this is an important thing we look at, is that as we look at growth, it didn't look like it mattered a lot with 6. Because it was one small number to one slightly smaller number. But this thing grows exponentially. It's a little bit complicated exactly how. But you can see as I go up to 30 I get a pretty big number. And 59 is a pretty small number. So we see that the memoization here is buying me a tremendous advantage. And this is what lies at the heart of this very general technique called dynamic programming.

And in fact, it lies at the heart of a lot of useful computational techniques where we save results. So if you think about the way something like, say, Mapquest works, and last week in recitation you looked at the fact that shortest path is exponential. Well, what it does is it saves a lot of paths. It kind of knows people are going to ask how do you get from Boston to New York City. And it may have saved that. And if you're going from Boston to someplace else where New York just happens to be on the way, it doesn't have to recompute that part of it. So it's saved a lot of things and squirreled them away. And that's essentially what we're doing here. Here we're doing it as part of one algorithm. There, they're just storing a database of previously solved problems. And relying on something called table lookup, Of which memoization is a special case. But table lookup is very common. When you do something complicated you save the answers and then you go get it later.

I should add that in some sense this is a phony straw-man Fibonacci. Nobody in their right mind actually implements a recursive Fibonacci the way I did it originally. Because the right way to do it is iteratively. And the right way to do it is not starting at the top, it's starting at the bottom. And so you can piece it together that way. But don't worry about it, it's not, I'm just using it because it's a simpler example than the one I really want to get to, which is knapsack. OK, people get this? And see the basic idea and why it's wonderful? Alright.

Now, when we talked about optimization problems in dynamic programming, I said there were two things to look for. One was overlapping sub-problems. And the other one was optimal substructure. The notion here is that you can get a globally optimal solution from locally optimal solutions to sub-problems.

This is not true of all problems. But as we'll see, it's true of a lot of problems. And when you have an optimal substructure and the local solutions overlap, that's when you can bring dynamic programming to bear. So when you're trying to think about is this a problem that I can solve with dynamic programming, these are the two questions you ask.

Let's now go back and instantiate these ideas for the knapsack problem we looked at last time. In particular, for the 0-1 knapsack problem. So, we have a collection of objects. We'll call it a. And for each object in 0, we have a value. In a, we have a value. And now we want to find the subset of a that has the maximum value, subject to the weight constraint. I'm just repeating the problem.

Now, what we saw last time is there's a brute force solution. As you have discovered in recent problem set, it is possible to construct all subsets of a set. And so you could construct all subsets, check that the weight is less than the weight of the knapsack, and then choose the subset with the maximum value. Or a subset with the maximum value, there may be more than one, and you're done.

On the other hand, we've seen that if the size of a is n, that's to say, we have n elements to choose from, then the number of possible subsets is 2 to the n. Remember, we saw that last time looking at the binary numbers. 2 to the n is a big number. And maybe we don't have to consider them all, because we can say, oh this one is going to be way too big. It's going to weigh too much, we don't need to look at it. But it'll still be order 2 to the n. If n is something like 50, not a big number, 2 to the 50 is a huge number.

So let's ask, is there an optimal substructure to this problem. That would let us tackle it with dynamic programming. And we're going to do this initially by looking at a straightforward implementation based upon what's called the decision tree.

This is a very important concept, and we'll see a lot of algorithms essentially implement decision trees. Let's look at an example. Let's assume that the weights, and I'll try a really small example to start with, are 5, 3 and 2, and the values, corresponding values, are 9, 7 and 8. And the maximum, we'll say, is 5.

So what we do is, we start by considering for each item whether to take it or not. For reasons that will become apparent when we implement it in code, I'm going to start at the back. The last element in the list. And what I'm going to use is the index of that element to keep track of where I am. So I'm not going to worry whether this item is a vase or a watch or painting. I'm just going to say it's the n'th element. Where n'th is somewhere between and 2 in this case. And then we'll construct our tree as follows: each node, well, let me put an example here. The first node will be the to-pull 2, 5 and 0. Standing for, let me make sure I get this in the right order, well, the index which is 2, the last element in this case, so that's the index. This is the weight still available. If you see that in the shadow. And this is the value currently obtained. So I haven't included anything. Means I have all 5 pounds left. But I don't have anything of value.

Now, the decision tree, if I branch left, it's a binary tree. This is going to be don't take. So I'm not going to take the item with an index of 2. So that means this node will have an index of 1.

Next item to be considered, I still have 5 pounds available. And I have value. To be systematic, I'm going to build this tree depth-first left-first. At each node, I'm going to go left until I can't go any further. So we'll take another don't-take branch here. And what is this node going to look like? Pardon?

STUDENT: [INAUDIBLE]

PROFESSOR: 0, 5, 0. And then we'll go one more. And I'll just put a minus indicating I'm done. I can't look below that. I still have five pounds left, and I still have zero value.

The next thing I'm going to do is backtrack. That is to say, I'm going to go back to a node I've already visited. Go up the tree 1. And now, of course, the only place to go is right. And now I get to include something. Yeah. And what does this node look like? Well, I'll give you a hint, it starts with a minus. What next?

STUDENT: [INAUDIBLE]

PROFESSOR: Pardon. 0. And?

STUDENT: [INAUDIBLE]

PROFESSOR: Pardon? 5. Alright. So far, this looks like the winner. But, we'd better keep going. We backtrack to here again. There's nothing useful to do. We backtrack to here. And we ask, what do we get with this node? 0, 3. Somebody?

STUDENT: [INAUDIBLE]

PROFESSOR: Louder. 2. And then?

STUDENT: [INAUDIBLE]

PROFESSOR: There's a value here, what's this value? I've included item number 1, which has a value of 7. Right? So now this looks like the winner. Pardon?

STUDENT: [INAUDIBLE]

PROFESSOR: This one?

STUDENT: Yeah. [INAUDIBLE]

PROFESSOR: Remember, I'm working from the back. So it shouldn't be 9. Should be what? Item 0, oh, you're right, items is 9. Thank you. Right you are. Still looks like the winner. I can't hear you.

STUDENT: [INAUDIBLE]

PROFESSOR: Let's be careful about this. I'm glad people are watching. So we're here. And now we've got 5 pounds available. That's good. And we're considering item number 0. Which happens to weigh 5 pounds. So that's a good thing. So, it's the last item to consider. If we include it, we'll have nothing left. Because we had 5 and we're using all 5. So that will be 0. And its value is 9. So we put one item in the backpack and we've got a value of 9. Anyone have a problem with that? So far, so good?

Now we've backed up to here. We're considering item 1 and we're trying to ask whether we can put it in. Item 1 has a weight of 3. So it would fit. And if we use it, we have 2 pounds left. And item 1 has a value of 7. So if we did this, we'd have a value of 7. But we're not done yet, right? We still have some things to consider. Well, we could consider not putting in item 0. That makes perfect sense. And we're back to where we're there, minus 2 and 7.

And now let's ask the question how, about putting in item 0. Well, we can't. Because it would weigh 5 pounds, I only have 2 left. So there is no right branch to this one. So I'm making whatever decisions I can make along the way. Let's back up to here. And now we're going to ask about taking item 2. If we take item 2, then, well, the index after that will of course be 1. And the available weight will be 3. And the value will be 8, right? Alright, now we say, can I take item 1. Yeah. I can. It only weighs 3 and I happen to have 3 left. So that's good. Don't take it, right. Sorry. This is the don't take branch. So I go to 0, 3, 8. And then I can do another don't take branch. And this gets me to what, minus 3, 8. I'll now back up. And I'll say, alright, suppose I do take item 0, well I can't take item 0, right? Weighs too much. So, don't have

that branch. Back up to here. Alright, can I take item 1? Yes, I can. And that gives me what?

STUDENT: [INAUDIBLE]

PROFESSOR: We have a winner. So it's kind of tedious, but it's important to see that it works. It's systematic. I have a way of exploring the possible solutions. And at the end I choose the winner. What's the complexity of this decision tree solution? Well, in the worst case, we're enumerating every possibility of in and out. Now I've shortened it a little bit by saying, ah, we've run out of weight, we're OK. But effectively it is, as we saw before, exponential. 2 to the n, every value in the bit vector we looked at last time is either 0 or 1. So it's a binary number of n bits, 2 to the n.

Let's look at a straightforward implementation of this. I'll get rid of Fibonacci here, we don't want to bother looking at that again. Hold on a second until I comment this out, yes.

STUDENT: [INAUDIBLE]

PROFESSOR: Yeah. There's a branch we could finish here, but since we're out of weight we sort of know we're going to be done. So we could complete it. But it's not very interesting. But yes, we probably should have done that.

So let's look at an implementation here. Whoops. You had these in the handout, by the way. So here's max val. It takes four arguments. The weight, w, and v, these are the two vectors we've seen here. Of the weights and the values. It takes i, which is in some sense the length of those vectors, minus 1, because of the way Python works. So that gives me my index. And the amount of weight available, a w, for available weight.

So again, I put in this num calls, which you can ignore. First line says, if i is 0, that means I'm looking at the very last element. Then if the weight of i is less than the available weight, I can return the value of i. Otherwise it's 0. I've got one element to look at. I either put it in if I can. If I can't, I don't. Alright, so if I'm at the end of the chain, that's my value. In either event, if I'm looking at the last element, I return.

The next line says alright, suppose I don't, I'm not at the last element. Suppose I don't include it. Then the maximum value I can get is the maximum value of what I had. But with index i minus 1. So that's the don't-take branch. As we've seen systematically in the don't-takes, the only thing that gets changed is the index.

The next line says if the weight of i is greater than a w, well then I know I can't put it in. So I might as well return the without i value I just computed. Otherwise, let's see what I get with i. And so the value of with i will be the value of i plus whatever I can get using the remaining items and decrementing the weight by the weight of i. So that's exactly what we see as we go down the right branches. I look at the rest of the list, but I've changed the value. And I've changed the available weight.

And then when I get to the very end, I'm going to return the bigger of with i and without i. I've computed the value if I include i. I computed the value if I don't include i. And then I'm going to just return the bigger of the two. Little bit complicated, but it's basically just implementing this decision tree.

Let's see what happens if we run it. Well, what I always do in anything like this is, the first thing I do is, I run it on something where I can actually compute the answer in my head. So I get a sense of whether or not I'm doing the right thing. So here's a little example.

And I'm going to pause for a minute, before I run it, and ask each of you to compute in your head what you think the answer should be. In line with what I said about debugging. Always guess before you run your program what you think it's going to do. And I'm going to wait until somebody raises their hand and gives me an answer. Yes.

STUDENT: 29?

PROFESSOR: So we have a hypothesis that the answer should be 29. Ooh. That's bad. So as people in the back are answering these questions because they want to test my arm. The camera probably didn't catch that, but it was a perfect throw. Anyone else think it's 29? Anyone think it's not 29? What do you think it is? Pardon

STUDENT: 62?

PROFESSOR: 62. That would be impressive. 62. Alright, well, we have a guess of 62. Well, let's run it and see. 29 it is. And it made a total of 13 calls. It didn't do this little optimization I did over here. But it gives us our answer. So that's pretty good. Let's try a slightly larger example.

So here I'm going to use the example we had in class last time. This was the burglar example where they had two copies of everything. Here, it gets a maximum value of 48 and 85 calls. So we see that I've doubled the size of the vector but I've much more than doubled the number of calls. This is one of these properties of this kind of exponential growth.

Well, let's be really brave here. And let's try a really big vector. So this particular vector, you probably can't even see the whole thing on your screen. Well, it's got 40 items in it. Let's see what happens here. Alright, who wants to tell me what the answer is while this computes away? Nobody in their right mind. I can tell you the answer, but that's because I've cheated, run the program before.

Alright, this is going to take a while. And why is it going to take a while? Actually it's not 40, I think these are, alright. So the answer is 75 and the number of calls is 1.7 million. Pardon? 17 million. Computers are fast, fortunately. Well, that's a lot of calls.

Let's try and figure out what's going on. But let's not try and figure out what's going on with this big, big example because we'll get really tired. Oh. Actually, before we do that, just for fun, what I want to do is write down for future reference, as we look at a fast one, that the answer is 75 and Eric, how many calls? 240,000. Alright. We'll come back to those numbers.

Let's look at it with a smaller example. We'll look at our example of 8. That we looked at before. And we'll turn on this print statement. Ooh, what was that? Notice that I'm only printing i and a w. Why is that? Because w and v are constant. I don't want you to print them over and over again. No, I'd better call it. That would be a good thing to do, right? So let's see. We'll call it with this one.

So it's printing a lot. It'll print, I think, 85 calls. And the thing you should notice here is that it's doing a lot of the same things over and over again. So, for example, we'll see 2, 1 here. And 2, 1 here. And 2, 1 here. Just like Fibonacci, it's doing the same work over and over again. So what's the solution?

Well, you won't be surprised to hear it's the same solution. So let's look at that code. So I'm going to do exactly the same trick we did before. I don't want b to the Fibonacci. I'm going to introduce a max val 0, which has exactly the same arguments as max val. Here I'll initiate the memo to be 0, or to be empty, rather, the dictionary. And then I'll call fast max val passing at this extra argument of the memo.

So the first thing I'm going to do is, I'm going to try and return the value in the memo. This is a good thing to do, right? If it's not already there, what will happen? It will raise an exception. And I'll go to the except clause. So I'm using the Python try except to check whether or not the thing is in the memo or not. I try and return the value. If it's there, I return it. If not I'll get a key error. And what do I do if I get a key error? Well, now I go through code very much like what I did for max val in the first place. I check whether it's 0, et cetera, et cetera. It's exactly, really, the same, except before I return the value I would have returned I squirrel it away in my memo for future use. So it's the same structure as before. Except, before I return the value, I save it. So the next time I need it, I can look it up.

Let's see if it works. Well, let's do a small example first. It's calling the old max val. With all those print statements. Sorry about that. But we'll let it run. Well, it's a little bit better. It got 85. Same answer, but 50 calls instead of 85. But let's try the big one. Because we're really brave. So here's where we have 30 elements, and a maximum weight of 40. I'm not going to call the other max val here, because we know what happens when I do that. I've created my own little memo over there.

And let's let it rip. Wow. Well, I got the same answer. That's a good thing. And instead of 17 million calls, I have 1800 calls. That's a huge improvement. And that's sort of the magic of dynamic programming.

On Thursday we'll do a more careful analysis and try and understand how I could have accomplished this seemingly magical task of solving an exponential problem so really quickly.