CS 4349 Lecture—September 13th, 2017

Main topics for #lecture include #dynamic_programming, #Fibonacci_numbers, and #rod_cutting.

Prelude

- Homework 2 due today in class.
- Homework 3 released, due next Wednesday September 20th.

Fibonacci Numbers

- Last time, we looked at computing Fibonacci numbers. F(0) = 0, F(1) = 1, and F(n) = F(n-1) + F(n-2) for all n ≥_2.
- RecFibo(n):

```
RecFibo(n):
    if n < 2
        return n
    else
        return RecFibo(n-1) + RecFibo(n-2)
```

- But this takes Theta(phi^n) where phi = (sqrt(5) + 1)/2 ~= 1.618. It takes exponential time to compute F(n)!
- The problem here is that we're repeating a ton of work. If we make a tree just listing the recursive calls, the calls for smaller values of n are done over and over again.

- So how do we fix this problem. The first solution is to stop repeating work we've already done.
- We can use a technique called memoization where we remember the result of recursive calls we've already done. We'll keep a global array F around that our algorithm will fill in and access as needed.
- MemFibo(n):

```
MemFibo(n):
    if n < 2
        return n
    else
        if F[n] is undefined
            F[n] ← MemFibo(n-1) + MemFibo(n-2)
        return F[n]
```

- So how much does this help? Each F[i] is filled in only after F[i-1] is filled in. Each recursive call takes constant time returning a base case, checking F[i], or filling it in. But even the calls checking F[i] are only done twice, once when filling F[i+1] and once when filling

F[i+2]. The whole things takes constant time per entry in F, so O(n) time.

- But we can make this process more explicit and easier to analyze. Instead of letting recursion fill in the entires one-by-one, we'll use a simple for loop to do it for us.
- IterFibo(n):

```
    • F[0] ← 0
    • F[1] ← 1
    • for i ← 2 to n
        • F[i] ← F[i-1] + F[i-2]
    • return F[n]
```

- Now the analysis is much easier. We just have one for loop going over O(n) values of i. It takes O(n) time.
- Correctness is still pretty straightforward, also. We just need to prove that after iteration i, F[j] = F(j) for all 0 ≤_j ≤_i. It's induction, but based directly on the definition of F(j).

- This is one example of a paradigm called "dynamic programming" where programming is used in the old fashioned sense of filling in a table or schedule.

1. Find a recursive algorithm or recurrence relation that describes an optimal (or only) solution to your problem.
2. Fill in a data structure with the recursive answers.

Rod Cutting

- For this problem, we are given a rod of length n. We need to cut it into smaller pieces that we can sell. The selling prices (in dollars) for different lengths of rod are given in an array P[1 .. n]. The total length of the pieces we sell must be n.
- For example, if n = 4, and P[0 .. n] = <0, 1, 5, 8, 9>, we could sell the whole rod for 9 dollars. Or we could instead cut the rod into four pieces of length 1 for 4 * 1 = 4 dollars. But the best option is to cut the rod into two pieces of length 2 for 5 + 5 = 10 dollars.
- We need to design an algorithm that finds a rod cutting that maximizes to the total selling price of the pieces. **we'll focus only on computing that maximum selling price for now**
- So let's try using that dynamic programming paradigm.

- First, we'll design a recursive algorithm. Often when trying to optimize something like revenue, there is a first decision you need to make. The rest of the optimal solution is an optimal solution for the remaining decisions.
- So, we need to find a collection of rod lengths that sum up to n. **what is a first decision we might make for rod cutting?**
- It's the length of the first piece.

- So now let's say we chop off that first piece of length i to sell it. We're left with a rod of length n - i. How do we decide what to do with the rest of the rod?
- Recursion! We'll use n = 0 as a base case, since there's nothing to sell!

```
    Cut-Rod-Rec(n):  P[] is a global array of piece prices
        if n = 0
            return 0
        q = 0  Eventually equals maximum selling price
        for i ← 1 to n
            q ← max(q, P[i] + Cut-Rod(n - i))
```

- We could also make a recurrence relation instead, so there's less to write and so it is more clear how to compute different solutions.
- Let Cut-Rod-Revenue(j) be the maximum revenue (total selling price) when cutting a rod of length j. **If you use recurrences like, you must always give a definition so we know what they represent**
- Cut-Rod-Revenue(j) = {0 if j = 0, max_{1 ≤ i ≤ j} P[i] + Cut-Rod-Revenue(j - i) o.w.}

- Now, we need to pick a data structure and fill it in.
- An array R[0 .. n] will do.
- But in what order can we fill the the array? **any suggestions?**
- We'll do so in increasing order of index, starting with R[0] = 0.
- We can depict it visually as [0 ← ← ← ← []] ← answer
- So we fill in the base case on the left, each other entry is filled in by applying the recurrences using all the values to the left, and the final answer for a rod of length n lies at R[n].
- For code:

```
    Cut-Rod-Iter(n):
        R[0] ← 0
        for j ← 1 to n
            q = 0
            for i ← 1 to j
                q = max(q, P[i] + R[j - i])
            R[j] = q
        return R[n]
```

- So is it correct? Well, we can easily prove the recurrence is correct using induction. Yes, you get 0 revenue from 0 length of rod. And assuming the recurrence is correct for smaller rod lengths, you do get the sum of what it returns and your selling price from the first piece. The actual iterative algorithm is just computing the recursive solutions, and recursive answers are always available because we only depend upon the smaller indices we already passed.

- Again, the running time is easy to see. We have two for loops where j goes from 1 to n and i goes from 1 to at most n. It takes O(n^2) time.

Dynamic Programming in Detail

- Dynamic programming a very powerful method for designing algorithms. It also has this interesting feature that you need some cleverness at the beginning, to figure out the right recursive algorithm or recurrences, and then the rest is almost completely mechanical.
- Let’s discuss those two steps in some more detail.

1. Find a recursive algorithm or recurrence relation that describes an optimal (or only) solution to your problem.
    - This is the hard part. You’ll need to think of some recursive structure in the solution you’re trying to find. Maybe it’s a first choice plus optimal solution to the consequences, or maybe it’s something less predictable.
    - Always think about the recursive structure first. Do not be tempted to start by filling in a table, because you don’t know yet how to fill it in correctly.
    1. Explain, in English, what your recursive algorithm or recurrence computes. Otherwise, it is impossible to tell if it is correct. If using a recurrence, say what parameters give the answer to the original problem.
    - For Fibonacci numbers, F(i) is simply the ith Fibonacci number. For rod cutting, Cut-Rod-Revenue(i) was the maximum revenue for a rob of length n. We returned F(n) and Cut-Rod-Revenue(n).
    3. Give a formal definition of the recursive algorithm or recurrence.
    - Usually, the recurrence is easier. It’s correctness is proved by induction, and you don’t care about the details of actually computing it yet so an algorithm is unnecessary.
    - Again, the definition of Fibonacci numbers was our recurrence, done. For rod cutting, we considered how making a first choice forces you to solve a smaller rod cutting instance optimally.
2. Fill in a data structure with the recursive answers.
    - This part is easier, and relatively mechanical.
    1. Identify the *subproblems* or recursive answers you’ll need to compute.
    - F(i) or Cut-Rod-Revenue(i) for all 0 ≤_ i ≤ n.
    2. Analyze your algorithm.
    - You algorithm will use space proportional to the number of subproblems. The time will be the number of subproblems * the time to solve each one ignoring the recursive calls.
    - So O(n) space and time for Fibonacci. O(n) space and O(n^2) time for rod cutting.
    3. Choose a data structure to store your subproblem answers.
    - Usually, you’ll use a multidimensional array where each dimension is a parameter in

your recurrence.

- We used a 1-dimensional array indexed 0 through n in both examples.

4. Identify dependencies between your subproblems.

- It helps to draw a picture of your data structure, pick a generic element (not a base case), and draw arrows to the elements it is dependent on.
- Fibonacci was dependent on just two elements to the left. Rod cutting was dependent on all elements to the left.

5. Find a good evaluation order.

- Order the subproblems so that a subproblem is evaluated *after* all subproblems it is dependent on. For multidimensional arrays this usually means solving the base cases and then progressing the indices. For example, by increasing row and column in row major order. You can often draw thick arrows in your picture to describe the evaluation order.
- For both Fibonacci and rod cutting, we evaluated subproblems in increasing order of i.

6. Write down the algorithm*

- If you've done all the steps earlier, you've already completely described the algorithm. An implementation would require a few lines of code. If the data structure is an array, you might have a few nested for loops. **you do not have to write out the iterative algorithm on homework or exams if you've already done the above steps**
- Next week, we'll discuss some more interesting examples, including having multiple parameters and possibly dynamic programming over trees instead of arrays.