CS 6363.003.21S Lecture 7—February 9, 2021

Main topics are #dynamic_programming with #example/fibonacci_numbers and #example/rod_cutting.

Fibonacci Numbers

- Today, we're going to look at a situation where recursion can go wrong and how to fix it.
- By now, I assume you're all familiar with Fibonacci numbers.
- These were described by Leonardo of Pisa (Fibonacci) around the 12th century, but the Fibonacci recurrence was originally discovered by Indian scholar Virahanka 500 years earlier during his study of classical Sanskrit poetry.
- In this class, Fibonacci numbers are defined using the following recurrence.
  - F_0 = 0
  - F_1 = 1
  - F_n = F_{n-1} + F_{n-2}

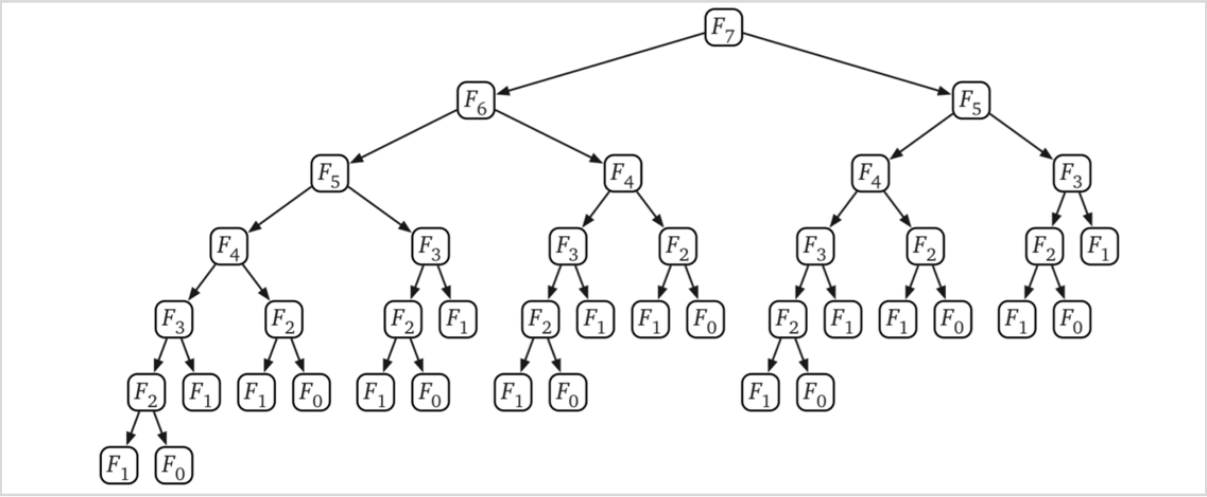
Recursion is Sometimes Slow

- Fibonacci numbers describe many different phenomena, so it would be good if there was some way to calculate them.
- Fibonacci numbers are defined recursively, so the most natural thing to do is compute them recursively:

```
RecFibo(n):
  if n = 0
      return 0
  else if n = 1
      return n
  else
      return RecFibo(n − 1) + RecFibo(n − 2)
```

- Unfortunately, this algorithm is horribly slow.
- Let T(n) be the number of subproblems for computing F_n assuming arithmetic operations takes constant time. Outside the recursive calls, we do only a constant number of steps, so we might write T(n) = T(n - 1) + T(n - 2) + 1 with T(0) = 1 and T(1) = 1.
- It's almost the same recurrence once again! And if we wrote out the first few terms we could correctly guess T(n) = 2F_{n+1} - 1.
- But that means computing F_n takes as long as counting to it.
- Using techniques that some of you may have seen in discrete math, we can figure out F_n = Theta(phi^n) where phi = (sqrt(5) + 1) / 2~ 1.61803, the *golden ratio*. This algorithm is exponential in n!
- But is there an intuitive explanation for why is it so slow? Let's look at a recursion tree. I'll

mark which number we're trying to compute.



- All these numbers come from adding up values computed in the recursive calls, so there must be F_n leaves with value 1. These are the calls to RecFibo(1). There's at most that many calls to RecFibo(0), so Theta(F_n) leaves total. It's a full binary tree, so there must be Theta(F_n) nodes. That's a lot of recursive calls!

Memoization

- Of course, even in this small example, you can tell we're wasting a lot of time recomputing the same numbers over and over again. You can even prove via induction that RecFibo(n - k) is called F_{k - 1} times.
- Now one thing we could do is try to remember which values we've already computed by storing them in a global array. This is called *memoization*.

```
MEMFIBO(n):
  if n = 0
      return 0
  else if n = 1
      return 1
  else
      if F[n] is undefined
          F[n] ← MEMFIBO(n − 1) + MEMFIBO(n − 2)
      return F[n]
```

- I claim this algorithm is *a lot* more efficient than the previous one, but the exact asymptotic running time isn't really clear.
- Rather than try to fully explain the running time, I will offer one useful observation: because computing each value F[i] depends upon calls to MemFibo(i -1) and MemFibo(i - 2), we must be filling in the array F[] from the bottom up. We compute F[i] for all i from 0 to n, *in that order*.

Filling Deliberately and Dynamic Programming

- But now that we see how the array is filled, why don't we just do so deliberately to make

the algorithm even simpler? We'll use a simple iterative algorithm that fills the array entries one by one.

```
ITERFIBO(n):
  F[0] ← 0
  F[1] ← 1
  for i ← 2 to n
      F[i] ← F[i−1] + F[i−2]
  return F[n]
```

- Not only does this algorithm avoid recursion and probably perform better in practice, but it's *a lot* easier to analyze. There's a single for loop over n entries, so it performs only O(n) additions. We can easily tell that it stores O(n) integers as well.
- This is an example of a *dynamic programming* algorithm, formalized and popularized by Richard Bellman in the mid-1950s, although others including Virahanka and Fibonacci already applied it to this example centuries earlier.
- Bellman claimed the term dynamic programming was used to hide the fact he was doing mathematics research from his industry-minded military bosses, although there are reasons to doubt that story.
- Note the word programming here doesn't refer to writing code. It's being used in the same way that television and radio stations plan or schedule their "programs", typically by filling a table. They were originally trying to optimize for certain time-varying processes, so Bellman used the term dynamic. Think of dynamic as a noun instead of an adjective, and maybe it makes more sense.
- Dynamic programming is now a standard tool for multistage planning in a variety of areas, and it's one of the most useful tools we have for designing algorithms. Even the unix diff utility is using a dynamic programing algorithm.

Saving Space

- **[Save for later unless somebody asks]**
- Our job with dynamic programming was (ultimately) to recognize how to fill a table with Fibonacci numbers, but keeping a whole table around when all you need is a single answer is somewhat wasteful.
- Since we only refer back to the last two entries in each iteration of the for loop, we can easily save some space in this instance.

```
ITERFIBO2(n):
  prev ← 1
  curr ← 0
  for i ← 1 to n
      next ← curr + prev
      prev ← curr
      curr ← next
  return curr
```

- Now we're only storing a constant number of integers.
- I *usually* won't ask you about space usage in this class, but I may for the next couple weeks since it's easy enough to figure out and you can sometimes make easy improvements when doing dynamic programming.

Rod Cutting

- By now, we've seen how certain computations, in particular those based on evaluating a recurrence relation, can be done more quickly by using dynamic programming. But now we should ask, where do these recurrences typically come from, and why are they useful?
- Recurrences, and their dynamic programing evaluations, often pop up when attempting to solve certain optimization problems.
- An *optimization problem* is one where they may be many valid or feasible solutions, and each solution has a numerical value. We wish to find the solution with optimal (maximum or minimum) value.
- Let's look at one more example before I discuss the general strategy behind the whole recurrence to dynamic programming paradigm.
- We'll look at what CLRS calls the *rod-cutting problem*. Imagine we're working for a company that buys large lengths of steal rod and cuts them up to resale the smaller pieces. We want to maximum has much revenue we make.
- For our problem input, we are given a non-negative *integer* n representing the length of our original rod. We are also given an array P[1 .. n] where P[i] is the revenue we obtain selling a rod piece of length i. A solution to our problem is a list of positive integer lengths i_1, i_2, …, i_k such that sum_{j = 1}^k i_j = n. We want to maximize the value of our solution sum_{j = 1}^k P[i_j]. Note we're making *no* additional assumptions about the price array. Maybe people are willing to spend huge for lengths 12 and 14, but nothing for length 13.
- For example, if n = 4, and P[1 .. n] = <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.
- Now, when initially designing algorithms for optimization problems, it's easiest if we focus not on computing the solution itself but instead computing the optimal *value* of the solution. So that's what we'll primarily focus on. We can come back to outputting the actual solutions later.

- The first and hardest step in designing a dynamic programming algorithm is to formulate and solve your problem recursively.
- I like to think of these kinds of optimization problems as us having to make a sequence of decisions. Once we've made a decision, we're left trying to deal with the consequences,

and that hopefully means solving a smaller version of the same problem.

- 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?**
- Let's go with length of the first piece.
- Say we chop off a first piece of length j to sell it. We're left with a rod of length n - j. The best thing to do at this point is to maximize revenue on a rod of length n - j!
- We can most easily write down this relationship between the max revenue for length n and the max revenue for length n - j using a recurrence relation.
- Let CutRod(i) denote the maximum revenue obtainable cutting a rod of length n given the array P[1 .. n]. If we know we start with a piece of length j, then, CutRod(i) = P[j] + CutRod(i - j)
- But wait. What is j? We don't know in advance which length j is the best. Therefore, we'll have to try them all! We need to use the j that maximizes the total revenue from the first cut and the remaining cuts.
- CutRod(i) =
  - max_{1 ≤ j ≤ i} P[j] + CutRod(i - j) if i > 0
- I want to emphasize that we're not doing anything particularly clever here. We're assuming almost nothing about the solution, other than we want to optimize how we cut up the remaining i - j units of rod after selling our first piece of length j. We have no way of knowing which j is best, so we really do have to try them all and take the choice that maximizes that sum.
- OK, so CutRode(i) still isn't defined for *every* non-negative integer i, because it's missing the case i = 0. But zero units of rod should sell for 0.
- CutRod(i) =
  - 0 if i = 0
  - max_{1 ≤ j ≤ i} P[j] + CutRod(i - j) if i > 0
- This recurrence immediately implies a (slow) algorithm for computing the optimal revenue. Simply evaluate CutRod(n) using the recurrence definition.
- This idea of trying each option for exactly one decision (e.g. length of the first piece) and then recursively solving problem instances consistent with each option is called *backtracking*.
- The specific idea that an optimal solution to this optimization problem incorporates optimal solutions to related subproblems is called the *optimal substructure* property by CLRS.

- OK, so we have *an* algorithm, but it's going to be very very slow.
- But! We already saw how to speed up the algorithm: we need to solve all the subproblems by writing their solutions into a data structure.
- An array CutRod[0 .. n] will do. It has one entry per parameter value to the recurrence.

- We need to figure out what order we can fill the array. I like to do this by drawing a picture.
- We draw the array. We write down the solution to any base cases. Then we pick a generic element and draw arrows *into* it depicting which subproblems are used to gets its solution.
- [0 → → → → []]
- So we fill in the table from left to right (low index to high index).
- Now we're ready to write the iterative algorithm.

```
RodCutting(n, P[1 .. n]):
  CutRod[0] ← 0
  for i ← 1 to n
      R[i] ← 0
      for j ← 1 to i
          if P[j] + R[i − j] > R[i]
              R[i] ← P[j] + R[i − j]
  return R[n]
```

- Now it's easy to figure out the running time. We have two for loops where i goes from 1 to n and j goes from 1 to at most n. The algorithm takes O(n^2) time.

Dynamic Programming

- Dynamic programming is *recursion without repetition*. You store the solutions to the intermediate subproblems, often but not always in an array or table.
- Many students focus on the table, because tables are easy and familiar. But you need to focus on the much more important (and difficult) problem of finding a correct recurrence. If you memoize the correct recurrence, you may not even need an explicit table, but if the recursion is incorrect, nothing works. In particular, if you jump straight to making a table on your homework or exam solutions, you're going to have a much harder time convincing me your algorithm is correct, and my default assumption will be that it's actually wrong.

Dynamic programming is *not* about filling in tables.
It's about smart recursion!

- That said, there is a framework that you can and should follow to make things easier. Even if you normally ignore the reading (which is a bad idea), you really really should look over Erickson Section 3.4 and commit its recommendations to heart.
    1. Formulate the problem recursively. Write down a recursive formula or algorithm for the whole problem in terms of smaller subproblems. This is the hard part. (I greatly prefer recursive formulas, as they make the second stage easier.)
        1. Specification. Describe the problem you want to solve recursively in coherent and precise English. Not *how* to solve the problem, but *what* the problem is. Without this, we cannot tell if your solution is correct (the solution to what?). Also, which call do you make to solve the original problem you were asked about? We needed to evaluate RecFibo(n) and CutRod(n).

    2. Solution. Give a clear recursive formula or algorithm for the whole problem in terms of answers to smaller instances of *exactly* the same problem.
2. Build solutions to the recurrence from the bottom up. Write an algorithm that starts with base cases for your recurrence and works its way up to the final solution by considering intermediate subproblems in the correct order. This is the easy(er) part, and can be broken down into smaller relatively mechanical steps (an algorithm for designing algorithms!)
    1. Identify the subproblems. What are the different ways your recursive algorithm can call itself? Both RecFibo and CutRod take integers between 0 and n.
    2. Choose a memoization data structure. Find a structure to store the solution to every subproblem from part (a). This is usually *but not always* a multidimensional array. I usually name it after the recurrence defined in the specification step.
    3. Identify dependencies. Except for base cases, every subproblem depends on other subproblems. Which ones? Draw a picture of your data structure, pick a generic element, and draw arrows from each element it depends upon.
    4. Find a good evaluation order. Order the subproblems so that each one comes *after* the subproblems it depends on.
    5. Analyze space and running time. The number of distinct subproblems determine the space complexity of your algorithm. For total running time, add up the running time for evaluating all possible subproblems, assuming deeper recursive calls have already been memoized.
        - Running time is *at most* [number of subproblems] * [max time per subproblem].
        - But sometimes you can argue for a lower running time using a more clever analysis.
    6. Write down the algorithm. You know the order to consider subproblems and how to solve them, so do that! If the data structure is an array, you'll usually write some nested for-loops around the recurrence and replace recursive calls with array lookups. Make sure you don't accidentally skip the base cases!
- Don't forget to prove these steps are correct. Why does your recurrence/recursive algorithm solve the problem (induction)? Tell me the dependencies so I know you're solving subproblems in the correct order.

Greed is Bad, Actually

- Now, if you're very lucky, you might be able to run a greedy algorithm.
- It's like backtracking, except you don't make one recursive call per option. Instead, you immediately commit to a "best" option and do one recursive call **total** to find out the consequences. So backtracking without ever tracking back.

- It seems natural, but very few problems can be solved correctly in this way.
- For example, for rod cutting, you might set j to the value that maximizes P[j] and then attempt to recursively cut the rest of the rod.
- But even a simple example like n = 4, and P[1~..~4] = <0, 50, 51, 0> shows that strategy won't work.
- So remember

(ok, sometimes they do and we'll discuss some examples later in the semester).

- If you ever get an inkling that a greedy approach might work, you really want to use dynamic programming instead. Really.
- In a couple weeks, we'll go over what's involved in designing a greedy algorithm and proving it correct. Until then, don't even try to use one.