# CS 4349.003.19F Lecture 8—September 16, 2019

Main topics for #lecture include #dynamic_programming including #example/fibonacci_numbers and #example/string_segmentation.

## Fibonacci Numbers

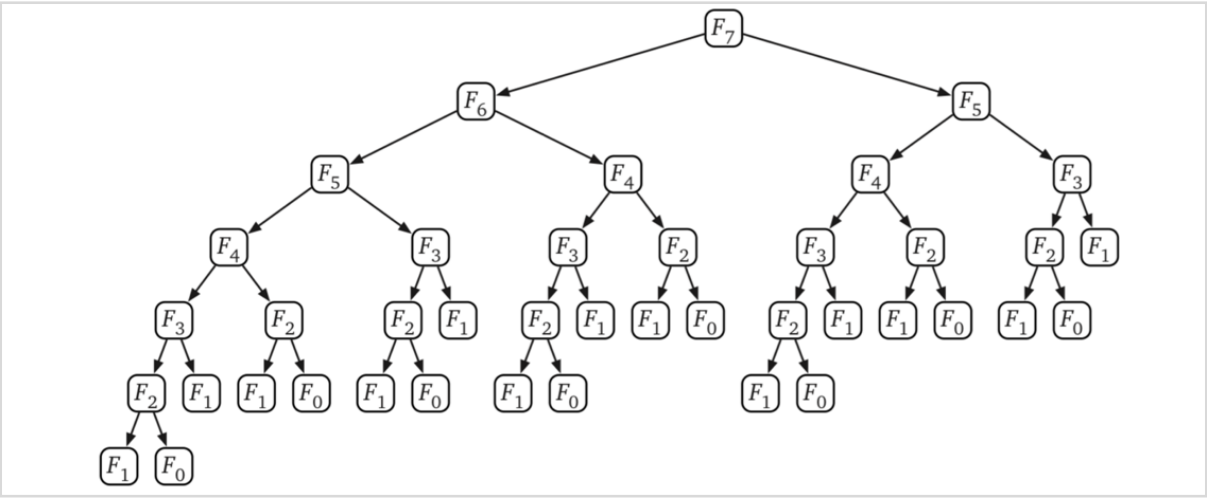
- By now, I assume you're all familiar with Fibonacci numbers.
- These were described by Leonardo of Pisa (Fibonacci) around the 12th century, but the Fibonacci recurrence was originally discovered by Indian scholar Virahanka 500 years earlier during his study of classical Sanskrit poetry.
- In this 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 we can design a recursive algorithm pretty easily:

```
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]
```

- If we look through the recursive calls of MemFibo, we see that the array F[] is filled from the bottom up. First F[2], then F[3], and so on. In a sense, we've significantly trimmed the recursion tree!
- Outside recursive calls, it only takes O(1) time to evaluate each F_i. Also, each call MemFibo(i) is made only twice: within MemFibo(i + 1) and MemFibo(i + 2). The total number of additions is only O(n). That's *a lot* faster.

### Filling Deliberately and Dynamic Programming

- But now that we see how the table is filled, why don't we just do so deliberately to make the algorithm even simpler. We'll just 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. 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 Wednesday 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.

## String Segmentation Redux

- Let's look at another example. Last week, we looked at a backtracking algorithm for string segmentation.
- Given a string A[1 .. n] and a subroutine IsWord, we wanted to know if we could partition A into a sequence of words.
- We came up with the following backtracking algorithm: Iteratively guess the first word by looping over all final characters, and recursively see if the remainder of the string can be segmented.

```
SPLITTABLE(A[1..n]):
  if n = 0
      return TRUE
  for i ← 1 to n
      if ISWORD(A[1..i])
          if SPLITTABLE(A[i+1..n])
              return TRUE
  return FALSE
```

- Unfortunately, this algorithm makes O(2^n) calls to IsWord!
- At the end of the lecture, when discussing index formulations of backtracking problems, we solved string segmentation by recursively computing a boolean value Splittable(i) which is True if and only if A[i .. n] can be split into a sequence of words. Here, IsWord(i, j) is shorthand for IsWord(A[i .. j]).

$$Splittable(i) = \begin{cases} \text{TRUE} & \text{if } i > n \\ \bigvee_{j=i}^{n} \big(IsWord(i,j) \wedge Splittable(j+1)\big) & \text{otherwise} \end{cases}$$

- So A[1 .. n] is splittable if and only if Splittable(1) is true. But evaluating the recurrence directly would still take O(2^n) time.
- But like with Fibonacci numbers, that's being wasteful. There are only n + 1 different ways to call Splittable(i) and only O(n^2) ways to call IsWord(i, j). A naive recursive algorithm would repeat many of the same computations many times.
- So instead, let's memoize the results of Splittable(i) into an array Splittable[1 .. n + 1].
- Each subproblem Splittable(i) only depends upon subproblems Splittable(j) where j > i, so the memoization algorithm will have to fill the array in *decreasing* index order.
- **[draw a figure with arrows going** *from* **dependencies]**
- But at this point we can skip straight to the iterative algorithm.

```
FastSplittable(A[1..n]):
  SplitTable[n+1] ← True
  for i ← n down to 1
      SplitTable[i] ← False
      for j ← i to n
          if IsWord(i, j) and SplitTable[j+1]
              SplitTable[i] ← True
  return SplitTable[1]
```

- We can tell just from the nested for loops that the algorithm makes O(n^2) calls to IsWord, spends O(n^2) time outside IsWord, and uses O(n) space to hold Splittable[1 .. n + 1].
- Hopefully you see now why we discussed index formulation last week. If you can describe the solution to a problem using index formulation and a recurrence, then it's straightforward to write an iterative dynamic programming algorithm: loop over the subproblems in the right way and directly evaluate the code in the recurrence

## 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 Splittable(1).
        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? RecFibo took integers between 0 and n. Splittable took integers between 1 and n + 1.
    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. Base cases first.
    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.
    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.

- Don't forget to prove these steps are correct. Why does your recurrence/recursive algorithm solve the problem? 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 text segmentation, you might find the shortest prefix that is a word, accept that word **is** going into your segmentation, and then attempt to recursively segment the rest of the string.
- But even a simple example like "ARE" shows that strategy won't work.
- So remember


  (except when they do).
- 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.