Dynamic Programming: Memoized Recursion

Sam McCauley

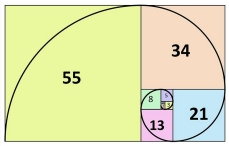
March 20, 2025

Welcome Back!

- Problem set 4 deadline postponed
- Problem Set 5 out this afternoon
- Shorter, lots of parts filled in; on Problems 2–3 the parts you should fill in are in red
- Focus: practice with dynamic programming
- Relies entirely on *today* (not Monday after break, though we'll get more DP practice then)
- Looking ahead: on midterm 2 you can have a 1-page cheat sheet
- Questions?

Fibonacci Numbers

Fibonacci Numbers



- Definition: $F_n = F_{n-1} + F_{n-2}$, $F_0 = 0$, $F_1 = 1$
- $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$
- How can we calculate these numbers?

Fibonacci Algorithm

```
1 fib(n):
2     if n == 0:
3         return 0
4     if n == 1:
5         return 1
6     return fib(n-1) + fib(n-2)
```

- Clearly correct (can prove using strong induction); what's the running time?
- $T(n) = T(n-1) + T(n-2) + \Theta(1)$; $T(0) = T(1) = 1$
- At least as large as: $T'(n) = T'(n-1) + T'(n-2)$; $T(1) = T(2) = 1$
- Which *is* the nth fibonacci number!

Fibonacci Algorithm

```
fib(n):
    if n == 1:
        return 1
    if n == 2:
        return 1
    return fib(n-1) + fib(n-2)
```

- 200th Fibonacci number is $\geq 2 \times 10^{41}$. So would need *at least* 2×10^{41} operations to calculate.
- World's fastest supercomputer would need $10,000 \times$ (age of the universe) years to finish this calculation
- Let's discuss: is there a better way?

Fibonacci Algorithm Improved

- Create an array F, where $F[i]$ stores the ith largest Fibonacci number
- Set $F[0] = 1$ and $F[1] = 1$.
- Now we can fill out $F[2]$, then $F[3]$, and so on.

Fibonacci Algorithm Improved

```
1  fib(n):
2      create an array F
3      F[0] = F[1] = 1
4      for j = 2 to n:
5          F[j] = F[j − 1] + F[j − 2]
```

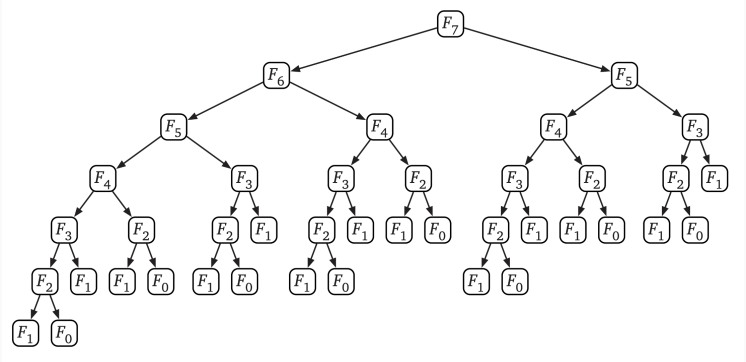
- Correctness: when we fill in $F[j]$, we have already put the correct value in $F[j-1]$ and $F[j-2]$
- What is the running time?
- $O(n)$
- Can calculate the 200th Fibonacci number with ≈ 200 additions!

What happened here?

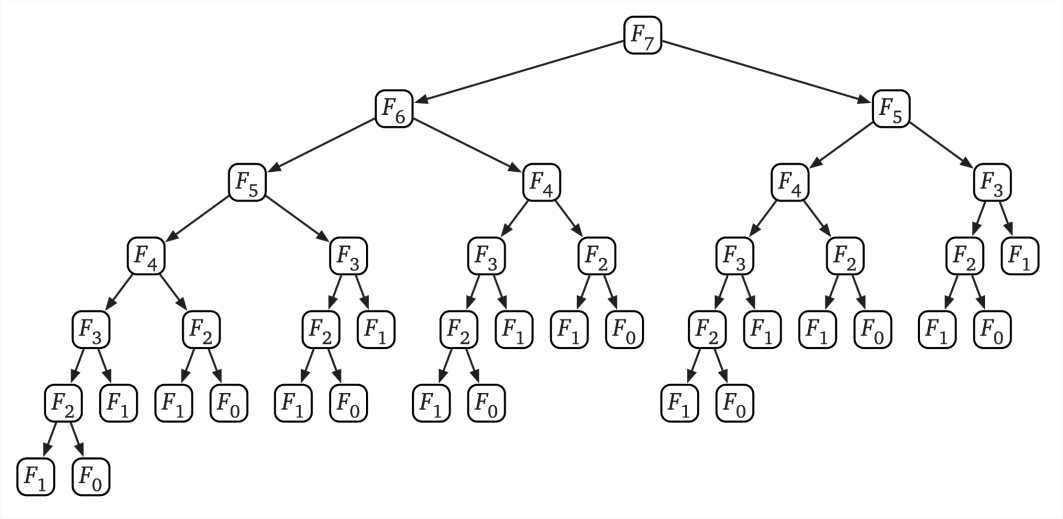
- Both algorithms seem reasonable
- One takes 200 operations, the other takes 10^{41} operations
- Where are we *losing time*?

Recursive Fibonacci

```
1  fib(n):
2      if n == 0:
3          return 0
4      if n == 1:
5          return 1
6      return fib(n-1) + fib(n-2)
```



Recursive Fibonacci

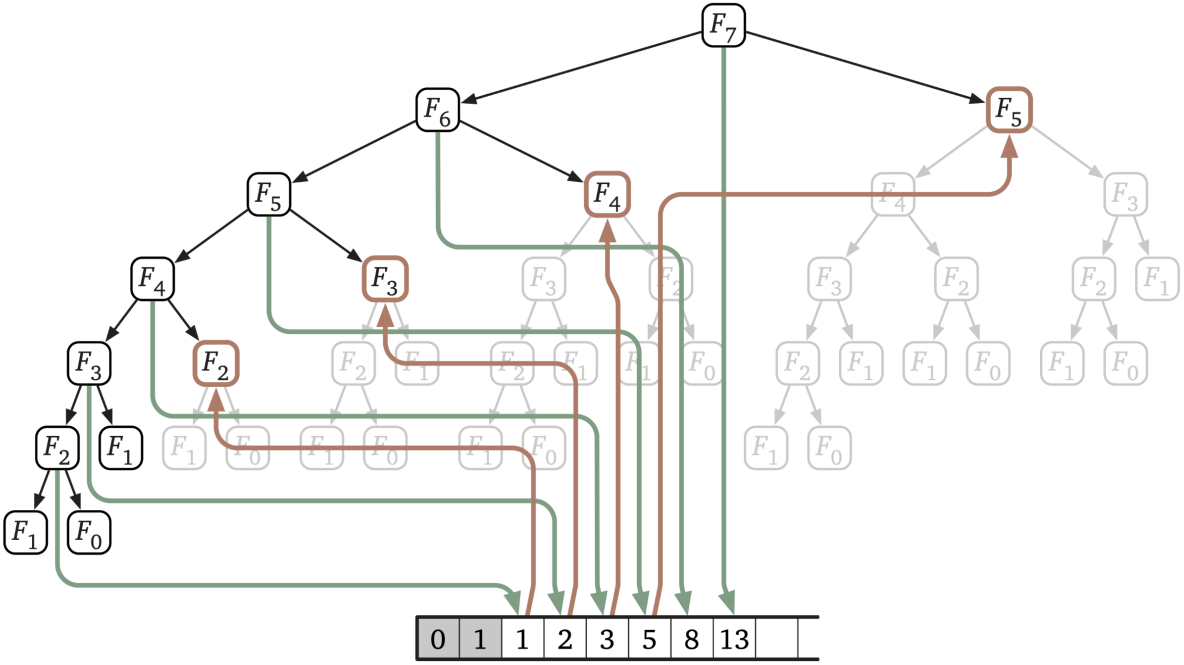


Recursive Fibonacci Slowness

- We recompute the *entire recursive call* each time we "need" a number
- So to compute F_{200}, we need F_{199} and F_{198}
- The first thing F_{199} does is call F_{198}. We do that entire computation twice!
- We compute (say) F_{100} many, many, many times

Array-Based Fibonacci

- Same recursive structure, but
- each number is computed *exactly once*.
- We write down the solutions in our array so that we can reuse them later
- Dynamic Programming: recursive algorithm where we write down solutions we calculated to reuse them later
- Writing down already-calculated values is called *memoization*. So dynamic programming is recursion with memoization


F_7
F_6
F_5
F_5
F_4
F_4
F_3
F_4
F_3
F_3
F_2
F_3
F_2
F_2
F_1
F_3
F_2
F_2
F_1
F_2
F_1
F_1
F_0
F_2
F_1
F_1
F_0
F_1
F_0
F_2
F_1
F_1
F_0
F_1
F_0
F_1
F_0
F_1
F_0
F_1
F_0
0
1
1
2
3
5
8
13

Dynamic Programming

- Fibonacci numbers are a nice example
- Dynamic programming is *extremely* powerful, and can solve a wide variety of problems
- We'll be exploring these problems over the next 2-3 lectures, obtaining increasingly powerful strategies for dynamic programming solutions

Longest Increasing Subsequence

Longest Increasing Subsequence

- Given: an arbitrary array A of length n
- Goal: find the length of the longest subsequence of elements that are in increasing order

Longest Increasing Subsequence

- Given: an arbitrary array A of length n
- Goal: find the length of the longest subsequence of elements that are in sorted order

The longest increasing subsequence has length 6.

Longest Increasing Subsequence

- Given: an arbitrary array A of length n
- Goal: find the length of the longest subsequence of elements that are in sorted order

An increasing subsequence of length 4 (not longest!)

Greedy Doesn’t Work

- Greedy: (repeatedly) take the next item larger than our current item and add it to our solution.
- Why doesn’t this work?
- Greedy makes bad early decisions which prevents us from getting the optimal solution
- Taking 10 means we can’t take 3, 4, 8!

Longest Increasing Subsequence Definition (and extension)

- Given an array A of length n
- Find the largest ℓ such that there is a sequence of indices $i_1 < i_2 < \ldots < i_\ell$ such that for all $k < \ell$, $A[i_k] < A[i_{k+1}]$
- Also called LIS
- Note that today we will just get the *length* of the sequence
- We'll talk about how to get the *sequence* (not just the length) after spring break

LISE

- Consider the following restricted problem: the Longest Increasing Subsequence Ending at $n-1$ (LISE)
  - Find the length of the longest increasing subsequence that includes $A[n-1]$
  - 0-indexed; so this is the last element of the array
- Let's focus on LISE for now. Then we'll double back for LIS

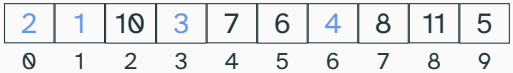
1	2	10	3	7	6	4	8	11	3	1

LISE of this array is: 1!

Solving LISE Recursively

- How can I recursively find the LISE of this array?
- What do I know about any increasing subsequence ending at the last element (5)?
    - The *second to last* element must be < 5

Solving LISE Recursively



- One of $2, 1, 3, 4$ (in slots $0, 1, 3, 6$) must be the second-to-last element in my LISE
- Which one is best?
- Let's say I *already wrote down* the length of the longest increasing subsequence ending at 2, 1, 3, 4? How does that help us?

Solving LISE Recursively

- The length of the longest increasing subsequence ending at 5 is 1+ the max of:
  - The longest increasing subsequence ending at 4 (slot 6)
  - The longest increasing subsequence ending at 3 (slot 3)
  - The longest increasing subsequence ending at 1 (slot 1)
  - The longest increasing subsequence ending at 2 (slot 0)

Solving LISE Recursively (Slow if we're not careful!)

- We can solve LISE on an array A with $< n$ recursive calls, each to a prefix of A
- Running time?
  - Something like $T(n) = T(n-1) + T(n-2) + \ldots + T(1) + O(n)$.
  - VERY large! $\Theta(n2^n)$
- Dynamic Programming to the rescue

LISE Using Dynamic Programming (Same idea, but fast!)

- Create an array L
- $L[i]$ stores LISE of $A[0, \ldots, i]$
- Let's fill in $L[i]$ for the above example [Blackboard] (Recall that we take $1+$ the max, over all previous smaller elements, of their LISE)

LISE Using Dynamic Programming

Let's formalize what we just did on the board.

- Base Case: What is $L[0]$? 1
- How to Fill in $L[i]$: First, create a set M consisting of all entries in A that are:
  - before i in A, and
  - less than $A[i]$
- $L[i] = 1 + \max_{m \in M} L[m]$ (so $L[i] = 1$ if $M = \emptyset$)
- Running time?
  - It takes $O(n)$ time to calculate $L[i]$
  - Do that for $i \in \{0, \ldots, n-1\}$ (n values)
  - $O(n^2)$

LIS Using Dynamic Programming

- Recursive algorithm, but
- Since we write down solutions as we get them, we obtain $O(n^2)$ running time rather than $\Theta(n2^n)$
- What about LIS?
- The Longest Increasing Subsequence must end at *some* entry j of A
- After we fill out the table L, can find:
  - LIS = $\max_j L[j]$
- $O(n^2)$ algorithm for Longest Increasing Subsequence

LIS Using Dynamic Programming

- First set $L[0] = 1$
- Fill out each $L[i]$ by finding previous elemements smaller than i and taking the max
- Take the max $L[i]$ after we are done to find the LIS

Dynamic Programming Structure

Dynamic Programming

- Memoized recursion
- All dynamic programs have a common structure (which we'll go over on the next slide)
- To ensure clarity in expectations, I will always ask you to use this structure when giving a dynamic program
- Some of these entries may be *very easy* to fill out!
- On this assignment, I have written these out specifically for you. But I may not do that the future
- You do not need to memorize this; I will give it to you on midterm/exams

Elements of a Dynamic Program

- Subproblem Definition: what subproblem are you using in your DP?
- Memoization Structure: what data structure are you using? (Almost always an array. But: how big? How many dimensions?)
- Recurrence: State the recurrence used for the DP.
- Base Case: Base case for the recurrence/first entry we can fill out in the table
- Evaluation Order: In what order should we fill out our table? (Almost always left to right for 1-D tables.)
- Final Solution: After we filled out the table, how do we read off the final solution?
- Time and Space Analysis

Elements of a Dynamic Program: LIS

- Subproblem Definition: Longest increasing subsequence ending at an element i
- Memoization Structure: 1-dimensional array of length n (this was $L[]$)
- Recurrence $L[i] = 1 + \max_{m \in M} L[m]$ where $m = \{j \mid j < i, A[j] < A[i]\}$
- Base Case: $L[0] = 1$
- Final Solution: $\max_{i \in \{0,\ldots,n-1\}} L[i]$
- Evaluation Order: Calculate $L[j]$ for $j = 0$ to $n - 1$ (left to right)
- Time and Space Analysis: $O(n^2)$ time, $O(n)$ space

Improving LIS

2	1	10	3	7	6	4	8	11	5

- Can we calculate the LIS more quickly than $\Theta(n^2)$?
- (Seems like we're wasting some time on the above.)
- Yes! We'll see next week how to solve this in $O(n \log n)$ time
- (I want to focus on how DP works, rather than optimizing speed, for now.)

Approaching a Dynamic Programming Problem

Designing a DP

- The first question to ask yourself is: what does a solution look like?
- Usually: there are several *cases* for what a solution looks like
- We can recursively figure out the cost of the solution for each case
- Taking the best cost gives us the best answer!

LISE in Cases

- We want to find the longest increasing subsequence ending at position $n - 1$
- What does a solution look like?
  - The element at position $n - 1$ is in the solution
  - Either the solution has length 1, or there is a *second-to-last* element
  - The second-to-last element is smaller than the element at position $n - 1$
- **Cases:** there are $n - 2$ possible second-to-last elements
- For each second-to-last element, I want the best solution ending at that element.
  - Recursive call!

Phrased Another Way

The optimal solution ending at element $n-1$ consists of:

- Element $n-1$, and
- The optimal solution ending at some earlier element with value less than the $n-1$st element

Can someone give a short proof of why we always want the *optimal* solution ending at element $n-1$?

Weighted Interval Scheduling