Divide and Conquer

Sam McCauley

March 17, 2025

Welcome Back!

- Due date wrong on homework (it’s due Thursday)
- Let’s look at the midterm
- Then get back to D & C

More Recurrences

Divide and Conquer and Recurrences

- We analyze divide and conquer algorithms using *recurrences*
- Gives us a bird's eye view of the cost of the algorithm
- Recurrence relations can also guide us in searching for algorithms
  - "How can I sort in $O(n \log n)$ time?"
  - If my sorting method recurses on two halves, and does $O(n)$ additional work, I get $T(n) = 2T(n/2) + O(n)$, which gives $O(n \log n)$
  - (Of course, this is just a starting point: many other recurrences solve to $O(n \log n)$.)
- Let's look at some other recurrences

Three practice recurrences

Let's do the following recurrences [Blackboard]

For all of these assume $T(1) = 1$.

$$T(n) = 4T(n/2) + O(1)$$

$$T(n) = 3T(n/3) + O(n)$$

Recall: Floors and Ceilings in Recurrences

- Most input sizes are not (say) powers of 2
- Merge sort's actual recurrence is:

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n)$$

- Does this change the solution?
- No. *We will ignore all floors and ceilings in this class.* See Erikson 1.7 for some formal justification

Tree Height and Recurrences that Don't Branch

Let's do the following recurrences [Blackboard]

$$T(n) = T(n/2) + O(1)$$

$$T(n) = T(\sqrt{n}) + O(1)$$

$$T(n) = T(n/2) + O(n)$$

Lemma: if we start with n, after taking the square root $\Theta(\log \log n)$ times we reach 1.

Three kinds of recurrences

Recurrences often fit into one of three types:

- Cost at the root dominates
- Cost at the leaves dominate
- Cost at each level is the same

Ways to Solve Recurrences

- Recursion tree (recommended)
- Guess and check
  - If we have the solution for $T(n)$, we can substitute it into the recurrence to check that it is satisfied
  - Can formalize using induction
  - “Unroll” recurrence a few steps to get intuition before guessing
- Master theorem (next slide) *gives* the solution for many common recurrences

Master Theorem (Simple Version)

For *constants* a and b and a function $f(n)$, to solve

$$T(n) = aT(n/b) + f(n); \qquad T(1) = 1$$

- If $f(n) = O(n^c)$ for $c < \log_b a$ then $T(n) = \Theta(n^{\log_b a})$
  - So $T(n) = 4T(n/2) + O(n)$ solves to $T(n) = \Theta(n^2)$
- If $f(n) = \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} \log n)$
  - So $T(n) = 2T(n/2) + O(n)$ solves to $T(n) = \Theta(n \log n)$
- A fast way to solve simpler recurrences. But a pain to memorize and only works situationally.

Binary Search

Binary Search

```
1 binary_search(key, A, start, end):
2     mid = (start + end)/2
3     if key == A[mid]:
4         return mid
5     else if key < A[mid]:
6         return binary_search(key, A, start, mid-1)
7     else:
8         return binary_search(key, A, mid+1, end)
```

- Correctness intuition: we recurse on the half of `A` that *must contain* `key`.
- How would we prove correctness formally?
- Running time? $T(n) = T(n/2) + O(1)$ We've seen: $T(n) = O(\log n)$

Binary Search on a Linked List?

This is *not a good algorithm*. But I've seen people implement it many times.

Today: how efficient is it?

We can binary search by:

- Find the middle item of the linked list
  - By iterating through the linked list
- Compare to query item
- Recurse on first or second half of the linked list
- Recurrence?
- $T(n) = T(n/2) + \Theta(n)$
- Solution: $\Theta(n)$ time
- (Could have just scanned!)

Matrix Multiplication

Matrix Multiplication

Problem: For two $n \times n$ matrices A and B, compute matrix $C = A \cdot B$.

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix}$$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

Matrix Multiplication

$$\begin{bmatrix} & & & \\ & \square & & \\ & & & \\ & & & \end{bmatrix} = \begin{bmatrix} 7 & 12 & 21 & 5 \\ 18 & 2 & 24 & 11 \\ 19 & 9 & 17 & 13 \\ 28 & 6 & 10 & 4 \end{bmatrix} \times \begin{bmatrix} 14 & 30 & 1 & 22 \\ 5 & 18 & 19 & 9 \\ 8 & 7 & 13 & 25 \\ 11 & 16 & 20 & 3 \end{bmatrix}$$

The value of c_{23} is $18 \cdot 1 + 2 \cdot 19 + 24 \cdot 13 + 11 \cdot 20 = 588$.

Matrix Multiplication

Problem: For two $n \times n$ matrices A and B, compute matrix $C = A \cdot B$.

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

$$\mathbf{C} = \mathbf{A} \times \mathbf{B} = \begin{bmatrix} \sum_{k=1}^{n} a_{1k} b_{k1} & \sum_{k=1}^{n} a_{1k} b_{k2} & \cdots & \sum_{k=1}^{n} a_{1k} b_{kn} \\ \sum_{k=1}^{n} a_{2k} b_{k1} & \sum_{k=1}^{n} a_{2k} b_{k2} & \cdots & \sum_{k=1}^{n} a_{2k} b_{kn} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{k=1}^{n} a_{nk} b_{k1} & \sum_{k=1}^{n} a_{nk} b_{k2} & \cdots & \sum_{k=1}^{n} a_{nk} b_{kn} \end{bmatrix}$$

Matrix Multiplication

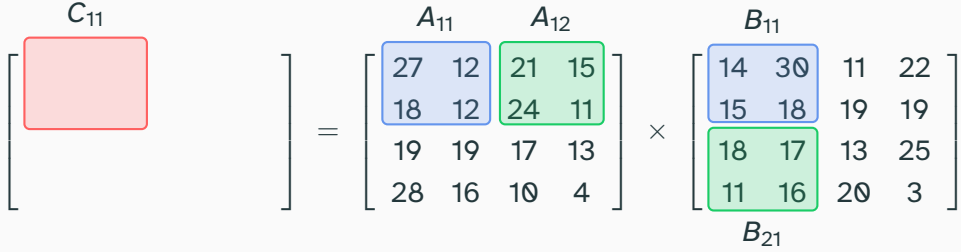
- Used *extremely* often in modern computing
  - Graphics, machine learning applications; many others
- GPUs are basically purpose-built hardware to do matrix multiplication quickly
  - Not literally; what they do is significantly broader than that
  - But it's not a coincidence that areas where matrix multiplication are most used are also known for heavy GPU requirements

Matrix Multiplication Algorithm

- First attempt: do all multiplications
- Running time?
  - C is an $n \times n$ matrix so has n^2 entries
  - Each takes $O(n)$ time to compute
  - $O(n^3)$ time in total
- Can we do better?
- Seems difficult, since there are $\Omega(n^3)$ terms we need to multiply across all entries of C

Block Matrix Multiplication

If we divide each matrix into four pieces, can rewrite matrix multiplication using recursive calls.



In other words: in a 2×2 matrix, we have $c_{11} = a_{11} \cdot b_{11} + a_{12} \cdot b_{21}$. We can fill in an *entire* quadrant of C by multiplying the quadrants of A and B:

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}.$$

Divide and Conquer Matrix Multiplication

We have:

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$
$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$$
$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$$
$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$$

This gives a divide and conquer algorithm!

- How many *recursive calls* do we make? How large are they? How much extra work do we need to do?
- 8 calls of size $n/2$. Takes $O(n^2)$ time in total to do all additions
- $T(n) = 8T(n/2) + O(n^2)$. Let's solve [Blackboard]
- Answer: $T(n) = O(n^3)$. No gain over normal matrix multiplication!

Improving Blocked Matrix Multiplication

- What do we want to improve?
- It would be nice if we could improve the $O(n^2)$. But seems difficult...
- What if we could reduce the number of calls from 8? Would that make a difference?
  - Yes. This affects running time by *more than a constant*
  - Same thing we saw with large-integer multiplication

Let’s look at a magic algorithm.

Strassen's Algorithm

First, let's define 7 completely random-looking matrices (1 multiplication each):

$$P_1 = A_{11} \cdot (B_{12} - B_{22}) \qquad P_2 = (A_{11} + A_{12}) \cdot B_{22}$$
$$P_3 = (A_{21} + A_{22}) \cdot B_{11} \qquad P_4 = A_{22} \cdot (B_{21} - B_{11})$$
$$P_5 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \qquad P_6 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$$
$$P_7 = (A_{11} - A_{21}) \cdot (B_{11} + B_{12})$$

Now we can calculate C using only addition and subtraction:

$$C_{11} = P_5 + P_4 - P_2 + P_6$$
$$C_{12} = P_1 + P_2$$
$$C_{21} = P_3 + P_4$$
$$C_{22} = P_1 + P_5 - P_3 - P_7$$

Analyzing Strassen's Algorithm

- Need to do 7 recursive calls, each of size $n/2$
- Then add together in $O(n^2)$ time
- $T(n) = 7T(n/2) + O(n^2)$; $T(1) = O(1)$ [Blackboard]
- $T(n) = O(n^{\log_2 7}) = O(n^{2.81})$

Matrix Multiplication

- Surprising that we don't need to do all the calculations
- How fast can matrix multiplication run?
  - One of the best-known open problems in algorithms
  - It is unknown if a linear time $O(n^2)$ algorithm is possible
- Best algorithm as of today: $O(n^{2.371339})$ [Alman et al. 2024]

Selection

Median finding

- Goal: given an *unsorted* array A of length n, find the median of A
- Can someone give an $O(n \log n)$ time algorithm to solve this?
- Sort A using Merge Sort. Return $A[\lceil n/2 \rceil]$
- Can we do better?

Linear-Time Median Finding

- Goal: an $O(n)$ algorithm to find the median of any unsorted array A
- Can't sort! (Sorting takes $\Omega(n \log n)$ time.) Is it really possible to find the median of an array without sorting it?
- We'll solve a more general problem: find the kth largest element in the array
- Divide and conquer algorithm; invested by Blum, Floyd, Pratt, Rivest, Tarjan 1973

Partition (Selection Subroutine)

```
1  Partition(A, p):
2      Create empty arrays A_<p and A_>p
3      for i = 0 to |A| − 1:
4          if A[i] < p:
5              add A[i] to A_<p
6          if A[i] > p:
7              add A[i] to A_>p
8      return |A_<p|, A_<p, A_>p
```

Returns two arrays, one with elements $< p$ and one with elements $> p$

The *rank* of p is the number of elements in A smaller than p; also returns the rank of p.

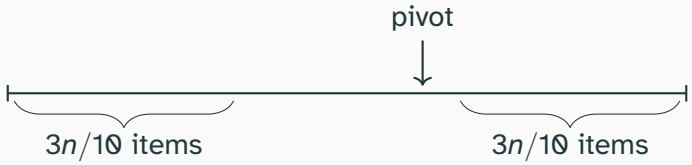
Selection (First Attempt)

```
1  Select(A, k):
2      if |A| = 1:
3          return A[0]
4      else:
5          choose a pivot p # we'll define how later
6          r,A_<p,A_>p = Partition(A, p)
7          if k == r:
8              return p
9          else:
10             if k < r:
11                 return Select(A_<p, k)
12             else:
13                 return Select(A_>p, k − r − 1)
```

The main question is: How do we select our pivot? (And how does that impact performance?)

How good does our pivot selection need to be?

- Let's say our pivot is *not* in the first or last $3n/10$ items of A (where $n = |A|$)
- It's in the middle $4n/10$ items
- What is our recurrence?
- $T(n) \leq T(7n/10) + O(n)$
- $T(n) = O(n)$

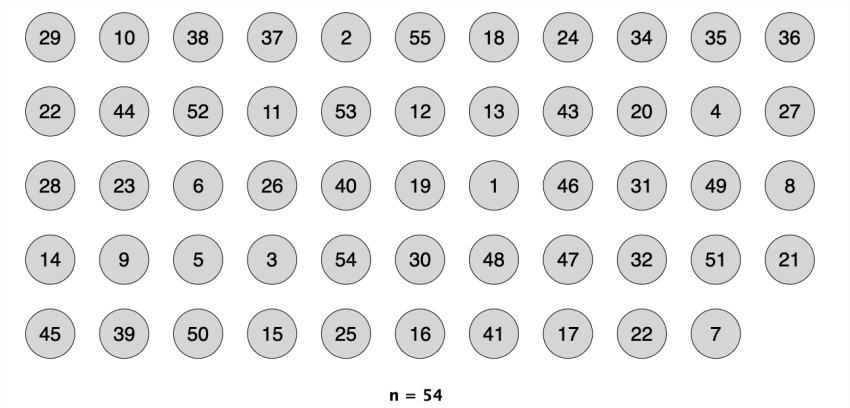


Finding the Pivot: Goal

- Find a pivot that has rank between $3n/10$ and $7n/10$ in time $O(n)$
- The array is *unsorted*
- Want to always be successful
- Note: Can verify in $O(n)$ time!
- In practice: pick a random number to be the pivot. You'll probably find a good one pretty quickly.

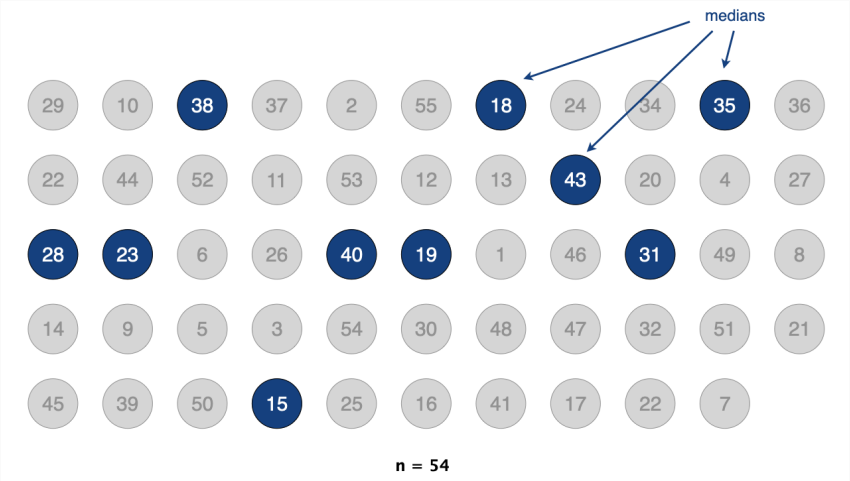
Finding an Approximate Median

- Divide the array into $\lceil n/5 \rceil$ groups of 5 elements (ignore leftovers)
- Find median of each group



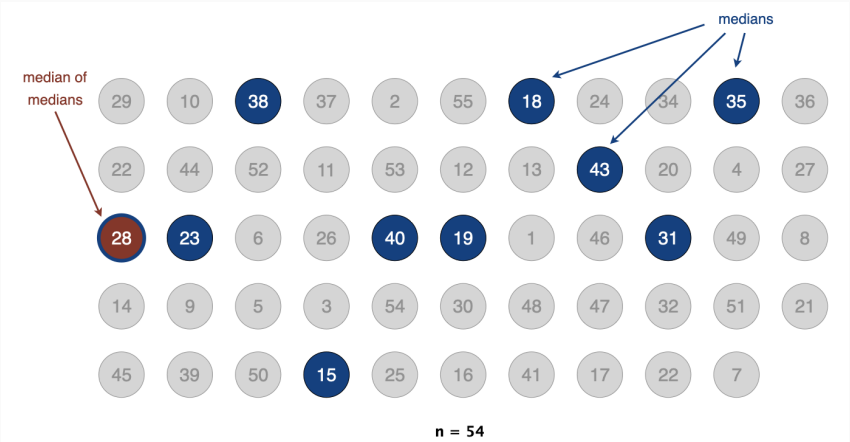
Finding an Approximate Median

- Divide the array into $\lceil n/5 \rceil$ groups of 5 elements (ignore leftovers)
- Find median of each group



Finding an Approximate Median

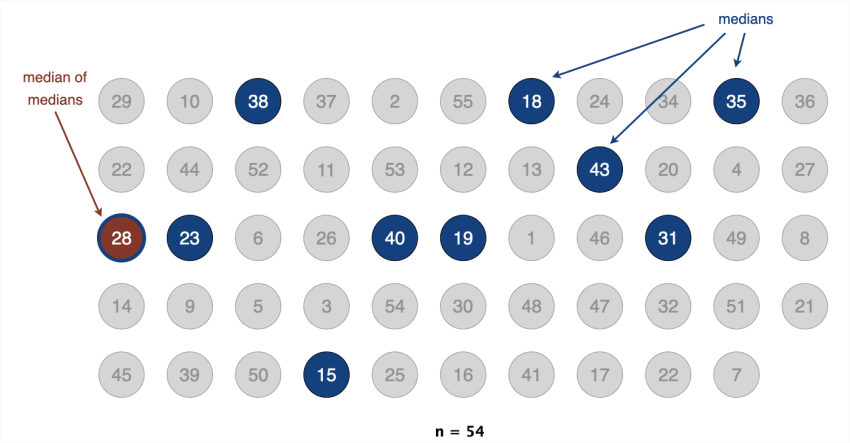
- Divide the array into $G = \lceil n/5 \rceil$ groups of 5 elements (ignore leftovers)
- Find median of each group
- Find the median of these G medians (the element of rank $\lfloor G/2 \rfloor$); this is our pivot



Finding an Approximate Median

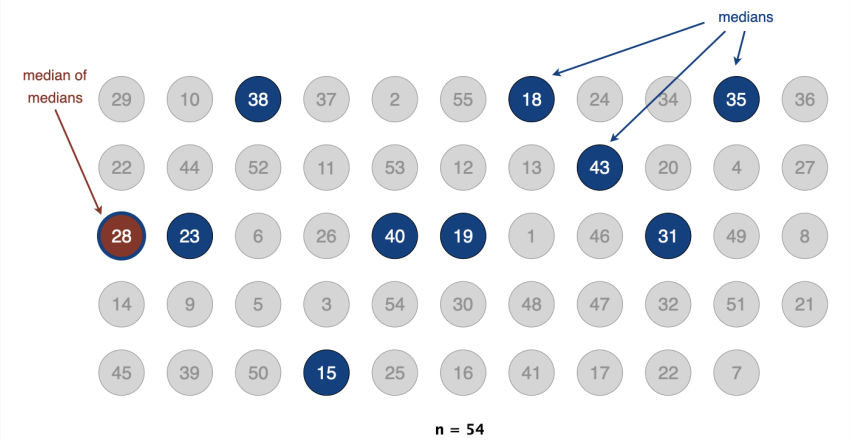
- Divide the array into $G = \lfloor n/5 \rfloor$ groups of 5 elements (ignore leftovers)
- Find median of each group
- Find the median of these G medians (the element of rank $\lceil G/2 \rceil$); this is our pivot (call it M)
- How can we find the median of these medians? Recursively!
    - This is a median-finding algorithm! We call `Select` to find the median of these medians to get our pivot

Rank of the Median of Medians



- What elements are smaller than the median of medians M?
- Half the medians (at least $G/2 \approx n/10$ elements)
- Also: for each such median, two elements in the median's list ($2n/10$ elements)

Rank of the Median of Medians



- $\geq 3n/10$ are less than M
- Similarly: $\geq 3n/10$ are greater than M
- So M is a good pivot!

Linear-Time Selection

```
1  Select(A, k):
2      if |A| ≤ 5:
3          return kth largest element of A
4      else:
5          divide A into ⌊n/5⌋ groups of 5 elements
6          Create array A_m containing the median of each group
7          p = Select(A_m, ⌈|A_m|/2⌉)
8          r, A_<p, A_>p = Partition(A, p)
9          if k == r:
10             return p
11         else:
12             if k < r:
13                 return Select(A_<p, k)
14             else:
15                 return Select(A_>p, k − r − 1)
```

Recurrence: $T(n) = T(n/5) + T(7n/10) + O(n)$; $T(5) = O(1)$ [Blackboard]

Median Finding

- An advanced Divide and Conquer application
- Uses a nontrivial recurrence
- Can find median of an unsorted array in $O(n)$ time—strictly faster than sorting!

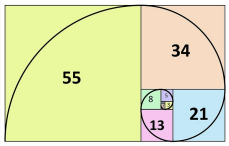
Dynamic Programming

Algorithmic Design Paradigms

- Greedy Algorithms
  - Gas-filling; maximum interval scheduling
  - Prim's, Kruskal's, Dijkstra's
  - Idea: we choose an item to add *permanently* to the solution
  - Proof that each item we have is correct
- Divide and Conquer
  - Divide problem into multiple parts
  - *Combine* solutions into a new correct solution
- Dynamic Programming ⇐ we are here!
  - Use recursive solutions *repeatedly* to avoid wasted work
- Network Flow

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]
```

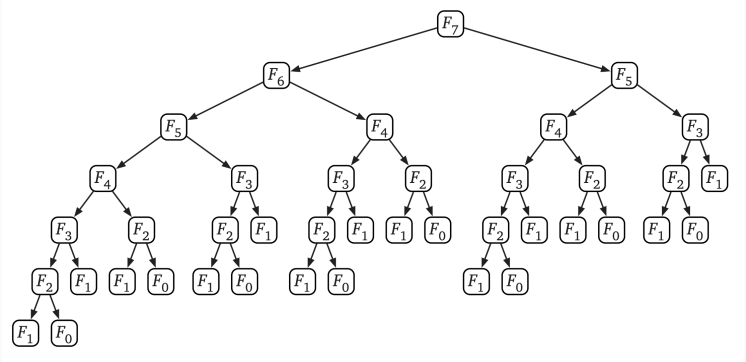
- Correctness: when we fill in $F[j]$, we have already put the correct value in $F[j-1]$ and $F[j-2]$
- What is the running time?
- $O(n)$
- Can calculate the 200th Fibonacci number with ≈ 200 additions!

What happened here?

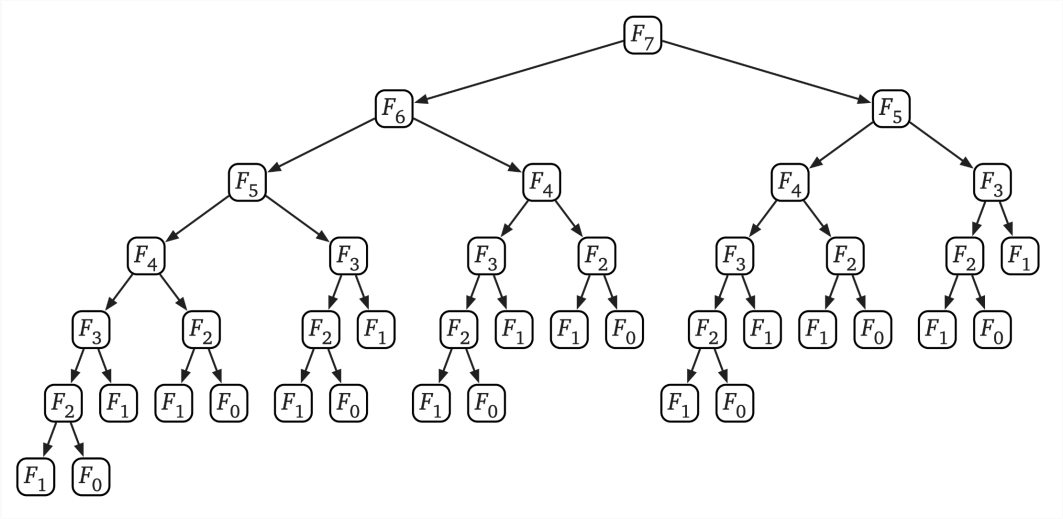
- Both algorithms seem reasonable
- One takes 200 operations, the other takes 10^{41} operations
- Where are we *losing time*?

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

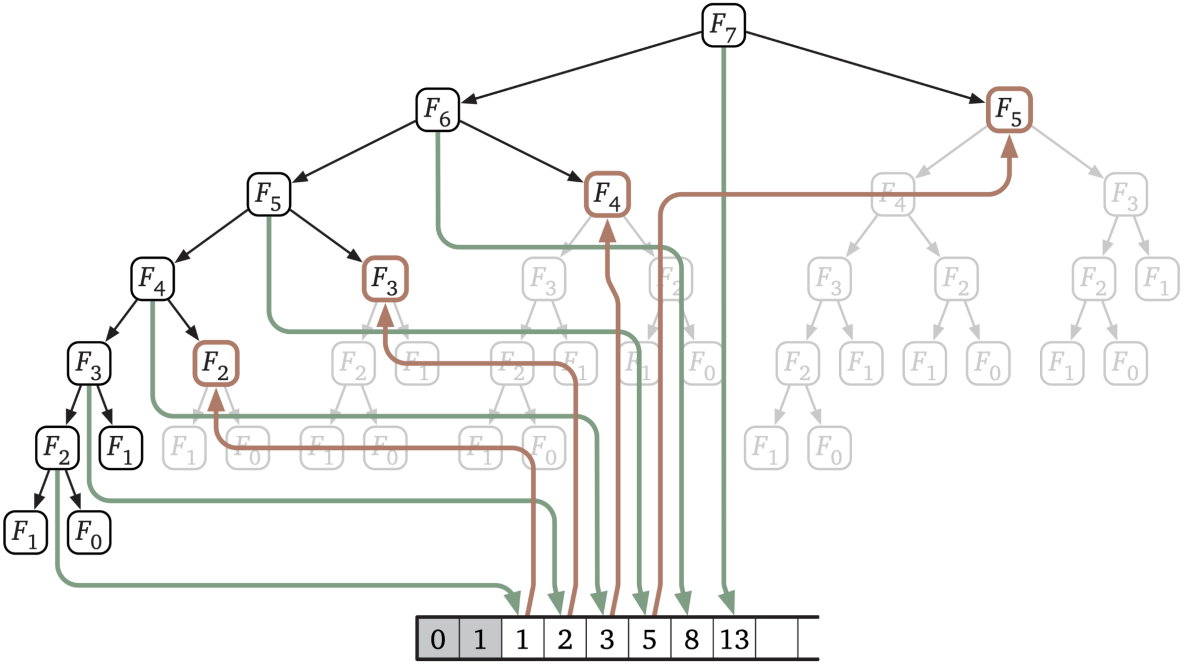


Recursive Fibonacci Slowness

- 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 sorted 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 $i_1 \leq i_2 \leq \ldots \leq i_\ell$ such that for all $k < \ell$, $A[i_k] < A[i_{k+1}]$
- Also called LIS
- 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 $A[n - 1]$
- Let's focus on LISE for now. Then we'll double back for LIS

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 $2, 1, 3, 4$ 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
  - The longest increasing subsequence ending at 3
  - The longest increasing subsequence ending at 1
  - The longest increasing subsequence ending at 2

Solving LISE Recursively

- 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

2	1	10	3	7	6	4	8	11	5

- 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 $A[i]$
- $L[i] = 1 + \max_{m \in M} L[m]$
- Running time? $O(n^2)$

LISE 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

LISE 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