Programming Abstractions

CS106B

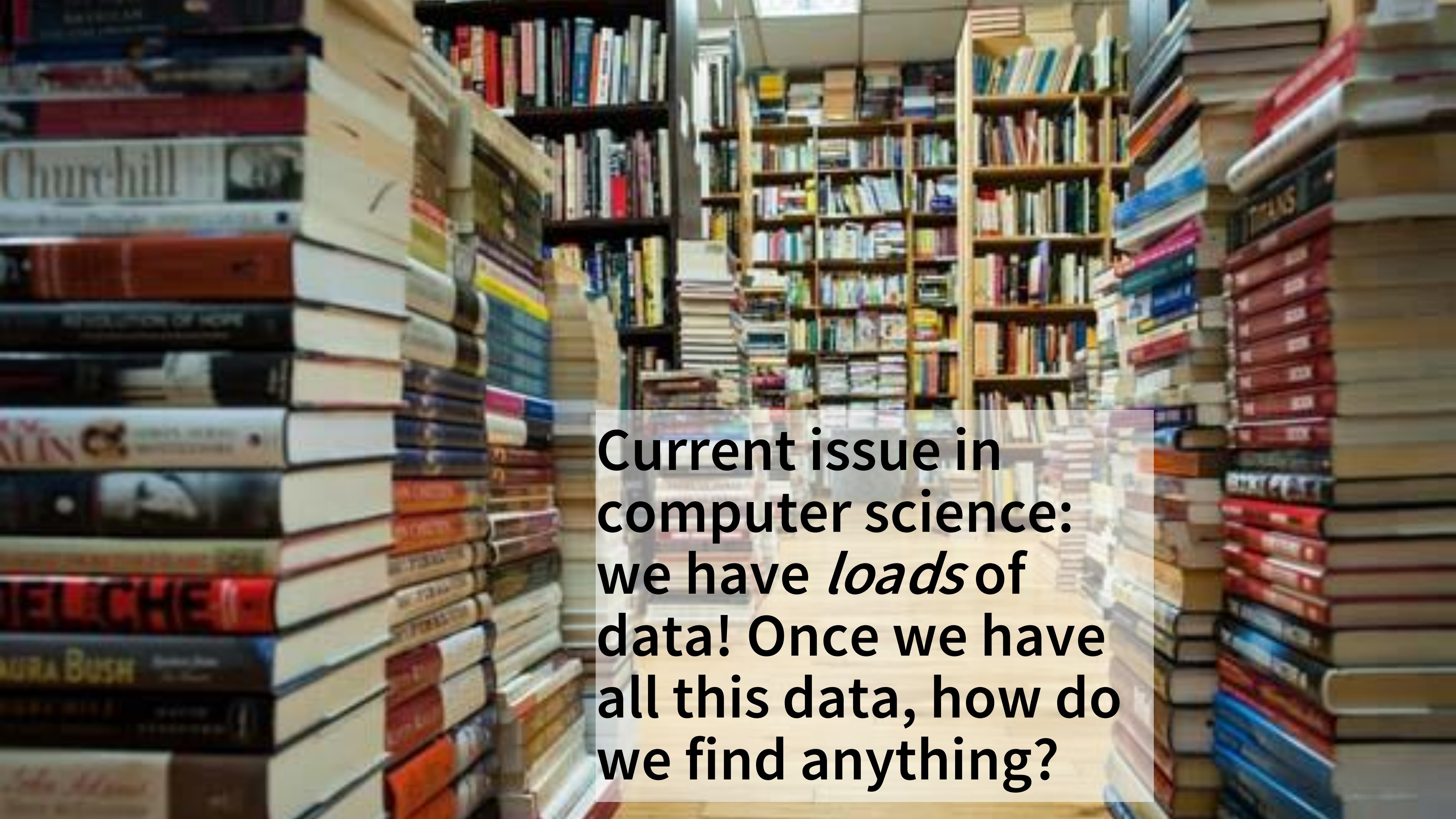
Cynthia Bailey Lee
Julie Zelenski

Today's Topics:

- Contrasting performance of 3 recursive algorithms
- Quantifying algorithm performance with Big-O analysis
- Getting a sense of scale in Big-O analysis

Binary Search

AN ELEGANT SOLUTION TO THE PROBLEM OF TOO MUCH DATA



Current issue in computer science: we have *loads* of data! Once we have all this data, how do we find anything?

Does this list of numbers contain X?

The question we're trying to answer is, given a list of numbers, does this list contain some particular value, or not? For convenience, we have kept our list **sorted**.

How long does it take us to find a number we are looking for?

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

Does this list of numbers contain X?

The question we're trying to answer is, given a list of numbers, does this list contain some particular value, or not? For convenience, we have kept our list **sorted**.

How long does it take us to find a number we are looking for?

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

If you start at the front and proceed forward, each item you examine rules out 1 item

Does this list of numbers contain X?

The question we're trying to answer is, given a list of numbers, does this list contain some particular value, or not? For convenience, we have kept our list **sorted**.

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

If instead we **jump right to the middle**, one of three things can happen:

1. The middle one happens to be the number we were looking for, yay!
2. We realize we went too far
3. We realize we didn't go far enough

Does this list of numbers contain X?

The question we're trying to answer is, given a list of numbers, does this list contain some particular value, or not? For convenience, we have kept our list **sorted**.

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

If instead we **jump right to the middle**, one of three things can happen:

1. The middle one happens to be the number we were looking for, yay!
2. We realize we went too far
3. We realize we didn't go far enough

Ruling out HALF the options in one step is so much faster than only ruling out one!

Binary search

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

Let’s say the answer was case 3, “we didn’t go far enough”

- We ruled out the entire first half, and now only have the second half to search
- We could start at the front of the second half and proceed forward checking each item one at a time…

Binary search

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

Let's say the answer was case 3, "we didn't go far enough"

- We ruled out the entire first half, and now only have the second half to search
- We could start at the front of the second half and proceed forward checking each item one at a time… but why do that when we know we have a **better way**?

Jump right to the middle of the region to search

Binary search

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

Let's say the answer was case 3, "we didn't go far enough"

- We ruled out the enti fir alf, and now only have the second half t ar
- We cou econd half and proceed forward c a time… but why do that wh r way?

RECURSION!!

Jump righ t f the region to search

Binary Search pseudocode

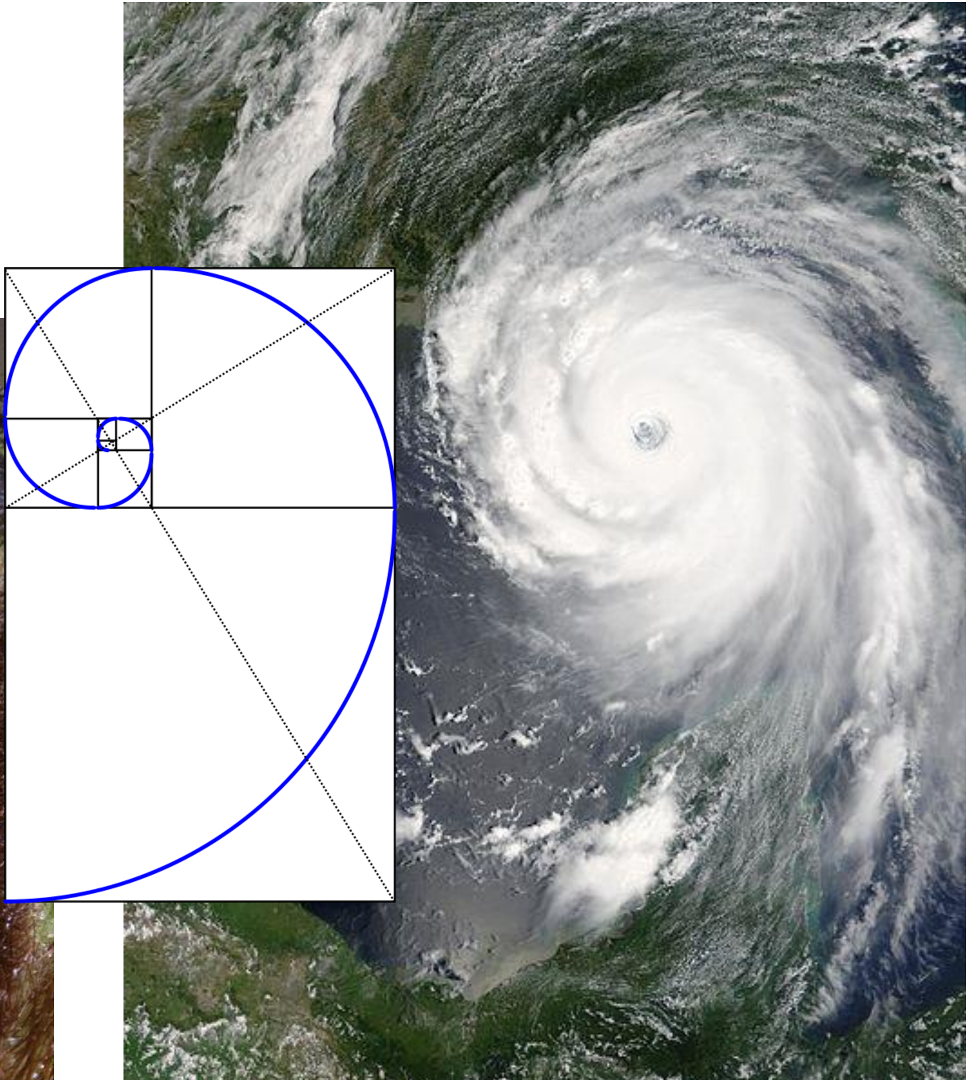
- We'll write the real C++ code together on Friday, but here's the outline/pseudocode of how it works:

```
bool binarySearch(Vector<int>& data, int key)
{
    if (data.size() == 0) {
        return false;
    }
    if (key == data[midpoint]) {
        return true;
    } else if (key < data[midpoint]) {
        return binarySearch(data[first half only], key);
    } else {
        return binarySearch(data[second half only], key);
    }
}
```

The Fibonacci Sequence

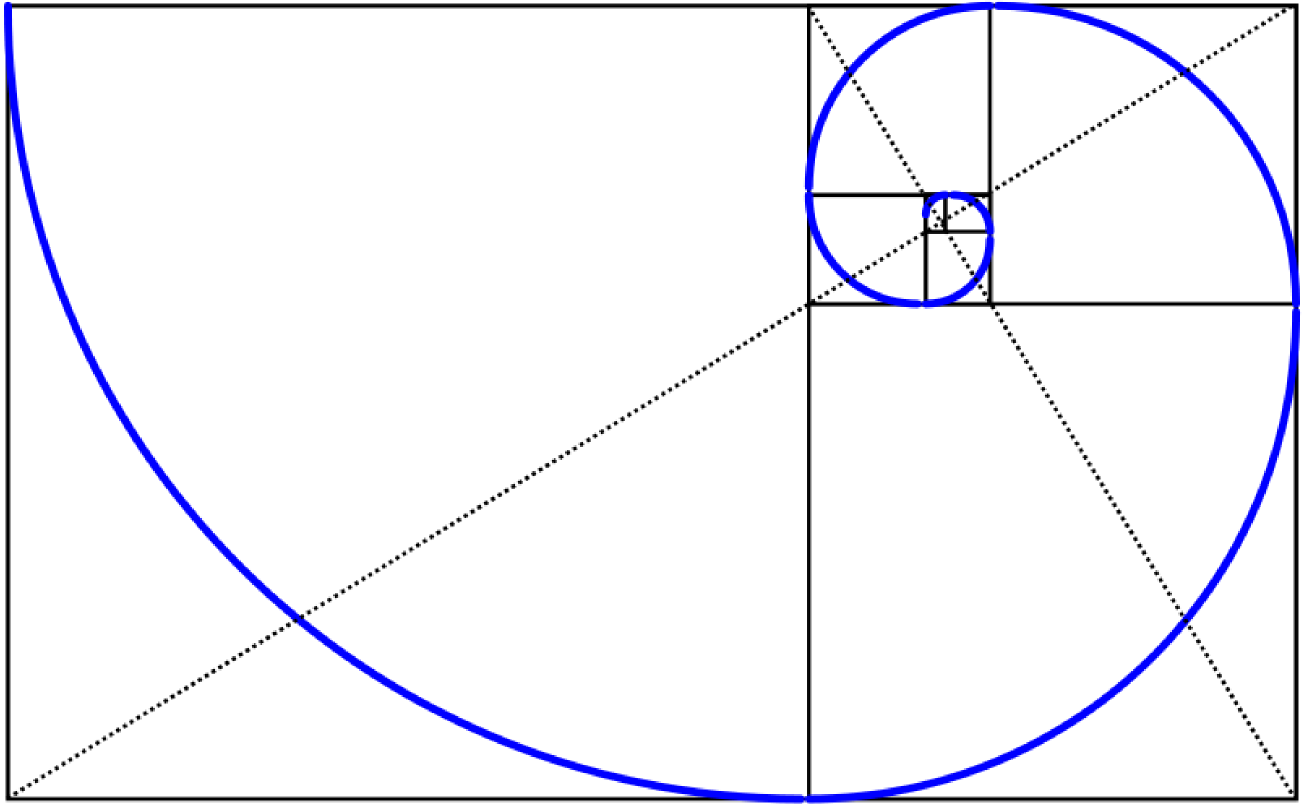
MATH NERD REJOICING INTENSIFIES

Fibonacci in nature



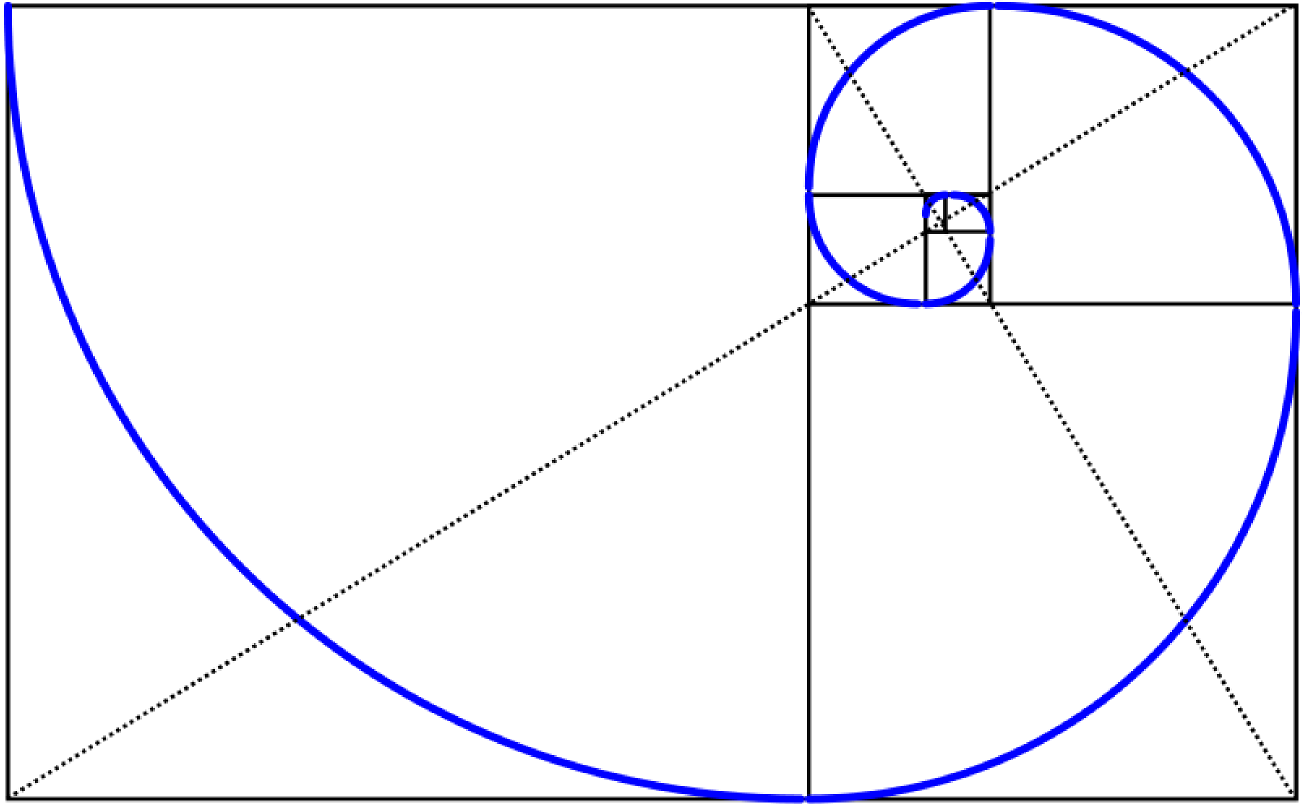
Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,



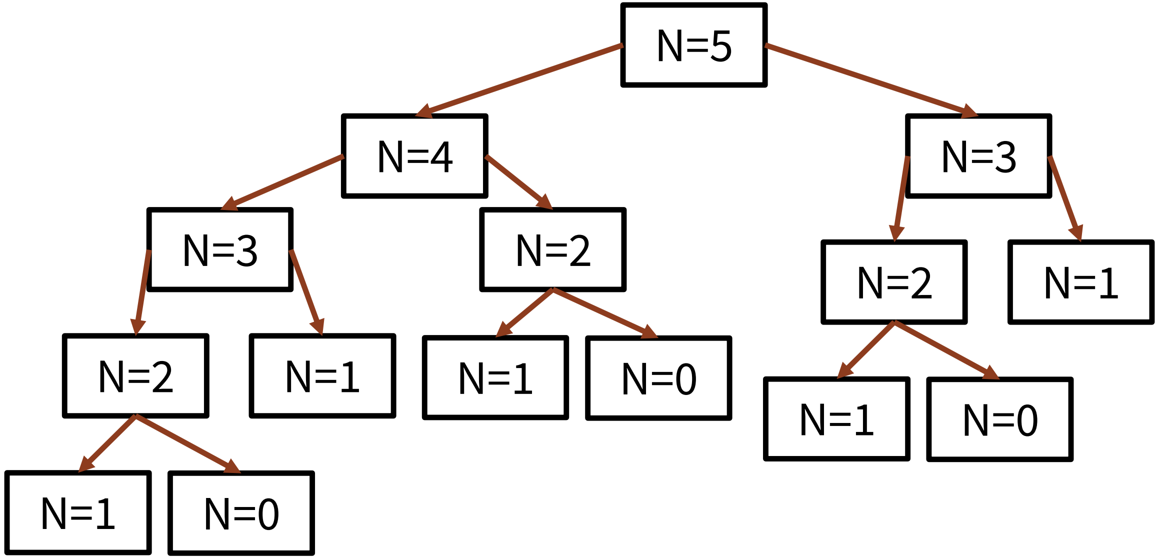
Fibonacci

0	1	2	3	4	5	6	7	8	9	10	11
0	1	1	2	3	5	8	13	21	34	55	89



Fibonacci

```
int fib(int n)
{
    if (n == 0) {
        return 0;
    } else if (n == 1)
        return 1;
    } else {
        return fib(n - 1) + fib(n - 2);
    }
}
```

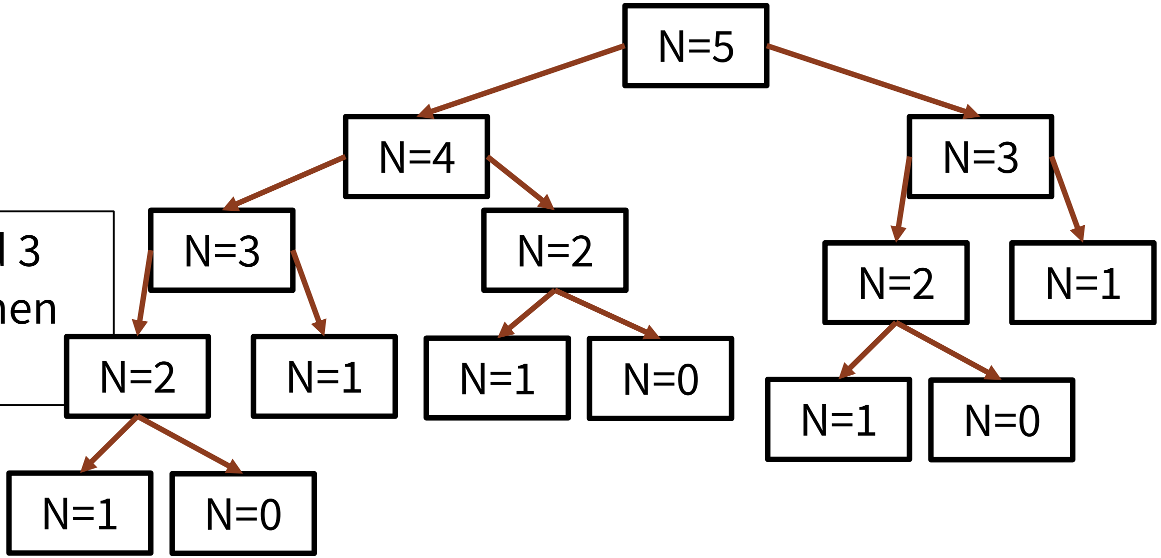


Work is duplicated throughout the call tree

- fib(2) is calculated 3 separate times when calculating fib(5)!
- 15 function calls in total for fib(5)!

Fibonacci

fib(2) is calculated 3 separate times when calculating fib(5)!



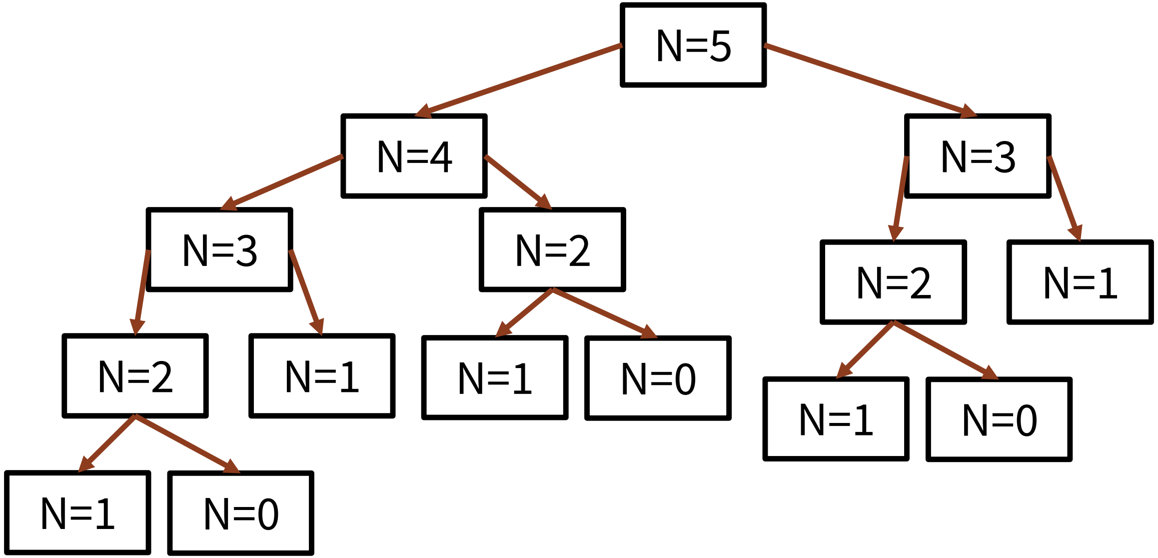
How many times would we calculate fib(2) while calculating fib(6)?

See if you can just "read" it off the chart above.

A. 4 times
B. 5 times
C. 6 times
D. Other/none/more

Fibonacci

N	fib(N)	# of calls to fib(2)
2	1	1
3	2	1
4	3	2
5	5	3
6	8	
7	13	
8	21	
9	34	
10	55	



Efficiency of naïve Fibonacci implementation

When we **added 1** to the input N, the number of times we had to calculate fib(2) **nearly doubled** (~1.6* times)

- Ouch!

* This number is called the "Golden Ratio" in math—cool!

Goal: predict how much time it will take to compute for arbitrary input N.

Calculation: "approximately" $\mathbf{(1.6)^N}$

Big-O Performance Analysis

A WAY TO COMPARE THE NUMBER OF STEPS TO RUN THESE FUNCTIONS

Big-O analysis in computer science

Vector × +

web.stanford.edu/dept/cs_edu/resources/cslib_docs/Vector.html

The Stanford `libcs106` library, Fall Quarter 2021

`#include "vector.h"`

`class Vector<ValueType>`

This class stores an ordered list of values similar to an array. It supports traditional array selection u
square brackets, as well as inserting and removing elements. Operations that access elements by in
in O(1) time. Operations, such as insert and remove, that must rearrange elements run in O(N) time

Constructor

`Vector()`	O(1)	Initializes a new empty vector.
`Vector(n, value)`	O(N)	Initializes a new vector storing *n* copies of the given v

Methods

Big-O analysis in computer science

Article | Talk | Read | Edit | View history | Search Wikipedia

WIKIPEDIA
The Free Encyclopedia

- Main page
- Contents
- Featured content
- Current events
- Random article
- Donate to Wikipedia
- Wikipedia store

Interaction

- Help
- About Wikipedia
- Community portal
- Recent changes
- Contact page

Tools

- What links here
- Related changes
- Upload file
- Special pages
- Permanent link

Binary search algo

From Wikipedia, the free encyclopedia
(Redirected from Binary search)

This article is about searching a fi

In computer science, **binary search**,
is a search algorithm that finds the po
target value to the middle element of
and the search continues on the rema
empty, the target is not in the array.

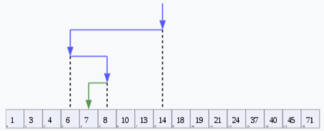
Worst-case performance	$O(\log n)$
Best-case performance	$O(1)$
Average performance	$O(\log n)$
Worst-case space complexity	$O(1)$

Binary search runs in at worst logarithmic time, making $O(\log n)$ comparisons, where n is the number of elements in the array, the O is Big O notation, and log is the logarithm. Binary search takes constant ($O(1)$) space, meaning that the space taken by the algorithm is the same for any number of elements in the array.[6] Although specialized data structures designed for fast searching—such as hash tables—can be searched more efficiently, binary search applies to a wider range of problems.

Although the idea is simple, implementing binary search correctly requires attention to some subtleties about its exit conditions and midpoint calculation.

There are numerous variations of binary search. In particular, fractional cascading speeds up binary searches for the same value in multiple arrays, efficiently solving a series of search problems in computational geometry and numerous other fields. Exponential search extends binary search to unbounded lists. The binary search tree and B-tree data

Binary search algorithm



Visualization of the binary search algorithm where 7 is the target value.

Class	Search algorithm
Data structure	Array
Worst-case performance	$O(\log n)$
Best-case performance	$O(1)$
Average performance	$O(\log n)$
Worst-case space complexity	$O(1)$

Formal definition of big-O

We say a function $f(n)$ is "big-O" of another function $g(n)$
(written "$f(n)$ is $O(g(n))$")

if and only if

there exist positive constants c and n_0 such that

$f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

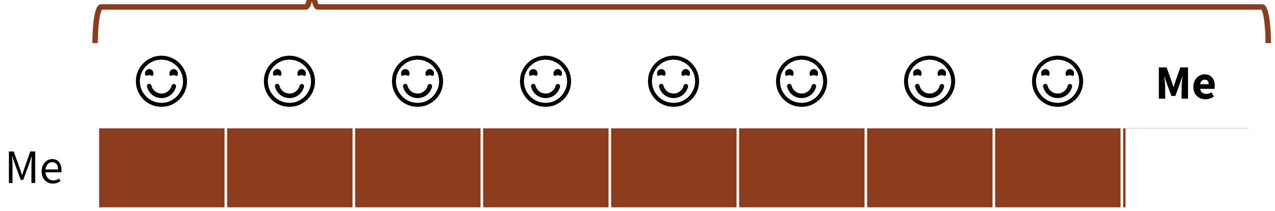
Before we start, let's get introduced

Before we start, let’s get introduced

Lets say I want to meet each of you today with a handshake and *you* tell *me* your name…

How many introductions need to happen?

There are **N** people in the room including me



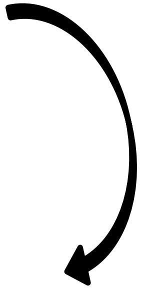
But do I need to shake hands with myself, or tell myself my name?

N-1 introductions

Putting this in Big-O terms

Big-O is a way of categorizing amount of work to be done in general terms, with a focus on:

- ***Rate of growth*** as a function of the problem size N
- What that rate looks like ***on the horizon*** (i.e., for large N)

Therefore, we don't really care about an insignificant ±1 

Putting this in Big-O terms

For the first handshake problem, the rate N is important and the -1 constant is not, so **N – 1** introductions becomes:

$$O(\ N-1\ )$$

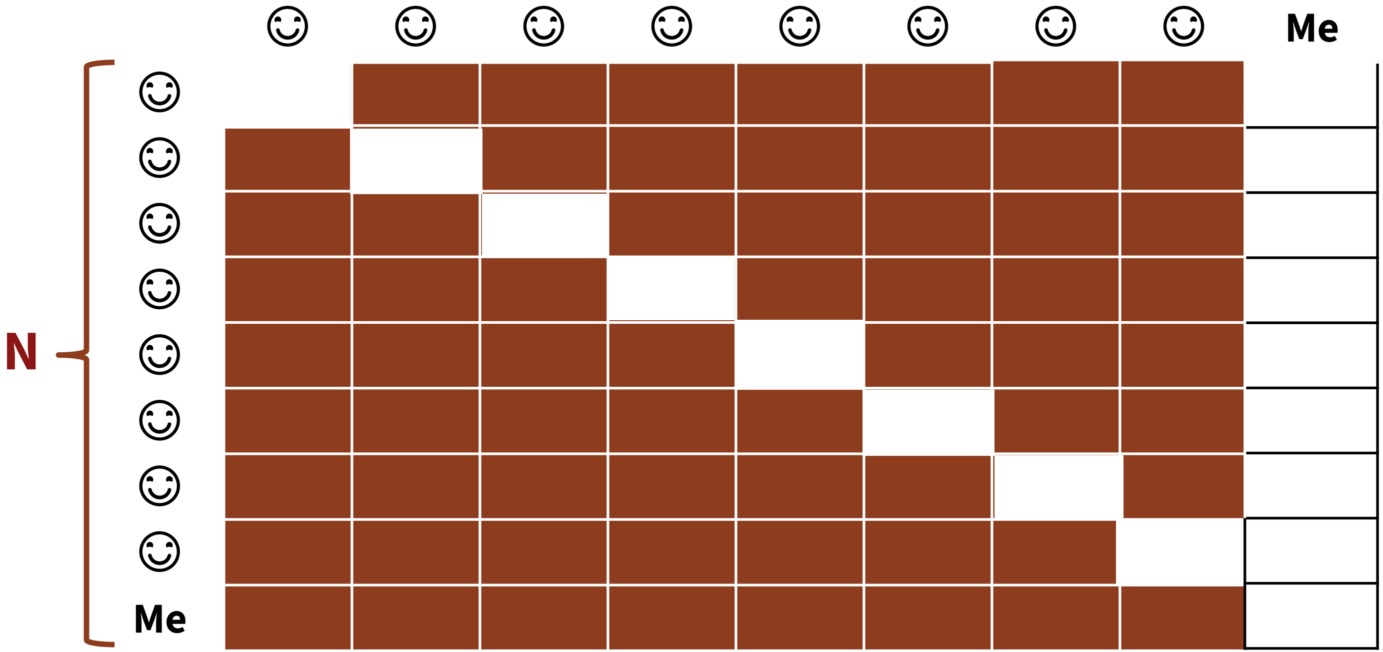
Similarly, if we said that each introduction **takes 3 seconds**, the amount of time is **3(N – 1) = 3N – 3**, but we disregard the constant 3s:

$$O(3N - 3)$$

Before we start, let's get introduced

What if I not only want you to be introduced to me, but to each other?

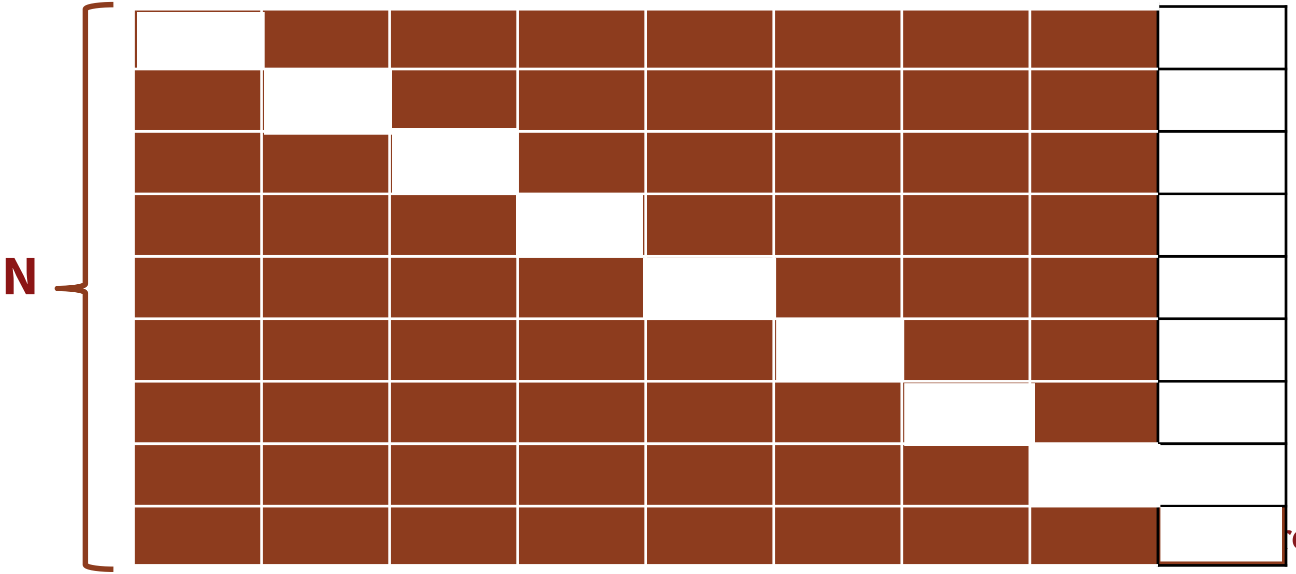
Now how many introductions? **N^2**



Before we start, let's get introduced

What if I not only want you to be introduced to me, but to each other?

Now how many introductions? $\mathbf{N^2 - 2N + 1}$



Putting this in Big-O terms

For the second handshake problem, the introductions was $\mathbf{N^2 - N}$:

$$O(\quad N^2 - 2N + 1)$$

But wait, didn't we just say that a term of +/- N *was* important?

For Big-O, we only care about the **largest term** of the polynomial

Big-O and Binary Search

SPOILER: FAST!!

Binary search

2	7	8	13	25	29	33	51	89	90	95

Jump right to the middle of the region to search, then repeat this process of roughly cutting the array in half again and again until we either find the item or (worst case) cut it down to nothing.

Worst case cost is number of times we can divide length in half:

$$O(\log_2 N)$$

Putting it all together

Binary search

Handshake #1

Handshake #2

MANY important optimization and other problems

$\log_2 n$	n	$n \log_2 n$	n^2	2^n
2	**4**	8	16	
3	**8**	24	64	256
4	**16**	64	256	65,536
5	**32**	160	1,024	4,294,967,296
6	**64**			
7	**128**			
8	**256**			
9	**512**			
10	**1,024**			
30	**2,700,000,000**			

Naïve Recursive Fibonacci ($O(1.6^n)$)

$\log_2 n$	**n**	**$n \log_2 n$**	**n^2**	**2^n**
2	**4**	8	16	16
3	**8**	24	64	256
4	**16**	64	256	65,536
5	**32**	160	1,024	4,294,967,296
6	**64**			
7	**128**			
8	**256**			
9	**512**			
10	**1,024**			
30	**2,700,000,000**			

2.4s

Easy!

Traveling Salesperson Problem:

We have a bunch of cities to visit. In what order should we visit them to minimize total travel distance?

Traveling Salesperson Problem:
We have a bunch of cities to visit. In what order should we visit them to minimize total travel distance?

Exhaustively try all orderings: **O(n!)**

Use current best known algorithm: **$O(n^2 2^n)$**

Maybe we could invent an algorithm that fits in our rightmost column: **$O(2^n)$**

So let’s say we come up with a way to solve Traveling Salesperson Problem in $O(2^n)$.

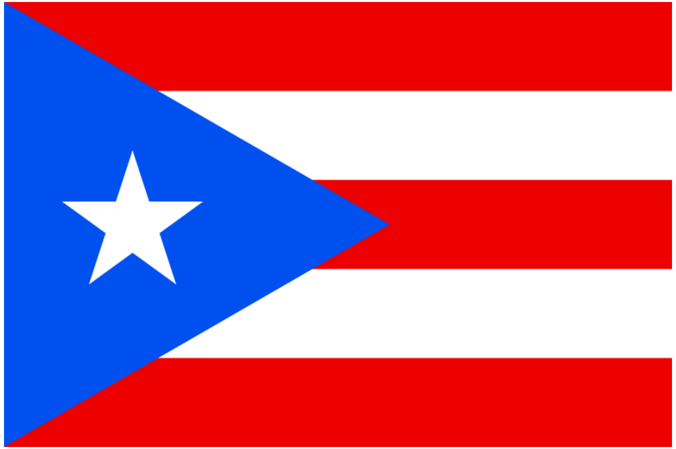
It would take **4 days** to solve Traveling Salesperson Problem on 50 state capitals.

Two *tiny* little updates

Imagine we approve statehood for US territory Puerto Rico

- Add San Juan, the capital city

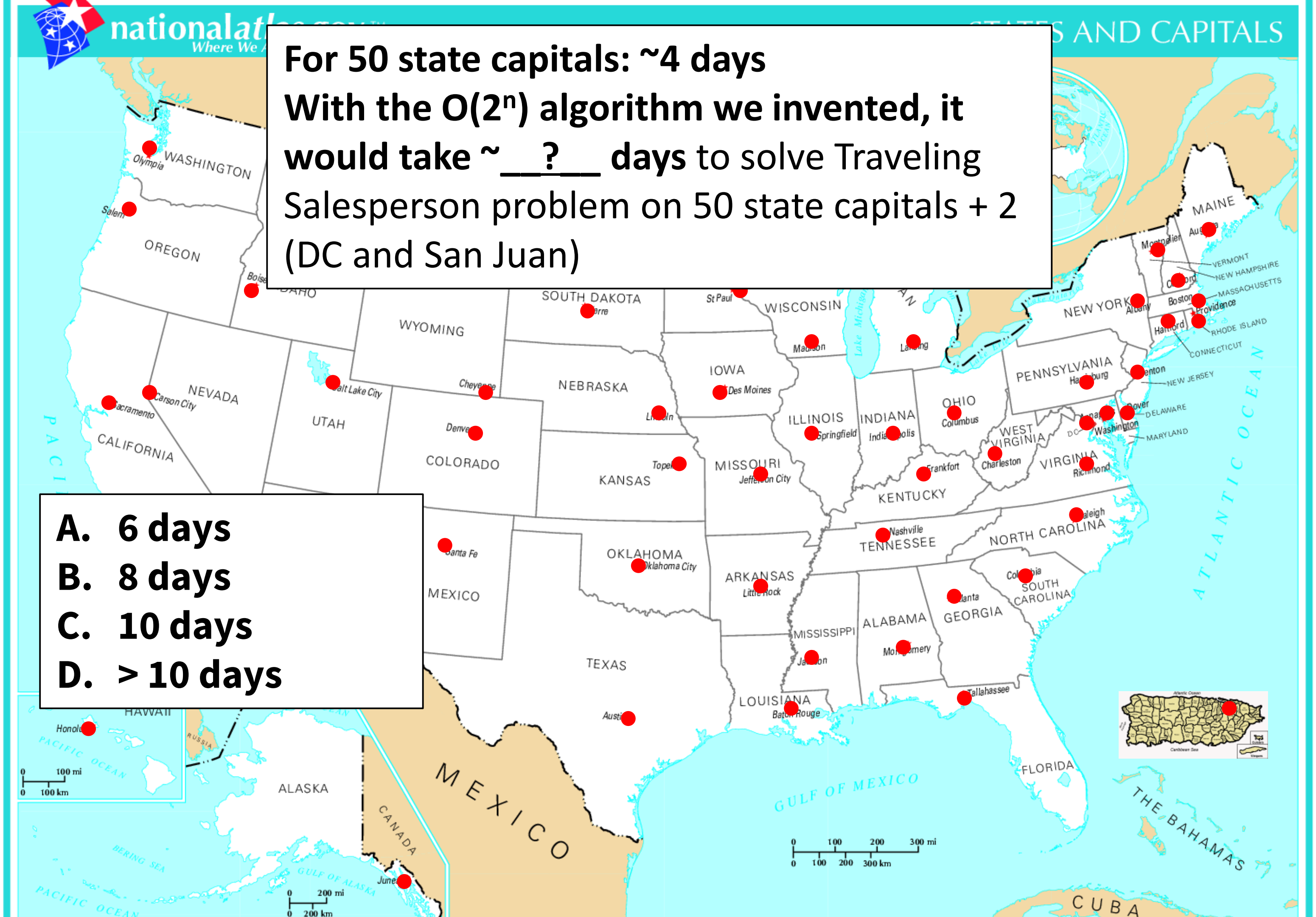
Also add Washington, DC



Now 52 capital cities instead of 50

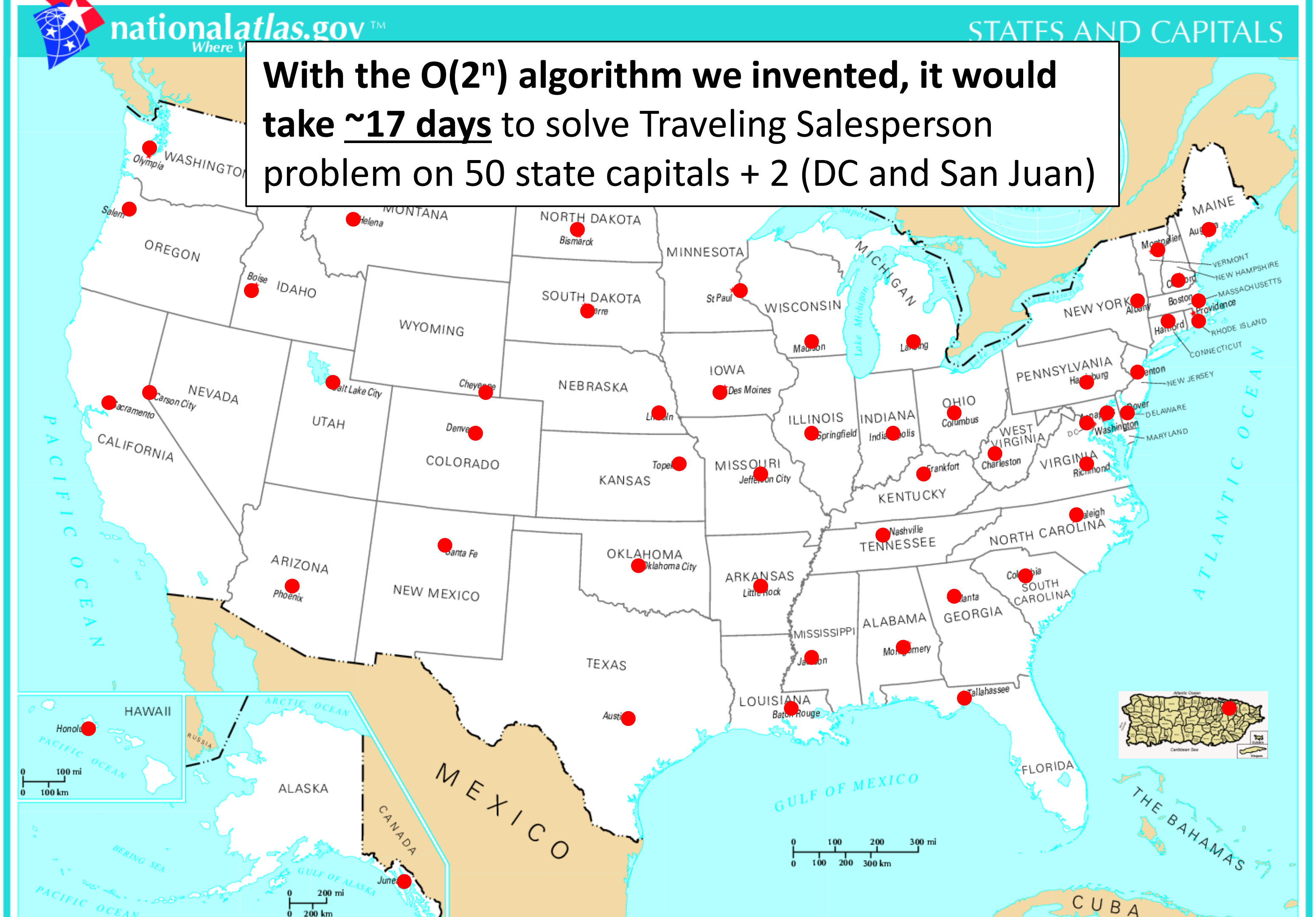
For 50 state capitals: ~4 days

With the $O(2^n)$ algorithm we invented, it would take ~__?__ days to solve Traveling Salesperson problem on 50 state capitals + 2 (DC and San Juan)



A. **6 days**
B. **8 days**
C. **10 days**
D. **> 10 days**

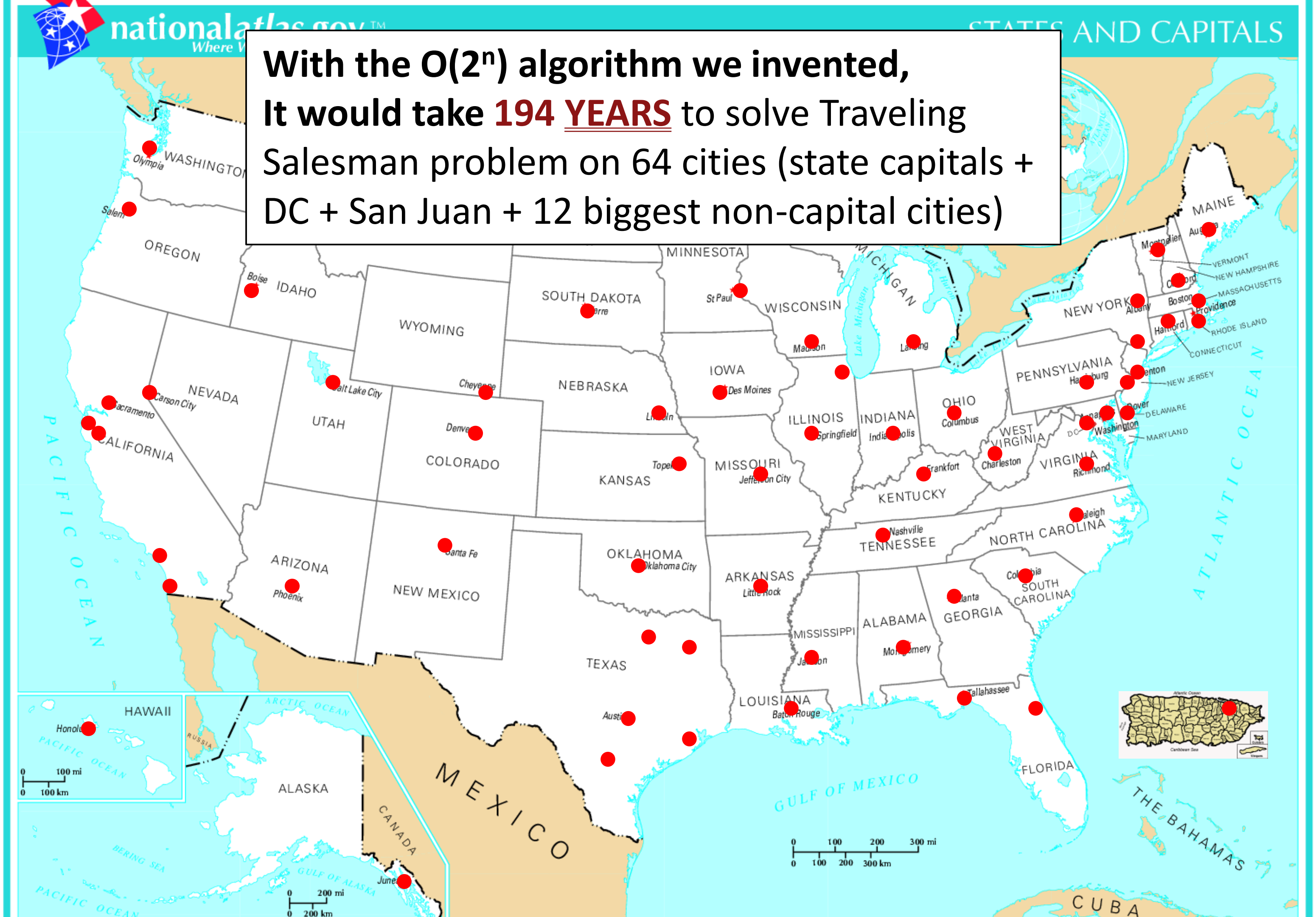
With the $O(2^n)$ algorithm we invented, it would take <u>~17 days</u> to solve Traveling Salesperson problem on 50 state capitals + 2 (DC and San Juan)



Sacramento is not exactly the most interesting or important city in California (sorry, Sacramento).

What if we add the 12 biggest non-capital cities in the United States to our map?

With the $O(2^n)$ algorithm we invented,
It would take **194 YEARS** to solve Traveling Salesman problem on 64 cities (state capitals + DC + San Juan + 12 biggest non-capital cities)



$\log_2 n$	n	$n \log_2 n$	n^2	2^n
2	**4**	8	16	16
3	**8**	24	64	256
4	**16**	64	256	65,536
5	**32**	160	1,024	4,294,967,296
6	**64**	384	4,096	1.84×10^{19}
7	**128**			
8	**256**			
9	**512**			
10	**1,024**			
30	**2,700,000,000**			

194 **YEARS**

$\log_2 n$	n	$n \log_2 n$	n^2	2^n
2	**4**	8	16	16
3	**8**	24	64	256
4	**16**	64	256	65,536
5	**32**	160	1,024	4,294,967,296
6	**64**	384	4,096	1.84×10^{19}
7	**128**	896	16,384	3.40×10^{38}
8	**256**			
9	**512**			
10	**1,024**			
30	**2,700,000,000**			

3.59E+21 **YEARS**

$\log_2 n$	n	$n \log_2 n$	n^2	2^n
2	**4**	8	16	16
3	**8**	24	64	256
4	**16**	64	256	65,536
5	**32**	160	1,024	4,294,967,296
6	**64**	384	4,096	1.84×10^{19}
7	**128**	896	16,384	3.40×10^{38}
8	**256**			
9	**512**			
10	**1,024**			
30	**2,700,000,000**			

3,590,000,000,000,000,000,000

YEARS

$\log_2 n$	n	$n \log_2 n$	n^2	2^n
2	**4**	8	16	16
3	**8**	24	64	256
4	**16**	64	256	65,536
5	**32**	160	1,024	4,294,967,296
6	**64**	384	4,096	1.84×10^{19}
7	**128**	896	16,384	3.40×10^{38}
8	**256**	2,048	65,536	1.16×10^{77}
9	**512**			
10	**1,024**			
30	**2,700,000,000**			

For comparison: there are about 10E+80 atoms in the universe. No big deal.

$\log_2 n$	n	$n \log_2 n$	n^2	2^n
2	**4**	8	16	16
3	**8**	24	64	256
4	**16**	64	256	65,536
5	**32**	160	1,024	4,294,967,296
6	**64**	384	4,096	1.84×10^{19}
7	**128**	896	16,384	3.40×10^{38}
8	**256**	2,048	65,536	1.16×10^{77}
9	**512**	4,608	262,144	1.34×10^{154}
10	**1,024**			
30	**2,700,000,000**			

1.42E+137 **YEARS**

$\log_2 n$	n	$n \log_2 n$	n^2	2^n
2	**4**	8	16	16
3	**8**	24	64	256
4	**16**	64	256	65,536
5	**32**	160	1,024	4,294,967,296
6	**64**	384	4,096	1.84×10^{19}
7	**128**	896	16,384	3.40×10^{38}
8	**256**	2,048	65,536	1.16×10^{77}
9	**512**	4,608	262,144	1.34×10^{154}
10	**1,024**	10,240 (.000003s)	1,048,576 (.0003s)	1.80×10^{308}
30	**2,700,000,000**	84,591,843,105 (28s)	7,290,000,000,000,000,000 (77 years)	LOL

$\log_2 n$	n	$n \log_2 n$	n^2	2^n
2	**4**	8	16	16
3	**8**	24	64	256
4	**16**	64	256	65,536
5	**32**	160	1,024	4,294,967,296
6	**64**	384	4,096	1.84×10^{19}
7	**128**	896	16,384	3.40×10^{38}
8	**256**	2,048	65,536	1.16×10^{77}
9	**512**	4,608	262,144	1.34×10^{154}
10	**1,024**	10,240 (.000003s)	1,048,576 (.0003s)	1.80×10^{308}
31	**2,700,000,000**	84,591,843,105 (28s)	7,290,000,000,000,000,000 (77 years)	$1.962227 \times 10^{812,780,998}$

$\log_2 n$	n	$n \log_2 n$	n^2	2^n
2	**4**	8	16	16
3	**8**	24	64	256
4	**16**	64	256	65,536
5	**32**	160	1,024	4,294,967,296
6	**64**	384	4,096	1.84×10^{19}
7	**128**	896	16,384	3.40×10^{38}
8	**256**	2,048	65,536	1.16×10^{77}
9	**512**	4,608	262,144	1.34×10^{154}
[illegible]	[illegible]	[illegible]40 ([illegible]s)	1,048,576 (.0003s)	1.80×10^{308}
30	**2,700,000,000**	84,591,843,105 (28s)	7,290,000,000,000,000,000 (77 years)	$1.962227 \times 10^{812,780,998}$

2^n is clearly infeasible, but **look at $\log_2 n$—only a tiny fraction of a second!**

In Conclusion

- **NOT worth doing:** Optimization of your code that **just trims** a bit
  - Like that +/-1 handshake—we don't need to worry ourselves about it!
  - Just write clean, easy-to-read code!!!!!
- **MAY be worth doing:** Optimization of your code that **changes Big-O**
  - **If** performance of a particular function is important, focus on this!
  - *(but if performance of the function is not very important, for example it will only run on small inputs, focus on just writing clean, easy-to-read code!!)*

- (Also remember that efficiency is not necessarily a virtue—first and foremost focus on correctness, both technical and ethical/moral/societal justice)