

# Programming Abstractions

## CS106B

Cynthia Bailey Lee  
Julie Zelenski

# Today's Topics:

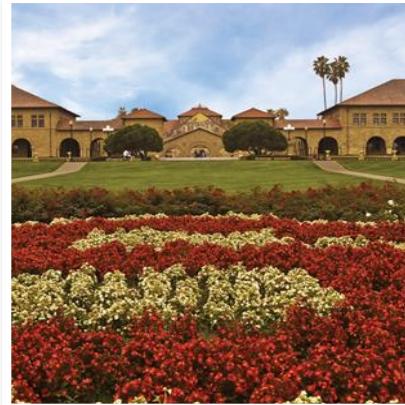
- Contrasting performance of 3 recursive algorithms
- Quantifying algorithm performance with Big-O analysis
- Getting a sense of scale in Big-O analysis

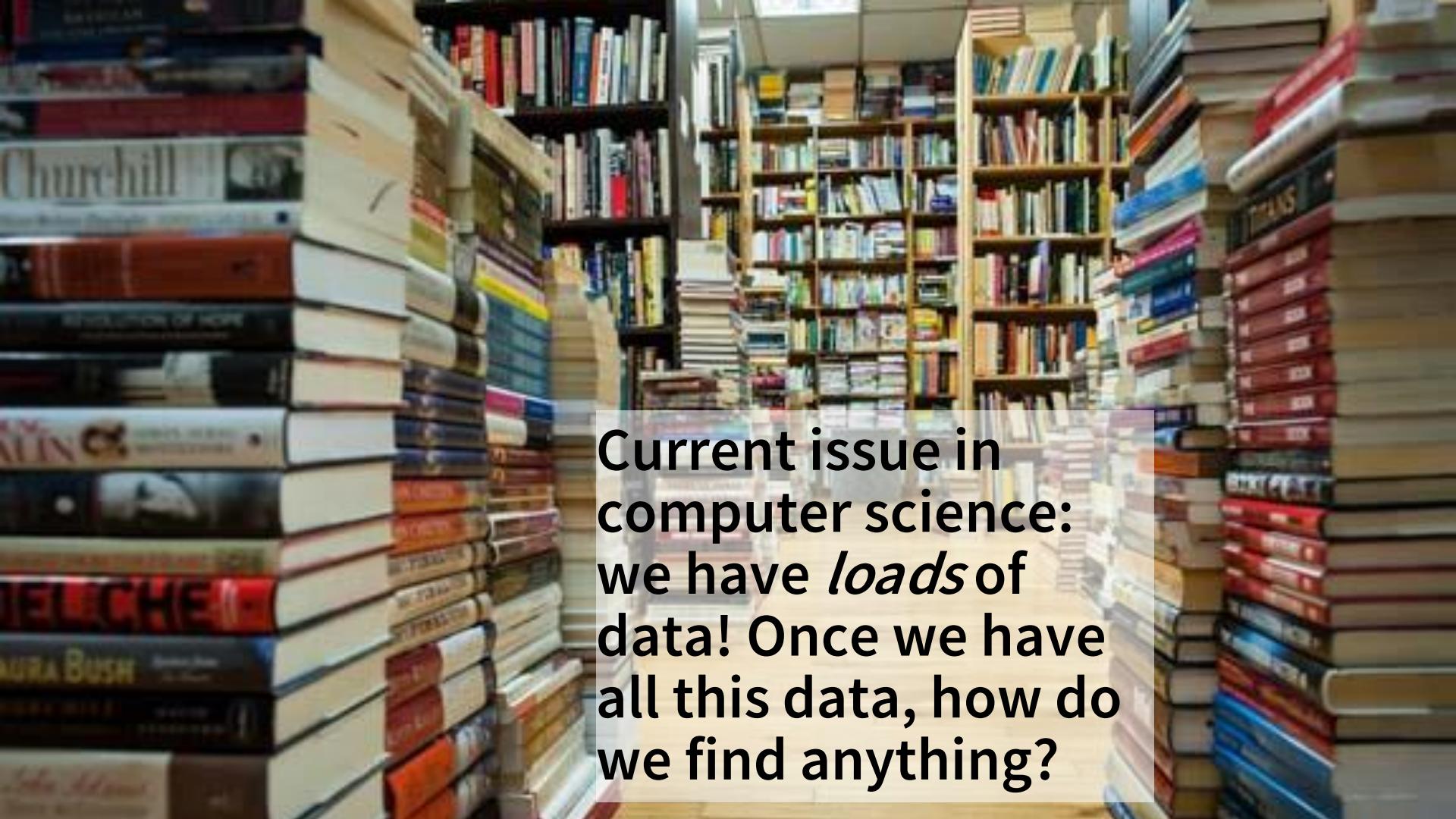
## Announcements:

- Please don't start assignments on the deadline day! Yikes!
  - At Stanford, classes are supposed to be 3hrs/unit. CS106B is 5 units, so that's 15 hours of work per week. Of that, you should budget about 8-10 hours working on assignments.
- Remember that assignments are DUE FRIDAY.
  - The Sunday grace period is for emergencies such as injuries, illness, laptop died, etc. *only*.
  - If you plan on routine use of the grace period, you rob yourself of that safety buffer.
  - **Extension requests emailed to Neel are for issues whose scale and duration exceed the ability of the grace period to address**, not because something unexpectedly interfered with a *choice* to complete the assignment during the grace period.

# 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?

# The context:

- You have a **collection of numbers**
  - › Say product IDs for items in stock in a store
  - › We're going to store our collection of numbers in a Vector
  - › We're going to keep them *in sorted order*

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

- It's important to be able to **find out whether you have a particular number** in your collection or not
  - › A customer asks, "Do you have item 8 in stock?" (Yes.)
  - › A customer asks, "Do you have item 55 in stock?" (No.)
- **Key question: How long does this take?**

# Does this list of numbers contain X?

Context: we have a collection of numbers in a Vector, in sorted order.

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

- **Basic approach: Start at the front and proceed forward until you find:**
  - › X (answer Yes)
  - › A number greater than X (answer No)
  - › End of the list (answer No)
- Key observation: each time you compare against the contents of a cell of the Vector and it's not X, you rule out \*1\* of the N cells in the Vector

# Does this list of numbers contain X?

Context: we have a collection of numbers in a Vector, in sorted order.

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

- **Efficiency Hack: Jump to the middle of the Vector and look there to find:**

- › X (answer Yes)
- › A number greater than X (rule out entire second half of Vector)

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

- › A number less than X (rule out entire first half of Vector)

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

- Key observation: with *\*one\** comparison, you ruled out *\*N/2\** of the N cells in the Vector!

# Does this list of numbers contain X?

Context: we have a collection of numbers in a Vector, in sorted order.

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

- **Efficiency Hack: Jump to the middle of the Vector and look there to find:**

- › X (answer Yes)
- › A number greater than X (rule out entire second half of Vector)

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

- › A number less than X (rule out entire first half of Vector)

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

Now we could do our Basic Approach, but in **half the time**.

Thanks, Efficiency Hack!!

- Key observation: with *\*one\** comparison, you ruled out *\*N/2\** of the Numbers in the Vector!

...but I have an even better idea...

# Does this list of numbers contain X?

Context: we have a collection of numbers in a Vector, in sorted order.

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

- **Extreme Efficiency Hack: Keep jumping to the middle!**

- › Let's say our first jump to the middle found a number less than X, so we ruled out the whole first half:

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

- › Now jump to the middle of the remaining second half:

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

- Key observation: we do one piece of work, then delegate the rest. **Recursion!!**

# 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);
    }
}
```

# 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);
    }
}
```

**Base case:** we shrank the search problem so tiny it no longer exists!

**Recursive case:**

Do one piece of work (comparison)

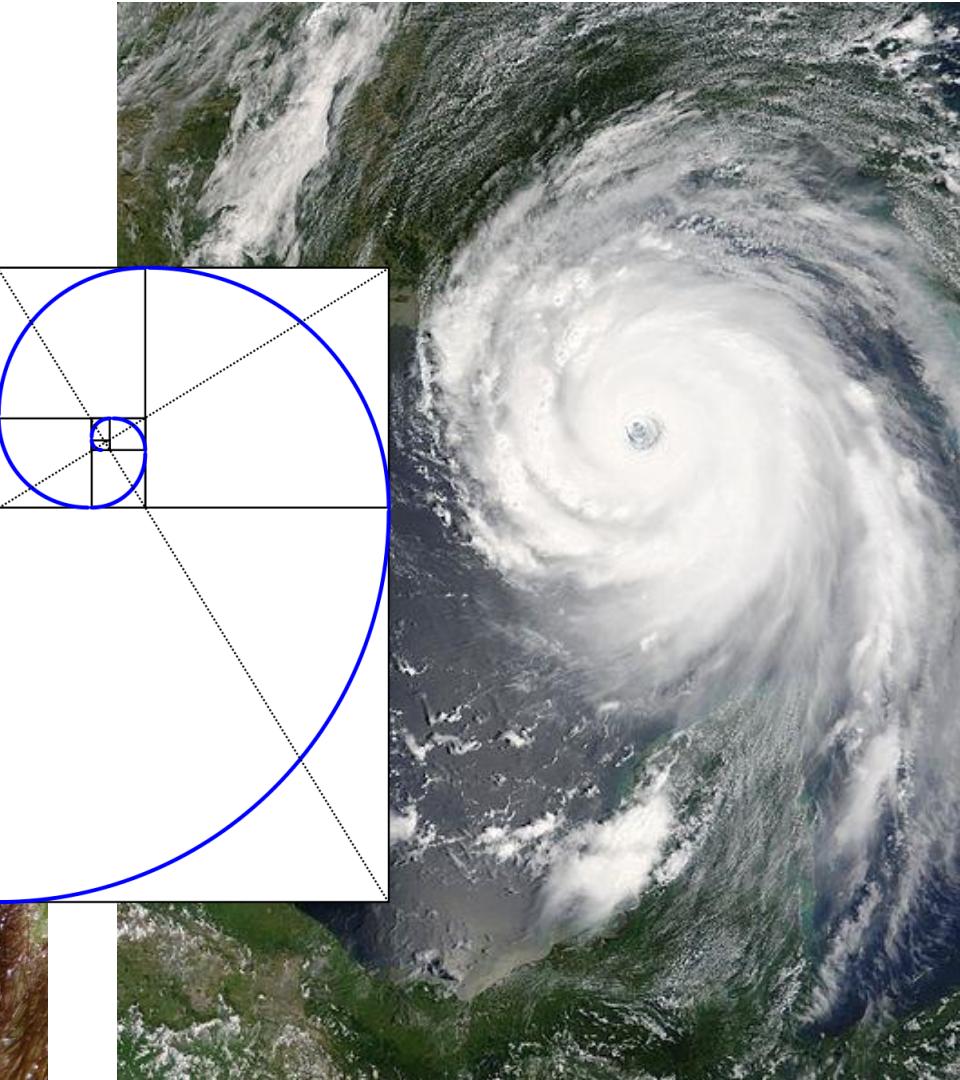
Delegate the rest of the work

# The Fibonacci Sequence

\*MATH NERD REJOICING  
INTENSIFIES\*



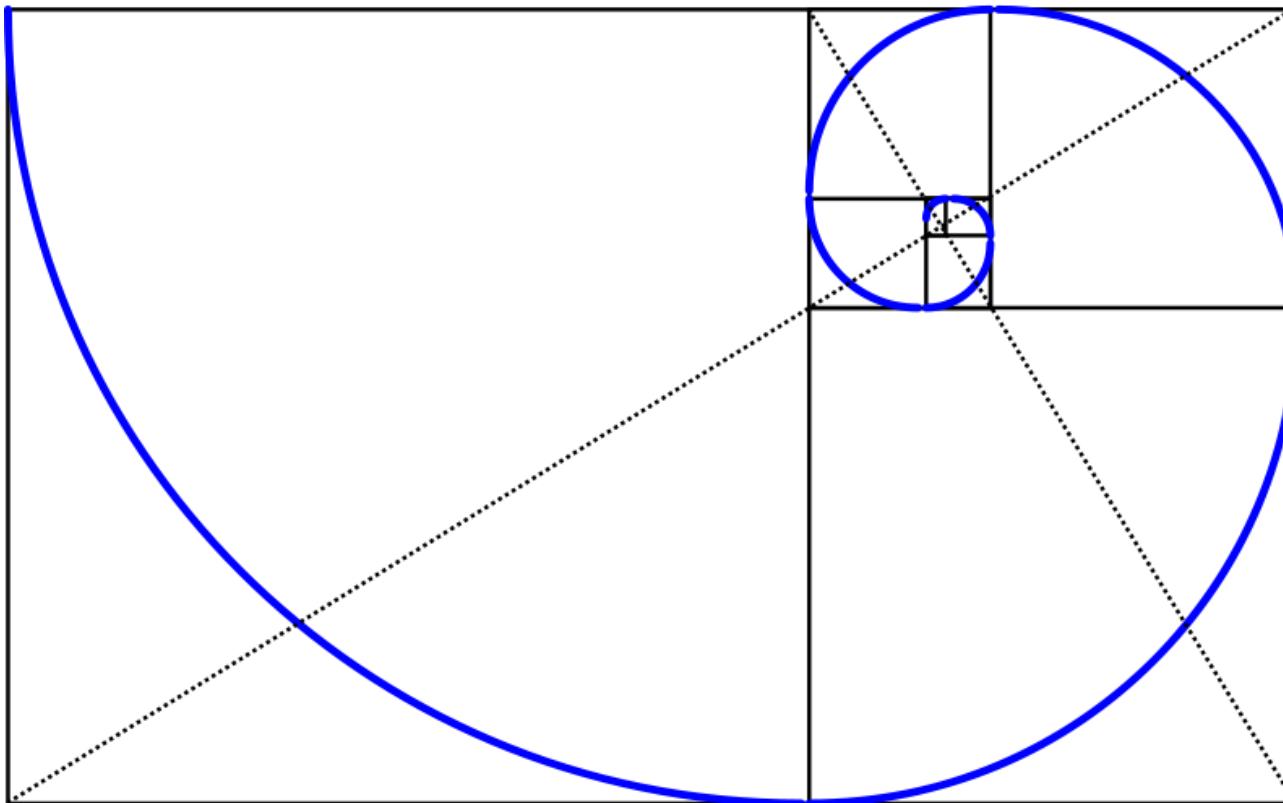
# Fibonacci in nature



These files are, respectively: public domain (hurricane) and licensed under the [Creative Commons Attribution 2.0 Generic license](#) (fibonacci and fern).

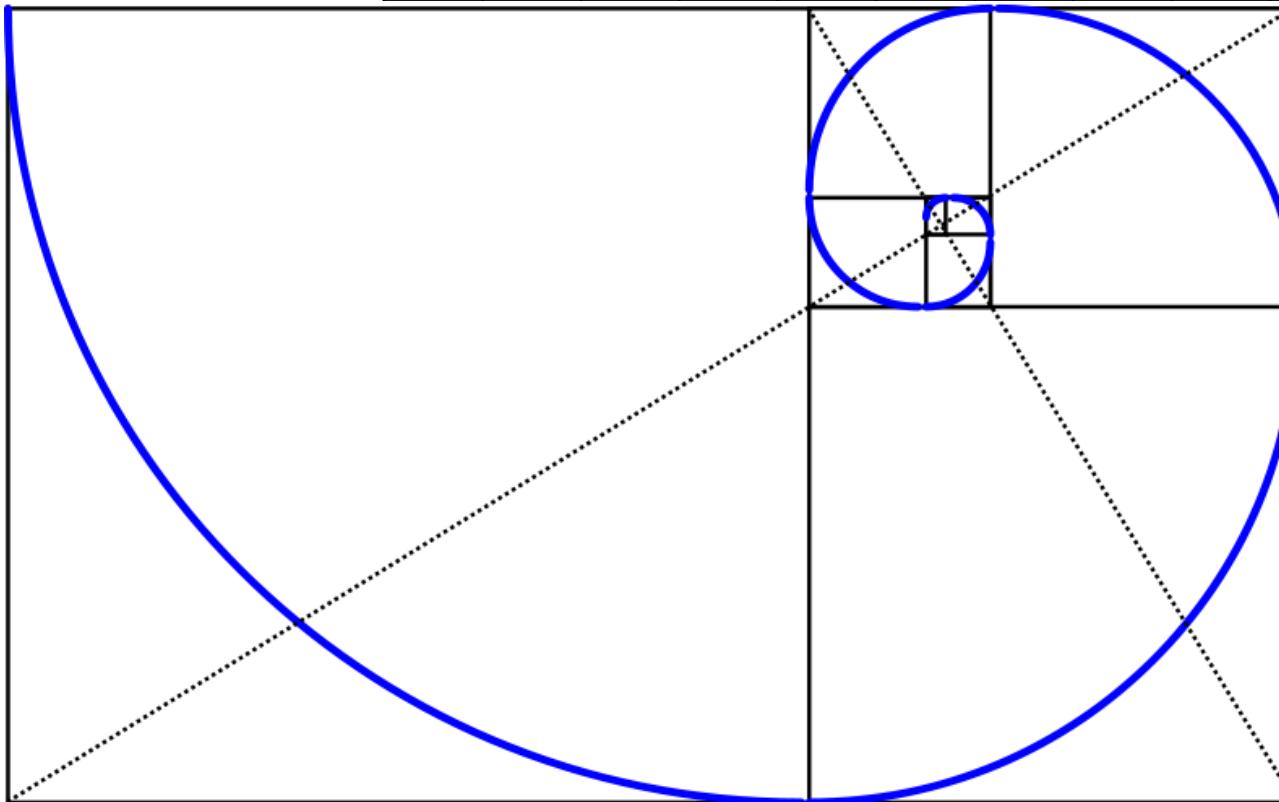
# Fibonacci

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



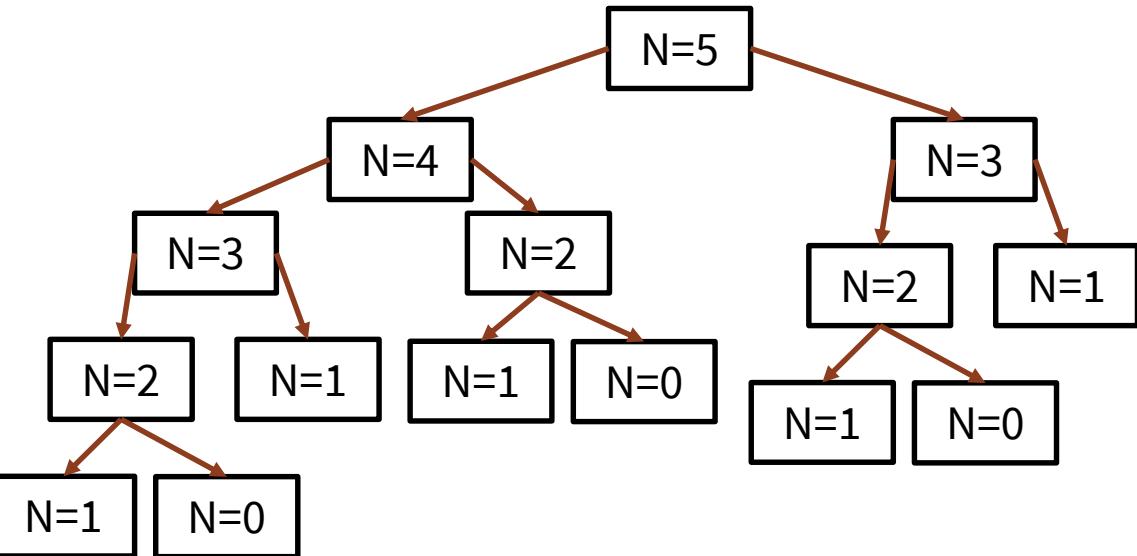
# Fibonacci

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);
    }
}
```

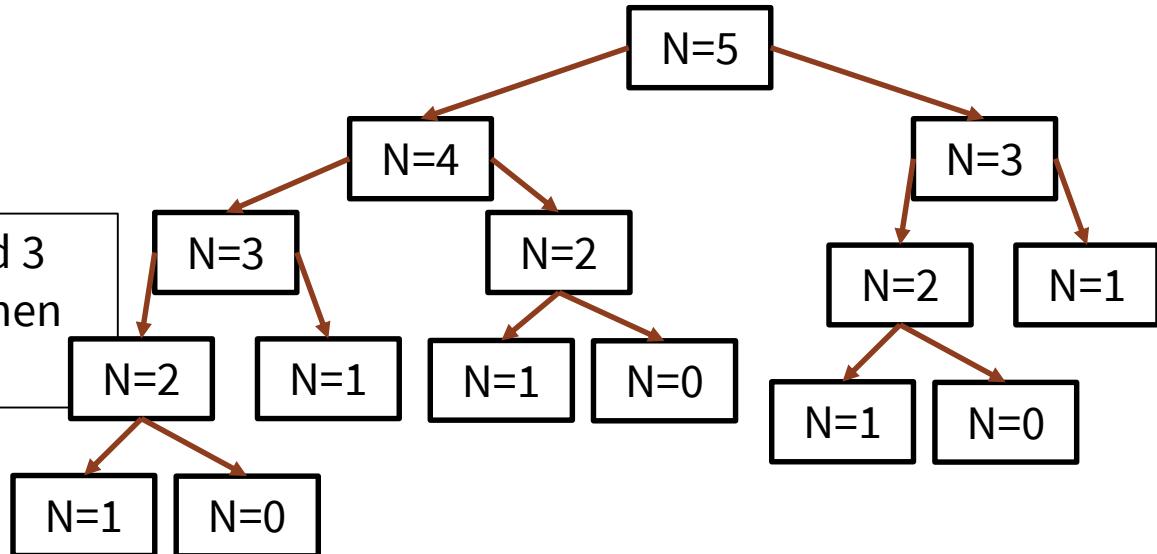


Work is duplicated throughout the call tree

- 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)!



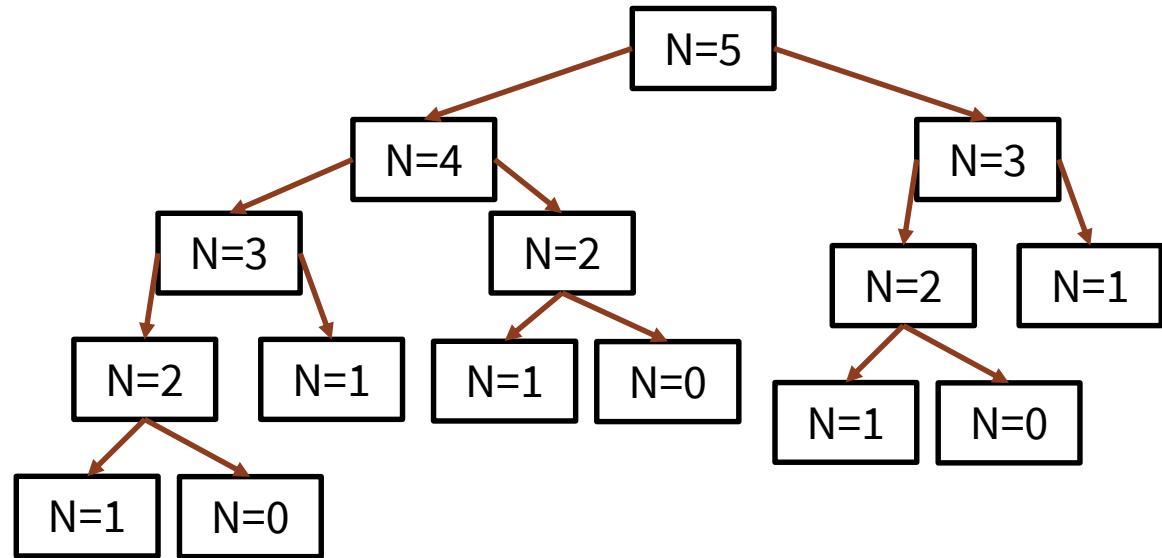
How many times would we calculate fib(2) while calculating fib(6)?

***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	5
7	13	8
8	21	13
9	34	21
10	55	34



# Efficiency of naïve Fibonacci implementation

When we **added 1** to the input  $N$ , the number of times we had to calculate  $\text{fib}(2)$  **nearly doubled** ( $\sim 1.6^*$  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  $N$ .**

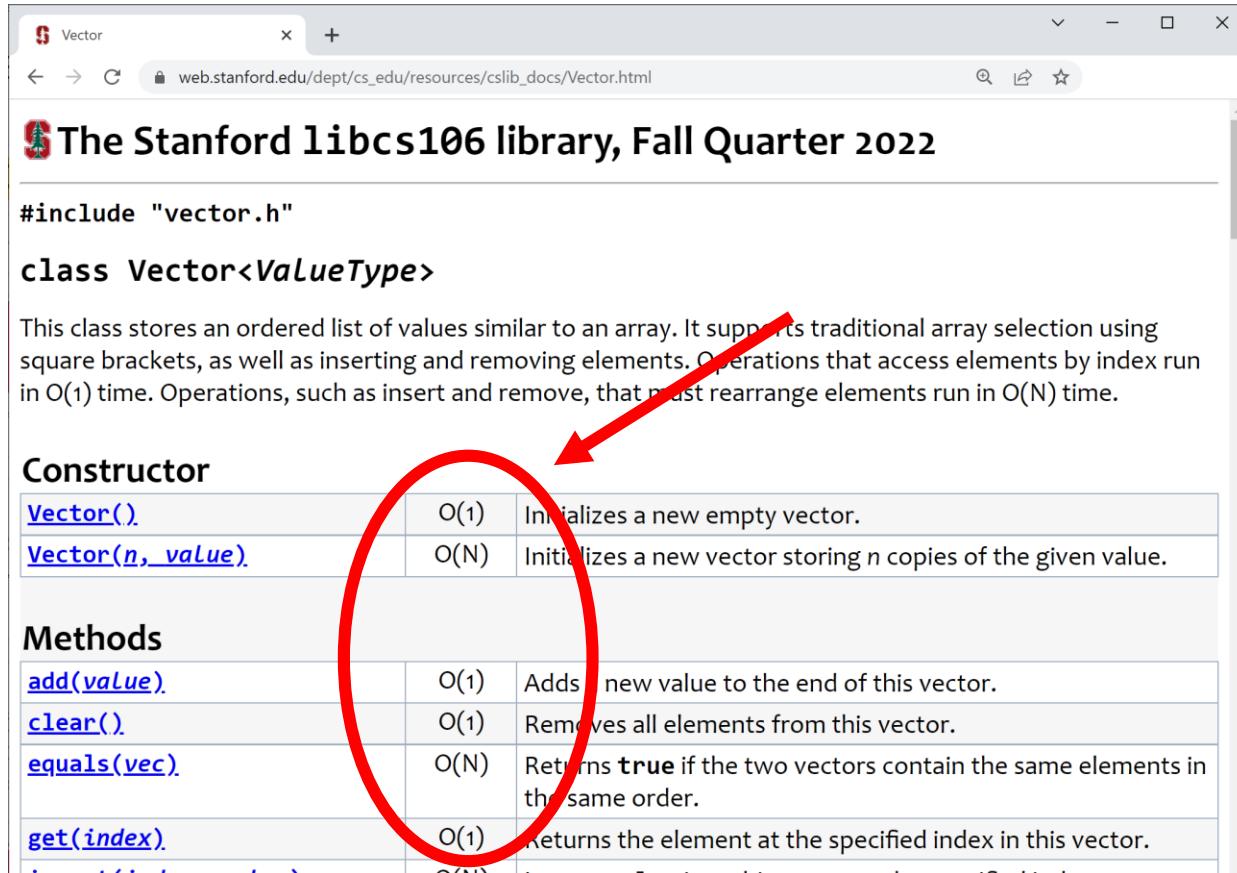
Calculation: “approximately”  $(1.6)^N$

# Big-O Performance Analysis

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



# Big-O analysis in computer science



**The Stanford libcs106 library, Fall Quarter 2022**

```
#include "vector.h"

class Vector<ValueType>
```

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

### Constructor

<a href="#"><u>Vector()</u></a>	$O(1)$	Initializes a new empty vector.
<a href="#"><u>Vector(n, value)</u></a>	$O(N)$	Initializes a new vector storing $n$ copies of the given value.

### Methods

<a href="#"><u>add(value)</u></a>	$O(1)$	Adds new value to the end of this vector.
<a href="#"><u>clear()</u></a>	$O(1)$	Removes all elements from this vector.
<a href="#"><u>equals(vec)</u></a>	$O(N)$	Returns <b>true</b> if the two vectors contain the same elements in the same order.
<a href="#"><u>get(index)</u></a>	$O(1)$	Returns the element at the specified index in this vector.
<a href="#"><u>insert(index, value)</u></a>	$O(N)$	Inserts a new value at the specified index in this vector.
<a href="#"><u>remove(index)</u></a>	$O(N)$	Removes the element at the specified index in this vector.

# Big-O analysis in computer science

Binary search algorithm - Wikipedia

en.wikipedia.org/wiki/Binary\_search\_algorithm

Not logged in Talk Contributions Create account Log in

Article Talk Read Edit View history Search Wikipedia

## Binary search algorithm

From Wikipedia, the free encyclopedia

*This article is about searching a finite sorted array. For searching continuous function values, see [bisection method](#).*

In computer science, **binary search**, also known as **half-interval search**,<sup>[1]</sup> **logarithmic search**,<sup>[2]</sup> or **binary chop**,<sup>[3]</sup> is a **search algorithm** that finds the position of a target value within a **sorted array**.<sup>[4][5]</sup> Binary search compares the target value to the middle element of the array. If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this until the target value is found. If the search ends with the remaining half being empty, the target is not in the array.

**Binary search algorithm**

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

1	3	4	6	7	8	10	13	14	18	19	21	24	37	40	45	75
---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----

Class

Search algorithm

Data structure

Array

Worst-case performance

$O(\log n)$

Best-case performance

$O(1)$

Average performance

$O(\log n)$



A red arrow points from the text "Binary search compares the target value to the middle element of the array." to the "Data structure" section of the table. A red circle highlights the "Worst-case performance" section of the table.

## Formal definition of big-O

We say a function  $f(n)$  is “big-O” of another function  $g(n)$   
(written  $f(n) = O(g(n))$ )  
if and only if

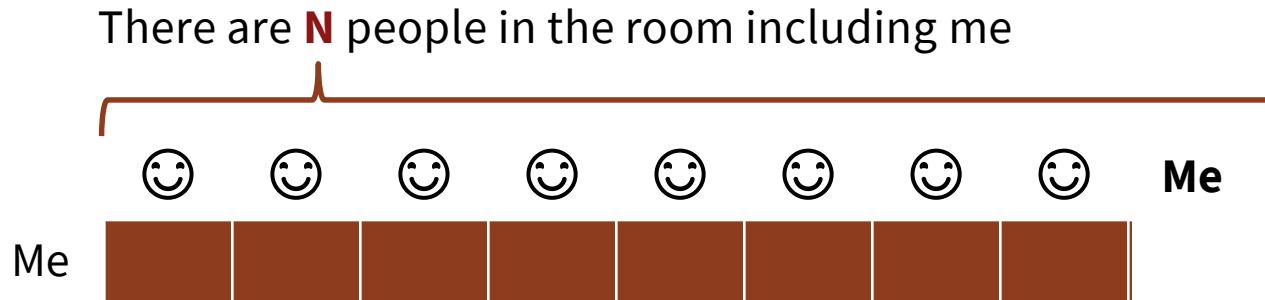
there exist positive constants  $c$  and  $n_0$  such that  
$$f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0.$$

# 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?



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  $\pm 1$



## 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:

$$O(N - 1)$$

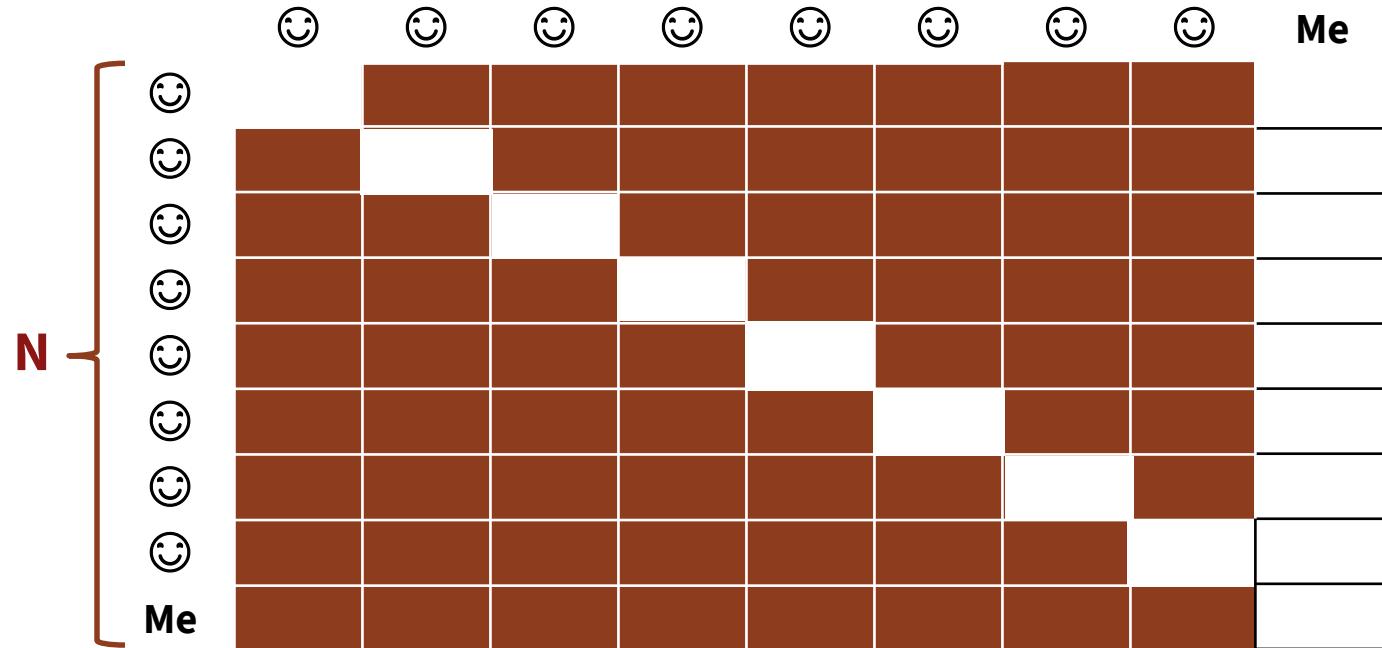
Similarly, if we said that each introduction **takes 3 seconds**, the amount of time is  **$3(N - 1) = 3N - 3$** , but we disregard the constant 3s:

$$O(3N - 3)$$

# Before we start, let's get introduced

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

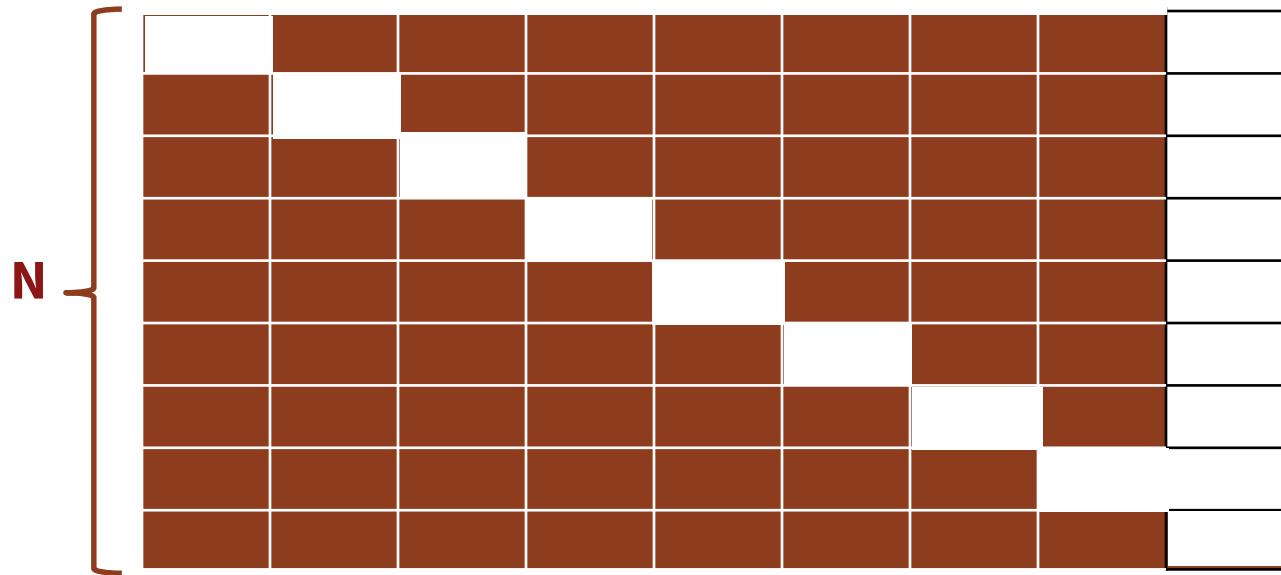
Now how many introductions?  $N^2$



# 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?  $N^2 - 2N + 1$



## Putting this in Big-O terms

For the second handshake problem, the introductions was  $N^2 - N$ :

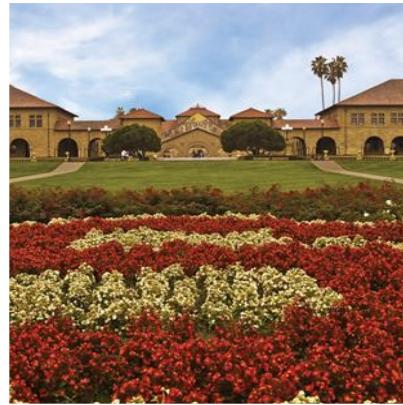
$$O(N^2 - 2N + 1)$$

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



**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			2.4s
7	128			Easy!
8	256			
9	512			
10	1,024			
30	2,700,000,000			

## 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



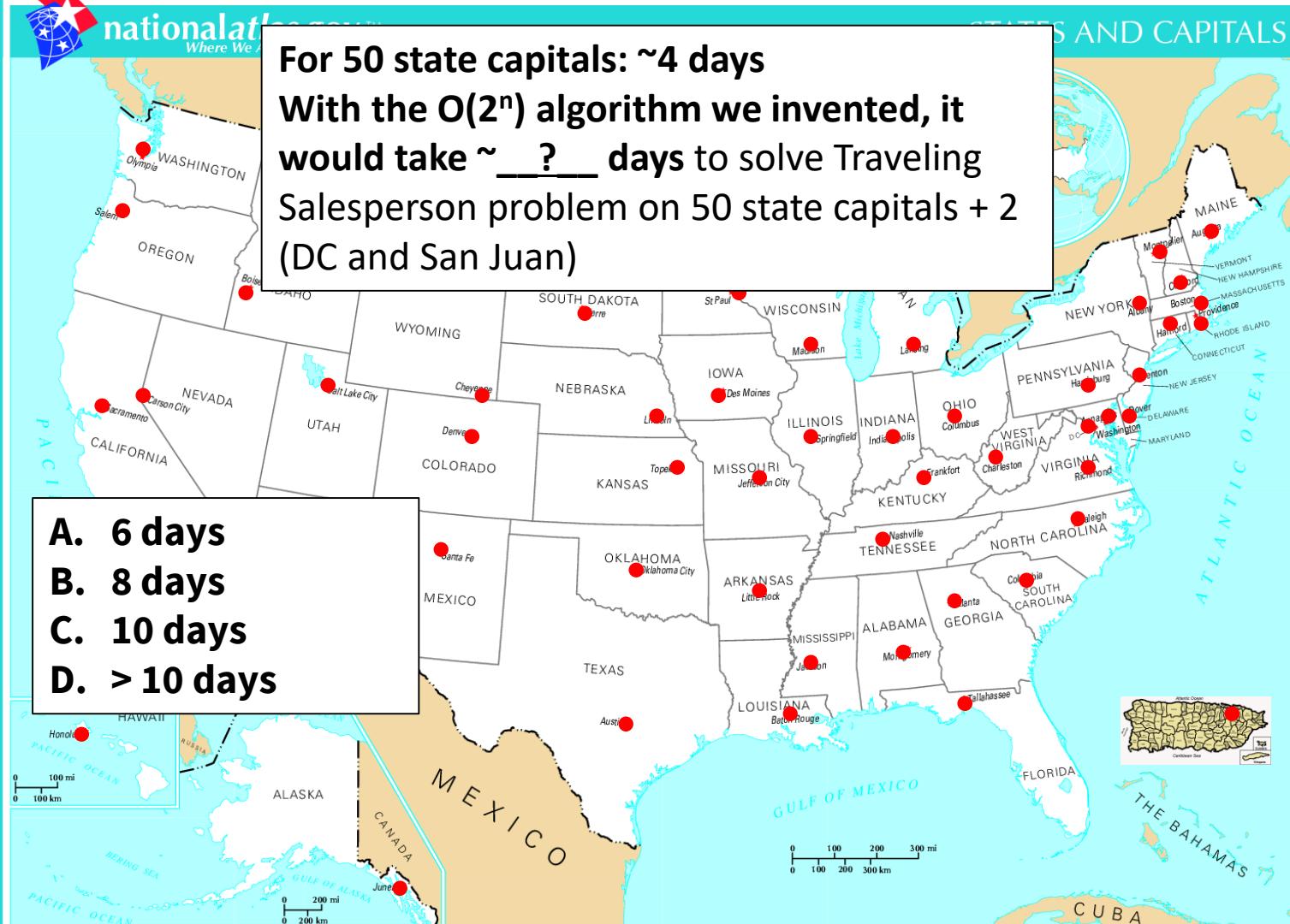
This work has been released into the [public domain](#) by its author, [Madden](#).  
This applies worldwide.

## **Now 52 capital cities instead of 50**

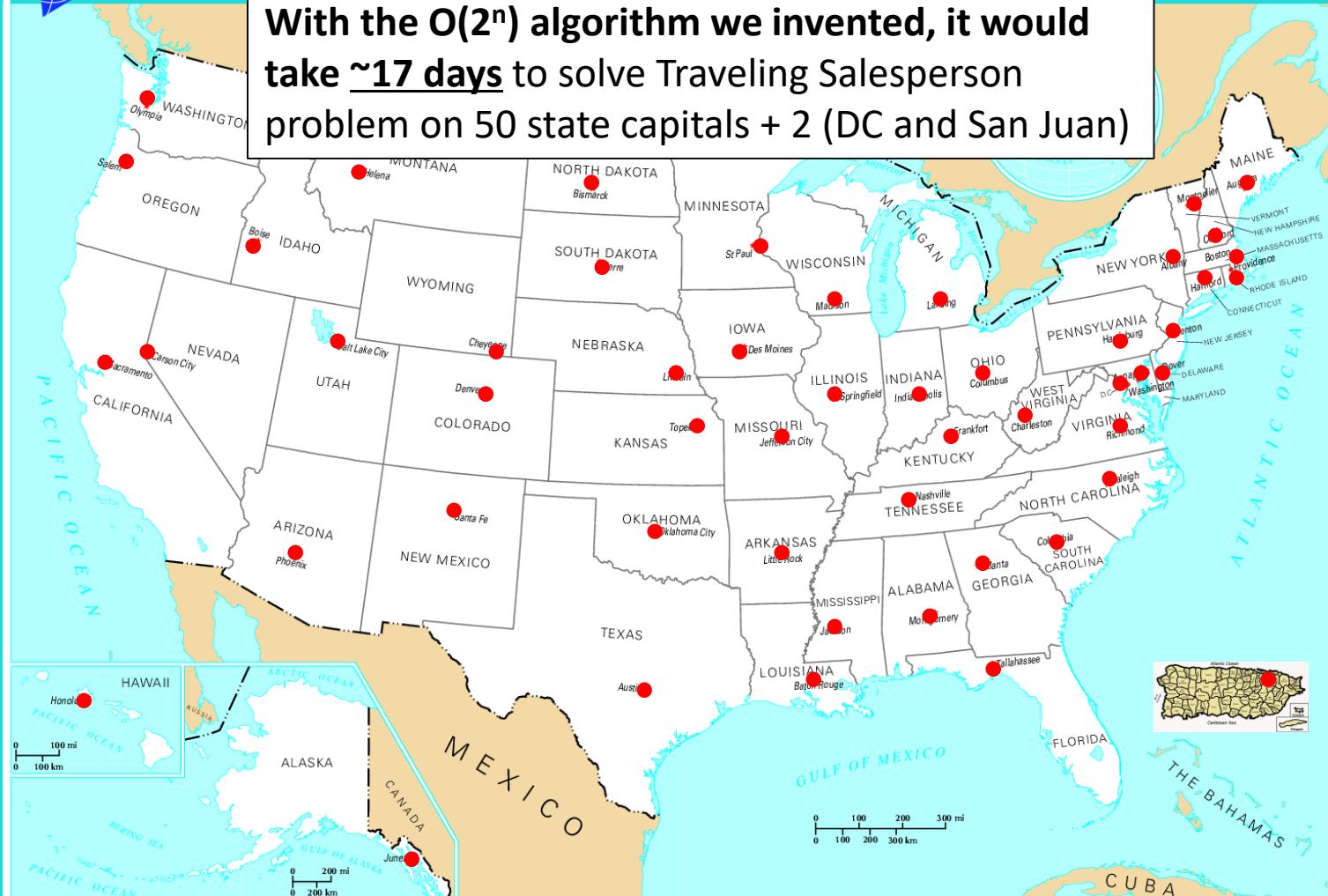


For 50 state capitals: ~4 days  
With the  $O(2^n)$  algorithm we invented, it  
would take ~   ? days to solve Traveling  
Salesperson problem on 50 state capitals + 2  
(DC and San Juan)

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



**With the  $O(2^n)$  algorithm we invented, it would take ~17 days 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	$1.84 \times 10^{19}$
7	128			<b>194 YEARS</b>
8	256			
9	512			
10	1,024			
30	<b>2,700,000,000</b>			

$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$2^n$
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	$1.84 \times 10^{19}$
7	128	896	16,384	$3.40 \times 10^{38}$
8	256			<b>3.59E+21 YEARS</b>
9	512			
10	1,024			
30	<b>2,700,000,000</b>			

$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$2^n$
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	$1.84 \times 10^{19}$
7	128	896	16,384	$3.40 \times 10^{38}$
8	256	<b>3,590,000,000,000,000,000,000 YEARS</b>		
9	512			
10	1,024			
30	<b>2,700,000,000</b>			

$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$2^n$
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	$1.84 \times 10^{19}$
7	128	896	16,384	$3.40 \times 10^{38}$
8	256	2,048	65,536	$1.16 \times 10^{77}$
9	512			
10	1,024			
30	2,700,000,000			

For comparison: there are about  $10^{80}$  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	$1.84 \times 10^{19}$
7	128	896	16,384	$3.40 \times 10^{38}$
8	256	2,048	65,536	$1.16 \times 10^{77}$
9	512	4,608	262,144	$1.34 \times 10^{154}$
10	1,024			<span style="border: 2px solid red; padding: 2px;"><b>1.42E+137 YEARS</b></span>
30	<b>2,700,000,000</b>			

$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$2^n$
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	$1.84 \times 10^{19}$
7	128	896	16,384	$3.40 \times 10^{38}$
8	256	2,048	65,536	$1.16 \times 10^{77}$
9	512	4,608	262,144	$1.34 \times 10^{154}$
10	1,024	10,240 (.000003s)	1,048,576 (.0003s)	$1.80 \times 10^{308}$
30	2,700,000,000	84,591,843,105 (28s)	7,290,000,000,000,000 (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	$1.84 \times 10^{19}$
7	128	896	16,384	$3.40 \times 10^{38}$
8	256	2,048	65,536	$1.16 \times 10^{77}$
9	512	4,608	262,144	$1.34 \times 10^{154}$
10	1,024	10,240 (.000003s)	1,048,576 (.0003s)	$1.80 \times 10^{308}$
31	<b>2,700,000,000</b>	84,591,843,105 (28s)	7,290,000,000,000,000 (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	$1.84 \times 10^{19}$
7	128	896	16,384	$3.40 \times 10^{38}$
8	256	2,048	65,536	$1.16 \times 10^{77}$
9	512	4,608	262,144	$1.34 \times 10^{154}$
10	1,024	9,216	1,048,576	$1.80 \times 10^{308}$
11	2,048	18,432	(.0003s)	
12	4,096	36,864		
13	8,192	73,728		
14	16,384	147,456		
15	32,768	294,912		
16	65,536	589,824		
17	131,072	1,179,648		
18	262,144	2,359,296		
19	524,288	4,718,592		
20	1,048,576	9,437,184		
21	2,097,152	18,874,368		
22	4,194,304	37,748,736		
23	8,388,608	75,497,472		
24	16,777,216	150,994,944		
25	33,554,432	301,989,888		
26	67,108,864	603,979,776		
27	134,217,728	1,207,959,552		
28	268,435,456	2,415,918,104		
29	536,870,912	4,831,836,208		
30	1,073,741,824	9,663,672,416		
31	2,147,483,648	19,327,344,832		
32	4,294,967,296	38,654,688,664		
33	8,589,934,592	77,309,377,328		
34	17,179,869,184	154,618,754,656		
35	34,359,738,368	309,237,509,312		
36	68,719,476,736	618,475,018,624		
37	137,438,953,472	1,236,950,037,248		
38	274,877,906,944	2,473,900,074,496		
39	549,755,813,888	4,947,800,148,992		
40	1,099,511,627,776	9,895,600,297,984		
41	2,199,023,255,552	19,791,200,595,968		
42	4,398,046,511,104	39,582,400,191,936		
43	8,796,093,022,208	79,164,800,383,872		
44	17,592,186,044,416	158,329,600,767,744		
45	35,184,372,088,832	316,658,201,535,488		
46	70,368,744,177,664	633,316,403,070,976		
47	140,737,488,355,328	1,266,632,806,141,952		
48	281,474,976,710,656	2,533,265,612,283,904		
49	562,949,953,421,312	5,066,531,224,567,808		
50	1,125,899,906,842,624	10,133,062,449,135,616		
51	2,251,799,813,685,248	20,266,124,898,271,232		
52	4,503,599,627,370,496	40,532,249,796,542,464		
53	9,007,199,254,740,992	81,064,499,593,084,928		
54	18,014,398,509,481,984	162,128,999,186,169,856		
55	36,028,797,018,963,968	324,257,998,372,339,712		
56	72,057,594,037,927,936	648,515,996,744,679,424		
57	144,115,188,075,855,872	1,296,031,993,489,358,848		
58	288,230,376,151,711,744	2,592,063,986,978,717,696		
59	576,460,752,303,423,488	5,184,127,973,957,435,392		
60	1,152,921,504,606,846,976	10,368,255,947,914,870,784		
61	2,305,843,009,213,693,952	20,736,511,895,829,741,568		
62	4,611,686,018,427,387,904	41,473,023,791,659,483,136		
63	9,223,372,036,854,775,808	82,946,047,583,318,966,272		
64	18,446,744,073,709,551,616	165,892,095,166,637,932,544		
65	36,893,488,147,419,103,232	331,784,190,333,275,865,088		
66	73,786,976,294,838,206,464	663,568,380,666,551,730,176		
67	147,573,952,589,676,412,928	1,327,136,761,333,103,460,352		
68	295,147,905,179,352,825,856	2,654,273,522,666,206,920,704		
69	590,295,810,358,705,651,712	5,308,547,045,332,413,841,408		
70	1,180,591,620,717,411,303,424	10,617,094,090,664,827,682,816		
71	2,361,183,241,434,822,606,848	21,234,188,181,329,655,365,632		
72	4,722,366,482,869,644,813,696	42,468,376,362,659,310,731,264		
73	9,444,732,965,739,289,627,392	84,936,752,725,318,621,462,528		
74	18,889,465,931,478,579,254,784	169,873,505,450,637,242,925,056		
75	37,778,931,862,957,158,509,568	339,747,010,901,274,485,850,112		
76	75,557,863,725,914,317,018,136	679,494,021,802,548,971,700,224		
77	151,115,727,451,828,634,036,272	1,358,988,043,605,097,943,400,448		
78	302,231,454,903,657,268,072,544	2,717,976,087,210,195,886,800,896		
79	604,462,909,807,314,536,145,088	5,435,952,174,420,391,773,600,192		
80	1,208,925,819,614,629,072,290,176	10,871,904,348,840,783,547,200,384		
81	2,417,851,639,229,258,144,580,352	21,743,808,697,681,567,094,400,768		
82	4,835,703,278,458,516,288,160,704	43,487,617,395,363,134,188,800,136		
83	9,671,406,556,917,032,576,320,408	86,975,234,790,726,268,377,600,272		
84	19,342,813,113,834,065,152,640,816	173,950,469,581,452,535,755,200,544		
85	38,685,626,227,668,130,305,281,632	347,900,939,162,905,071,510,400,184		
86	77,371,252,455,336,260,610,563,264	695,801,878,325,810,142,020,800,368		
87	154,742,504,910,672,521,221,126,528	1,391,603,756,651,620,284,041,600,736		
88	309,485,009,821,345,042,442,253,056	2,783,207,513,303,240,568,083,200,152		
89	618,970,019,642,687,084,884,506,112	5,566,414,026,606,480,136,166,400,304		
90	1,237,940,039,285,374,168,769,012,224	11,132,828,053,212,960,272,332,800,608		
91	2,475,880,078,570,748,337,538,024,448	22,265,656,106,425,920,544,665,600,128		
92	4,951,760,157,141,496,675,076,048,896	44,531,312,212,851,840,189,331,200,256		
93	9,903,520,314,282,993,350,152,096,192	89,062,624,425,703,680,378,662,400,512		
94	19,807,040,628,565,986,700,304,192,384	178,125,248,851,407,360,757,324,800,024		
95	39,614,081,257,131,973,400,608,384,768	356,250,497,672,814,720,154,649,600,048		
96	79,228,162,514,263,946,801,216,768,536	712,500,995,345,628,440,309,299,200,096		
97	158,456,325,028,527,893,602,433,536,072	1,425,001,990,691,256,880,618,598,400,192		
98	316,912,650,057,055,787,204,867,072,144	2,850,003,981,382,513,760,137,196,800,384		
99	633,825,300,114,111,574,409,734,144,288	5,700,007,962,765,027,520,274,393,600,768		
100	1,267,650,600,228,222,148,819,468,288,576	11,400,015,925,530,055,040,548,787,200,136		

30

2,700,000,000

84,591,843,105  
(28s)7,290,000,000,000,000,  
000 (77 years)1.962227 x  
10<sup>812,780,998</sup>

**2<sup>n</sup> 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)