Programming Abstractions

CS106B

Cynthia Lee

Topics roadmap:

Previous classes:

- Recursion intro: factorial and stack frames (Friday)
- Designing recursive solutions: binary search and fractals (Monday)
- Loops + recursion: permutations and backtracking (Wednesday)

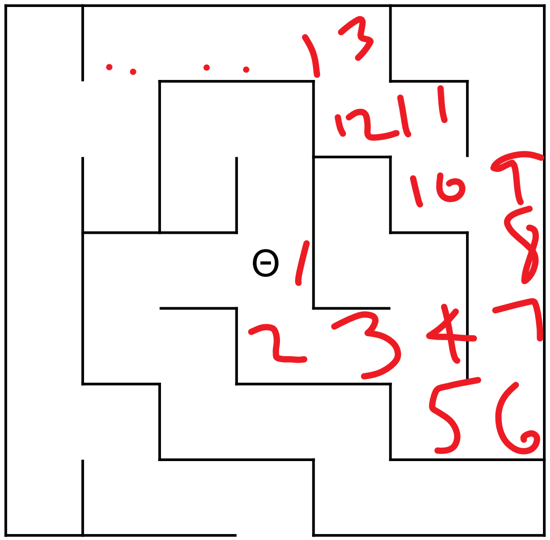
Today:

- Contrast: Word ladder and Maze solving
  - Revisiting Wednesday's maze solving example
- Performance issues in recursion
- Big-O performance analysis

Monday:

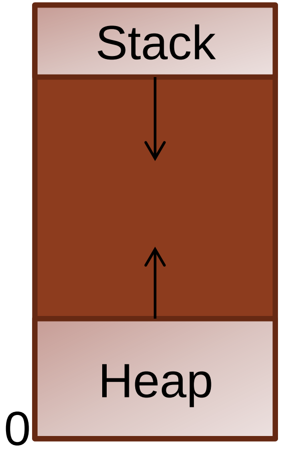
- More big-O performance analysis

The stack



What is the deepest the Stack gets (number of stack frames) during the solving of this maze?

A. Less than 5
B. 5-10
C. 11-20
D. More than 20
E. Other/none/more



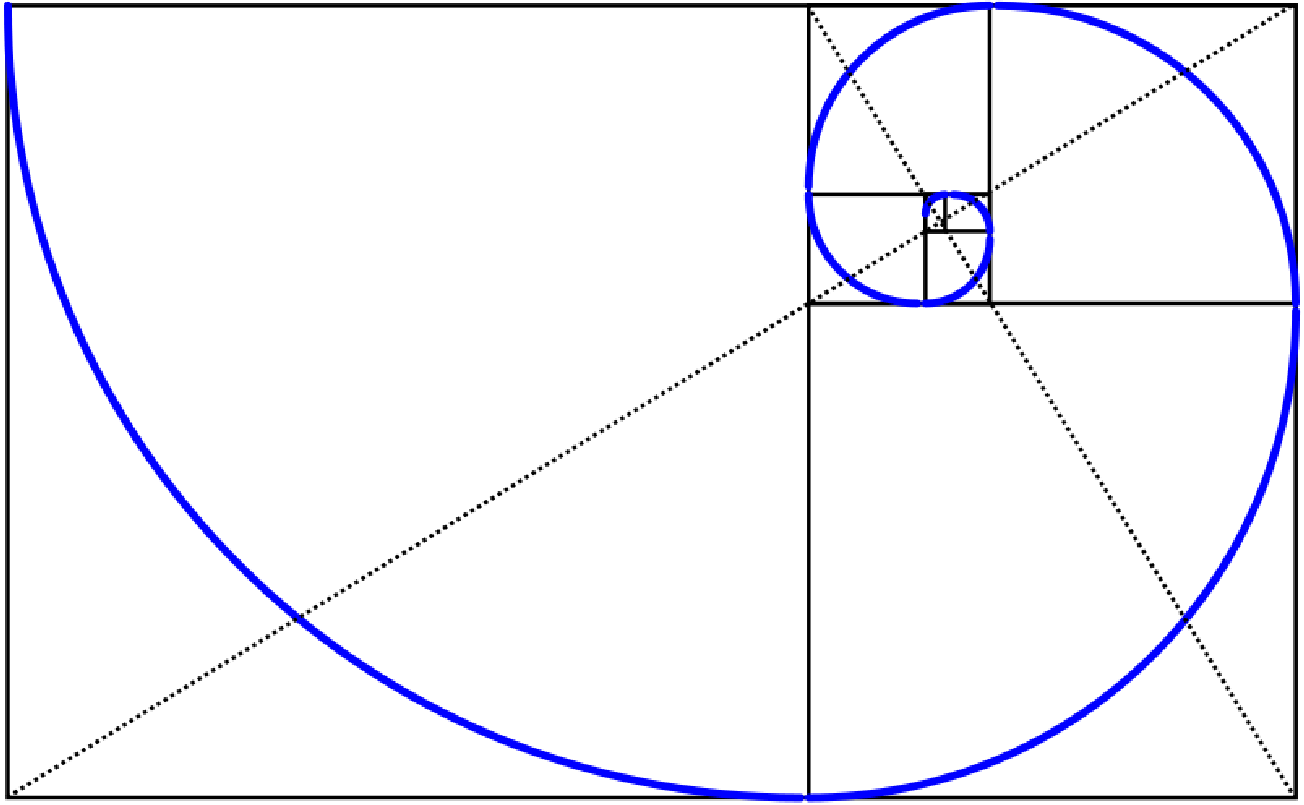
Contrast: Recursive maze-solving vs. Word ladder

- With word ladder, you did **breadth-first search**
- Our recursive maze-solver uses **depth-first search**

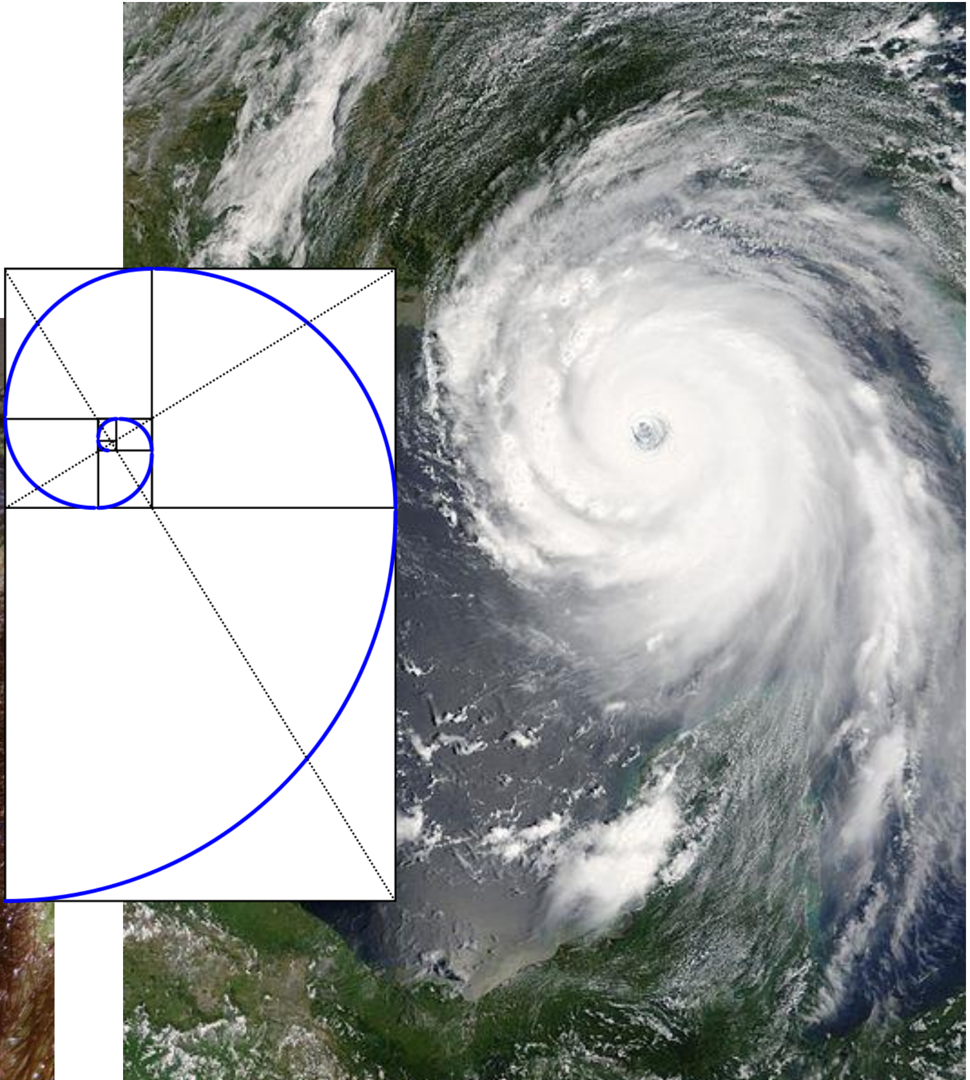
- Both are possible for maze-solving!

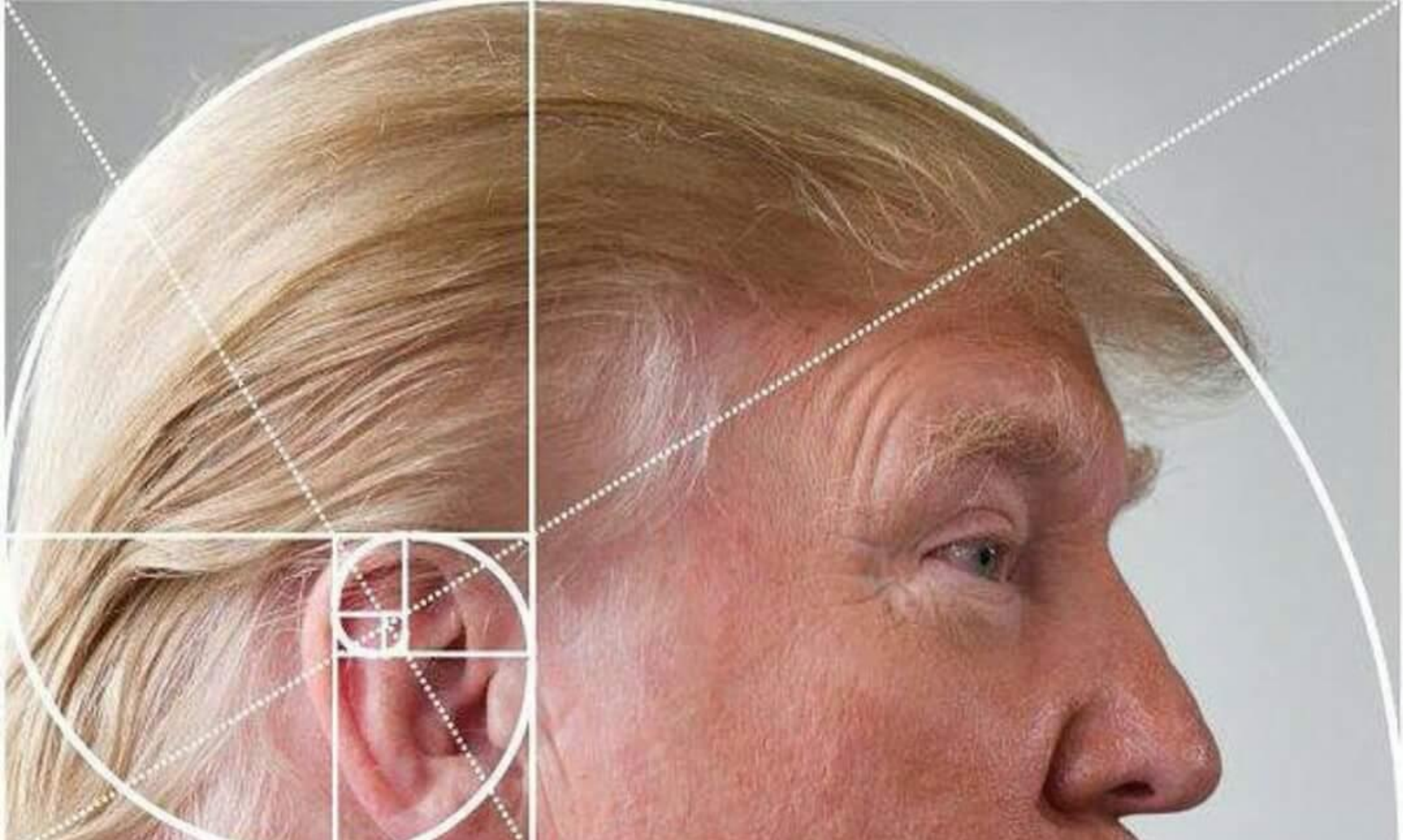
- The contrast between these approaches is a theme that you'll see again and again in your CS career

Fibonacci



Fibonacci in nature





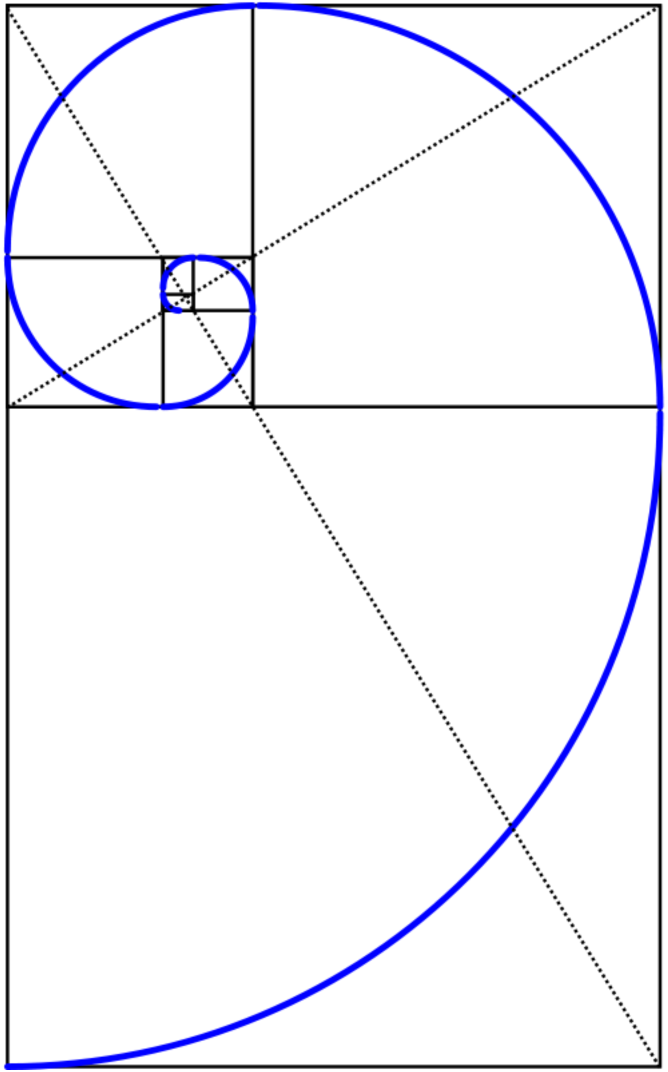
Fibonacci

f(0) = 0

f(1) = 1

For all n > 1:

- f(n) = f(n-1) + f(n-2)



Fibonacci

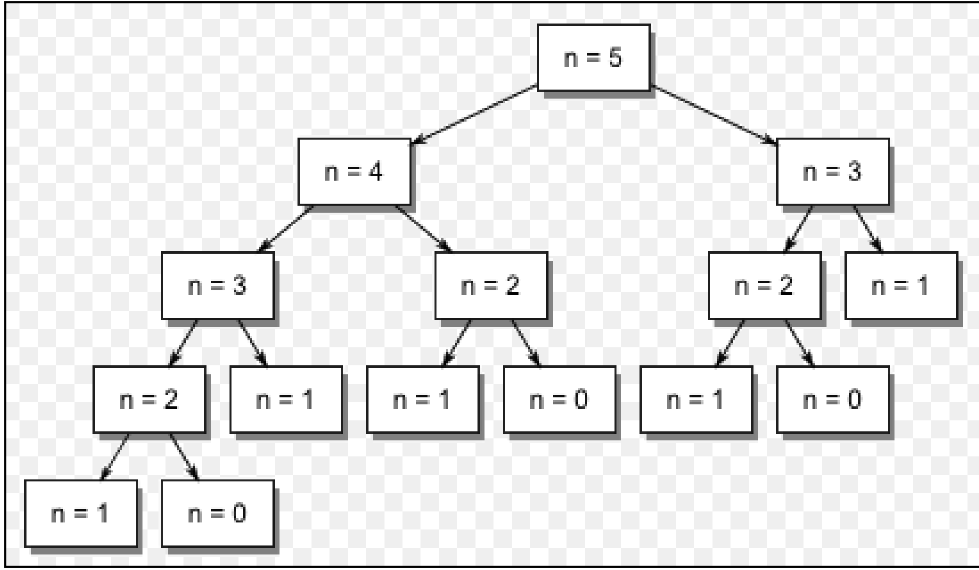


Image is in the public domain.
http://commons.wikimedia.org/wiki/File:Fibonacci_call_tree_5.gif

F(0) = 0

F(1) = 1

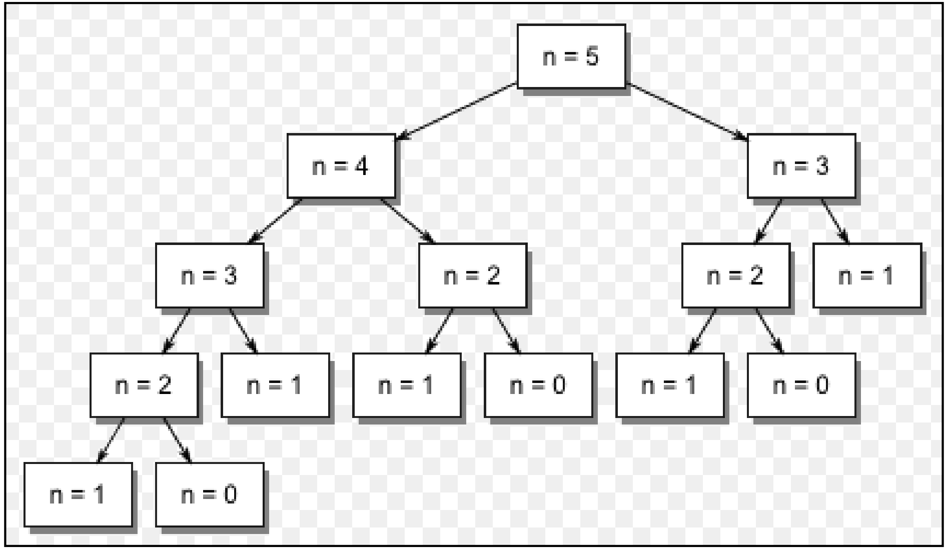
F(n) = F(n-1) + F(n-2)   *for n > 1*

Work is duplicated throughout the call tree

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

Fibonacci

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

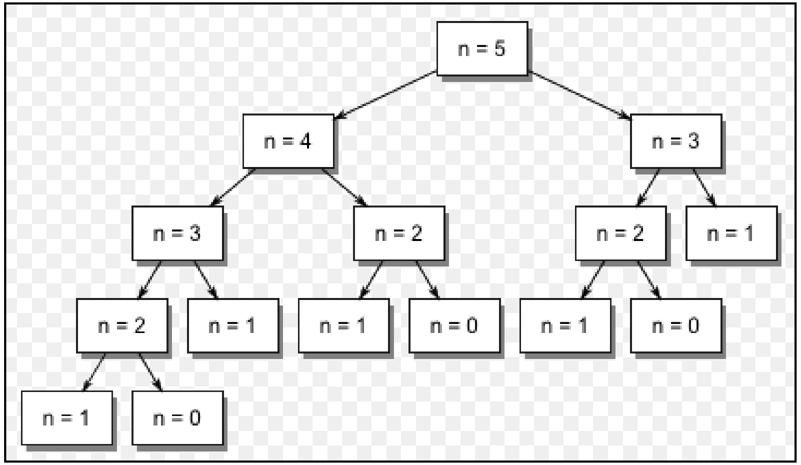


How many times would we calculate Fib(2) while calculating Fib(6)? ***See if you can just "read" it off the chart above.***

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	



How many times would we calculate Fib(2) while calculating Fib(7)?

How many times would we calculate Fib(2) while calculating Fib(8)?

Efficiency of naïve Fibonacci implementation

When we **added 1** to the input to Fibonacci, the number of times we had to calculate a given subroutine **nearly doubled** (~1.6 times*)

- Ouch!

Can we predict how much time it will take to compute for arbitrary input n?

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

Efficiency of naïve Fibonacci implementation

Can we predict how much time it will take to compute for arbitrary input n?

Each time we add 1 to the input, the time increases by a factor of 1.6

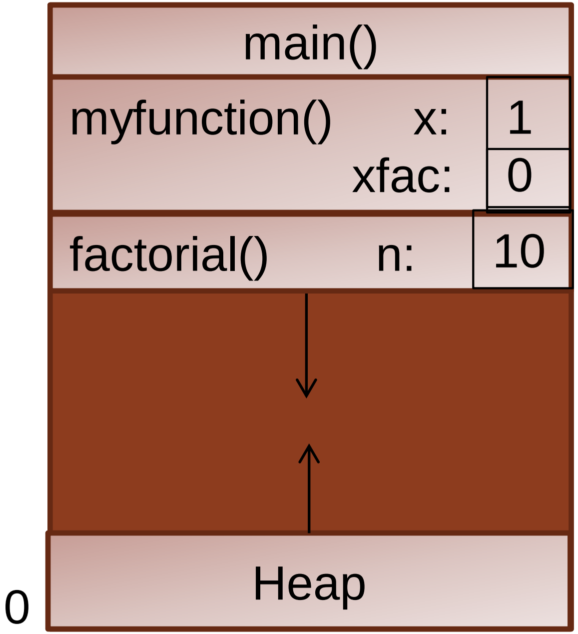
For input n, we multiply the "baseline" time by 1.6 n times:

- $\underbrace{b * 1.6 * 1.6 * 1.6 * \ldots * 1.6}_{\text{n times}} = b * 1.6^n$

- We don't really care what b is exactly (different on every machine anyway), so we just normalize by saying b = 1 "time unit" (i.e. we remove b)

Aside: recursion isn't *always* this bad!

Memory



Recursive code

```
long factorial(int n) {
     cout << n << endl;
     if (n == 1) return 1;
     else return n * factorial(n – 1);
}
```

"Roughly" how much time does factorial take, as a function of the input n?

It's better!! Just b * n = n (when we say that b=1 because we define b = one "time unit")

Fibonacci

Assume we have to calculate each unique function call once, *but never again*

We “remember” the answer from the first time

How many rectangles <u>remain</u> in the above chart for n=5?

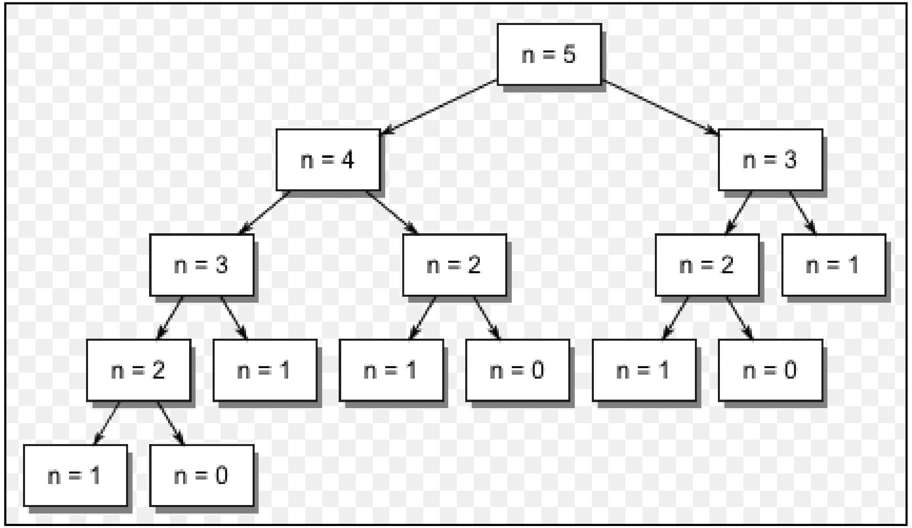


Image is in the public domain.
http://commons.wikimedia.org/wiki/File:Fibonacci_call_tree_5.gif

Memo-ization

Take notes (“**memos**”) as you go

For Fibonacci, we will have answers for F(i) for all i, $0 \le i \le n$, so a simple array or Vector can store intermediate results:

- results[i] stores Fib(i)

Big-O Performance Analysis

$\log_2 n$	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	**1,300,000,000**			

# of Facebook accounts

$\log_2 n$	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	**1,300,000,000**			

2.4s

Easy!

# of Facebook accounts

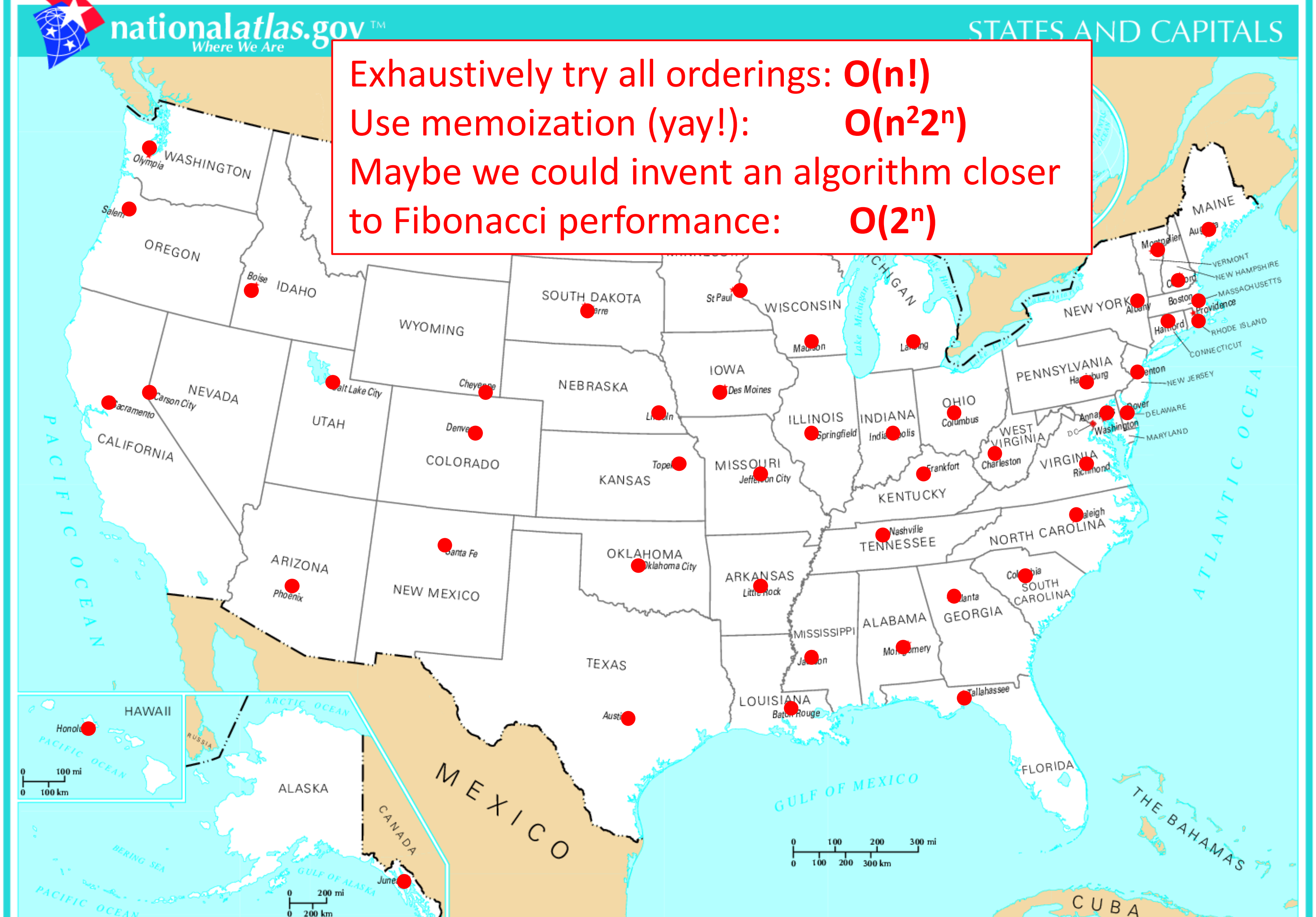
Traveling Salesperson Problem:

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!)**

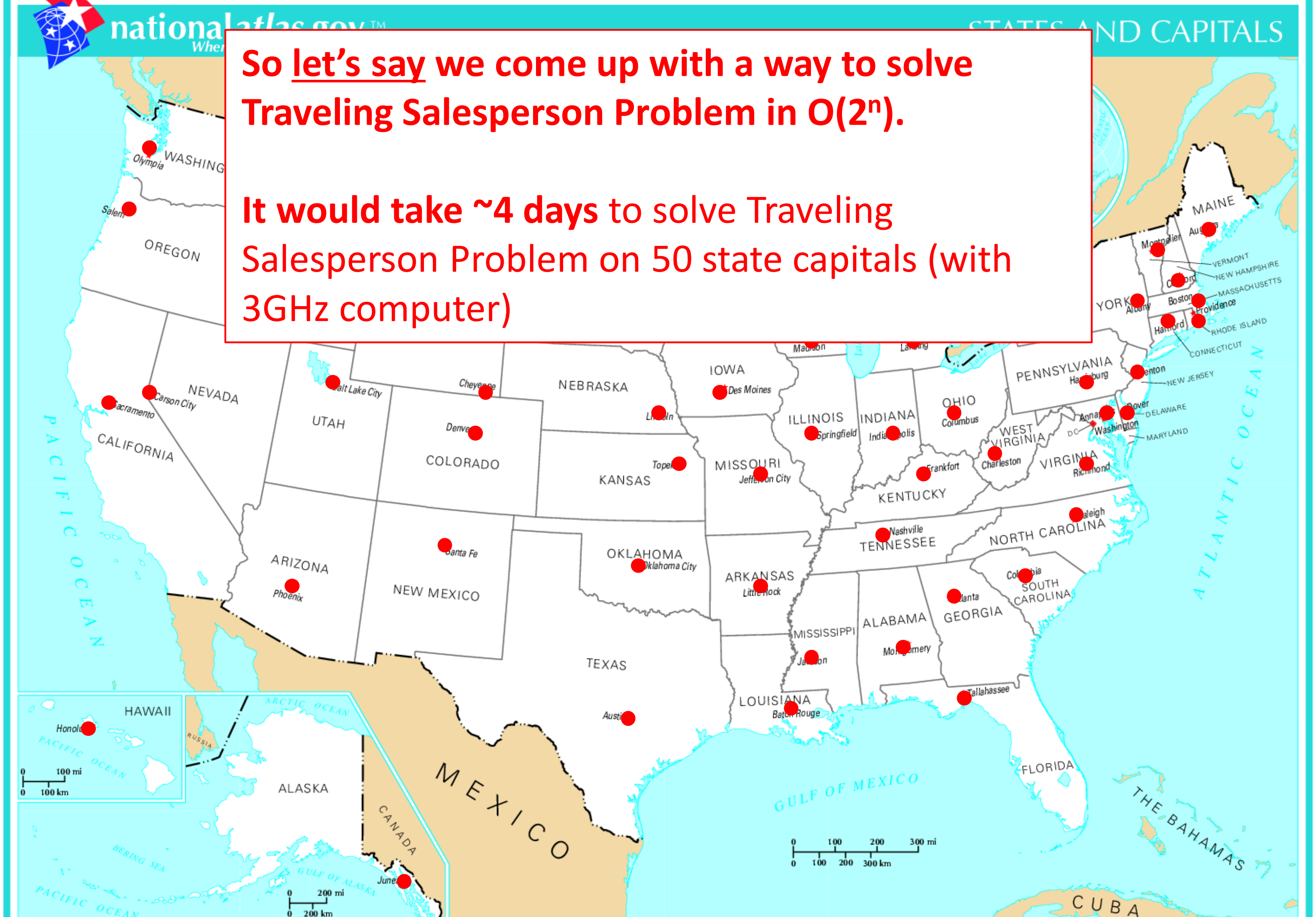
Use memoization (yay!): **$O(n^2 2^n)$**

Maybe we could invent an algorithm closer to Fibonacci performance: **$O(2^n)$**



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 (with 3GHz computer)



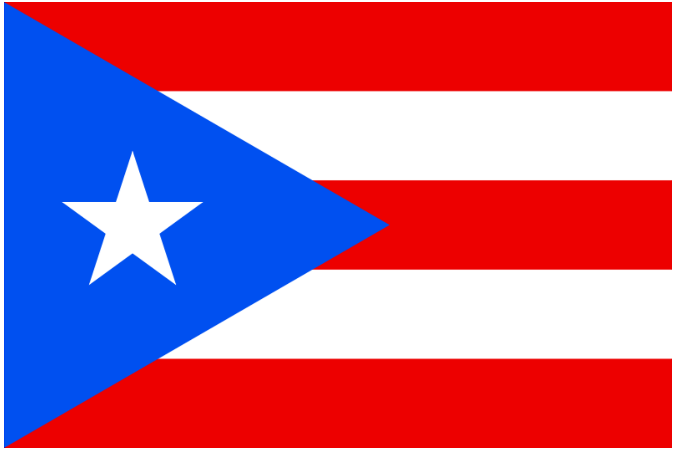
Two ***tiny*** little updates

Imagine we approve statehood for Puerto Rico

- Add San Juan, the capital city

Also add Washington, DC

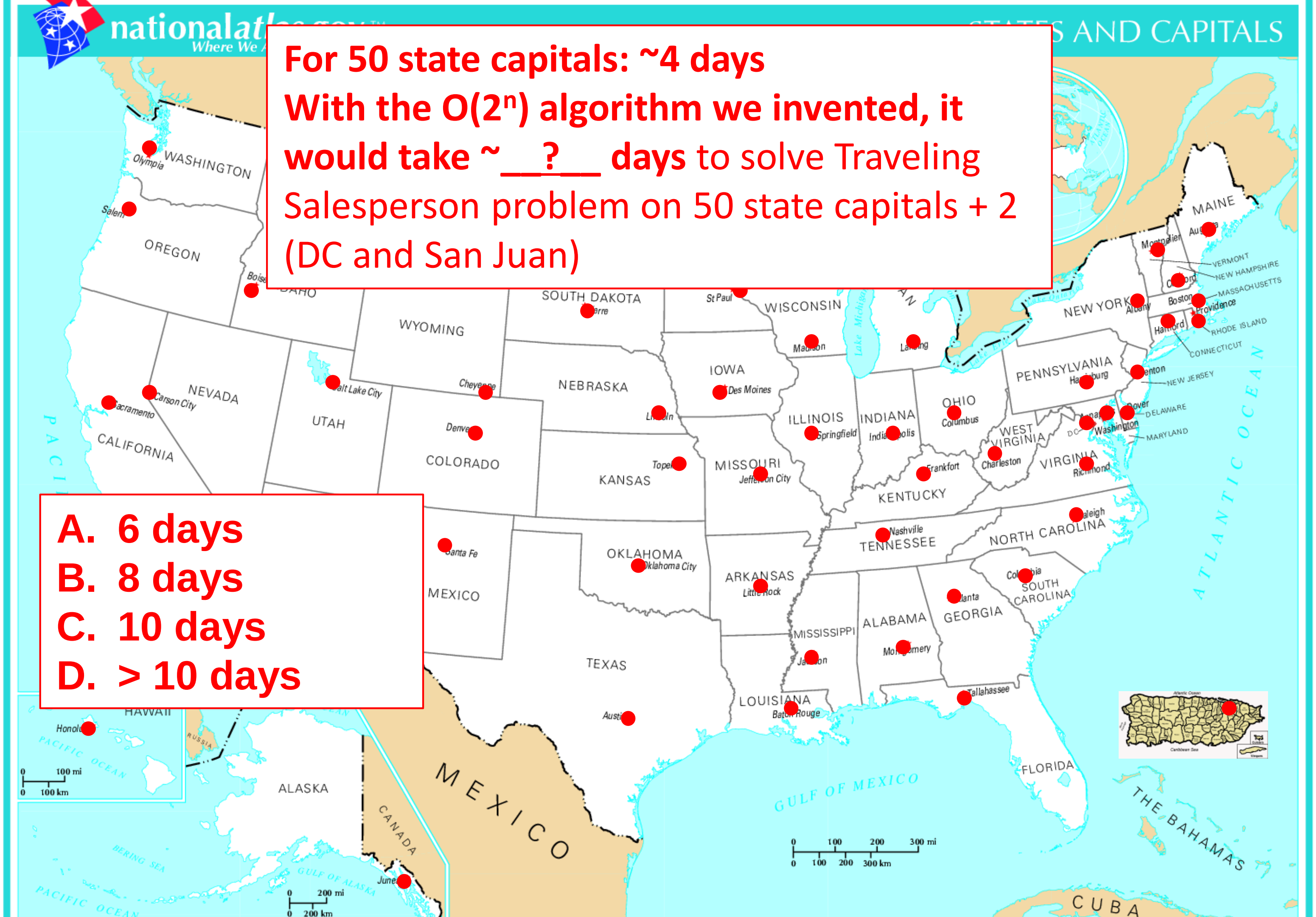
Now 52 capital cities instead of 50



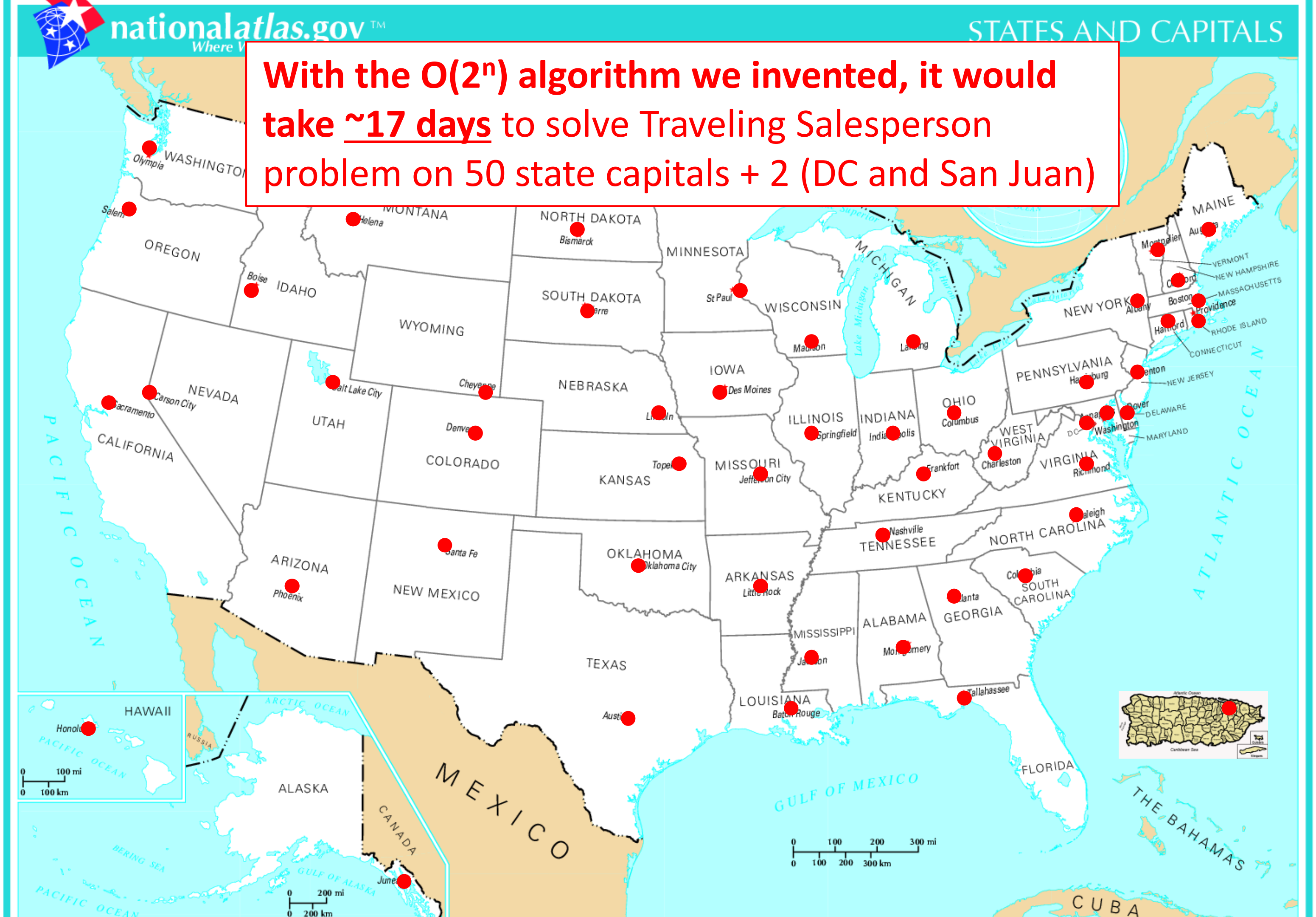
For 50 state capitals: ~4 days

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



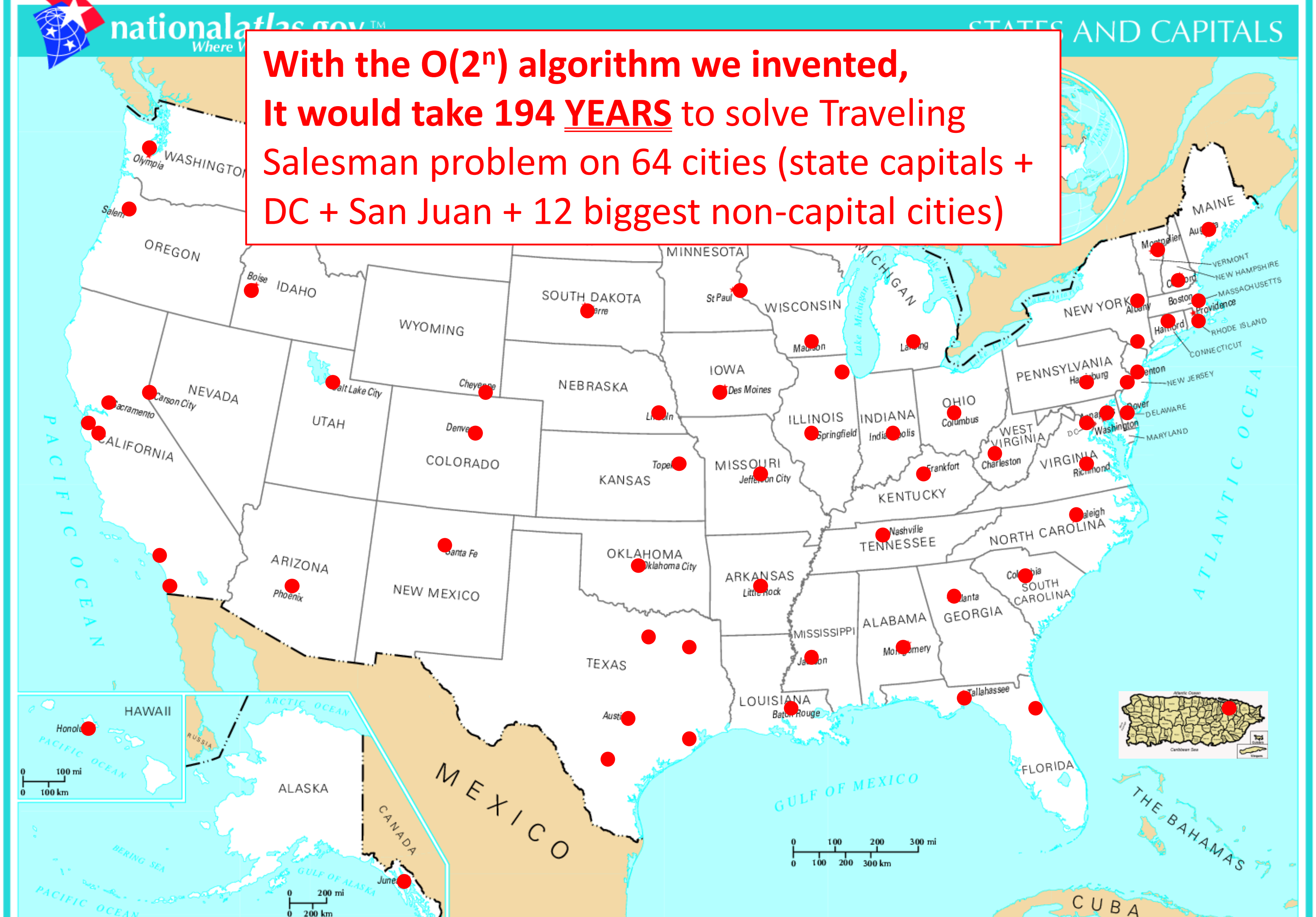
With the $O(2^n)$ algorithm we invented, it would take <u>~17 days</u> to solve Traveling Salesperson problem on 50 state capitals + 2 (DC and San Juan)



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 <u>YEARS</u> 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	**1,300,000,000**			

194 **YEARS**

# of Facebook accounts

$\log_2 n$	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	**1,300,000,000**			

3.59E+21 **YEARS**

# of Facebook accounts

$\log_2 n$	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	**1,300,000,000**			

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

YEARS

# of Facebook accounts

$\log_2 n$	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	**1,300,000,000**			

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

# of Facebook accounts

$\log_2 n$	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	**1,300,000,000**			

# of Facebook accounts

1.42E+137 **YEARS** (another way of thinking about the size: including commas, this number of years cannot be written in a single tweet)

$\log_2 n$	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	**1,300,000,000**	39000000000 (13s)	1690000000000000000 (18 years)	LOL

# of Facebook accounts

$\log_2 n$	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	**1,300,000,000**	39000000000 (13s)	1690000000000000000 (18 years)	$2.3 \times 10^{391,338,994}$

# of Facebook accounts

$\log_2 n$	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**	[illegible]	[illegible]	[illegible]
10	**1,024**	[illegible]	[illegible]	[illegible]
30	**1,300,000,000**	39000000000 (13s)	1690000000000000000 (18 years)	$2.3 \times 10^{391,338,994}$

2^n is way into crazy LOL territory, but **look at $n\log_2 n$—only 13 seconds!!**

# of Facebook accounts


THIS IS ME NOT CARING
ABOUT PERFORMANCE TUNING UNLESS IT CHANGES BIG-O
memegenerator.net

Big-O

Extracting time cost from example code

Translating code to a f(n) model of the performance

(n = size of vector)

	Statements	Cost
1	double findAvg (Vector<int>& grades){	
2	double sum = 0;	1
3	int count = 0;	1
4	while (count < grades.size()) {	*n* + 1
5	sum += grades[count];	*n*
6	count++;	*n*
7	}	
[illegible]	[illegible])	1
[illegible]	[illegible] .size();	
[illegible]	[illegible]	1
11	return 0.0;	
12	}	
ALL		**3n+5**

Do we really care about the +5?
Or the 3 for that matter?

Formal definition of Big-O

We say a function f(n) is **"big-O"** of another function g(n), and write "f(n) is **O**(g(n))" if there exist positive constants c and n_0 such that:

$$f(n) \leq c\, g(n) \textit{ for all } n \geq n_0.$$

Big-O

We say a function f(n) is **"big-O"** of another function g(n), and write "f(n) is **O**(g(n))" if there exist positive constants c and n_0 such that:

$$f(n) \leq c\, g(n) \textit{ for all } n \geq n_0.$$

What you need to know:

O(X) describes an "upper bound"—**the algorithm will perform <u>no worse</u> than X**

- We ignore constant factors in saying that
- We ignore behavior for "small" n