

Programming Abstractions

CS106B

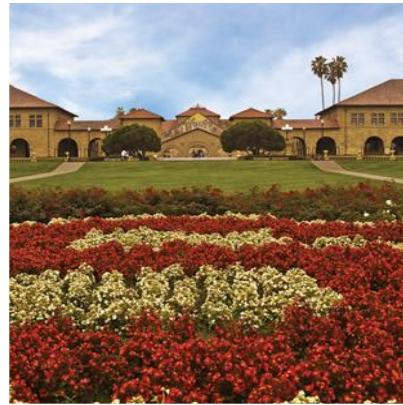
Cynthia Bailey Lee
Julie Zelenski

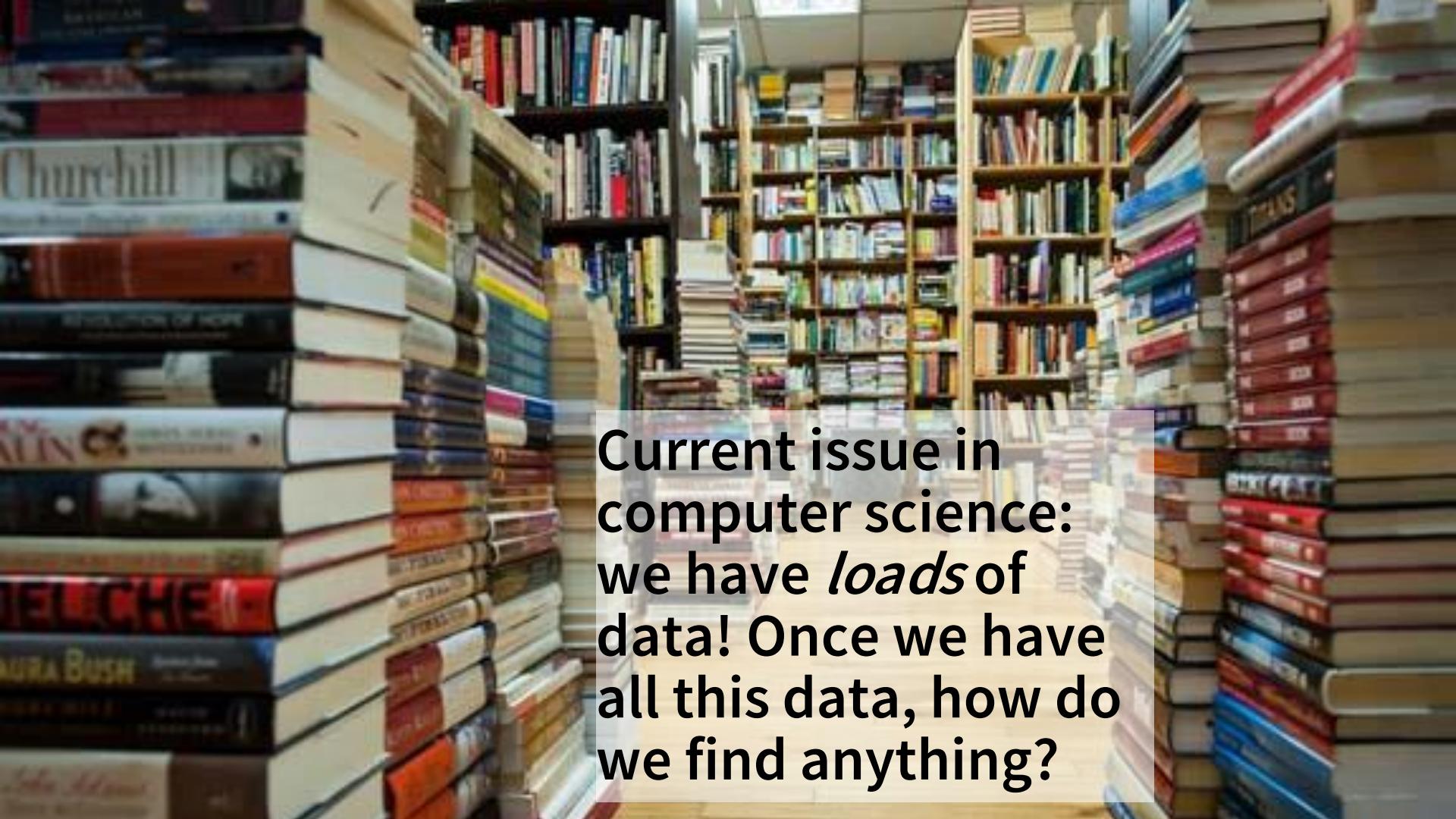
Today's Topics:

- 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 entire first half, and now only have the second half to search
- We could search the second half and proceed forward... but why do that when we can do it **recursively**?

Jump right to the middle of the region to search

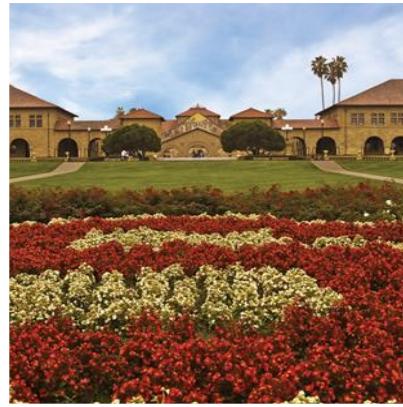
Binary Search pseudocode

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

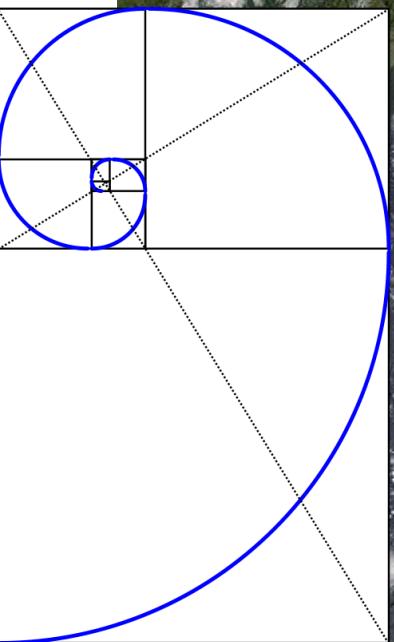
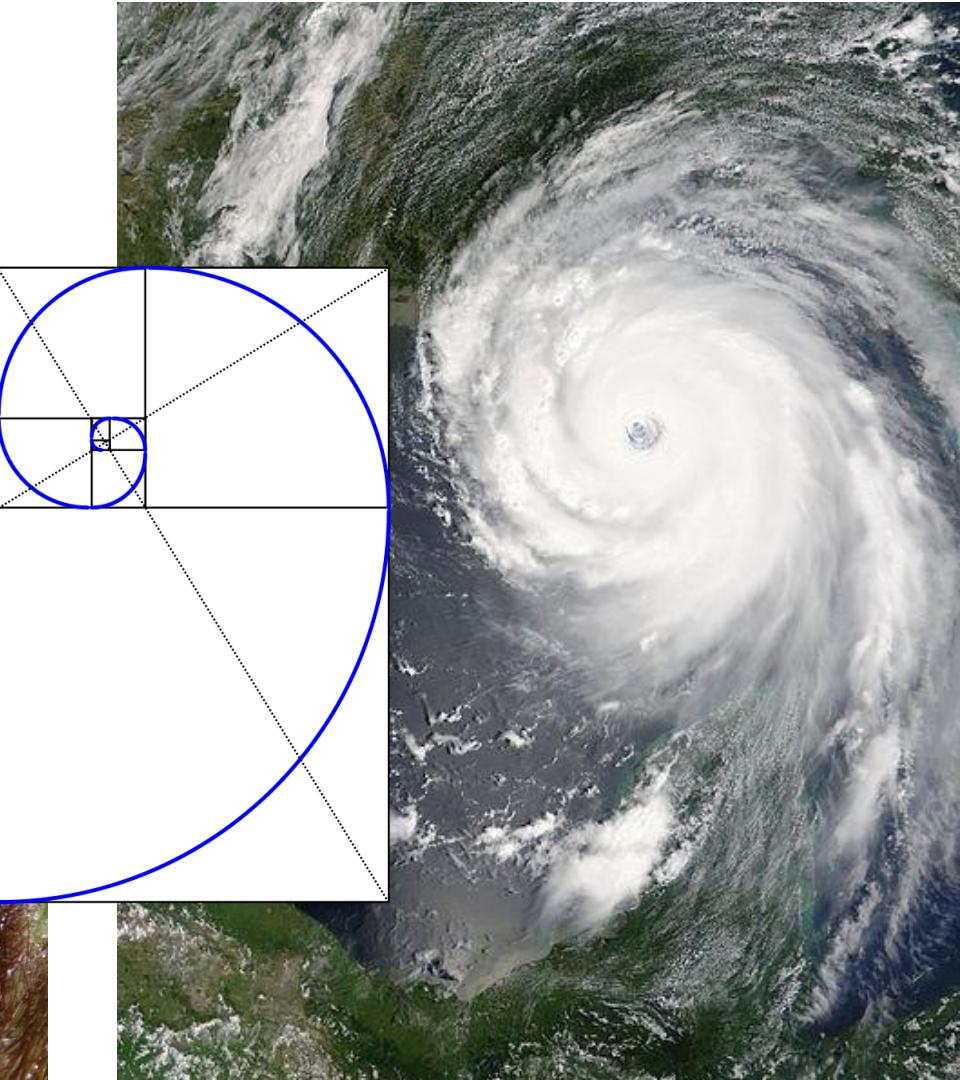
```
bool binarySearch(Vector<int>& data, int key)
{
    if (data.size() == 0) {
        return false;
    }
    if (key == data[midpoint]) {
        return true;
    } else if (key < data[midpoint]) {
        return binarySearch(data[first half only], key);
    } else {
        return binarySearch(data[second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

The Fibonacci Sequence

*MATH NERD REJOICING
INTENSIFIES*



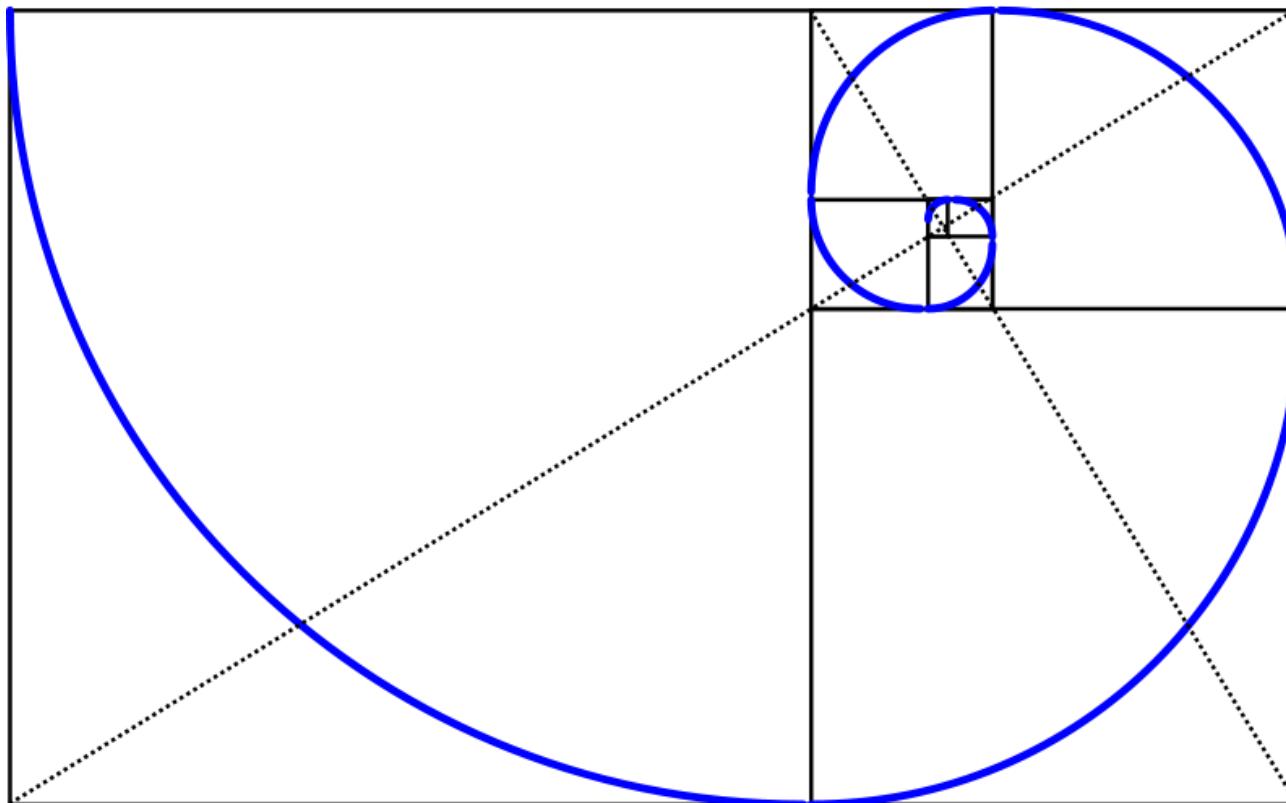
Fibonacci in nature



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

Fibonacci

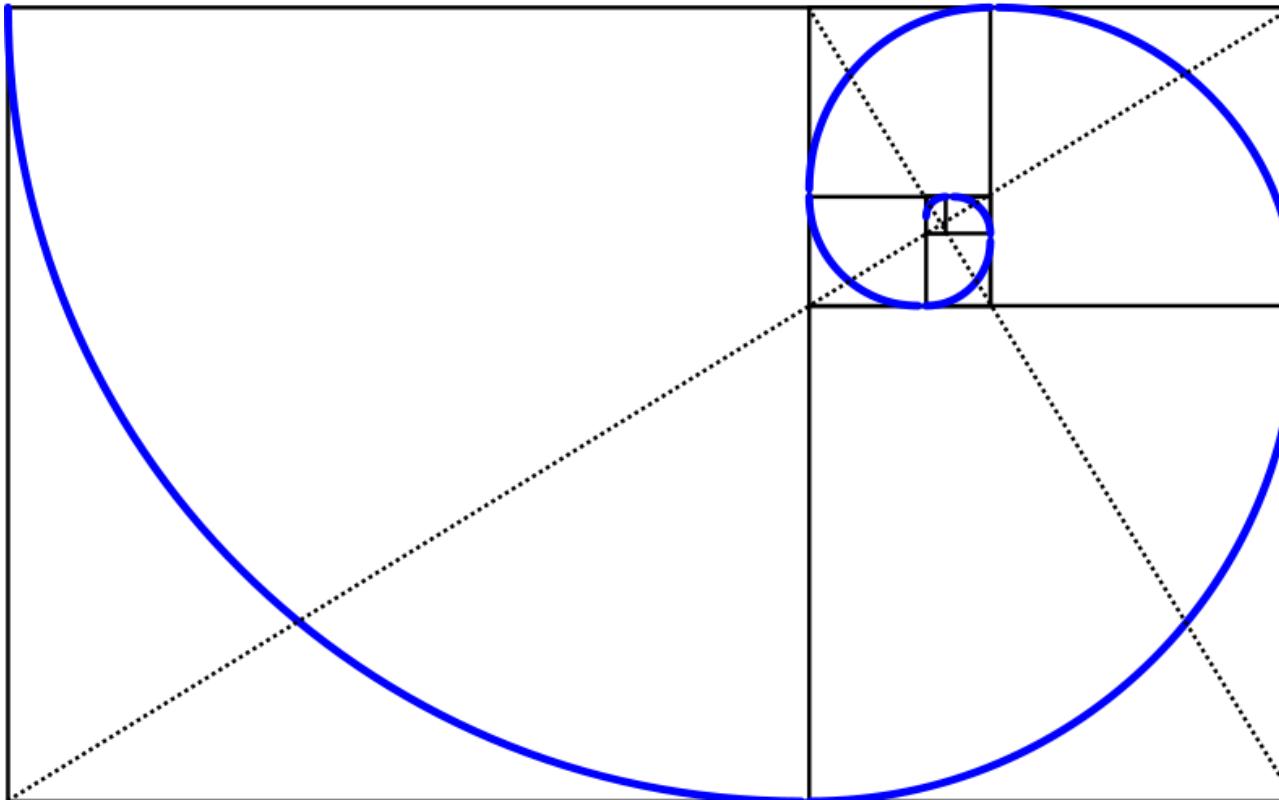
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,



This image is licensed under the [Creative Commons Attribution-Share Alike 3.0 Unported](#) license. http://commons.wikimedia.org/wiki/File:Golden_spiral_in_rectangles.png

Fibonacci

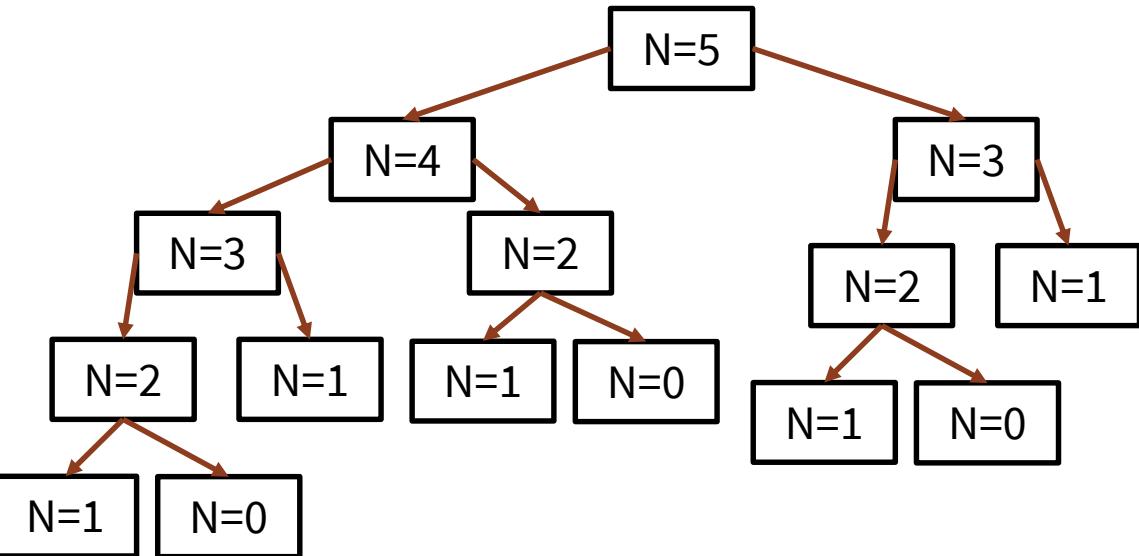
0	1	2	3	4	5	6	7	8	9	10	11
0	1	1	2	3	5	8	13	21	34	55	89



This image is licensed under the [Creative Commons Attribution-Share Alike 3.0 Unported](https://creativecommons.org/licenses/by-sa/3.0/) license. http://commons.wikimedia.org/wiki/File:Golden_spiral_in_rectangles.png

Fibonacci

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

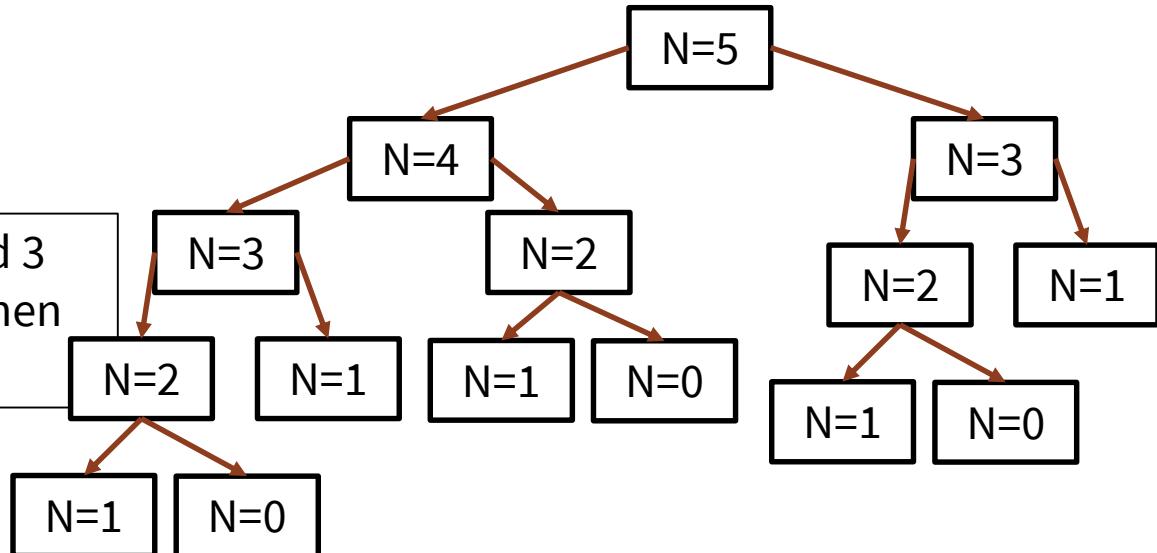


Work is duplicated throughout the call tree

- 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)!



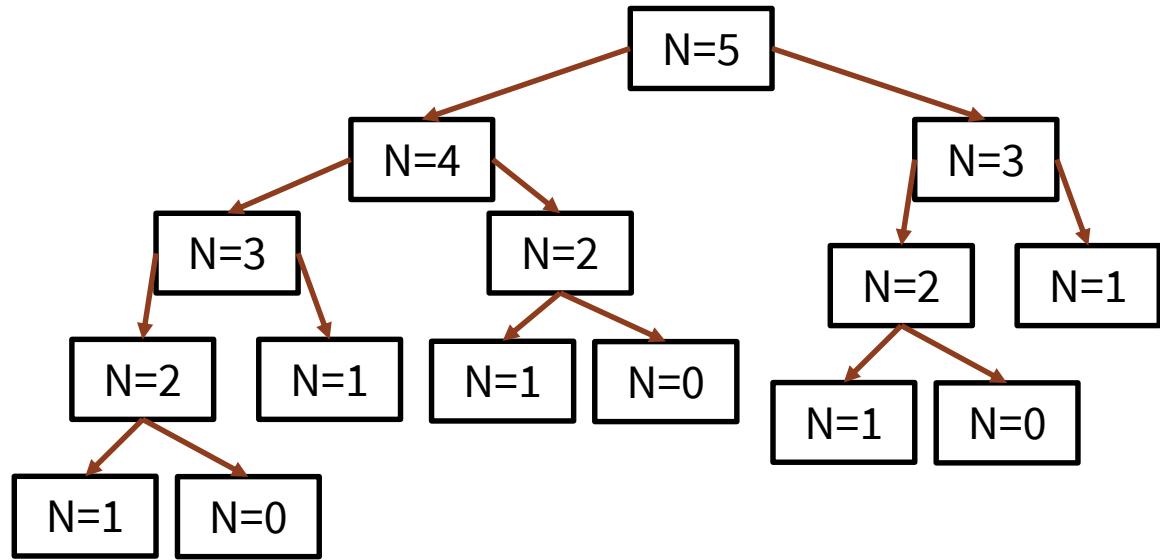
How many times would we calculate fib(2) while calculating fib(6)?

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** ($\sim 1.6^*$ 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

S Vector x + ▾

← → ⌂ 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 using square brackets, as well as inserting and removing elements. Operations that access elements by index in $O(1)$ time. Operations, such as insert and remove, that must rearrange elements run in $O(N)$ time.

Constructor

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

Methods

Big-O analysis in computer science



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

Article Talk

Read Edit View history

Search Wikipedia



Binary search algorithm

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 position of a target value in a sorted array. It compares the target value to the middle element of the array. If they are unequal, the search continues on the remaining half of the array. If the target is found, the search ends. If the array is empty, the target is not in the array.

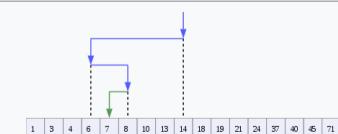
Binary search runs in at worst logarithmic time, making $O(\log n)$ comparisons, where n is the number of elements in the array, the O is Big O notation, and log is the logarithm. Binary search takes constant ($O(1)$) space, meaning that the space taken by the algorithm is the same for any number of 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

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

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

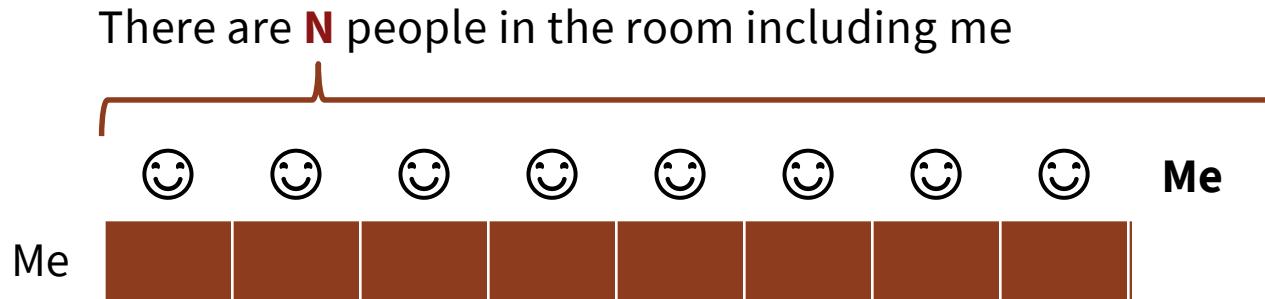
there exist positive constants c and n_0 such that
$$f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0.$$

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 ± 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:

$$O(N - 1)$$

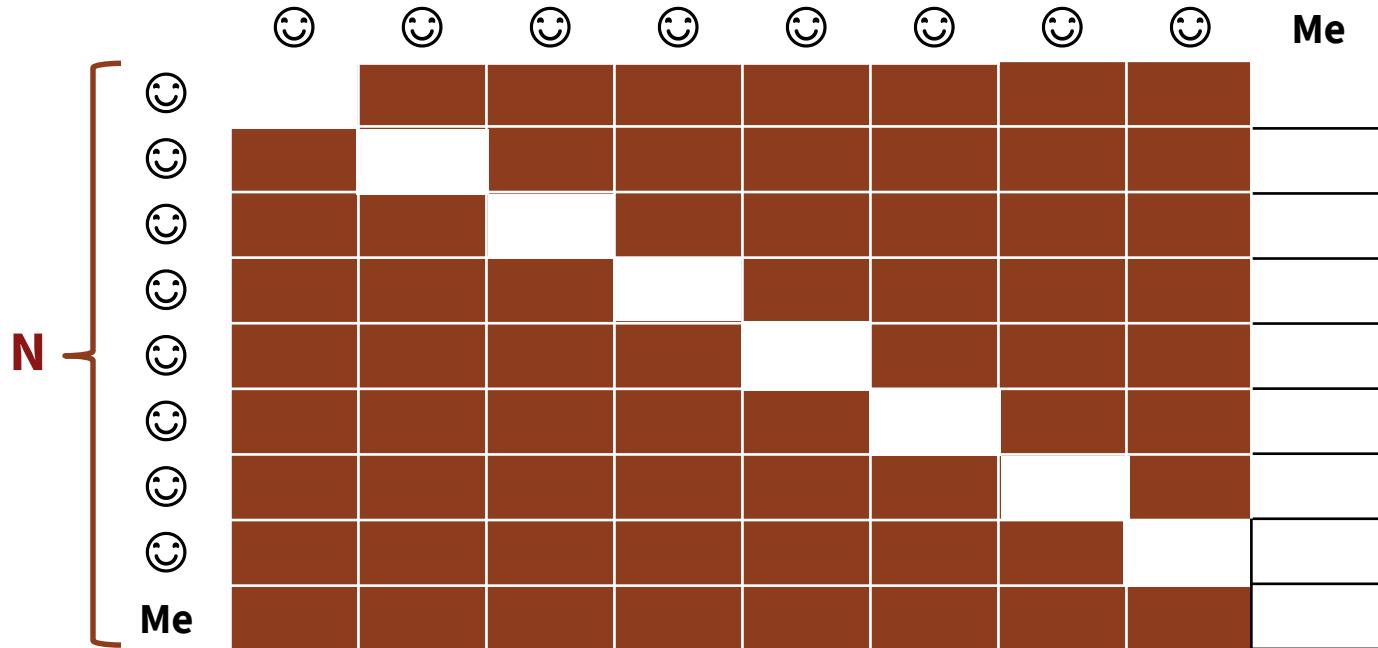
Similarly, if we said that each introduction **takes 3 seconds**, the amount of time is **$3(N - 1) = 3N - 3$** , but we disregard the constant 3s:

$$O(3N - 3)$$

Before we start, let's get introduced

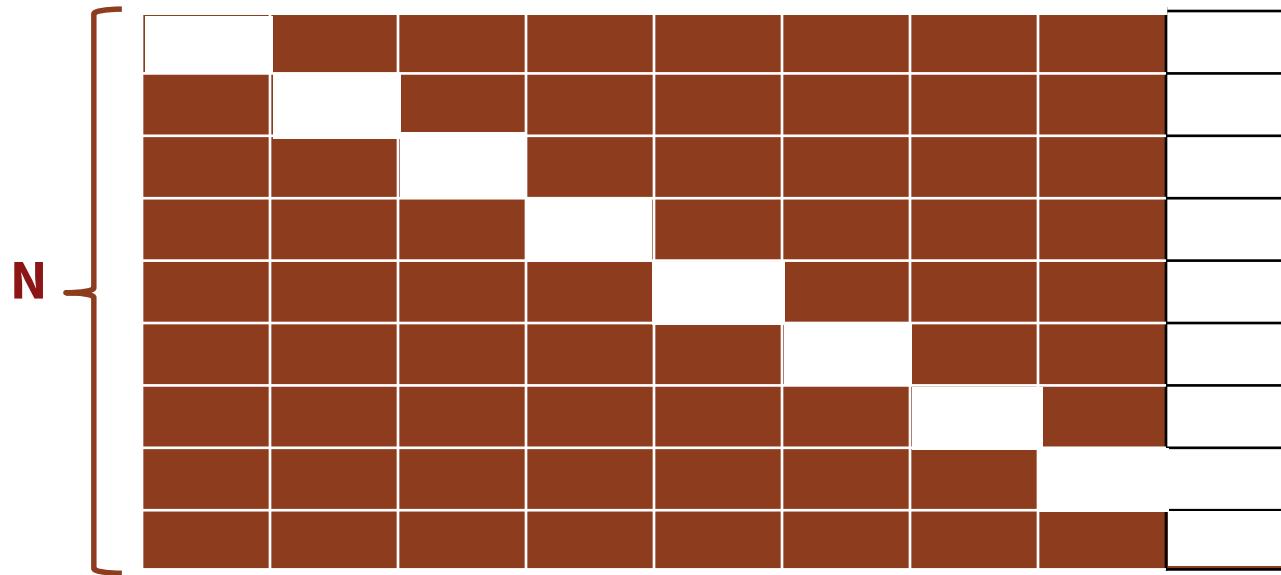
What if I not only want you to be introduced to me, but to each other?

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: $O(n^2n)$

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



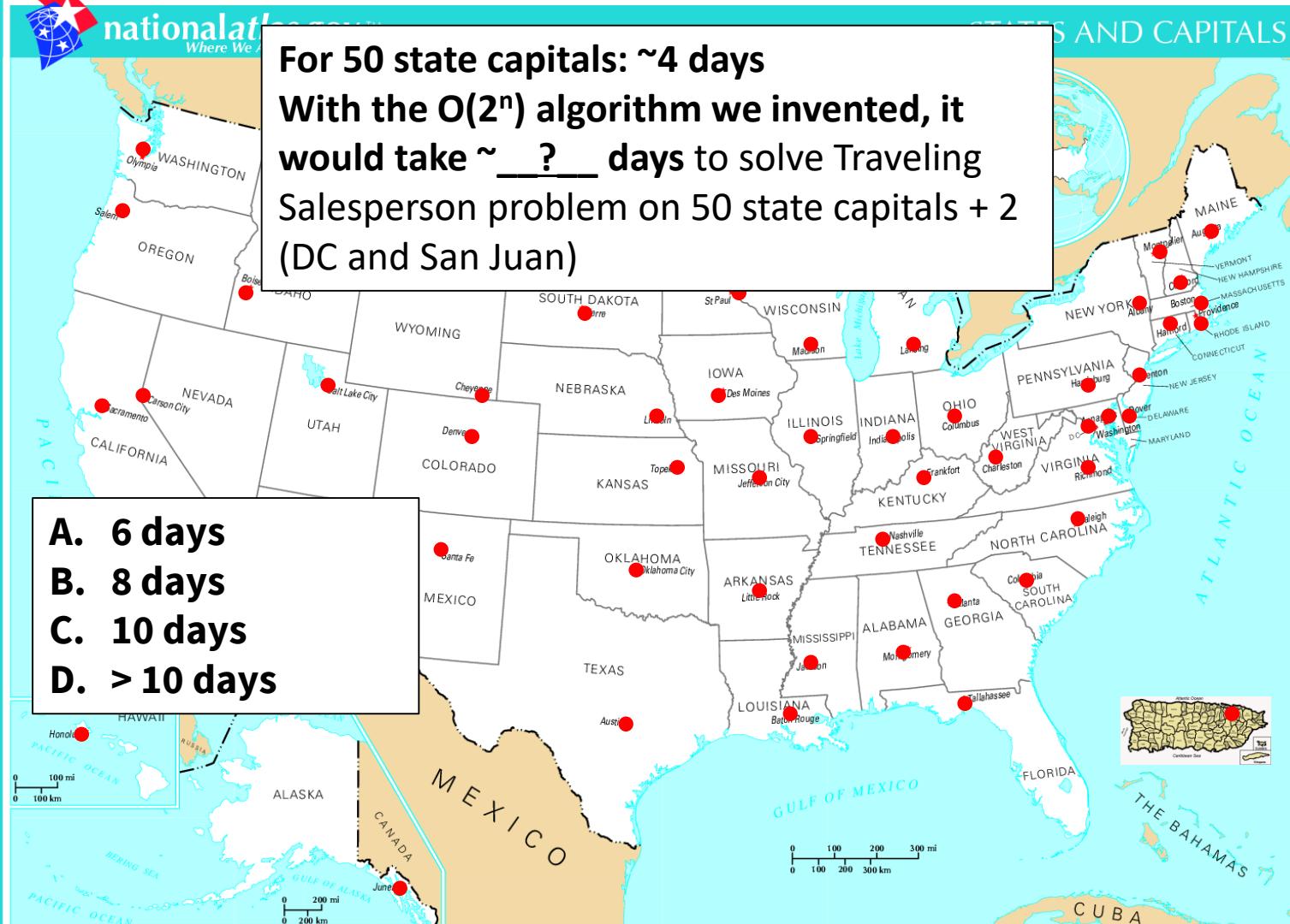
This work has been released into the [public domain](#) by its author, [Madden](#).
This applies worldwide.

Now 52 capital cities instead of 50



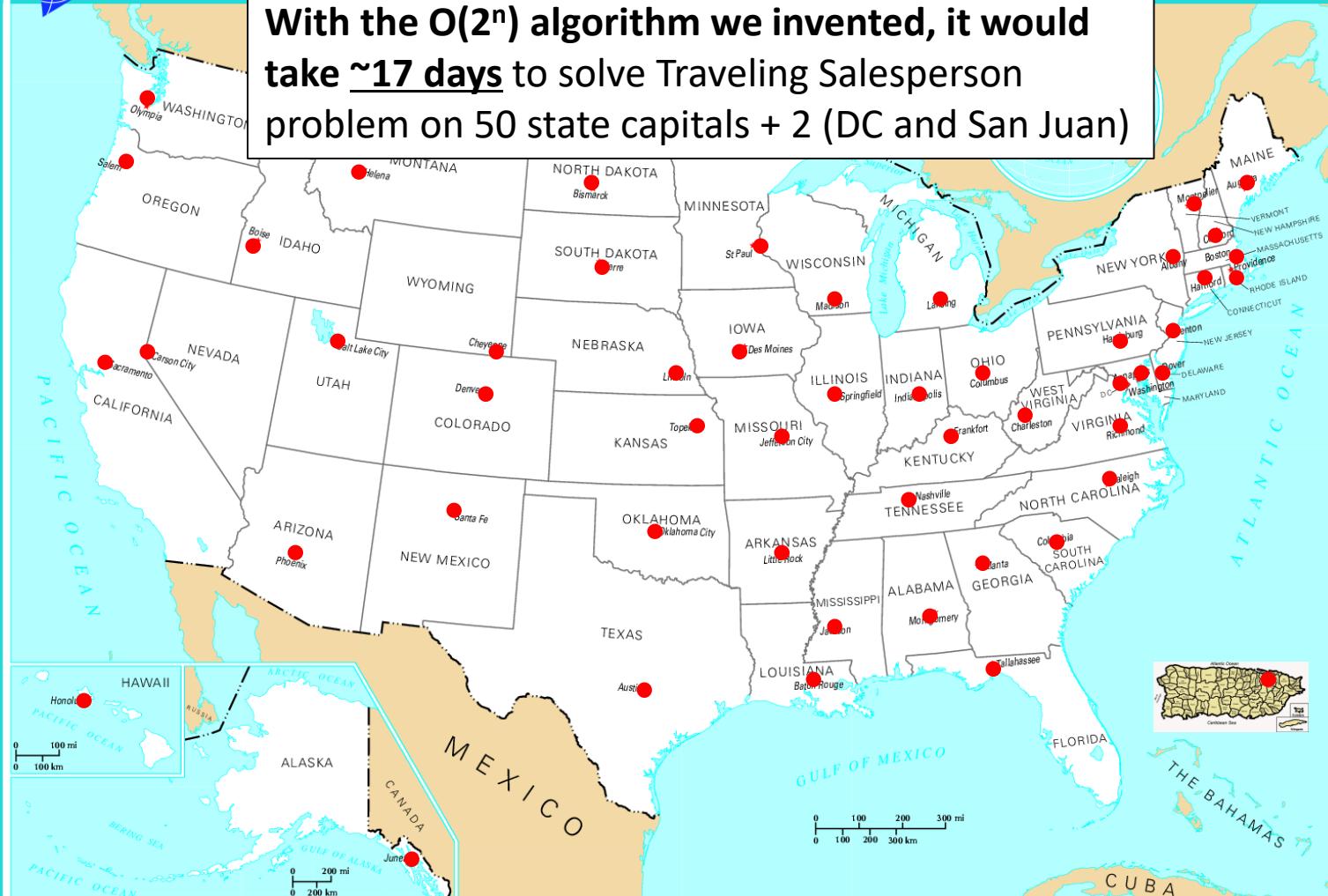
For 50 state capitals: ~4 days
With the $O(2^n)$ algorithm we invented, it would take ~ ? days to solve Traveling Salesperson problem on 50 state capitals + 2 (DC and San Juan)

- 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	1.84×10^{19}
7	128			194 YEARS
8	256			
9	512			
10	1,024			
30	2,700,000,000			

$\log_2 n$	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			3.59E+21 YEARS
9	512			
10	1,024			
30	2,700,000,000			

$\log_2 n$	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	3,590,000,000,000,000,000,000 YEARS		
9	512			
10	1,024			
30	2,700,000,000			

$\log_2 n$	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 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	1.84×10^{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			1.42E+137 YEARS
30	2,700,000,000			

$\log_2 n$	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 (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 00 (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}
10	1,024	9,216	1,048,576	1.80×10^{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	316,659,201,535,488		
46	70,368,744,177,664	633,318,403,070,976		
47	140,737,488,355,328	1,266,636,806,141,952		
48	281,474,976,710,656	2,533,273,612,283,904		
49	562,949,953,421,312	5,066,547,224,566,808		
50	1,125,899,906,842,624	10,133,094,449,133,616		
51	2,251,799,813,685,248	20,266,188,898,267,232		
52	4,503,599,627,370,496	40,532,377,796,534,464		
53	9,007,199,254,740,992	81,064,755,593,068,928		
54	18,014,398,509,481,984	162,129,511,186,137,856		
55	36,028,797,018,963,968	324,258,022,372,275,712		
56	72,057,594,037,927,936	648,516,044,744,551,424		
57	144,115,188,075,855,872	1,296,032,089,489,102,848		
58	288,230,376,151,711,744	2,592,064,178,978,205,696		
59	576,460,752,303,423,488	5,184,128,357,956,411,392		
60	1,152,921,504,606,846,976	10,368,256,715,912,822,784		
61	2,305,843,009,213,693,952	20,736,513,431,825,645,568		
62	4,611,686,018,427,387,904	41,473,026,863,651,291,136		
63	9,223,372,036,854,775,808	82,946,053,727,302,582,272		
64	18,446,744,073,709,551,616	165,892,107,454,605,164,544		
65	36,893,488,147,419,103,232	331,784,214,909,210,328,088		
66	73,786,976,294,838,206,464	663,568,429,818,420,656,176		
67	147,573,952,589,676,412,928	1,327,136,859,636,841,312,352		
68	295,147,905,179,352,825,856	2,654,273,719,273,682,624,704		
69	590,295,810,358,705,651,712	5,308,547,438,547,365,249,408		
70	1,180,591,620,717,411,303,424	10,617,094,877,094,730,498,816		
71	2,361,183,241,434,822,606,848	21,234,189,754,189,461,997,632		
72	4,722,366,482,869,644,813,696	42,468,379,508,378,923,995,264		
73	9,444,732,965,739,289,627,392	84,936,759,016,757,847,990,528		
74	18,889,465,931,478,579,254,784	169,873,518,033,515,695,981,056		
75	37,778,931,862,957,158,509,568	339,747,036,067,031,391,962,112		
76	75,557,863,725,914,317,018,136	679,494,072,134,062,783,924,224		
77	151,115,727,451,828,634,036,272	1,358,988,144,268,125,567,848,448		
78	302,231,454,903,657,268,072,544	2,717,976,288,536,251,135,696,896		
79	604,462,909,807,314,536,145,088	5,435,952,577,072,502,271,393,792		
80	1,208,925,819,614,629,072,290,176	10,871,905,154,145,004,542,787,584		
81	2,417,851,639,229,258,144,580,352	21,743,810,308,290,008,085,575,168		
82	4,835,703,278,458,516,288,160,704	43,487,620,616,580,016,171,150,336		
83	9,671,406,556,917,032,576,320,408	86,975,241,233,160,032,342,300,672		
84	19,342,813,113,834,065,152,640,816	173,950,482,466,320,064,684,601,344		
85	38,685,626,227,668,130,305,281,632	347,900,964,932,640,128,369,202,688		
86	77,371,252,455,336,260,610,563,264	695,801,928,865,280,256,738,405,376		
87	154,742,504,910,672,521,221,126,528	1,391,603,857,730,560,513,476,810,752		
88	309,485,009,821,345,042,442,253,056	2,783,207,715,461,120,026,953,621,504		
89	618,970,019,642,690,084,884,506,112	5,566,415,430,922,240,053,907,243,008		
90	1,237,940,039,285,380,168,769,012,224	11,132,830,861,844,480,107,814,486,016		
91	2,475,880,078,570,760,337,538,024,448	22,265,661,723,688,960,215,628,972,032		
92	4,951,760,157,141,520,675,076,048,896	44,531,323,447,377,920,431,257,944,064		
93	9,903,520,314,283,040,150,152,096,192	89,062,646,894,754,840,862,515,888,128		
94	19,807,040,628,566,080,300,304,192,384	178,125,293,789,508,680,725,031,776,256		
95	39,614,081,257,132,160,600,608,384,768	356,250,587,579,017,360,450,063,552,512		
96	79,228,162,514,264,320,120,121,768,536	712,501,175,158,034,720,900,127,105,024		
97	158,456,325,028,528,640,240,243,536,072	142,502,350,317,068,440,800,254,210,048		
98	316,912,650,057,056,160,480,487,072,144	285,004,700,634,136,880,600,508,420,096		
99	633,825,300,114,112,320,960,974,144,288	570,009,401,268,273,760,120,016,840,192		
100	1,267,650,600,228,224,641,921,948,288,576	1,140,018,802,536,547,520,240,033,680,384		

2ⁿ 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)