

# S O T N I R G

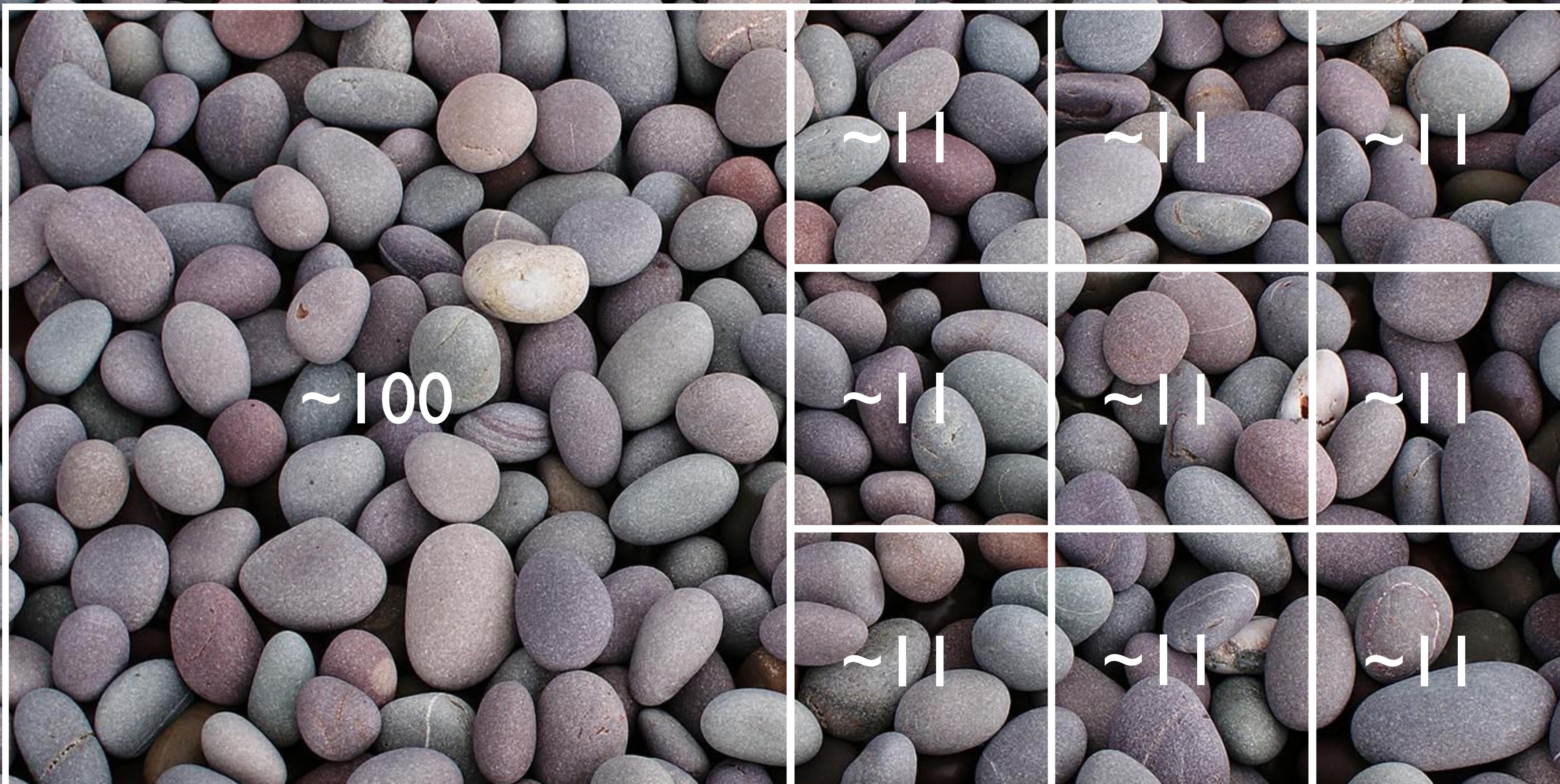


*Algorithms & Analysis*

# But first: how many pebbles?

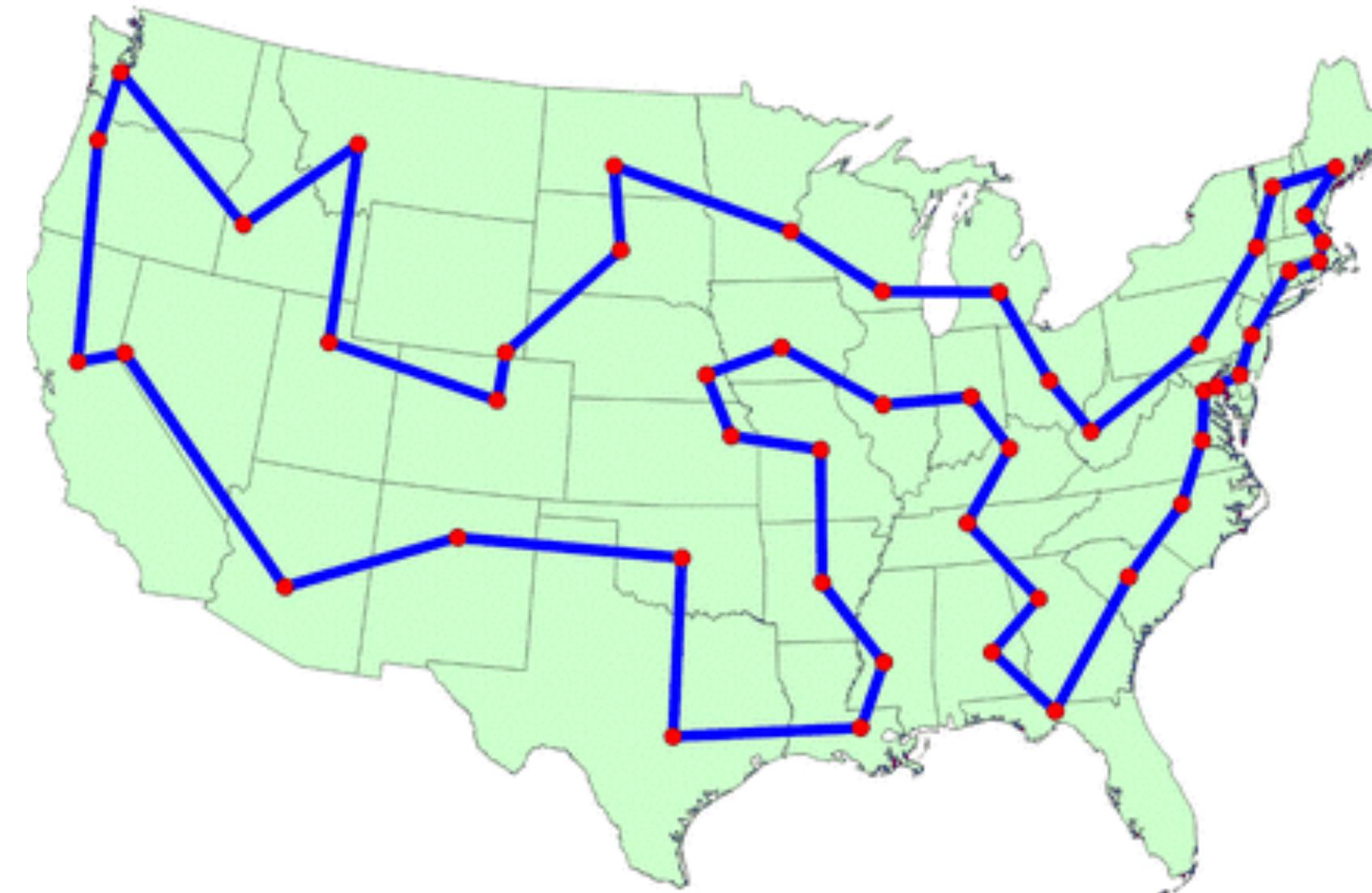


# HEURISTIC



# Heuristics

- Not necessarily correct (but gets you a "*good enough*" answer)
- Advantage: *fast* (often way faster than an algorithm)
- Famous example: the Traveling Salesman Problem



# Traveling Salesman Problem

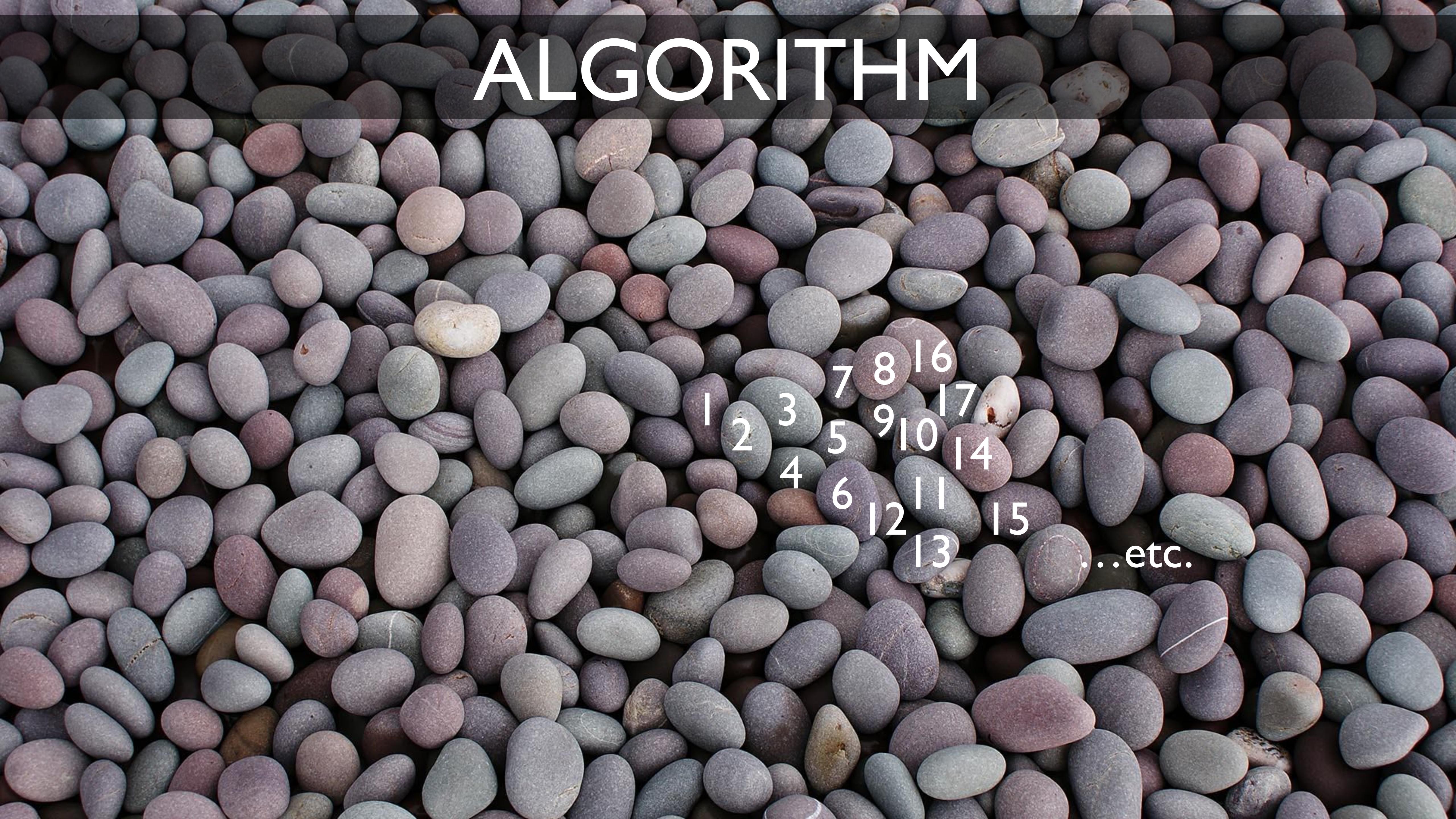
- Given **N** cities with a given **cost** of traveling between each pair, what is the **cheapest way** to travel to all of them?

Arriving

	NYC	SF	CHICAGO
NYC	NA	\$250	\$120
SF	\$210	NA	\$150
CHICAGO	\$100	\$115	NA

<b>NYC → SF → CHI</b>	<b>\$400</b>
<b>NYC → CHI → SF</b>	<b>\$235</b>
<b>SF → NYC → CHI</b>	<b>\$330</b>
<b>SF → CHI → NYC</b>	<b>\$250</b>
<b>CHI → NYC → SF</b>	<b>\$350</b>
<b>CHI → SF → NYC</b>	<b>\$325</b>

# ALGORITHM

A close-up photograph of a large pile of dark, smooth, rounded stones, likely beach pebbles, in shades of grey, black, and brown. They are scattered across the frame, creating a textured background.

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ...etc.

# Algorithms

- **Step-by-step instructions (deterministic)**
- **Complete** (gets you an answer)
- **Finite** (...given enough time)
- **Efficient** (doesn't waste time getting you the correct answer)
- **Correct** (the answer isn't just close, it is true)
- Downside: some problems are very **hard / slow**

*Often we loosely call functions algorithms, because much of the time a function is implementing an algorithm.*

# How can we compare algorithms?

# THE BIG



# In Plain English

**Big O: an abstract measure of how many steps a function takes **relative to its input**, as that input gets **arbitrarily large** (i.e. approaches Infinity)**

# Examples

- Example A: <https://repl.it/la7L/l>
- Example B: <https://repl.it/la7g/2>
- Example C: <https://repl.it/lbMY/2>
- Example D: <https://repl.it/la8L/l>
- Example E: <https://repl.it/laal/0>
- Big, Scary Graphing Calculator: <https://www.desmos.com/calculator>

# Quick review of logarithms

Logarithms are just the opposite of exponents

$$\log_2(n)$$

Read as: *what power do we need to raise 2 to in order to get n?*

$$\log_2(2) = 1$$

$$\log_2(4) = 2$$

$$\log_2(8) = 3$$

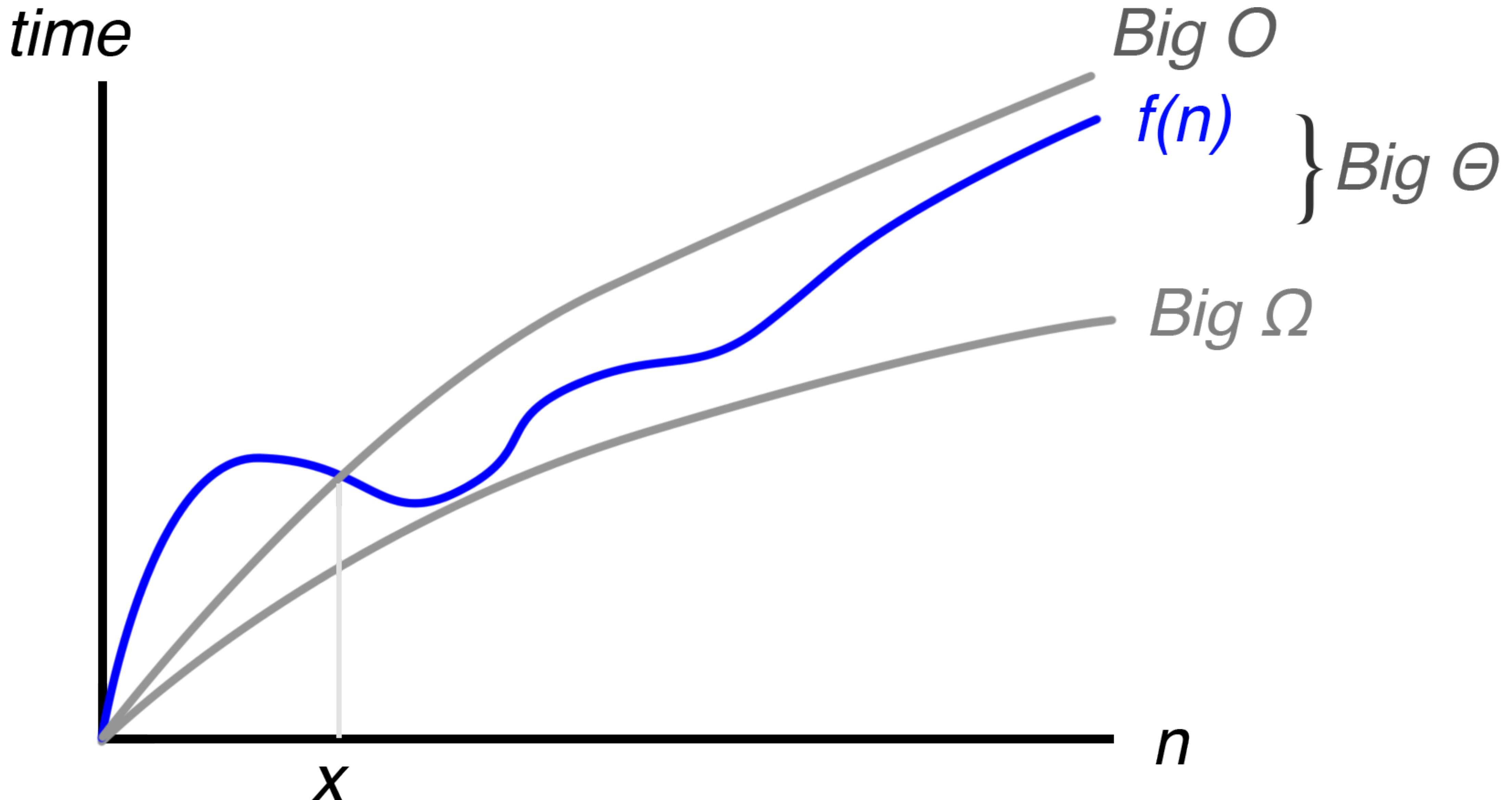
$$\log_2(5) = 2.32192809489$$

# Algorithm Analysis: Big O Notation

- A **comparative** way to classify different algorithms
- Based on **shape of growth curve** (*time vs input size(s)*)
- For **big enough** inputs
  - Might not be true when  $n$  is small, but who cares when  $n$  is small?
- Establishing an **upper bound** on the time
  - Not worse than this. Might be better, but it ain't worse!
- Including just the **highest order** term
  - In  $f(n) = n^3 + 5n + 3$ , only  $n^3$  matters as  $n$  gets large
- **Ignores constants** (mostly irrelevant;  $0.1 \cdot n^2$  will overtake  $10 \cdot n$ )



# What?



# Big O: comparative

- A very coarse, broad tool — big simplification
- Only useful when algorithms have *different* Big O notations
  - $O(n)$  will always beat  $O(n^2)$ , for *big enough n*
- If two algorithms have the same Big O, we don't know much.  
One might actually be quite slower than the other.

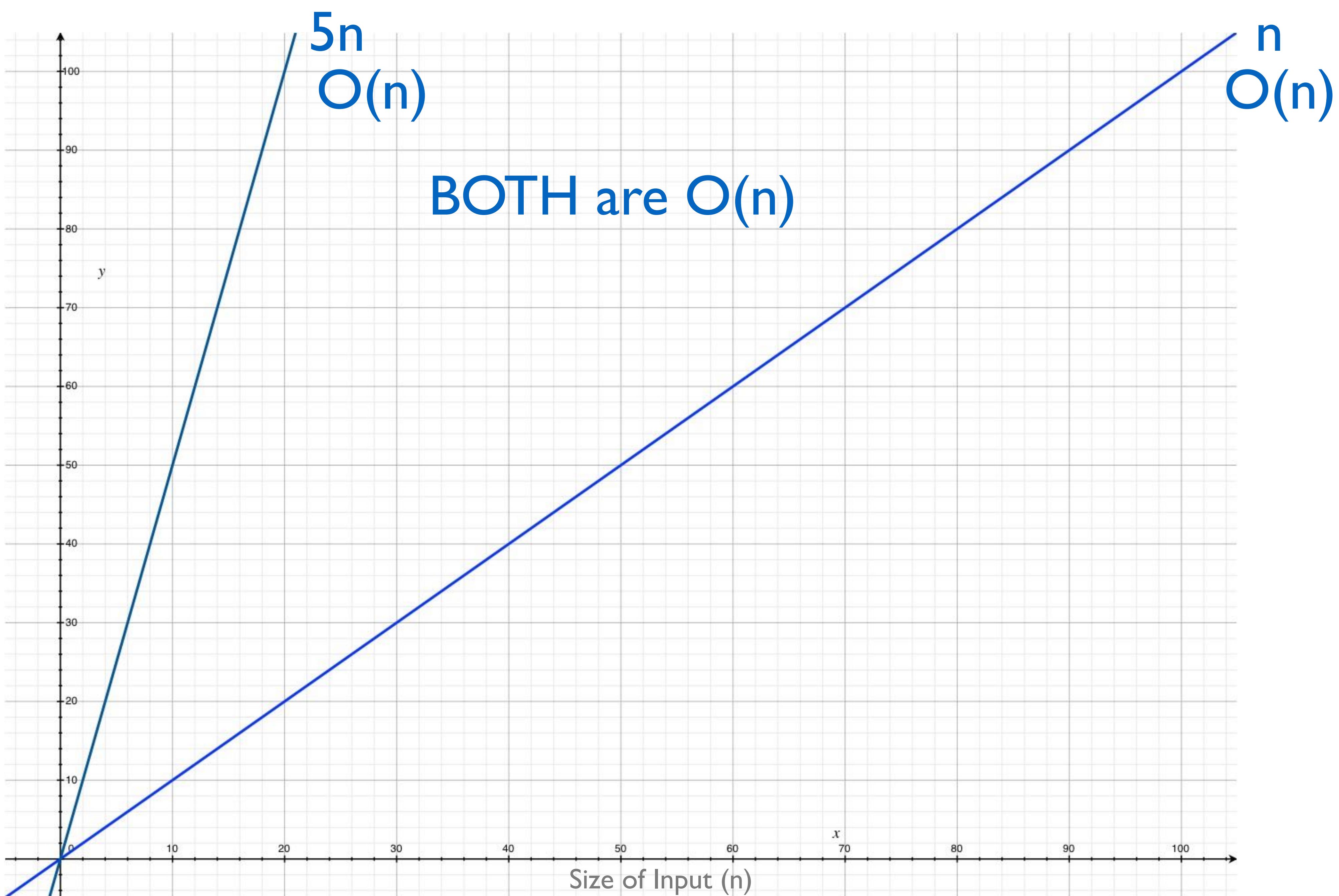


# Two Linear Functions

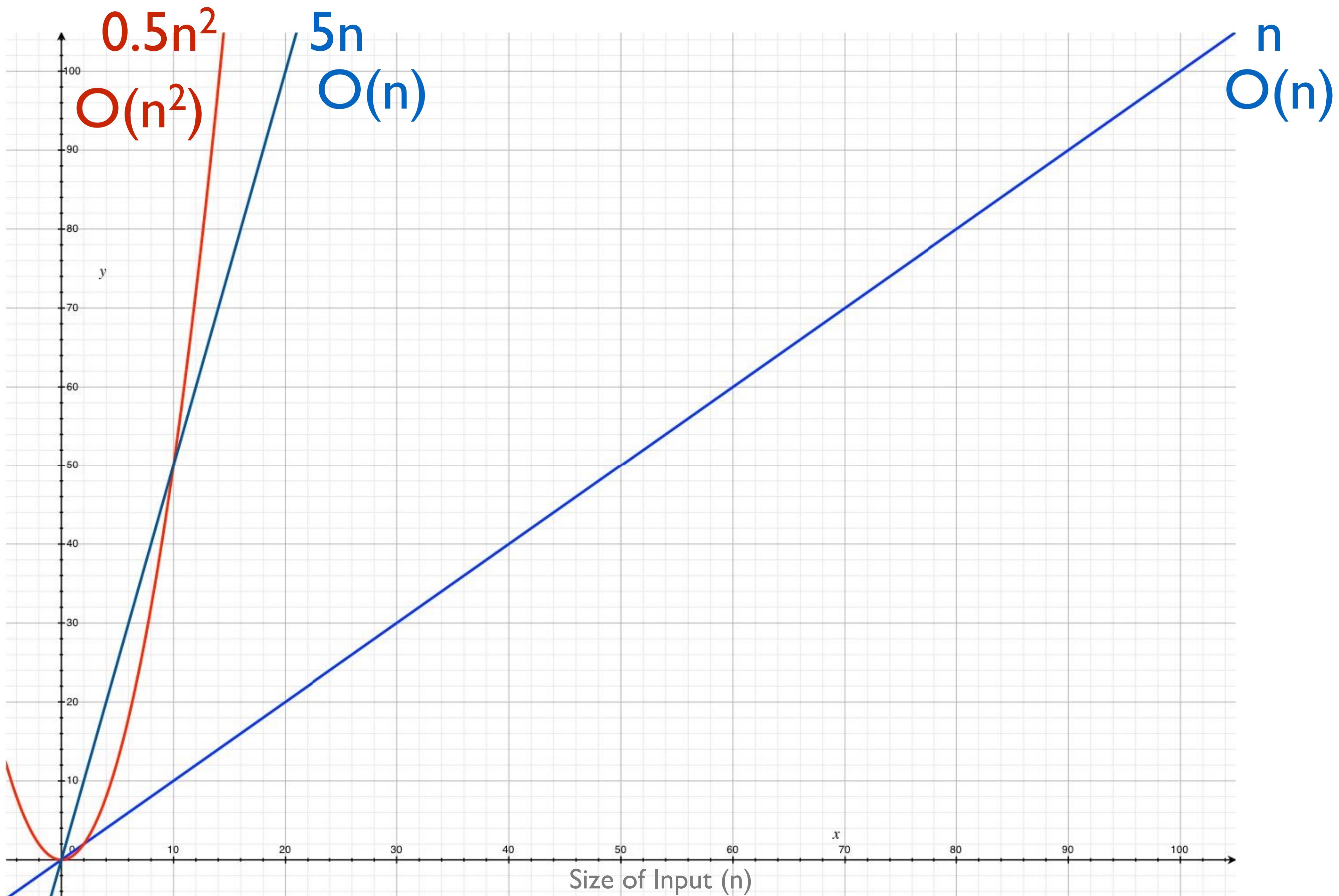
```
function findColors (arr) {  
  var colors = {  
    red: true,  
    orange: true,  
    yellow: true,  
    green: true,  
    blue: true  
  };  
  arr.forEach(function (val, i) {  
    if (colors[val]) console.log(i, val);  
  });  
}
```

```
function findColorsSlow (arr) {  
  arr.forEach(function (val, i) {  
    if (val === 'red') console.log(i, val);  
  });  
  arr.forEach(function (val, i) {  
    if (val === 'orange') console.log(i, val);  
  });  
  arr.forEach(function (val, i) {  
    if (val === 'yellow') console.log(i, val);  
  });  
  arr.forEach(function (val, i) {  
    if (val === 'green') console.log(i, val);  
  });  
  arr.forEach(function (val, i) {  
    if (val === 'blue') console.log(i, val);  
  });  
}
```

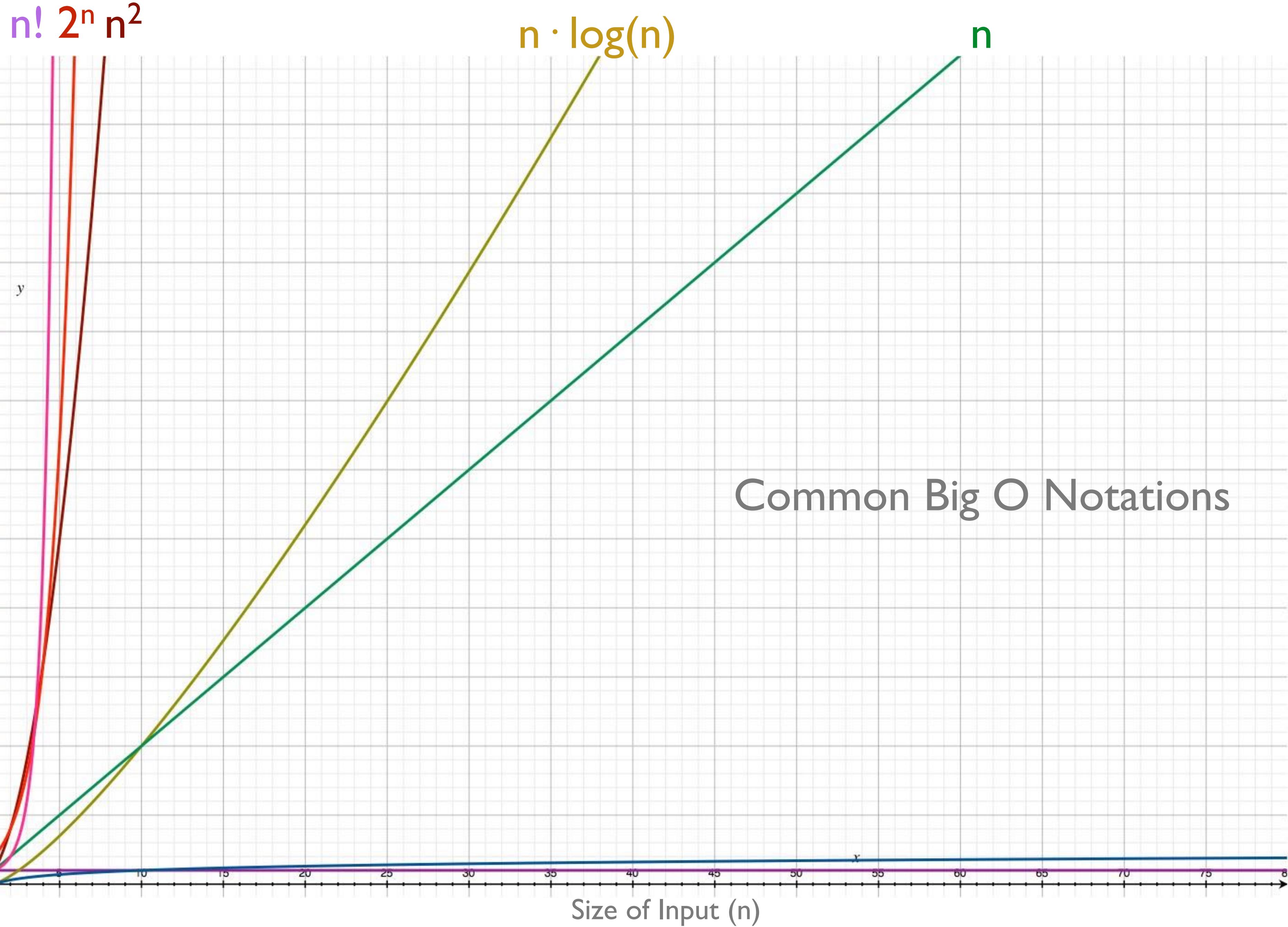
Time for  
Function  
to Complete



Time for  
Function  
to Complete



Time for  
Function  
to Complete



$\log(n)$



\*Source: Skiena, The Algorithm Design Manual

# Time Complexities (if 1 op = 1 ns)

input size n	$\log n$	n	$n \cdot \log n$	$n^2$	$2^n$	$n!$
10	0.003 µs	0.01 µs	0.03 µs	0.1 µs	1 µs	3.63 ms
20	0.004 µs	0.02 µs	0.09 µs	0.4 µs	1 ms	77.1 years
30	0.005 µs	0.03 µs	0.15 µs	0.9 µs	1 sec	8.4 × 10 <sup>15</sup> yrs
40	0.005 µs	0.04 µs	0.21 µs	1.6 µs	18.3 min	
50	0.006 µs	0.05 µs	0.28 µs	2.5 µs	13 days	
100	0.007 µs	0.10 µs	0.64 µs	10.0 µs	4 × 10 <sup>13</sup> yrs	
1 000	0.010 µs	1.00 µs	9.97 µs	1 ms		
10 000	0.013 µs	10.00 µs	~130.00 µs	100 ms		
100 000	0.017 µs	100.00 µs	1.7 ms	10 sec		
1 000 000	0.020 µs	1 ms	19.9 ms	16.7 min		
10 000 000	0.023 µs	10 ms	230.0 ms	1.16 days		
100 000 000	0.027 µs	100 ms	2.66 sec	115.7 days		
1 000 000 000	0.030 µs	1 sec	29.90 sec	31.7 years		

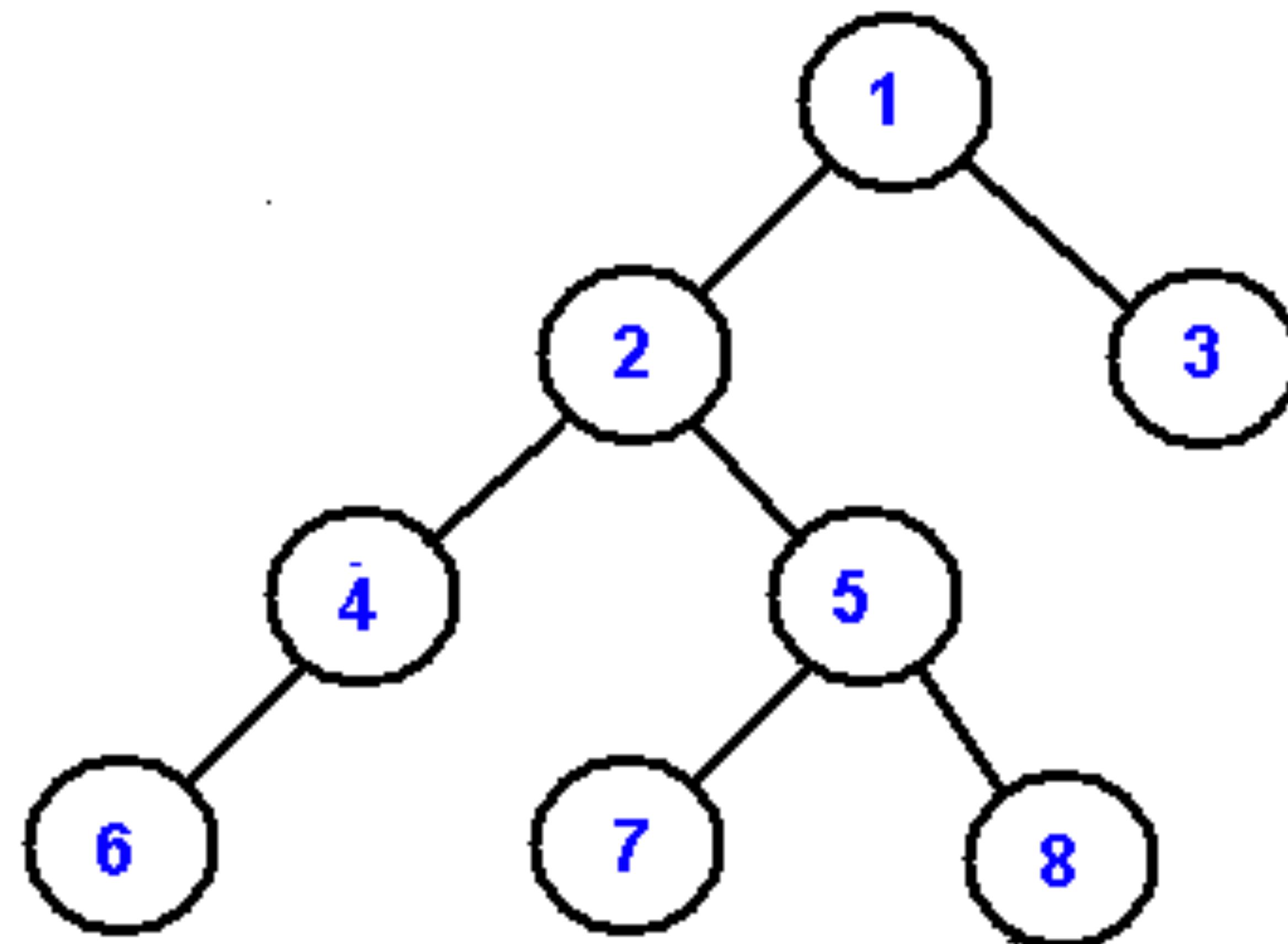


# Time Complexities

Big O	Name	Think	Example
$O(1)$	<i>Constant</i>	Doesn't depend on input	get array value by index
$O(\log n)$	<i>Logarithmic</i>	Using a tree	find min element of BST
$O(n)$	<i>Linear</i>	Checking (up to) all elements	search through linked list
$O(n \cdot \log n)$	<i>Loglinear</i>	tree levels * elements	merge sort average & worst case
$O(n^2)$	<i>Quadratic</i>	Checking pairs of elements	bubble sort average & worst case
$O(2^n)$	<i>Exponential</i>	Generating all subsets	brute-force n-long binary number
$O(n!)$	<i>Factorial</i>	Generating all permutations	the Traveling Salesman



Data Structure	Time Complexity								
	Average				Worst				
	Access	Search	Insertion	Deletion		Access	Search	Insertion	Deletion
Array	0(1)	0(n)	0(n)	0(n)	0(1)	0(n)	0(n)	0(n)	0(n)
Stack	0(n)	0(n)	0(1)	0(1)	0(n)	0(n)	0(1)	0(1)	0(1)
Singly-Linked List	0(n)	0(n)	0(1)	0(1)	0(n)	0(n)	0(1)	0(1)	0(1)
Doubly-Linked List	0(n)	0(n)	0(1)	0(1)	0(n)	0(n)	0(1)	0(1)	0(1)
Skip List	0(log(n))	0(log(n))	0(log(n))	0(log(n))	0(n)	0(n)	0(n)	0(n)	0(n)
Hash Table	-	0(1)	0(1)	0(1)	-	0(n)	0(n)	0(n)	0(n)
Binary Search Tree	0(log(n))	0(log(n))	0(log(n))	0(log(n))	0(n)	0(n)	0(n)	0(n)	0(n)



# Sorting (really this time)

*“By understanding sorting, we obtain an amazing amount of power to solve other problems.”*

– STEVEN SKIENA, THE ALGORITHM DESIGN MANUAL

# (Some) Classic Sorting Algorithms

- **Bubble**
- **Selection**
- **Insertion**
- **Merge: 1945 Jon von Neumann**
- **Quick: 1959 Tony Hoare**
- **Heap: 1964 J. W. J. Williams**
- **Radix: 1887 Hermann Hollerith, for his Tabulating Machine**
- **Bogo?**

# Bubble Sort

6 5 3 1 8 7 2 4

# Bubble Sort

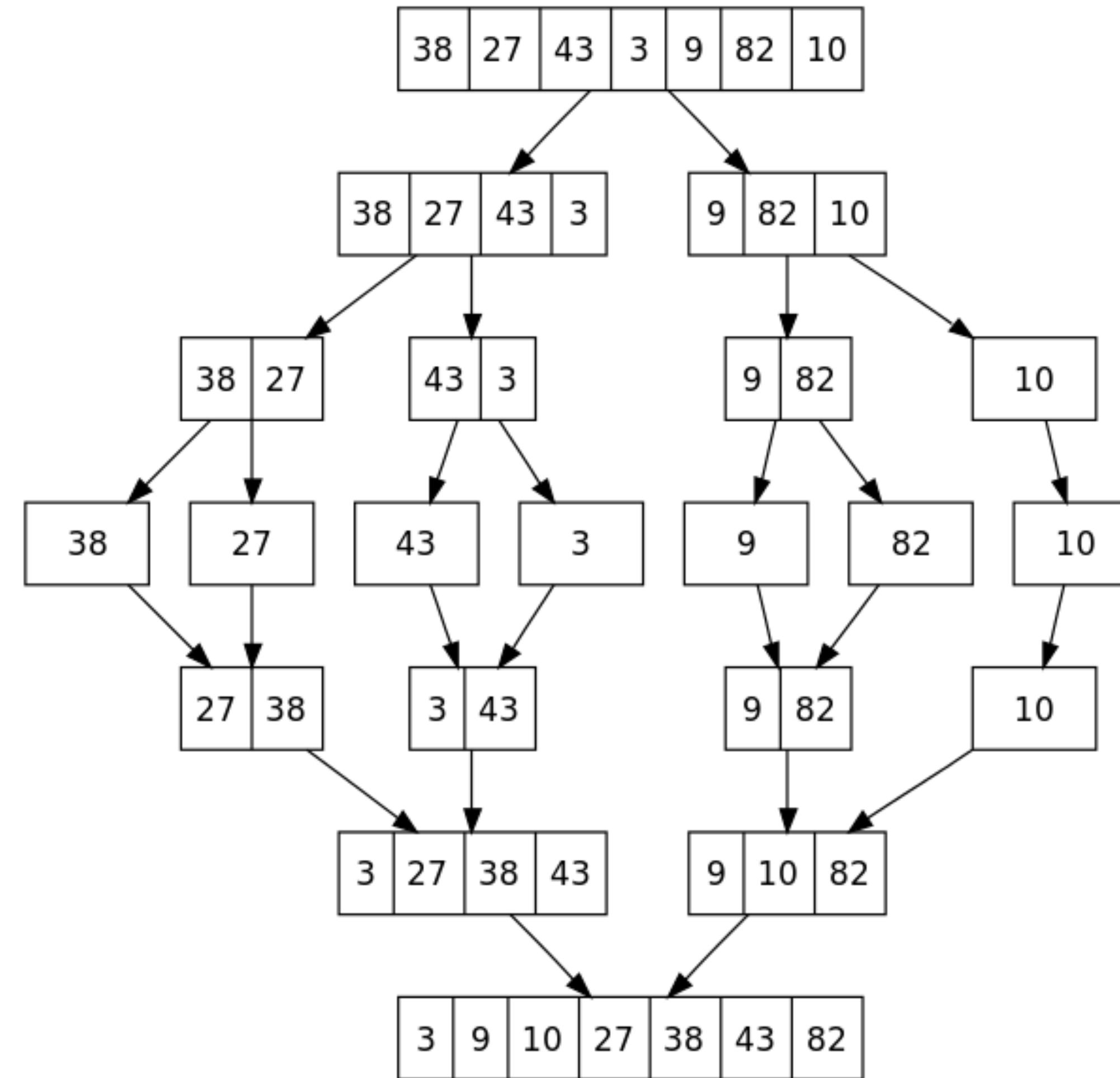
- 1. Loop over elements**
- 2. Swap anything that's out of order**
- 3. Repeat 1-2 until there are no swaps**

[https://www.youtube.com/watch?v=k4RRi\\_ntQc8](https://www.youtube.com/watch?v=k4RRi_ntQc8)

# Merge Sort

6 5 3 1 8 7 2 4

# Merge Sort



# Merge Sort (iterative)

1. Divide array of  $n$  elements into  $n$  arrays of 1 element
2. Merge neighboring arrays in sorted order
3. Repeat 2 until there's only one array

# Merge Sort (recursive)

1. If array is one element, good job it's sorted!
2. Otherwise, split the array and merge sort each half
3. Merge combined halves into sorted whole

# Big O

	Bubble Sort	Merge Sort
Time	$O(n^2)$	$O(n \cdot \log n)$
Space	$O(1)$	$O(n)$

# Why is merge sort faster?



# Merge Sort Speedup

- Combining two lists that are each already sorted into one list that is sorted is a linear time operation
- There are  $\log_2(n)$  steps needed to go from  $n$  lists of one item each to one list of  $n$  items

# Stable vs. Unstable



# Stable Sorts Preserve Order of "Equal" Elements

name: Harry  
role: student

name: McGonagall  
role: professor

name: Hermione  
role: student

**Sort by role (stable):**

name: McGonagall  
role: professor

name: Harry  
role: student

name: Hermione  
role: student

**Harry and Hermione in original order**



# Unstable Sorts Might Not Preserve Order of "Equal" Els

name: Harry  
role: student

name: McGonagall  
role: professor

name: Hermione  
role: student

**Sort by role (unstable):**

name: McGonagall  
role: professor

name: Harry  
role: student

name: Hermione  
role: student

**OR**

name: McGonagall  
role: professor

name: Hermione  
role: student

name: Harry  
role: student

**Harry and Hermione in different order**

# Sorting Stability: (Some) Examples

**Stable**

**Unstable\***

**Bubble**

**Quick<sup>†</sup>**

**Merge**

**Heap**

**Insertion**

**Selection**

**Bucket**

**Shell**

\* Any sort can be made stable with  $O(n)$  extra space

† If implemented in a standard way

WHAT ABOUT JS ?

**ES `sort` is *not required* to be stable.**

**V8 `sort` is unstable.**

# In-Place Sorting

# In-Place Sorting

- An in-place algorithm uses only a *small, constant* amount of extra space ( $O(1)$  space complexity) to achieve its goal

```
function sumArray (arr) {  
    return arr.reduce(function (sum, el) { return sum + el; });  
}
```

- As a **consequence** (but not summary!) of this definition, in-place sorting algorithms **mutate the input array**
  - This is intuitive; any sort that doesn't mutate the array must copy it, and if it copies the array then it has minimum  $O(n)$  space complexity.

# Sorting Memory: (Some) Examples

In-Place ( $O(1)$ )

Not In-Place

Bubble

Merge:  $O(n)$

Heap

Quick:  $O(\log(n))$  |  $n$

Insertion

Tim:  $O(n)$

Shell

Cube:  $O(n)$

WHAT ABOUT JS ?

**ES *doesn't require* .sort to be in-place.**  
**But it *does require* it to mutate the array.**

**V8 .sort is *not* in-place.**  
**But it *does* mutate the array.**

*(Note: many programmers misuse "in-place" to mean "mutates the array")*

# JavaScript Native Sort Summary

- **ECMAScript**

- Must mutate input array
- Not required to be stable (though it is allowed)
- Not required to be in-place (though it is allowed)
- Takes an optional comparator function which returns negative, 0, or positive num

- **V8 (Node, Chrome — but not other browsers)**

- Hybrid approach
  - Insertion sort for very small arrays
  - Quicksort for larger arrays
- Unstable
- Not in-place (but does mutate array!)



# Bubble vs. Merge Sort, One More Time

	Bubble	Merge
Time Complexity	$O(n^2)$	$O(n \cdot \log(n))$
Space Complexity / In-Place	$O(1)$ / Yes	$O(n)$ / no
Stable	Yes	Yes

# Other Sorting Considerations

- Some sorts are far better or far worse when data is:
  - Random
  - Nearly sorted
  - Backwards
  - Duplicated
- Some sorts are significantly faster in the average case
  - Quicksort is  $O(n^2)$  worst-case, yet often preferred over merge sort ( $O(n \cdot \log(n))$ ) because it can be implemented with less memory and faster average (i.e. typical) time!
- [Click here for animations](#)

# Special Note

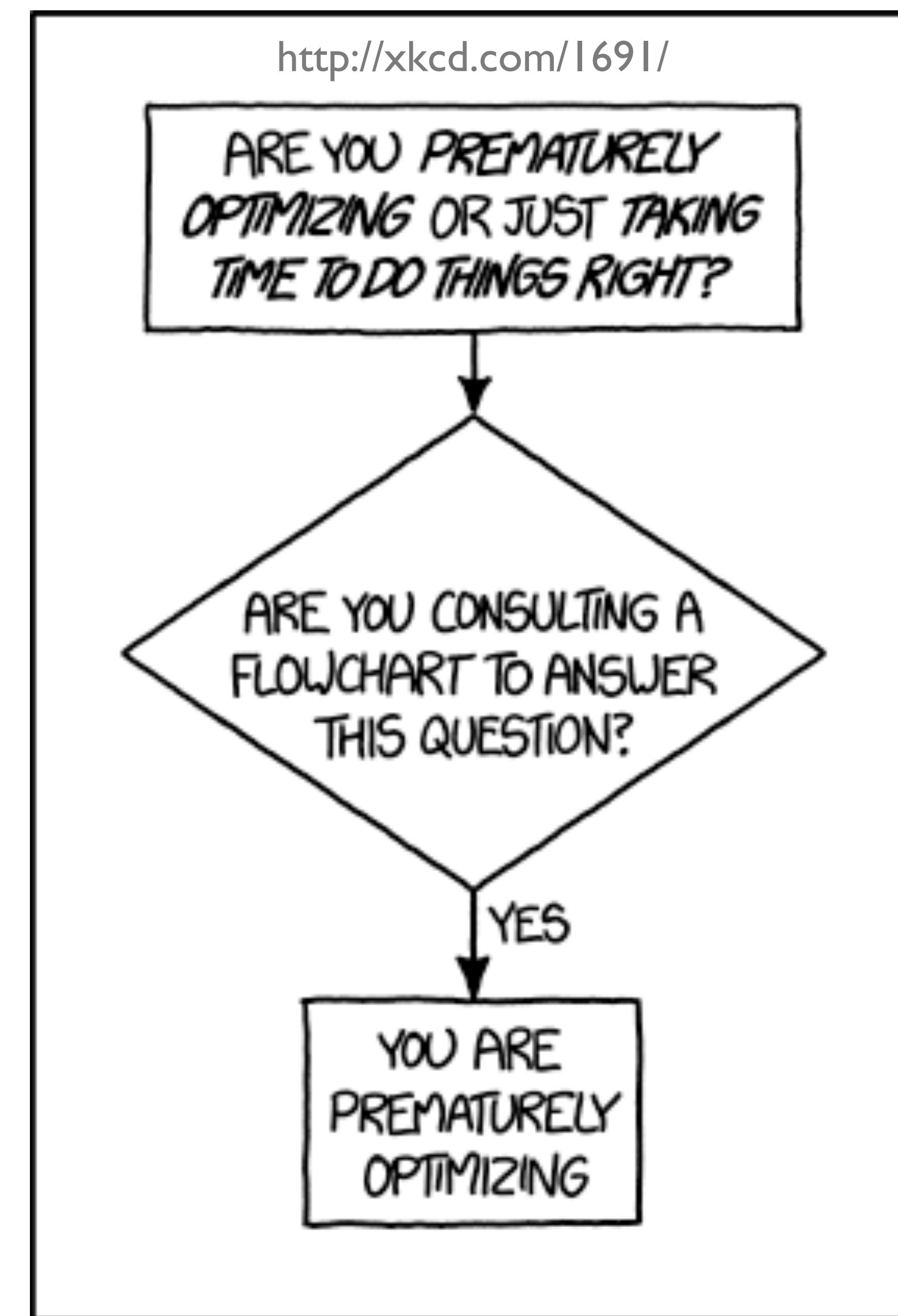


# Rob Pike's 5 Rules of Programming

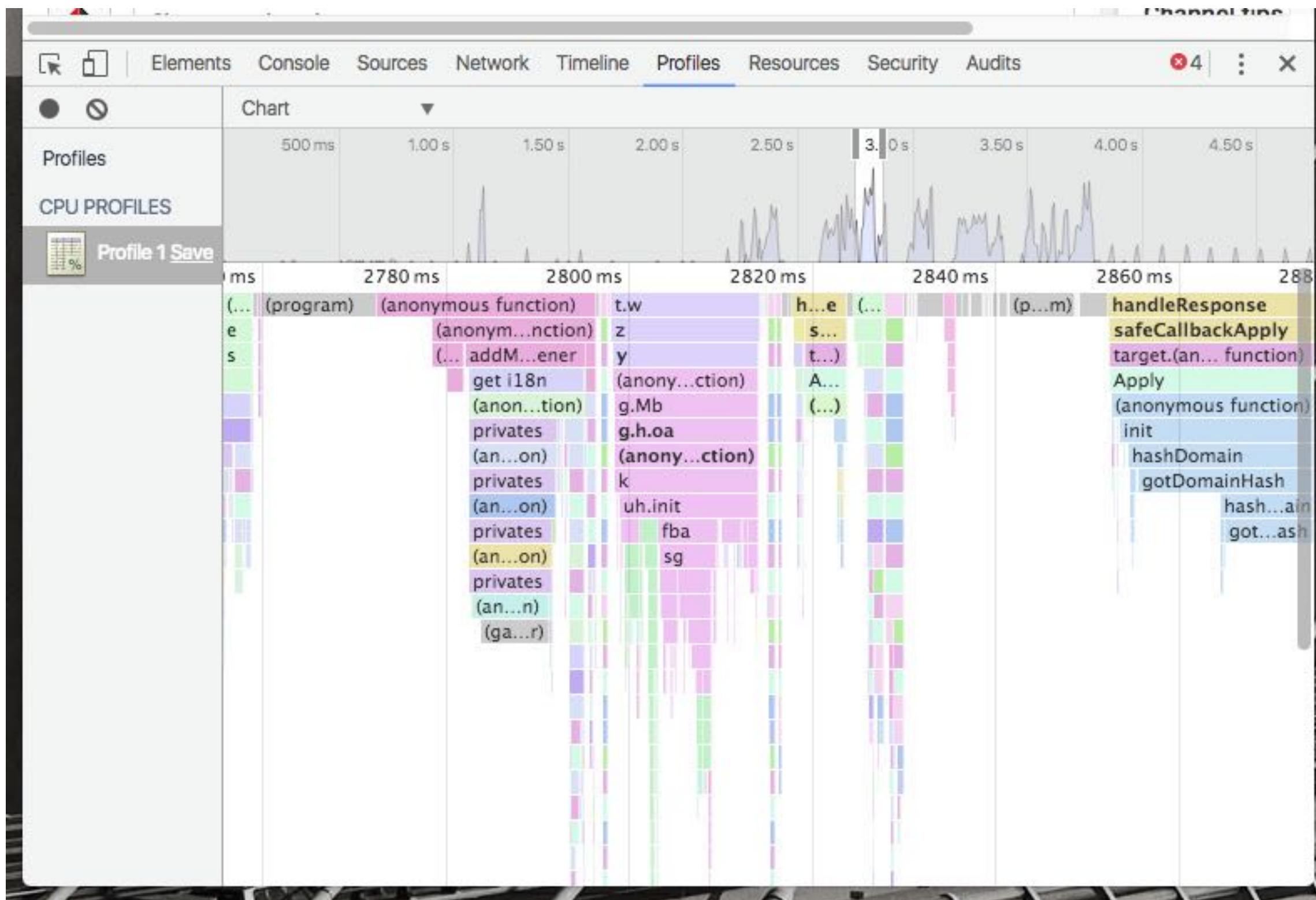
Bell Labs  
Unix Team  
UTF-8  
Go Language  
...and a lot more

# 1

- You can't tell where a program is going to spend its time. Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you've proven that's where the bottleneck is.



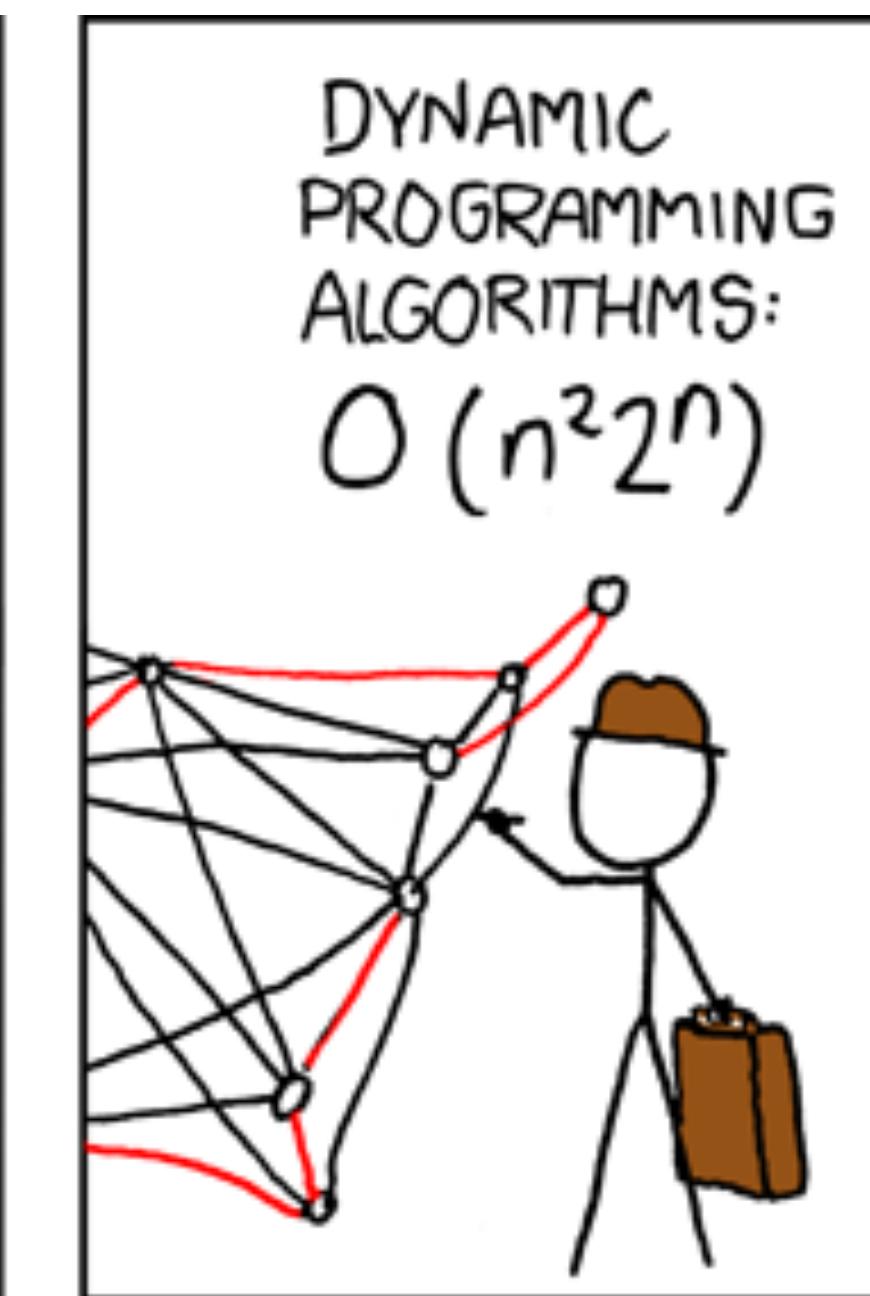
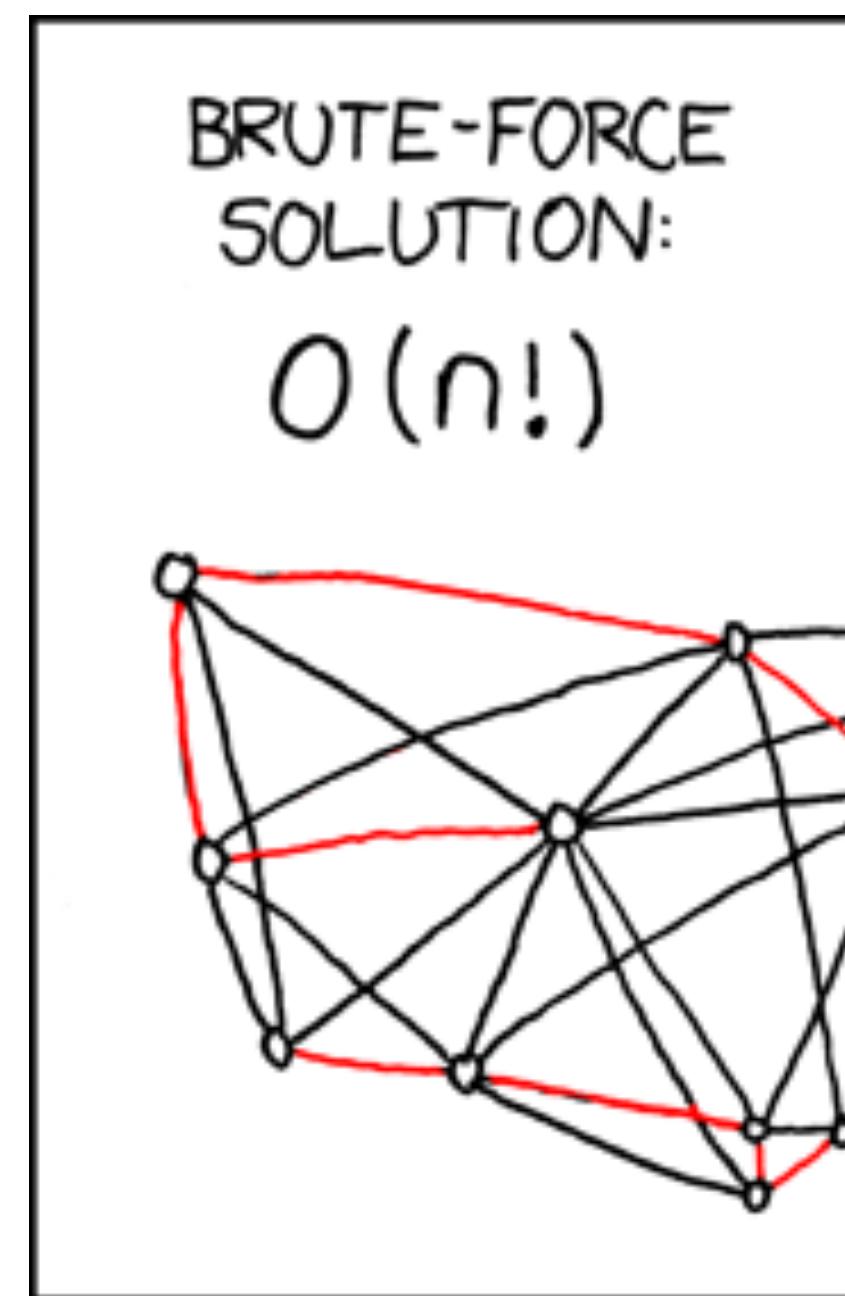
# 2



- Measure. Don't tune for speed until you've measured, and even then don't unless one part of the code overwhelms the rest.

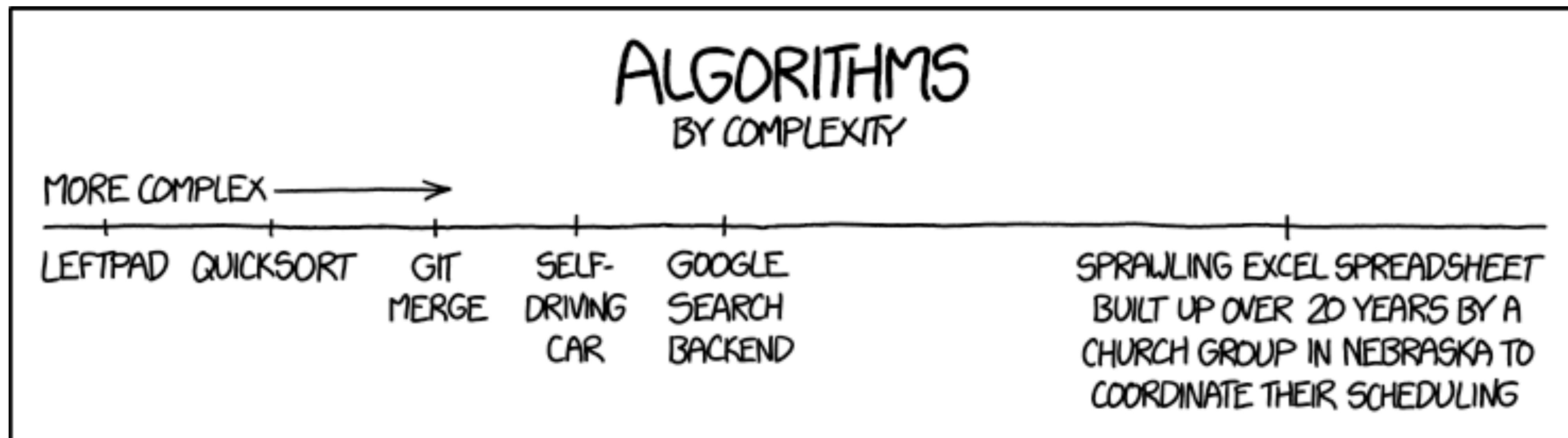
# 3

- Fancy algorithms are slow when  $n$  is small, and  $n$  is usually small. Fancy algorithms have big constants. Until you know that  $n$  is frequently going to be big, don't get fancy.



# 4

- Fancy algorithms are buggier than simple ones, and they're much harder to implement. Use simple algorithms as well as simple data structures.



# 5

- ◎ Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming.

# WORKSHOP

