S O T N I R G

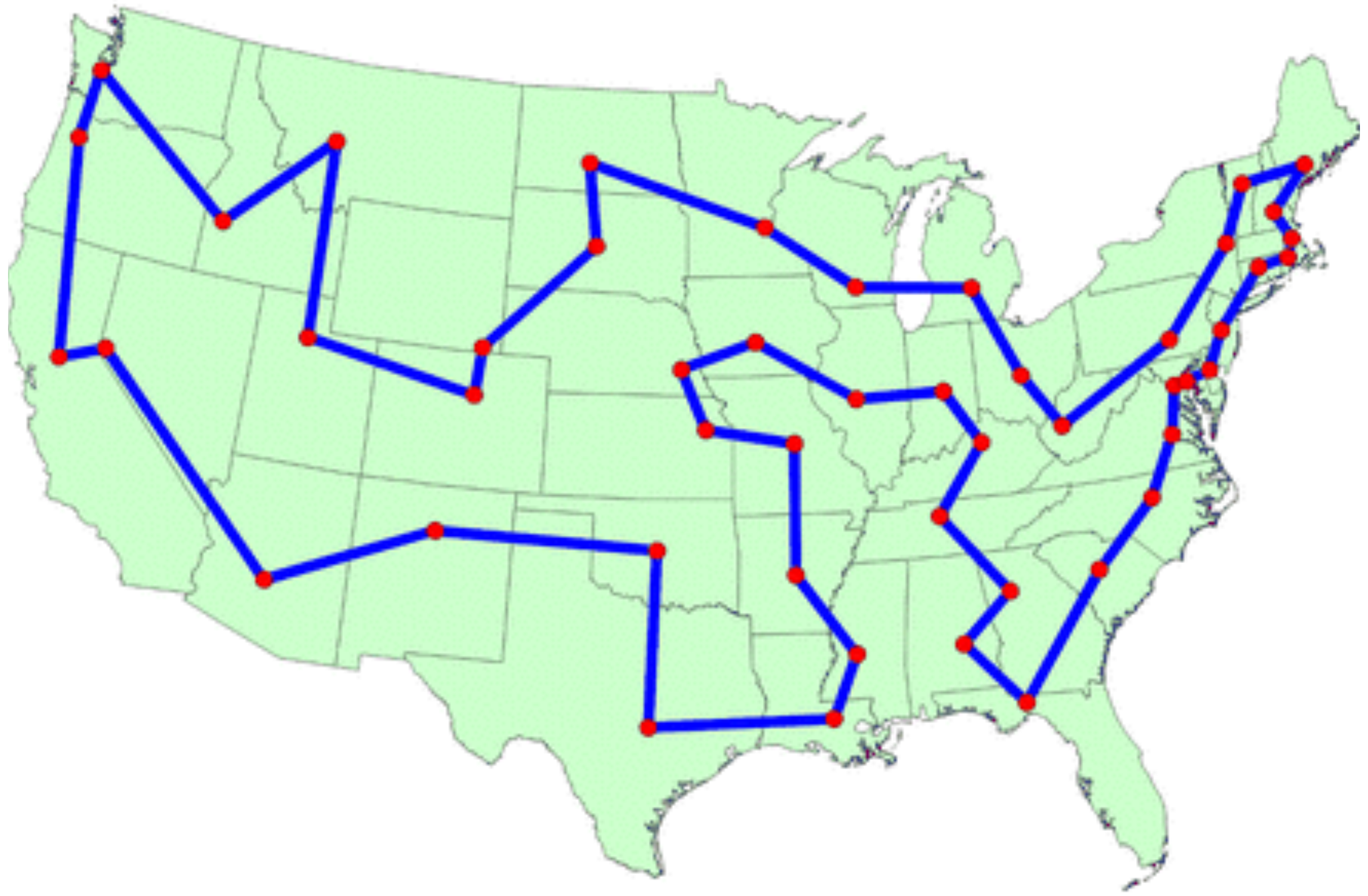
Algorithms & Analysis

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

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

NYC → SF → CHI	$400
NYC → CHI → SF	$235
SF→ NYC → CHI	$330
SF → CHI → NYC	$250
CHI → NYC → SF	$350
CHI → SF → NYC	$325

ALGORITHM

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

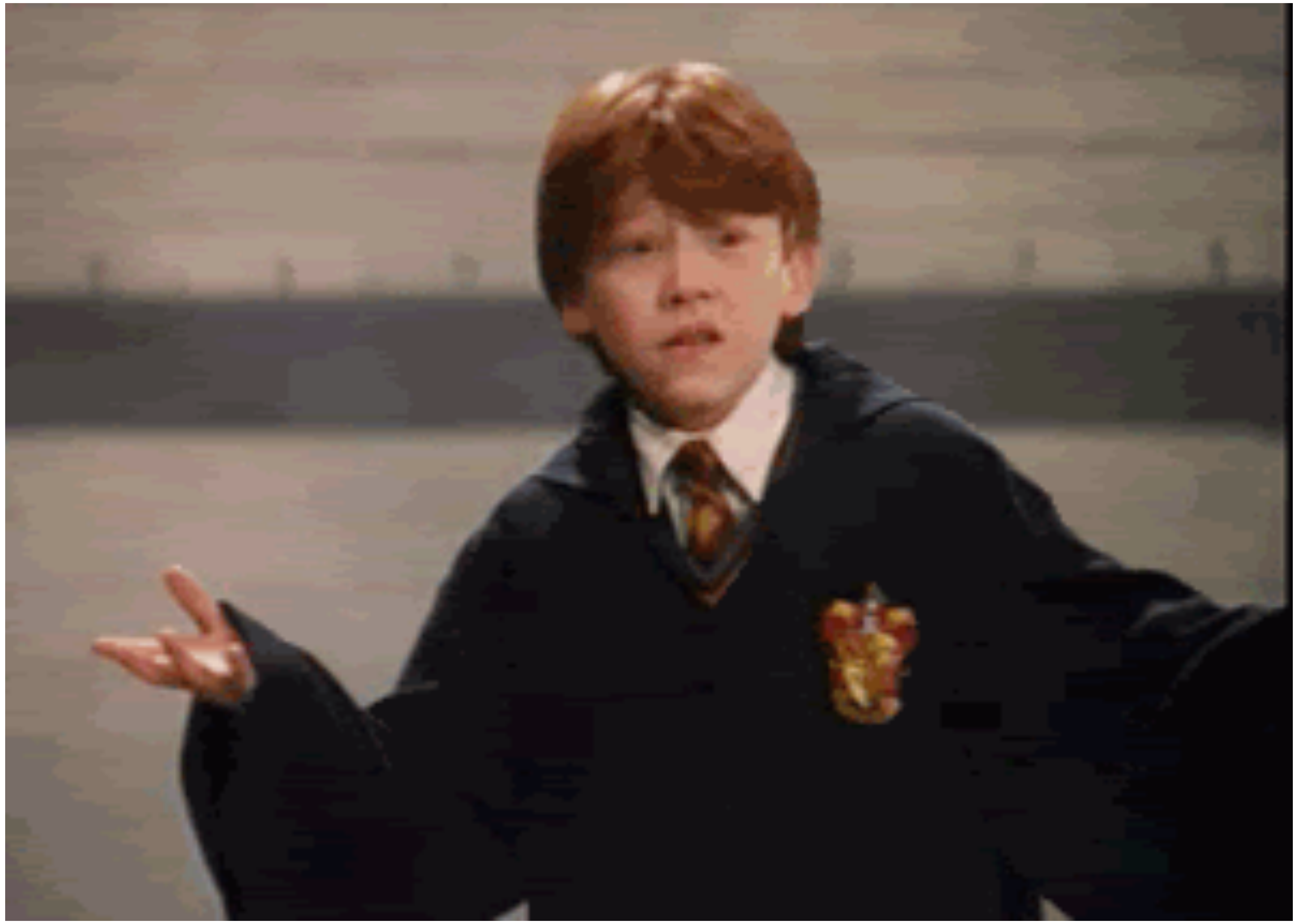
Often we loosely call functions algorithms, because much of the time a function is implementing an algorithm.

How can we compare algorithms?


THE BIG O

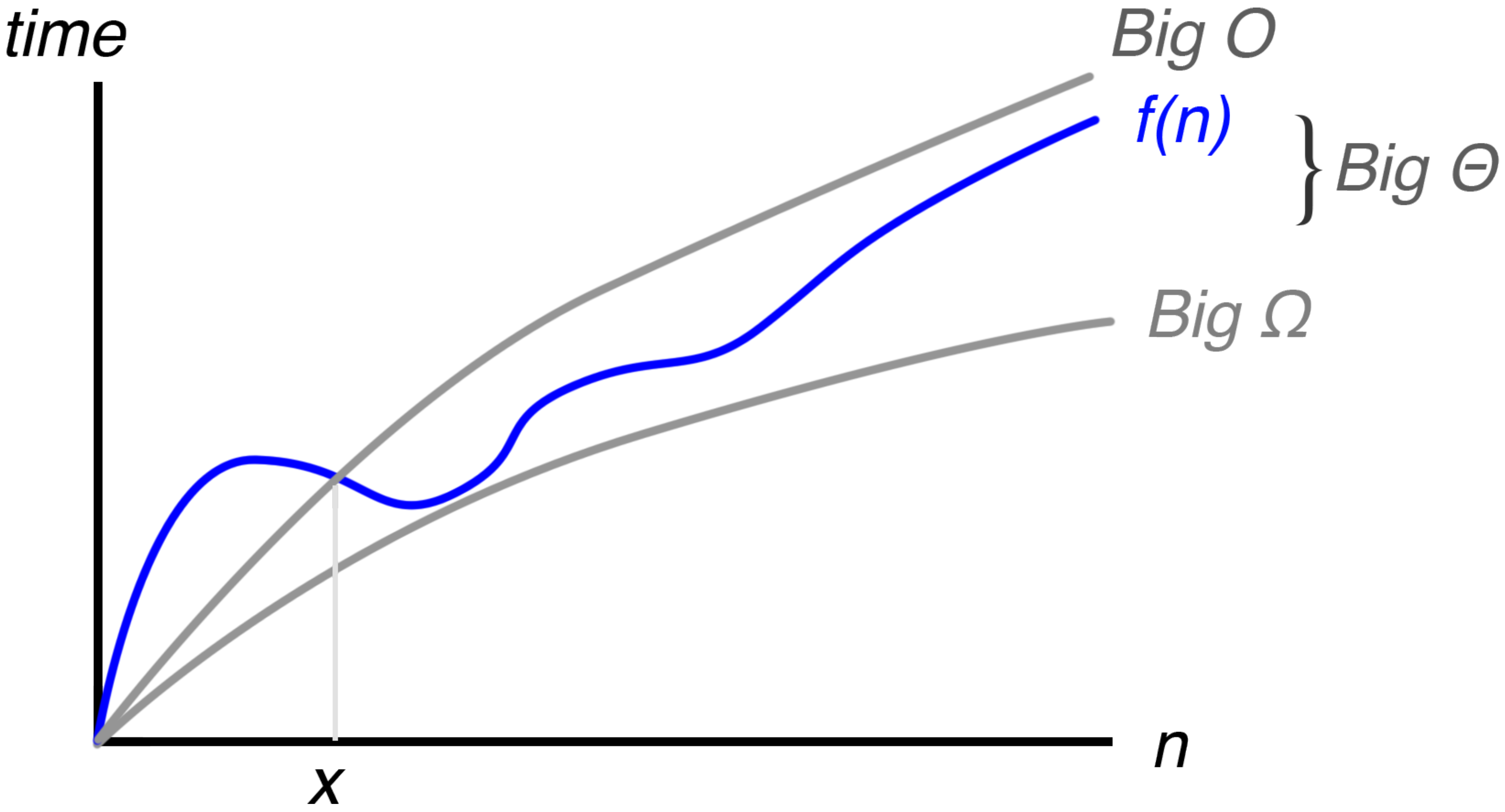
Algorithm Analysis: Big O Notation

- 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?

Adapted from The Algorithm Design Manual



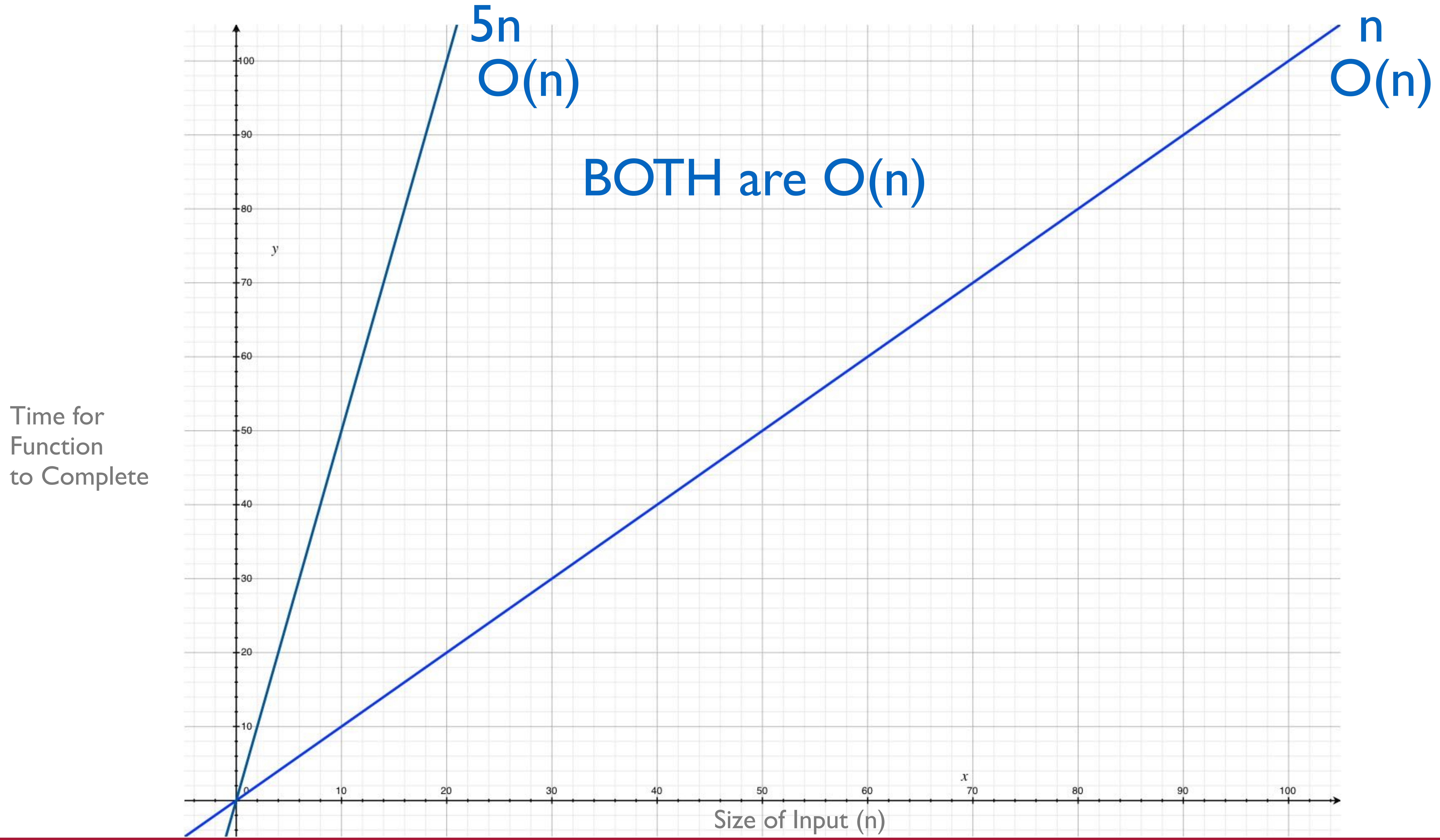
Big O: comparative

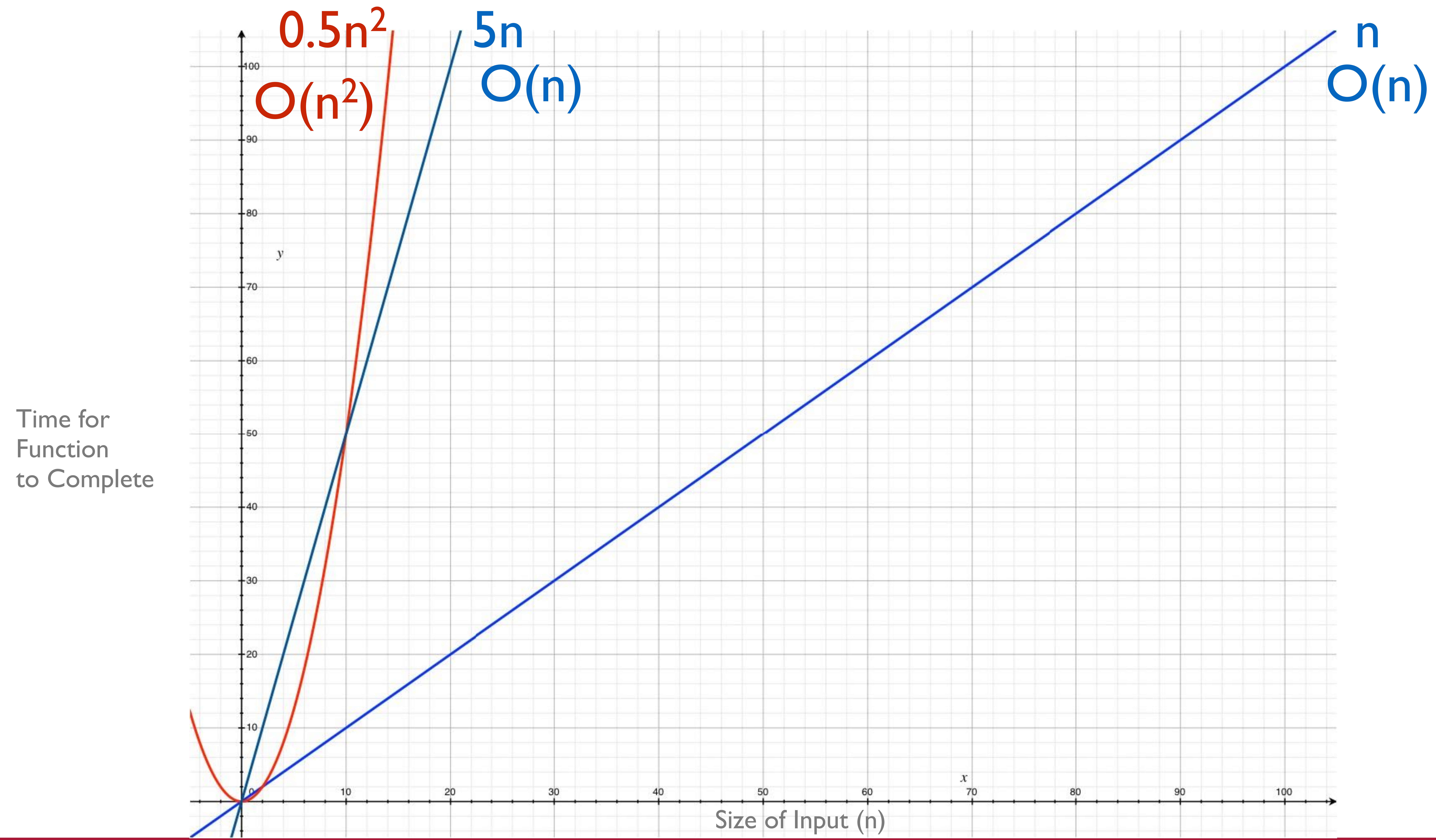
- A very coarse, broad tool — big simplification
- Only useful when algorithms have *different* Big O notations
  - $O(n)$ will always beat $O(n^2)$, *for big enough n*
- If two algorithms have the same Big O, we don't know much. One might actually be quite slower than the other.

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);
  });
}
```


5n
O(n)
n
O(n)
BOTH are O(n)
Time for
Function
to Complete
Size of Input (n)
y
x
100
90
80
70
60
50
40
30
20
10
0
10
20
30
40
50
60
70
80
90
100



$0.5n^2$
$O(n^2)$
5n
O(n)
n
O(n)
Time for
Function
to Complete
y
x
100
90
80
70
60
50
40
30
20
10
0
10
20
30
40
50
60
70
80
90
100
Size of Input (n)


n! 2^n n^2
n · log(n)
n
Time for Function to Complete
y
55
50
45
40
35
30
25
20
15
10
5
Common Big O Notations
log(n)
1
x
10
15
20
25
30
35
40
45
50
55
60
65
70
75
Size of Input (n)

*Source: Skiena, The Algorithm Design Manual

Time Complexities (if 1 op = 1 ns)

input size n	log n	n	n·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^{15} 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^{13} 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
$O(1)$	*Constant*	Doesn't depend on input	get array value by index
$O(\log n)$	*Logarithmic*	Using a tree	find min element of BST
$O(n)$	*Linear*	Checking (up to) all elements	search through linked list
$O(n \cdot \log n)$	*Loglinear*	A tree for each element	merge sort average & worst case
$O(n^2)$	*Quadratic*	Checking pairs of elements	bubble sort average & worst case
$O(2^n)$	*Exponential*	Generating all subsets	brute-force n-long binary number
$O(n!)$	*Factorial*	Generating all permutations	the Traveling Salesman

bigocheatsheet.com

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

"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

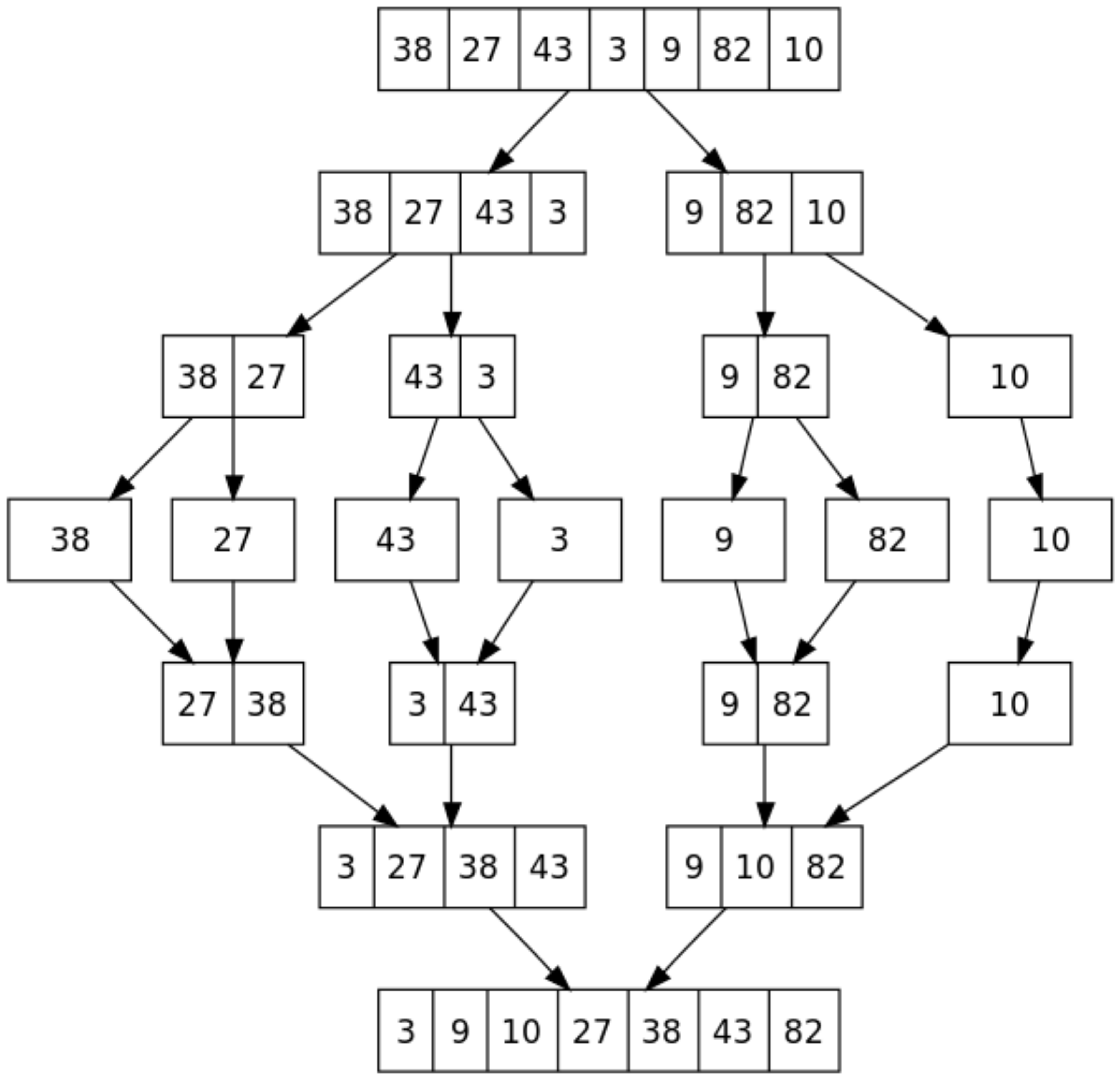
Bubble Sort

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

Merge Sort

6 5 3 1 8 7 2 4

Merge Sort



38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

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

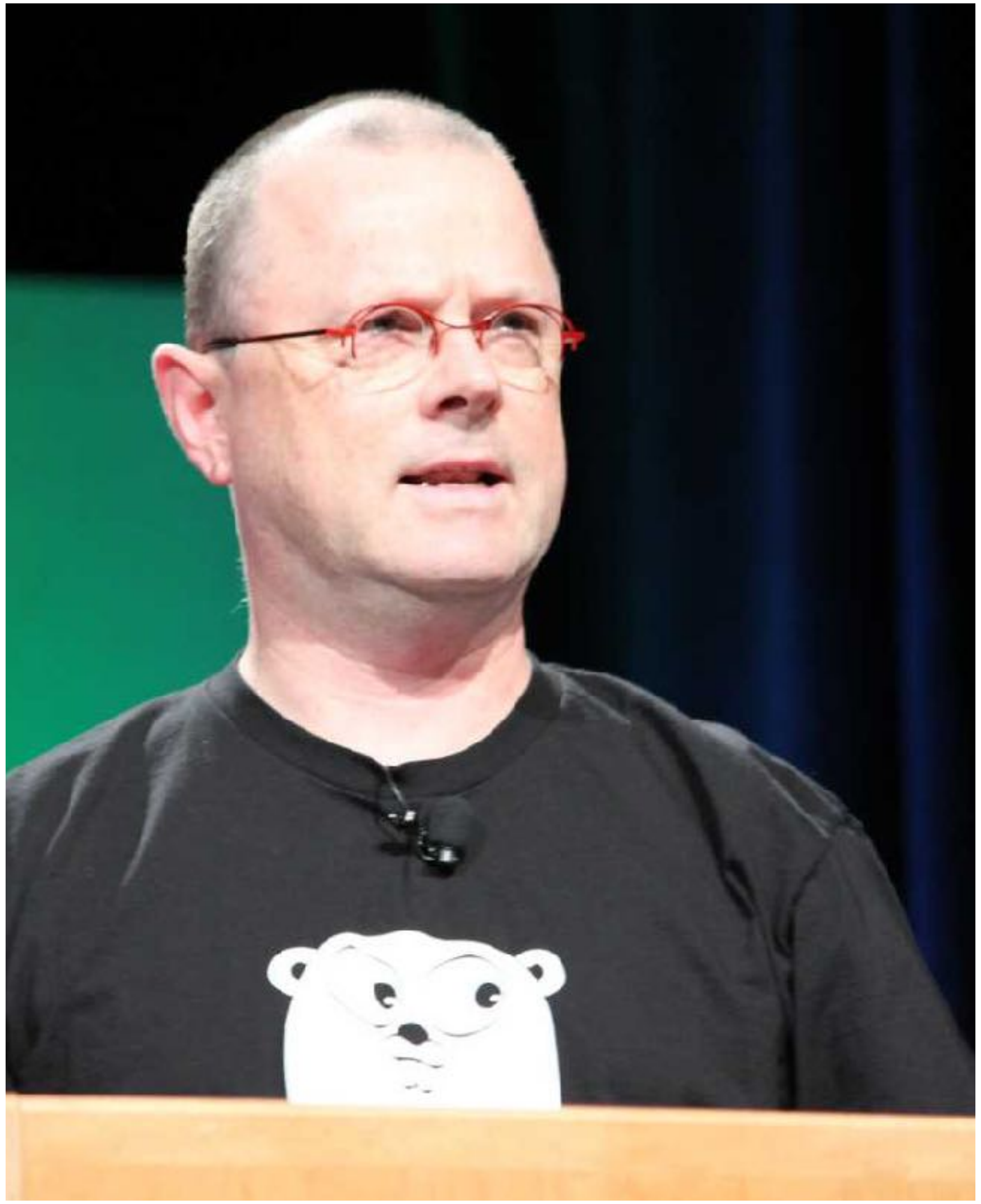
	Bubble Sort	Merge Sort
Time	$O(n^2)$	$O(n \cdot \log n)$
Space	$O(1)$	$O(n)$

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 `log₂(n)` steps needed to go from `n` lists of one item each to one list of `n` items

Special Note



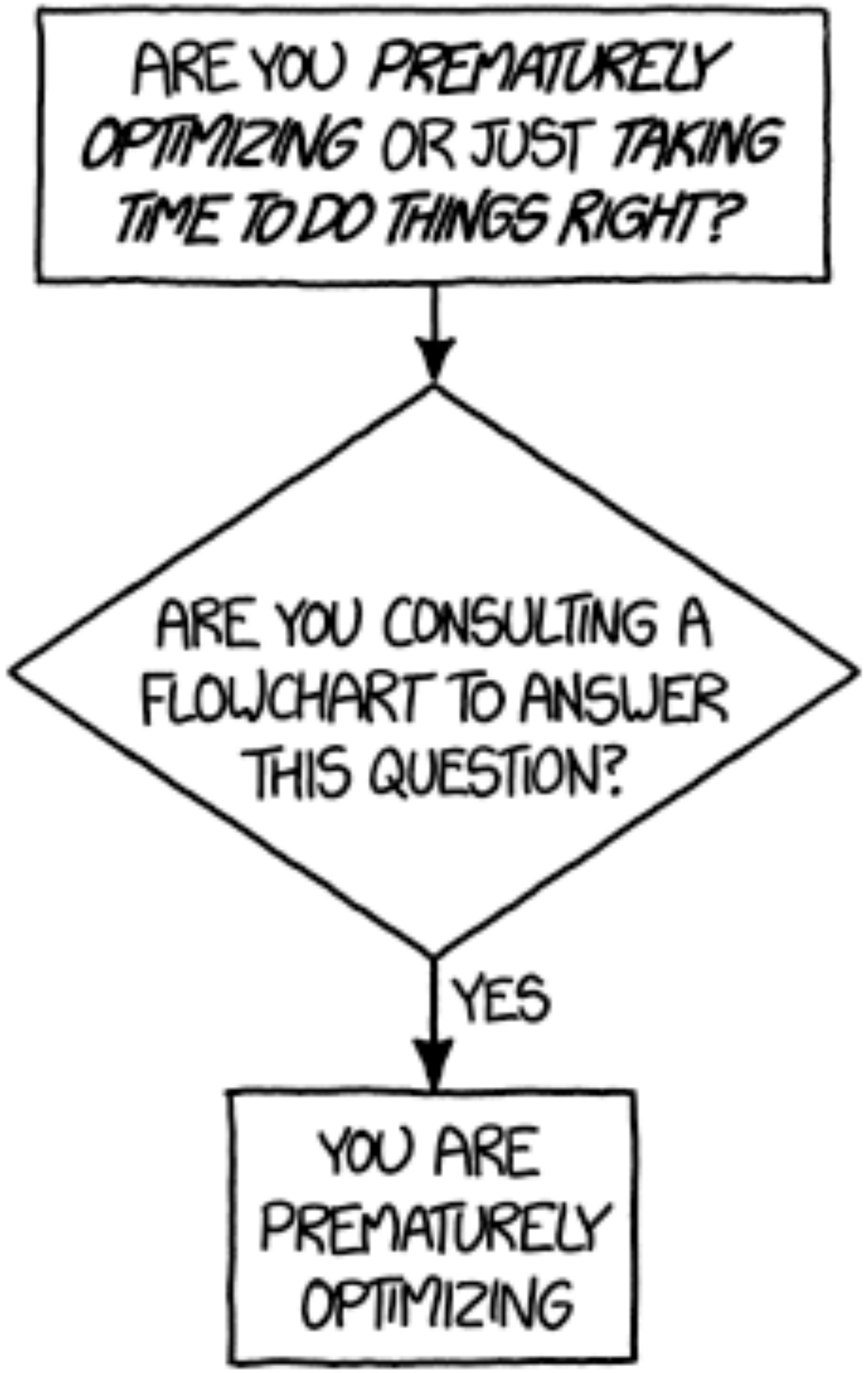
Rob Pike's 5 Rules of Programming

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

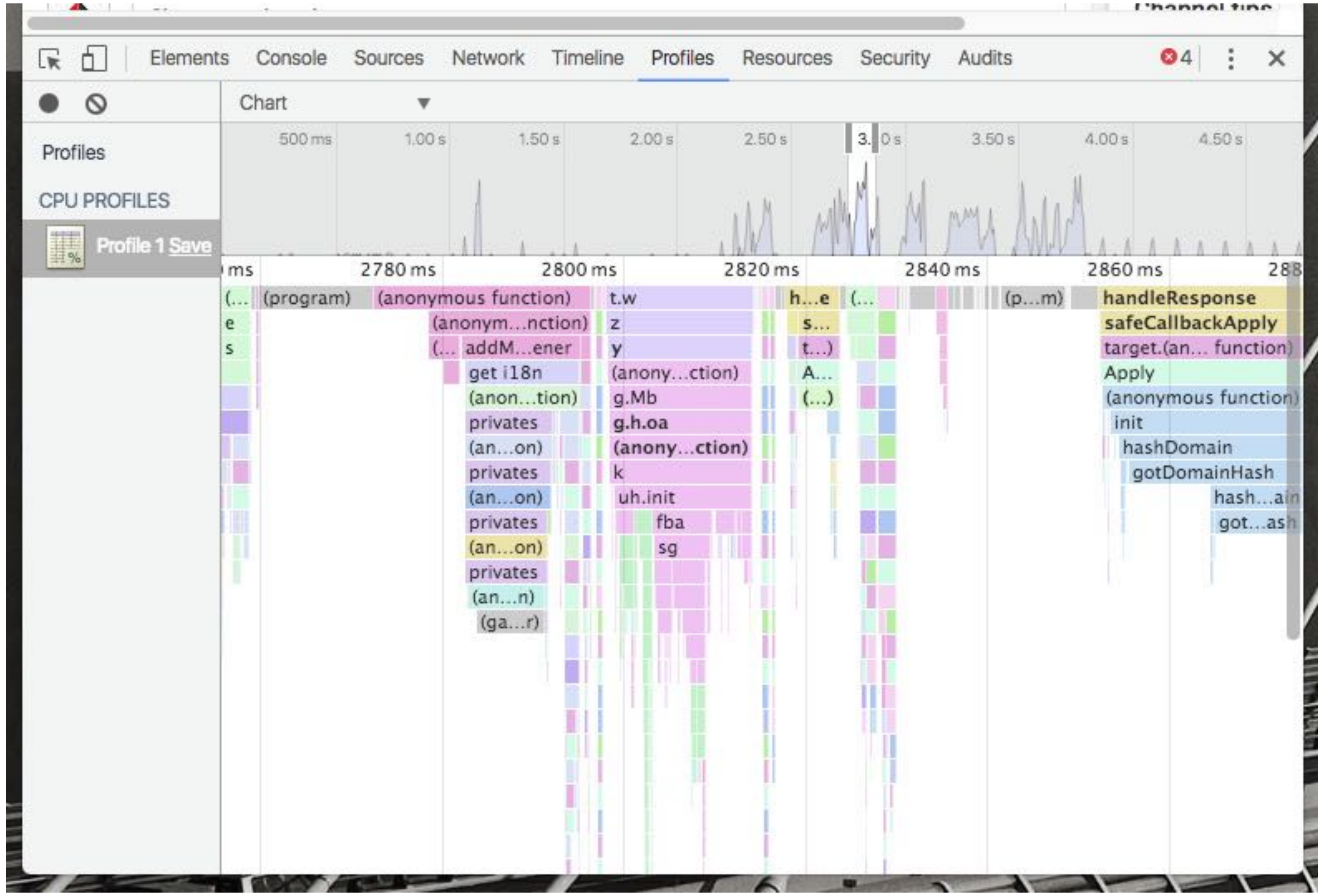
1

- 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.

http://xkcd.com/1691/



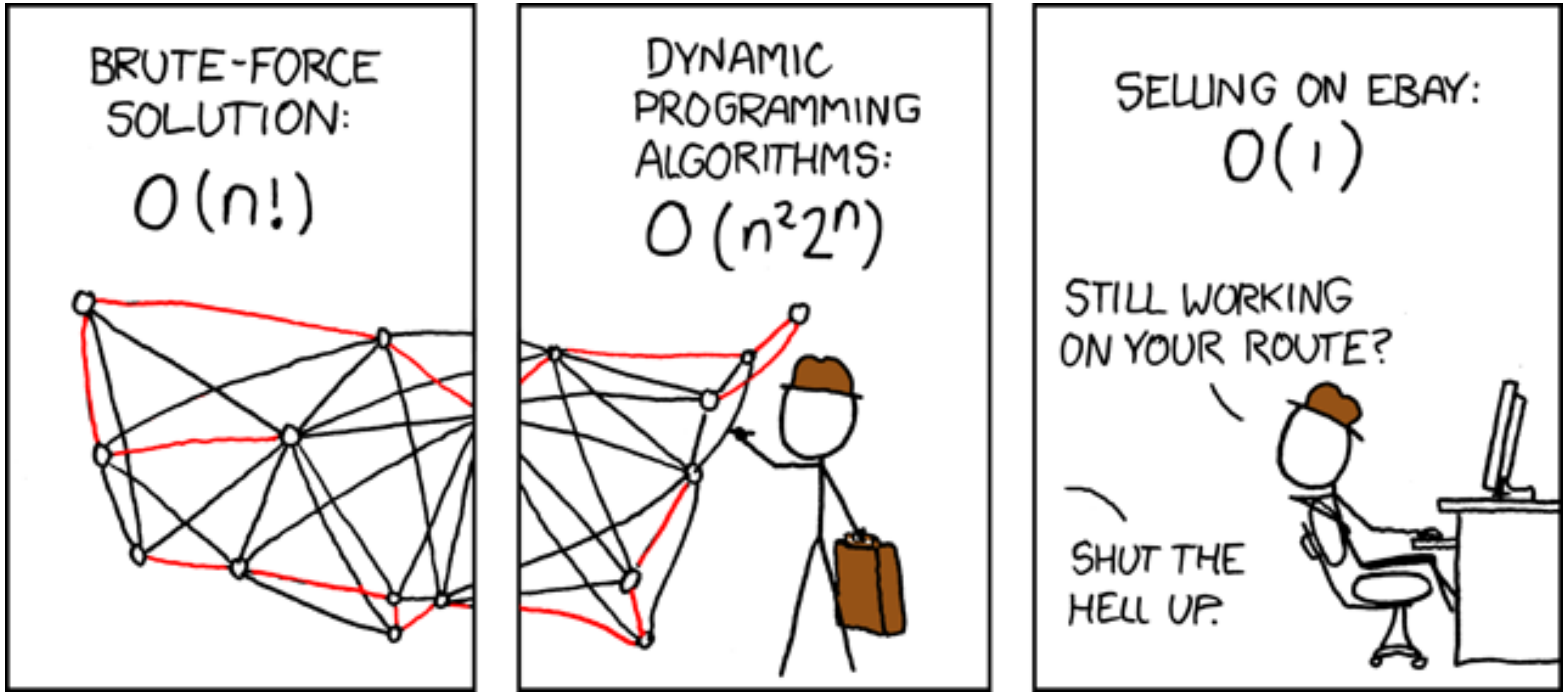
2



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

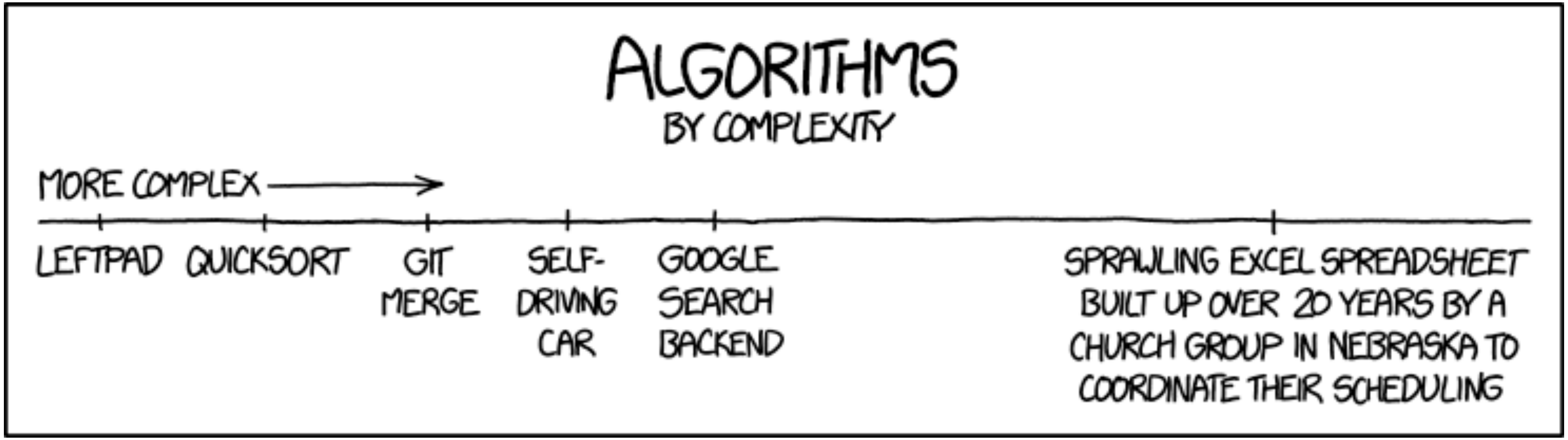
3

- 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