Algorithms & Analysis

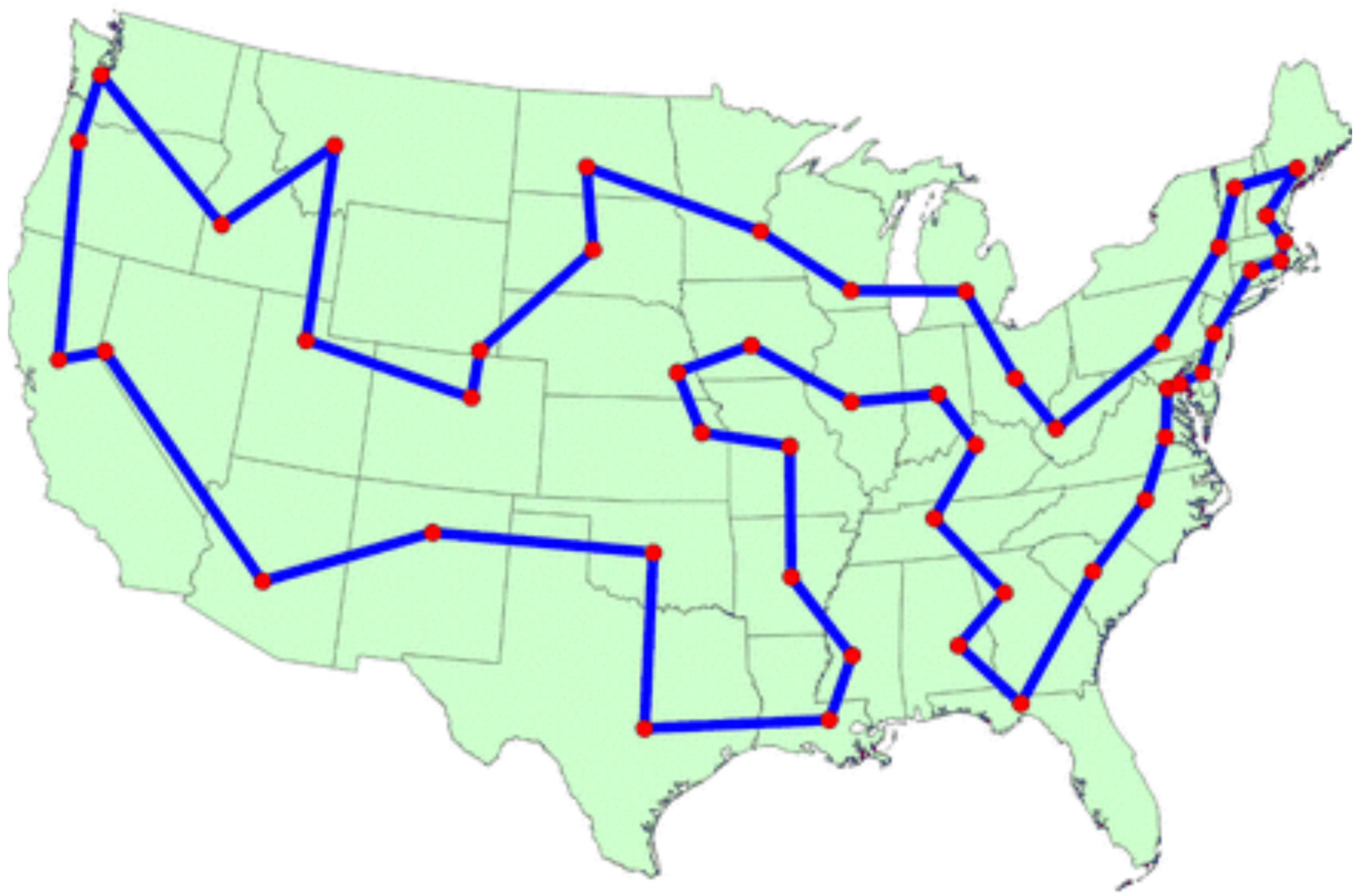
Bring the Big O


How many pebbles?


HEURISTIC
~100
~11
~11
~11
~11
~11
~11
~11
~11
~11

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 (columns) / Departing (rows)

	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

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

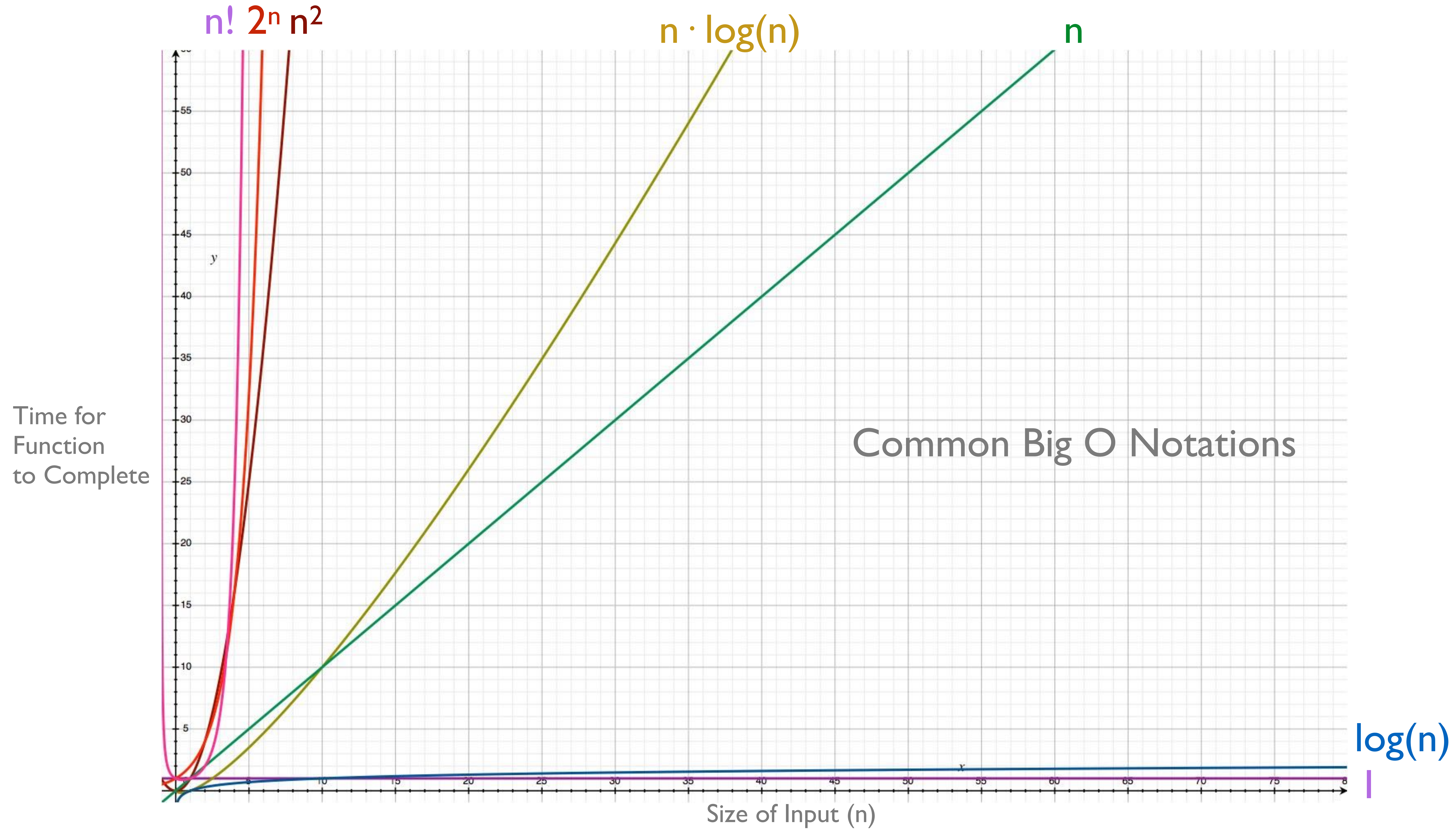
Often we loosely call functions algorithms, because much of the time a function is implementing an algorithm.

ALGORITHM

How can we compare algorithms?

Algorithm Analysis: Big O Notation

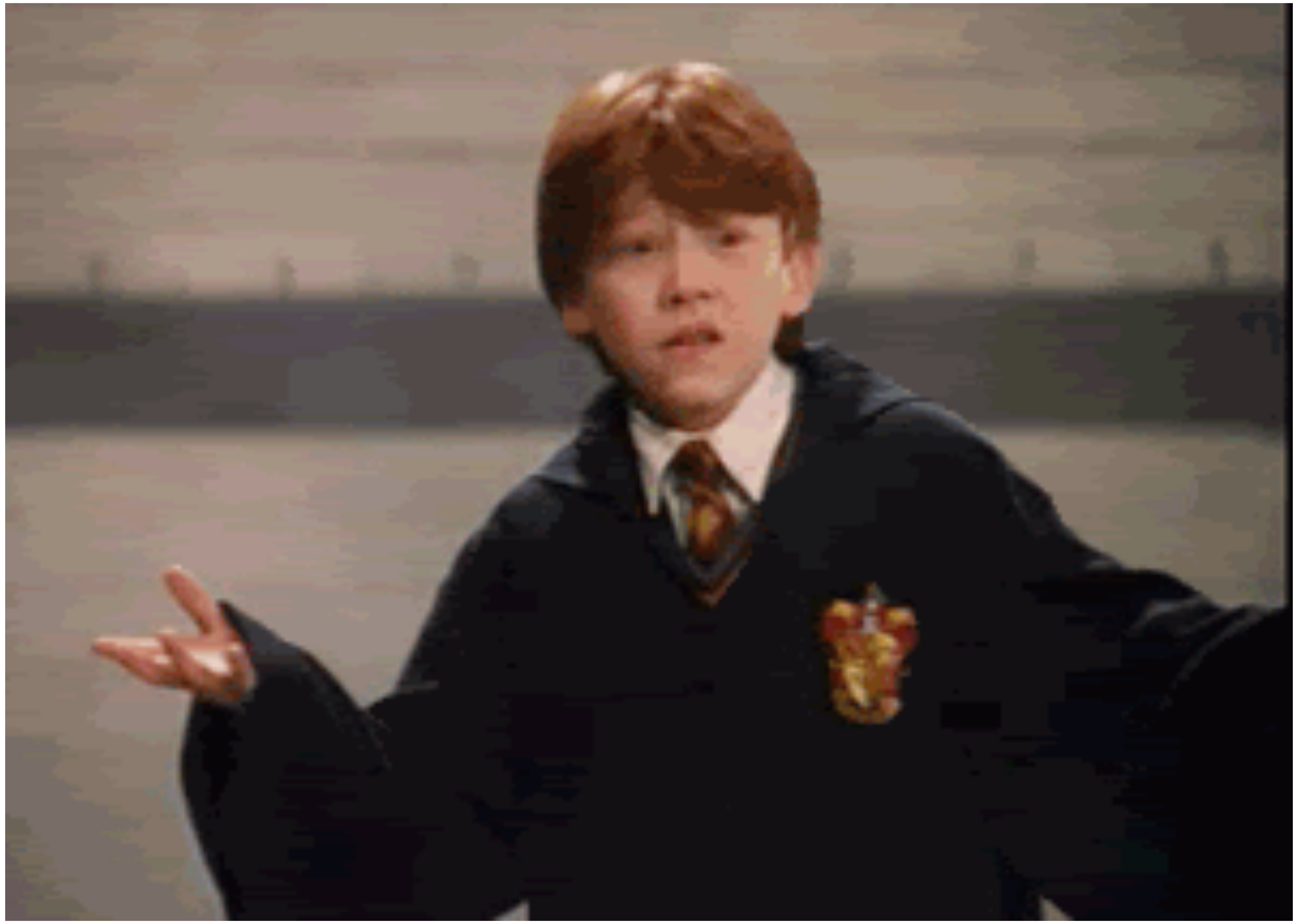
- A *comparative* way to classify different algorithms
- Based on *shape* of *growth curve* (*time* vs *input size(s)*)
- For *big enough* inputs
  - Might not be true when n is small, but who cares when n is small?
- Establishing an *upper bound* on the time
  - Not worse than this. Might be better, 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 $10 \cdot n$)


n! 2ⁿ n²
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
5
10
15
20
25
30
35
40
45
50
55
60
65
70
75
x
Size of Input (n)

*Source: Skiena, The Algorithm Design Manual

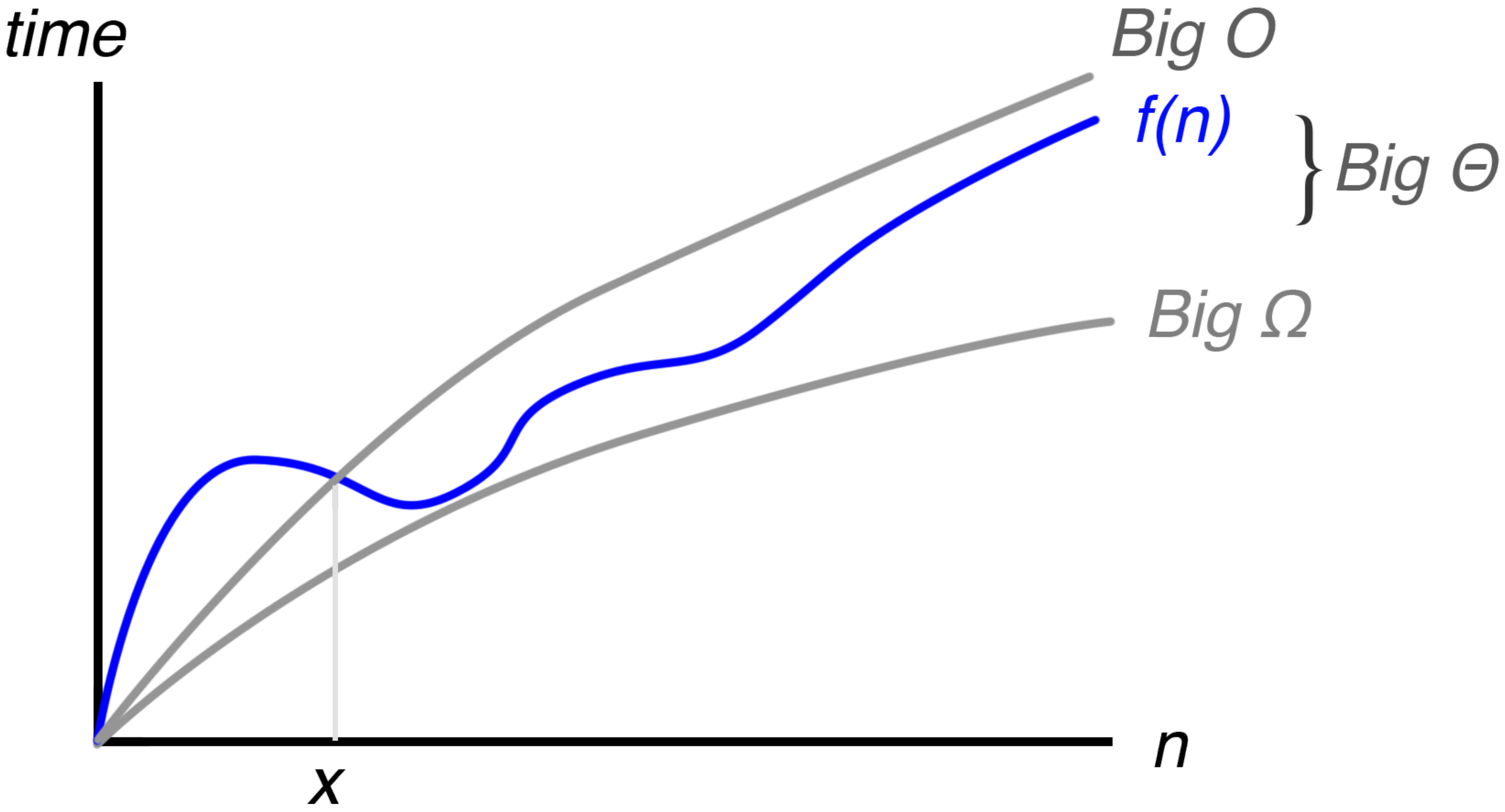
Time Complexities (if 1 op = 1 ns)

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		



What?

Adapted from The Algorithm Design Manual



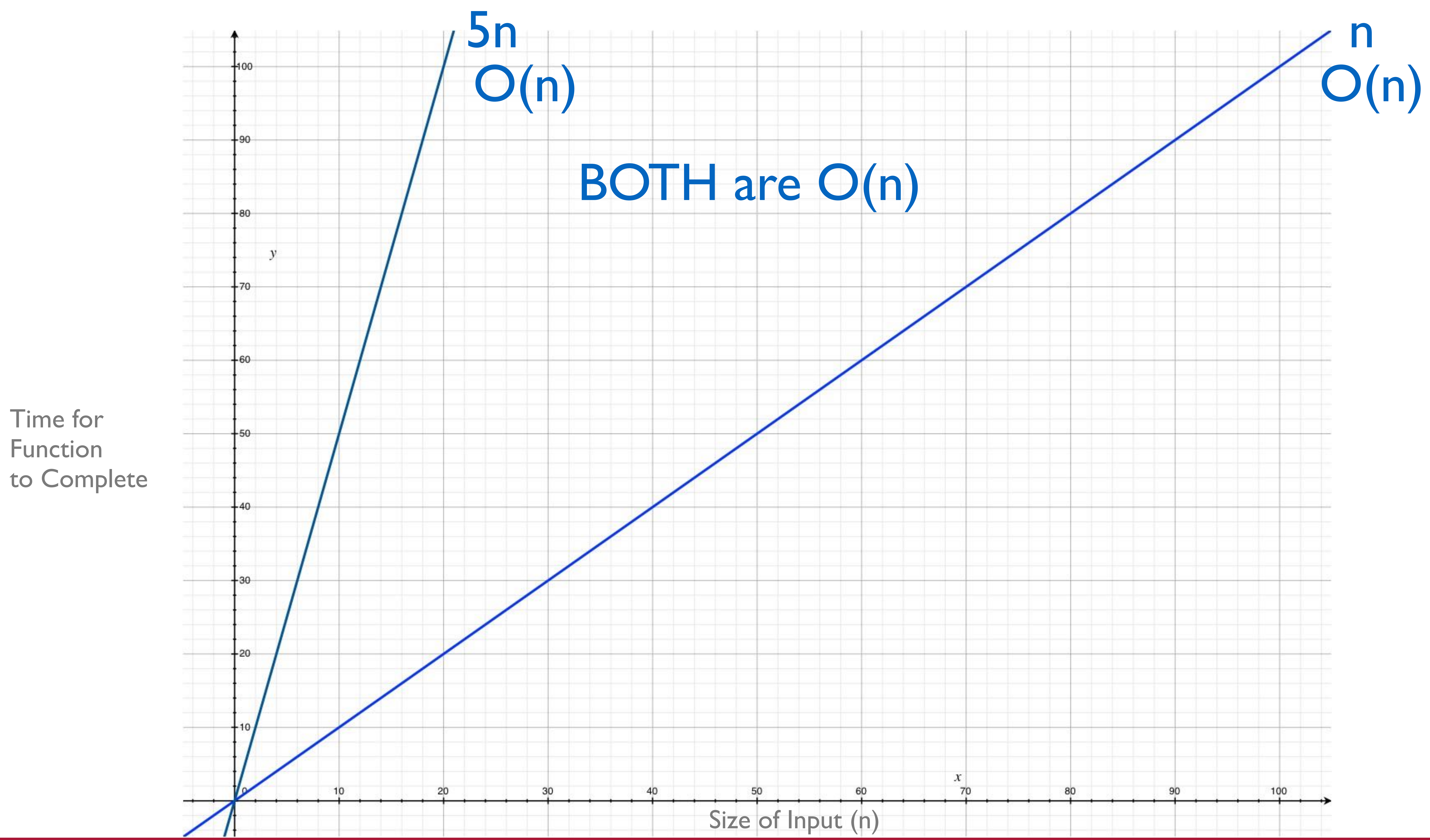
Big O: comparative

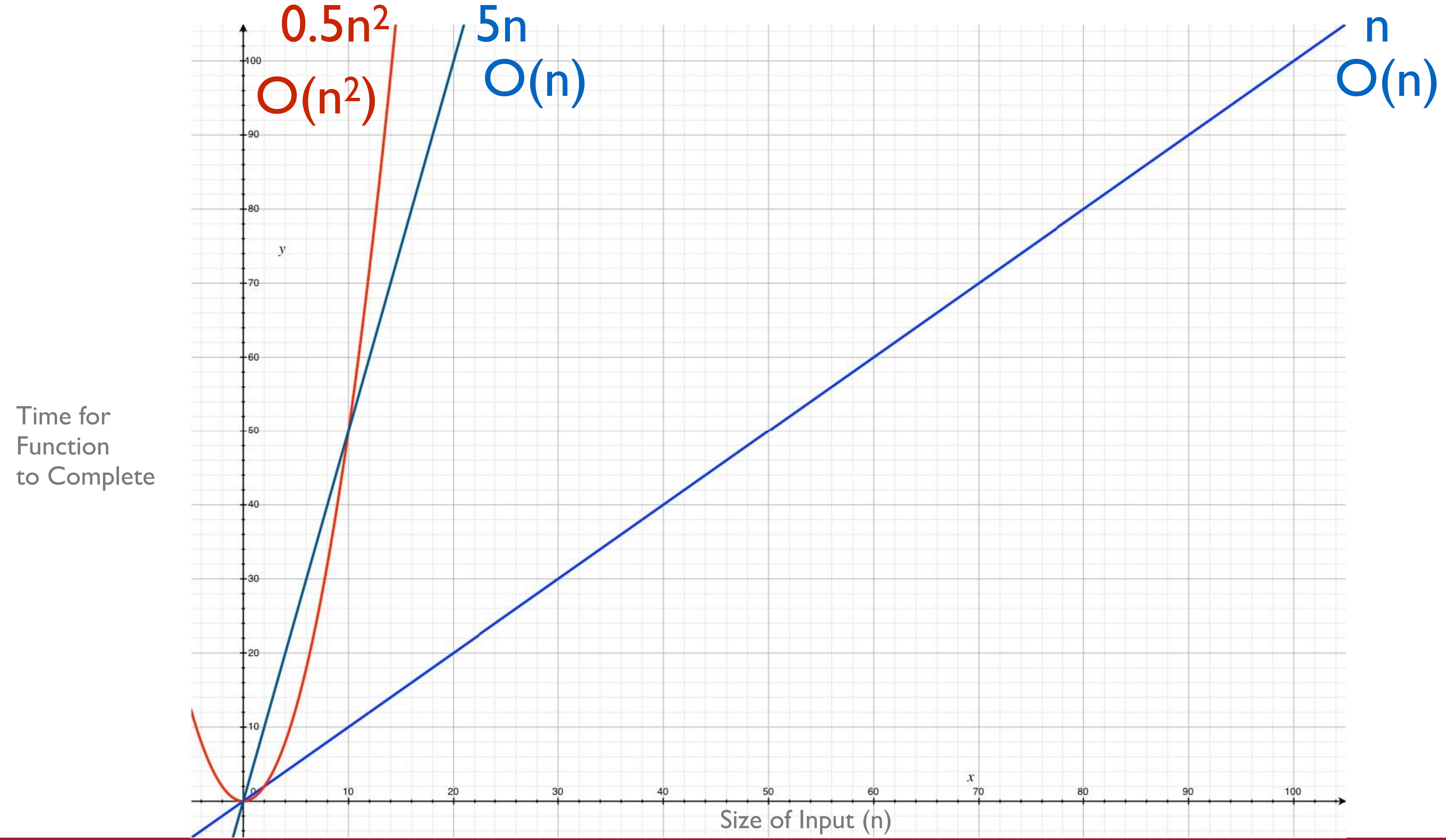
- A very coarse, broad tool — big simplification
- Only useful when algorithms have *different* Big O notations
  - O(n) will always beat O(n^2), *for big enough n*
- If two algorithms have the same Big O, we don't know much. One might actually be quite slower than the other.

Two Linear Functions: O(n)

```
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
0
10
20
30
40
50
60
70
80
90
100



0.5n²
O(n²)
5n
O(n)
n
O(n)
Time for
Function
to Complete
Size of Input (n)
y
x

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*	tree levels * elements	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)