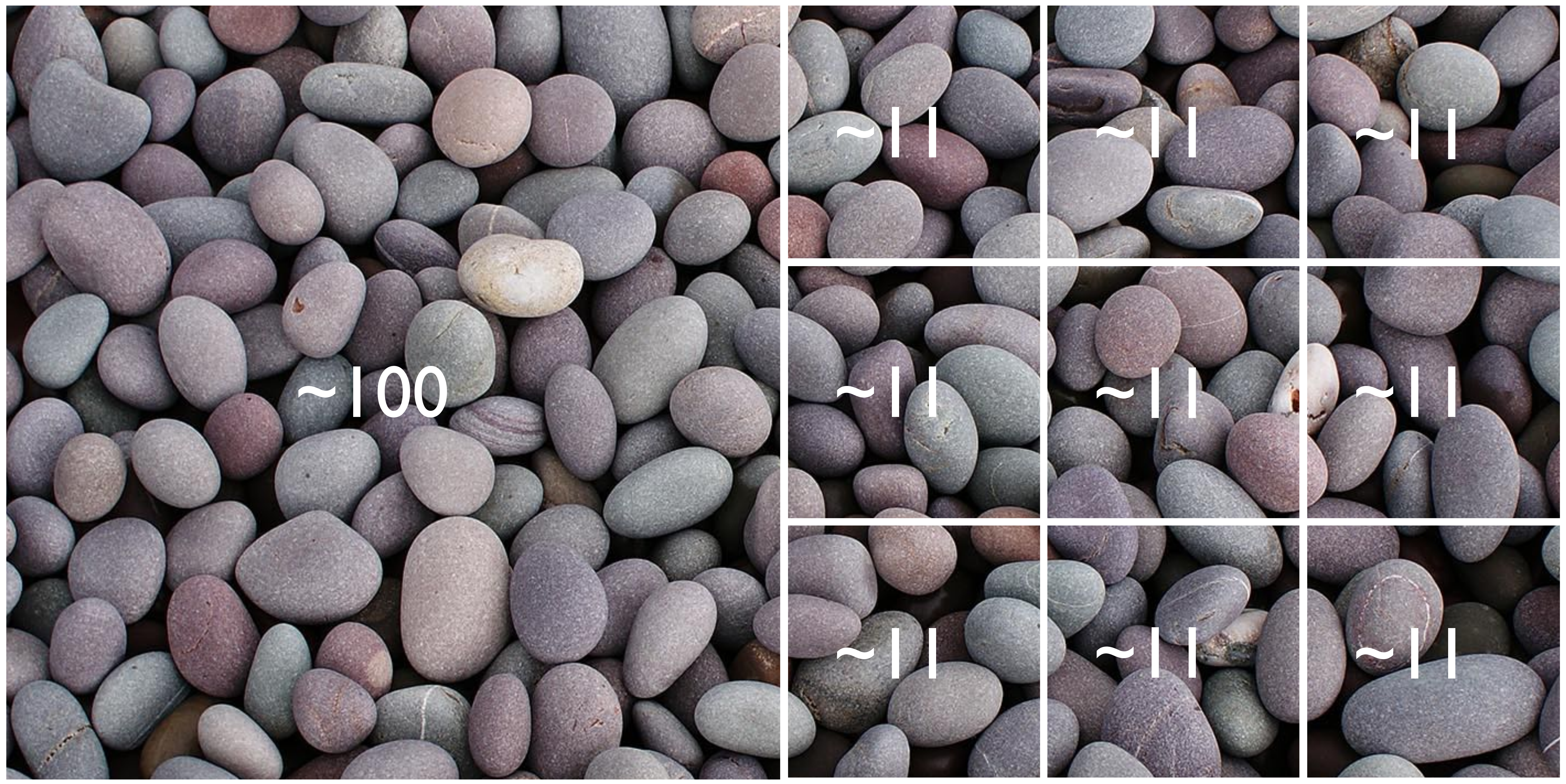
Algorithms & Analysis

Bring the Big O

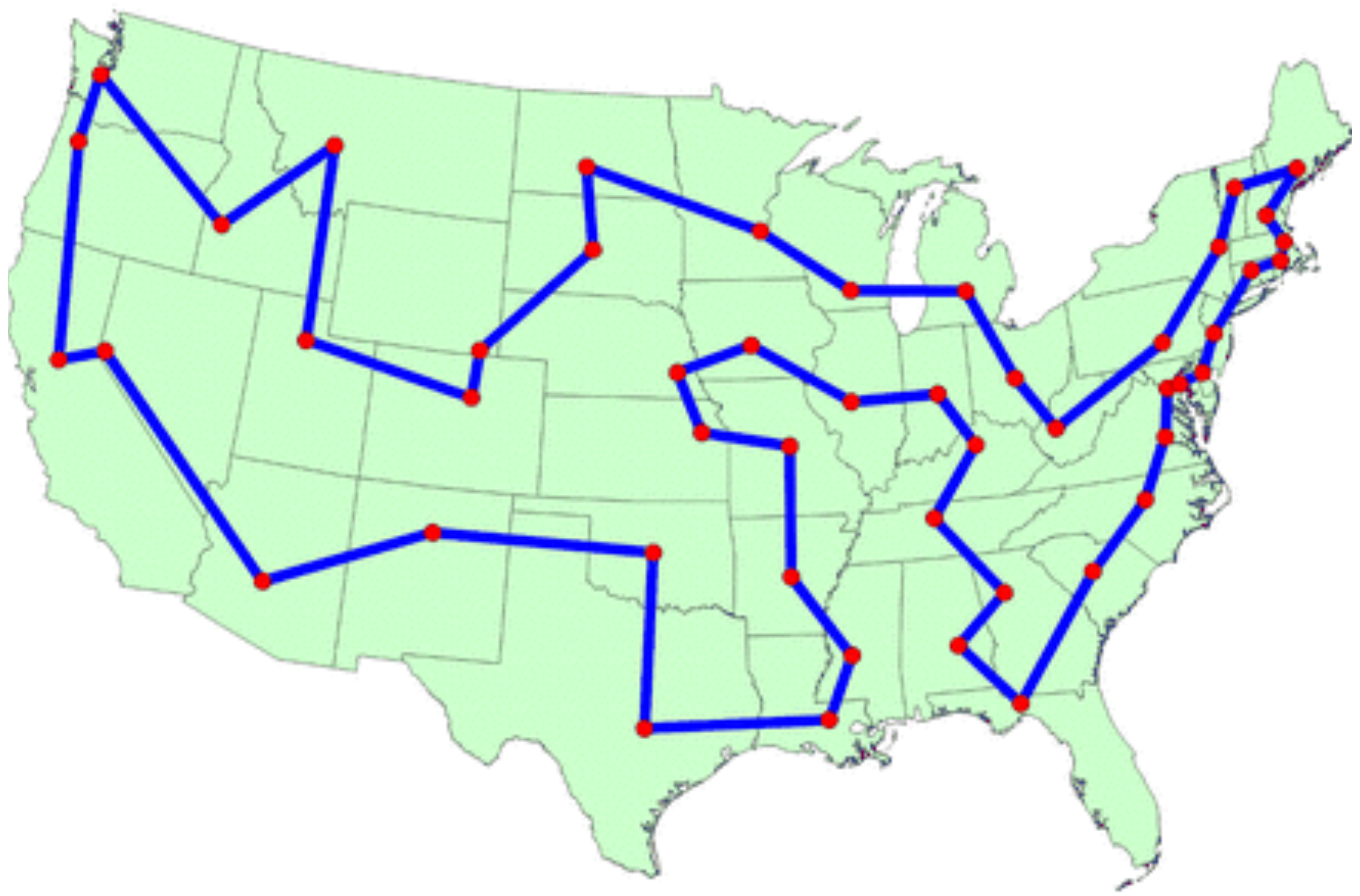

How many pebbles?

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

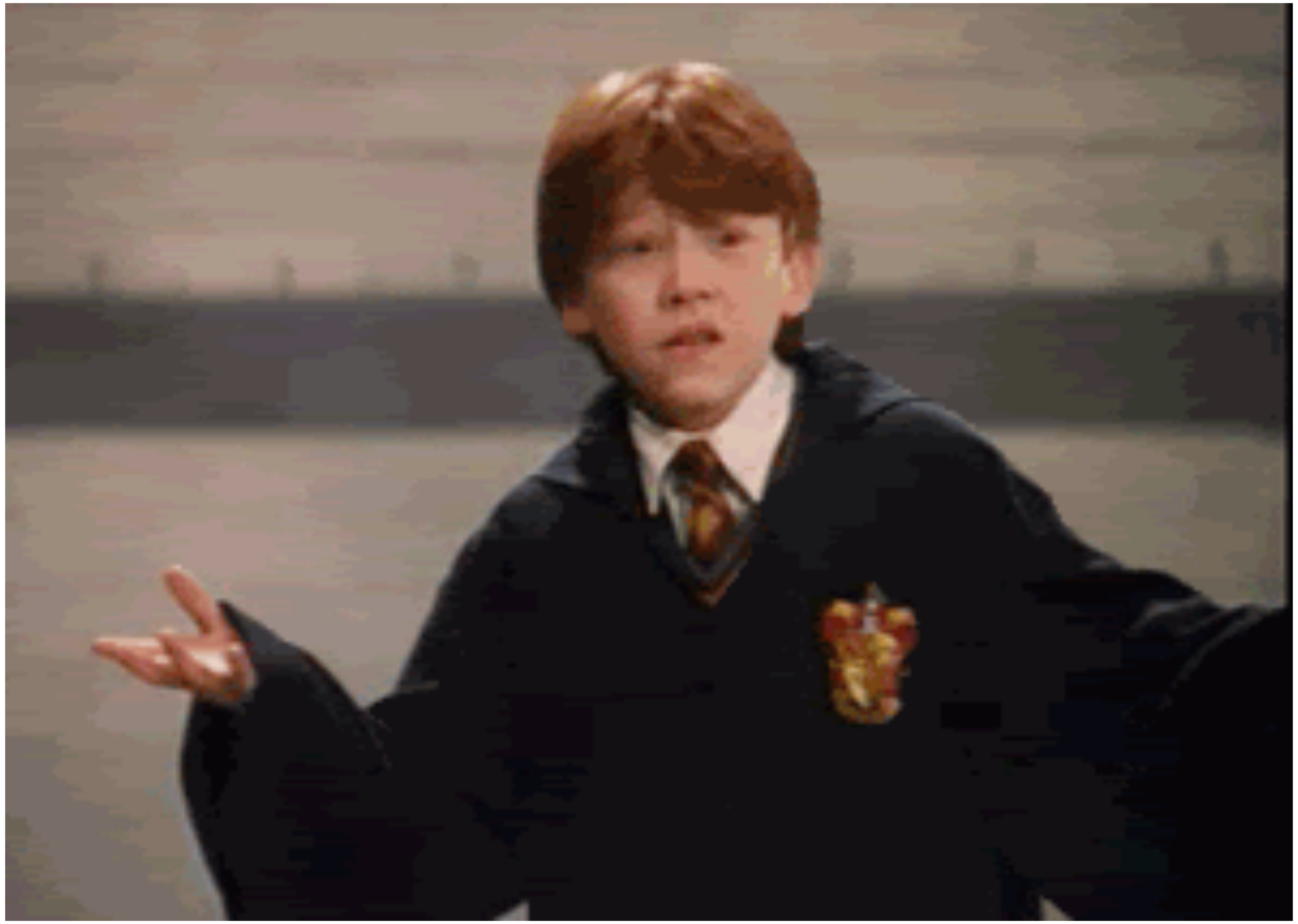
Often we loosely call functions algorithms, because much of the time a function is implementing an algorithm.

How can we compare algorithms?


THE BIG O

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)



What?

Example 1

```
function example (array) {
  console.log(array.length)
  let someNumber = 4
  someNumber += array.length
  return someNumber
}
```

```javascript
function example (array) {
  console.log(array.length)  // 1
  let someNumber = 4
  someNumber += array.length
  return someNumber
}
```

```
function example (array) {
  console.log(array.length)  // 1
  let someNumber = 4         // 1
  someNumber += array.length
  return someNumber
}
```

```
function example (array) {
  console.log(array.length)  // 1
  let someNumber = 4         // 1
  someNumber += array.length // 1
  return someNumber
}
```

```
function example (array) {
  console.log(array.length)  // 1
  let someNumber = 4         // 1
  someNumber += array.length // 1
  return someNumber          // 1
}
```

```
function example (array) {
  console.log(array.length)  // 1
  let someNumber = 4         // 1
  someNumber += array.length // 1
  return someNumber          // 1
}
// O(1 + 1 + 1 + 1) = O(4) = O(1)
```

Example 2

```javascript
// re-naming the array 'n'
function example (n) {
  const len = n.length
  let sum = 0

  for (let i = 0; i < len; i++) {
    sum += n[i]
  }

  return sum
}
```

```javascript
// re-naming the array 'n'
function example (n) {
  const len = n.length                  // 1
  let sum = 0                           // 1

  for (let i = 0; i < len; i++) {
    sum += n[i]
  }

  return sum                           // 1
}
```

```
// re-naming the array 'n'
function example (n) {
  const len = n.length                // 1
  let sum = 0                         // 1

  for (let i = 0; i < len; i++) { // n
    sum += n[i]                        // 1
  }

  return sum                         // 1
}
```

```
// re-naming the array 'n'
function example (n) {
  const len = n.length                    // 1
  let sum = 0                             // 1

  for (let i = 0; i < len; i++) { // n
    sum += n[i]                            // 1
  }

  return sum                             // 1
}
// O(1 + 1 + (n * 1) + 1) = O(3 + n) = O(n)
```

Example 3

```
function example (n) {
  const len = n.length

  for (let i = 0; i < len; i++) {
    console.log(n[i])
  }

  for (let j = 0; j < len; j++) {
    if (n[i] > 5) {
      console.log(n[i])
    }
  }

  return len
}
```

```
function example (n) {
  const len = n.length                  // 1

  for (let i = 0; i < len; i++) {
    console.log(n[i])
  }

  for (let j = 0; j < len; j++) {
    if (n[i] > 5) {
      console.log(n[i])
    }
  }

  return len                            // 1
}
```

```
function example (n) {
  const len = n.length                      // 1

  for (let i = 0; i < len; i++) { // n
    console.log(n[i])                         // 1
  }

  for (let j = 0; j < len; j++) { // n
    if (n[i] > 5) {                           // assume this always runs
      console.log(n[i])                       // 1
    }
  }

  return len                                // 1
}
```

```
function example (n) {
  const len = n.length                     // 1

  for (let i = 0; i < len; i++) { // n
    console.log(n[i])                       // 1
  }

  for (let j = 0; j < len; j++) { // n
    if (n[i] > 5) {                         // assume this always runs
      console.log(n[i])                     // 1
    }
  }

  return len                               // 1
}
// O(1 + (n * 1) + (n * 1) + 1) = O(2 + 2n) = O(2n) = O(n)
```

Example 4

```
function example (n) {
  for (let i = 0; i < n.length; i++) {
    for (let j = 0; j < n.length; j++) {
      console.log(n[i] + n[j])
    }
  }
}
```

```
function example (n) {
  for (let i = 0; i < n.length; i++) {   // n
    for (let j = 0; j < n.length; j++) {
      console.log(n[i] + n[j])
    }
  }
}
```

```
function example (n) {
  for (let i = 0; i < n.length; i++) {   // n
    for (let j = 0; j < n.length; j++) {   // n
      console.log(n[i] + n[j])
    }
  }
}
```

```
function example (n) {
  for (let i = 0; i < n.length; i++) {   // n
    for (let j = 0; j < n.length; j++) {   // n
      console.log(n[i] + n[j])                   // 1
    }
  }
}
```

```
function example (n) {
  for (let i = 0; i < n.length; i++) {   // n
    for (let j = 0; j < n.length; j++) {   // n
      console.log(n[i] + n[j])                   // 1
    }
  }
}
// O((n * (n * 1)) = O(n^2)
```

Example 5

```
// now, n is a number
function example (n) {
  let counter = 0

  while (n > 1) {
    n = n / 2
    counter++
  }

  return counter
}
```

```
// now, n is a number
function example (n) {
  let counter = 0  // 1

  while (n > 1) {
    n = n / 2
    counter++
  }

  return counter  // 1
}
```

```
// now, n is a number
function example (n) {
  let counter = 0  // 1

  while (n > 1) {  // ?
    n = n / 2
    counter++
  }

  return counter  // 1
}
```

```
// now, n is a number
function example (n) {
  let counter = 0  // 1

  while (n > 1) {  // log(n)
    n = n / 2
    counter++
  }

  return counter  // 1
}
```

```
// now, n is a number
function example (n) {
  let counter = 0  // 1

  while (n > 1) {  // log(n)
    n = n / 2
    counter++
  }

  return counter  // 1
}
// O(2 + log(n)) = O(log(n))
```

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?*

$\log_2(2) = 1$

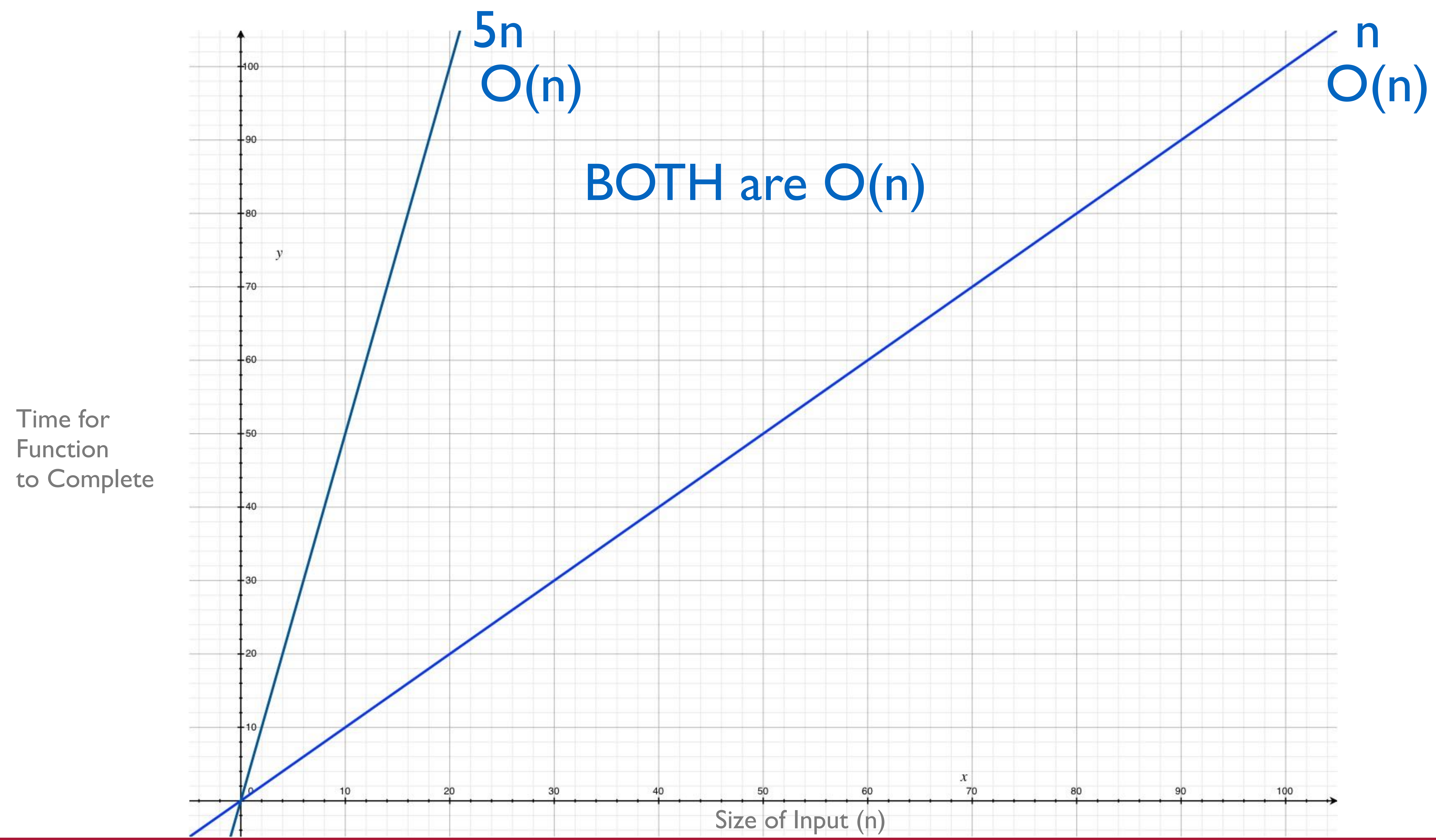
$\log_2(4) = 2$

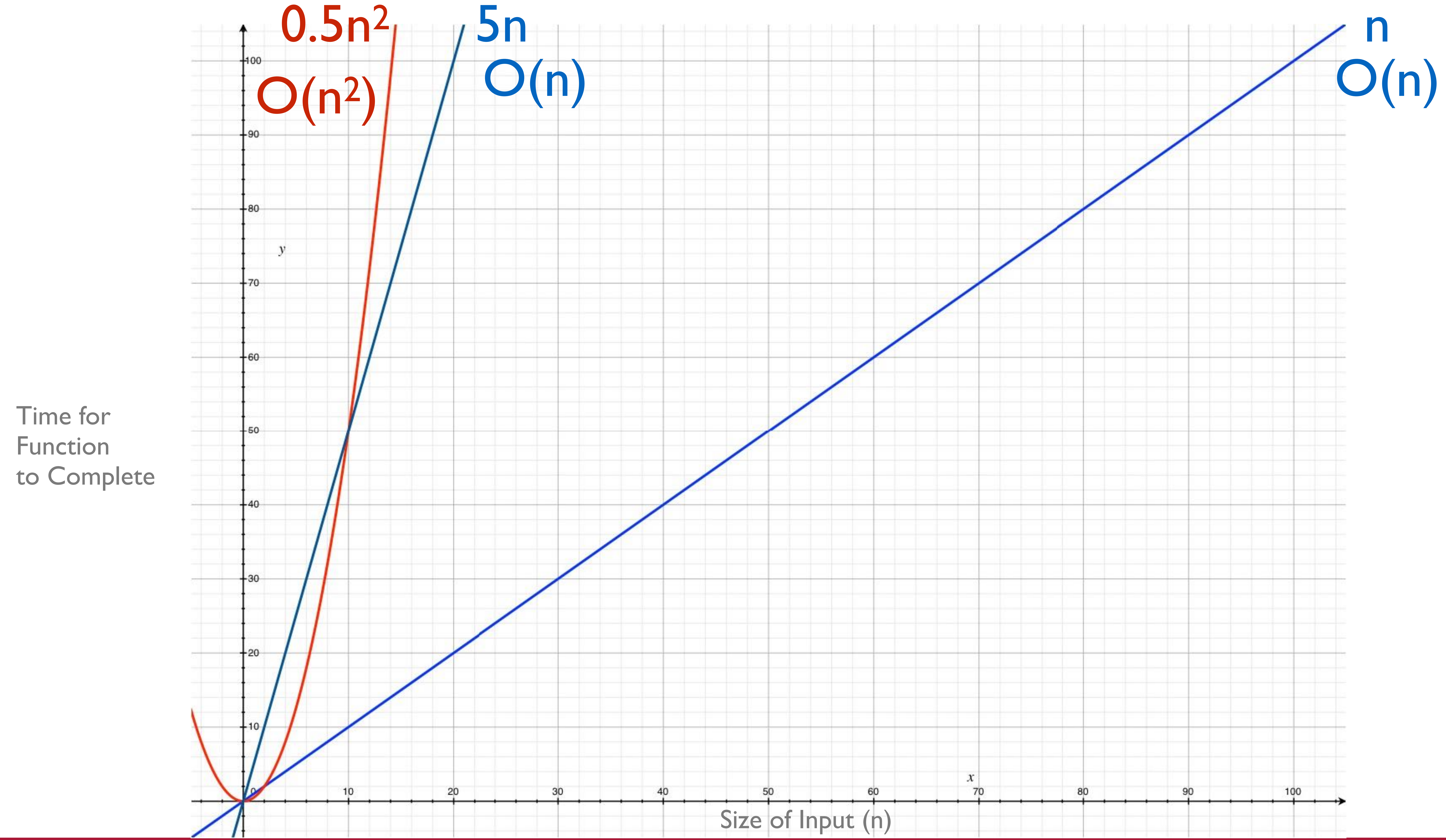
$\log_2(8) = 3$

$\log_2(5) = 2.32192809489$

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 $10 \cdot n$)


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
$n \cdot \log(n)$
n
Time for Function to Complete
Common Big O Notations
log(n)
1
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*	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)