

# Algorithms & Analysis

---

*Bring the Big O*







A close-up, top-down view of a large number of smooth, rounded pebbles. The pebbles are densely packed and vary in color, including shades of purple, blue, red, and grey. The lighting is even, highlighting the smooth texture of the stones.

*How many pebbles?*

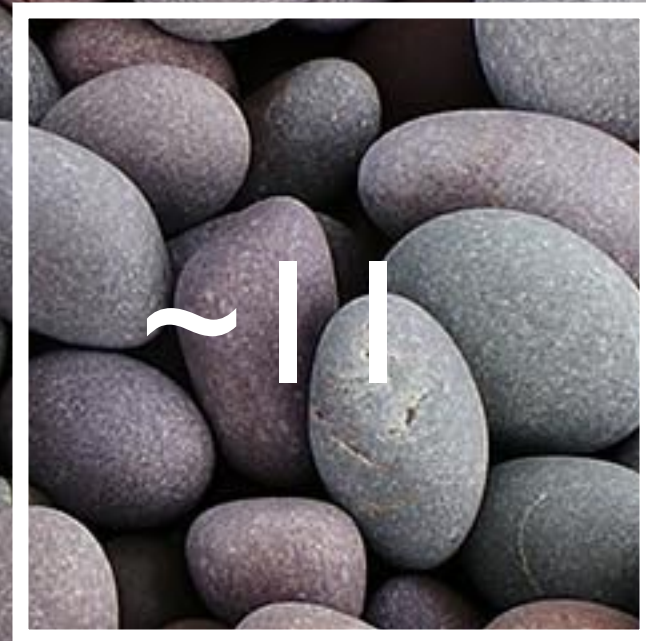




# HEURISTIC

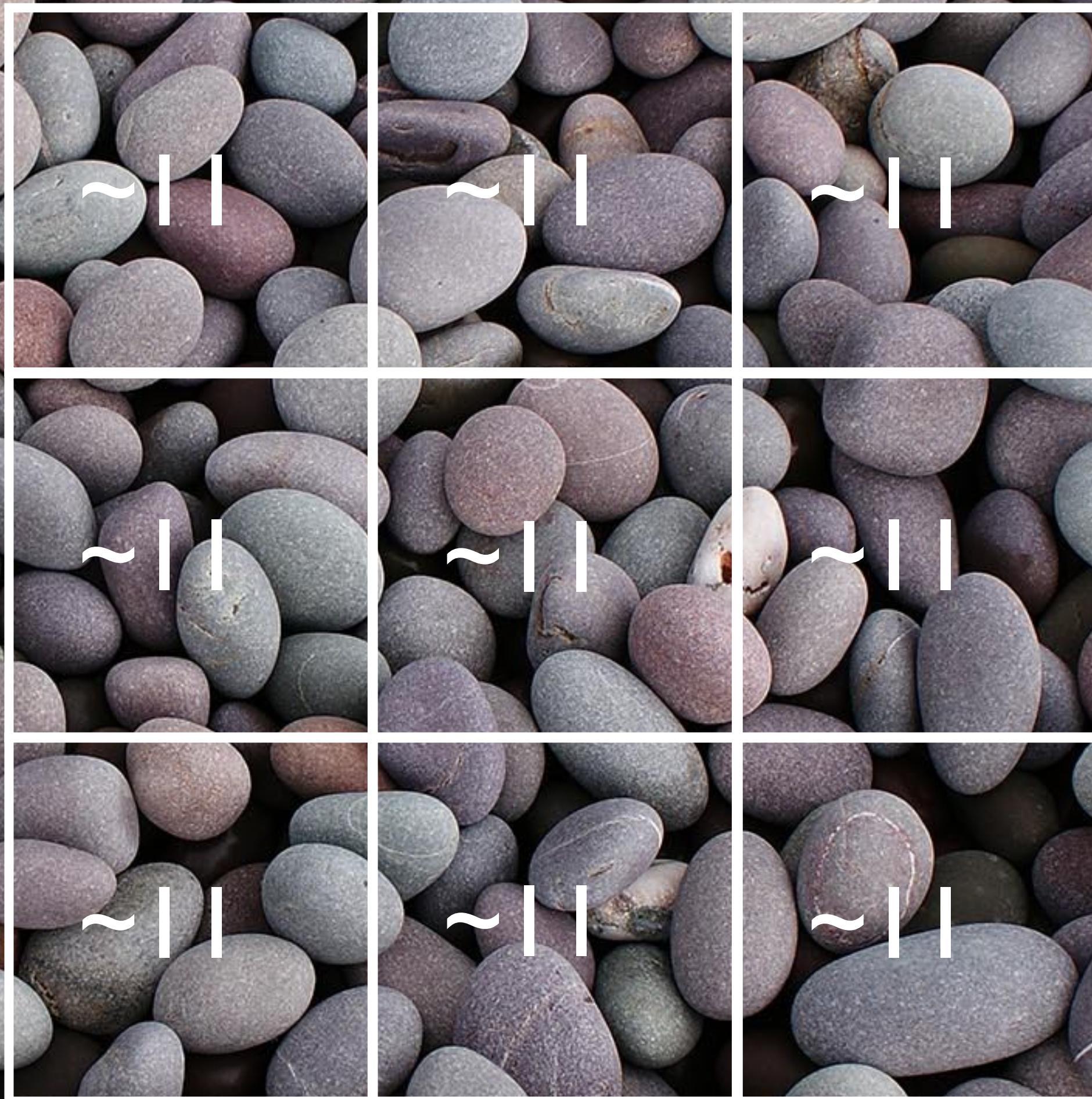


# HEURISTIC



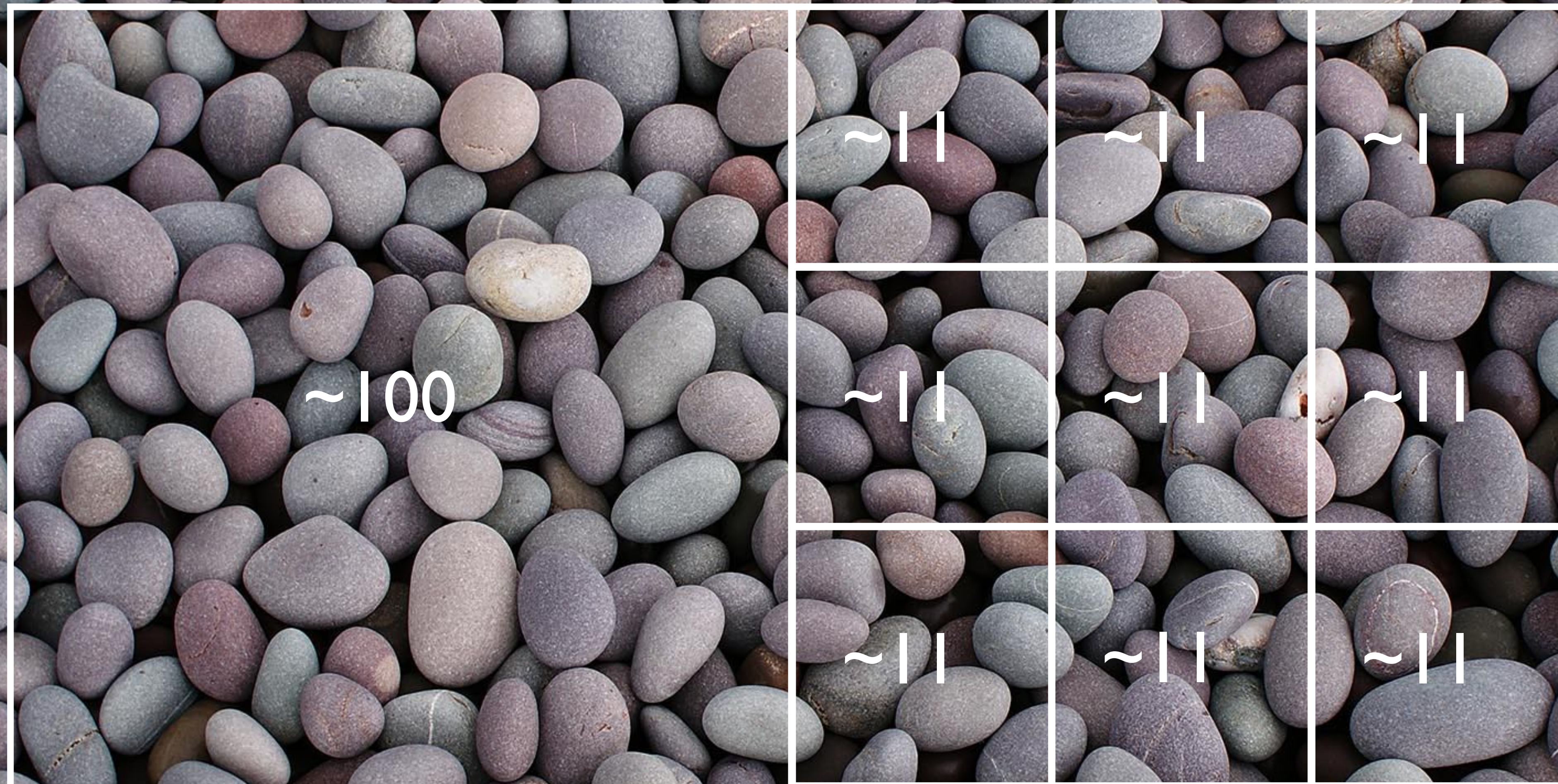


# HEURISTIC





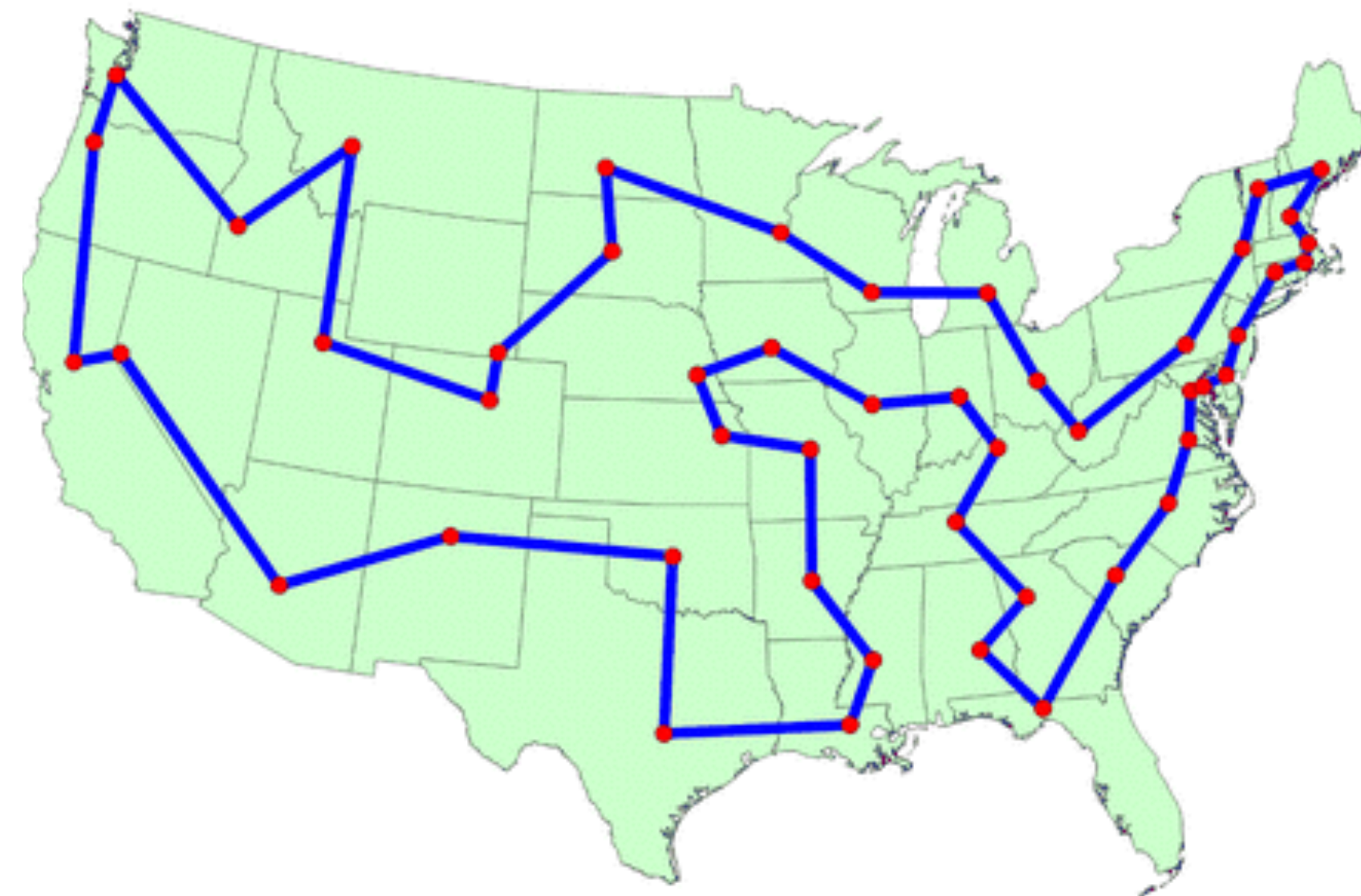
# HEURISTIC





# 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



# 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?



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



# 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

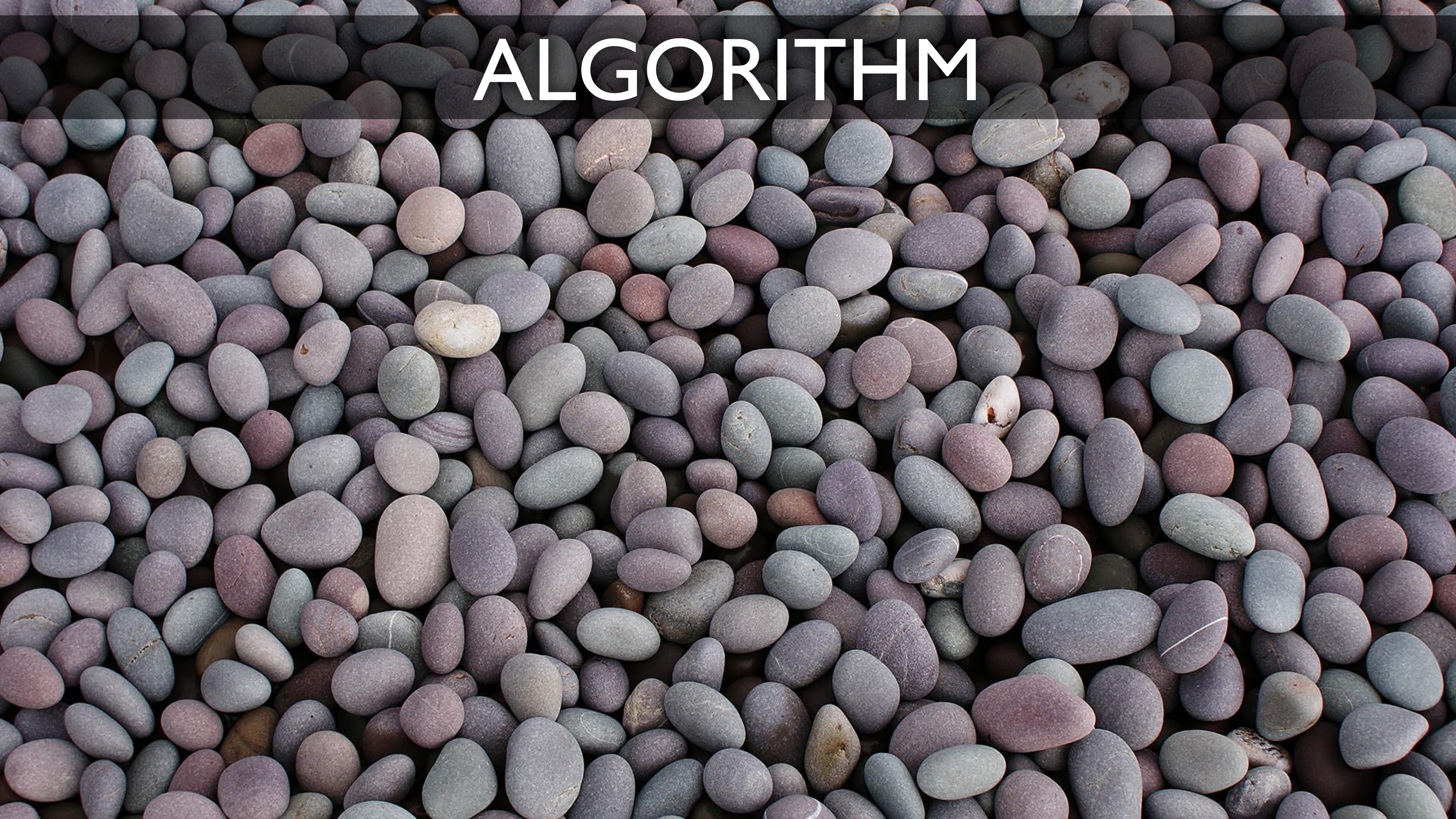
Departing

	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





# ALGORITHM

I



# ALGORITHM

1 2



# ALGORITHM

1 2 3



# ALGORITHM

1 2 3  
4



# ALGORITHM

1 2 3 5  
4



# ALGORITHM

1 2 3 5  
4 6



# ALGORITHM

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 ...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





# 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)**





**What?**





**What?**



# Example 1



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



```
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  
}  
// 0(1 + 1 + 1 + 1) = 0(4) = 0(1)
```



# Example 2



```
// 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
}
```



```
// 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
}
// 0(1 + 1 + (n * 1) + 1) = 0(3 + n) = 0(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
}

// 0(1 + (n * 1) + (n * 1) + 1) = 0(2 + 2n) = 0(2n) = 0(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?*



$$\log_2(2) = 1$$

$$\log_2(4) = 2$$

$$\log_2(8) = 3$$

$$\log_2(5) = 2.32192809489$$

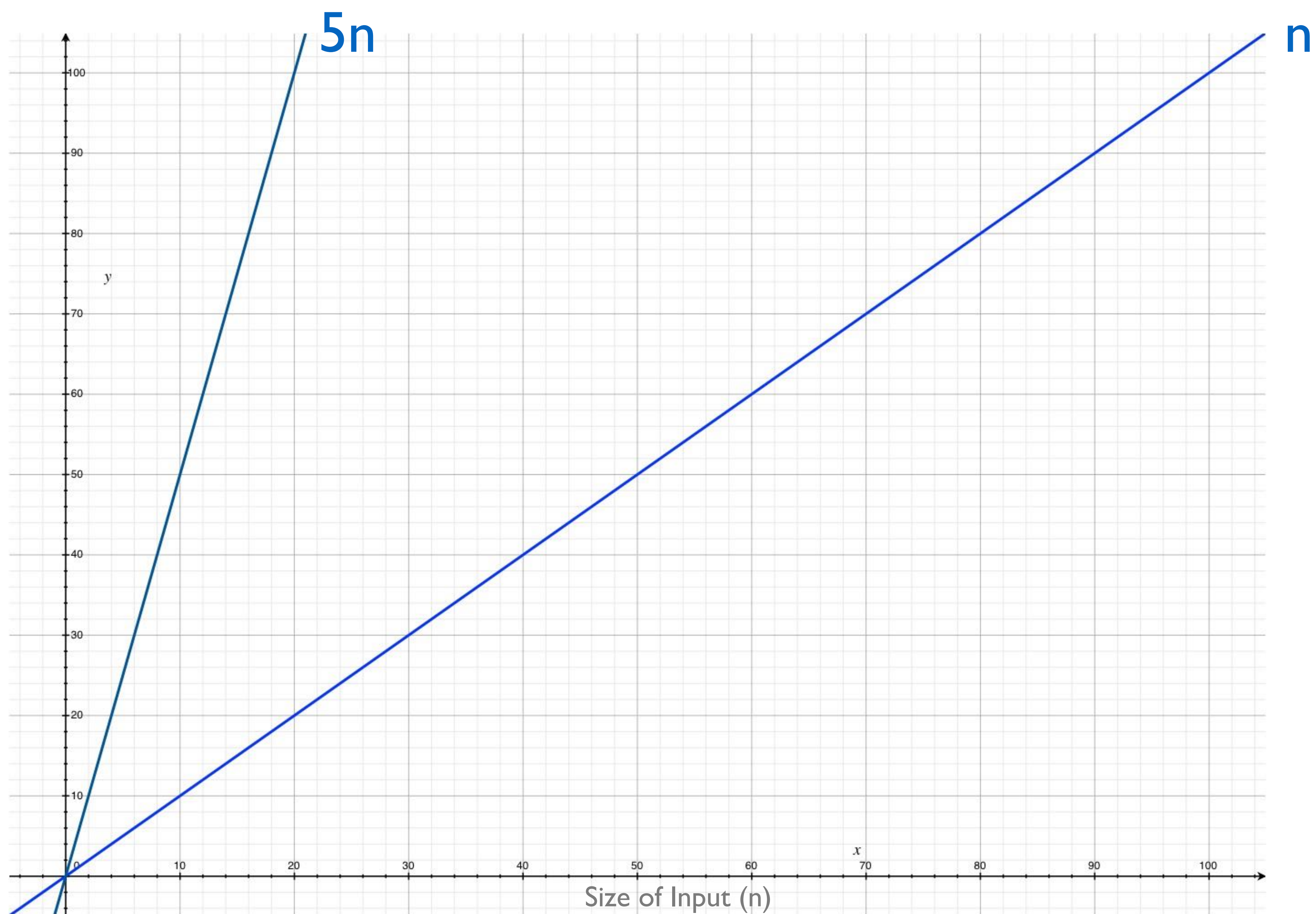


# Algorithm Analysis: Big O Notation

- 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  $10 \cdot n$ )

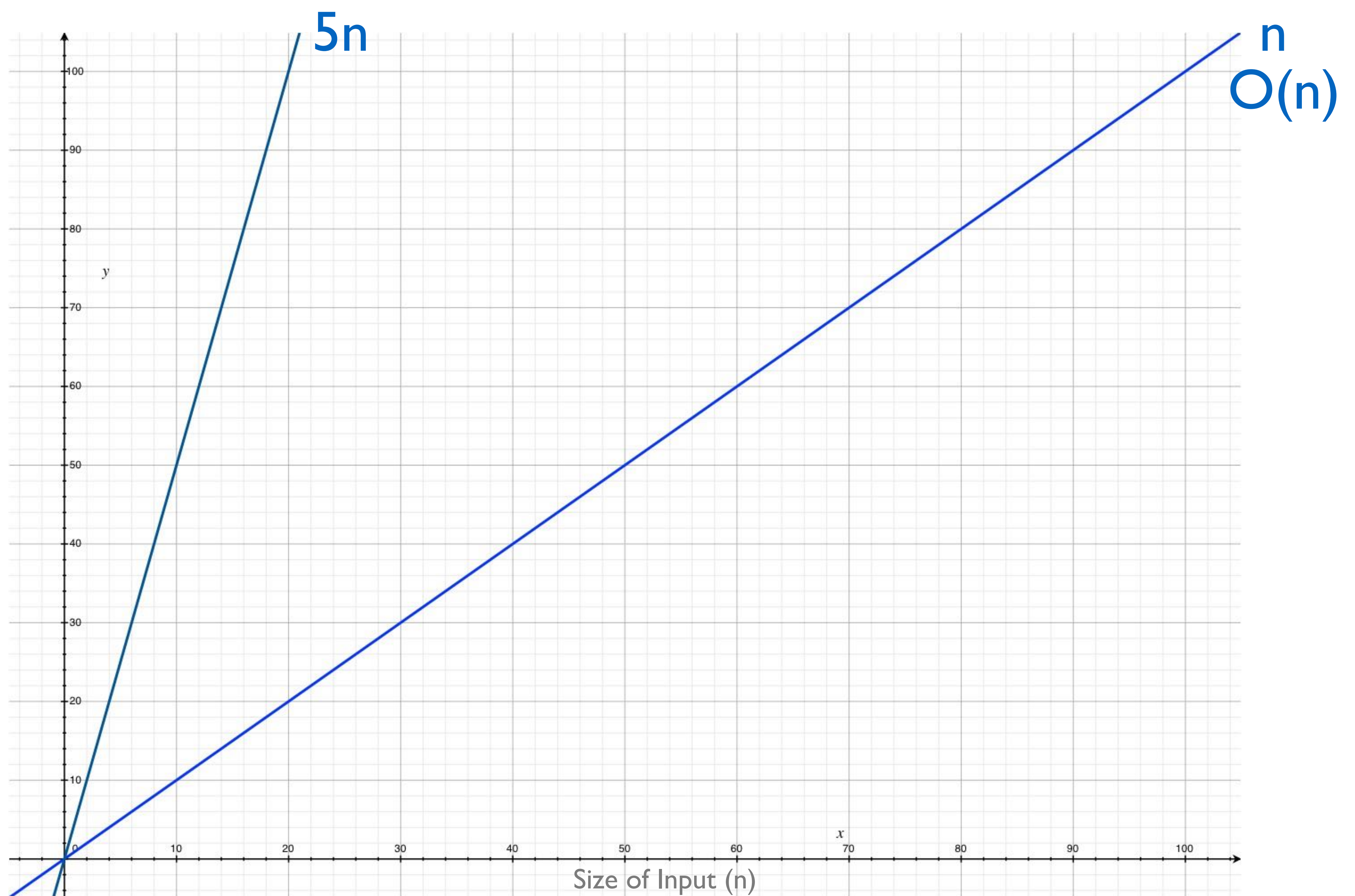


Time for  
Function  
to Complete



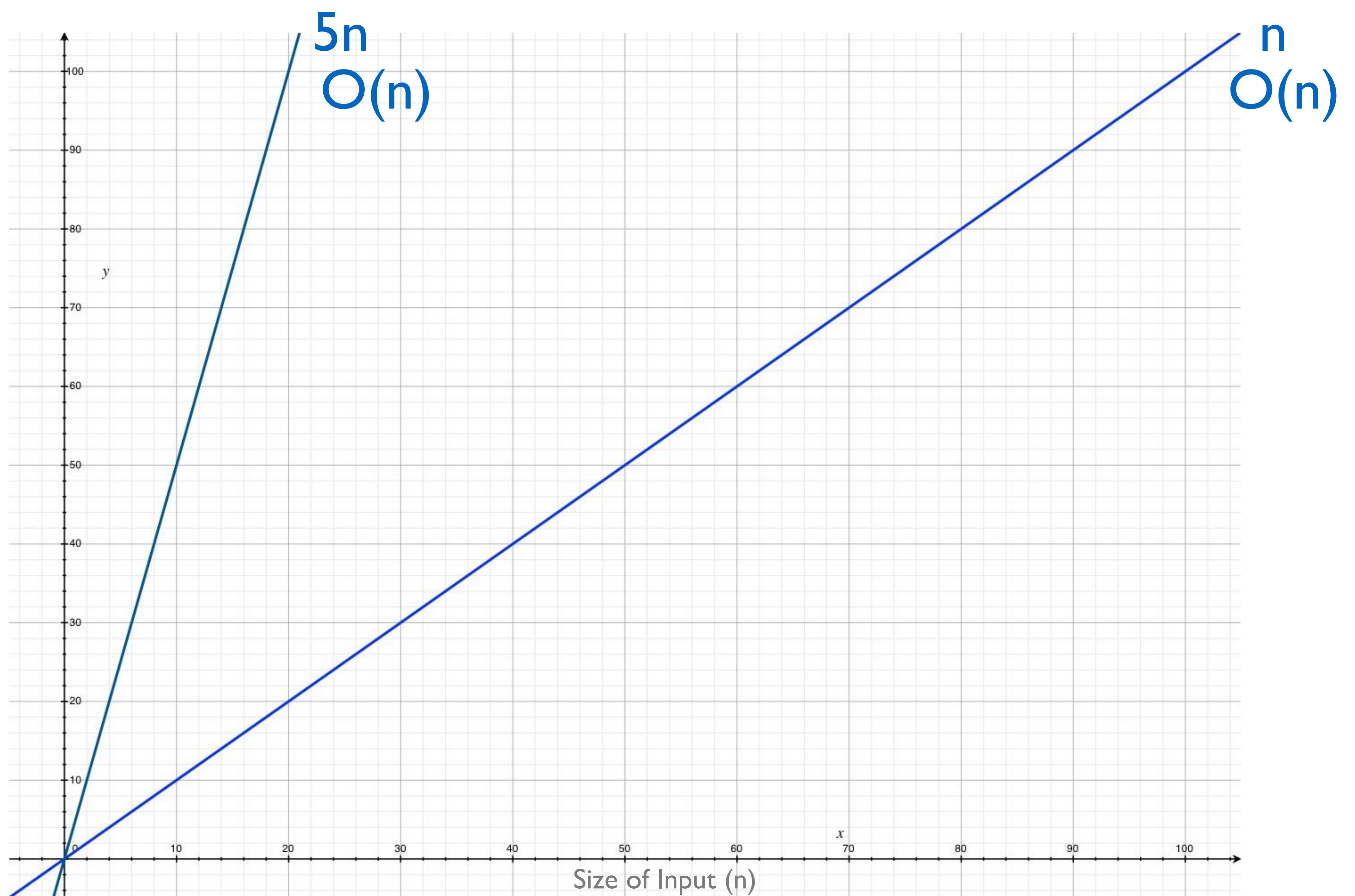


Time for  
Function  
to Complete



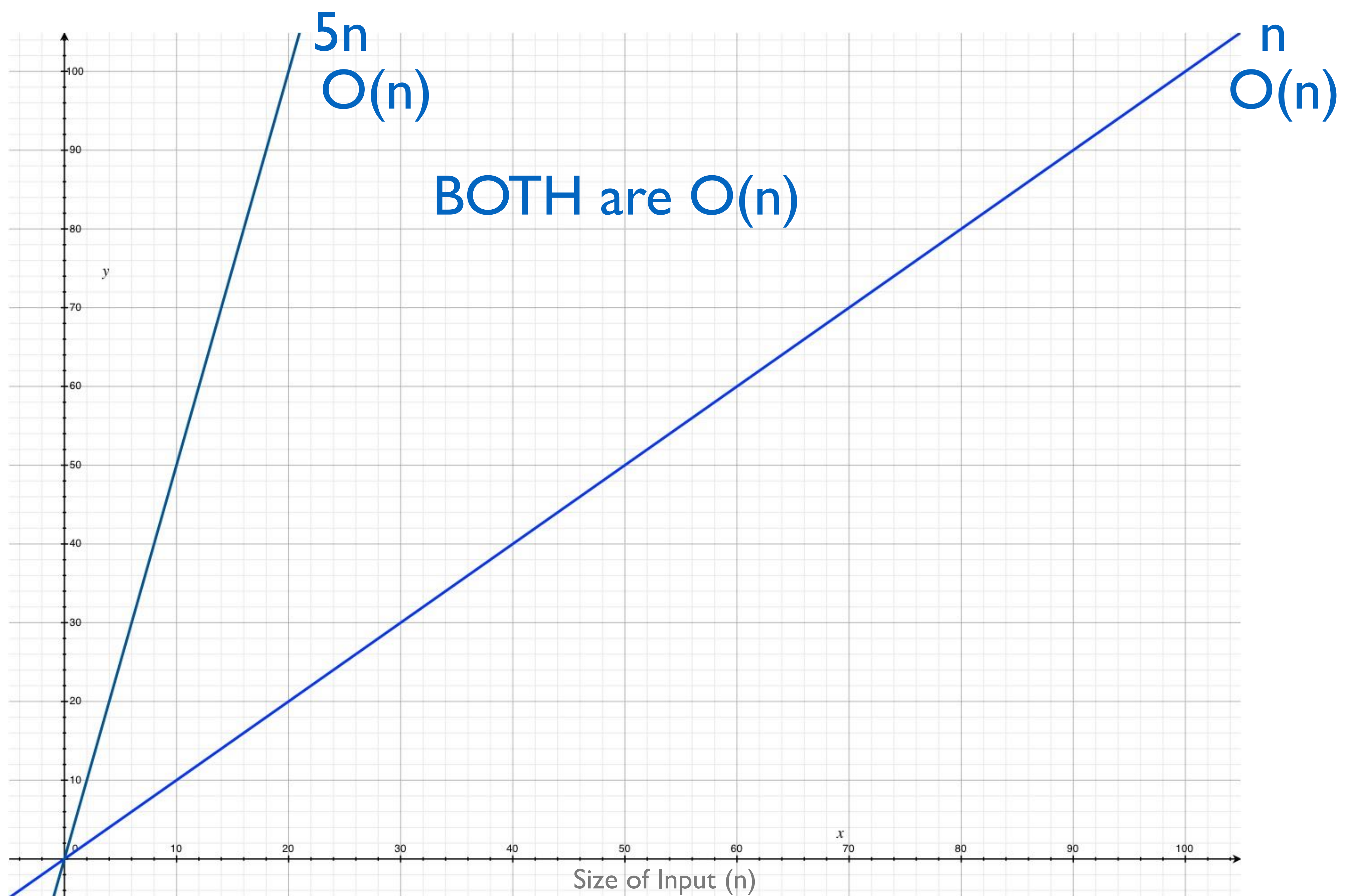


Time for  
Function  
to Complete



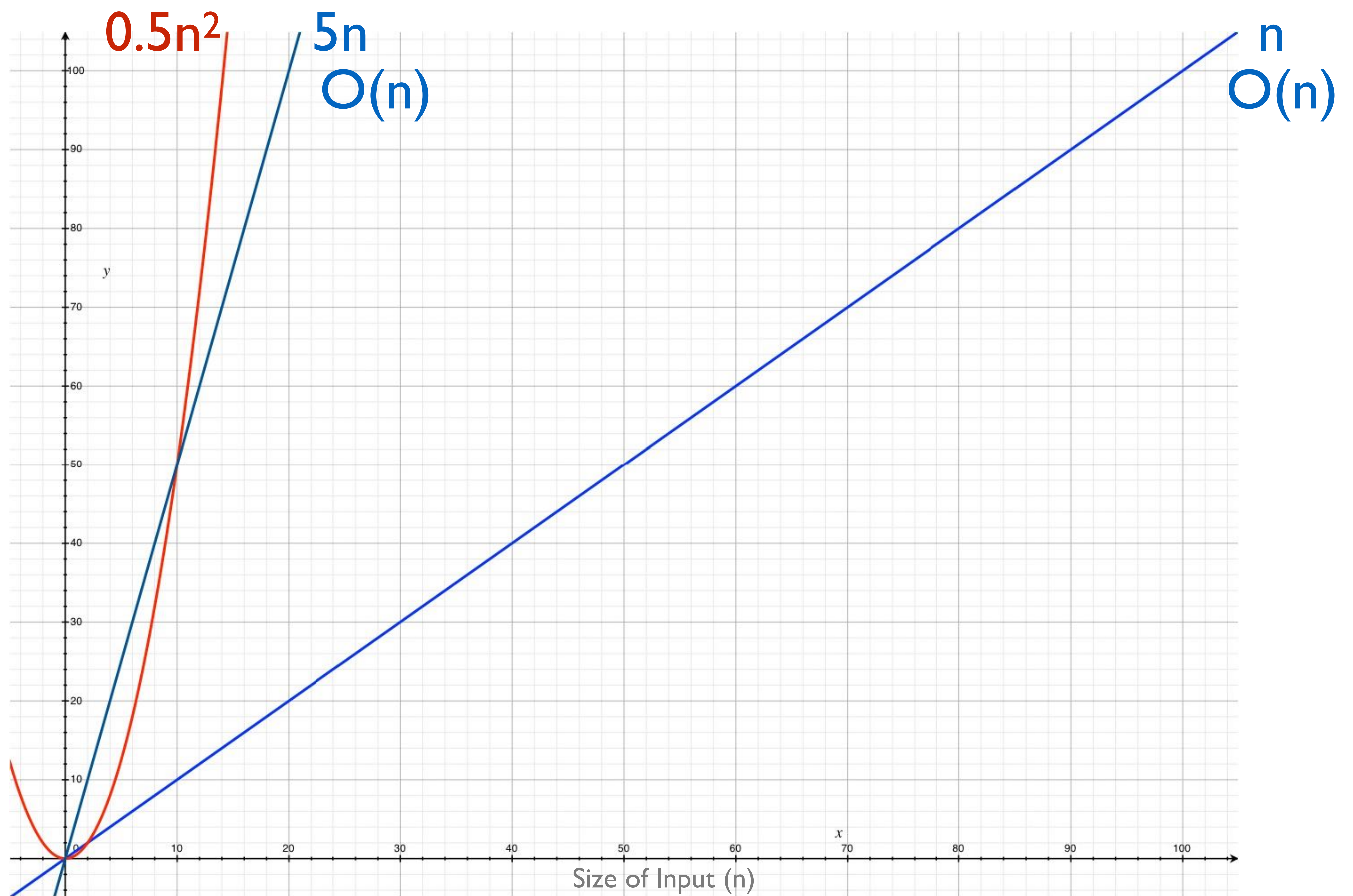


Time for  
Function  
to Complete



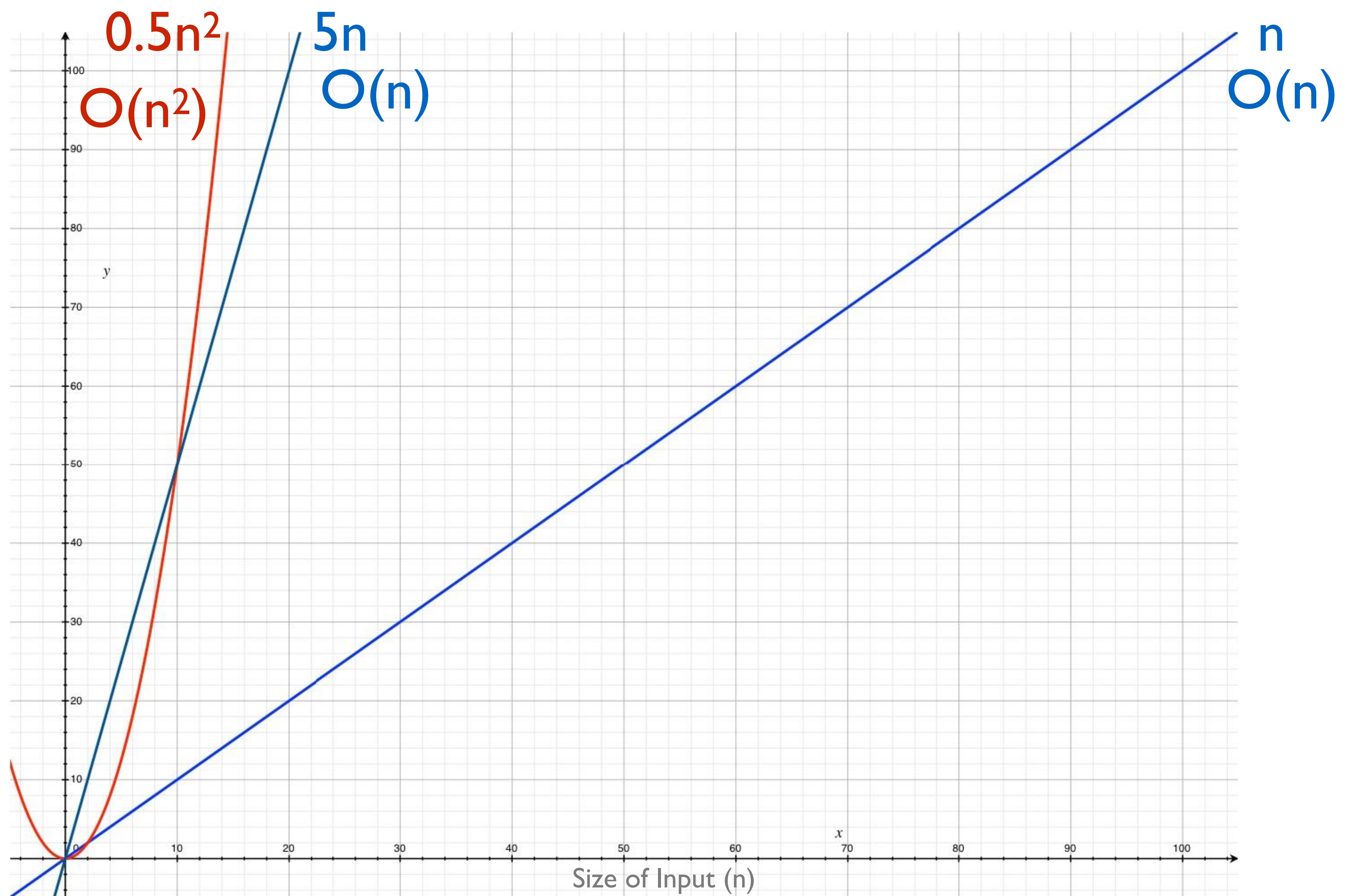


Time for  
Function  
to Complete

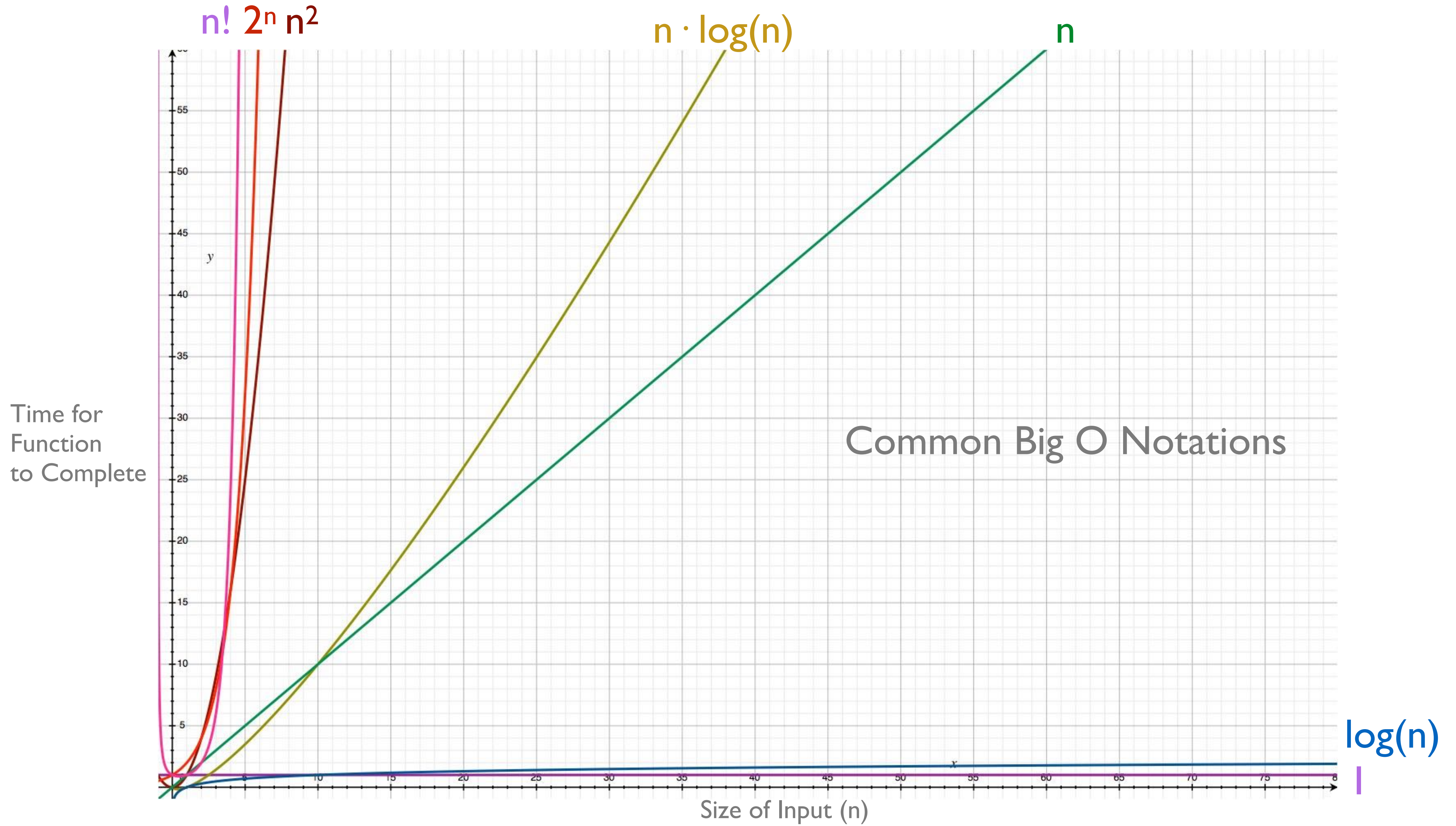




Time for  
Function  
to Complete











\*Source: Skiena, The Algorithm Design Manual

# Time Complexities (if 1 op = 1 ns)

input size n		log n	n	n·log n	n <sup>2</sup>	2 <sup>n</sup>	n!
input size 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





# 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	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$