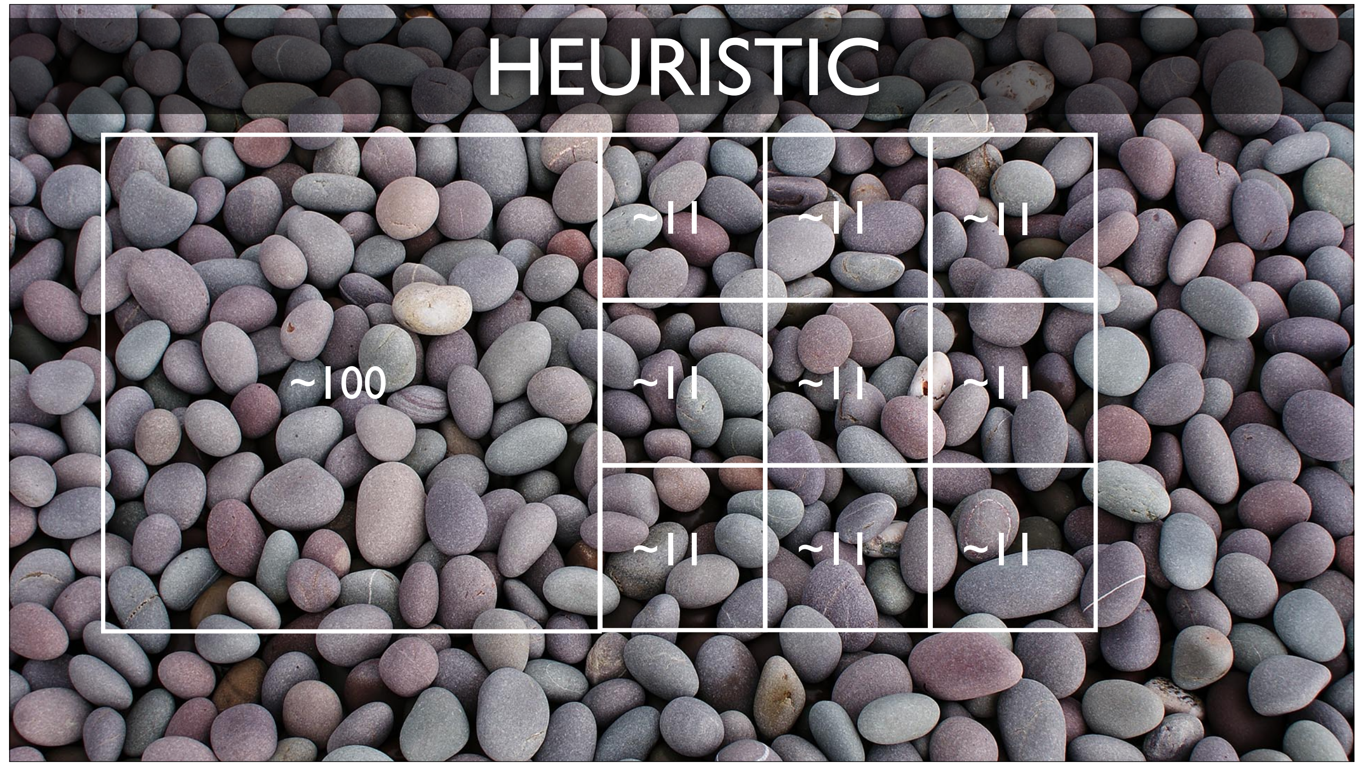
Algorithms & Analysis

Bring the Big O


How many pebbles?

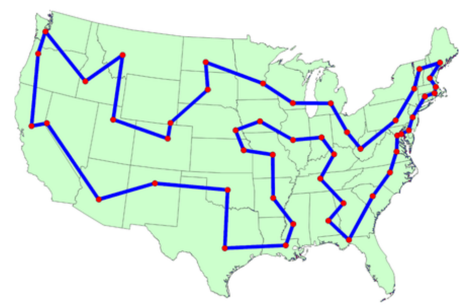


Example of heuristic vs. algorithm. How many pebbles? If we make a box, it fits about 11 pebbles. If we make a box of three by three boxes, that's ~100 pebbles. If we start adding even bigger boxes, we get 100 * 2 = 200 pebbles. This is probably not accurate, but it's also probably somewhat useful — the right ballpark.

Not accurate, but close enough. I've heard of open-ended Qs like "how much would you charge to clean all the windows in NYC". This question is asking you to make assumptions and to come up with some pseudo-algorithm to approximate an answer.

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 is given list of cities and distances between, what is the shortest path visiting every city once and returning to start? Actually extremely long to find true answer, but heuristic can find approximate answer much quicker.

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?

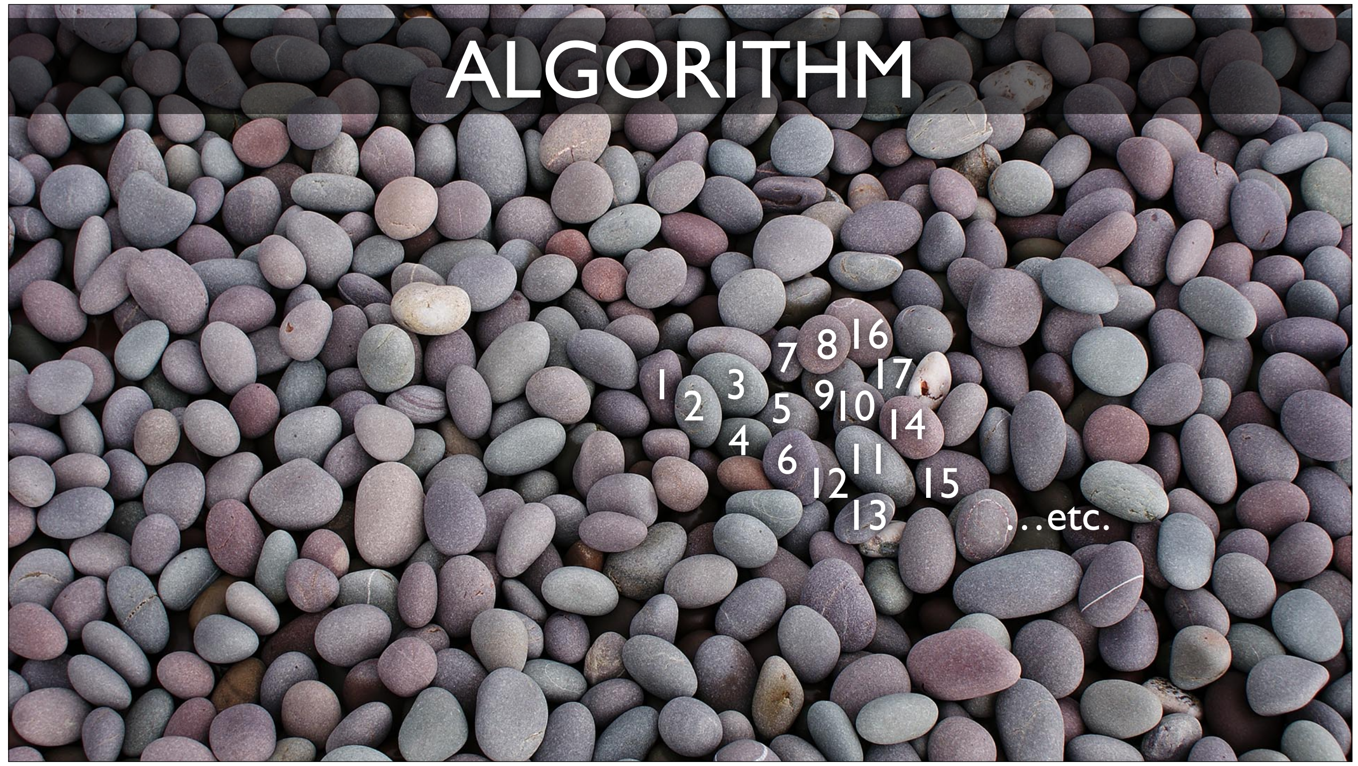
Arriving

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

In this case, instead of distance we are analyzing it by cost

4 cities would give us 24 options
5 cities is 120 options



Algorithm: count the pebbles.

Get’s a precise answer, but it takes more time than our heuristic approach

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**

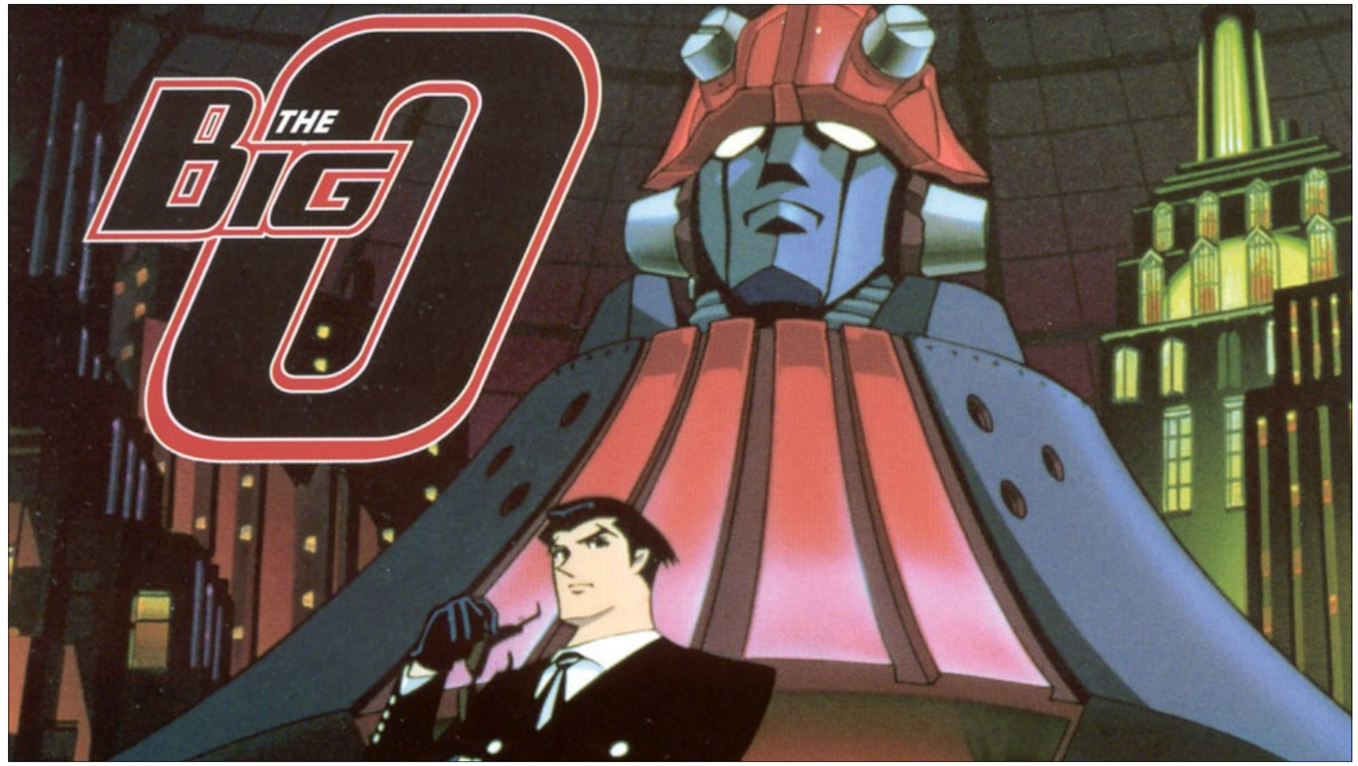
Often we loosely call functions algorithms, because much of the time a function is implementing an algorithm.

So what makes something an 'algorithm'?

Step-by-step/deterministic - repeatable steps always give us the same result
Complete + Finite = 'Decidable' - we can get an answer, and do it in a finite amount of time
Efficient - not *necessary* but always a goal of an algorithm. We should strive to do the minimum work to arrive at an answer.
Correct - this is why its not a heuristic - it's only a (true) algorithm if it gives an exact correct result for all inputs

Downside to the need for correctness - some problems are easier than others

How can we compare algorithms?



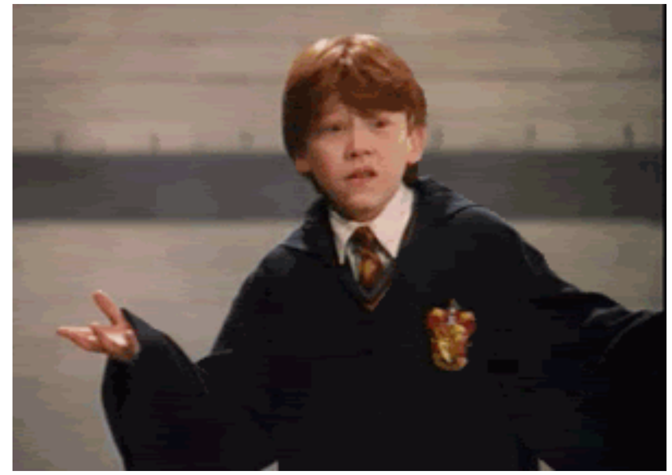
The Big O was a 1999 cartoon about a big robot

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)

Short definition - I've highlighted the two important concepts

Big O is not a measure of real time - that's called benchmarking or profiling - Big O doesn't measure real time, it measures what we call *time complexity* - it's an abstract measurement



What?

Okay, even in plain english, it’s still kind of elusive. Let’s try to make it more concrete

Example 1

```javascript
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
}
// O(1 + 1 + 1 + 1) = O(4) = O(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
}
```

```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

```javascript
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])
    }
  }
}
```

```javascript
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
}
```

```javascript
// 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?***

Warning, going to bring up the blackboard for a moment…

Typically, in high school math, we usually assumed with logs with base 10 (called the common logarithm) or base e (2.71828…), which is also called the natural logarithm. In computer science, we frequently assume base 2, because we're particularly interested in operations and data structures that allow us to cut time in half, like BSTs.

$$\log_2(2) = 1$$

$$\log_2(4) = 2$$

$$\log_2(8) = 3$$

$$\log_2(5) = 2.32192809489$$

Here’s how they relate - for example, we have an tree with 8 elements, and we want to find one node in it, in the worst case we’ll have to make 3 jumps in order find that node

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 $10 \cdot n$)

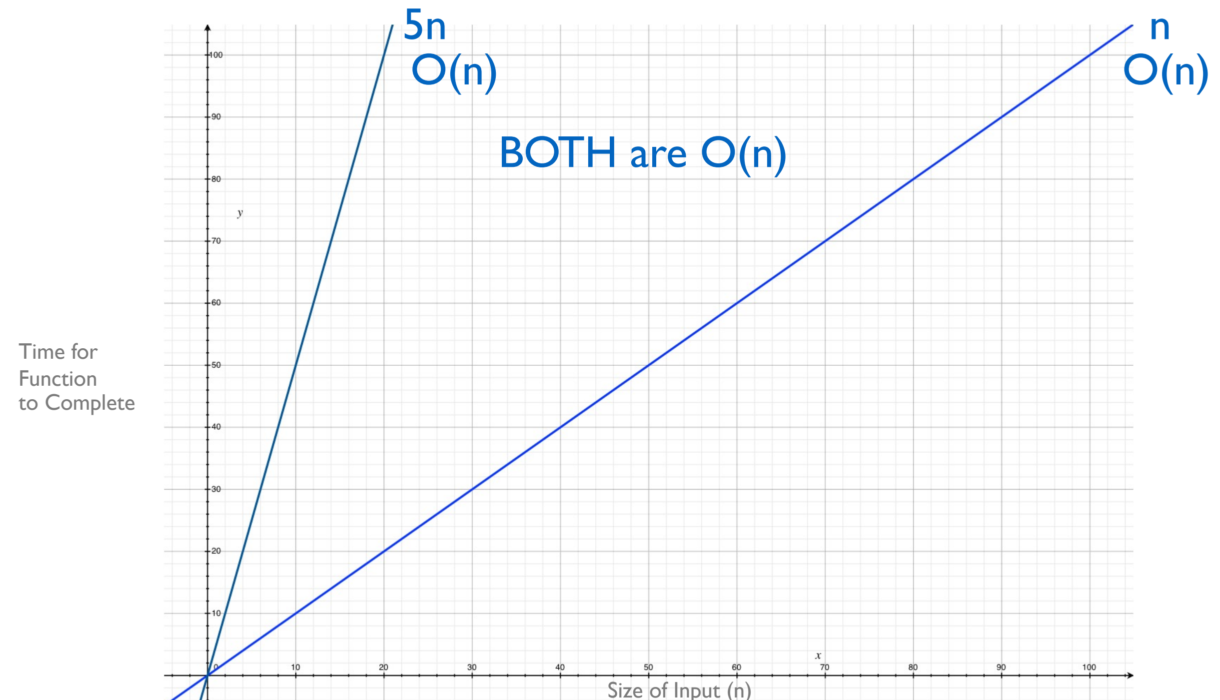
Warning, this is going to get a little mathy, as it is comes from math, and is a way of analyzing functions.

comparitive, can compare big O of one algorithm to another. provides a standard.
shape - we want to see the performance as a function of input size, particularly for how the performance changes as the input gets very large. Sorting a list of a million numbers for instance.

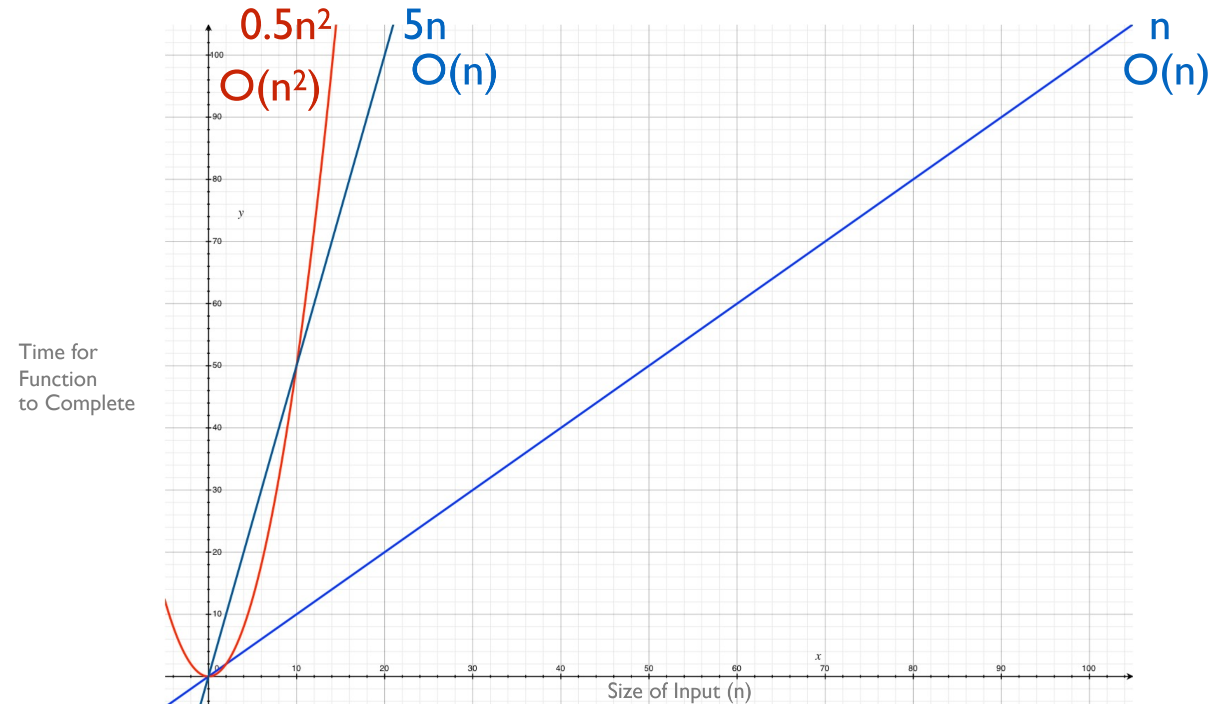
upper-bound - we care about the worst-case

because we care about when N is large, we usually only care about the highest order term

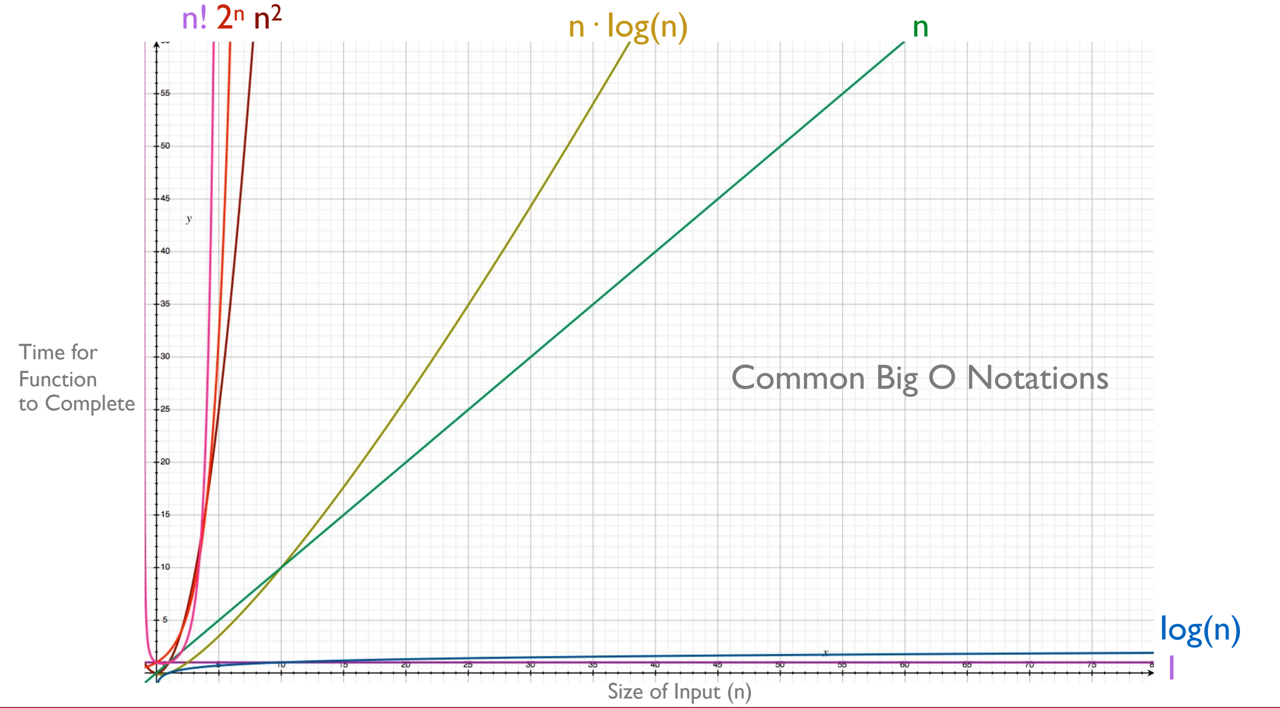
so for function f, it's big O would be O(n^3)



If we were to do a pseudo-measurement of each function, deriving an actual mathematical function to describe each, we might say that the "good" function follows the f(n) = n curve, and the "bad" function follows the f(n) = 5n curve. Even this is not entirely accurate as it doesn't necessarily include the time to add the function to the call stack, initialize variables, invoke `forEach` function, etc; however, we will ignore that for the sake of argument. >>> Main point: BOTH of these functions are classified as O(n). Huh?



Remember, "Big O" is NOT useful when talking about two algorithms which have the same Big O notation. Indeed, one O(n) function may be five times slower than another one, and Big O doesn't say anything about that. But an O(n^2) algorithm will ALWAYS be slower than both — once *n* gets high enough! No matter the constants! So Big O is about *classifying* similar algorithms. Both O(n) algorithms are linear — double the input size, double the time they take. Any linear function is eventually better than any quadratic function, for big enough n.


n! 2n n2
n · 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		

Suppose a computer can do one algorithm step ("operation") in one nanosecond (billionth of a second). If your algorithm f(n) has an input n of a given size, this chart shows how long it would take to perform that algorithm for common time complexities.

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 · 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

let's look at out examples

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)