



Week 10: Analysis of Algorithms: A Brief Introduction

CS 220: Software Design II — D. Mathias

Your programming journey thus far:

“Please just let this program work.”

(and maybe do so elegantly)

Collecting Data

Recent technological advancements are enabling greater data collection
ubiquitous computing: computers/sensors that are everywhere



There is a lot of data to store and process.

Take CS 364 (Databases) to learn more about this!

Other Program Considerations

Consider a piece of software or website you like to use. Would you use it if...

- ...it didn't do what you expected it to do?

- ...it prevented your computer from doing anything else while it was running?

- ...it took many seconds (or even minutes! hours! days!) to complete an action?

Programs that work also need to be usable:

- memory-efficient

- execute quickly

Algorithm Analysis

algorithm analysis: determining resources necessary to execute an algorithm

*** typically consider the *worst-case* scenario ***

Resources considered:

space (i.e., memory)

time (i.e., speed)

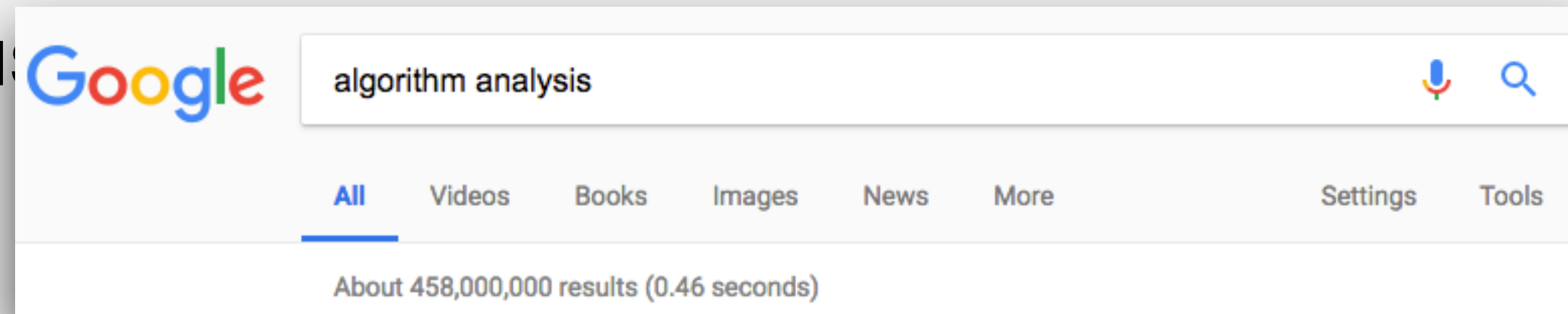
Memory is now very cheap and plentiful; speed is the bottleneck

Why Speed?

Directly affects a user's experience/satisfaction

unhappy users don't continue to use software (unless forced to)

if a program is unresponsive
problematic



speed can become

Makes a difference in the performance of critical applications

e.g., self-driving cars need to analyze many inputs and make quick decisions

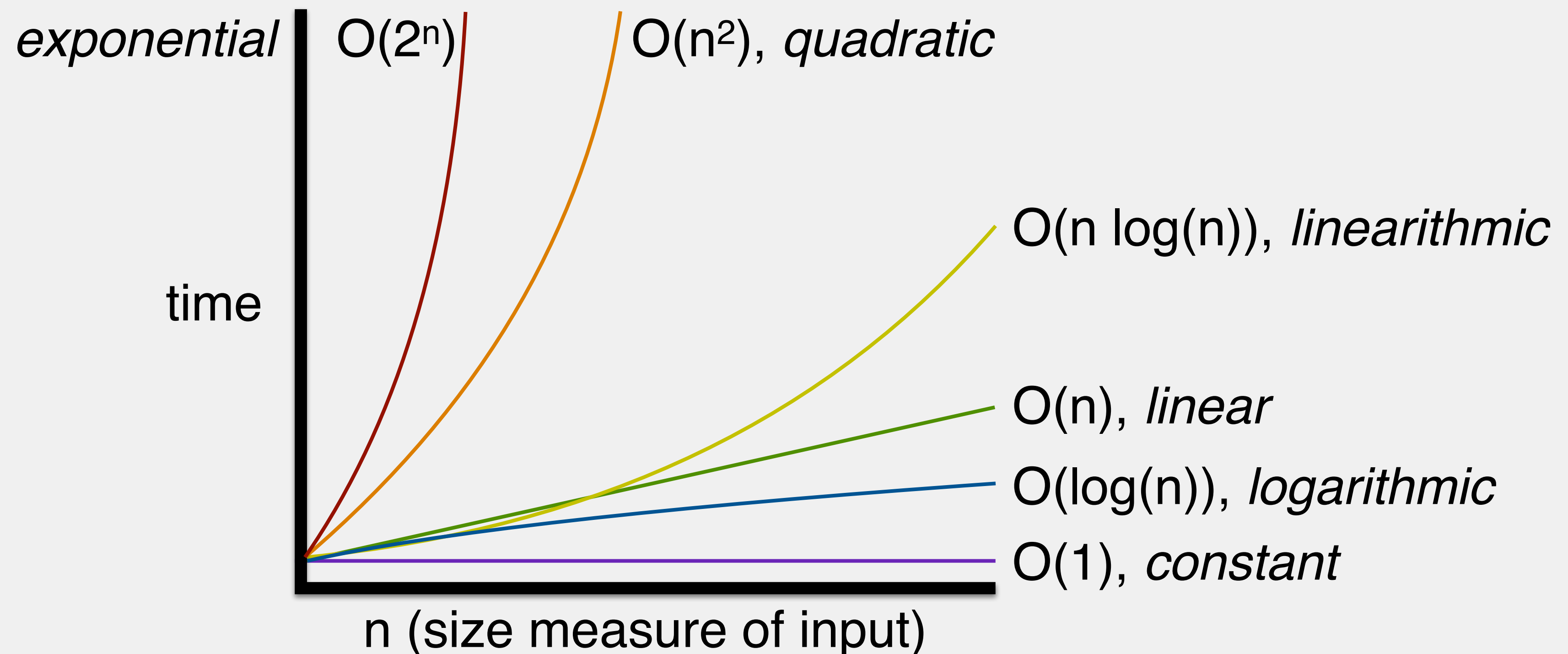
Determines whether a problem is solvable by a computer within our lifetime¹

¹: https://en.wikipedia.org/wiki/NP-completeness#NP-complete_problems

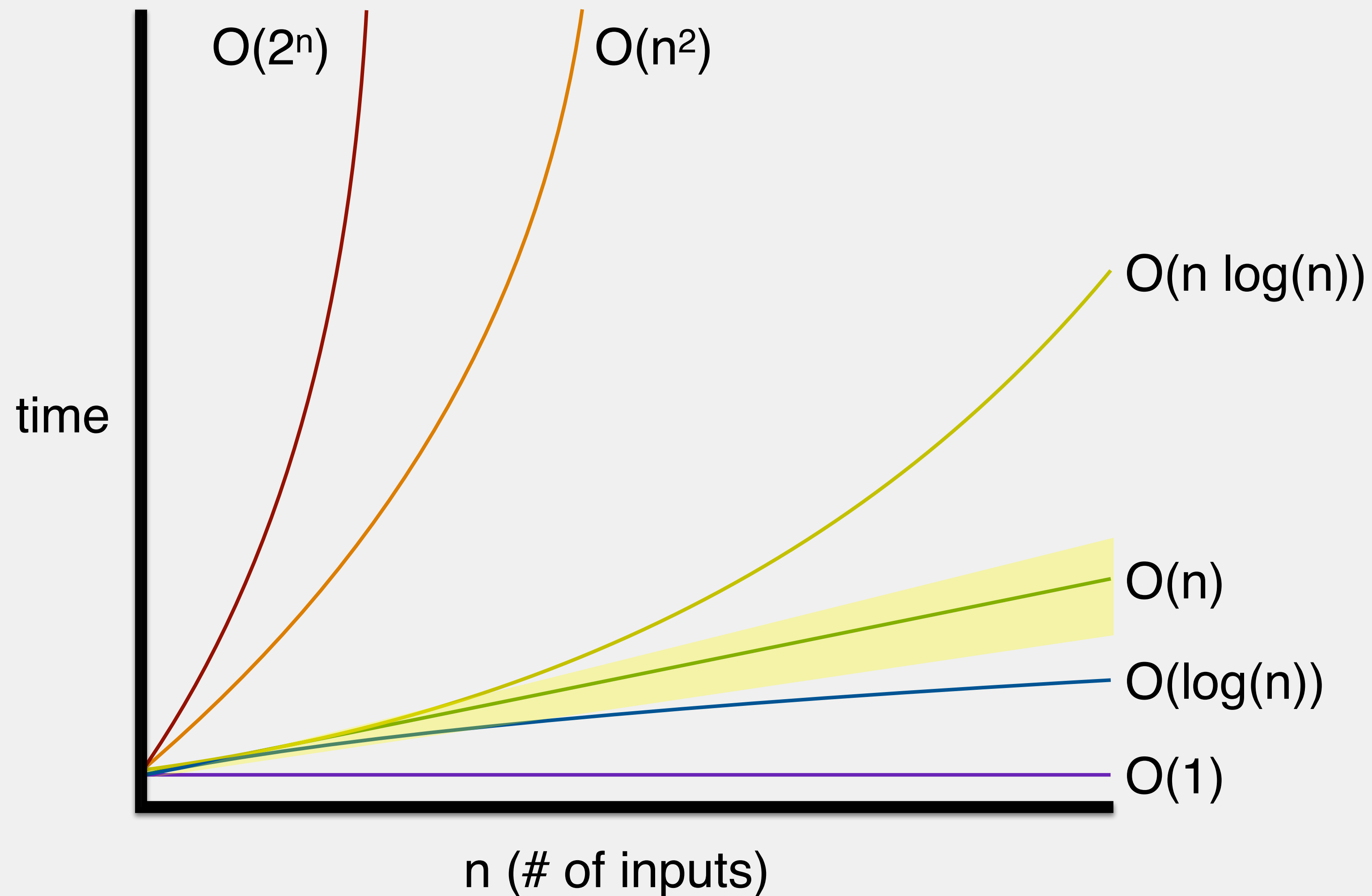
Big O Notation

big O notation: mathematical notation to characterize the speed of an algorithm as a function of the size of its input (i.e., *runtime* or *computational complexity*)

i.e., will more data mean my algorithm takes longer to run? how much longer?



Big O Notation

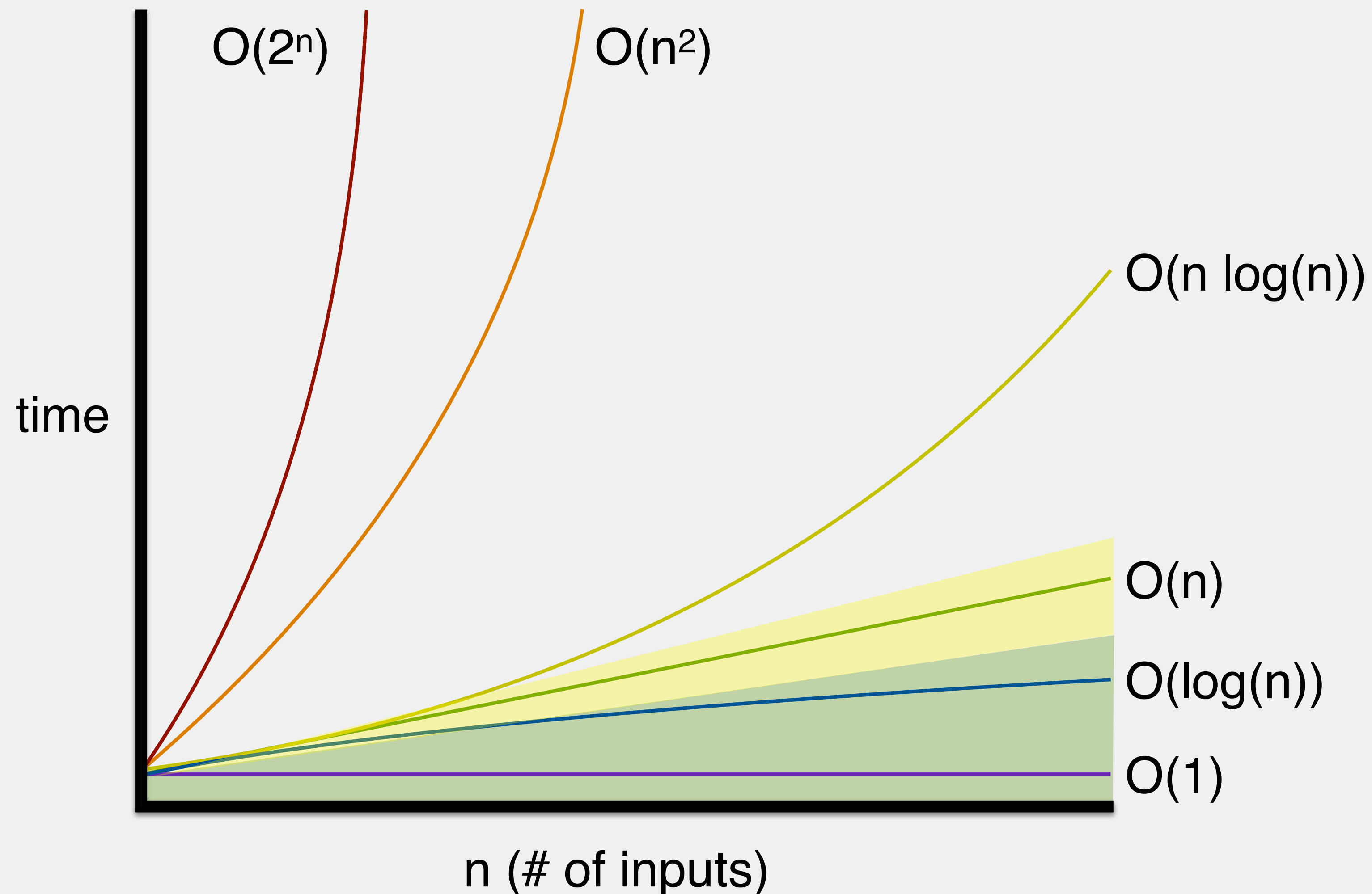


Every input we add will proportionally increase the amount of time the algorithm runs

e.g., if an input of size 1 takes 3 units of time, an input of size 2 will take 6 units of time

Not too bad!

Big O Notation



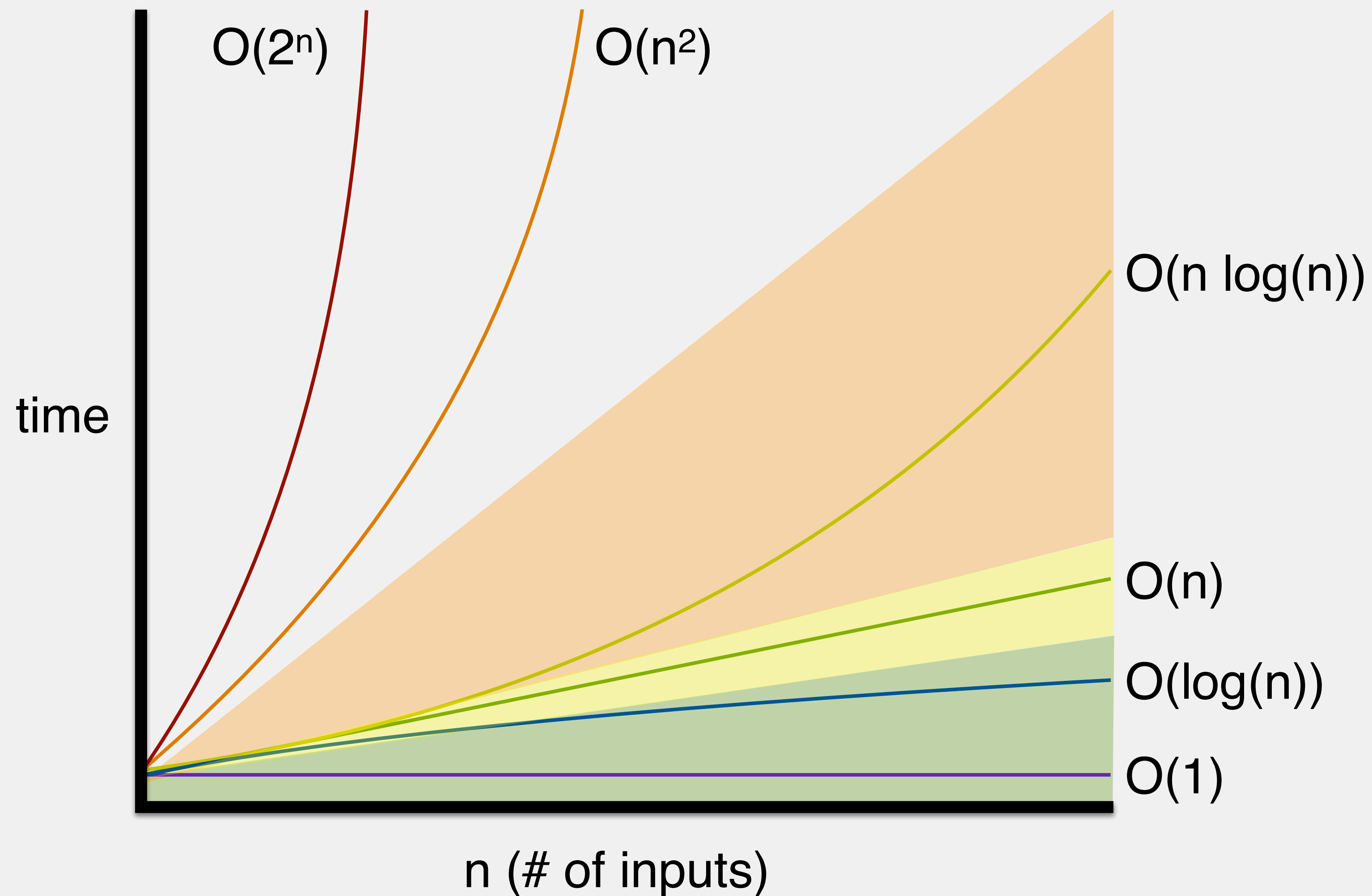
Every input we add will increase the time necessary to run the algorithm, but not by much

$O(\log(n))$ = slight increase for each input, but not proportional like $O(n)$

$O(1)$ = no increase at all!

Great!

Big O Notation

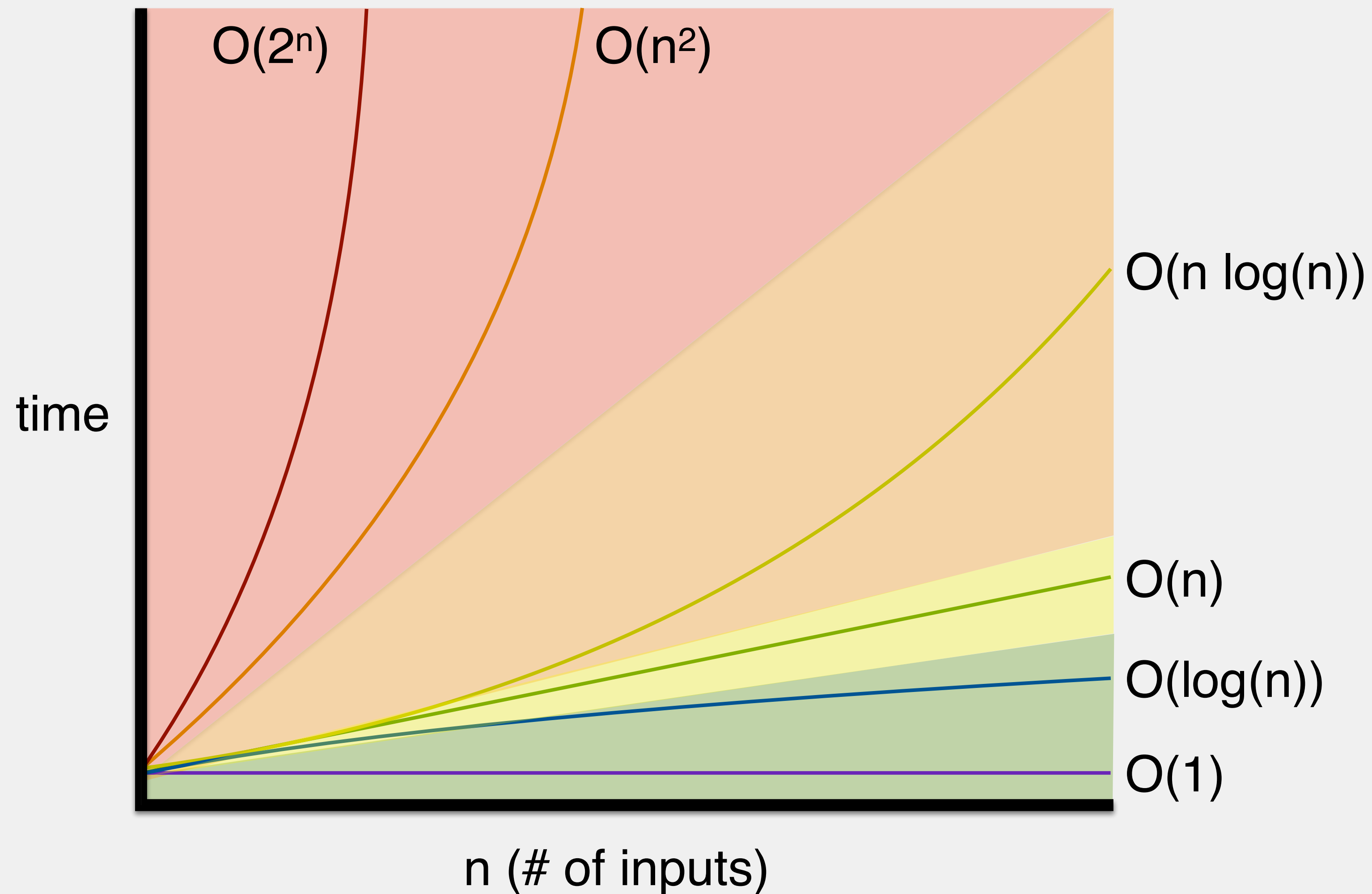


Every input will more than proportionally increase the time to run the algorithm

e.g., if an input of size 1 takes 3 units of time, an input of size 2 might take 7 or 8 units of time, size 10 might take 80 units of time

Is this good? — It depends on the problem

Big O Notation



Every additional input will drastically increase the runtime (more than double!)

e.g., if an input of size 1 takes 3 units of time, an input of size 2 might take 20 units of time

Is this good? — It depends on the problem.

Really Slow Algorithms

Surprising number of problems that **seem** easy to solve are actually **very difficult** for computers to solve

might take millions of years of computational time!

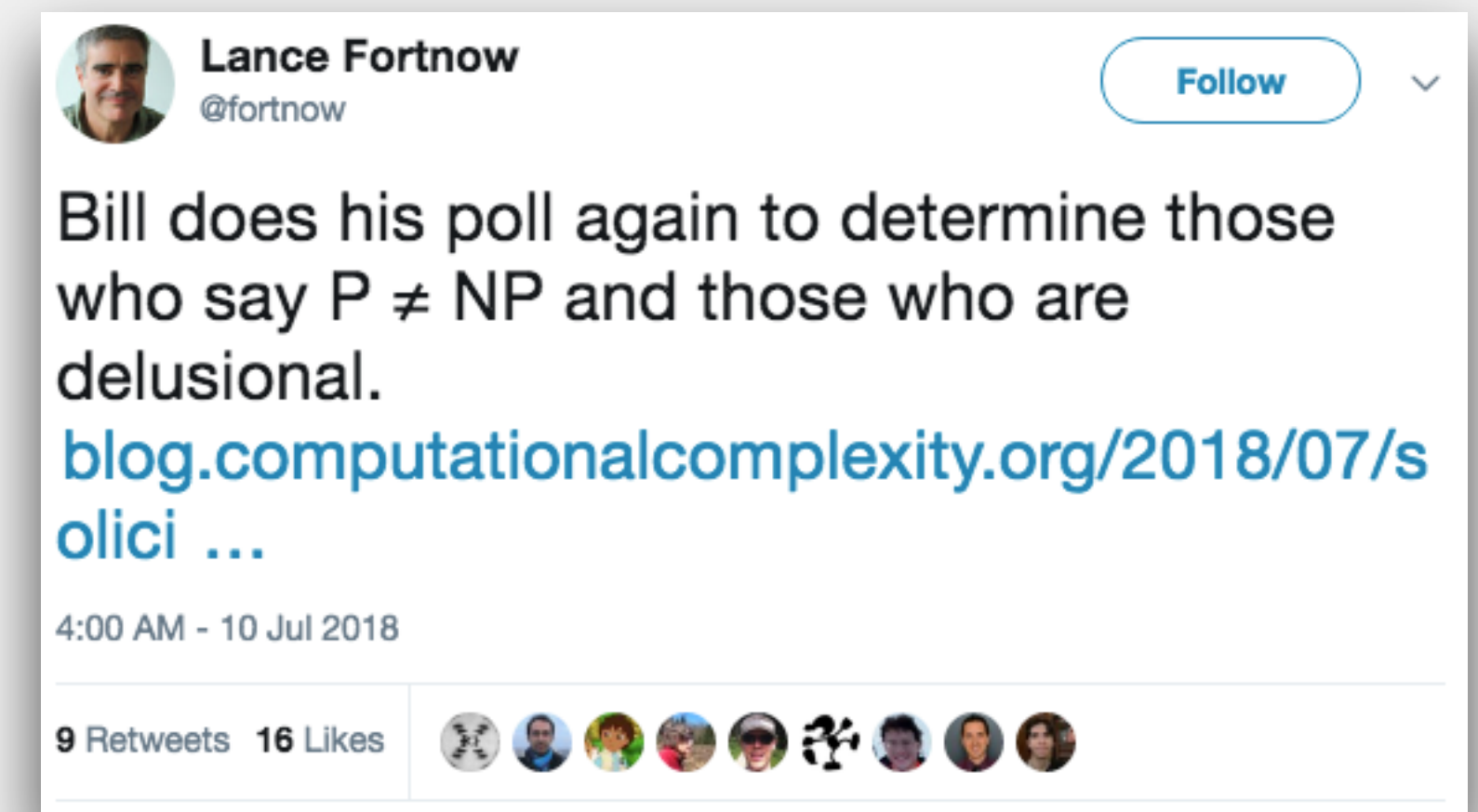
Part of the focus of the question “does $P = NP$?”

P = problems that are relatively easy for computers to solve

NP = problems that computers can easily verify a solution to, but are not easy for computers to solve; we can't yet prove definitively that computers cannot solve them in a reasonable amount of time

1: <https://medium.com/@niruhan/p-vs-np-problem-8d2b6fc2b697>

2: <https://medium.com/omarelgabrys-blog/the-big-scary-o-notation-ce9352d827ce>



Anatomy of an Algorithm

algorithm: a segment of code that solves some problem

- calculating a person's age

- searching for a number in an array

- sorting numbers in an array

All code, including an algorithm, is made up of *statements*, units of instruction

Example: Algorithm

```
int a, b;  
double c;  
  
a = 5;  
b = 3;  
c = (a * a) + (b * b);  
c = Math.sqrt(c);
```

Each atomic operation (i.e., an operation that cannot be broken down further) is a single unit of work

Examples:

- declaration

- assignment

- casting

- mathematical operations

- return

Example: Algorithm

```
int a, b;           // 2
double c;           // 1

a = 5;              // 1
b = 3;              // 1
c = (a * a) + (b * b); // 4
c = Math.sqrt(c);   // ?
```

Each atomic operation (i.e., an operation that cannot be broken down further) is a single unit of work

Examples:

- declaration

- assignment

- casting

- mathematical operations

- return

Example: Algorithm

```
int a, b;           // 2
double c;           // 1

a = 5;              // 1
b = 3;              // 1
c = (a * a) + (b * b); // 4
// c = Math.sqrt(c); // ?

// 2 + 1 + 1 + 1 + 4 = 9 = O(9)
```

Add together the runtime of sequential statements

Simplifying Analysis

```
int a, b;           // 2
double c;           // 1

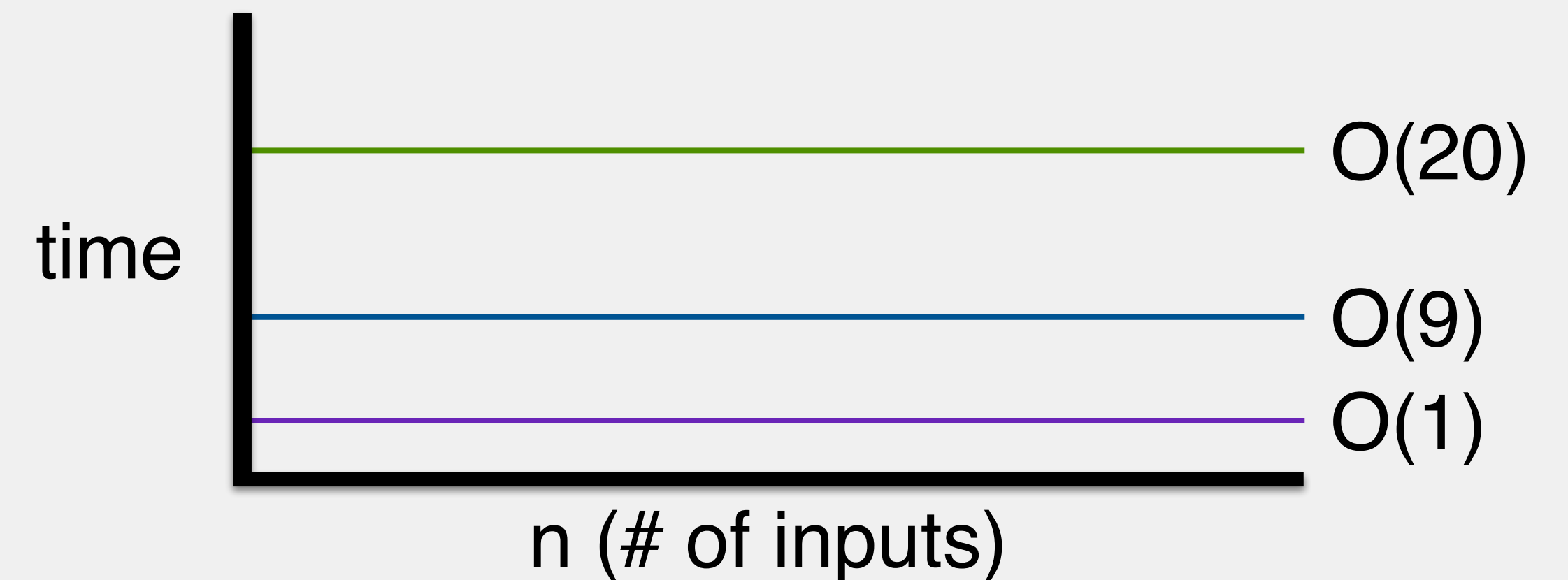
a = 5;              // 1
b = 3;              // 1
c = (a * a) + (b * b); // 4
// c = Math.sqrt(c); // ?

// 2 + 1 + 1 + 1 + 4 = 9 = O(9)
```

Don't want to count out every line

is there a difference, practically, between $O(1)$, $O(9)$, $O(20)$?

We want to generally describe the shape (i.e., *trend*) of the graph



Simplifying Analysis

```
int a, b;           // 2
double c;           // 1

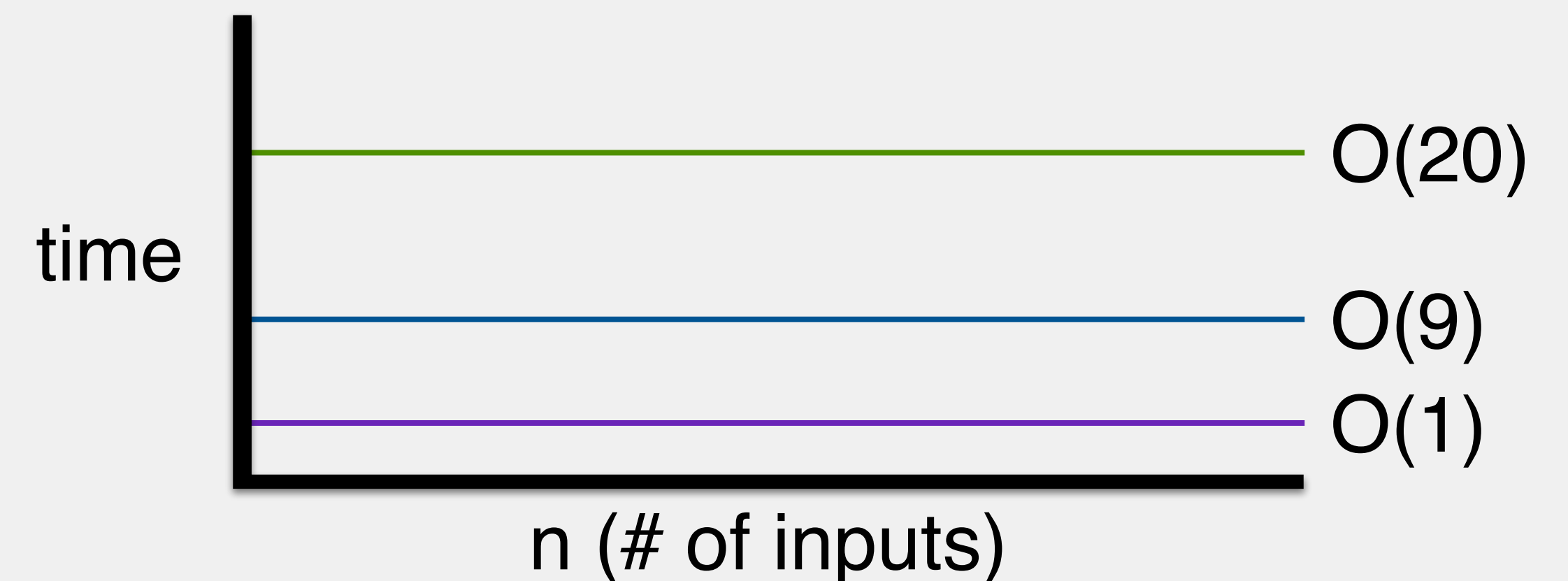
a = 5;              // 1
b = 3;              // 1
c = (a * a) + (b * b); // 4
// c = Math.sqrt(c); // ?

// 2 + 1 + 1 + 1 + 4 = 9 = O(1)
```

Don't want to count out every line

is there a difference, practically, between $O(1)$, $O(9)$, $O(20)$?

We want to generally describe the shape (i.e., *trend*) of the graph



O(1) Algorithm

```
int a, b;  
double c;  
  
a = 5;  
b = 3;  
c = (a * a) + (b * b);  
// c = Math.sqrt(c);
```

Algorithm made up entirely of atomic operations

O(n) Algorithm

```
int[] grades = // instantiate and fill new array  
for (int i = 0; i < grades.length; i++) {  
    System.out.print(grades[i] + ", ");  
}
```

Loop runtimes are calculated by:

1. calculating the runtime of everything **inside** the loop
2. multiplying that value by the number of times the loop runs (i.e., number of inputs, n)

O(n) Algorithm

```
int[] grades = // 1 to instantiate, n to fill  
for (int i = 0; i < grades.length; i++) { // n  
    System.out.print(grades[i] + ", "); // 1  
}  
  
// 1 + n + n * 1  
// 1 + n + n  
// 2n + 1
```

Loop runtimes are calculated by:

1. calculating the runtime of everything **inside** the loop
2. multiplying that value by the number of times the loop runs (i.e., number of inputs, n)

Simplifying Analysis

```
int[] grades = // 1 to instantiate, n to fill
for (int i = 0; i < grades.length; i++) { // n
    System.out.print(grades[i] + ", "); // 1
}

// 1 + n + n * 1
// 1 + n + n
// 2n + 1
```

Drop all the terms **except** the most expensive one

$2n$ (linear) is more expensive than 1 (constant)

Simplifying Analysis

```
int[] grades = // 1 to instantiate, n to fill
for (int i = 0; i < grades.length; i++) { // n
    System.out.print(grades[i] + ", "); // 1
}

// 1 + n + n * 1
// 1 + n + n
// 2n + 1
// 2n
```

Drop all the terms **except** the most expensive one

$2n$ (linear) is more expensive than 1 (constant)

Simplifying Analysis

```
int[] grades = // 1 to instantiate, n to fill
for (int i = 0; i < grades.length; i++) { // n
    System.out.print(grades[i] + ", "); // 1
}

// 1 + n + n * 1
// 1 + n + n
// 2n + 1
// 2n
```

Drop all the terms **except** the most expensive one

$2n$ (linear) is more expensive than 1 (constant)

Remove the constant multiplier

remember, we're looking for the **trend** of the graph

Simplifying Analysis

```
int[] grades = // 1 to instantiate, n to fill
for (int i = 0; i < grades.length; i++) { // n
    System.out.print(grades[i] + ", "); // 1
}

// 1 + n + n * 1
// 1 + n + n
// 2n + 1
// 2n
// O(n)
```

Drop all the terms **except** the most expensive one

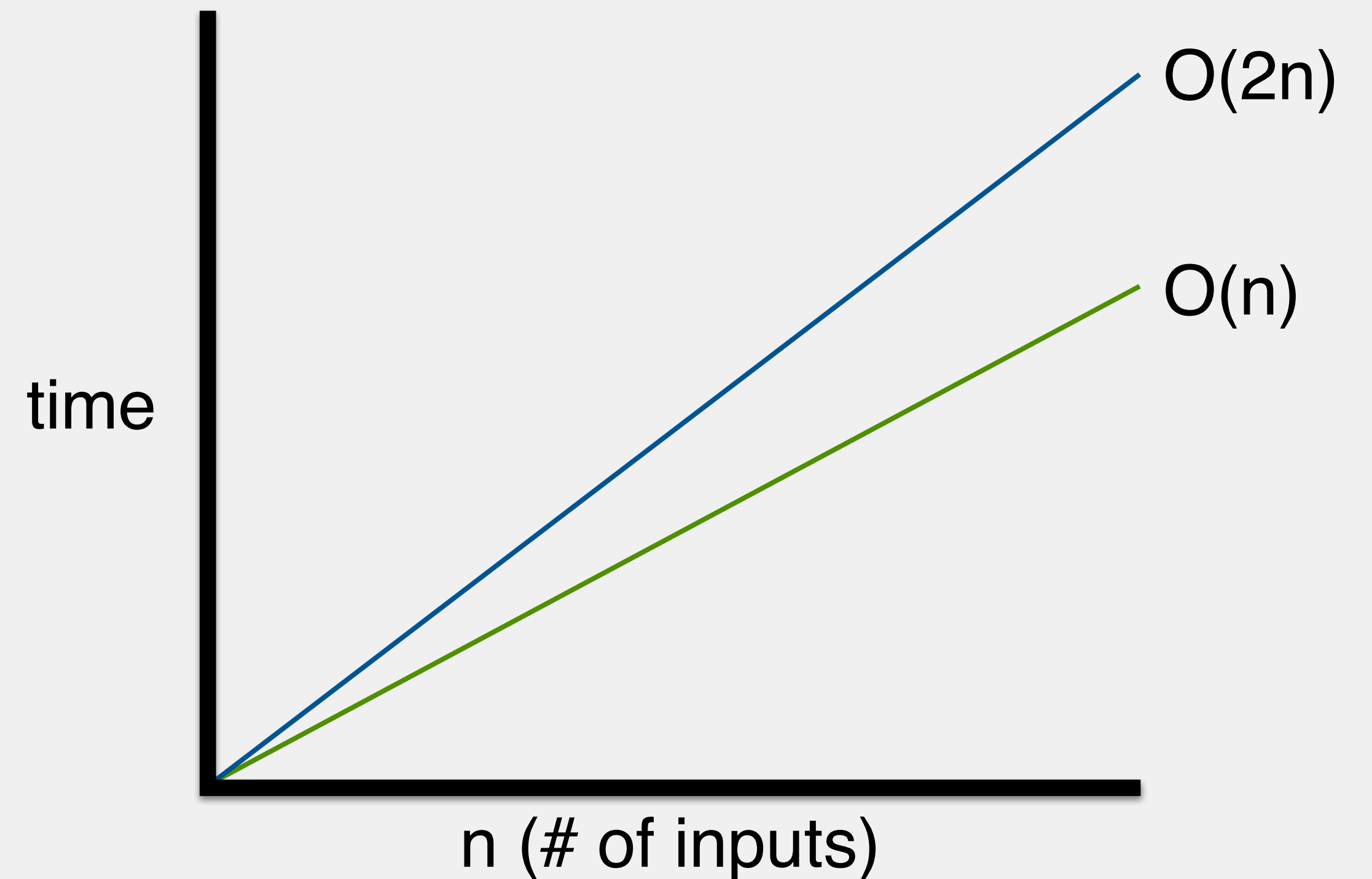
2n (linear) is more expensive than 1 (constant)

Remove the constant multiplier

remember, we're looking for the **trend** of the graph

Simplifying Analysis

```
int[] grades = // 1 to instantiate, n to fill  
  
for (int i = 0; i < grades.length; i++) { // n  
    System.out.print(grades[i] + ", "); // 1  
}  
  
// 1 + n + n * 1  
// 1 + n + n  
// 2n + 1  
// 2n  
// O(n)
```



$O(n^2)$ Algorithm

```
int[][] grades = // instantiate and fill new 2D array

for (int row = 0; row < grades.length; row++) {
    for (int col = 0; col < grades[row].length; col++) {
        System.out.print(grades[row][col] + ", ");
    }
    System.out.println();
}
```

Nested loops work like regular loops:

1. start at innermost loop
2. calculate runtime for that loop
3. move to next outer loop, using result from inner loop as input for outer loop
4. repeat until out of loops

$O(n^2)$ Algorithm

```
int[][] grades = // will ignore this for now

// n
for (int row = 0; row < grades.length; row++) {
    // n
    for (int col = 0; col < grades[row].length; col++) {
        // 1
        System.out.print(grades[row][col] + ", ");
    }
    // 1
    System.out.println();
}
```

Nested loops work like regular loops:

1. start at innermost loop
2. calculate runtime for that loop
3. move to next outer loop, using result from inner loop as input for outer loop
4. repeat until out of loops

$O(n^2)$ Algorithm

```
int[][] grades = // will ignore this for now

// n
for (int row = 0; row < grades.length; row++) {
    // n
    for (int col = 0; col < grades[row].length; col++) {
        // 1
        System.out.print(grades[row][col] + ", ");
    }
    // 1
    System.out.println();
}

// n * 1 = n - inner loop
```

Nested loops work like regular loops:

1. start at innermost loop
2. calculate runtime for that loop
3. move to next outer loop, using result from inner loop as input for outer loop
4. repeat until out of loops

$O(n^2)$ Algorithm

```
int[][] grades = // will ignore this for now

// n
for (int row = 0; row < grades.length; row++) {
    // n
    for (int col = 0; col < grades[row].length; col++) {
        // 1
        System.out.print(grades[row][col] + ", ");
    }
    // 1
    System.out.println();
}

// n - inner loop
// n * (n + 1) - outer loop
//  $n^2 + n$ 
```

Nested loops work like regular loops:

1. start at innermost loop
2. calculate runtime for that loop
3. move to next outer loop, using result from inner loop as input for outer loop
4. repeat until out of loops

Simplifying Analysis

```
int[][] grades = // will ignore this for now

// n
for (int row = 0; row < grades.length; row++) {
    // n
    for (int col = 0; col < grades[row].length; col++) {
        // 1
        System.out.print(grades[row][col] + ", ");
    }
    // 1
    System.out.println();
}

// n - inner loop
// n * (n + 1) - outer loop
// n2 + n
```

Drop all the terms **except** the most expensive one

n^2 (quadratic) is more expensive than n (linear)

Simplifying Analysis

```
int[][] grades = // will ignore this for now

// n
for (int row = 0; row < grades.length; row++) {
    // n
    for (int col = 0; col < grades[row].length; col++) {
        // 1
        System.out.print(grades[row][col] + ", ");
    }
    // 1
    System.out.println();
}

// n - inner loop
// n * (n + 1) - outer loop
//  $n^2 + n$ 
//  $O(n^2)$ 
```

Drop all the terms **except** the most expensive one

n^2 (quadratic) is more expensive than n (linear)

Calculating Runtime Complexity

General rules

1. add up the runtime associated with sequential statements
2. reduce to the highest order term
3. remove any **constant** coefficients (e.g., 2, 3)

Remember, the big O associated with an algorithm is **not** an exact number of instructions! It describes the trend of the algorithm

Exercise: Calculating Runtime Complexity

exercise 1

```
for (int r = 0; r < arr.length; r++) {  
    for (int c = 0; c < arr[r].length; c++) {  
        arr[r][c] = r * c;  
    }  
}  
  
for (int r = 0; r < arr.length; r++) {  
    System.out.print(arr[r][0] + " ");  
}
```

exercise 2

```
if (r % 2 == 0) {  
    arr[r] = r * 2;  
}  
  
else {  
    int i = arr.length;  
    while (i >= 0) {  
        arr[r] *= arr[i];  
    }  
}
```

Exercise: Calculating Runtime Complexity

exercise 1

```
for (int r = 0; r < arr.length; r++) {  
    for (int c = 0; c < arr[r].length; c++) {  
        arr[r][c] = r * c;  
    }  
}  
  
for (int r = 0; r < arr.length; r++) {  
    System.out.print(arr[r][0] + " ");  
}
```

exercise 2

```
if (r % 2 == 0) {  
    arr[r] = r * 2;  
}  
  
else {  
    int i = arr.length;  
    while (i >= 0) {  
        arr[r] *= arr[i];  
    }  
}
```

Exercise: Calculating Runtime Complexity

exercise 1

```
// n
for (int r = 0; r < arr.length; r++) {
    // n
    for (int c = 0; c < arr[r].length; c++) {
        // 2
        arr[r][c] = r * c;
    }
}

// n
for (int r = 0; r < arr.length; r++) {
    // 1
    System.out.print(arr[r][0] + " ");
}
}
```

exercise 2

```
if (r % 2 == 0) {
    arr[r] = r * 2;
}

else {

    int i = arr.length;

    while (i >= 0) {

        arr[r] *= arr[i];
    }
}
```

Exercise: Calculating Runtime Complexity

exercise 1

```
// n
for (int r = 0; r < arr.length; r++) {
    // n
    for (int c = 0; c < arr[r].length; c++) {
        // 2
        arr[r][c] = r * c;
    }
}

// n
for (int r = 0; r < arr.length; r++) {
    // 1
    System.out.print(arr[r][0] + " ");
}

// (n * n * 2) + (n * 1) = 2n2 + n = O(n2)
```

exercise 2

```
if (r % 2 == 0) {
    arr[r] = r * 2;
}

else {

    int i = arr.length;

    while (i >= 0) {

        arr[r] *= arr[i];
    }
}
```

Exercise: Calculating Runtime Complexity

exercise 1

```
// n
for (int r = 0; r < arr.length; r++) {
    // n
    for (int c = 0; c < arr[r].length; c++) {
        // 2
        arr[r][c] = r * c;
    }
}

// n
for (int r = 0; r < arr.length; r++) {
    // 1
    System.out.print(arr[r][0] + " ");
}

// (n * n * 2) + (n * 1) = 2n2 + n = O(n2)
```

exercise 2

```
if (r % 2 == 0) {
    arr[r] = r * 2;
}

else {

    int i = arr.length;

    while (i >= 0) {

        arr[r] *= arr[i];
    }
}
```


Exercise: Calculating Runtime Complexity

exercise 1

```
// n
for (int r = 0; r < arr.length; r++) {
    // n
    for (int c = 0; c < arr[r].length; c++) {
        // 2
        arr[r][c] = r * c;
    }
}

// n
for (int r = 0; r < arr.length; r++) {
    // 1
    System.out.print(arr[r][0] + " ");
}

// (n * n * 2) + (n * 1) = 2n2 + n = O(n2)
```

exercise 2

```
// 4
if (r % 2 == 0) {
    arr[r] = r * 2;
}
// 0
else {
    // 1
    int i = arr.length;
    // n
    while (i >= 0) {
        // 2
        arr[r] *= arr[i];
    }
}
```

Exercise: Calculating Runtime Complexity

exercise 1

```
// n
for (int r = 0; r < arr.length; r++) {
    // n
    for (int c = 0; c < arr[r].length; c++) {
        // 2
        arr[r][c] = r * c;
    }
}

// n
for (int r = 0; r < arr.length; r++) {
    // 1
    System.out.print(arr[r][0] + " ");
}

// (n * n * 2) + (n * 1) = 2n2 + n = O(n2)
```

exercise 2

```
// 4
if (r % 2 == 0) {
    arr[r] = r * 2;
}
// 0
else {
    // 1
    int i = arr.length;
    // n
    while (i >= 0) {
        // 2
        arr[r] *= arr[i];
    }
}

// 2 * n + 1 = 2n + 1 = O(n)
```

Calculating Runtime Complexity

General rules

~~1. add up the runtime associated with sequential statements~~

1. focus on the loops

if there are loops, those are always your most expensive terms

2. reduce to the highest order term

3. remove any **constant** coefficients (e.g., 2, 3)

Remember, the big O associated with an algorithm is **not** an exact number of instructions! It describes the trend of the algorithm.

Best- vs Worst-Case Scenario

```
int[] arr = // instantiate and fill new array
int num = // number we are searching for

for (int i = 0; i < arr.length; i++) {
    if (arr[i] == num) {
        return i;
    }
}
```

What is the best-case scenario for this particular array?

What is the worst-case scenario?

What are the runtimes for each?

arr (int[])

5	13	0	4	27	8
---	----	---	---	----	---

Best- vs Worst-Case Scenario

```
int[] arr = // instantiate and fill new array
int num = // number we are searching for

for (int i = 0; i < arr.length; i++) {
    if (arr[i] == num) {
        return i;
    }
}
```

What is the best-case scenario for this particular array?

What is the worst-case scenario?

What are the runtimes for each?

arr (int[])

5	13	0	4	27	8
---	----	---	---	----	---

num (int)

5

Best- vs Worst-Case Scenario

```
int[] arr = // instantiate and fill new array
int num = // number we are searching for

for (int i = 0; i < arr.length; i++) {
    if (arr[i] == num) {
        return i;
    }
}
```

What is the best-case scenario for this particular array?

What is the worst-case scenario?

What are the runtimes for each?

arr (int[])

5	13	0	4	27	8
---	----	---	---	----	---

num (int)

10

Best- vs Worst-Case Scenario

```
int[] arr = // instantiate and fill new array
int num = // number we are searching for

for (int i = 0; i < arr.length; i++) {
    if (arr[i] == num) {
        return i;
    }
}
```

What is the best-case scenario for this particular array?

What is the worst-case scenario?

What are the runtimes for each?

arr (int[])					
0	60	75	83	92	99

num (int)
99

Why Is This All Important?

The ability to make informed choices between different algorithms/data structures relies on the ability to...

- ...understand the data you have and how it is organized (if at all)

- ...understand the best and worst case scenarios for accessing that data

Doing this well enables you to write more efficient programs!

Array Lists vs Linked Lists

Seemingly has identical functionality due to the `Collection` interface

i.e., adding in an array list and linked list will both add the value in the same place

Runtime of these operations depends on the data structure

Exercise: Runtime Analysis

Fill in the following chart with worst-case runtimes

assume array list never needs to grow/shrink as part of the calculations

access of a position in an array is $O(1)$

	array list	singly linked list	doubly linked list
<i>add value to beginning</i>			
<i>add value to end</i>			
<i>remove value at beginning</i>			
<i>remove value at end</i>			
<i>search for value (best case)</i>			
<i>search for value (worst case)</i>			
<i>access value at position 0</i>			
<i>access value at position n</i>			

Exercise: Runtime Analysis

Fill in the following chart with worst-case runtimes

assume array list never needs to grow/shrink as part of the calculations

access of a position in an array is $O(1)$

	array list	singly linked list	doubly linked list
<i>add value to beginning</i>	$O(n)$	$O(1)$	$O(1)$
<i>add value to end</i>	$O(1)$	$O(n)$	$O(1)$
<i>remove value at beginning</i>	$O(n)$	$O(1)$	$O(1)$
<i>remove value at end</i>	$O(1)$	$O(n)$	$O(1)$
<i>search for value (best case)</i>	$O(1)$	$O(1)$	$O(1)$
<i>search for value (worst case)</i>	$O(n)$	$O(n)$	$O(n)$
<i>access value at position 0</i>	$O(1)$	$O(1)$	$O(1)$
<i>access value at position n</i>	$O(1)$	$O(n)$	$O(1)$

Exercise: Runtime Analysis

Fill in the following chart with runtimes for lists of some arbitrarily long length

assume array list never needs to grow/shrink as part of the calculations

access of a position in an array is $O(1)$

	array list	singly linked list	doubly linked list
<i>add value to beginning</i>	$O(n)$	$O(1)$	$O(1)$
<i>add value to end</i>	$O(1)$	$O(n)$	$O(1)$
<i>remove value at beginning</i>	$O(n)$	$O(1)$	$O(1)$
<i>remove value at end</i>	$O(1)$	$O(n)$	$O(1)$
<i>search for value (best case)</i>	$O(1)$	$O(1)$	$O(1)$
<i>search for value (worst case)</i>	$O(n)$	$O(n)$	$O(n)$
<i>access value at position 0</i>	$O(1)$	$O(1)$	$O(1)$
<i>access value at position n</i>	$O(1)$	$O(n)$	$O(1)$