

# Algorithm Efficiency (More formally)

Tiziana Ligorio  
[tligorio@hunter.cuny.edu](mailto:tligorio@hunter.cuny.edu)

# Today's Plan



Midterm Feedback

Recap

Algorithm Efficiency

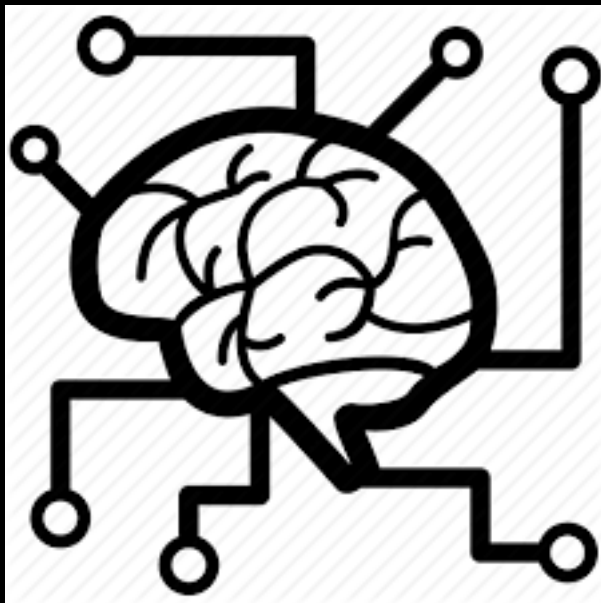
# Announcements

Future projects:

- Not incremental -> Count more
- Expect will take longer



# Midterm Solution



# Midterm Feedback

What is the value of y below?

```
int x = 5;  
int y = x/2;
```

# Midterm Feedback

What is the value of y below?

```
int x = 5;  
int y = x/2;
```

Answer questions with complete and meaningful sentences

*"Arrays can access easier and faster"*

Access what? What does "easier" mean?

*"Arrays have random access thus any array location can be accessed directly in constant time"*

# Midterm Feedback

What is the value of y below?

```
int x = 5;  
int y = x/2;
```

Answer questions with complete and meaningful sentences

*"Arrays can access easier and faster"*

Access what? What does "easier" mean?

*"Arrays have random access thus any array location can be accessed directly in constant time"*

Read and follow instructions carefully (e.g. **write a main function / C++ instructions**)

Max: 106, Mean: 69 (Problematic Questions: 2.4 - 6)

Midterm 20% - Final 40%

Use the midterm to gauge the final exam, asses what you need to work on

# What is CSCI 235?

Programming => Software Analysis and Design

Think like a Computer Scientist:

Design and maintain complex programs

Software Engineering, Abstraction, OOP

Design and represent data and its management

Abstract Data Types

Implement data representation and operations

Data Structures

Algorithms

Understand Algorithm Complexity





# Algorithm Efficiency

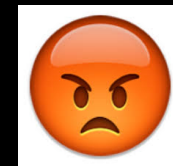
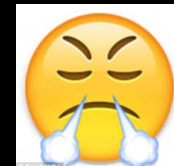
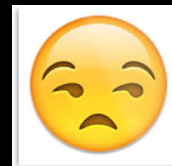
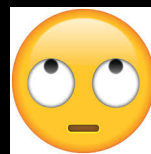
# Scenario 1

You are using an app and suddenly it stalls...  
whatever it is doing it's taking way too long...

# Scenario 1

You are using an app and suddenly it stalls...  
whatever it is doing it's taking way too long...

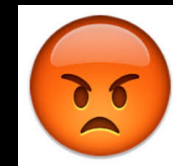
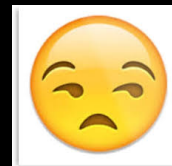
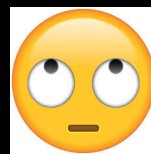
how “long” does that have to be for you to become  
ridiculously frustrated?



# Scenario 1

You are using an app and suddenly it stalls...  
whatever it is doing it's taking way too long...

how "*long*" does that have to be for you to become  
ridiculously frustrated?



... probably not that long

# Scenario 2

At your next super high-end job with the company/research-center of your dreams you are given a very difficult problem to solve

You work hard on it, find a solution, code it up and it works!!!!

Proudly you present it the next day



but...

# Scenario 2

At your next super high-end job with the company/research-center of your dreams you are given a very difficult problem to solve

You work hard on it, find a solution, code it up and it works!!!!

Proudly you present it the next day 

but...

Given some new (large) input it keeps stalling...

Well... sorry but your solution is no good!!!



You need to have a means to estimate/predict the efficiency of your algorithms on **unknown input**.

What is a good solution?

How can we compare solutions  
to a problem? (Algorithms)



# What is a good solution?

Exact

If it's not exact  
it is not a  
solution at all

# What is a good solution?

Exact

Efficient

Time

Space

# What is a good solution?

Exact

Efficient

Time

Space

We are going to  
focus on time

How can we measure time  
efficiency?

How can we measure time  
efficiency?

**Runtime?**

# Problems with actual runtime for comparison

What computer are you using?

Runtime is highly sensitive to hardware

# Problems with actual runtime for comparison

What computer are you using?

Runtime is highly sensitive to hardware

What implementation are you using?

Implementation details may affect runtime but are not reflective of algorithm efficiency

How should we measure  
execution time?



Constant

How should we measure  
execution time?

Number of "steps" or "operations"  
as a function of the size of the input

Varies

```
template<class ItemType>
void List<ItemType>::traverse()
{
    for(Node<ItemType>* ptr = first; ptr != nullptr; ptr = ptr->getNext())
    {
        std::cout << ptr->getItem() << std::endl;
    }
}
```

What are the operations?  
Let  $n$  be the number of nodes

```
template<class ItemType>
void List<ItemType>::traverse()
{
    for(Node<ItemType>* ptr = first; ptr != nullptr; ptr = ptr->getNext())
    {
        std::cout << ptr->getItem() << std::endl;
    }
}
```

What are the operations?  
Let  $n$  be the number of nodes

1 node instantiation and assignment  
upon entering the loop

```
template<class ItemType>
void List<ItemType>::traverse()
{
    for(Node<ItemType>* ptr = first; ptr != nullptr; ptr = ptr->getNext())
    {
        std::cout << ptr->getItem() << std::endl;
    }
}
```

What are the operations?  
Let  $n$  be the number of nodes

```
template<class ItemType>
void List<ItemType>::traverse()
{
    for(Node<ItemType>* ptr = first; ptr != nullptr; ptr = ptr->getNext())
    {
        std::cout << ptr->getItem() << std::endl;
    }
}
```

1 node instantiation and assignment  
upon entering the loop

pointer comparison

What are the operations?  
Let  $n$  be the number of nodes

```
template<class ItemType>
void List<ItemType>::traverse()
{
    for(Node<ItemType>* ptr = first; ptr != nullptr; ptr = ptr->getNext())
    {
        std::cout << ptr->getItem() << std::endl;
    }
}
```

1 node instantiation and assignment  
upon entering the loop

call to getNext ( )

pointer comparison

What are the operations?  
Let  $n$  be the number of nodes

```
template<class ItemType>
void List<ItemType>::traverse()
{
    for(Node<ItemType>* ptr = first; ptr != nullptr; ptr = ptr->getNext())
    {
        std::cout << ptr->getItem() << std::endl;
    }
}
```

1 node instantiation and assignment  
upon entering the loop

call to getNext()

pointer assignment

pointer comparison

What are the operations?  
Let  $n$  be the number of nodes

```
template<class ItemType>
void List<ItemType>::traverse()
{
    for(Node<ItemType>* ptr = first; ptr != nullptr; ptr = ptr->getNext())
    {
        std::cout << ptr->getItem() << std::endl;
    }
}
```

1 node instantiation and assignment  
upon entering the loop

call to getNext ( )

pointer assignment

pointer comparison

call to getItem ( )

write to the console



What are the operations?  
Let  $n$  be the number of nodes

```
template<class ItemType>
void List<ItemType>::traverse()
{
    for(Node<ItemType>* ptr = first; ptr != nullptr; ptr = ptr->getNext())
    {
        std::cout << ptr->getItem() << std::endl;
    }
}
```

$K_0$

1 node instantiation and assignment  
upon entering the loop

$K_1$

call to getNext ( )

$K_2$

pointer assignment

$K_3$

pointer comparison

$K_4$

call to getItem ( )

$K_5$

write to the console

What are the operations?  
Let  $n$  be the number of nodes

```
template<class ItemType>
void List<ItemType>::traverse()
{
    for(Node<ItemType>* ptr = first; ptr != nullptr; ptr = ptr->getNext())
    {
        std::cout << ptr->getItem() << std::endl;
    }
}
```

$K_0$

1 node instantiation and assignment  
upon entering the loop

$K_1$

call to getNext ( )

$K_2$

pointer assignment

$K_3$

pointer comparison

$K_4$

call to getItem ( )

$K_5$

write to the console

$$\text{Operations} = K_0 + n(K_1 + K_2 + K_3 + K_4 + K_5)$$

What are the operations?  
Let  $n$  be the number of nodes

```
template<class ItemType>
void List<ItemType>::traverse()
{
    for(Node<ItemType>* ptr = first; ptr != null; ptr = ptr->getNext())
    {
        std::cout << ptr->getItem() << std::endl;
    }
}
```

$K_0$

1 node instantiation and assignment  
upon entering the loop

$K_1$

call to getNext ( )

$K_2$

pointer assignment

$K_3$

pointer comparison

$K_4$

call to getItem ( )

$K_5$

write to the console

$$\text{Operations} = K_0 + nK_6$$

# Lecture Activity

Identify the steps and write down an expression for execution time

```
bool linearSearch(const std::string& str, char ch)
{
    for (int i = 0; i < str.length(); i++)
    {
        if (str[i] == ch) {
            return true;
        }
    }
    return false;
}
```

# Lecture Activity

Identify the steps and write down an expression for execution time

```
bool linearSearch(const std::string& str, char ch)
{
    for (int i = 0; i < str.length(); i++)
    {
        if (str[i] == ch) {
            return true;
        }
    }
    return false;
}
```



Was this tricky?

n here is the length of the string

```
bool linearSearch(const std::string& str, char ch)
{
    // 1 int assignment upon entering loop
    for (int i = 0; i < str.length(); i++)
    { // call to length() and increment
        if (str[i] == ch) { // Comparisons
            return true; //return operation, maybe
        }
    }
    return false; //return operation, maybe
}
```

n here is the length of the string

```
bool linearSearch(const std::string& str, char ch)
{
    // 1 int assignment upon entering loop
    for (int i = 0; i < std::str.length(); i++)
    { // call to length() and increment
        if (str[i] == ch) { // Comparisons
            return true; //return operation, maybe
        }
    }
    return false; //return operation, maybe
}
```

Maybe stop in  
the middle

Maybe stop at  
end of loop

n here is the length of the string

```
bool linearSearch(const std::string& str, char ch)
{
    for (int i = 0; i < str.length(); i++)
    {
        if (str[i] == ch) {
            return true;
        }
    }
    return false;
}
```

In the  
**WORST CASE**

Execution completes in **at most:**

$k_0n + k_1$  operations



# Types of Analysis

**Best case analysis:** running time under best input (not reflective of overall performance)

**Average case analysis:** assumes equal probability of input (not usually the case)

**Expected case analysis:** assumes probability of occurrence of input is known or can be estimated, and if possible may be too expensive



**Worst case analysis:** running time under worst input, gives upper bound, it can't get worse, good for sleeping well at night!

**n** here is the length of the string

```
bool linearSearch(const std::string& str, char ch)
{
    for (int i = 0; i < str.length(); i++)
    {
        if (str[i] == ch) {
            return true;
        }
    }
    return false;
}
```

Execution completes in **at most:**

$k_0n + k_1$  operations

Some constant number  
of operations repeated  
inside the loop

Some constant number  
of operations performed  
outside the loop

$n$  here is the length of the string

```
bool linearSearch(const std::string& str, char ch)
{
    for (int i = 0; i < str.length(); i++)
    {
        if (str[i] == ch) {
            return true;
        }
    }
    return false;
}
```

The number of times the loop is repeated, i.e. the size of `str`

Execution completes in **at most:**

$k_0n + k_1$  operations

Some constant number of operations repeated inside the loop

Some constant number of operations performed outside the loop

# Observation

Don't need to explicitly compute the constants  $k_i$

$$4n + 1000$$

$$n + 137$$

***Dominant term*** is sufficient to explain overall behavior (in this case linear)

# Big-O Notation

Ignores everything except **dominant term**

Examples:

Notation: describes the overall behavior

$$T(n) = 4n + 4 = O(n)$$

$$T(n) = 164n + 35 = O(n)$$

$$T(n) = n^2 + 35n + 5 = O(n^2)$$

$$T(n) = 2n^3 + 98n^2 + 210 = O(n^3)$$

$$T(n) = 2^n + 5 = O(2^n)$$

# Big-O Notation

$T(n)$  is the running time

$n$  is the size of the input

Ignores everything except **dominant term**

Examples:

Notation: describes the overall behavior

$$T(n) = 4n + 4 = O(n)$$

$$T(n) = 164n + 35 = O(n)$$

$$T(n) = n^2 + 35n + 5 = O(n^2)$$

$$T(n) = 2n^3 + 98n^2 + 210 = O(n^3)$$

$$T(n) = 2^n + 5 = O(2^n)$$

# Big-O Notation

Ignores everything except **dominant term**

Examples:

$$T(n) = 4n + 4 = O(n)$$

$$T(n) = 164n + 35 = O(n)$$

$$T(n) = n^2 + 35n + 5 = O(n^2)$$

$$T(n) = 2n^3 + 98n^2 + 210 = O(n^3)$$

$$T(n) = 2^n + 5 = O(2^n)$$

Big-O describes the overall behavior

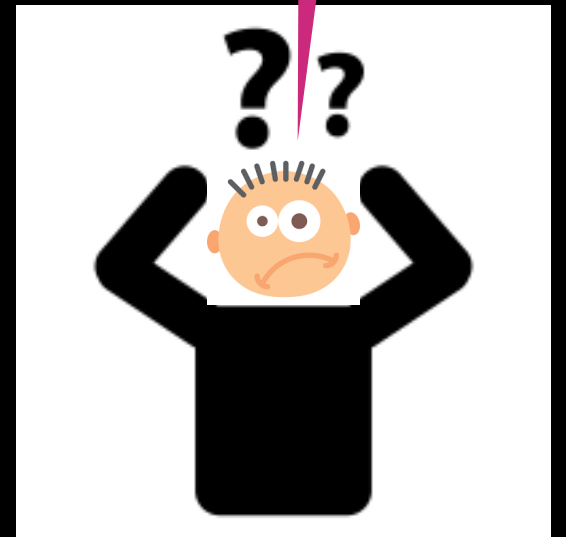
Let  $T(n)$  be the *running time* of an algorithm measured as number of operations given **input of size  $n$** .

$T(n)$  is  $O(f(n))$

if it grows **no faster** than  $f(n)$

# Big-O Notation

But  
 $164n+35 > n$



Ignores everything except **dominant term**

Examples:

$$T(n) = 4n + 4 = O(n)$$

$$T(n) = 164n + 35 = O(n)$$

$$T(n) = n^2 + 35n + 5 = O(n^2)$$

$$T(n) = 2n^3 + 98n^2 + 210 = O(n^3)$$

$$T(n) = 2^n + 5 = O(2^n)$$

Big-O describes the overall behavior

Let  $T(n)$  be the **running time** of an algorithm measured as number of operations given **input of size  $n$** .

$T(n)$  is  $O(f(n))$

if it grows **no faster** than  $f(n)$



# Big-O Notation

Ignores everything except **dominant term**

Examples:

$$T(n) = 4n + 4 = O(n)$$

$$T(n) = 164n + 35 = O(n)$$

$$T(n) = n^2 + 35n + 5 = O(n^2)$$

$$T(n) = 2n^3 + 98n^2 + 210 = O(n^3)$$

$$T(n) = 2^n + 5 = O(2^n)$$

Notation: describes the overall behavior

More formally:

**$T(n)$  is  $O(f(n))$**

if there exist constants  **$k$**  and  **$n_0$**

such that for all  **$n \geq n_0$**

$$T(n) \leq k f(n)$$

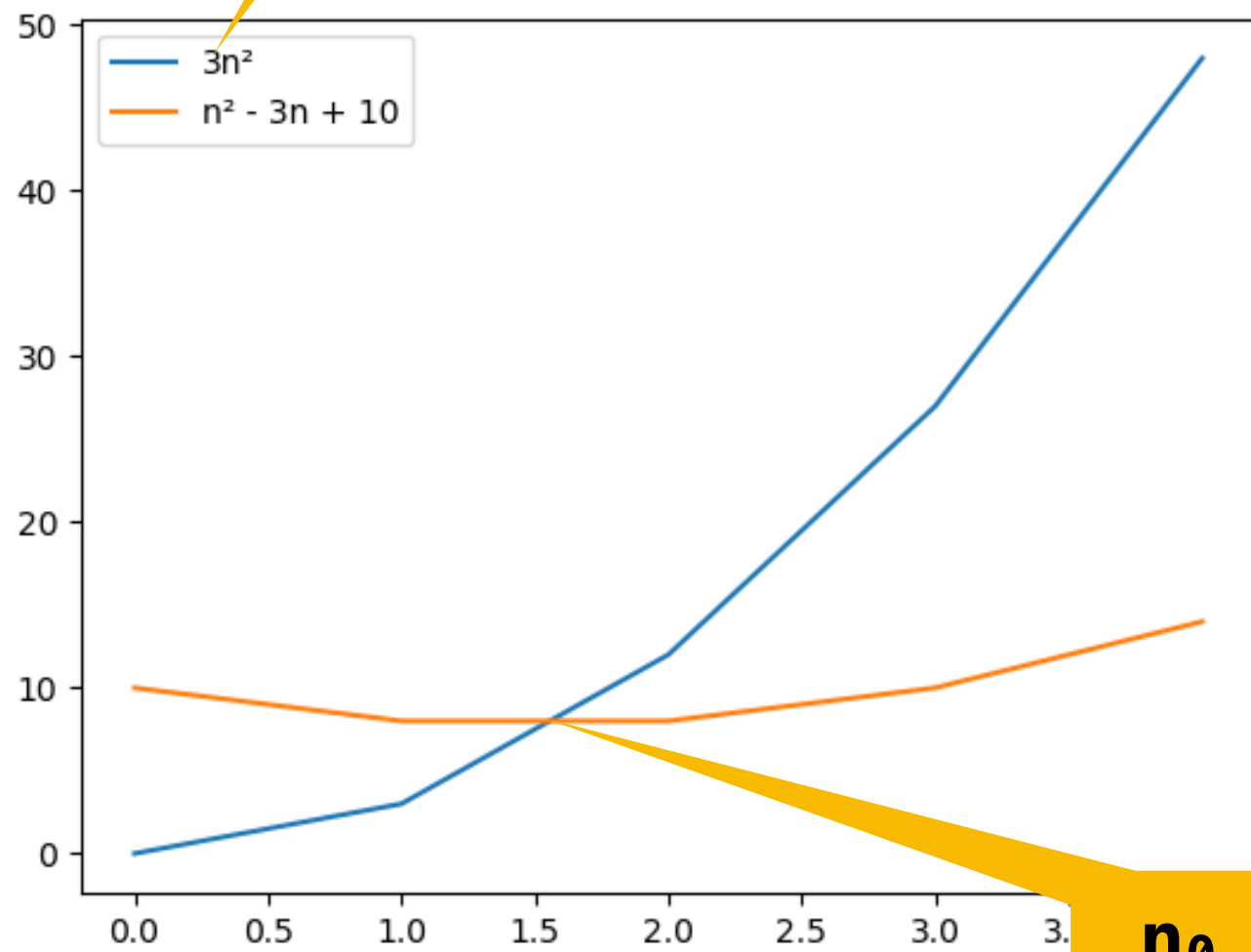


**More formally:**

$T(n)$  is  $O(f(n))$

if there exist constants  $k$  and  $n_0$   
such that for all  $n \geq n_0$ ,  
 $T(n) \leq kf(n)$

**$k = 3$**



**$n_0$**

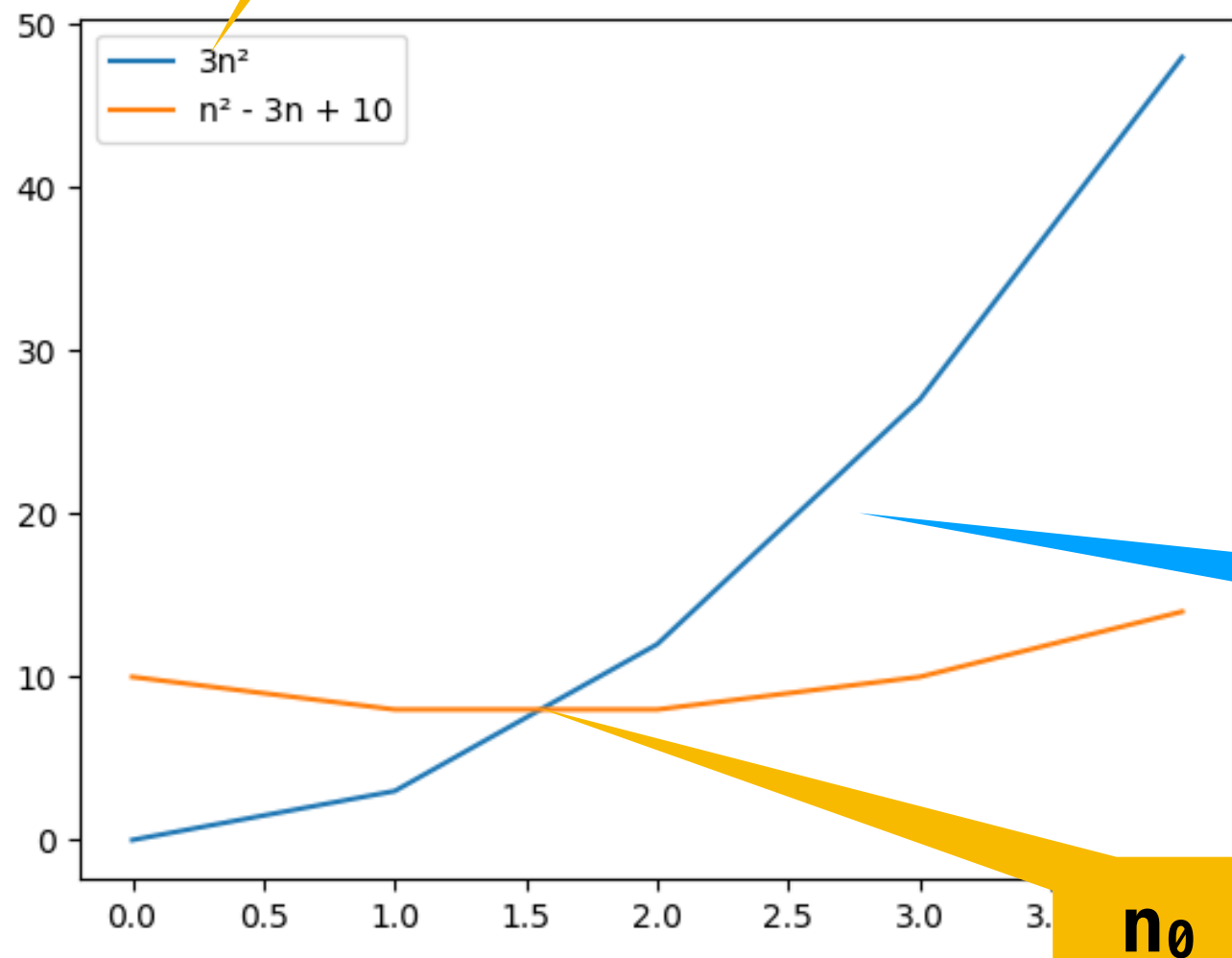
$T(n) = n^2 - 3n + 10$   
 $T(n)$  is  $O(n^2)$   
For  $k=3$  and  $n \geq 1.5$

**More formally:**

$T(n)$  is  $O(f(n))$

if there exist constants  $k$  and  $n_0$   
such that for all  $n \geq n_0$ ,  
 $T(n) \leq kf(n)$

**$k = 3$**



$T(n) = n^2 - 3n + 10$   
 $T(n)$  is  $O(n^2)$   
For  $k=3$  and  $n \geq 1.5$

This is why we can  
look at **dominant**  
**term only** to explain  
behavior

Big-O describes the overall growth rate of an algorithms for large  $n$

# Proving Big-O Relationship

Apply definition of Big-O to prove that  $T(n)$  is  $O(f(n))$  for particular functions  $T$  and  $f$

Do so by choosing  $k$  and  $n_0$  s.t. for all  $n \geq n_0$ ,  
 $T(n) \leq kf(n)$

# Proving Big-O Relationship

## Example:

Suppose  $T(n) = (n+1)^2$

We can say that  $T(n)$  is  $O(n^2)$

To prove it must find  $k$  and  $n_0$  s.t. for all  $n \geq n_0$ ,

$$(n+1)^2 \leq kn^2$$

# Proving Big-O Relationship

## Example:

Suppose  $T(n) = (n+1)^2$

We can say that  $T(n)$  is  $O(n^2)$

To prove it must find  $k$  and  $n_0$  s.t. for all  $n \geq n_0$ ,

$$(n+1)^2 \leq kn^2$$

Expand  $(n+1)^2 = n^2 + 2n + 1$

Observe that, as long as  $n \geq 1$ ,  $n \leq n^2$  and  $1 \leq n^2$

# Proving Big-O Relationship

## Example:

Suppose  $T(n) = (n+1)^2$

We can say that  $T(n)$  is  $O(n^2)$

To prove it must find  $k$  and  $n_0$  s.t. for all  $n \geq n_0$ ,

$$(n+1)^2 \leq kn^2$$

Expand  $(n+1)^2 = n^2 + 2n + 1$

Observe that, as long as  $n \geq 1$ ,  $n \leq n^2$  and  $1 \leq n^2$

**Thus** if we choose  $n_0 = 1$  and  $k = 4$  we have

$$n^2 + 2n + 1 \leq n^2 + 2n^2 + n^2 = 4n^2$$

$T(n)$

$k$

$f(n)$



# Proving Big-O Relationship

## Example:

Suppose  $T(n) = (n+1)^2$

We can say that  $T(n)$  is  $O(n^2)$

To prove it must find  $k$  and  $n_0$  s.t. for all  $n \geq n_0$ ,

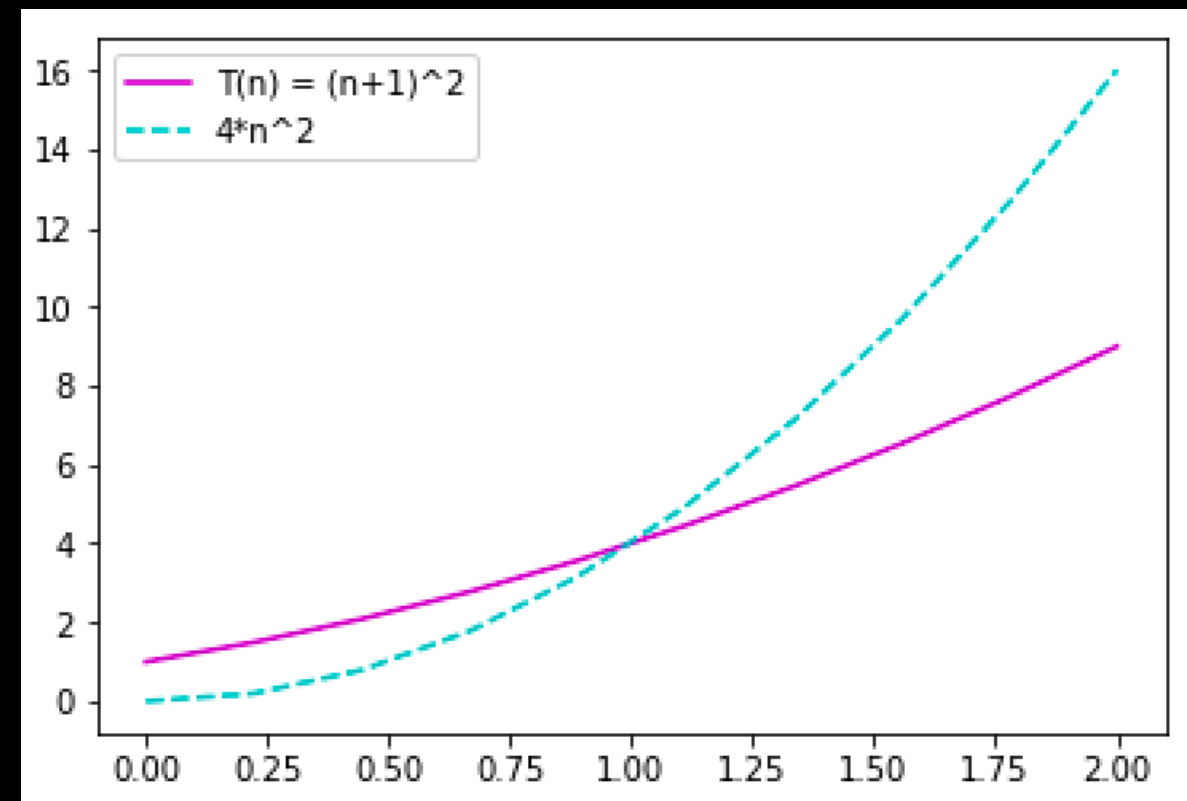
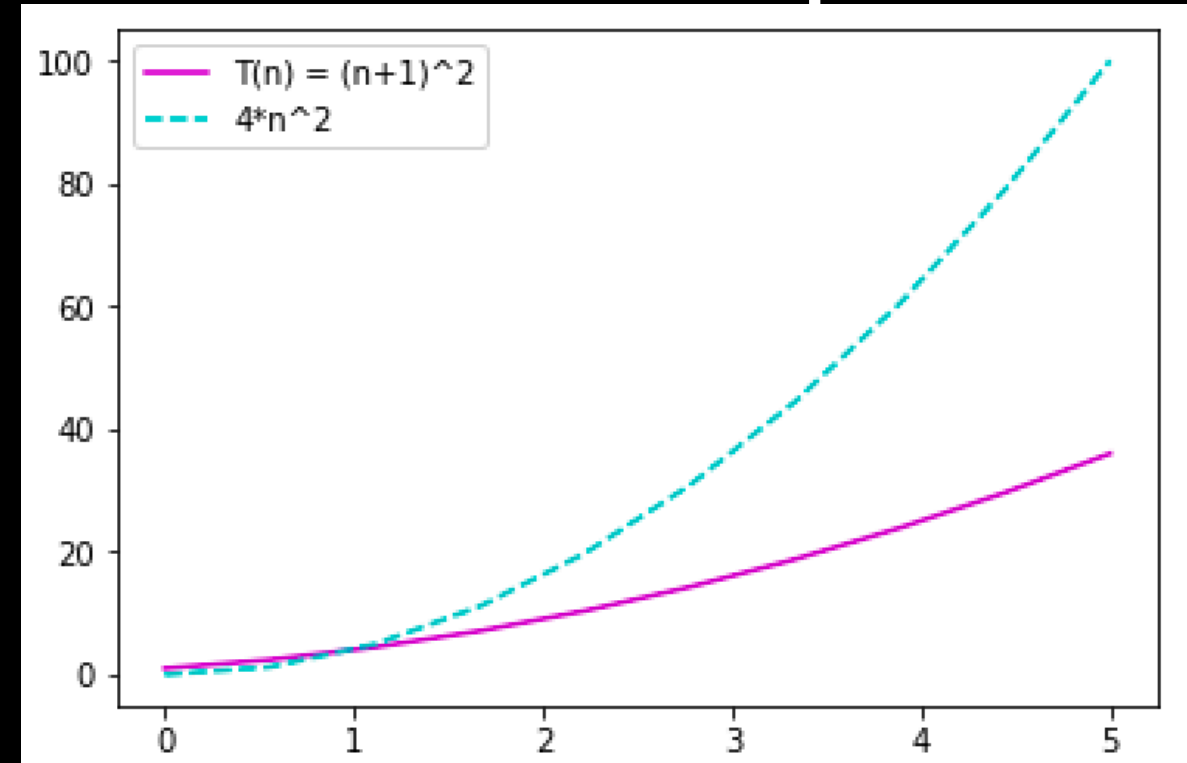
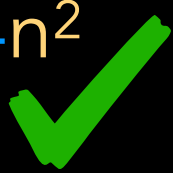
$$(n+1)^2 \leq kn^2$$

Expand  $(n+1)^2 = n^2 + 2n + 1$

Observe that, as long as  $n \geq 1$ ,  $n \leq n^2$  and  $1 \leq n^2$

**Thus** if we choose  $n_0 = 1$  and  $k = 4$  we have

$$n^2 + 2n + 1 \leq n^2 + 2n^2 + n^2 = 4n^2$$



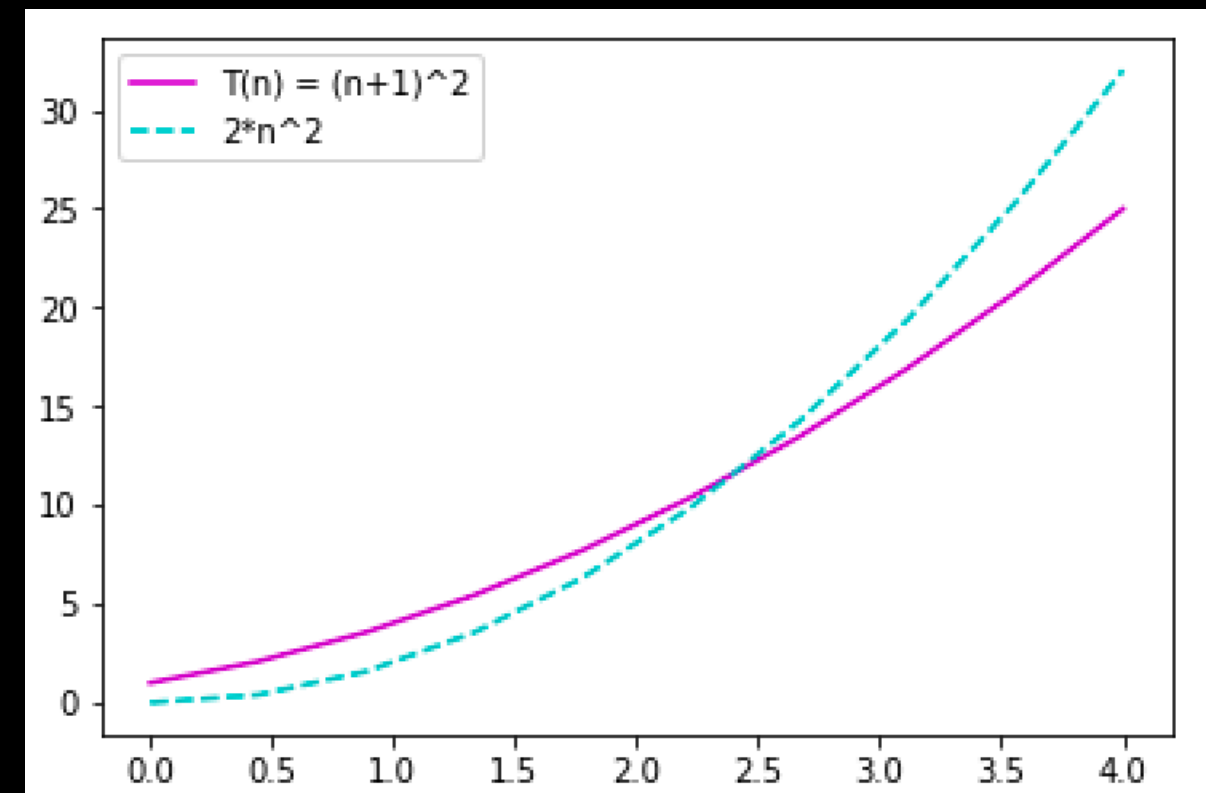
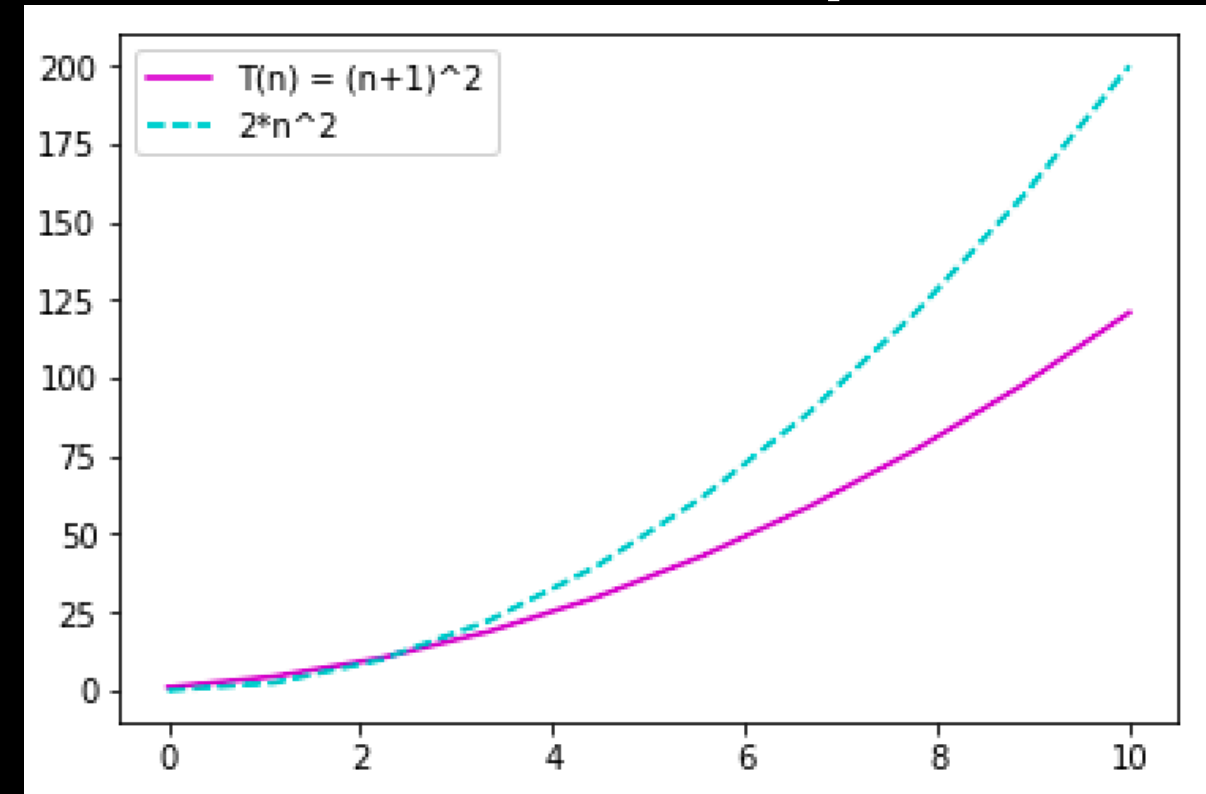
# Proving Big-O Relationship

**Not Unique:**

Could also choose  $n_0 = 3$  and  $k = 2$  because

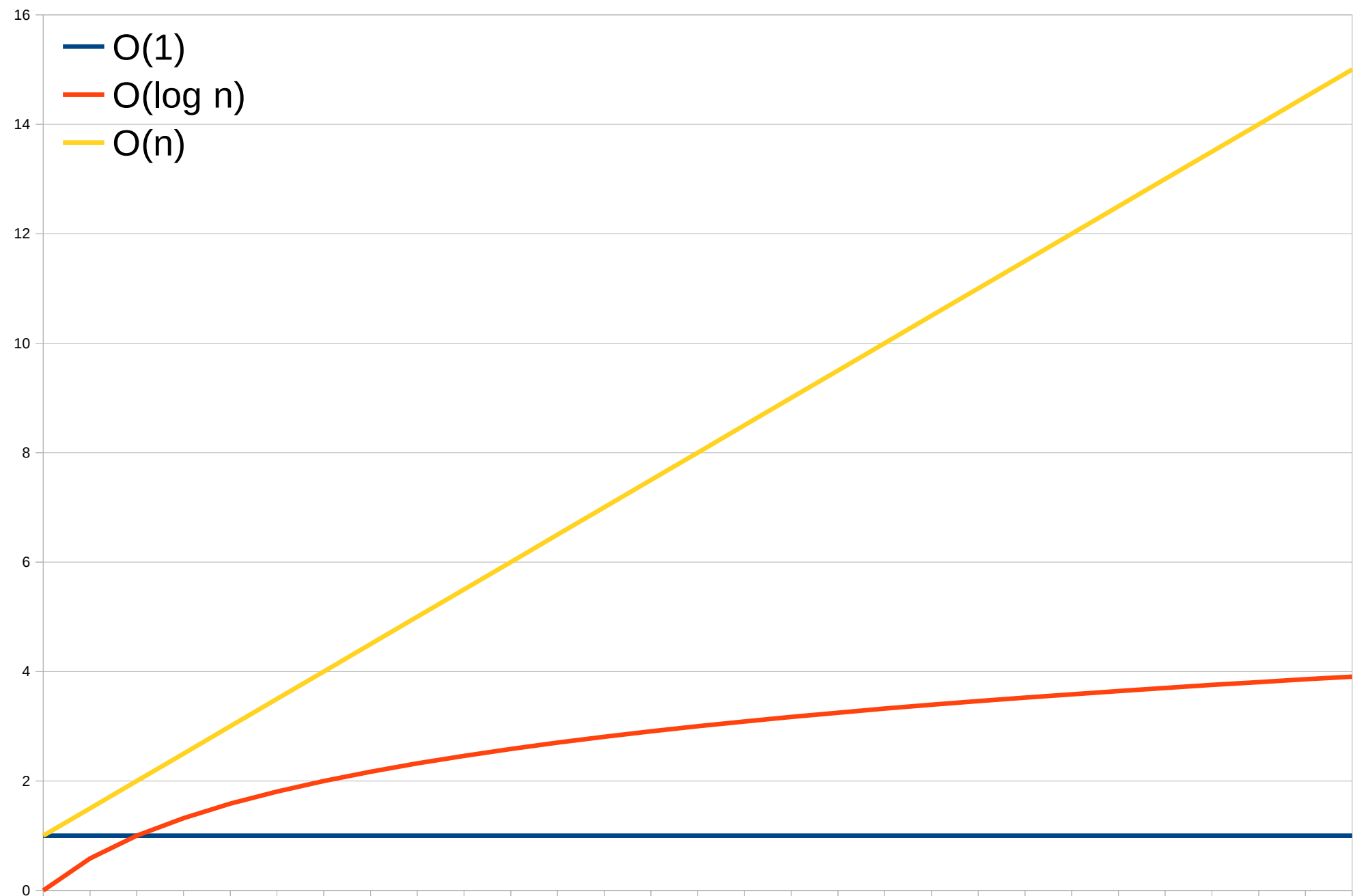
$(n+1)^2 \leq 2n^2$  for all  $n \geq 3$

For proof one is enough

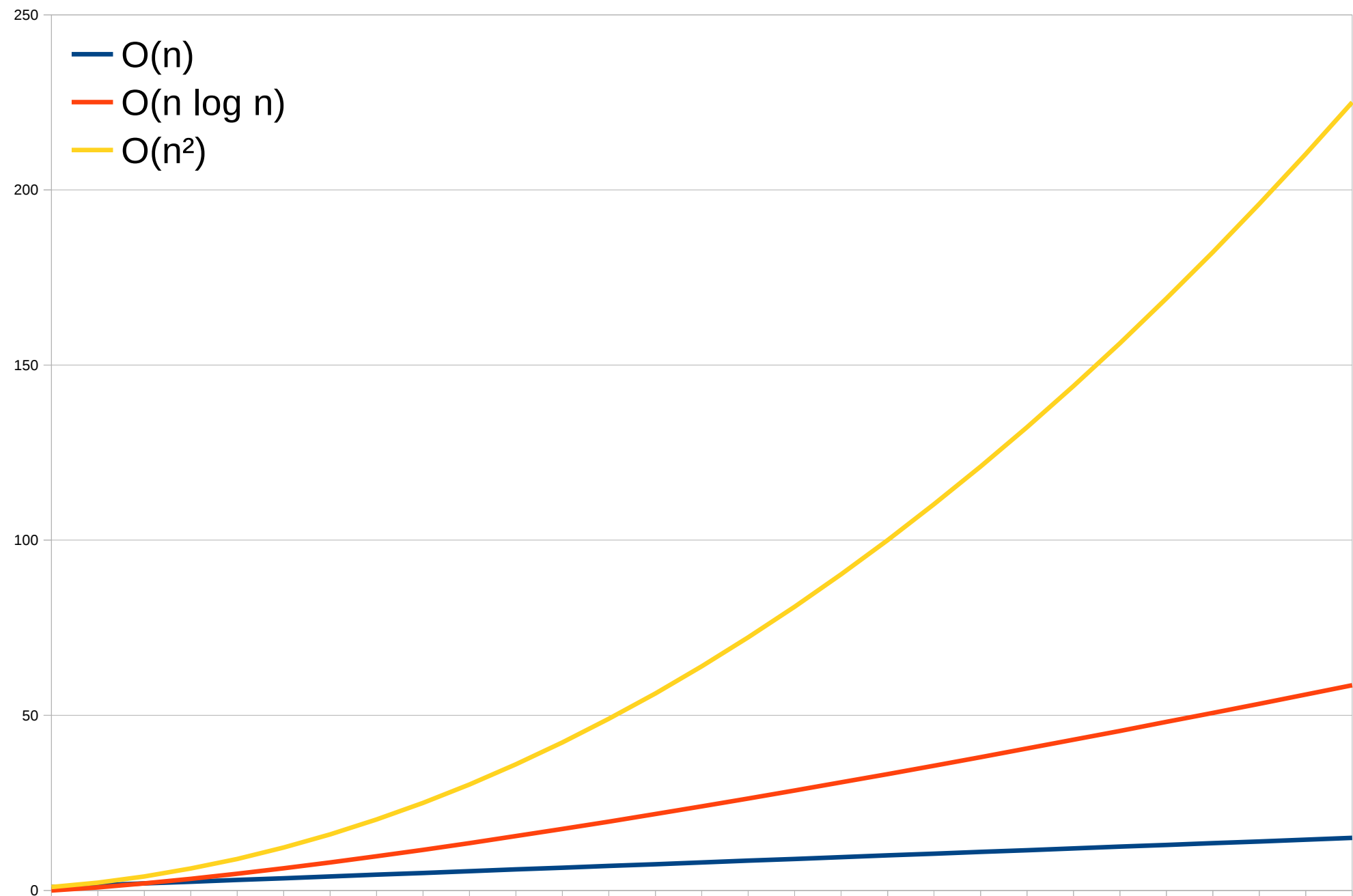


# A visual comparison of growth rates

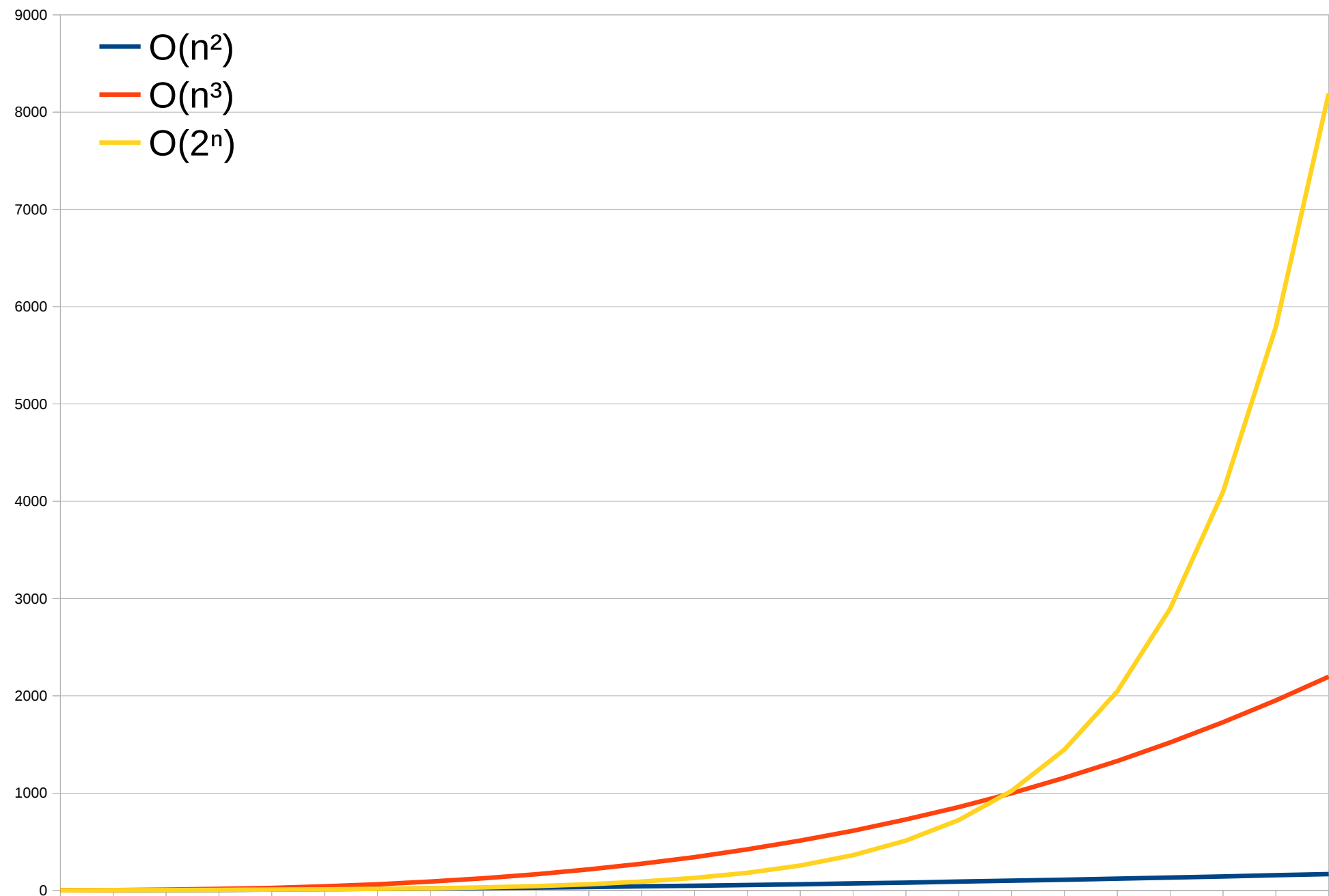
## Growth Rates, Part One



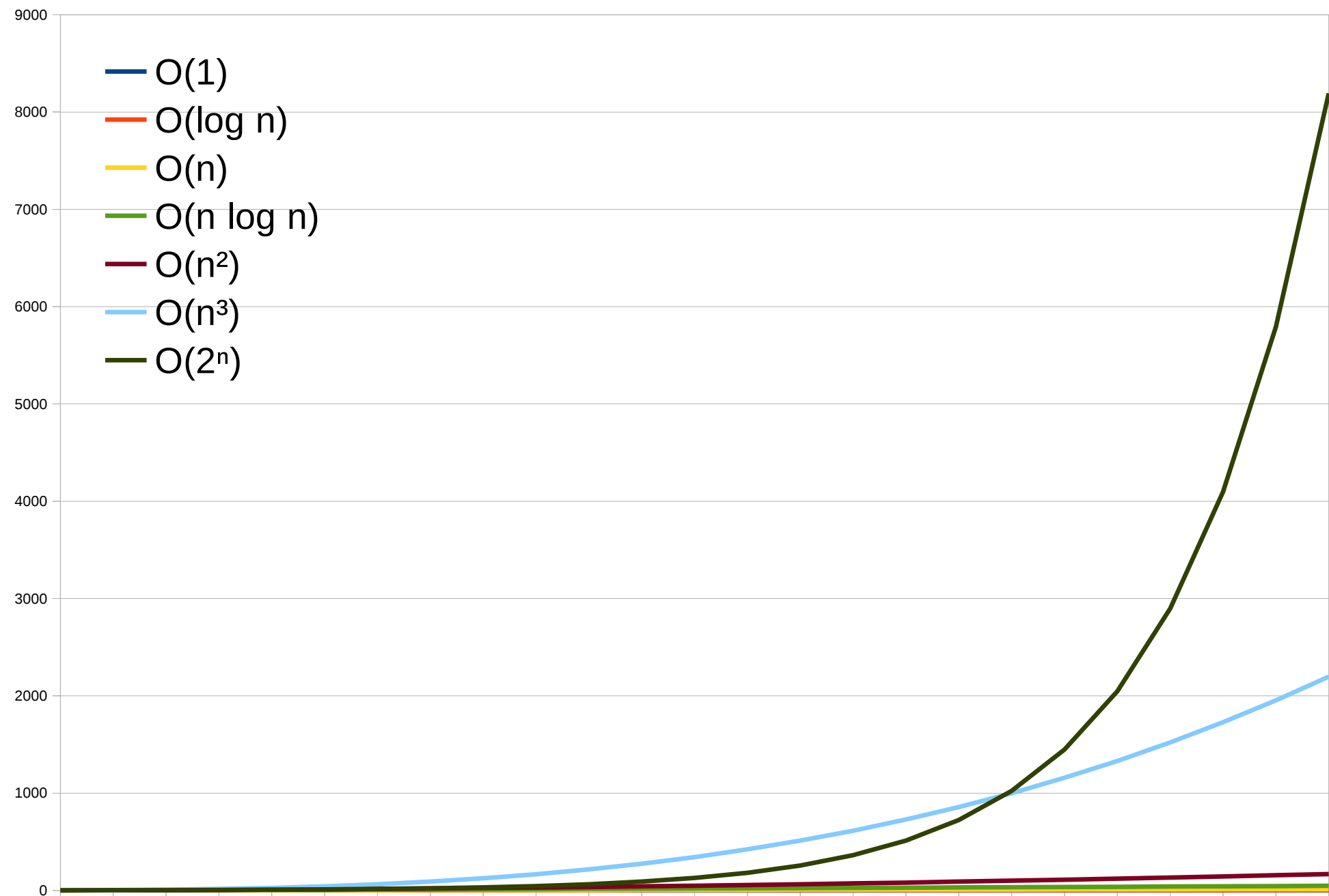
## Growth Rates, Part Two



## Growth Rates, Part Three



To Give You A Better Sense...



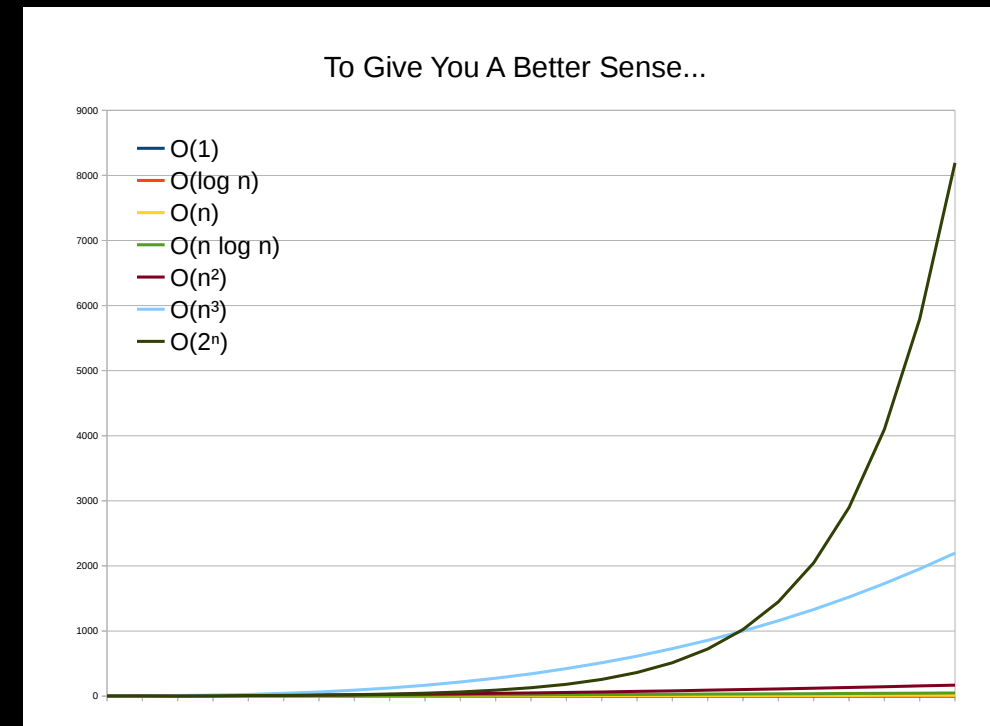
# Tight is more meaningful

We can say that if  $T(n)$  is  $O(n)$

It is also true that  $T(n)$  is  $O(n^3)$

And it is also true that  $T(n)$  is  $O(2^n)$

But what does it mean???



The closest Big-O is the most descriptive of the overall worst-case behavior



# Tightening the bounds

Big-O: upper bound

$T(n)$  is  $O(f(n))$

if there exist constants  $k$  and  $n_0$  such that for all  $n \geq n_0$   $T(n) \leq k f(n)$

Grows no faster than  $f(n)$

# Tightening the bounds

Big-O: upper bound

$T(n)$  is  $O(f(n))$

if there exist constants  $k$  and  $n_0$  such that for all  $n \geq n_0$   $T(n) \leq k f(n)$

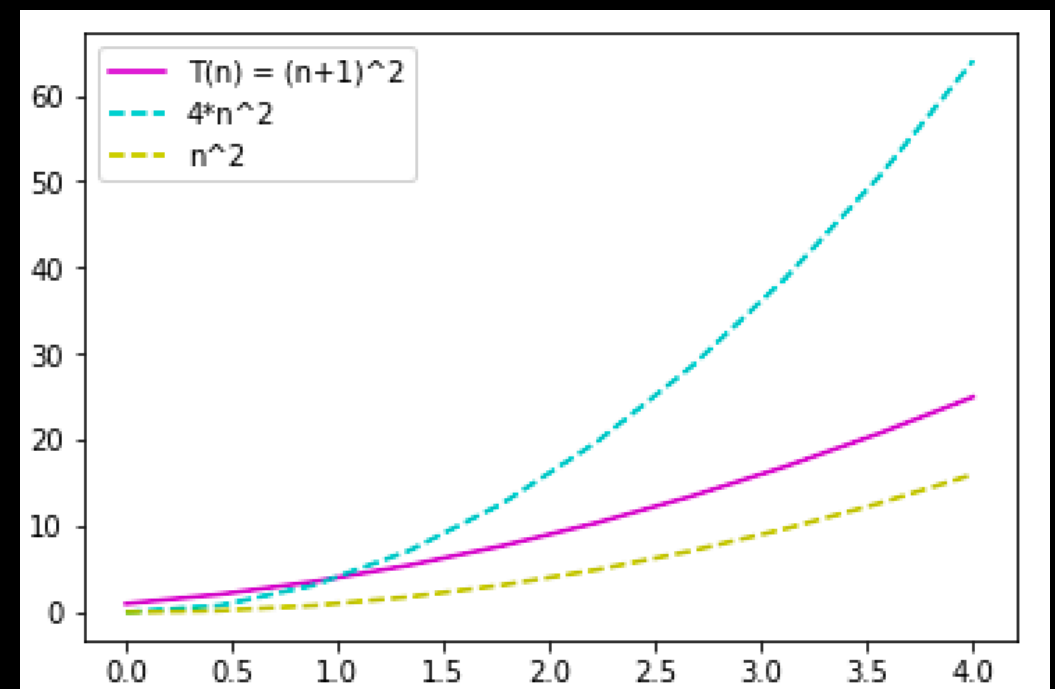
Grows no faster than  $f(n)$

Omega: lower bound

$T(n)$  is  $\Omega(f(n))$

if there exist constants  $k$  and  $n_0$  such that for all  $n \geq n_0$   $T(n) \geq k f(n)$

Grows at least as fast as  $f(n)$

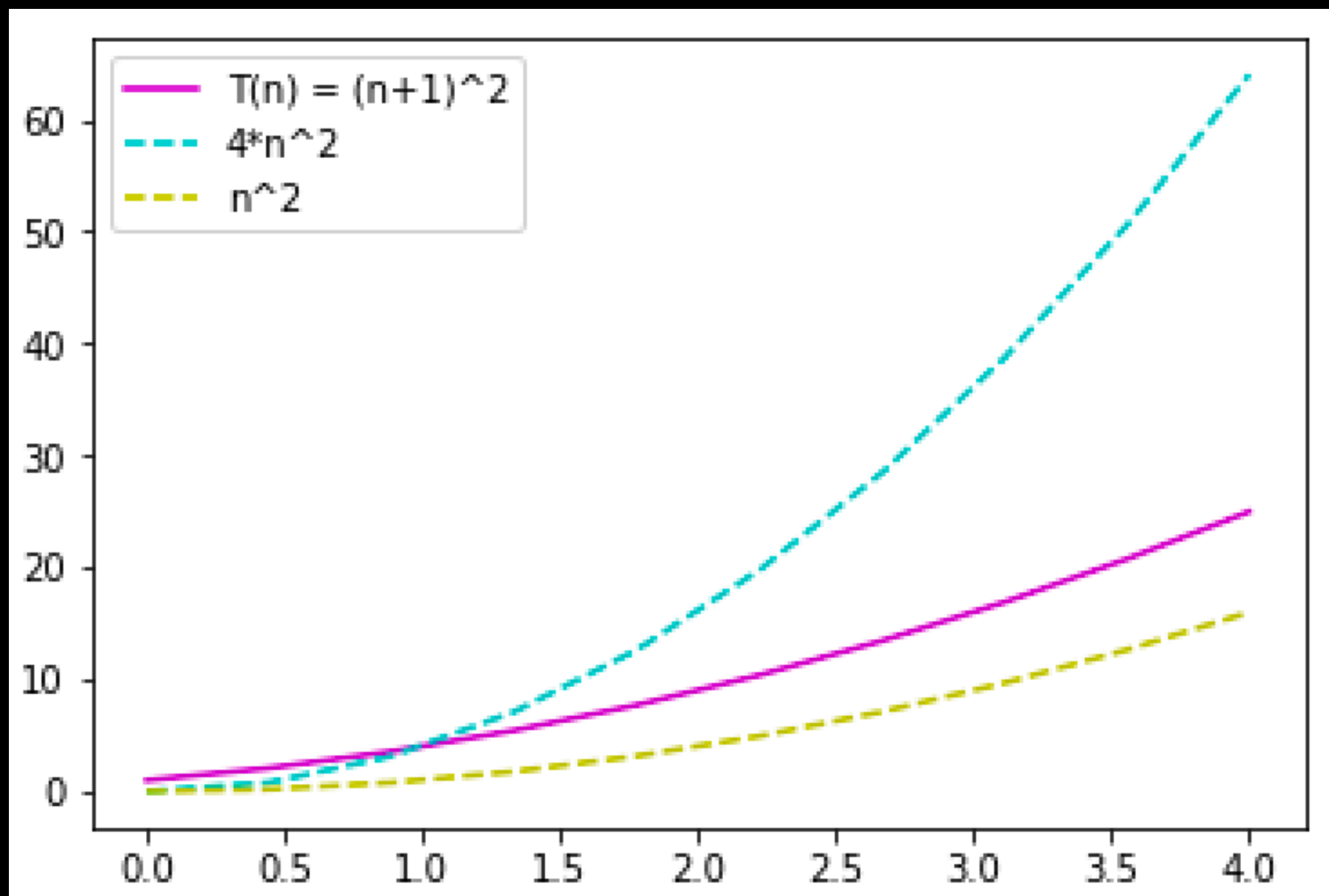


# Tightening the bounds

Theta: **tight bound**

$T(n)$  is  $\Theta(f(n))$

Grows at the same rate as  $f(n)$  : iff both  $T(n)$  is  $O(f(n))$  and  $\Omega(f(n))$



# A numerical comparison of growth rates

<div><div>n</div><div>f(n)</div></div>	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	$10^2$	$10^3$	$10^4$	$10^5$	$10^6$
$n * \log_2 n$	30	664	9,965	$10^5$	$10^6$	$10^7$
$n^2$	$10^2$	$10^4$	$10^6$	$10^8$	$10^{10}$	$10^{12}$
$n^3$	$10^3$	$10^6$	$10^9$	$10^{12}$	$10^{15}$	$10^{18}$
$2^n$	$10^3$	$10^{30}$	$10^{301}$	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$



# What **does** Big-O describe?

“Long term” behavior of a function

Compare behavior  
of 2 algorithms

If algorithm A has runtime  $O(n)$  and algorithm B has runtime  $O(n^2)$ , for large inputs A will always be faster.

If algorithm A has runtime  $O(n)$ , for large inputs doubling the size of the input will double the runtime

Analyze algorithm behavior  
with growing input

# What **can't** Big-O describe?

The actual runtime of an algorithm

$$10^{100}n = O(n)$$

$$10^{-100}n = O(n)$$

How an algorithm behaves on small input

$$n^3 = O(n^3)$$

$$10^6 = O(1)$$

# To summarize Big-O

It is a means of describing the growth rate of a function

It ignores all but the dominant term

It ignores constants

Allows for quantitative ranking of algorithms

Allows for quantitative reasoning about algorithms



More examples next time