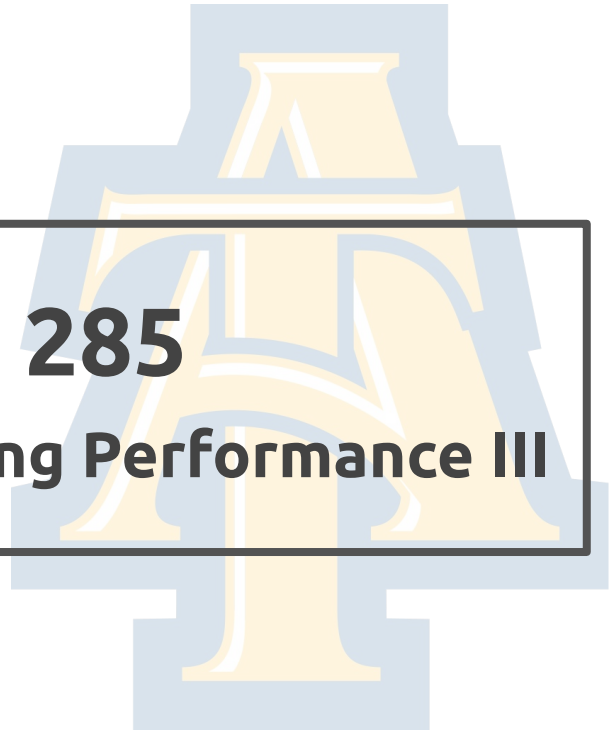
COMP - 285

Advanced Analysis of Algorithms

Welcome to COMP 285

Lecture 4: (Formal) Measuring Performance III



Chris Lucas (cflucas@ncat.edu)

HW0/Q0 are graded!

Thank you Priya and Tolu!

HW1 is due on Tuesday @ 1:59PM!

HW1 is due on Tuesday @ 1:59PM!

“Rendering Homework in Repl.it How-To”

Piazza for Questions!

Q1 on Tuesday!

(Similar setup on Lectures 3, 4)


Kahoot!
Enter your @aggies.ncat email

Big Questions!

- How to Big-Oh? (recursion edition)
- Who really is Big-Oh?

Recall where we ended last lecture...

Space Complexity

- We can use Big-O to describe the amount of additional space “units” we use.
- When we declare primitive types, that takes constant space (i.e. int x = 4 is O(1)).
- In general, whenever you create data structures that depend on the size of your input, you’ll have to keep track of usage.

What’s the space complexity?

```
void doSomething(const std::vector<int>& vec) {
 std::vector<int> results;
 for(int i = 0; i < vec.size(); i++) {
   results.push_back(vec[i]);
 }
}
```

What's the space complexity?

```
void doSomething(const std::vector<int>& vec) {
 std::vector<int> results;
 for(int i = 0; i < vec.size(); i++) {
   results.push_back(vec[i]);
 }
}
```

O(N) space complexity,
O(N) time complexity

The number of elements in the *results* vector determines our space complexity!!!

Poll - What's the space complexity?

```
void doSomething(const std::vector<int>& vec) {
 std::vector<int> results;
 for(int i = 0; i < vec.size(); i++) {
   for(int j = 0; j < vec.size(); j++) {
     for(int k = 0; k < vec.size(); k++) {
       results.push_back(vec[i] * vec[j] * vec[k]);
     }
   }
 }
}
```

The number of elements in the *results* vector determines our space complexity!!!

1. O(1) 2. O(N) 3. O(N log N) 4. $O(N^2)$ 5. $O(N^3)$

Poll - What's the space complexity?

```
void doSomething(const std::vector<int>& vec) {
 std::vector<int> results;
 for(int i = 0; i < vec.size(); i++) {
   for(int j = 0; j < vec.size(); j++) {
     for(int k = 0; k < vec.size(); k++) {
       results.push_back(vec[i] * vec[j] * vec[k]);
     }
   }
 }
}
```

1. O(1) 2. O(N) 3. O(N log N) 4. $O(N^2)$ 5. $O(N^3)$

What's the space complexity?

```
void doSomething(const std::vector<int>& vec) {
 for(int i = 0; i < vec.size(); i++) {
   for(int j = 0; j < vec.size(); j++) {
     std::vector<int> results;
     for(int k = 0; k < vec.size(); k++) {
       results.push_back(vec[i] * vec[j] * vec[k]);
     }
   }
 }
}
```

1. O(1) 2. O(N) 3. O(N log N) 4. $O(N^2)$ 5. $O(N^3)$

What's the space complexity?

```
void doSomething(const std::vector<int>& vec) {
 for(int i = 0; i < vec.size(); i++) {
   for(int j = 0; j < vec.size(); j++) {
     std::vector<int> results;
     for(int k = 0; k < vec.size(); k++) {
       results.push_back(vec[i] * vec[j] * vec[k]);
     }
   }
 }
}
```

1. O(1) 2. O(N) 3. O(N log N) 4. $O(N^2)$ 5. $O(N^3)$

What's the space complexity?

```
void doSomething(const std::vector<int>& vec) {
 for(int i = 0; i < vec.size(); i++) {
   for(int j = 0; j < vec.size(); j++) {
     std::vector<int> results;
     for(int k = 0; k < vec.size(); k++) {
       results.push_back(vec[i] * vec[j] * vec[k]);
     }
   }
 }
}
```

The number of elements in the *results* vector determines our space complexity!!!

1. O(1) 2. O(N) 3. O(N log N) 4. $O(N^2)$ 5. $O(N^3)$

Big Questions!

- How to Big-Oh? (recursion edition)
- Who really is Big-Oh?

How Recursion Actually Works: Call Stack

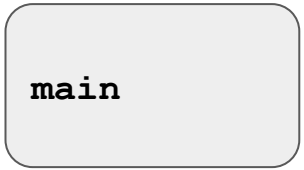
- Every time a function is called (any function, not just a recursive one), a **stack frame** is pushed onto to the call stack.
- The stack frame keeps track of the local variables for that function and where to return after completion.
- When the function returns, the stack frame is popped off the call stack.

What’s the space complexity?

```
void factorial(const int n) {
  if (n == 0) {
    return 1;
  }
  return n * factorial(n - 1);
}
```

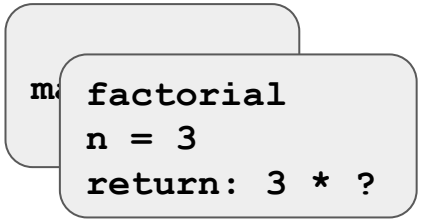
What’s the space complexity?

```
void factorial(const int n) {
  if (n == 0) {
    return 1;
  }
  return n * factorial(n - 1);
}
```



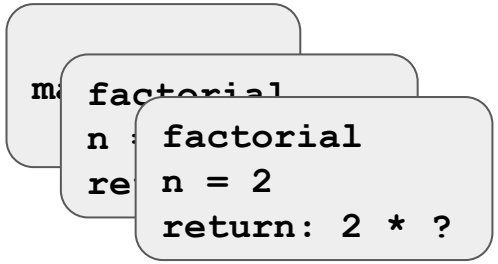
What's the space complexity?

```
void factorial(const int n) {
  if (n == 0) {
    return 1;
  }
  return n * factorial(n - 1);
}
```



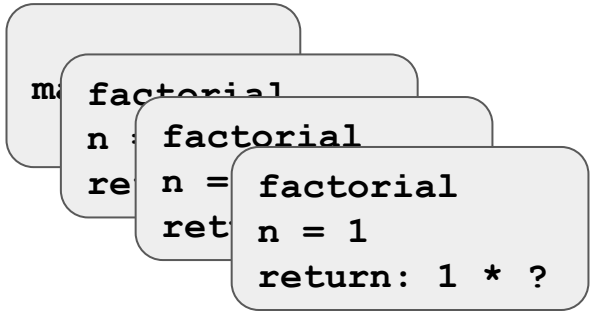
What's the space complexity?

```
void factorial(const int n) {
  if (n == 0) {
    return 1;
  }
  return n * factorial(n - 1);
}
```



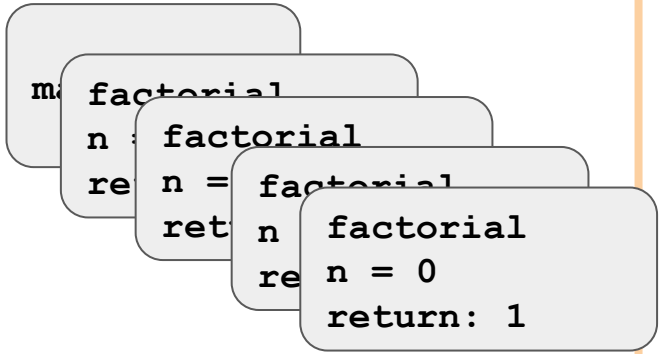
What's the space complexity?

```
void factorial(const int n) {
  if (n == 0) {
    return 1;
  }
  return n * factorial(n - 1);
}
```



What's the space complexity?

```
void factorial(const int n) {
  if (n == 0) {
    return 1;
  }
  return n * factorial(n - 1);
}
```



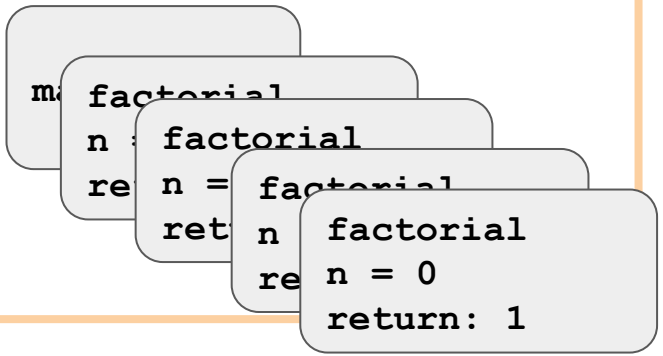
Recursive Space Complexity

Stack frames takes up memory on our computer

- Each stack frame needs to at least keep track of local variables and where to return which is usually constant space.

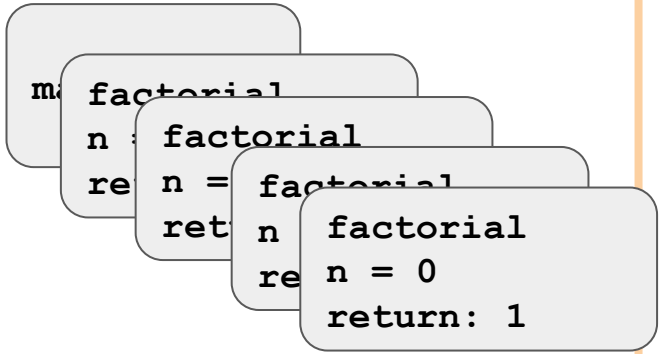
Note that when we've reached our base case, we have n stack frame objects (each holding O(1) amount of information), meaning our space complexity is O(n).

= # of recursive calls * space used in each frame



What's the space complexity?

```
void factorial(const int n) {
  if (n == 0) {
    return 1;
  }
  return n * factorial(n - 1);
}
```



 # recursive calls * space used per call = 1 * N

= O(N)

What's the space complexity?

```
void myFunction(const int n, const int m) {
  std::vector<int> results;
  for(int i = 0; i < m; i++) {
    results.push_back(i);
  }
  if (n == 0) {
    return 1;
  }
  return n * myFunction(n - 1);
}
```

= # of recursive calls * space used in each frame

1. O(N) 2. O(N^m) 3. O(N*M) 4. O(N^2) 5. O(N^3)

What's the space complexity?

```
void myFunction(const int n, const int m) {
  std::vector<int> results;
  for(int i = 0; i < m; i++) {
    results.push_back(i);
  }
  if (n == 0) {
    return 1;
  }
  return n * myFunction(n - 1);
}
```

= # of recursive calls * space used in each frame

1. O(N) 2. O(N^m) 3. O(N*M) 4. O(N^2) 5. O(N^3)

Big Questions!

- How to Big-Oh? (recursion edition)
- Who really is Big-Oh?

Cocktail Formal Definition

- Let $T(n)$, $g(n)$ be functions of positive integers.
  - Think of $T(n)$ as a runtime: positive and increasing in n.

Cocktail Formal Definition

- Let $T(n)$, $g(n)$ be functions of positive integers.
  - Think of $T(n)$ as a runtime: positive and increasing in n.

- We say "$T(n)$ is $O\big(g(n)\big)$" if:

  for large enough n,

  $T(n)$ is at most some constant multiple of $g(n)$.

A step-by-step example of what this means!

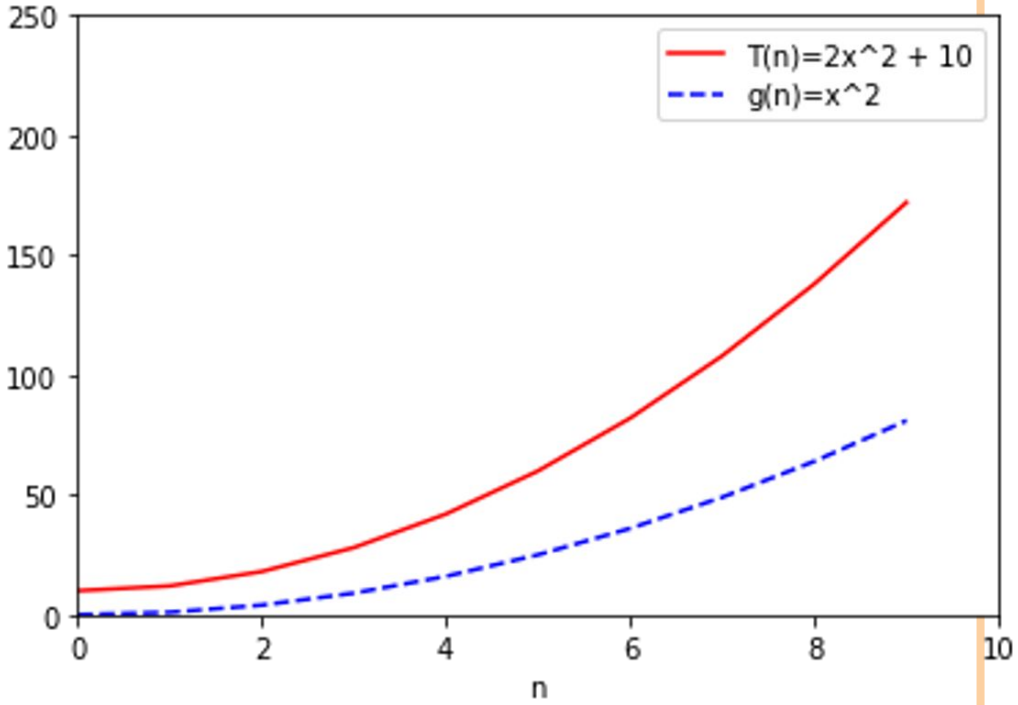
$$2n^2 + 10 = O(n^2)$$

For large enough n, T(n) is at most some *constant* multiple of g(n)

$$\underbrace{2n^2 + 10}_{T(n)} = \underbrace{O(n^2)}_{g(n)}$$

For large enough n, T(n) is at most some *constant* multiple of g(n)

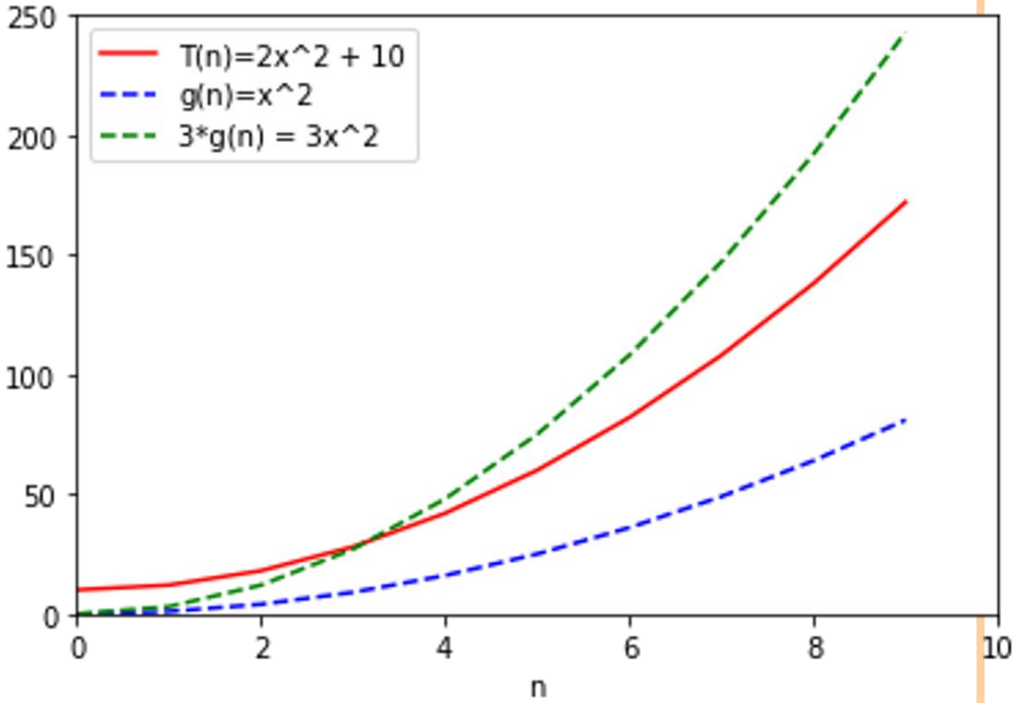
$$\underbrace{2n^2 + 10}_{T(n)} = O(\underbrace{n^2}_{g(n)})$$



For large enough n, T(n) is at most some *constant* multiple of g(n)

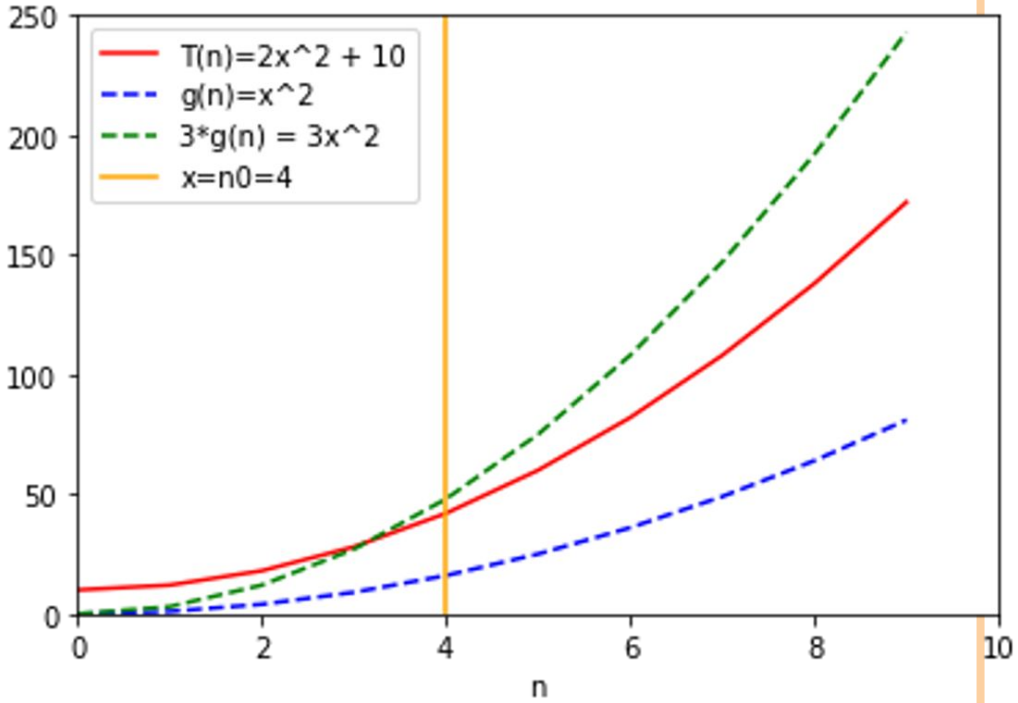
$$\underbrace{2n^2 + 10}_{T(n)} = O(\underbrace{n^2}_{g(n)})$$

We should try a bigger constant, let's say 3!



$3n^2 >= 2n^2 + 10$ for all n > 4.

$$\underbrace{2n^2 + 10}_{T(n)} = O(\underbrace{n^2}_{g(n)})$$



Black Tie Formal Definition of O(...)

Black Tie Formal Definition of O(...)

$$T(n) = O(g(n))$$

$$\Leftrightarrow$$

$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$T(n) \leq c \cdot g(n)$$

It’s okay if this looks scary

It’s kinda like learning a new language

Formal Definition of O(...)

$$T(n) = O(g(n))$$

"If and only if" → $\Leftrightarrow$

$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$T(n) \leq c \cdot g(n)$$

Formal Definition of O(...)

$$T(n) = O(g(n))$$

"If and only if" → $\Leftrightarrow$

$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$T(n) \leq c \cdot g(n)$$

"There exists"

Formal Definition of O(...)

$$T(n) = O(g(n))$$

"If and only if" $\iff$

$$\exists c, n_0 > 0 \ \ s.t. \ \ \forall n \geq n_0,$$
$$T(n) \leq c \cdot g(n)$$

"There exists"

"such that"

Formal Definition of O(...)

$$T(n) = O(g(n))$$

"If and only if" $\iff$ "For all"

$$\exists c, n_0 > 0 \ \ s.t. \ \ \forall n \geq n_0,$$

$$T(n) \leq c \cdot g(n)$$

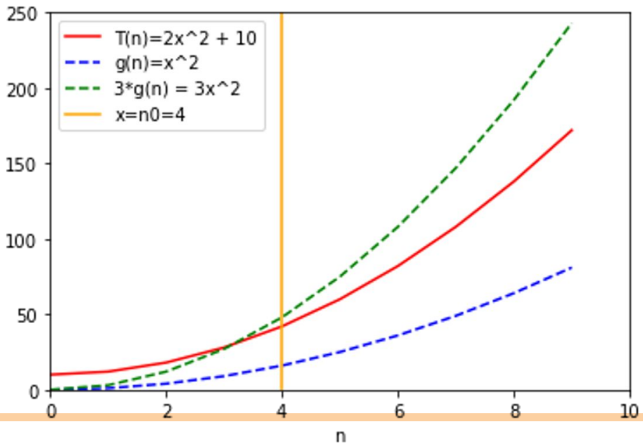
"There exists" "such that"

You guys just did this, actually…

$$T(n) = O(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \ \ s.t. \ \ \forall n \geq n_0, \ T(n) \leq c \cdot g(n)$$

$2n^2 + 10 <= 3n^2$ for all n > 4.

$$\underbrace{2n^2 + 10}_{T(n)} = O(\underbrace{n^2}_{g(n)})$$

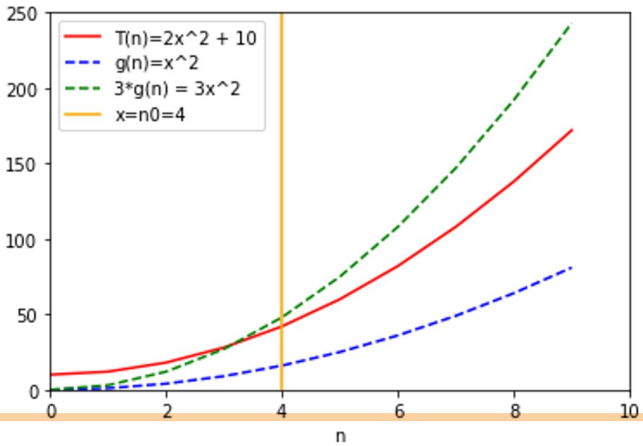


- Formally:
  - Choose c = 3
  - Choose n_0 = 4
  - Then:

$$T(n) = O(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \ \ s.t. \ \ \forall n \geq n_0, \ T(n) \leq c \cdot g(n)$$

$2n^2 + 10 <= 3n^2$ for all n > 4.

$$\underbrace{2n^2 + 10}_{T(n)} = O(\underbrace{n^2}_{g(n)})$$



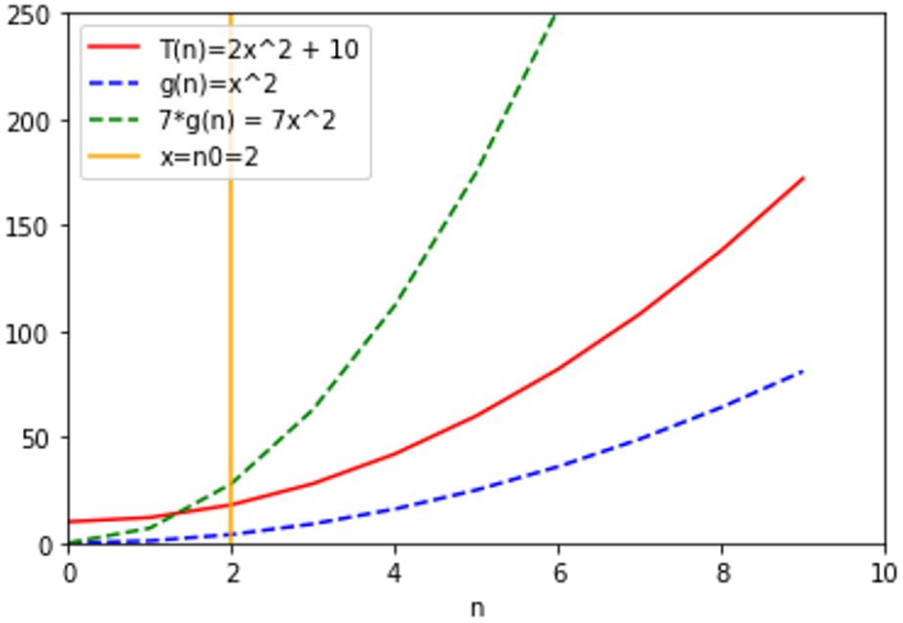
- Formally:
  - Choose c = 3
  - Choose $n_0 = 4$
  - Then:

$$\forall n \geq 4,$$

$$2n^2 + 10 \leq 3 \cdot n^2$$

$$T(n) = O(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \ \ s.t. \ \ \forall n \geq n_0, \ T(n) \leq c \cdot g(n)$$

$2n^2 + 10 <= 3n^2$ for all n > 4.



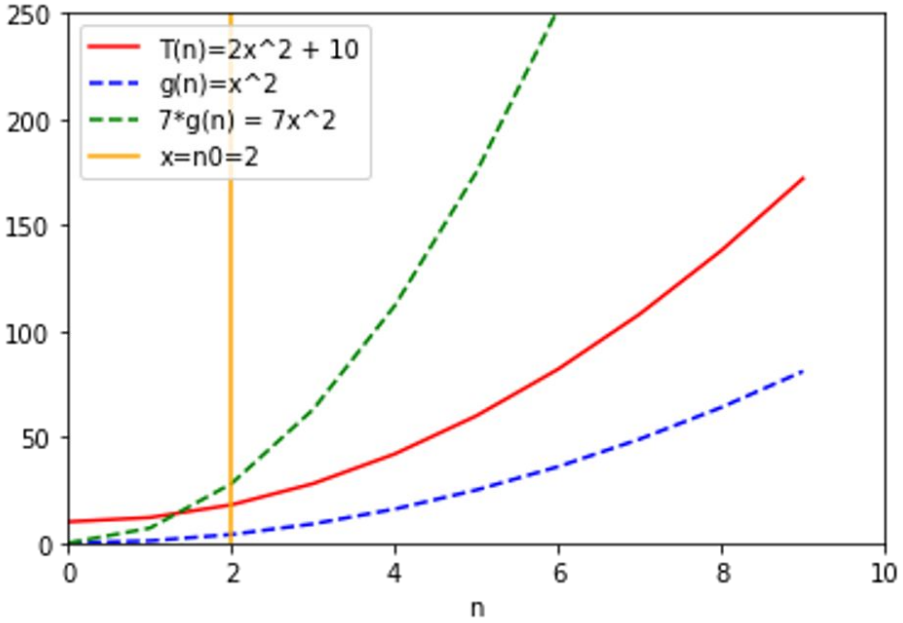
- Formally:
  - Choose c = 7
  - Choose $n_0 = 2$
  - Then:

$$\forall n \geq 2,$$

$$2n^2 + 10 \leq 7 \cdot n^2$$

$$T(n) = O(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \ \ s.t. \ \ \forall n \geq n_0, \ T(n) \leq c \cdot g(n)$$

$2n^2 + 10 <= 3n^2$ for all n > 4.



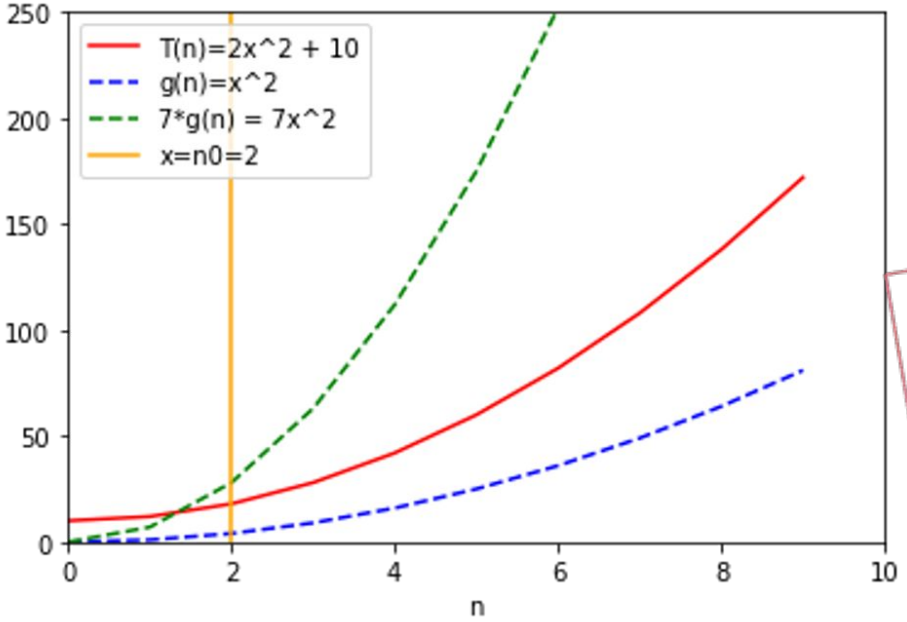
- Formally:
  - Choose c = 7
  - Choose $n_0 = 2$
  - Then:

$$\forall n \geq 2,$$

$$2n^2 + 10 \leq 7 \cdot n^2$$

$$T(n) = O(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \ \ s.t. \ \ \forall n \geq n_0, \ T(n) \leq c \cdot g(n)$$

$2n^2 + 10 <= 3n^2$ for all n > 4.



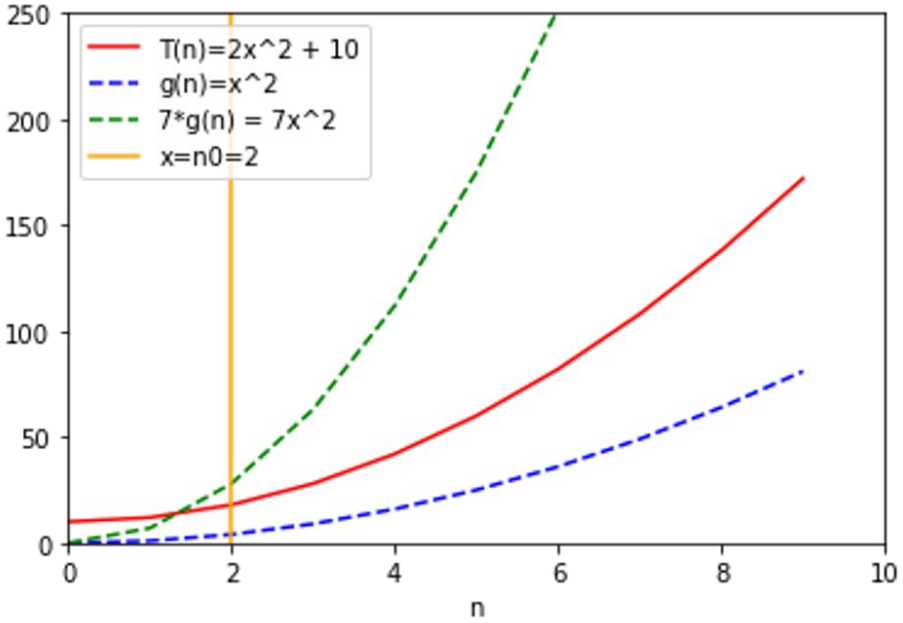
- Formally:
  - Choose c = 7
  - Choose $n_0 = 2$
  - Then:

$$\forall n \geq 2, \quad 2n^2 + 10 \leq 7 \cdot n^2$$

There is not a "correct" choice of c and n_0

$$T(n) = O(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \ \ s.t. \ \ \forall n \geq n_0, \ T(n) \leq c \cdot g(n)$$

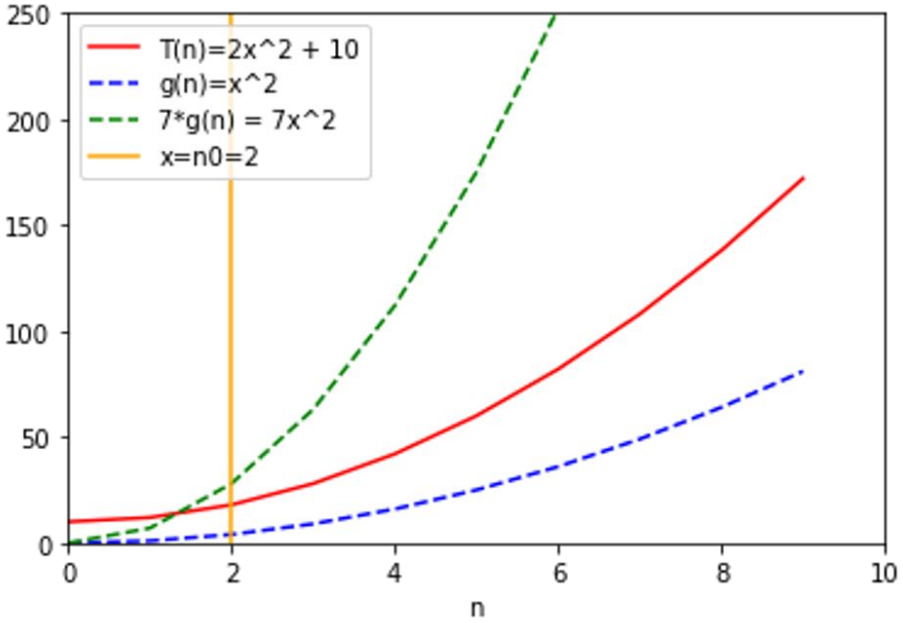
$2n^2 + 10 <= 3n^2$ for all n > 4.



$$\underbrace{2n^2 + 10}_{T(n)} = O(\underbrace{n^2}_{g(n)})$$

$$T(n) = O(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \ \ s.t. \ \ \forall n \geq n_0, \ T(n) \leq c \cdot g(n)$$

$2n^2 + 10 <= 3n^2$ for all n > 4.

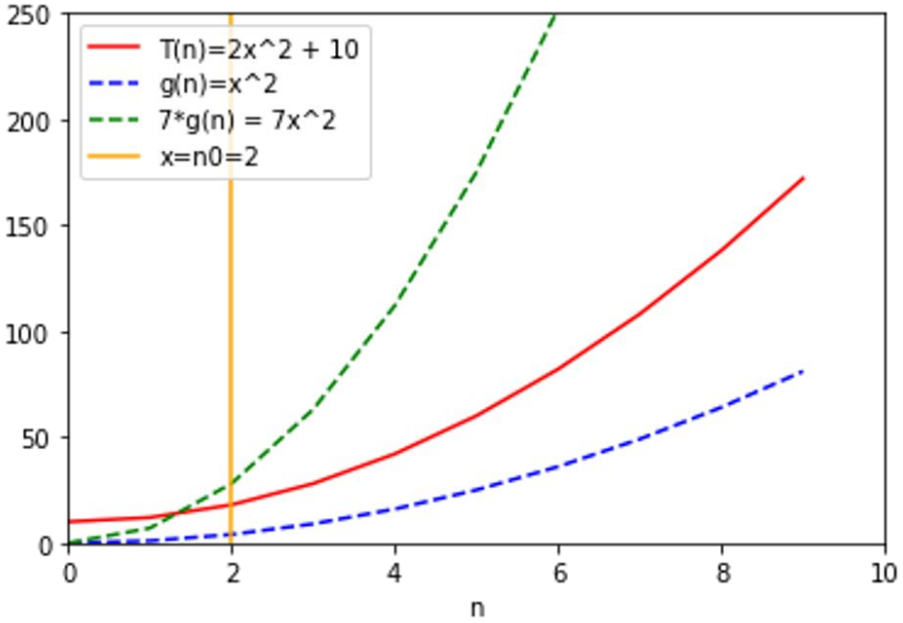


$$\underbrace{2n^2 + 10}_{T(n)} = O(\underbrace{n^2}_{g(n)})$$

$$\Leftrightarrow$$

$$T(n) = O(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \ \ s.t. \ \ \forall n \geq n_0, \ T(n) \leq c \cdot g(n)$$

$2n^2 + 10 <= 3n^2$ for all n > 4.



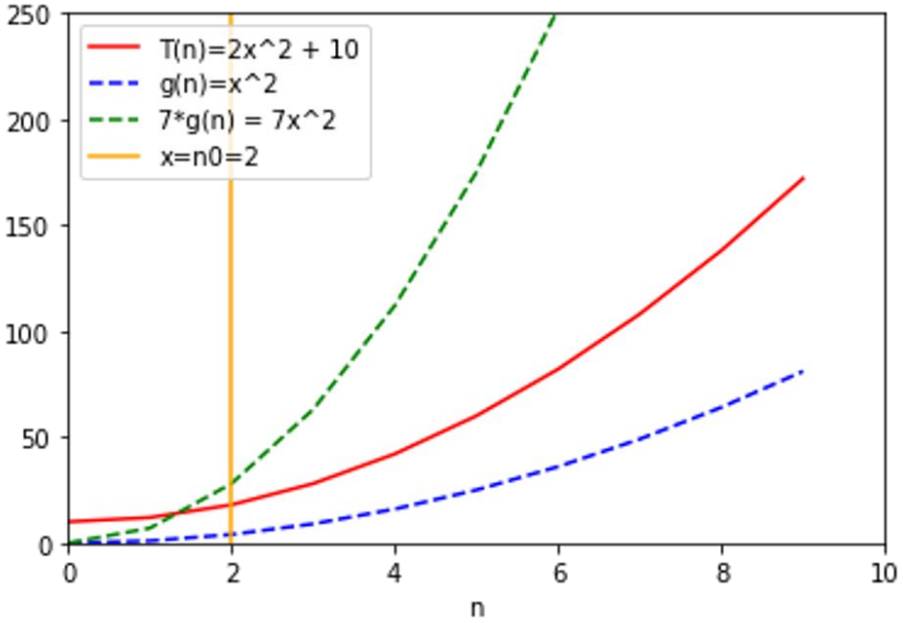
$$\underbrace{2n^2 + 10}_{T(n)} = O(\underbrace{n^2}_{g(n)})$$

$\Leftrightarrow$

c = 7,
n_0 = 2

$$T(n) = O(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \ \ s.t. \ \ \forall n \geq n_0, \ T(n) \leq c \cdot g(n)$$

$2n^2 + 10 <= 3n^2$ for all n > 4.



$$\underbrace{2n^2 + 10}_{T(n)} = O(\underbrace{n^2}_{g(n)})$$

$\Leftrightarrow$

c = 7,
n_0 = 2

$$\forall n \geq 2,$$
$$2n^2 + 10 \leq 7 \cdot n^2$$

O(...) is an upper bound

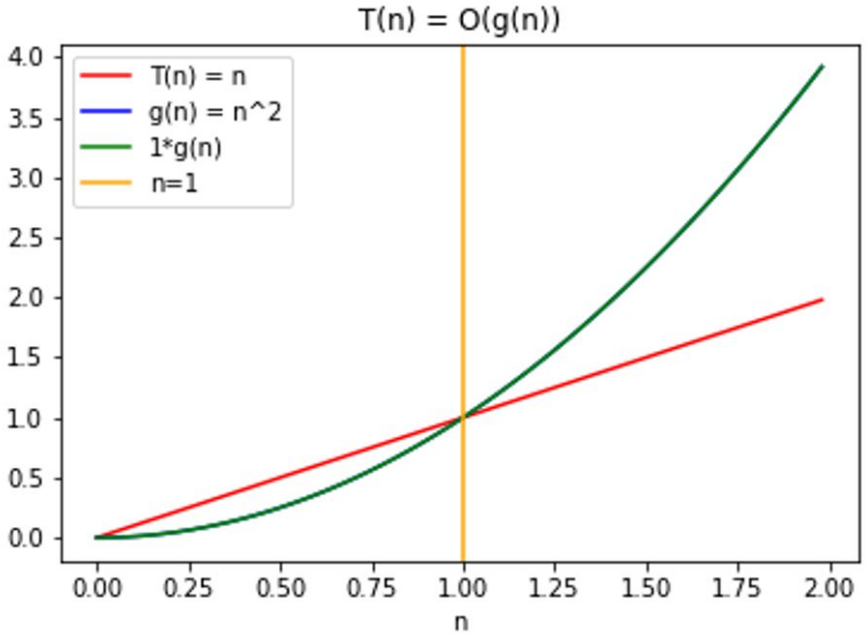
$$T(n) = O(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \ \ s.t. \ \ \forall n \geq n_0,$$
$$T(n) \leq c \cdot g(n)$$

Show that n = $O(n^2)$

O(...) is an upper bound

$$T(n) = O(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \ \ s.t. \ \ \forall n \geq n_0,$$
$$T(n) \leq c \cdot g(n)$$

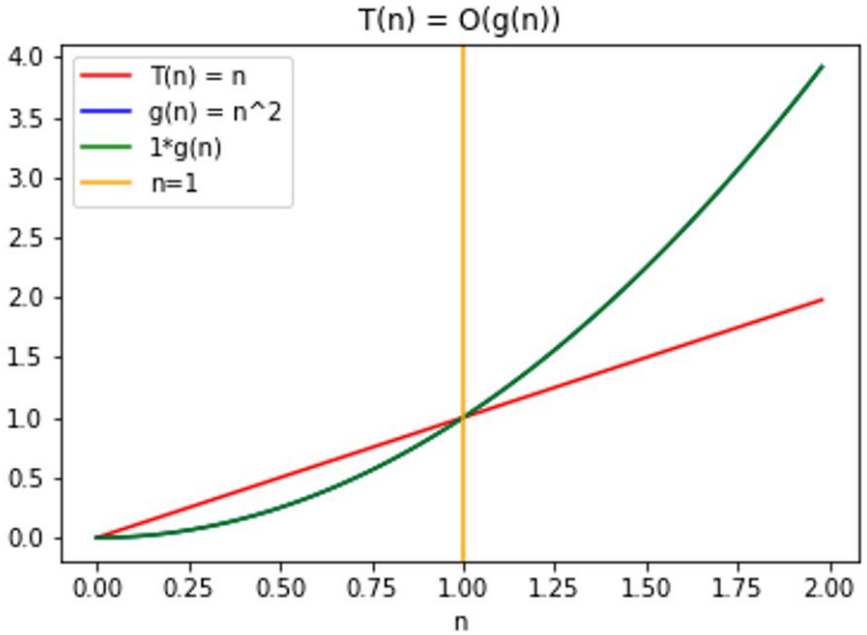
Show that n = O(n²)



O(...) is an upper bound

$$T(n) = O(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \ \ s.t. \ \ \forall n \geq n_0,$$
$$T(n) \leq c \cdot g(n)$$

Show that n = O(n²)

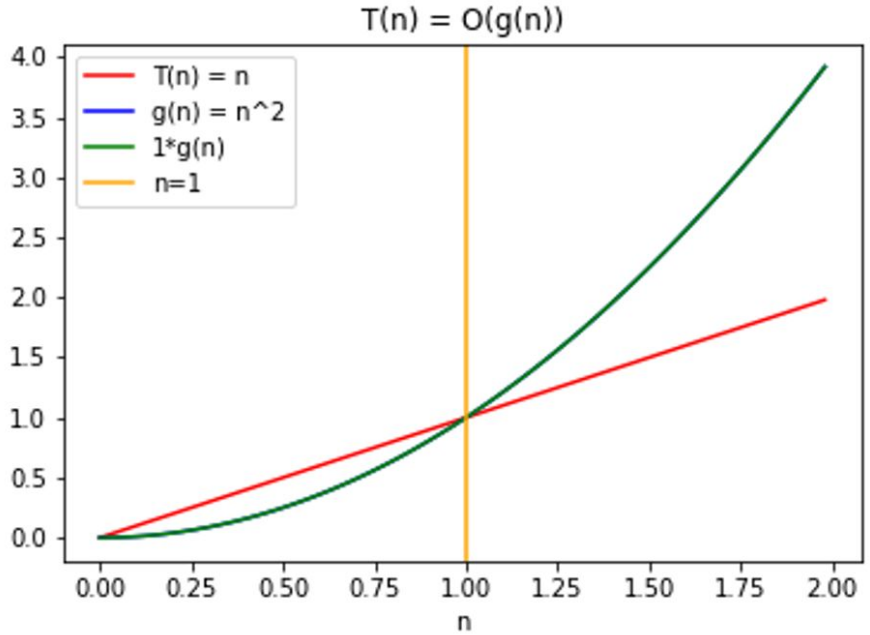


- Formally:
  - Choose c = 1
  - Choose n_0 = 1
  - Then:

O(...) is an upper bound

$$T(n) = O(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \ \ s.t. \ \ \forall n \geq n_0,$$
$$T(n) \leq c \cdot g(n)$$

Show that n = O(n²)



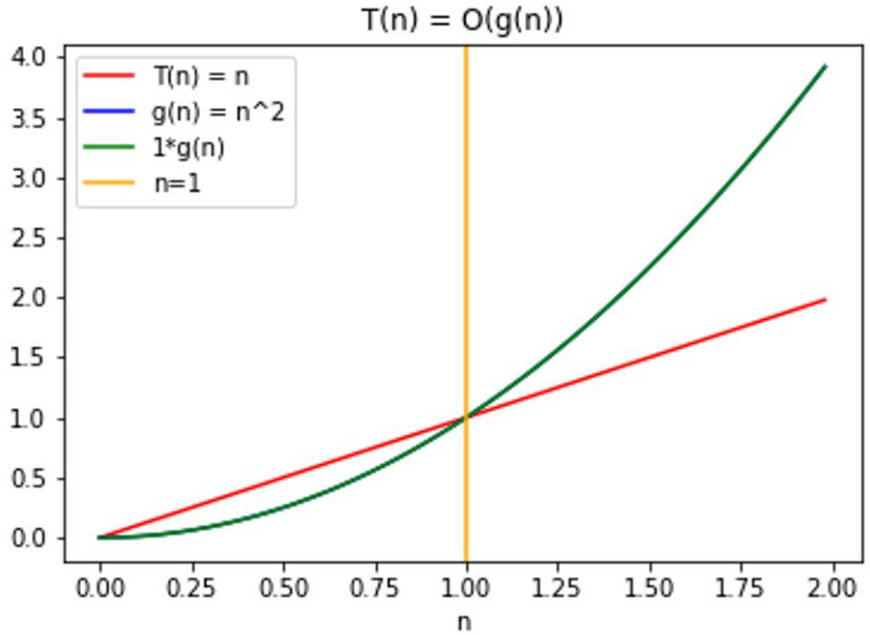
- Formally:
  - Choose c = 1
  - Choose n_0 = 1
  - Then:

$$\forall n \geq 1,$$
$$n \leq n^2$$

O(...) is an upper bound

$$T(n) = O(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \ \ s.t. \ \ \forall n \geq n_0,$$
$$T(n) \leq c \cdot g(n)$$

Show that n = O(n²)



- Formally:
  - Choose c = 1
  - Choose n_0 = 1
  - Then:

$$\forall n \geq 1,$$

$$n \leq n^2$$

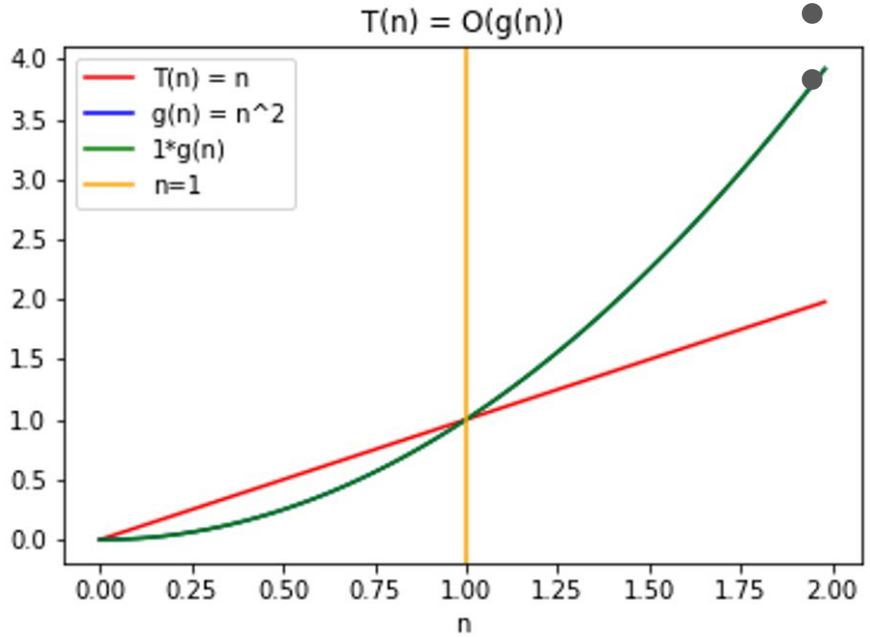
The ≤ highlights that

- **“T is upper-bounded by g”**
- **“T grows no faster than g”**

O(...) is an upper bound

$$T(n) = O(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \ \ s.t. \ \ \forall n \geq n_0,$$
$$T(n) \leq c \cdot g(n)$$

Show that n = O(n^2)



- Note: n^2 is **not a tight** upper-bound on n. When we talked about Big-O in the previous examples, we were always talking about the **tight** upper-bound.
  - It's important though to realize that n = O(n^2) is still a true mathematical statement.
  - But, **just because it's true, does not mean it is the most useful.**

Poll

If $g(n) = 1000n$ and $f(n) = n^2$, which of the following are true of g?

1. $g = O(n)$
2. $g = O(f)$
3. Both
4. Neither

Poll

If $g(n) = 1000n$ and $f(n) = n^2$, which of the following are true of g?

1. $g = O(n)$
2. $g = O(f)$
3. Both
4. Neither

Black-tie Formal Introduction (pt. 2)

$O(n^2)$

$\Theta(n^2)$

$\Omega(n^2)$

SIGACT News 18 Apr.-J

BIG OMICRON AND BIG OMEGA AND BIG THET

Donald E. Knuth
Computer Science Department
Stanford University
Stanford, California 94305

Most of us have gotten accustomed to the idea of using the notation O(f(n)) to stand for any function whose magnitude is upper-bounded by a constant times f(n), for all large n. Sometimes we also need a corresponding notation for lower-bounded functions, i.e., those functions which are at least as large as a constant times f(n) for all large n. Unfortunately, people have occasionally been using the O-notation for lower bounds, for example when they reject a particular sorting method "because its running time is $O(n^2)$." I have seen instances of this in

Ω(...) means a lower bound

- We say "T(n) is Ω(g(n))" if, for large enough n, T(n) is at least as big as a constant multiple of g(n).

Ω(...) means a lower bound

- We say "T(n) is Ω(g(n))" if, for large enough n, T(n) is at least as big as a constant multiple of g(n).
- Formally,

$$T(n) = \Omega\big(g(n)\big)$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$c \cdot g(n) \leq T(n)$$

Ω(...) means a lower bound

- We say "T(n) is Ω(g(n))" if, for large enough n, T(n) is at least as big as a constant multiple of g(n).
- Formally,

$$T(n) = \Omega(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \ \ s.t. \ \ \forall n \geq n_0,$$
$$c \cdot g(n) \leq T(n)$$

$$T(n) = O(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \ \ s.t. \ \ \forall n \geq n_0,$$
$$T(n) \leq c \cdot g(n)$$

We swap them! In big-Oh, we have an upper bound.

In big-Omega, we have a lower bound.

Let's do an example!

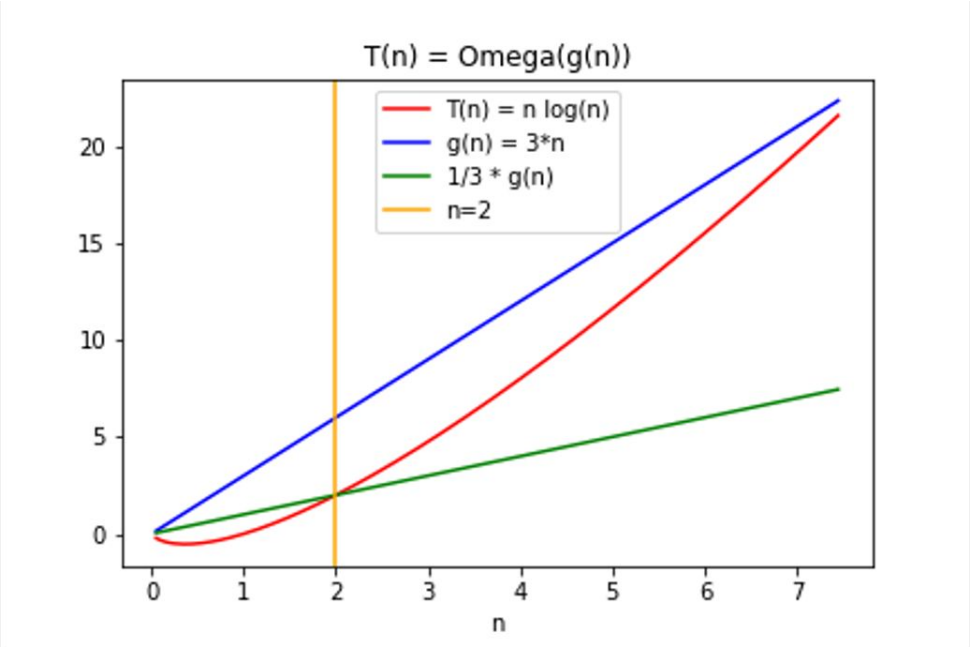
$$T(n) = \Omega\big(g(n)\big)$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$c \cdot g(n) \leq T(n)$$

$$n \log_2(n) = \Omega(3n)$$

Let's do an example!

$$T(n) = \Omega(g(n)) \iff \exists c, n_0 > 0 \ \ s.t. \ \ \forall n \geq n_0, \ c \cdot g(n) \leq T(n)$$

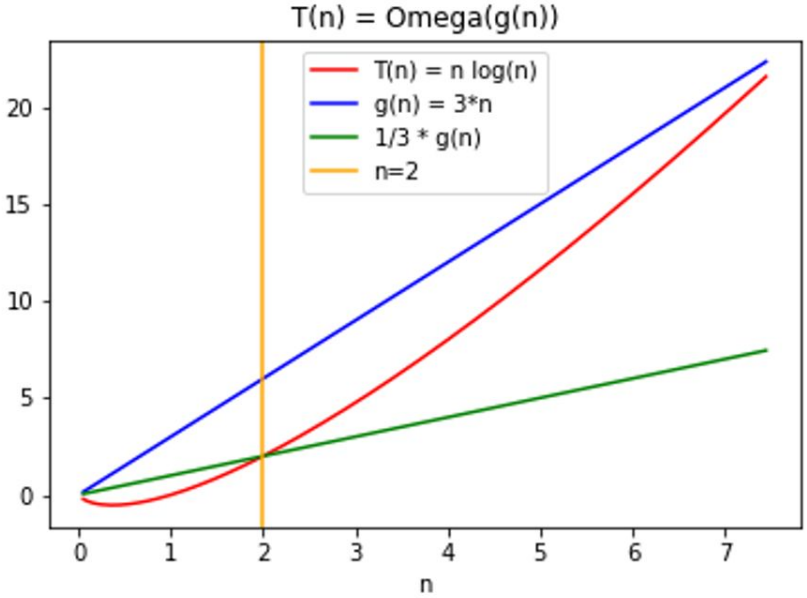
$$n \log_2(n) = \Omega(3n)$$



$$T(n) = \Omega(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \ \ s.t. \ \ \forall n \geq n_0,$$
$$c \cdot g(n) \leq T(n)$$

Let's do an example!

$$n \log_2(n) = \Omega(3n)$$



- Formally:
  - Choose c = 1/3
  - Choose $n_0 = 2$
  - Then:

$$\forall n \geq 2,$$

$$\frac{3n}{3} \leq n \log_2(n)$$

Black-tie Formal Introduction (pt. 3)

$$O(n^2)$$

$$\Theta(n^2)$$

$$\Omega(n^2)$$

SIGACT News 18 Apr.-J

BIG OMICRON AND BIG OMEGA AND BIG THET

Donald E. Knuth
Computer Science Department
Stanford University
Stanford, California 94305

Most of us have gotten accustomed to the idea of using the notation O(f(n)) to stand for any function whose magnitude is upper-bounded by a constant times f(n), for all large n. Sometimes we also need a corresponding notation for lower-bounded functions, i.e., those functions which are at least as large as a constant times f(n) for all large n. Unfortunately, people have occasionally been using the O-notation for lower bounds, for example when they reject a particular sorting method "because its running time is $O(n^2)$." I have seen instances of this in

Θ(...) means both!

- We say "$T(n)$ is $\Theta(g(n))$" iff both:

$$T(n) = O\big(g(n)\big)$$

and

$$T(n) = \Omega\big(g(n)\big)$$

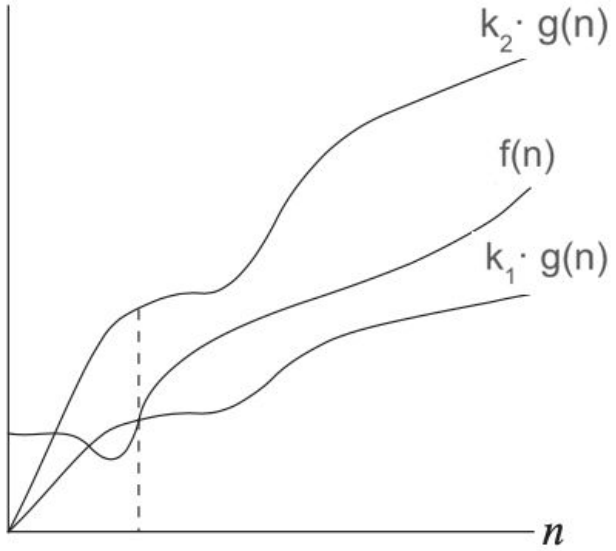
Θ(...) means both!

- We say "$T(n)$ is $\Theta(g(n))$" iff both:

$$T(n) = O(g(n))$$

and

$$T(n) = \Omega(g(n))$$



Notes

- "g is a tight-bound of T"
- This captures what we've been after when we're talking about Big-O

Poll

Which of the following pairs of functions does not meet the criteria that $f = \Theta(g)$?

1. $f = n+10, g = 20n$
2. $f = n^2+n, g = \log(n) + 5n^2$
3. $f = 10000n, g = 20n+3$
4. $f = 20n^2+n, g = n^3+n^2$

Poll

Which of the following pairs of functions does not meet the criteria that $f = \Theta(g)$?

1. $f = n+10, g = 20n$
2. $f = n^2+n, g = \log(n) + 5n^2$
3. $f = 10000n, g = 20n+3$
4. $f = 20n^2+n, g = n^3+n^2$

Why Haven't We Been Using Θ instead of O?

If we care about tight-bounds, why do engineers use O (upper-bound) when Θ (tight-bound) may have been more informative?

- Indeed, when discussing each of best-, worst-, or average-case, Θ should be used more often.
- But, when talking about the runtime of an algorithm overall (without specifying which case we are talking about), Θ can only be correctly used if best-case = worst-case, which isn't always true.
- So, to play it safe/lazy and not think of whether best-case = worst-case, people usually just describe the worst case (what we care about most often) with O instead of Θ (as O will be true regardless of whether or not the best-case = worst-case).
- That being said, this notation is used more rigorously in academia / research to analyze algorithms.
- But, your industry interviewer might not even fully understand the distinction between these symbols!

Example where you would have to be careful not to use Θ

```
algorithm linearSearch:
  input: vec of ints and a target
  output: whether or not target is in vec

for each element v in vec
     if v == target
          return true
return false
```

Example where you would have to be careful not to use Θ

```
algorithm linearSearch:
  input: vec of ints and a target
  output: whether or not target is in vec

for each element v in vec
     if v == target
          return true
return false
```

linearSearch in the worst-case is Θ(N)

linearSearch in the best-case is Θ(1)

linearSearch [in all cases] is Θ(N)

linearSearch [in all cases] is O(N)

Example where you would have to be careful not to use Θ

```
algorithm linearSearch:
  input: vec of ints and a target
  output: whether or not target is in vec

for each element v in vec
     if v == target
          return true
return false
```

✓ **linearSearch in the worst-case is Θ(N)**

linearSearch in the best-case is Θ(1)

linearSearch [in all cases] is Θ(N)

linearSearch [in all cases] is O(N)

Example where you would have to be careful not to use Θ

```
algorithm linearSearch:
  input: vec of ints and a target
  output: whether or not target is in vec

for each element v in vec
     if v == target
          return true
return false
```

✓ **linearSearch in the worst-case is Θ(N)**

✓ **linearSearch in the best-case is Θ(1)**

linearSearch [in all cases] is Θ(N)

linearSearch [in all cases] is O(N)

Example where you would have to be careful not to use Θ

```
algorithm linearSearch:
  input: vec of ints and a target
  output: whether or not target is in vec

for each element v in vec
     if v == target
          return true
return false
```

linearSearch in the worst-case is Θ(N)

linearSearch in the best-case is Θ(1)

linearSearch [in all cases] is Θ(N)

linearSearch [in all cases] is O(N)

Example where you would have to be careful not to use Θ

```
algorithm linearSearch:
  input: vec of ints and a target
  output: whether or not target is in vec

for each element v in vec
     if v == target
          return true
return false
```

linearSearch in the worst-case is Θ(N)

linearSearch in the best-case is Θ(1)

linearSearch [in all cases] is Θ(N)

linearSearch [in all cases] is O(N)

Example where you would have to be careful not to use Θ

```
algorithm linearSearch:
  input: vec of ints and a target
  output: whether or not target is in vec
answer = false
for each element v in vec
     if v == target
          answer = true
return answer
```

linearSearch in the worst-case is Θ(N)

linearSearch in the best-case is Θ(1)

linearSearch [in all cases] is Θ(N)

linearSearch [in all cases] is O(N)

Example where you would have to be careful not to use Θ

```
algorithm linearSearch:
  input: vec of ints and a target
  output: whether or not target is in vec
answer = false
for each element v in vec
     if v == target
          answer = true
return answer
```

linearSearch in the worst-case is Θ(N)

linearSearch in the best-case is Θ(1)

linearSearch [in all cases] is Θ(N)

linearSearch [in all cases] is O(N)

Why Haven't We Been Using Θ instead of O?

If we care about tight-bounds, why do engineers use O (upper-bound) when Θ (tight-bound) may have been more informative?

- Indeed, when discussing each of best-, worst-, or average-case, Θ should be used more often.
- But, when talking about the runtime of an algorithm overall (without specifying which case we are talking about), Θ can only be correctly used if best-case = worst-case, which isn't always true.
- **So, to play it safe/lazy and not think of whether best-case = worst-case, people usually just describe the worst case (what we care about most often) with O instead of Θ (as O will be true regardless of whether or not the best-case = worst-case).**
- That being said, this notation is used more rigorously in academia / research to analyze algorithms.
- But, your industry interviewer might not even fully understand the distinction between these symbols!

Recap

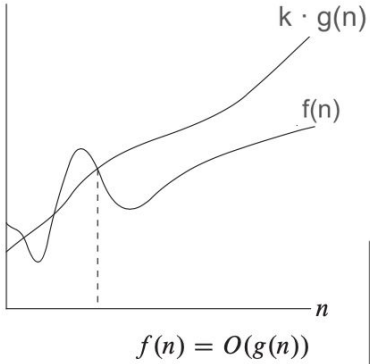
Upper-bound | f = O(g) is similar to f ≤ g
“f grows no faster than g”

Tight-bound | f = Θ(g) is similar to f = g
“f grows as fast as g”

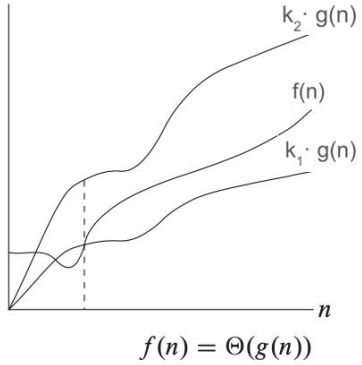
Lower-bound | f = Ω(g) is similar to f ≥ g
“f grows no slower than g”

If both f = O(g) and f = Ω(g), then f = Θ(g)
If f = O(g), then g = Ω(f)

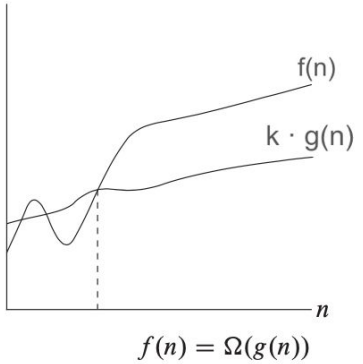
<u>Note</u>: O, Θ, and Ω are not the same thing as the best, worst, and average case. Do not confuse them (I’ve seen this mistake made in other academic materials!)



$f(n) = O(g(n))$



$f(n) = \Theta(g(n))$



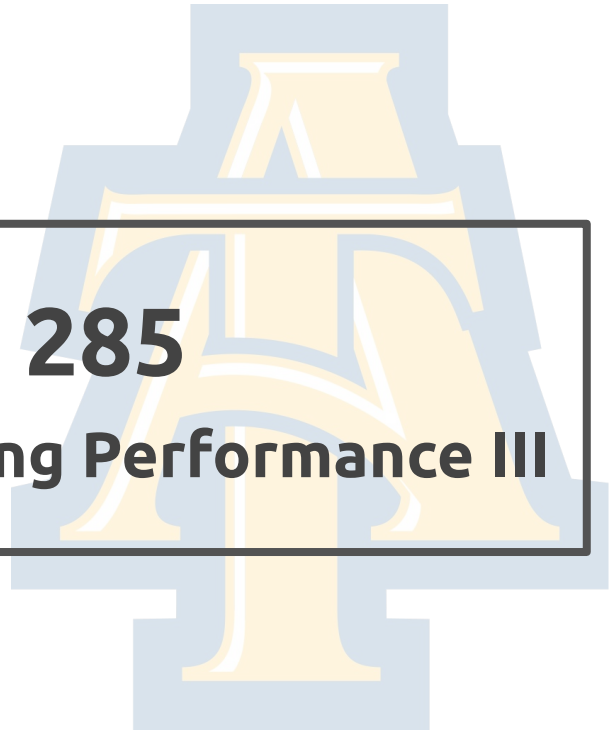
$f(n) = \Omega(g(n))$

COMP - 285

Advanced Analysis of Algorithms

Welcome to COMP 285

Lecture 4: (Formal) Measuring Performance III



Chris Lucas (cflucas@ncat.edu)