

Advanced Data Structure and Algorithm

Asymptotic Analysis

The plan

- **Sorting Algorithms**

- InsertionSort: does it work and is it fast?
- MergeSort: does it work and is it fast?
- Skills:
 - Analyzing correctness of iterative and recursive algorithms.
 - Analyzing running time of recursive algorithms



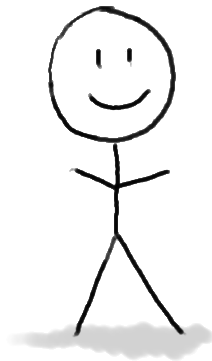
- **How do we measure the runtime of an algorithm?**

- Worst-case analysis
- Asymptotic Analysis

Worst-case analysis

Sorting a sorted list
should be fast!!

The “running time” for an algorithm is its
running time on the **worst possible input**.



Algorithm
designer

Here is my algorithm!

Algorithm:
Do the thing
Do the stuff
Return the answer



**HERE IS AN INPUT!
(WHICH I DESIGNED
TO BE TERRIBLE FOR
YOUR ALGORITHM!)**

Big-O notation



- What do we mean when we measure runtime?
 - We probably care about wall time: how long does it take to solve the problem, in seconds or minutes or hours?
- This is heavily dependent on the programming language, architecture, etc.
- These things are very important, but are **not the point of this class.**
- We want a way to talk about the running time of an algorithm, **independent of these considerations.**

Main idea:

Focus on how the runtime **scales** with n (the input size).

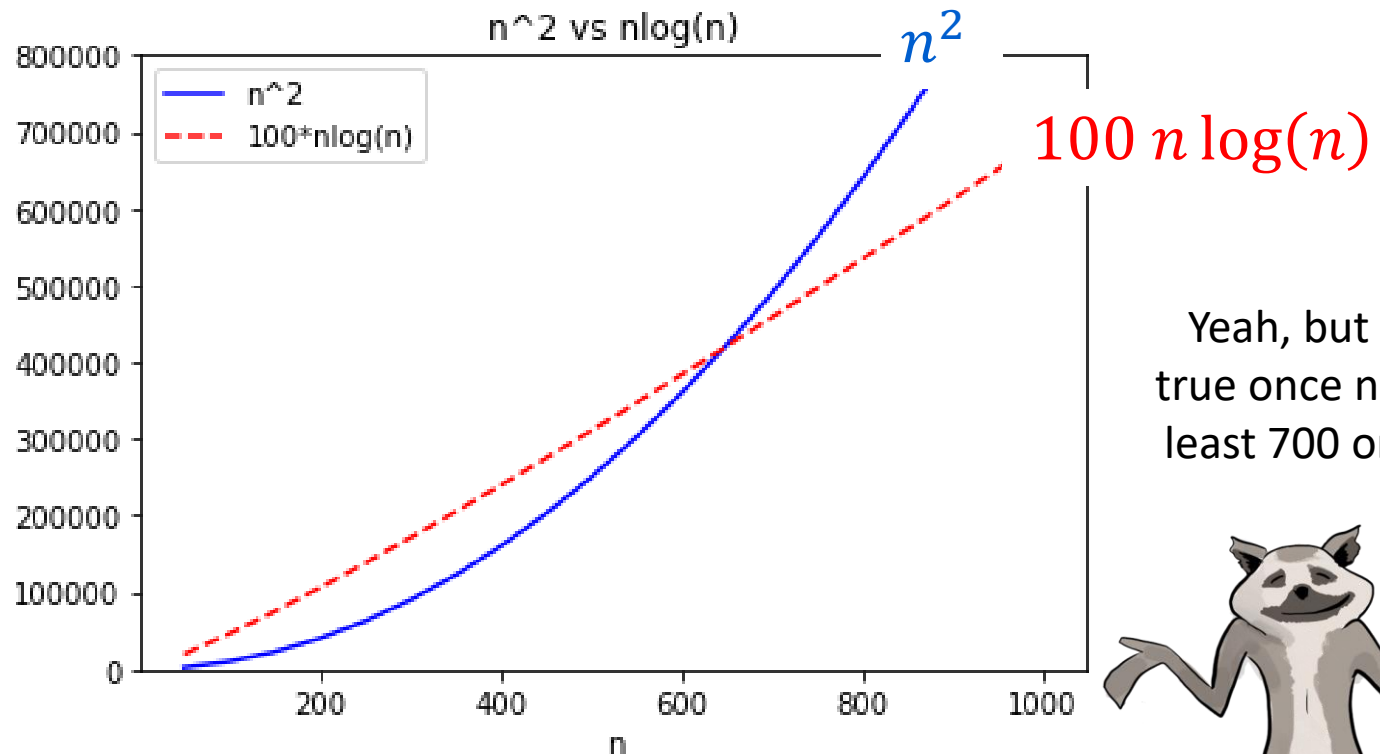
Informally....

(Only pay attention to the largest function of n that appears.)

| Number of operations | Asymptotic Running Time |
|---|-------------------------|
| $\frac{1}{10} \cdot n^2 + 100$ | $O(n^2)$ |
| $0.063 \cdot n^2 - .5n + 12.7$ | $O(n^2)$ |
| $100 \cdot n^{1.5} - 10^{10000} \sqrt{n}$ | $O(n^{1.5})$ |
| $11 \cdot n \log(n) - 1$ | $O(n \log(n))$ |

We say this algorithm is “asymptotically faster” than the others.

So $100 n \log(n)$ operations is
“better” than n^2 operations?



But when
 $n=200$, that's
not true at all!



Yeah, but it's
true once n is at
least 700 or so.



Asymptotic Analysis

One algorithm is “faster” than another if its runtime scales better with the size of the input.

Pros:

- Abstracts away from hardware- and language-specific issues.
- Makes algorithm analysis much more tractable.

Cons:

- Only makes sense if n is large (compared to the constant factors).

$1000000000 n$
is “better” than n^2 ?!?!

pronounced “big-oh of ...” or sometimes “oh of ...”

$O(\dots)$ means an upper bound

- 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 $O(g(n))$ ” if $T(n)$ grows no faster than $g(n)$ as n gets large.
- Formally,

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

$$\Leftrightarrow$$

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

$$0 \leq T(n) \leq c \cdot g(n)$$

Example

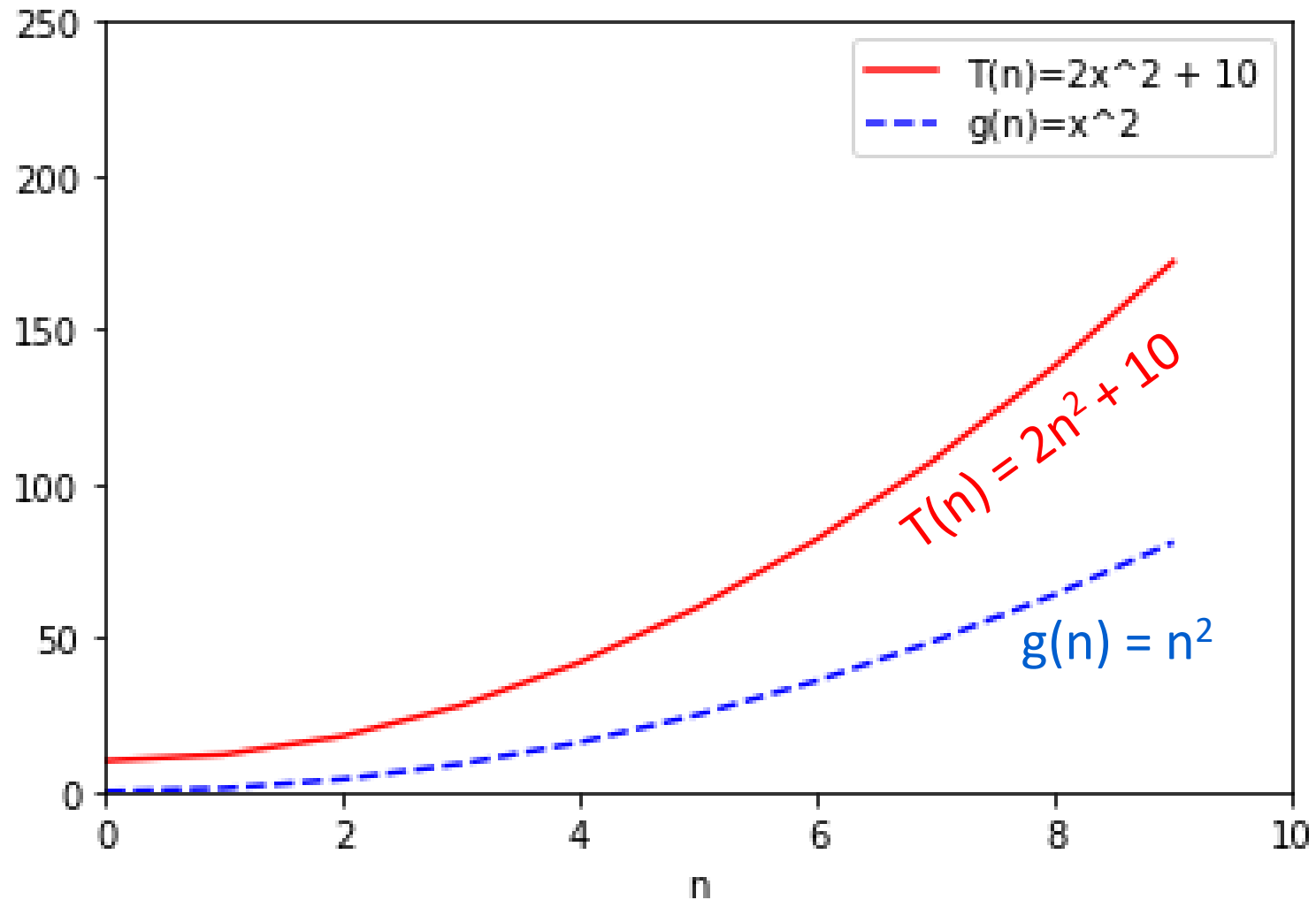
$$2n^2 + 10 = O(n^2)$$

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

$$\Leftrightarrow$$

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

$$0 \leq T(n) \leq c \cdot g(n)$$



Example

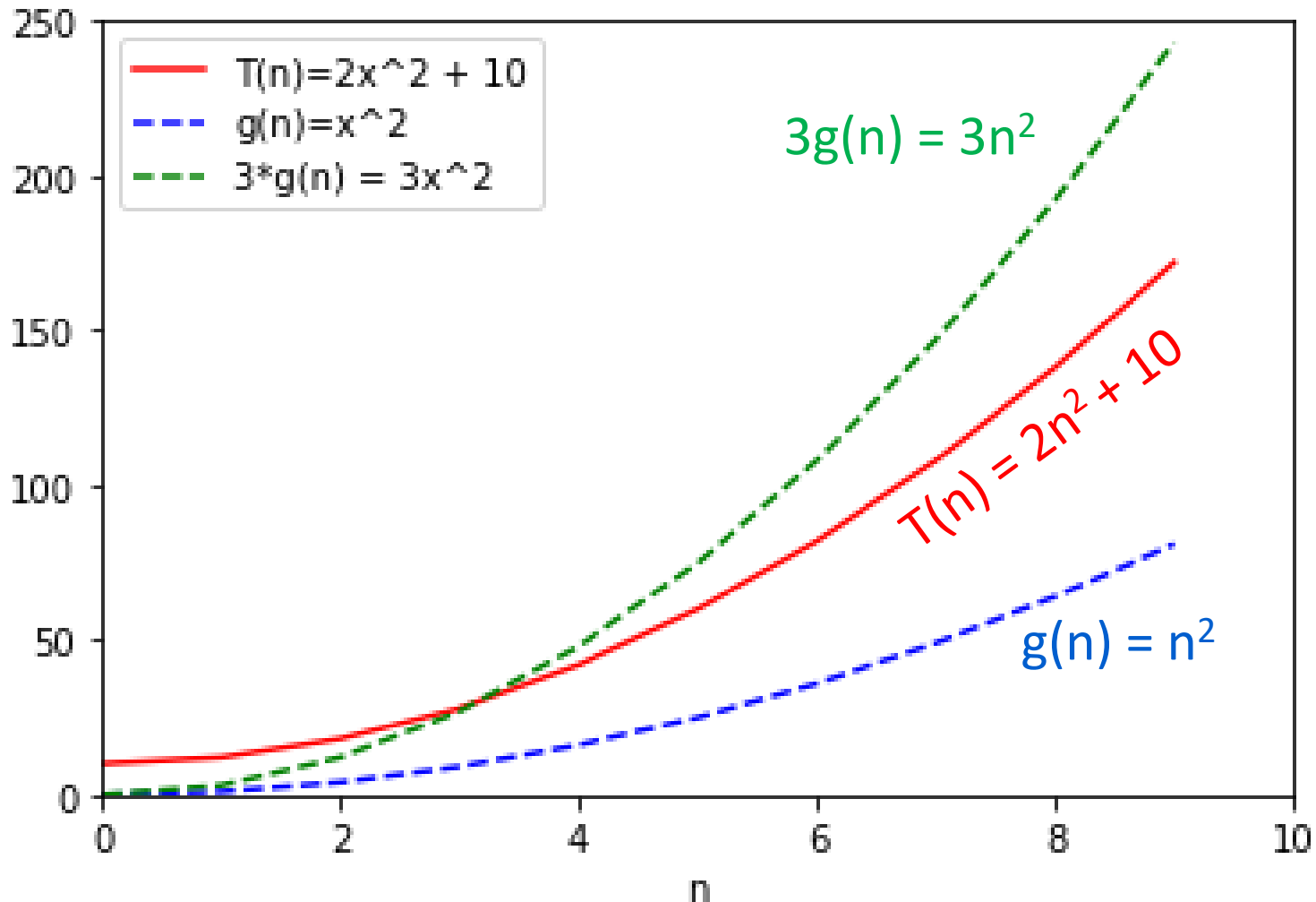
$$2n^2 + 10 = O(n^2)$$

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

$$\Leftrightarrow$$

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

$$0 \leq T(n) \leq c \cdot g(n)$$



Example

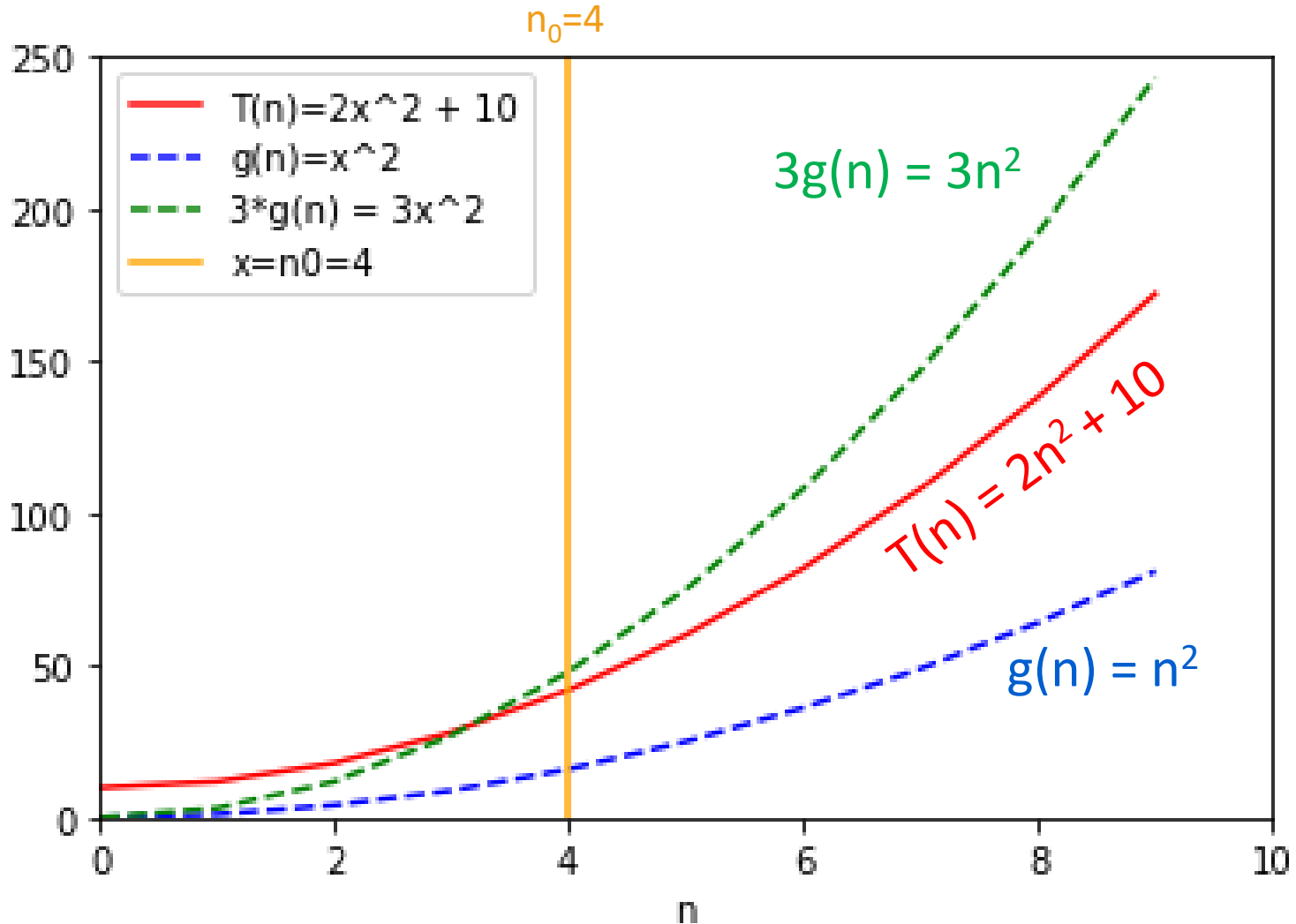
$$2n^2 + 10 = O(n^2)$$

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

$$\Leftrightarrow$$

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

$$0 \leq T(n) \leq c \cdot g(n)$$



Example

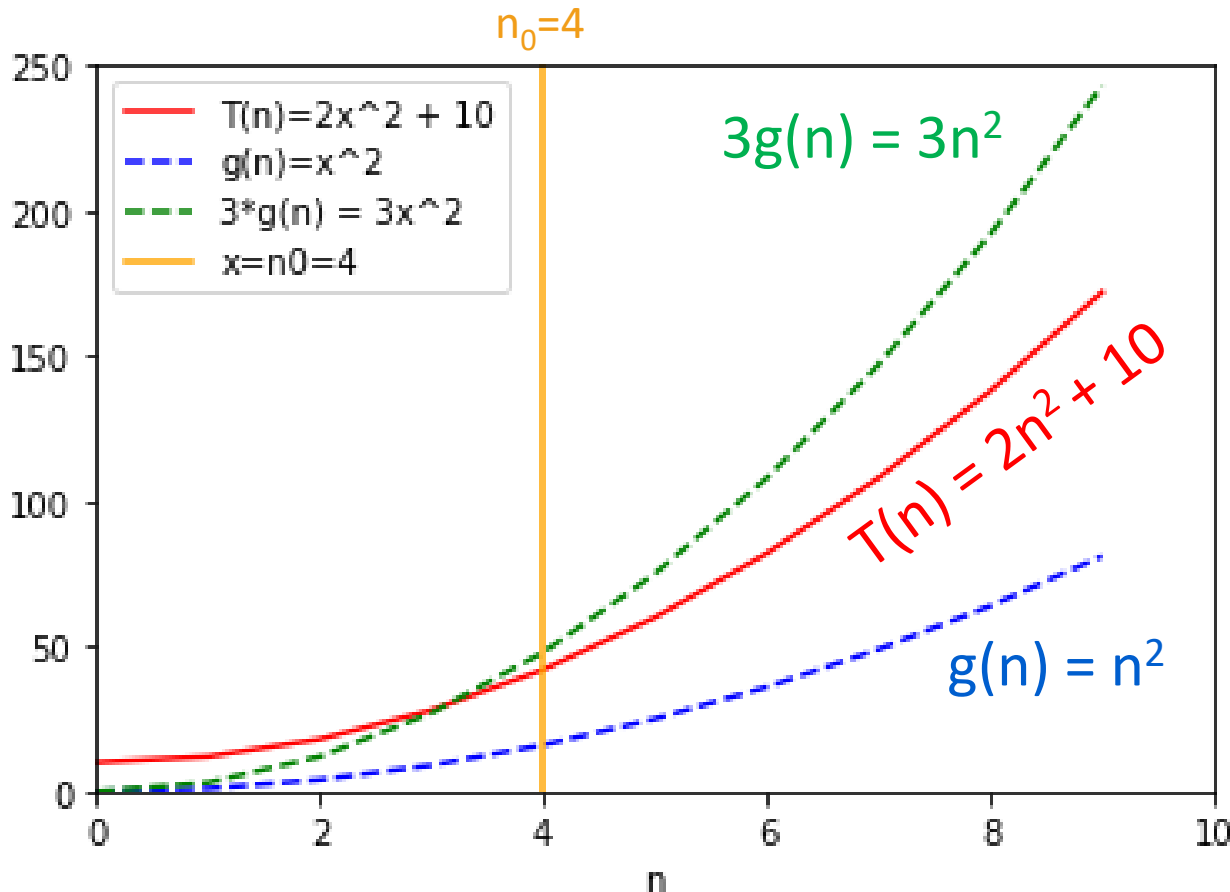
$$2n^2 + 10 = O(n^2)$$

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

\Leftrightarrow

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

$$0 \leq T(n) \leq c \cdot g(n)$$



Formally:

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

$$\forall n \geq 4,$$

$$0 \leq 2n^2 + 10 \leq 3 \cdot n^2$$

Same example

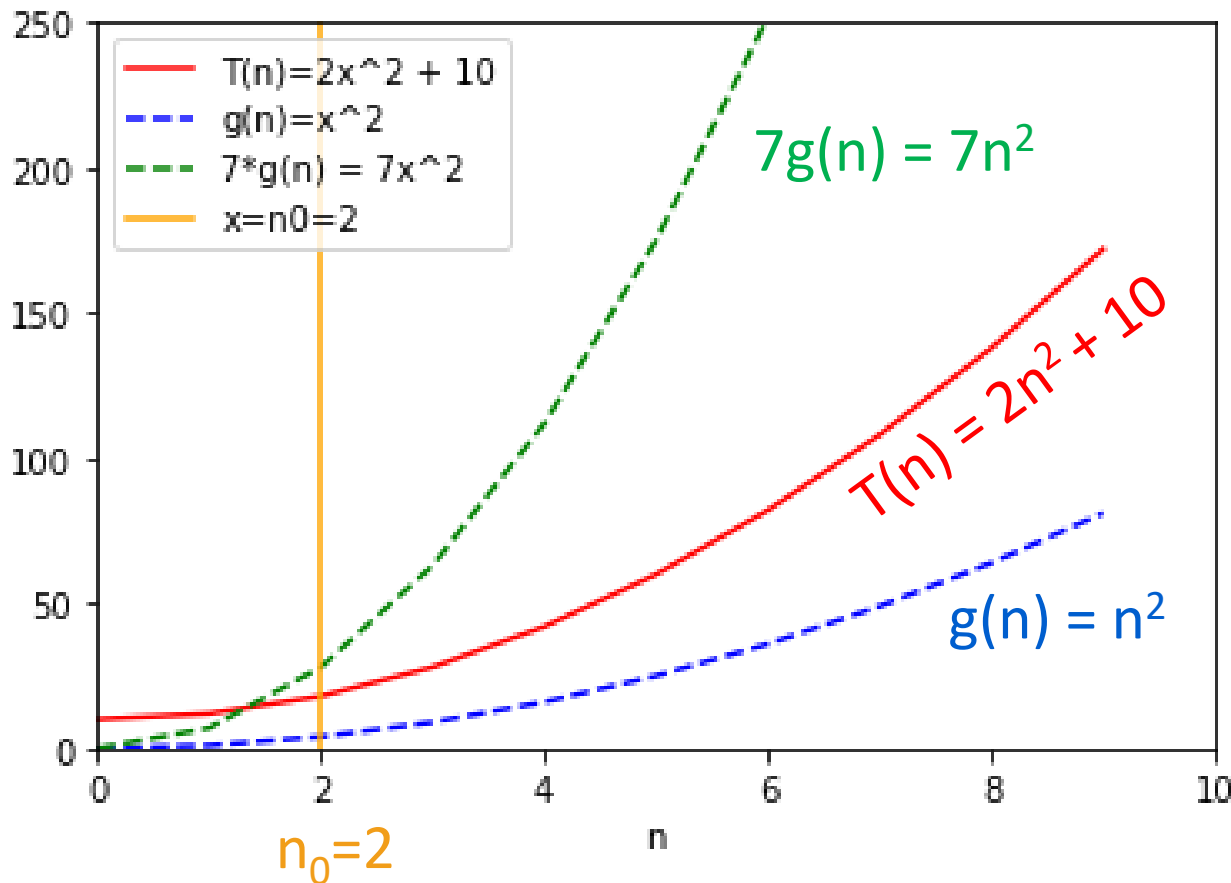
$2n^2 + 10 = O(n^2)$

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

\Leftrightarrow

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

$$0 \leq T(n) \leq c \cdot g(n)$$



Formally:

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

$$\forall n \geq 2,$$

$$0 \leq 2n^2 + 10 \leq 7 \cdot n^2$$

There is not a
“correct” choice
of c and n_0

$\Omega(\dots)$ means a lower bound

- We say “ $T(n)$ is $\Omega(g(n))$ ” if $T(n)$ grows at least as fast as $g(n)$ as n gets large.
- Formally,

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

$$\Leftrightarrow$$

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

$$0 \leq c \cdot g(n) \leq T(n)$$

Switched these!!

Example

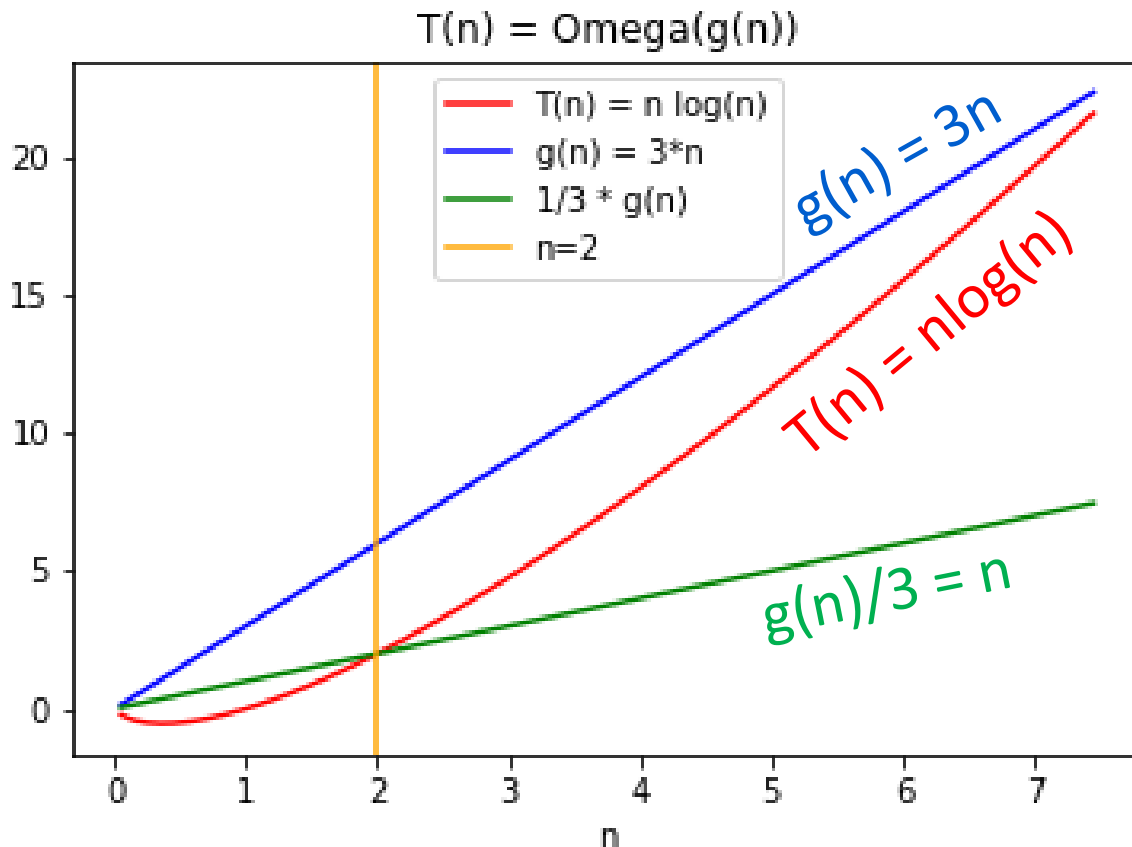
$n \log_2(n) = \Omega(3n)$

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

$$\Leftrightarrow$$

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

$$0 \leq c \cdot g(n) \leq T(n)$$



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

$$\forall n \geq 2,$$

$$0 \leq \frac{3n}{3} \leq n \log_2(n)$$

$\Theta(\dots)$ means both!

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

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

and

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

Example: polynomials

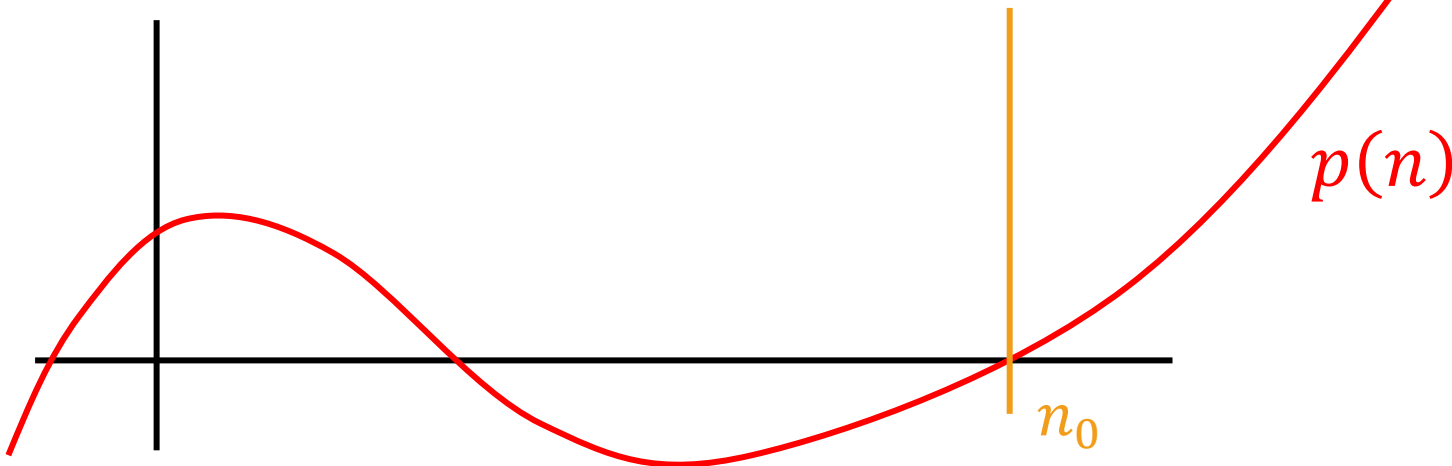
- Suppose the $p(n)$ is a polynomial of degree k :

$$p(n) = a_0 + a_1n + a_2n^2 + \cdots + a_kn^k \text{ where } a_k > 0.$$

- Then $p(n) = O(n^k)$

- Proof:

- Choose $n_0 \geq 1$ so that $p(n) \geq 0$ for all $n \geq n_0$.
- Choose $c = |a_0| + |a_1| + \cdots + |a_k|$



Example: polynomials

- Suppose the $p(n)$ is a polynomial of degree k :

$$p(n) = a_0 + a_1n + a_2n^2 + \cdots + a_kn^k \text{ where } a_k > 0.$$

- Then $p(n) = O(n^k)$

- Proof:

- Choose $n_0 \geq 1$ so that $p(n) \geq 0$ for all $n \geq n_0$.

- Choose $c = |a_0| + |a_1| + \cdots + |a_k|$

- Then for all $n \geq n_0$:

$$\begin{aligned} 0 \leq p(n) = |p(n)| &\leq |a_0| + |a_1|n + \cdots + |a_k|n^k \\ &\leq |a_0|n^k + |a_1|n^k + \cdots + |a_k|n^k \\ &= c \cdot n^k \end{aligned}$$

Definition of c

Because $n \leq n^k$
for $n \geq n_0 \geq 1$.

Example: more polynomials

- For any $k \geq 1$, n^k is **NOT** $O(n^{k-1})$.
- Proof:
 - Suppose that it were. Then there is some c, n_0 so that
$$n^k \leq c \cdot n^{k-1} \text{ for all } n \geq n_0$$
 - Aka, $n \leq c$ for all $n \geq n_0$
 - But that's not true!
 - We have a contradiction! It *can't* be that $n^k = O(n^{k-1})$.

Take-away from examples

- To prove $T(n) = O(g(n))$, you have to come up with c and n_0 so that the definition is satisfied.
- To prove $T(n)$ is **NOT** $O(g(n))$, one way is **proof by contradiction**:
 - Suppose (to get a contradiction) that someone gives you a c and an n_0 so that the definition *is* satisfied.
 - Show that this someone must be lying to you by deriving a contradiction.

Yet more examples

- $n^3 + 3n = O(n^3 - n^2)$
- $n^3 + 3n = \Omega(n^3 - n^2)$
- $n^3 + 3n = \Theta(n^3 - n^2)$

- 3^n is **NOT** $O(2^n)$
- $\log(n) = \Omega(\ln(n))$
- $\log(n) = \Theta(2^{\log \log(n)})$

Work through these
on your own!



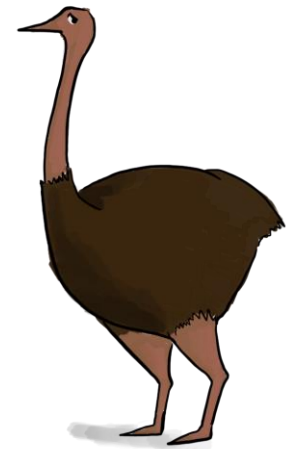
remember that $\log = \log_2$ in this class.

Some brainteasers

- Are there functions f, g so that **NEITHER** $f = O(g)$ nor $f = \Omega(g)$?
- Are there **non-decreasing** functions f, g so that the above is true?
- Define the n 'th fibonacci number by $F(0) = 1$, $F(1) = 1$, $F(n) = F(n-1) + F(n-2)$ for $n > 2$.
 - 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

True or false:

- $F(n) = O(2^n)$
- $F(n) = \Omega(2^n)$



Recap

- InsertionSort runs in time $O(n^2)$
- MergeSort is a divide-and-conquer algorithm that runs in time $O(n \log(n))$
- How do we show an algorithm is correct?
 - Today, we did it by induction
- How do we measure the runtime of an algorithm?
 - Worst-case analysis
 - Asymptotic analysis

Next time

- A more systematic approach to analyzing the runtime of recursive algorithms.

Acknowledgement

- Stanford University