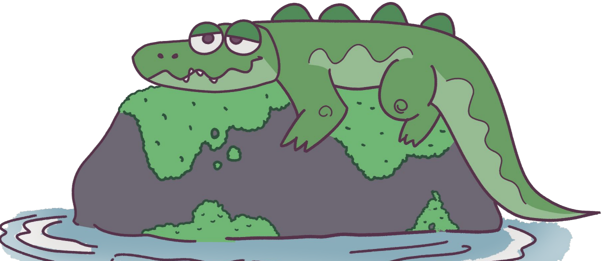
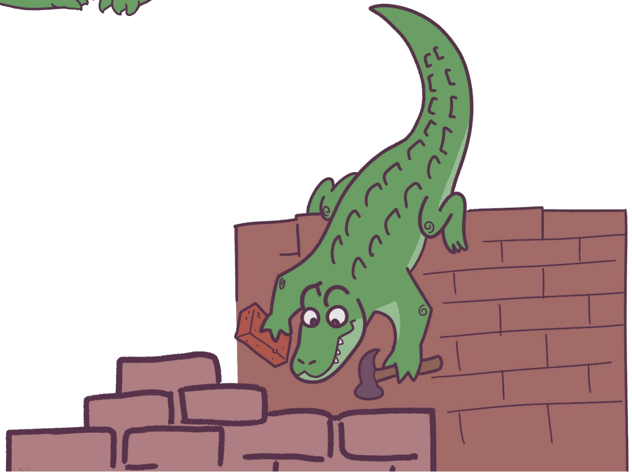
CS161: Design And Analysis of ~~Alligators~~ Algorithms

Summer 2022

Ian Tullis





6/20 Lecture Agenda

- Part 0: Course overview and policies
- 10 minute break!
- Part 1-1: Big-O and friends

6/20 Lecture Agenda

- Part 0: Course overview and policies
- 10 minute break!
- Part 1-1: Big-O and friends

Why are you here? Why take CS161?

Some reasons, maybe from less to more interesting?

- You might have to for your degree
- Heard it was useful for technical interviews?
- Algorithms are beautiful and fun!
- Algorithms can solve – and, uh, cause – pressing societal problems!

A Word On Tech Interviews

Good news: CS161 will help (to some extent!)

- Practice with designing algorithms / problem-solving
- Intro to some topics like **dynamic programming** which are overrepresented in tech interviews

Bad news? CS161 is not an interview prep class

- We have CS9 for that (I will probably teach it this Autumn)
- Tech interviews are their own weird, broken thing
- Not everyone wants to go into industry...

Algorithms and Society

What is "The Algorithm"?

And why does it

- deny me a loan
- hide my social media posts
- boost sensational and false news stories

What does that have to do with, like, sorting a list?

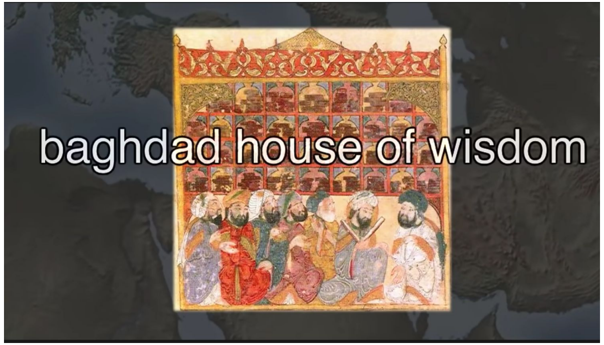
What even is an "algorithm"?

A process for solving a problem!

Name derives from Muḥammad ibn Mūsā al-Khwārizmī

- Headed the Grand Library of Baghdad (House of Wisdom) in the 9th century!
- Also gave us algebra

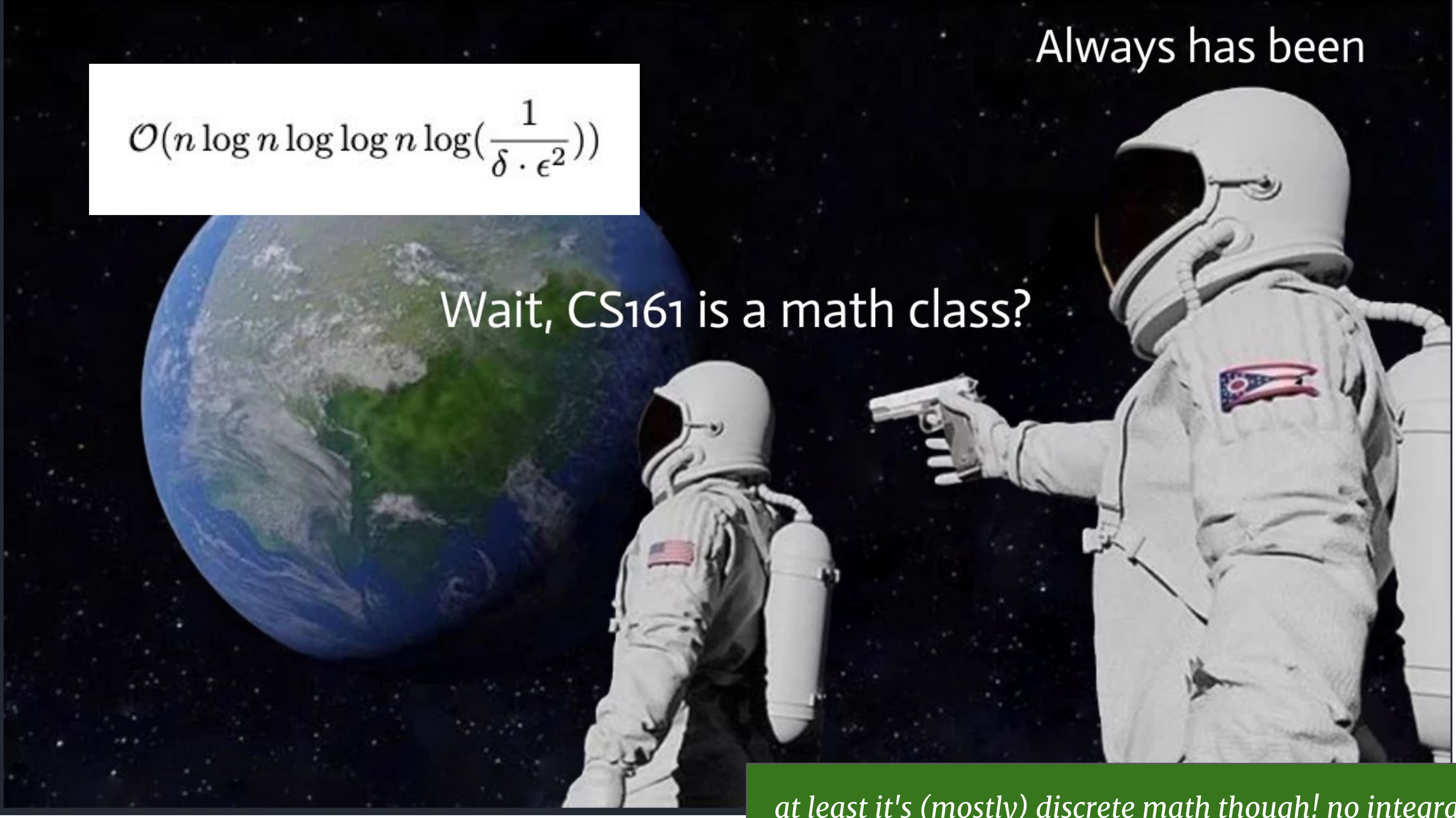
The Compendious Book on Calculation by Completion and Balancing, AKA *Algebra*



All of these are algorithms

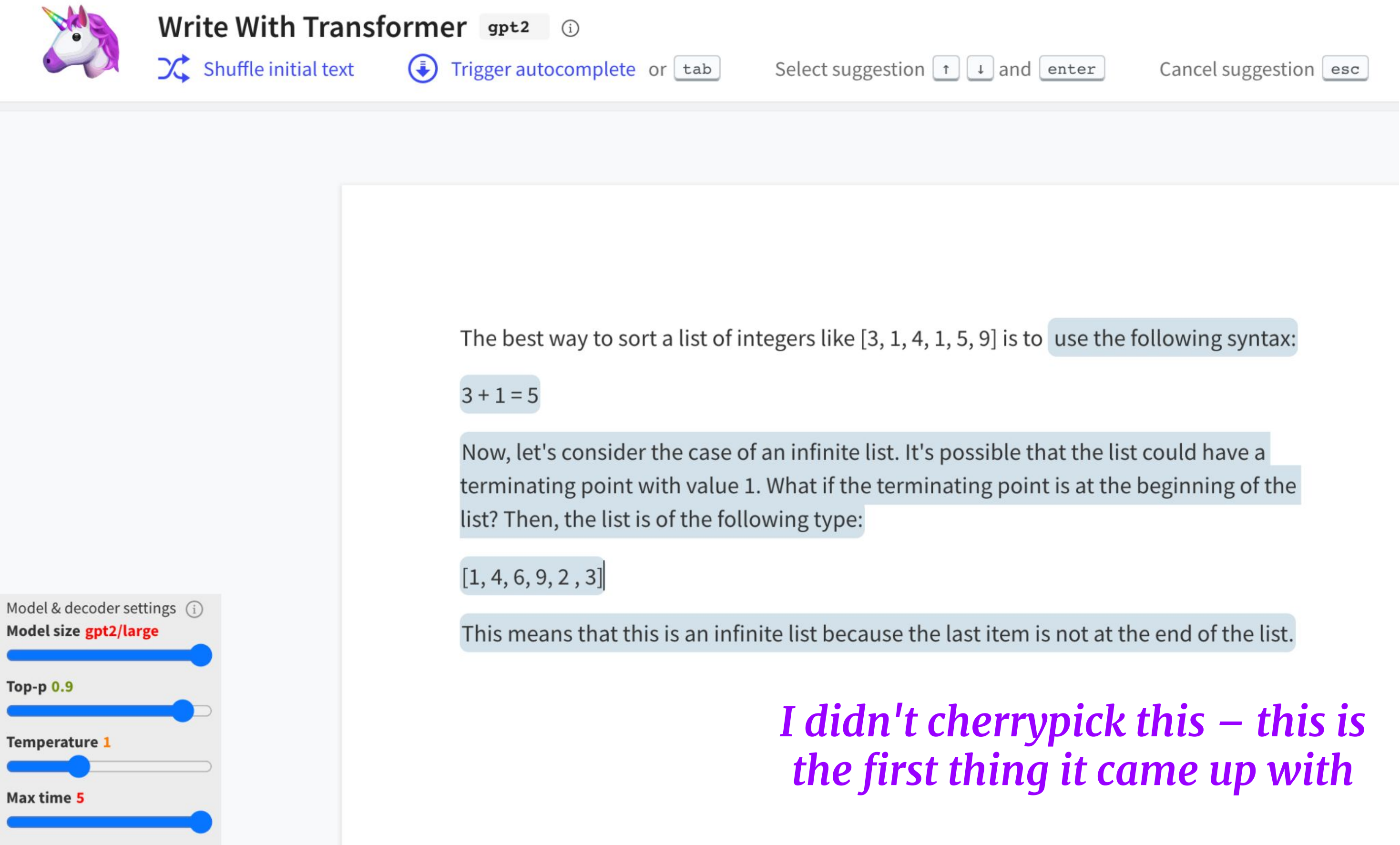
- Solving a quadratic equation with a formula (al-Khwārizmī)
- Finding a quotient via long division
- Sorting a list of integers using MergeSort
- Finding the fastest route between two places using Dijkstra's Algorithm
- Detecting spam using Naive Bayes modeling
- Deciding who should be released early from jail, using a complicated and probably biased model

} CS161 is here, roughly



at least it's (mostly) discrete math though! no integrals!

Why bother? Isn't CS just AI/ML now?



I didn't cherrypick this – this is the first thing it came up with

OK, that wasn't fair, but

- ML is offering some great insights into math and algorithms, but not every problem should be solved by throwing it into TensorFlow

- In CS161, we'll mostly study **very fast, deterministic** algorithms that produce **exact** solutions to **well-defined** problems

- But there's a bigger world out there too, e.g.:
  - **Intractable problems** for which we suspect that no efficient algorithm can possibly exist
  - Problems that don't fit on one machine and must be **distributed**
  - **Ambiguous problems** (like modeling climate change)
  - **Randomized algorithms** – is it OK to be wrong if we can try again?
  - **Approximation algorithms** – is a close solution good enough?
  - **Quantum algorithms** that are dark magic and break all the rules...

Think of CS161 as a "classic" toolbox

Spanning trees

MergeSort, QuickSort, Radix Sort

Heaps and priority queues

Topological sort

BFS, DFS

Universal hashing

Dynamic programming

Median and Selection

Divide and conquer

Dijkstra's algorithm

Self-balancing binary trees

Big-O

Greedy

Bloom filters

Minimax

Max flow and bipartite matching

But there are always more toolboxes!

CS 168: The Modern Algorithmic Toolbox

This course will provide a rigorous and hands-on introduction to the central ideas and algorithms that constitute the core of the modern algorithms toolkit. Emphasis will be on understanding the high-level theoretical intuitions and principles underlying the algorithms we discuss, as well as developing a concrete understanding of when and how to implement and

Why not just jump right to the modern toolbox?

- Well, people still use hammers all the time, right?

- (The algorithms we will learn in CS161 are still relevant!)

[Chorus]
And I am downright amazed at what I can destroy with just a hammer
And I am downright amazed at what I can destroy with just a hammer

Quora

Ian's Digest

TOP STORIES FOR YOU

Is AI/ML all about mathematics? If you take out the infrastructure required to handle computation, I find AI to be mostly computational maths. Do things like greedy algorithms, sorting and other cool CS things get applied here?

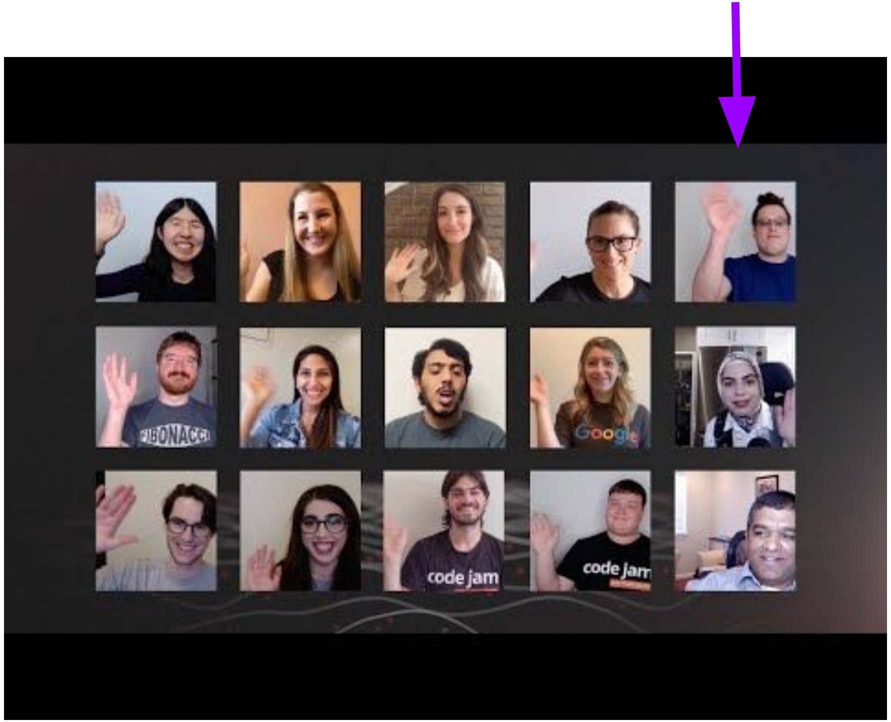
see, at least one person on Quora still thinks sorting algorithms are cool!

Me

- Weird background in chemistry, biology, environmental science, premodern Japanese literature (yes, I am a parody of myself)

- I love CS theory, math (especially discrete math and combinatorics), and AI!

- Worked at Google for 8 years on Search and then Code Jam (an algorithm coding contest)



Our awesome course staff

bios will be on the site!

Goli

Ivan

Lucas

Ricky

Rishu

Ziang

Some more shout-outs

- This class is professionally recorded!
  - Be *very* thankful that it's not me doing it
- I'll be borrowing some slides (and the mascots idea) from Mary Wootters, who is awesome
  - Seriously, take all her classes

Course Policies

- See the syllabus (linked from Canvas and cs161.stanford.edu) for full details!
- Overall theme: I believe we learn by *doing*, not just by listening
  - So, I want to provide lots of practice problems and opportunities, but not in an overwhelming or stressful way

Prereqs

- CS103 (mathematical foundations)
- CS106B (coding, basic data structures)
- CS109 (probability)

- As in general at Stanford, these are not "firm" prereqs. In this case I (mostly) agree

- See the Prereq Review notes I've put together for recaps of the most crucial topics. (I also take requests, but may not get to them ASAP)

Organization

- Six "units"

- Each has:
  - two lectures (2 halves each)
  - an optional Problem Session
  - a Pre-Homework
  - a Homework

- Two exams

Divide and Conquer
Sorting & Randomization
Data Structures
Graph Search
Dynamic Programming
Greed & Flow

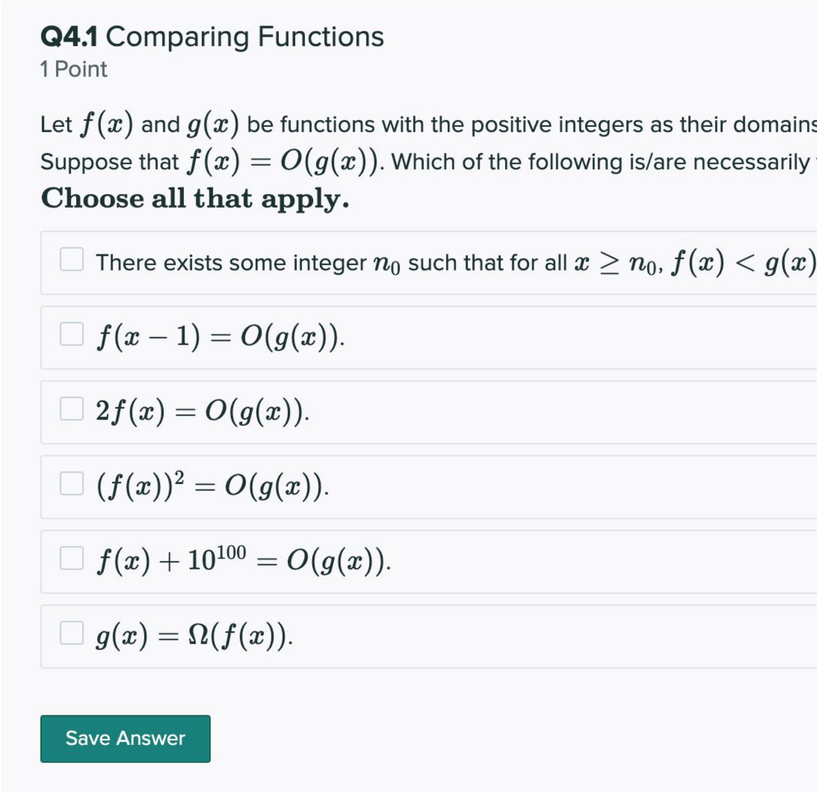
Special Topics

Problem Sessions

- Optional meetings (same room, same time of day, usually Fridays but sometimes Mondays)
- Solo or in groups, work through problems at your own pace. We'll circulate to help!
- We will post the problems and detailed solutions. The sessions will not be recorded since there is kinda nothing to record, and the solutions should be self-contained.

Pre-Homeworks

- Work on these on Gradescope
- Multiple-choice questions, may be quite challenging!
- You can try as many times as you want and will get immediate feedback, and a full explanation when correct
- Collaboration is OK!
- Ask for help on Ed / from staff!



Q4.1 Comparing Functions
1 Point

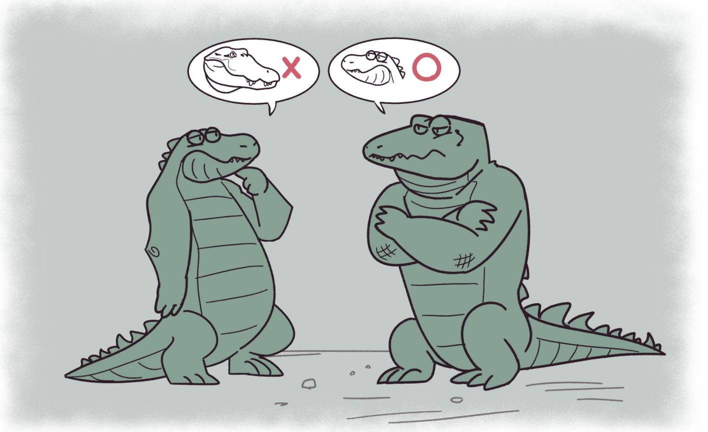
Let $f(x)$ and $g(x)$ be functions with the positive integers as their domains
Suppose that $f(x) = O(g(x))$. Which of the following is/are necessarily

Choose all that apply.

- [ ] There exists some integer n_0 such that for all $x \geq n_0$, $f(x) < g(x)$
- [ ] $f(x-1) = O(g(x))$.
- [ ] $2f(x) = O(g(x))$.
- [ ] $(f(x))^2 = O(g(x))$.
- [ ] $f(x) + 10^{100} = O(g(x))$.
- [ ] $g(x) = \Omega(f(x))$.

Save Answer

Homeworks



- Problem sets to ponder / write up
- 6 problems of equal weight
  - Always 1 coding problem
  - You can get full points from doing 5, but can do all 6!
- Collaboration is OK **but**
  - you cannot look at anyone else's solutions/code (or any online), and you must write up your own work
- We are here to help! (in office hours, on Ed...)
- 6 late days for Pre-HW/HW, max of 2 per assignment, see syllabus for details

A Word on Coding Problems...

- Still in development (since these are uncommon for CS161)
    - After all, we are a CS class, even if this is mostly math!
- Goal: practice implementing algorithms so that they work in practice and not just on paper
- Current plans are to support C++ and Python, but let us know if you do not know either language
- On Gradescope, autograded for immediate feedback (but test cases will not be visible)
- One per problem set. In theory you could skip all of them and still get full homework points

Exams

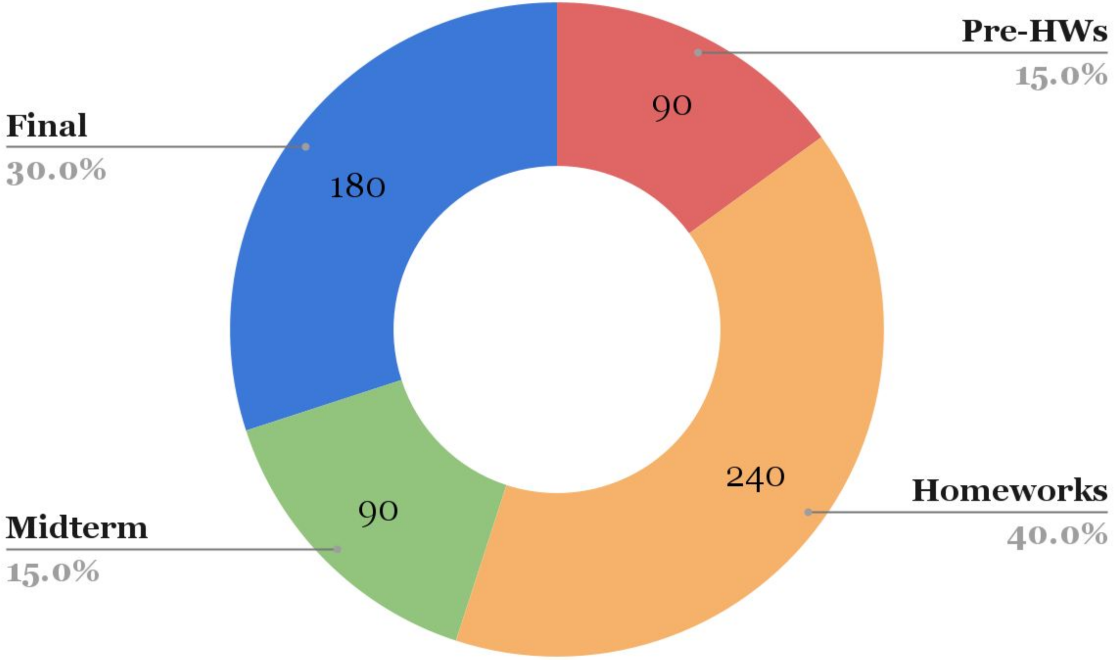
- Midterm: In class, July 22, covers Units 1, 2, 3 (and 4, in less depth)

- Final: August 12, 3:30-6:30 PM, covers entire class including Special Topics (more emphasis on Units 4, 5, 6)

- In scope: anything from the lectures, Pre-Homeworks, and Homeworks (though we won't ask about tiny details from these)

- Exams will be challenging to allow you to demonstrate mastery of the material, but not gratuitously hard to create a curve or whatever

Grading



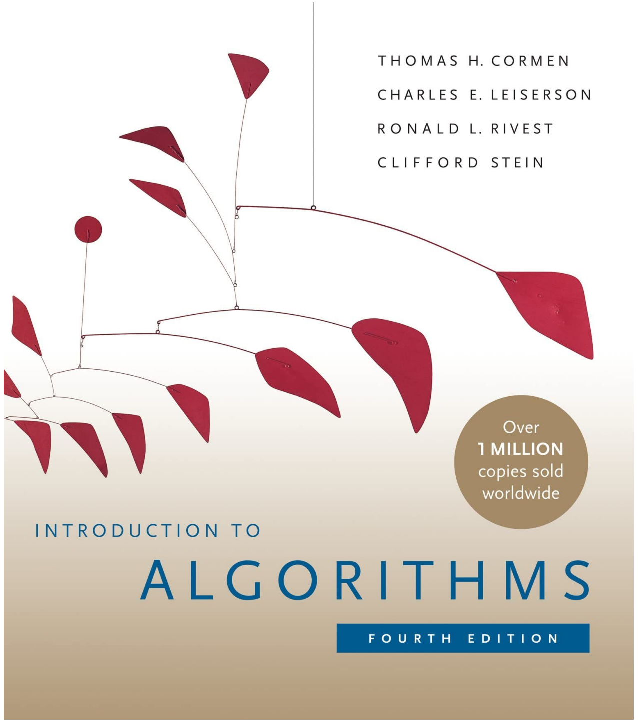
- Pre-HW + HW points over 330 become bonus (at ⅓ value)
- Also bonus for Ed contributions etc. (total bonus capped at 24pts)
- Final grade will be based on performance and **not** on a planned curve. We will give an estimate after the midterm

A Plea Re: Grades from a survivor of gifted-kid burnout

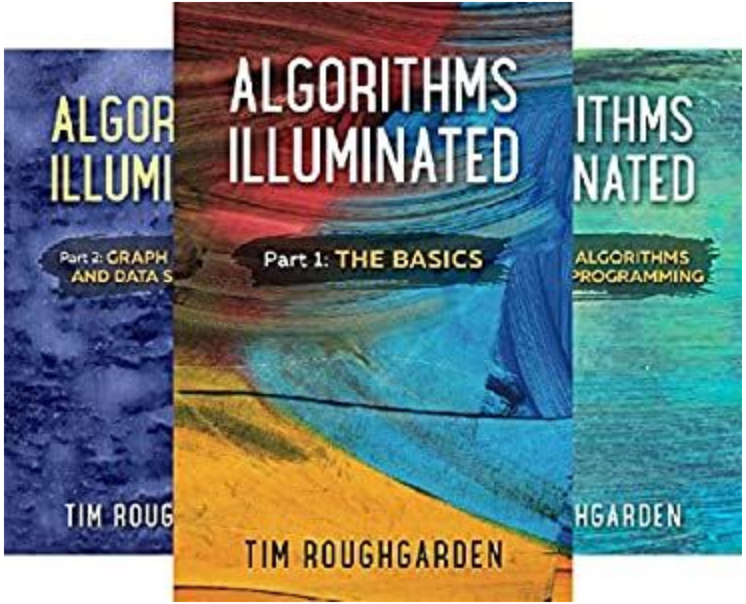
- We are trained to focus on grades to get into college, I get that, but…
  - Grades are not a measure of personal worth or even of potential in a field
  - Many employers (especially in tech) don't care about GPA
  - High grades from playing the game + lack of understanding is a bad combination

- Focus on learning and understanding, and the grades will follow (not the other way around!)

Textbooks (optional)



CLRS: the classic, but *buckle up*



AI: more easily digestible, by one of the best algorithms teachers Stanford has ever had

Other resources / Advice for success

- Go to **office hours.** (Schedule coming soon!)
- Post on our **Ed forum.** When something doesn't quite click, ask about it!
    - And don't be afraid to ask publicly (anonymously, if you prefer) – if you're confused, so are others. But private posts are OK.
- Get as much practice as you can! Attend the **Problem Sessions** (or work through the problems on your own, and read the solutions)
- Find a **study group**. (But try not to make the group too large.) There will be a thread about this on Ed.
- The Summer Academic Resource Center (SARC) may have free tutoring for CS161 (and other core classes). See https://summer.stanford.edu/summer-academic-resource-center-sarc

The Algorators!

- **BRUTUS** is the brute-force gator. Brutus is stronk. Brutus is in no danger of overthinking problems. Brutus is often in danger of underthinking problems.

Sometimes brute force really is the right approach! Easy to understand / maintain

The Algorators!

- **INDY** is the industry gator. "When would you ever use red/black trees?", he says, as he uses libraries based on red/black trees and asks candidates interview questions about red/black trees. Then he speeds home in his expensive car and rolls on his piles of quantum coins or whatever.

*I am going to poke gentle fun at industry in this course, but Indy is not **just** a comic figure. Also, he makes way more than I do.*

The Algorators!

- **SISI** is the systems gator. She is practical and cares more about implementations and speed than about abstract performance guarantees.

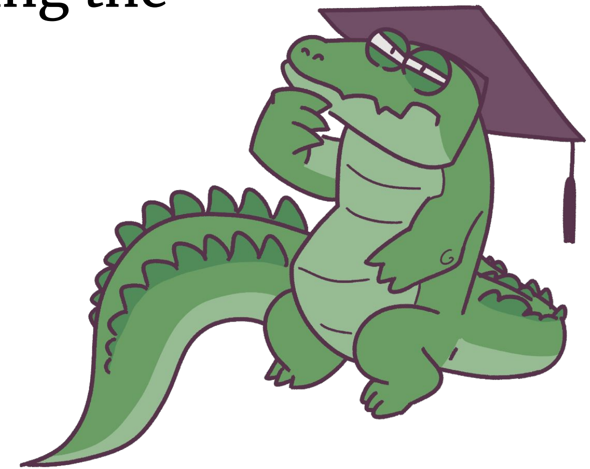
Sometimes CS theorists get spirited fully away into big-O land and stop thinking about practical concerns. Sisi will remind us.

The Algorators!

- **TERRY** is the theory / academic gator. They are passionate about proofs and fine details, sometimes to the point of exasperating the other Algorators.

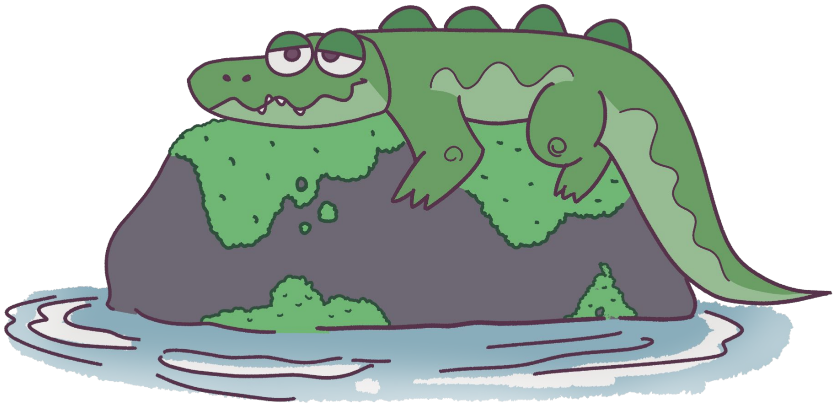
As we all know, academics wear their mortarboards around everywhere!

Terry can be pedantic, but hey, it's an algorithms class. ***Someone*** *has to be rigorous and not gloss over "minor" points.*



The Algorators!

- **WAVERLY** is the intuitive (some might say "handwavy") gator. She has a big-picture, intuitive understanding and does not like to get bogged down with extreme rigor.



Sometimes Terry can get so lost in the details that they miss high-level insights and ideas. Waverly explores and speculates and sometimes hands off a great new idea to Terry to examine thoroughly.

6/20 Lecture Agenda

- Part 0: Course overview and policies
- 10 minute break!
- Part 1-1: Big-O and friends

6/20 Lecture Agenda

- Part 0: Course overview and policies
- 10 minute break!
- Part 1-1: Big-O and friends

WORLD 1-1

Big-O and Friends

Divide and Conquer

Sorting & Randomization

Data Structures

Graph Search

Dynamic Programming

Greed & Flow

Special Topics

How Fast Does Our Code Run?

Suppose we are given a list L of integers, and we want to determine whether there are any repeated elements. One naive brute-force strategy is to check every pair of elements against each other.

```
i = 0
n = length(L)
while i < n - 1:
   j = i + 1
   while j < n:
      if L[i] == L[j]:
         return True
      j += 1
   i += 1
return False
```

This is "pseudocode" – i.e., not in any particular language, but readable by anyone familiar with at least one language. It has no fixed format, so don't worry about the specific syntax here.

Suppose that it takes 1 time unit to do any of these:

- initialize a value
- increment a value
- perform an addition/subtraction
- perform a comparison and react accordingly
- return a value
- find the length of a list
- access a list element

How long does this function take to run, depending on the size and content of the lists? Let's start with a simple example, the list [7, 6]...

```
i = 0
n = length(L)
while i < n - 1:
  j = i + 1
  while j < n:
    if L[i] == L[j]:
      return True
    j += 1
  i += 1
return False
```

- Initialize i to 0
- Find the length of L (2)
- Initialize n to 2
- Subtract 1 from 2 to get 1
- Compare i (0) and n-1 (1)
- Add i (0) and 1 to get 1
- Initialize j to 1
- Compare j (1) and n (2)
- Access L[0] to get 7
- Access L[1] to get 6
- Compare L[0] (7) and L[1] (6)
- Increment j to 2
- Compare j (2) and n (2)
- Increment i to 1
- Subtract 1 from 2 to get 1
- Compare i (1) and n-1 (1)
- Return False

*You will **not** have to do this on a HW or exam. It's a little gross. (don't tell the Software Theory people I said so)*

```
i = 0
n = length(L)
while i < n - 1:
   j = i + 1
   while j < n:
      if L[i] == L[j]:
         return True
      j += 1
   i += 1
return False
```

What A Mess!

- That was supposed to be a simple example, and look how complicated it got!

- Also, what if those operations don't really all take the same amount of time?
  - Some machine instructions are way more expensive than others!

- All we really want is some idea of how this function's running time depends on the size and content of the list, but here we're getting lost in details...

- And that was just for one input! What about others?

Pessimism to the Rescue

- One simplifying assumption we can make right away is that the contents of the list are **as bad as possible** for the algorithm.

- In this example, since this algorithm gets to quit early if it finds a duplicate, the worst case is for there to be no duplicates.

How Often Is Each Part Executed?

Here, for further simplicity, let's say each line takes the same time to execute (even if it includes both a subtraction and a comparison, for instance).

once	`i = 0`
once	`n = length(L)`
n times	`while i < n-1:`
n−1 times	`    j = i + 1`
n + (n−1) + … + 2 times	`    while j < n:`
(n−1) + (n−2) + … + 1 times	`        if L[i] == L[j]:`
never	`            return True`
(n−1) + (n−2) + … + 1 times	`        j += 1`
n−1 times	`    i += 1`
once	`return False`

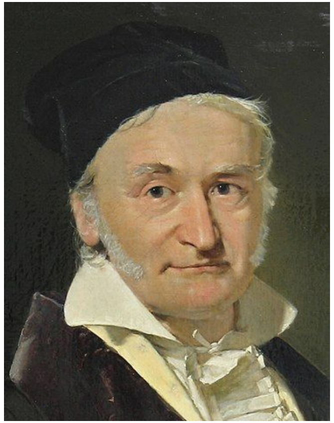
How Often Is Each Part Executed?

This stuff looks like it matters the most!

once	`i = 0`
once	`n = length(L)`
n times	`while i < n-1:`
$n-1$ times	`    j = i + 1`
$n + (n-1) + \ldots + 2$ times	`    while j < n:`
$(n-1) + (n-2) + \ldots + 1$ times	`        if L[i] == L[j]:`
never	`            return True`
$(n-1) + (n-2) + \ldots + 1$ times	`        j += 1`
$n-1$ times	`    i += 1`
once	`return False`

A Useful Math Fact

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + n = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$$



As a boy, Gauss was asked to add the integers from 1 to 100, and he observed that it's 50 pairs like the following: 1 + 100, 2 + 99, 3 + 98, ..., 50 + 51. So the answer is 50 times 101, i.e. n/2 times n+1. You're too freakin clever, Gauss! Leave some math for the rest of us!

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

Side note: This is lesser-known but also sometimes useful.

Totaling Up The Scorecard

once	`i = 0`
once	`n = length(L)`
n times	`while i < n-1:`
n−1 times	`    j = i + 1`
n + (n−1) + … + 2 times	`    while j < n:`
(n−1) + (n−2) + … + 1 times	`        if L[i] == L[j]:`
never	`            return True`
(n−1) + (n−2) + … + 1 times	`        j += 1`
n−1 times	`    i += 1`
once	`return False`

Now we know these resolve to expressions like $n^2 / 2 + n / 2$

$an^2 + bn + c$, for some a, b, c

As n gets arbitrarily big...

What happens to $an^2 + bn + c$?

- Eventually, bn dominates c, even if c was bigger than b to begin with.
- Eventually, an^2 dominates bn, even if b was bigger than a to begin with.

So we really only care about the an^2 part!

(Do we even care about the a? Wouldn't it be nice to only have to think in terms of n itself?)

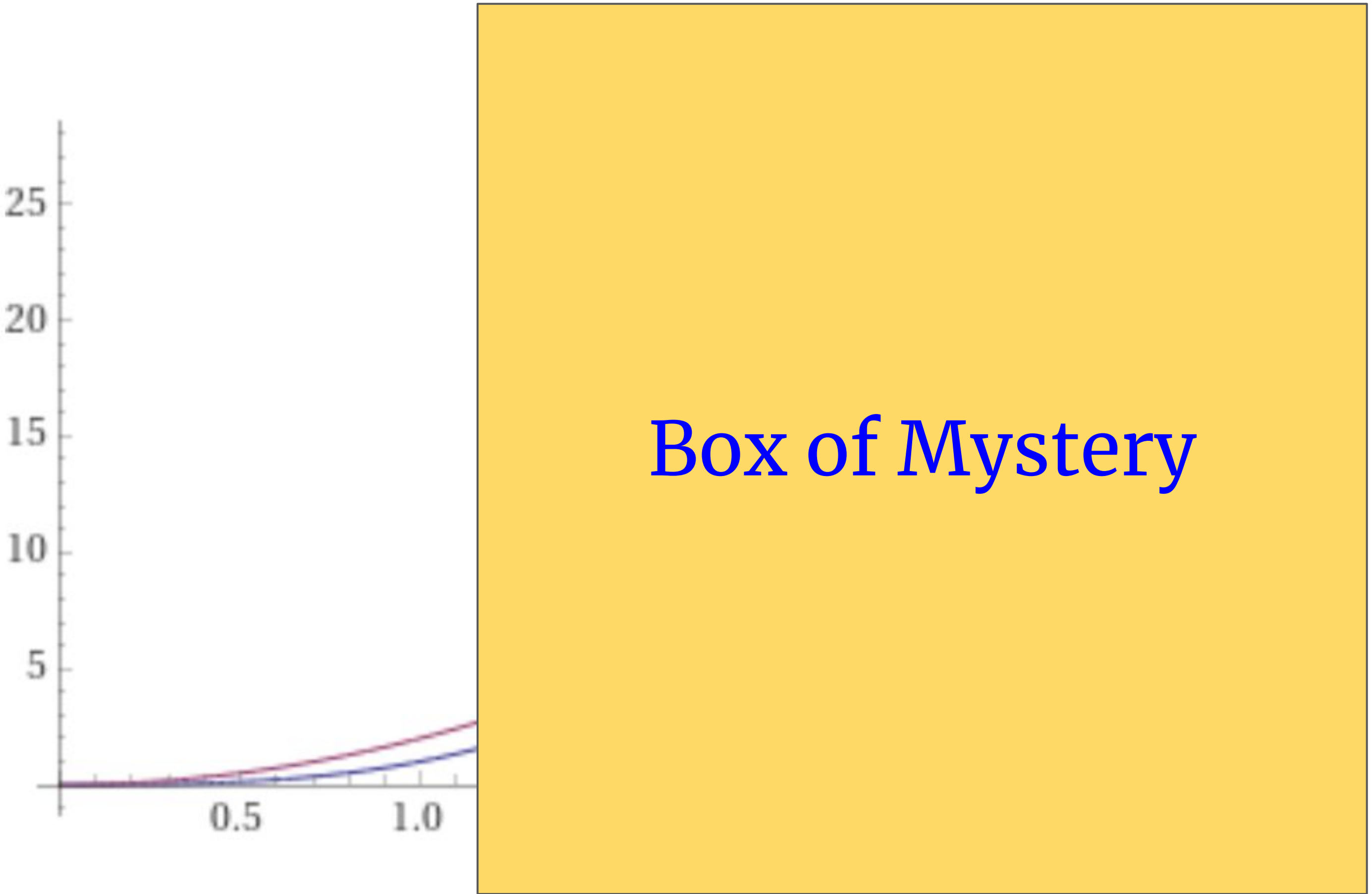
Big-O notation, informally

- Consider two functions *f(n)* and *g(n)*, each of which is defined (at least) on integer values.

- We say that *f(n)* is *O*(*g(n)*) ("big O of *g(n)*") if, as *n* gets bigger...
- ... eventually, **past a point**, *f(n)* is always bounded above **by some constant multiple** of *g(n)*.

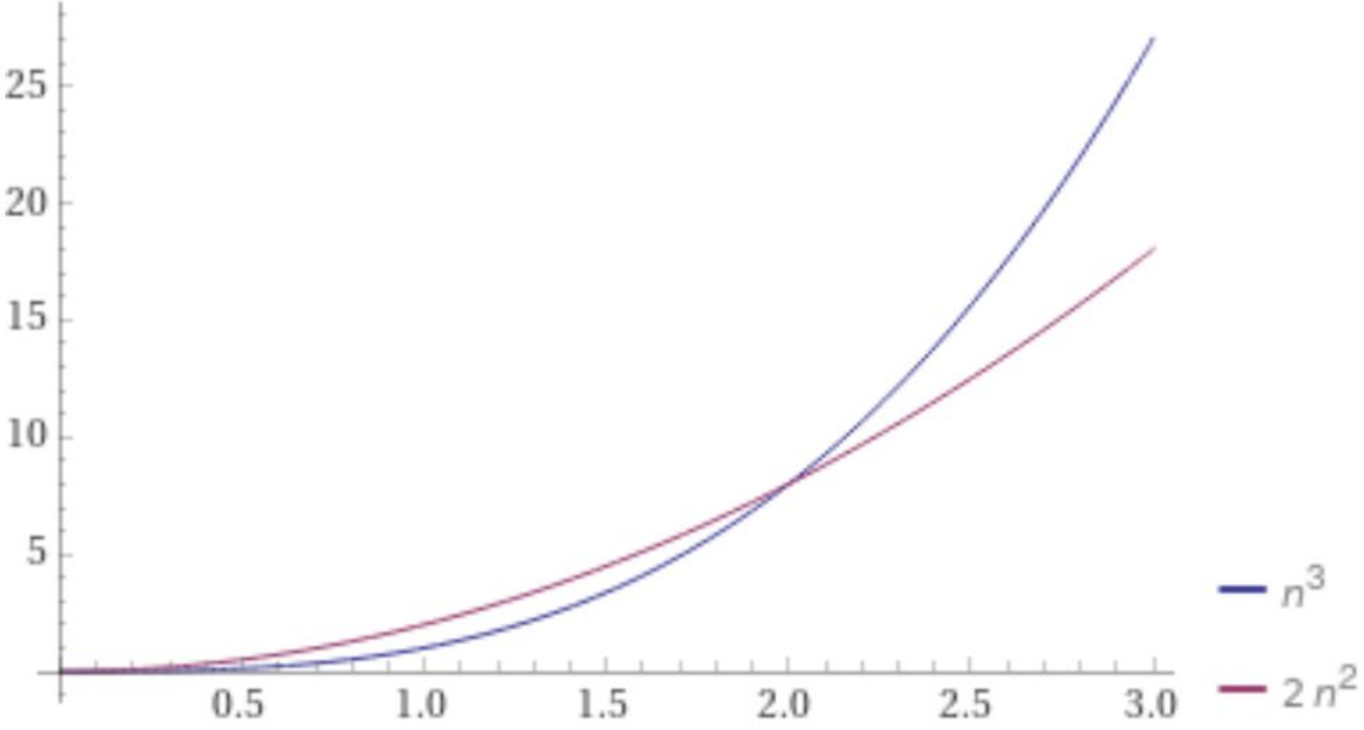
- "*f(n)* is no worse than *g(n)*", in an asymptotic sense.

Why do we need the "past a point" part?



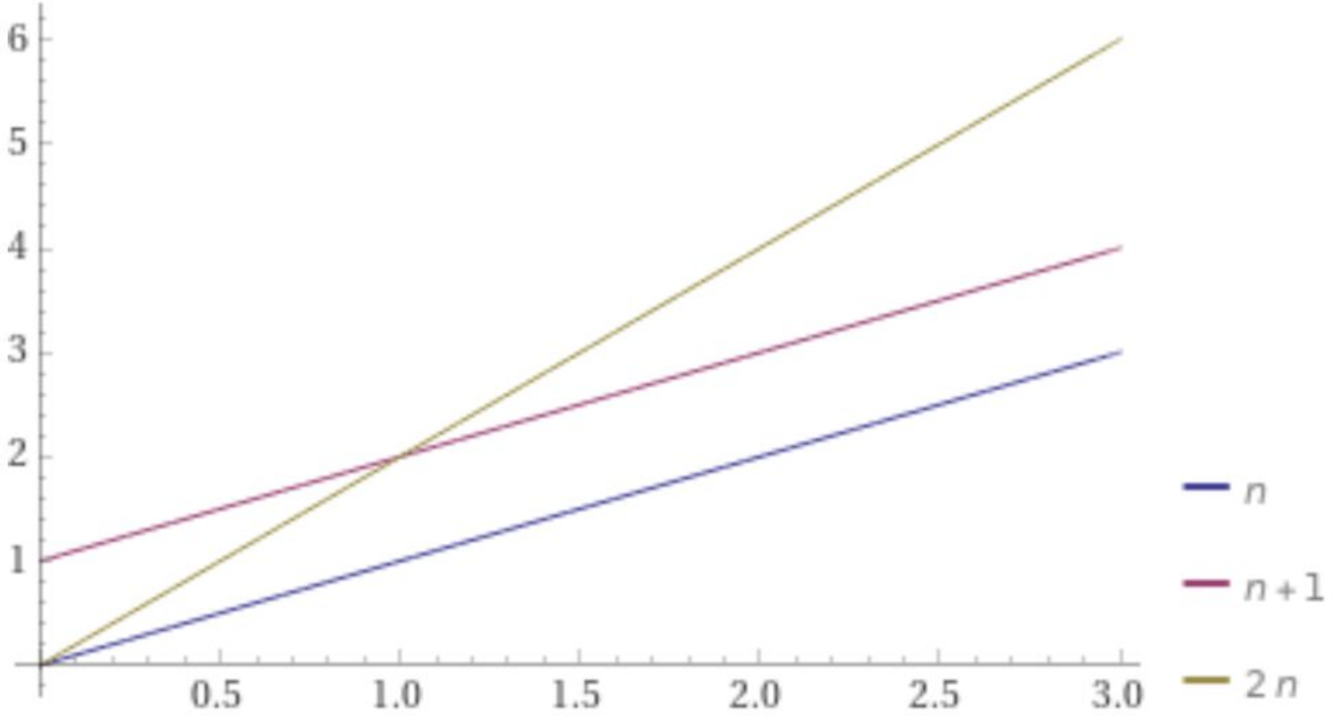
Looks like the blue curve is "smaller" than the magenta curve...

Why do we need the "past a point" part?



Oh no! The blue curve is actually "bigger" past a point.

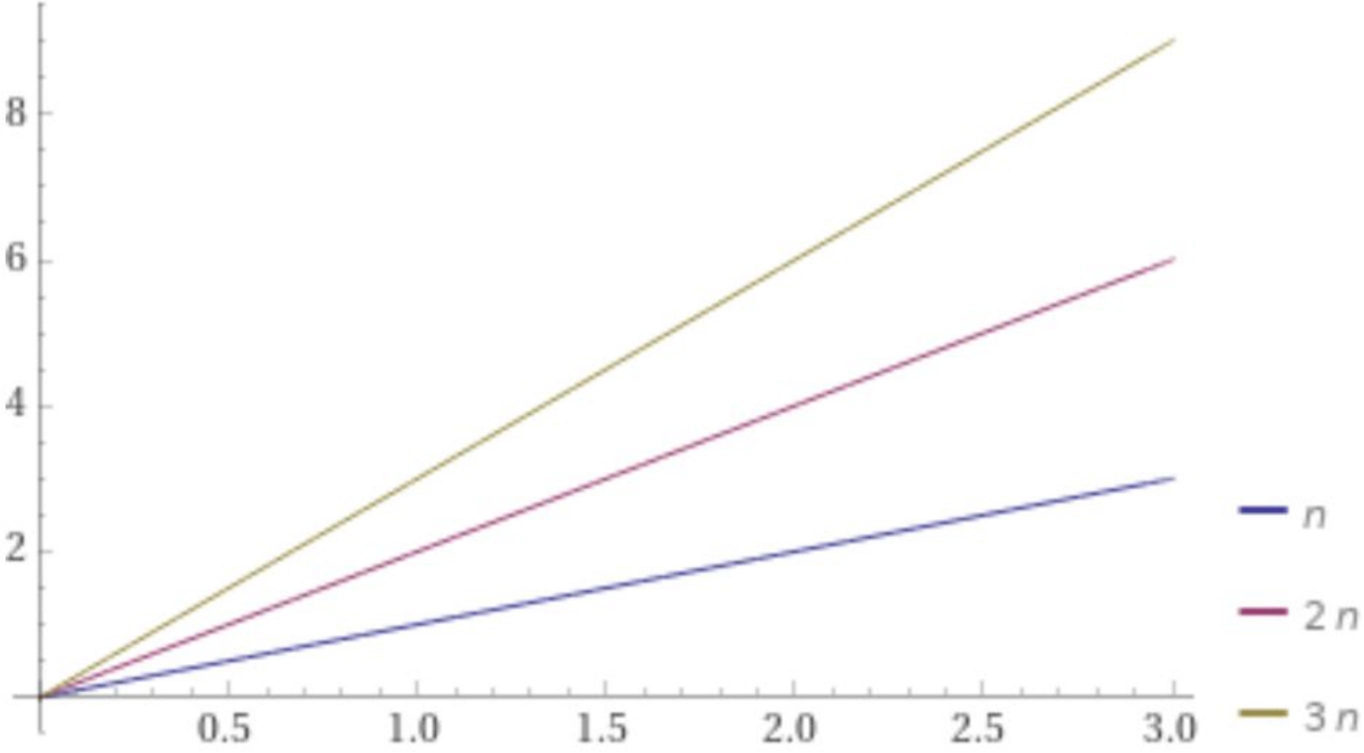
Why do we need the "constant multiple" part?



The whole idea is that we don't want to treat n and n+1 as different...

(2 times n dominates n+1)

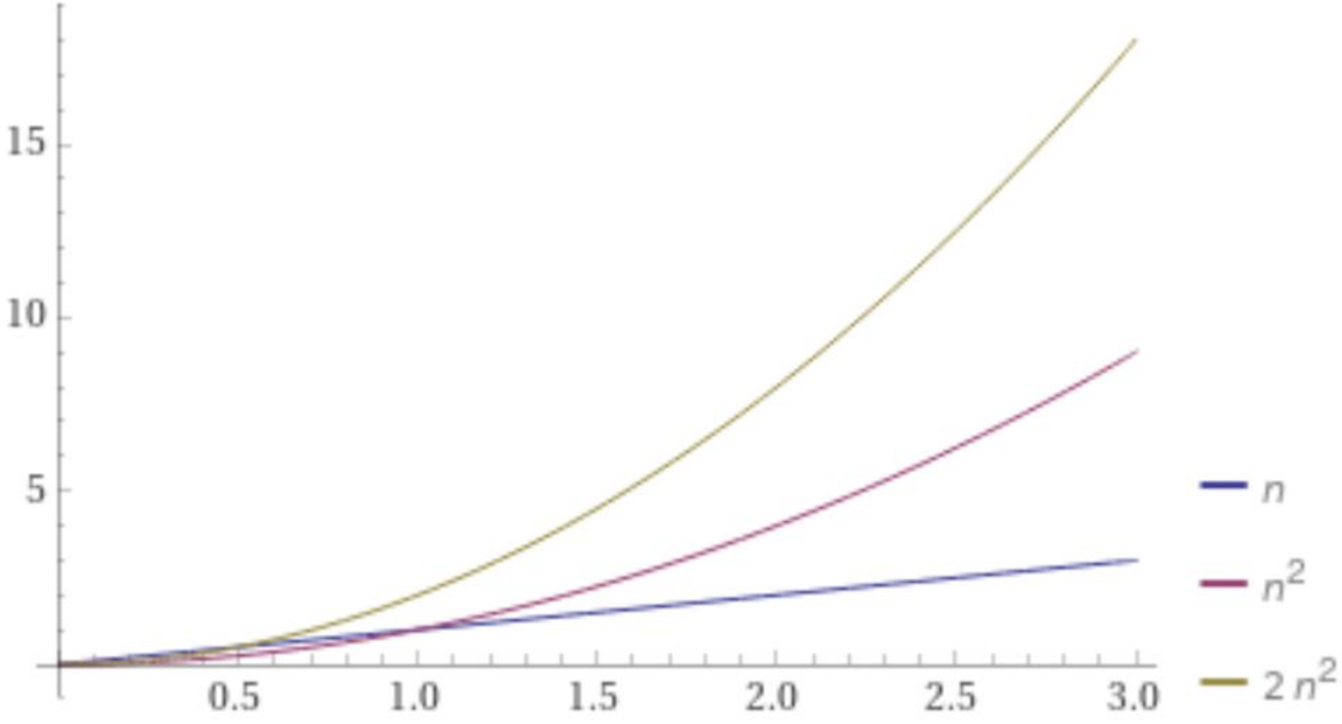
Why do we need the "constant multiple" part?



...or even treat n and 2n as different.

(3 times n dominates 2n)

Why a **constant** multiple?



It'd be silly to say that 2n times n dominates n^2, since we want n and n^2 to be meaningfully different.

Big-O notation, formally

- Consider two functions $f(n)$ and $g(n)$, each of which is defined (at least) on integer values.

- We say that $f(n)$ is $O(g(n))$ ("big O of $g(n)$") if and only if:
  - there exists some positive **constant** c
  - and there exists some integer n_0
  - such that for all integers $n \geq n_0$, $f(n) \leq c * g(n)$

- Equivalent notation: $f(n) = O(g(n))$, $f(n) \in O(g(n))$

Positive example: Show that $n^2 + 1$ is $O(n^3)$

- We need to find some positive **constant** *c*
  - and some integer n_0
  - such that for all integers $n \geq n_0$, $n^2 + 1 \leq c * n^3$

- *How do we do this? A good first step is to just play around with some values and get a feel for the functions…*

- We need to find some positive **constant** c
  - and some integer n_0
  - such that for all integers $n \geq n_0$, $n^2 + 1 \leq c * n^3$

- Let's examine the behavior of $n^2 + 1$ and n^3:
  - $(1)^2 + 1 = 2$, which is bigger than $1^3 = 1$.
  - $(2)^2 + 1 = 5$, which is smaller than $2^3 = 8$.
  - $(3)^2 + 1 = 10$, which is smaller than $3^3 = 27$.
    - and it's only going to go on like that...

- So it looks like we can choose $c = 1$, $n_0 = 2$...

Note: There is no requirement that you choose an "optimal" or "elegant" c and n_0. Any set that works is fine.

- How do we argue that $n^2 + 1$ really is always smaller than $1 * n^3$ for $n \geq 2$?

- One way:
  - Let $n \geq 2$. Consider the quantity $n^3 / (n^2 + 1)$.
  - *That + 1 in the denominator is annoying. Can we get rid of it?*
  - *We can do what CS theory does best – use a ridiculously loose bound.*

- How do we argue that $n^2 + 1$ really is always smaller than $1 * n^3$ for $n \geq 2$?

- One way:
  - Let $n \geq 2$. Consider the quantity $n^3 / (n^2 + 1)$.
  - Notice that $2n^2$ is always bigger than $n^2 + 1$, since $n \geq 2$.
  - Therefore replacing $n^3 / (n^2 + 1)$ with $n^3 / 2n^2$ can only make that quantity **smaller**.
  - Now we're almost there! We just need to make an argument about $n^3 / 2n^2$.

- How do we argue that $n^2 + 1$ really is always smaller than 1 * n^3 for $n \geq 2$?

- One way:
  - Let $n \geq 2$. Consider the quantity $n^3 / (n^2 + 1)$.
  - Notice that $2n^2$ is always bigger than $n^2 + 1$, since $n \geq 2$.
  - Therefore replacing $n^3 / (n^2 + 1)$ with $n^3 / 2n^2$ can only make that quantity **smaller**.
  - But what is $n^3 / 2n^2$? It's just $n/2$. And for $n \geq 2$, this is always at least 1.
  - Therefore $n^3 / (n^2 + 1)$ is also at least 1... i.e., n^3 is bigger than $n^2 + 1$ for $n \geq 2$.
    - which is what we wanted to show!

Some examples

- $1 = O(\log n)$ — *Logs grow faster than constants (ofc).*
- $\log n = O(n)$ — *Polynomials grow faster than logs.*
- $n = O(n \log n)$
- $n \log n = O(n^2)$
- $n^2 = O(2^n)$ — *Exponentials grow faster than polynomials.*
- $2^n = O(n!)$ — *Factorials grow faster than exponentials.*

This is far from the only set of "levels", though! For instance, what about $\log^2 n$? Or $n^{1.5}$?

How do $\log n$ and the square root of n compare?

A word on logarithms

- The logarithms on the last slide had no bases given!

- This is because the base *does not matter* asymptotically. Here's an example of the argument:

- Suppose that some $f(n)$ is $O(\log_2 n)$.
- Then there exist c, n_0 such that for all integers $n \geq n_0$, $f(n) \leq c * \log_2 n$.

Note that we don't know (or need to know) what these values c and n_0 actually are... just that they exist!

A word on logarithms

- The logarithms on the last slide had no bases given!

- This is because the base *does not matter* asymptotically. Here's an example of the argument:

- Suppose that some $f(n)$ is $O(\log_2 n)$.
- Then there exist c, n_0 such that for all integers $n \geq n_0$, $f(n) \leq c * \log_2 n$.
- But $\log_2 n = (\log_3 n) / (\log_3 2)$. (See the log review doc)

A word on logarithms

- The logarithms on the last slide had no bases given!

- This is because the base *does not matter* asymptotically. Here's an example of the argument:

- Suppose that some $f(n)$ is $O(\log_2 n)$.
- Then there exist c, n_0 such that for all integers $n \geq n_0$, $f(n) \leq c * \log_2 n$.
- But $\log_2 n = (\log_3 n) / (\log_3 2)$.
- Now take $c' = c / (\log_3 2)$.
- Then for all integers $n \geq n_0$, $f(n) \leq c' * \log_3 n$.
- **So $f(n)$ is also $O(\log_3 n)$.**

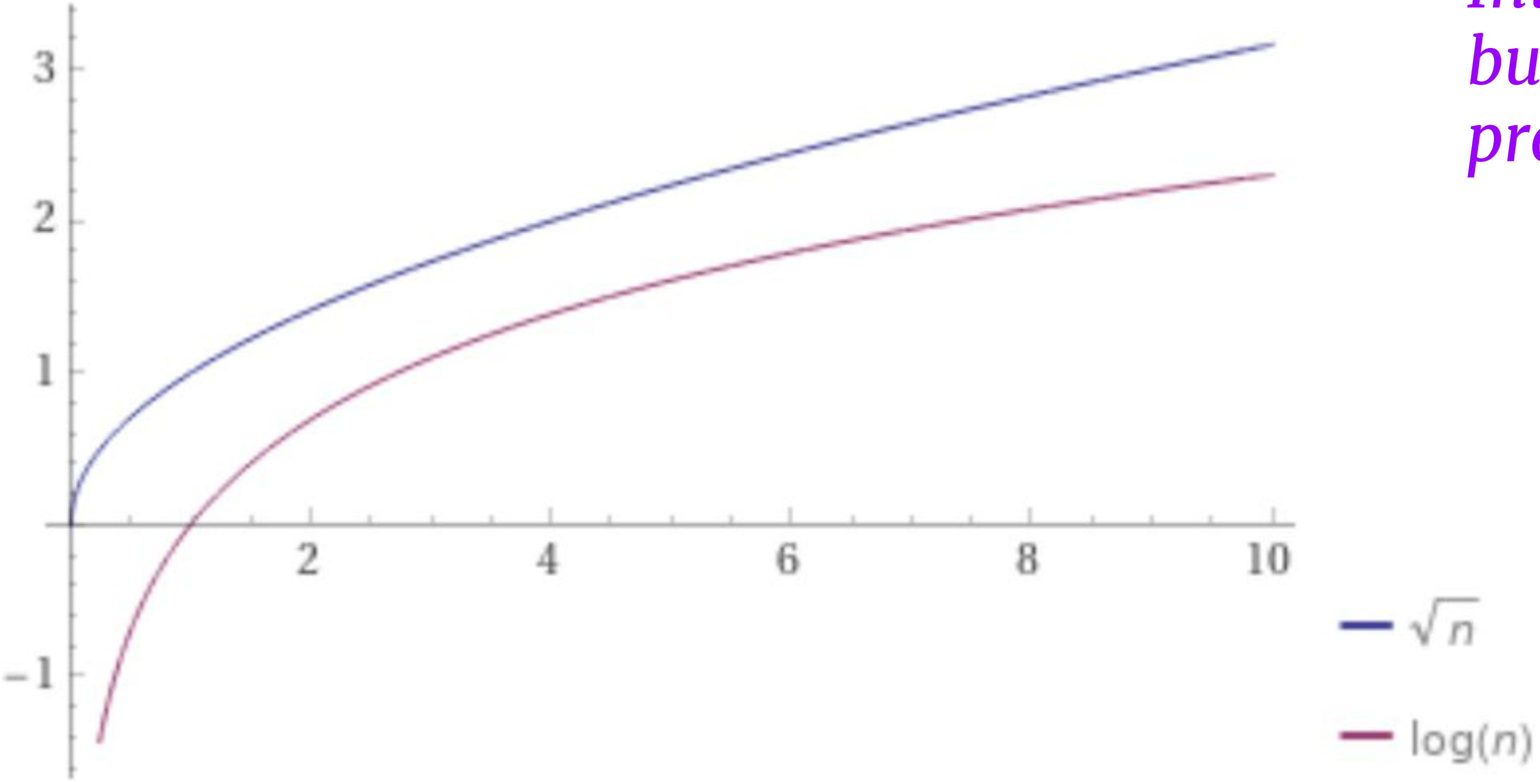
We used the big-O definition against itself!

OK but

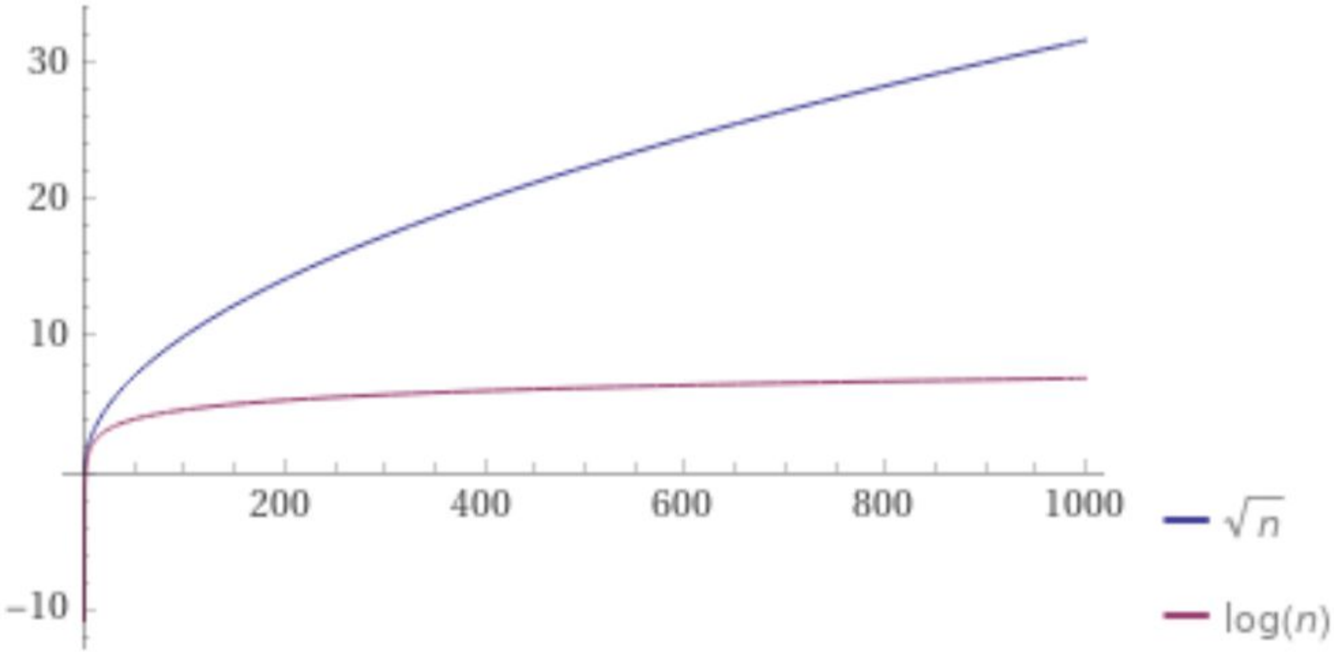
How do we prove that something is NOT Big-O of something else?

Negative example: Show that $n^{1/2}$ is not O(log n)

Intriguing, but not a proof!

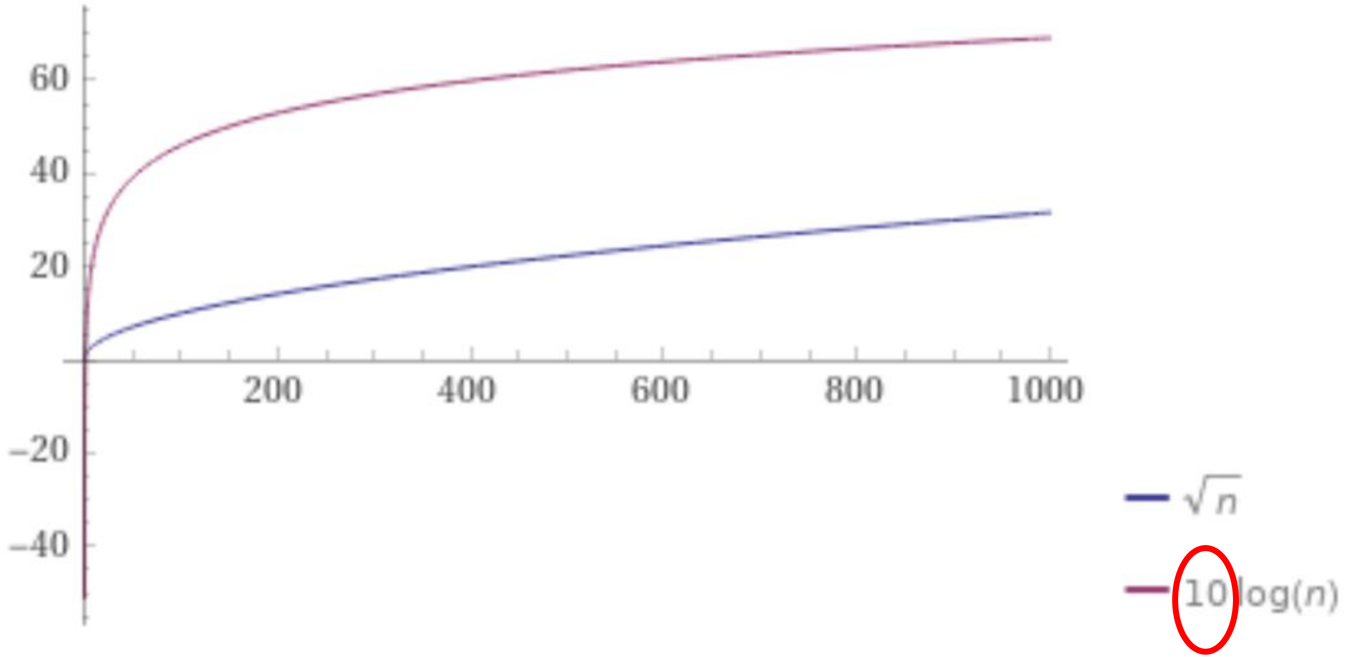


Showing that $n^{1/2}$ is not $O(\log n)$



Looks less ambiguous, but still not a proof!

Showing that $n^{1/2}$ is not O(log n)



Oh no! With the constant multiple in there, it's ambiguous again!

When in doubt, math it out...
...with a proof by contradiction!

Suppose (heading for a contradiction) that $n^{1/2} = O(\log_2 n)$.

(Now what? All we have to work with is the definition of big-O, so let's try using that...)

We're using a specific base here to make the argument more tractable. But the following idea would extend to any base.

Suppose (heading for a contradiction) that $n^{1/2} = O(\log_2 n)$.

Then there exist some constant c and some integer n_0 such that for all integers $n \geq n_0$, $n^{1/2} \leq c * \log_2 n$.

Suppose (heading for a contradiction) that $n^{1/2} = O(\log_2 n)$.

Then there exist some constant c and some integer n_0 such that for all integers $n \geq n_0$, $n^{1/2} \leq c * \log_2 n$.

Now what? How do we break this?

How about with a really big n?

The critical observation is that now the argument we're trying to break is stuck using a constant c, but we have the ability to make n as big as we want.

Suppose (heading for a contradiction) that $n^{1/2} = O(\log_2 n)$.

Then there exist some constant c and some integer n_0 such that for all integers $n \geq n_0$, $n^{1/2} \leq c * \log_2 n$.

Now take $n = 2^{2k}$, for any $k > 1$ chosen such that 2^{2k} is $\geq n_0$.
*We can do this because n_0 is a constant and we can just pump k as large as it needs to be. Notice that our argument **has to** ensure that $n \geq n_0$, because otherwise we are evaluating something outside of the scope of the original claim.*

Suppose (heading for a contradiction) that $n^{0.5} = O(\log_2 n)$.

Then there exist some constant c and some integer n_0 such that for all integers $n \geq n_0$, $n^{1/2} \leq c * \log_2 n$.

Now take $n = 2^{2k}$, for any $k > 1$ chosen such that 2^{2k} is $\geq n_0$. Then $(2^{2k})^{1/2} \leq c * \log_2 (2^{2k})$, i.e., $2^k \leq 2ck$.

Now what? We want to show that whatever c was chosen can't possibly be big enough. It sure looks like that, since 2^k *grows faster than 2k, but it can be a little tricky to pin down formally.*

Suppose (heading for a contradiction) that $n^{0.5} = O(\log_2 n)$.

Then there exist some constant c and some integer n_0 such that for all integers $n \geq n_0$, $n^{1/2} \leq c * \log_2 n$.

Now take $n = 2^{2k}$, for any $k > 1$ chosen such that 2^{2k} is $\geq n_0$.
Then $(2^{2k})^{1/2} \leq c * \log_2 (2^{2k})$, i.e., $2^k \leq 2ck$.

But now observe that if we increase k by 1, we multiply the left side by a factor of 2 and the right side by a factor of $(k+1)/k$, which is less than 2 since $k > 1$. Therefore, if we make k large enough, the left side becomes bigger than the right, **regardless of what *c* is**, and we have our contradiction.

- Therefore $n^{0.5}$ is not $O(\log_2 n)$.

Big-O was "no worse"; Big-Omega is "no better"

- We say that *f(n)* is Ω(*g(n)*) ("big Omega of *g(n)*") if and only if:
  - there exists some positive **constant** *c*
  - and there exists some integer n_0
  - such that for all integers $n \geq n_0$,
  - $f(n) \geq c * g(n)$

this ≥ is the only difference from Big-O!

- e.g., n^3 is $\Omega(n^2)$.
- In the context of algorithm analysis, we usually care more about how bounding how *bad* something can get, but sometimes it's also useful to know the *best* we can hope for.

Theta is "asymptotically the same"

- Consider two functions $f(n)$ and $g(n)$, each of which is defined (at least) on integer values.

- We say that $f(n)$ is $\Theta(g(n))$ ("Theta of $g(n)$") if and only if:
  - $f(n)$ is $O(g(n))$, **and**
  - $f(n)$ is $\Omega(g(n))$

why not "Big Theta"? We'll see in a bit

- e.g., $n^3 + 1$ is $\Theta(n^3)$.

- Another note: We never say $O(2n^3+n)$ or $\Theta(n^3+1)$, for instance... why not? (The whole point is to ditch the constants and lower-order terms)

A warning about "Big-O" in the "real world"

- In my experience, everywhere outside CS161 (even in CS theory classes), people often use big-O as if it were Theta.
- For example, it is not technically wrong to say that something that is $O(n^2)$ is also $O(n^3)$. But when people say "$O(n^3)$", what they usually *mean* is that they think (or know) that the algorithm is $\Theta(n^3)$. That is, they use big-O in a tight way.
- We will misuse this notation even in CS161.

Some other notation we won't use as much

- Little o is like Big-O, but with the strict sense of "better" rather than "no worse".
  - e.g., n^3 is $O(n^3)$, but **not** $o(n^3)$.
  - $n^{2.99999}$ is $o(n^3)$.

- Little ω is like Big-Ω, but with the strict sense of "worse" rather than "no better".
  - e.g., n^3 is $\Omega(n^3)$, but **not** $\omega(n^3)$.
  - $n^{3.00001}$ is $\omega(n^2)$.

- These have formal definitions but you aren't responsible for them. I may use them occasionally, so it's good to understand what they mean.

Asymptotics matter!

Me in 2013: A tragedy in one act

"This kind of record shows up pretty rarely. It's probably fine to just compare every pair of these. Sure, it's $O(n^2)$, but the code is simpler and more maintainable!"

But at a company that processes billions of records a day, one-in-a-billion things happen several times per day...

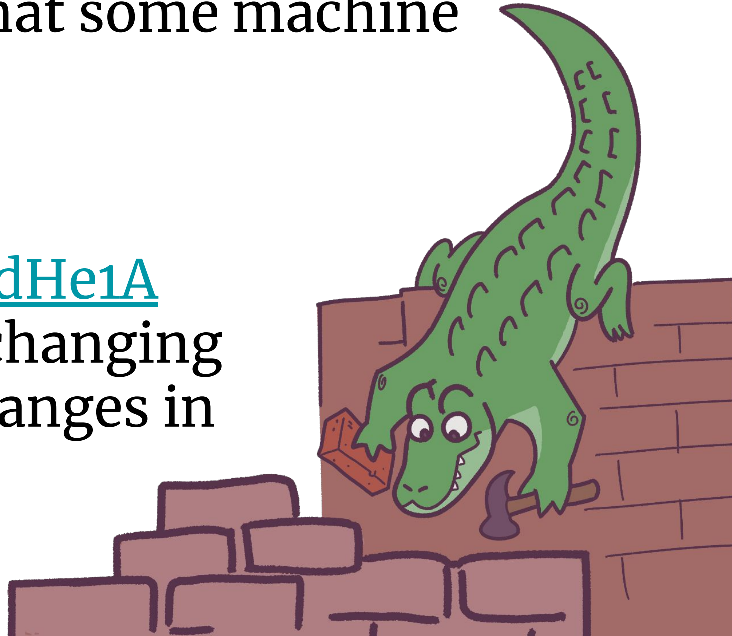
What if there's that one pathological example where suddenly, your $O(n^2)$ algorithm gets $n = 1000000$?

What about real-world systems details?

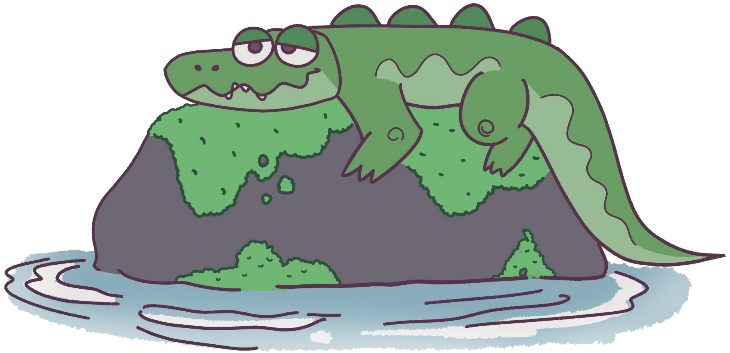
- Wait, did we just crawl up our own theory asses, so to speak?

- Constant factors do matter in the real world!
- We never actually dealt with the fact that some machine instructions cost much more!
- What about L1, L2, ... caching?

An alarming talk: https://youtu.be/r-TLSBdHe1A (essentially, runtime improvements from changing an algorithm may actually just be due to changes in memory layout)



Waverly says: Relax, it'll be fine



- The point of big-O is to simplify a complex problem so we can talk at a high level and get things done…
- …but never forget that it's still a complex problem!

See you on Wednesday!