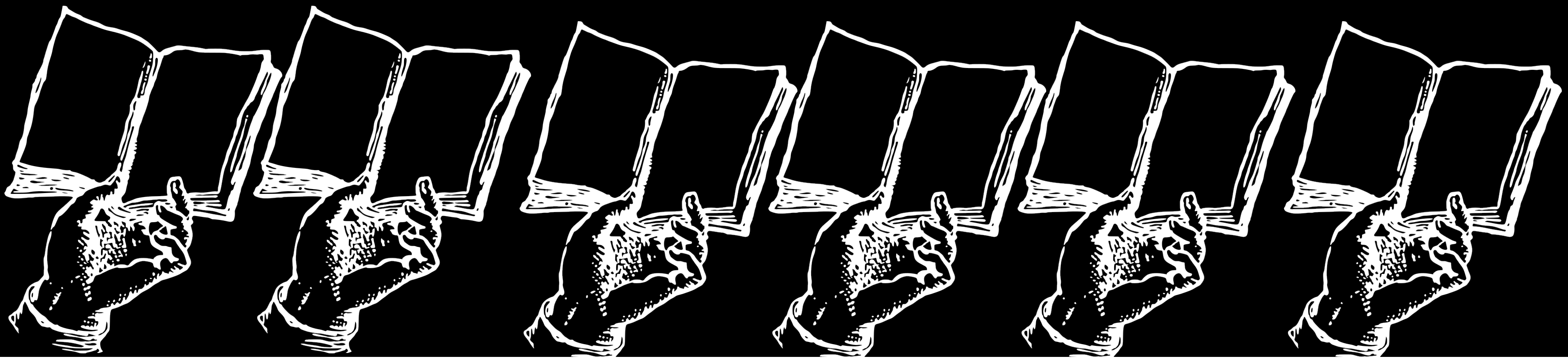
Array Expansion

About Arrays

Random Access

- Arrays are a proxy for RAM
  - RAM: “Random Access Memory” in the computer
  - Any (random) location be read/written in uniform time
- Metaphor: Well-staffed Library (staffed by pitchers!)



About Arrays

Array Limits & Management

- Size is “fixed” when they’re created
  - This is due to how memory is managed…

About Arrays

Array Limits & Management

- Overcoming Fixed Size: Managed Arrays
  - Common approach
    - Use a class (Ex: `ArrayList`)
      - Methods manage an array
        - Allow array-like operations
          `get() => x=A[i]` ; `set() =>A[i]=x` ; `append()/add()` ; `size()`)
    - Separate notion of “capacity” (`.length`) and “currently holding” (`n`)

About Arrays

Resizing Strategies

- “Perfect size”: Add exactly 1 space each time an item is added (capacity == n always)
- “Doubling”: When space is needed double the capacity

Java Debugger

Tips

- Running in the debugger
- Setting breakpoints
- Controlling execution: Resume, Step into, Step over, & Terminate
- The variables tab: inspecting values
- The breakpoints tab: adding counts and conditions
- Returning to the Java perspective

Credits!

The following are based on work of Prof. Buehler, Cole, and Cytron

Studio 0 Summary

Empirical Estimates of Performance

- “Ticks” are a useful way to measure operations *empirically*
- Ticks represent the “constant time operations”
    - If placed correctly, they are proportional to time

Studio 0 Summary

Limits of Empirical Approaches

- The empirical approach requires
  - Creating code,
  - Setting up experiments,
    - How do you choose the data/experiment?
  - Running experiments,
  - Analyzing results, and
  - Possibly repeating all that if there are errors in the process

The Analytical Approach

The Analytical Approach

vs. Empirical

- No code needed!
    - Can be used to decide which version of code is worth creating!
- Easier to focus on worst case
    - Can estimate operations needed in the worst case without knowing the precise worst case

Review of Ticks

How many times do we call `tick()`?

```
@Override
public void run() {
  for (int i=0; i < n; ++i) {
    //
    // Statement below is deemed to take one operation
    //
    this.value = this.value + i;
    ticker.tick();
  }
}
```

Ticks

Accounting

Loop counter stop value

$$\text{Total tick count} = \sum_{i=0}^{n-1} 1$$

Loop counter start value

Ticks

Accounting & Algebra

- One tick per iteration
- Total

$$= \sum_{i=0}^{n-1} 1$$

$= (n-1) - 0 + 1$

$= n$

Accounting

Rule 1: Counting Loop Iterations

A loop from `i`=LO to `i`=HI (inclusive) runs:

HI-LO+1 times

Examples:

Loop from -2 to 5:

8 total iterations!
(-2, -1, 0, 1, 2, 3, 4, 5)

Nested Loops

How many times do we call `tick()`?

```
@Override
public void run() {
  for (int i=0; i < n; ++i) {
    for (int j=0; j<i; ++j) {
      // Statement below is deemed to take one operation
      this.value = this.value + i;
      ticker.tick();
    }
  }
}
```

LO = 0

HI = (i-1)

$$\begin{aligned} \text{total} &= HI - LO + 1 \\ &= (i-1)-0+1 \\ &= i \end{aligned}$$

Nested Loops

How many times do we call `tick()`?

```
@Override
public void run() {
  for (int i=0; i < n; ++i) {
```

i ticks

```
  }
}
```

Nested Loops

How many times do we call `tick()`?

The “i ticks” part will run:

$$= \sum_{i=0}^{n-1} \quad \text{times}$$

Each time it will do “i” ticks

$$\text{Total ticks} = \sum_{i=0}^{n-1} i$$

```
@Override
public void run() {
  for (int i=0; i < n; ++i) {

                    i ticks

  }
}
```

Accounting

Rule 2: Counting ***Nested*** Loop Iterations

Work inside-out and form a summation!

More Abstract: Pseudocode

- Concepts covered here are not specific to programming language
- Pseudocode
  - Code-like (loops, logic)
  - Math expressions
  - Precise to people, but not runnable code

Pseudocode

Practice Problem

```
for j in 1 … n
    tick()
    for k in 0 … j
        tick()
        tick()
        tick()
```

Indicates inclusive

Pseudocode

Practice Problem

```
for j in 1 … n
    tick()
    for k in 0 … j
        tick()
        tick()
        tick()
```

3 ticks

3(j+1) ticks

Pseudocode

Practice Problem

```
for j in 1 … n
    tick()
    for k in 0 … j
        tick()
        tick()
        tick()
```

3 ticks

1…n

`tick()`!

3(j+1) ticks

Pseudocode

Practice Problem

```
for j in 1 … n
    tick()
    for k in 0 … j
        tick()
        tick()
        tick()
```

$1+3(j+1)$

Pseudocode

Practice Problem

```
for j in 1 … n
    tick()
    for k in 0 … j
        tick()
        tick()
        tick()
```

1+3j+3

Pseudocode

Accounting

```
for j in 1 … n
    tick()
    for k in 0 … j
        tick()
        tick()
        tick()
```

3j+4

$$\sum_{j=1}^{n} (3j + 4)$$

Pseudocode

Accounting & Algebra!

$$\sum_{j=1}^{n}(3j+4) \quad = \sum_{j=1}^{n}3j + \sum_{j=1}^{n}4 \quad = 3\sum_{j=1}^{n}j + 4\sum_{j=1}^{n}1$$

$$= 3\frac{n(n+1)}{2} + 4n \quad = \frac{3n^2+11n}{2}$$

Ugh.

Do we care?
Do we need this much detail?

Detail: How much is enough?

How do we ***use*** this information?

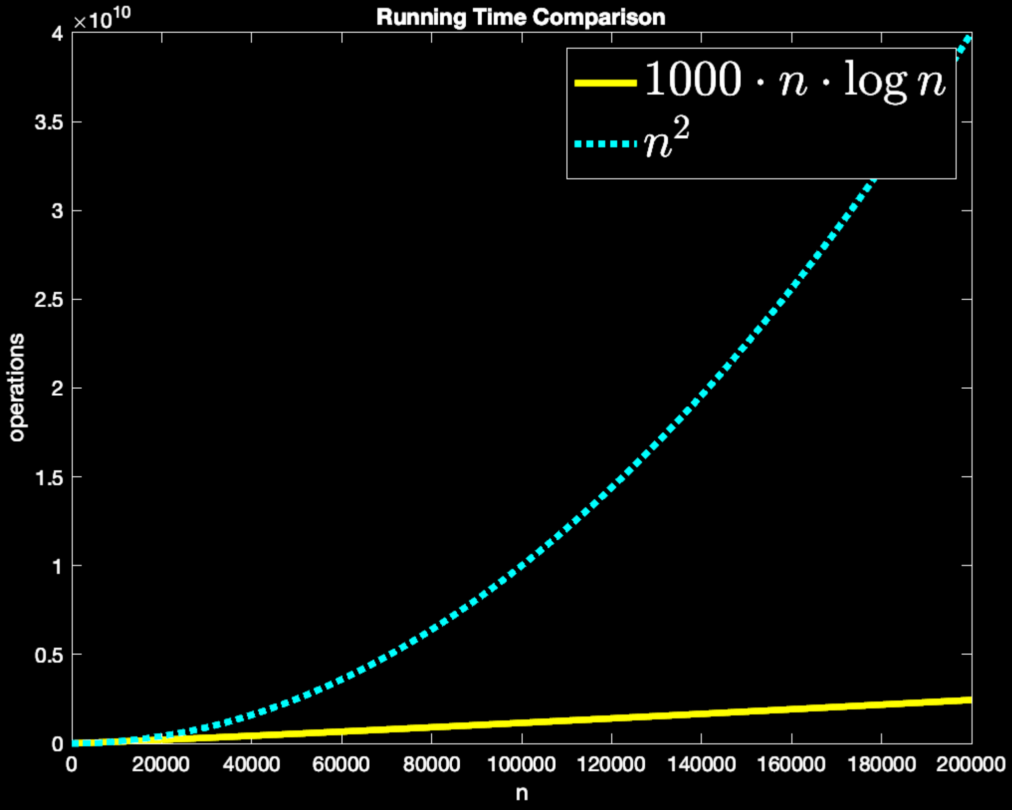
- Prediction: Predict exact time for an algorithm
    - Needs precise details
- Comparing two different algorithms

Ex 1: Alg. A is $1000 \cdot n \cdot \log n$ Alg. B is n^2

Ex 2: Alg. B is n^2 Alg. C is $3 \cdot n^2$

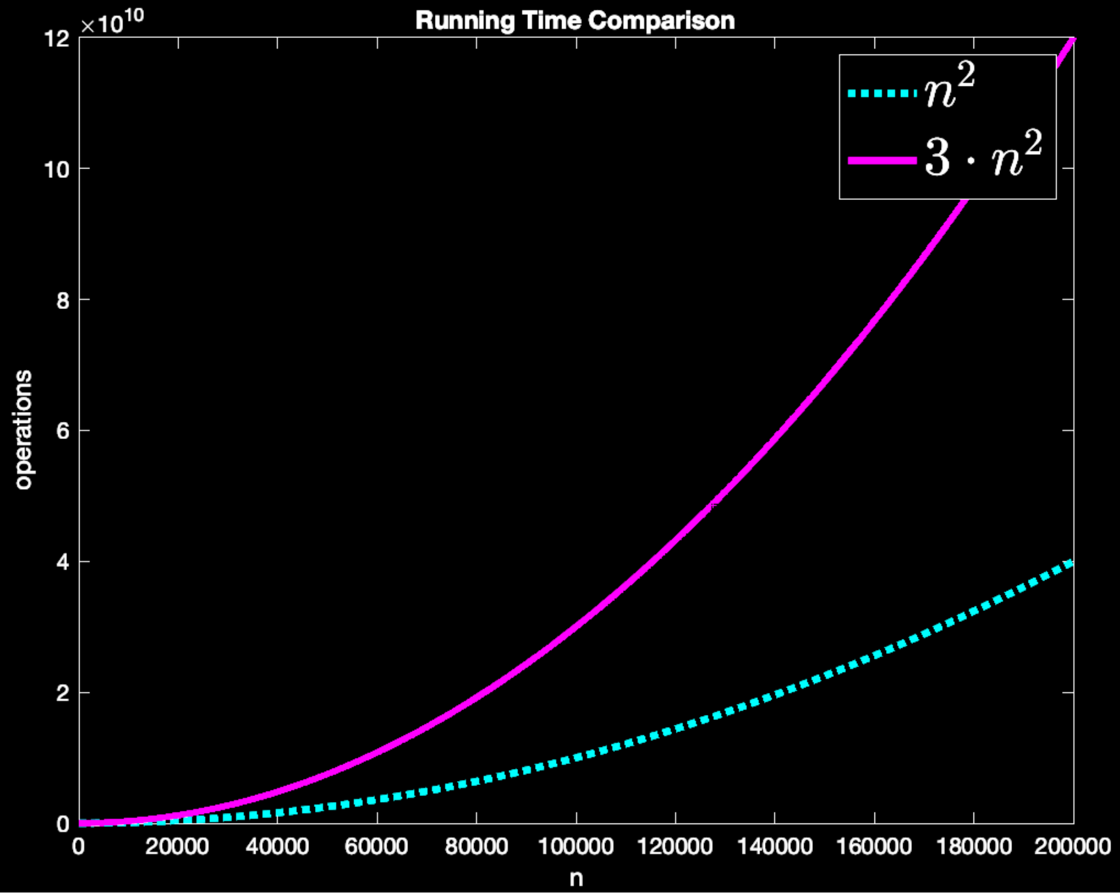
Comparing

A: $1000 \cdot n \cdot \log n$ **B:** n^2



Comparing

B: n^2 **C:** $3 \cdot n^2$



Moore’s “Law”

Computers get better, faster

- Gordon Moore: Co-founder of intel
- Roughly: Improvements double transistors on chip every two years
    - Implications
        - More memory!
        - More complex chips!
        - Typically also more speed!

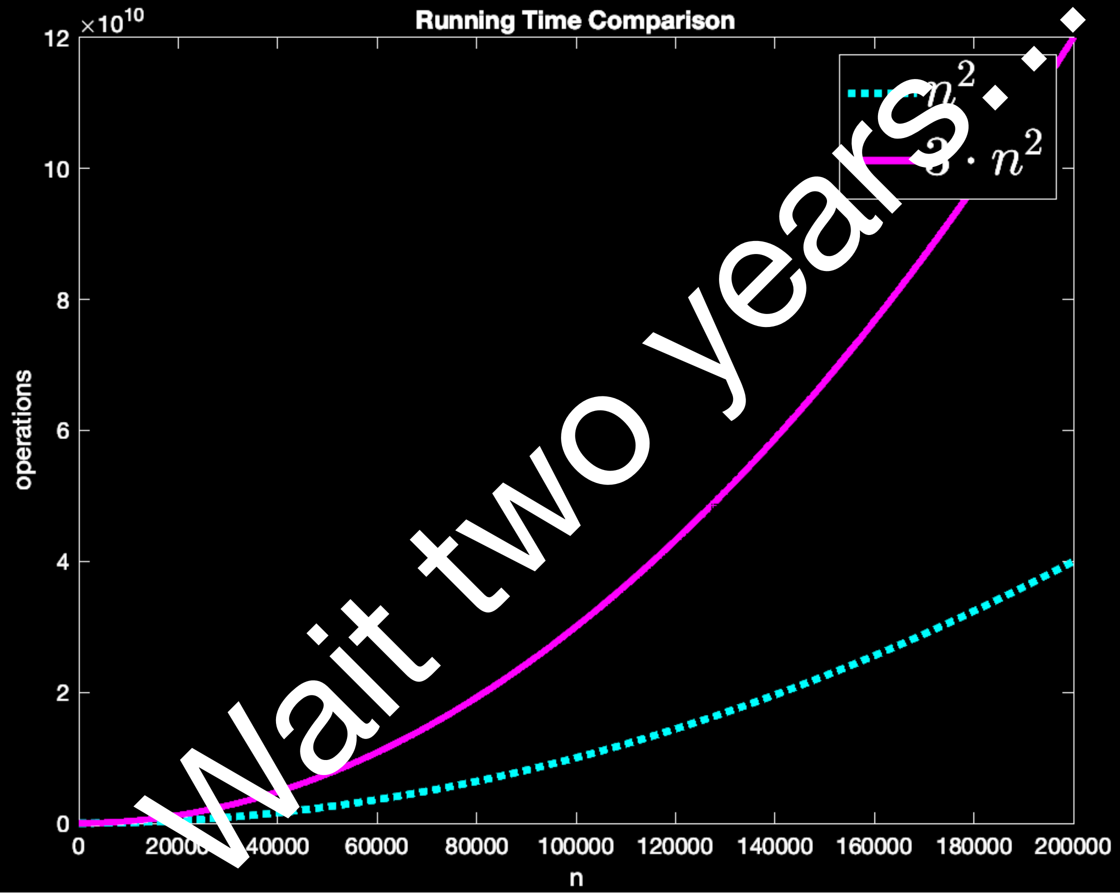
Moore’s “Law”

Computers get better, faster

- Historically: Computation speed doubles every ~2 years!
  - This is slowing down. Past history may not indicate future performance!

Comparing

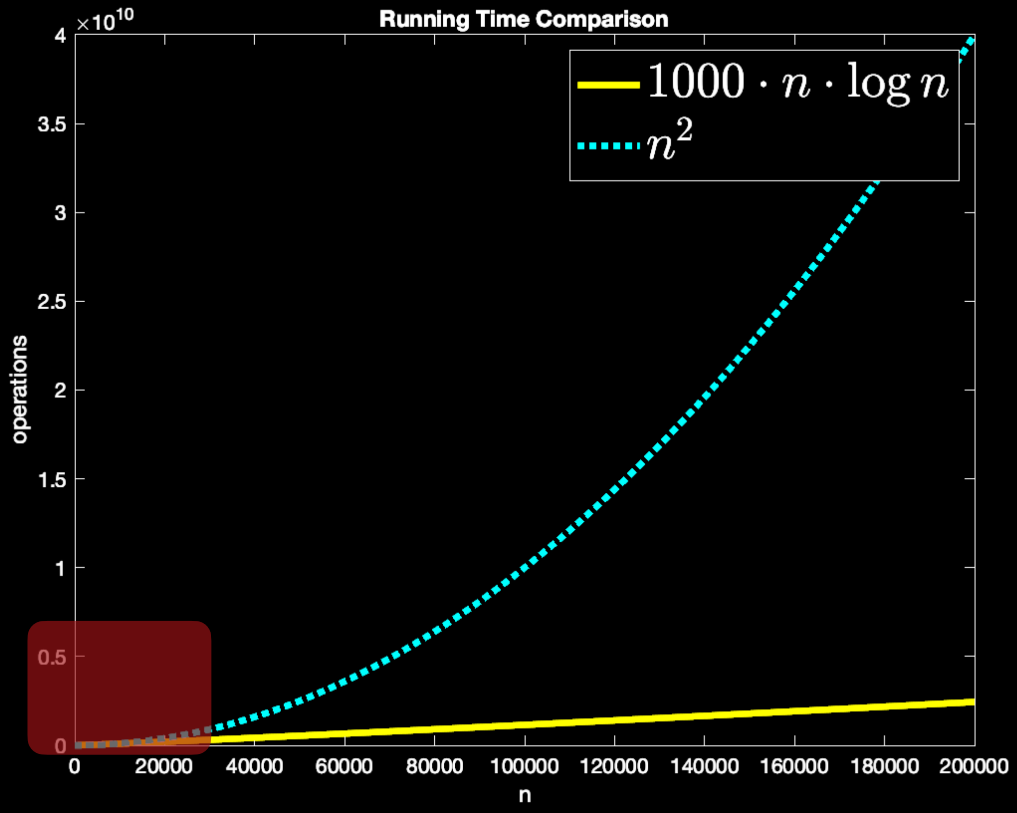
B: n^2 **C:** $3 \cdot n^2$



Wait two years…

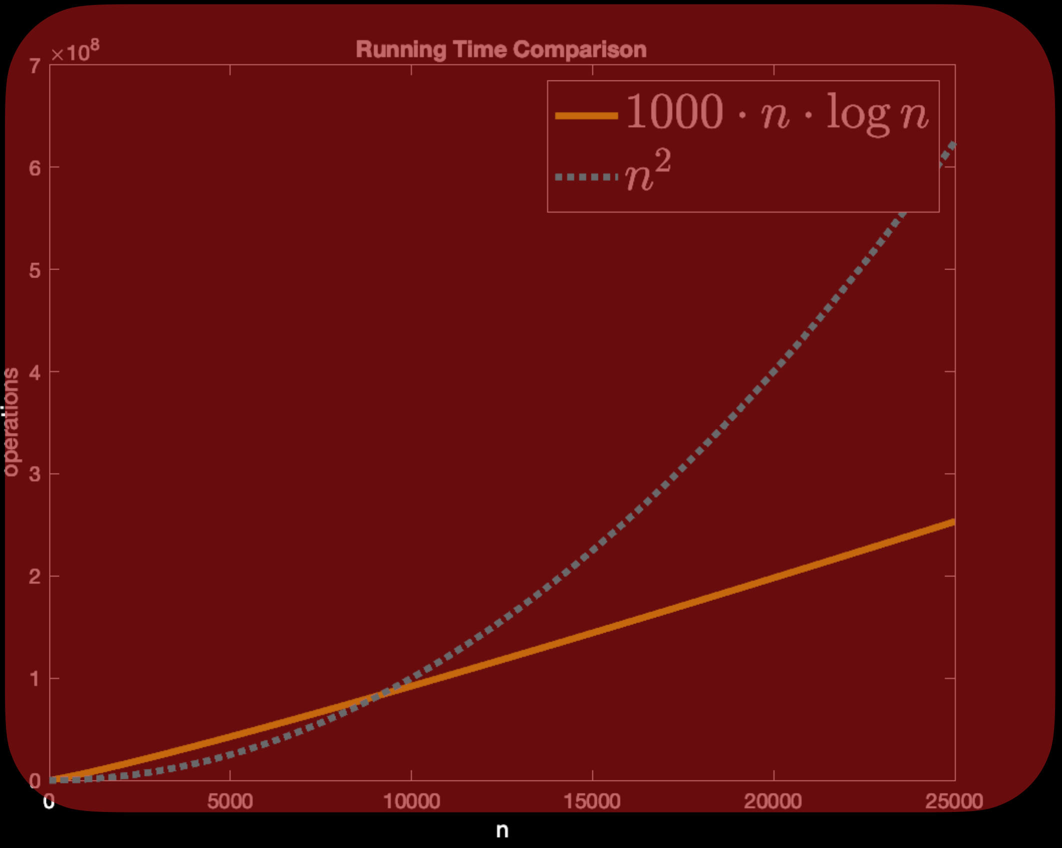
Comparing: Again

A: $1000 \cdot n \cdot \log n$ **B:** n^2



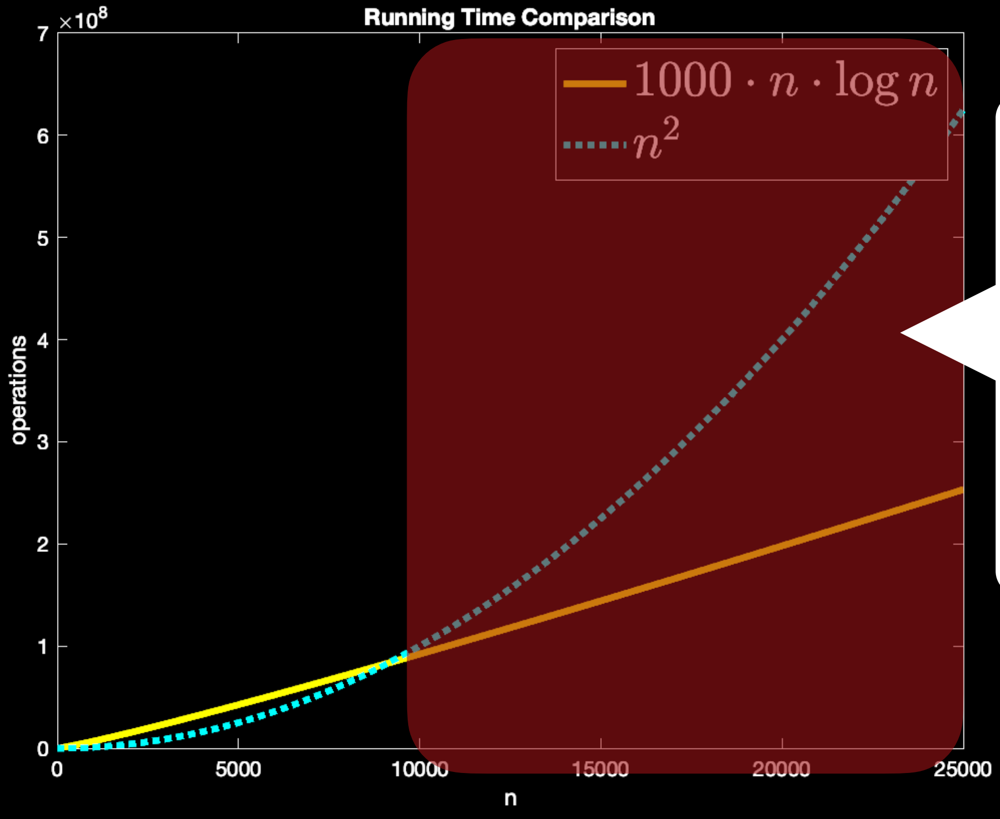
Comparing: Again

A: $1000 \cdot n \cdot \log n$ **B:** n^2



Comparing: Again

A: $1000 \cdot n \cdot \log n$ **B:** n^2



Our Focus:
The impact of
growth & large
problems

Run Time: Thinking Theoretically (The Big-O notation)

Run Time

What are useful goals?

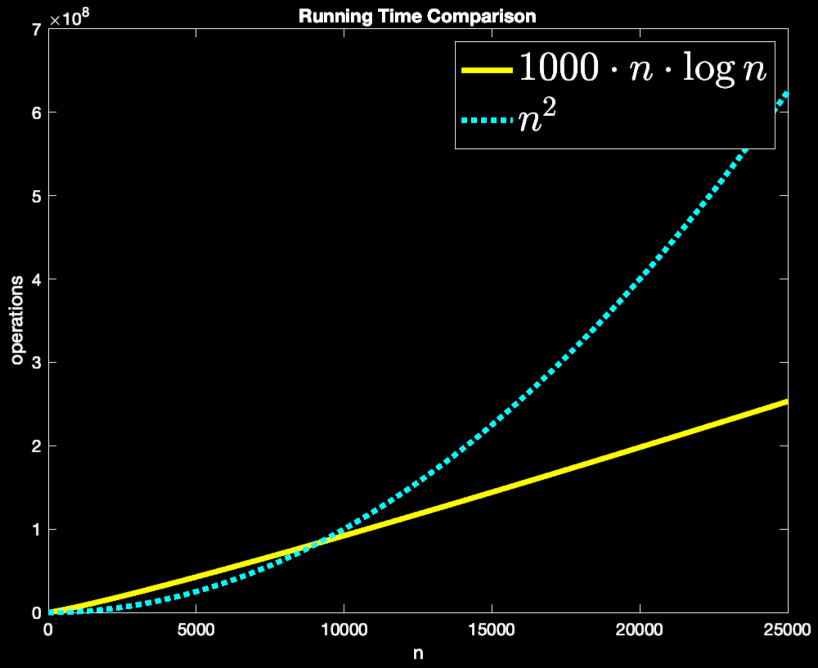
1. Distinguish between
   - Clear, significant differences in choices ($1000 \cdot n \cdot \log n$ vs. n^2)
     - That is, differences in the *order of growth*
   - “Close” cases that may merit looking at more precise details
2. Ignore small, transient cases

Run Time

What are useful goals?

Assumptions

- We'll ignore the transient issues
- We care about "growth"
  - That is asymptotic behavior



- asymptotic: adjective. "2. (of a function) approaching a given value as an expression containing a variable tends to infinity." (dictionary.com)

Definition of Big-O notation

"O" for "Order" (like order of magnitude)

- Let $f(n)$ and $g(n)$ be non-negative functions for $n > 0$
  - For our purposes, they are both measure of time (or memory) used
- We say: $f(n) = O(g(n))$ if there exists constants $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$
  - Clarification: $O(\cdots)$ defines a set of functions that are bounded above!
  - Often $f(n)$ is in $O(g(n))$ (f is in big-O of g)

Definition of Big-O Notation

What?

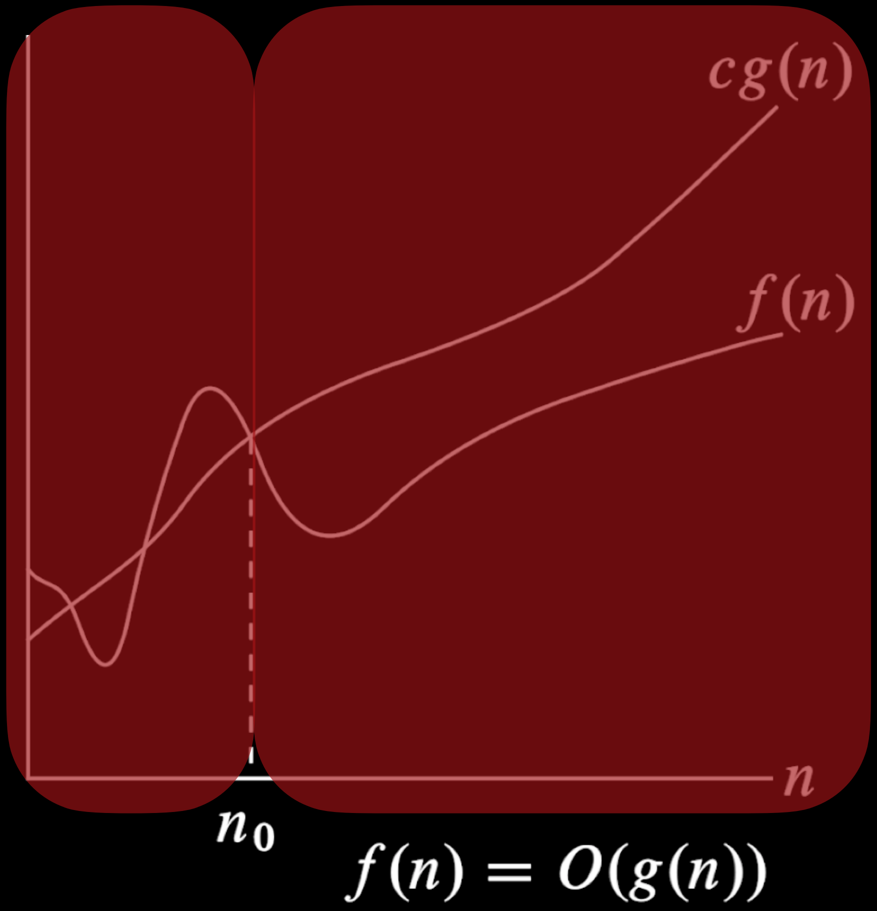
1. Let $f(n)$ and $g(n)$ be non-negative functions for $n > 0$

2. $f(n) = O(g(n))$ if

   For $c > 0$ and $n_0 > 0$

   such that

   for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$



Run Time

Does Big-O meet our goals?

1. Distinguish between
   - Clear, significant differences in choices ($1000 \cdot n \cdot \log n$ vs. n^2)
     - We can see if things are “equal” in their $O()$
   - “Close” cases that may merit looking at more precise details
     - The c constant => Similar orders of growth all in same $O()$
2. Ignore small, transient cases
   - The $n \geq n_0$ part!

Big-O Ignores Constants

As desired

- Lemma:

  If $f(n) =$

- Proof:

  - $f(n) = O($

  - But then f

  - Conclude that: $f(n) = O(a \cdot g(n))$ QED

Never write a constant inside the

$O(\cdots)$

It's unnecessary

Quod Erat Demonstrandum: That which was demonstrated

Does Big-O Match Intuition?

- Q: Which function grows faster, n or n^2
- So does $n = O(n^2)$
  - Set $c = ???$ and $n_0 = ???$
    - Ex: When $n \geq 1$ is $n^2 \geq n$?
      - YES: Multiply both sides by n: $n \cdot n \geq 1 \cdot n = n^2 \geq n$. QED

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

General Strategy

1. Pick $c > 0$ and $n_0 > 0$
   (Consider choices that will make the next steps easier)

2. Write down the desired inequality: $f(n) \leq c \cdot g(n)$

3. Prove that the inequality holds whenever $n \geq n_0$

Example: Does $3n^2 + 11n = O(n^2)$

- Does $3n^2 + 11n = O(n^2)$
  - Guess???

Example: Does $3n^2 + 11n = O(n^2)$

Proof

1. Pick $c > 0$ and $n_0 > 0$

   $c = 33$ and $n_0 = 1$

2. Write down the desired inequality: $f(n) \leq c \cdot g(n)$

   $3n^2 + 11n \leq 33 \cdot n^2$

3. Prove that the inequality holds whenever $n \geq n_0$

   …

Example: Does $3n^2 + 11n = O(n^2)$

Proof (using $c = 33$ and $n_0 = 1$)

3. Prove that the inequality holds whenever $n \geq n_0$

$$3n^2 + 11n \leq 33 \cdot n^2$$

$$= (3n^2 + 11n) - (3n^2 + 11n) \leq 33 \cdot n^2 - (3n^2 + 11n)$$

$$= 0 \leq 33 \cdot n^2 - 3n^2 + 11n$$

$$= 0 \leq 30 \cdot n^2 + 11n$$

When $n \geq n_0$ = $n \geq 1$, then $30 \cdot n^2 + 11n \geq 0$. QED

Generalization of Proof

- Theorem: A

(In simple po xponent)

- Proof: Pic k $n_0 = 1$

- Write $c \cdot n$

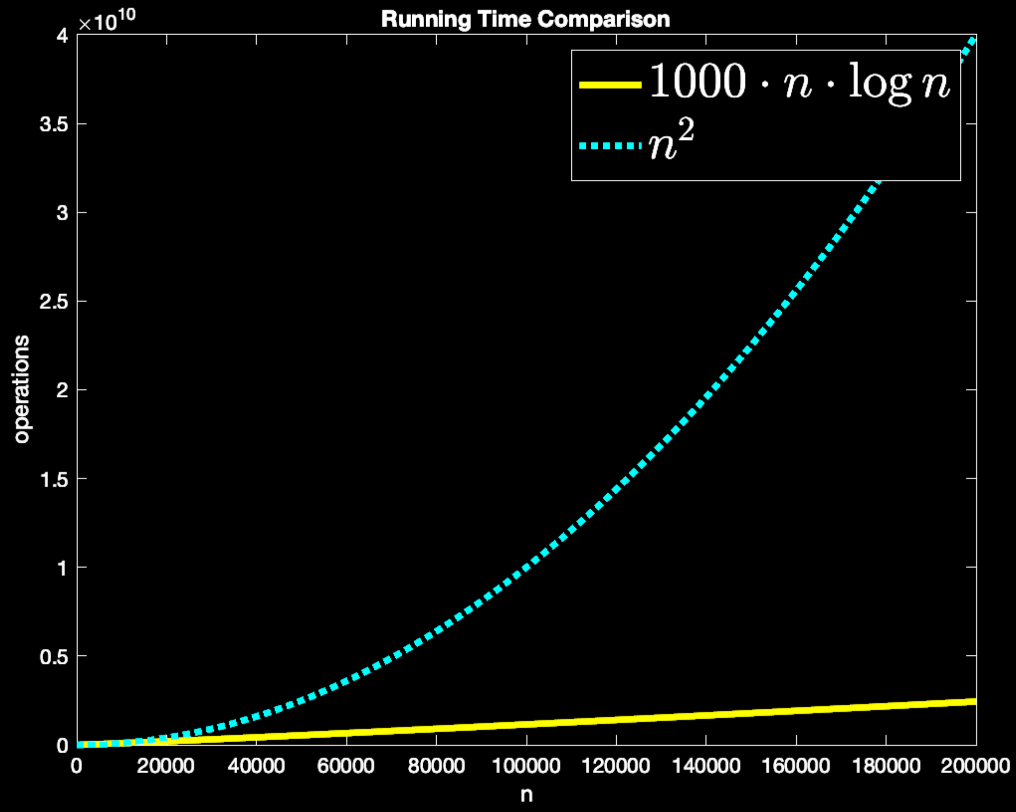
Never write lower order terms inside the $O(\cdots)$

It's unnecessary

- Each term is ≥ 0 for $n \geq 1$. QED

Example

Does $1000 \cdot n \cdot log(n) = O(n^2)$?



Example

Does $1000 \cdot n \cdot log(n) = O(n^2)$?

1. Set $c = ???$ and $n_0 = ???$
   - Set $c = 1000$ and $n_0 = 1$
2. Show that $1000 \cdot n \cdot log(n) \leq 1000 \cdot n^2$ when $n \geq 1$

$$0 \leq 1000 \cdot n^2 - 1000 \cdot n \cdot log(n) = 1000 \cdot n^2 - 1000 \cdot n \cdot log(n) \geq 0$$

3. When $n = 1, 1000 \cdot n^2 - 1000 \cdot n \cdot log(n) > 0.$ QED
   Moreover, the difference grows as n increases!

Example

3. When $n = 1, 1000 \cdot n^2 - 1000 \cdot n \cdot log(n) > 0.$ QED
   Moreover, the difference grows as n increases!

   Prove it!

   Consider the derivative of the difference:

   $$= \frac{d}{dn} 1000 \cdot n^2 - 1000 \cdot n \cdot log(n)$$

   $= 2000 \cdot n - 1000 - 1000 \cdot log(n),$ which is > 0 for $n = 1\ldots$

Example

$= 2000 \cdot n - 1000 - 1000 \cdot log(n),$ which is > 0 for $n = 1\ldots$

But does it stay > 0?

Consider the second derivative:

$$= \frac{d^2}{dn^2} 1000 \cdot n^2 - 1000 \cdot n \cdot log(n)$$

$= 2000 - \frac{1000}{n}$, which is > 0 for $n \geq 1$.

Hence is remains positive and the difference increases.

Summary

- You can use calculus to show that one function remains greater than another past a certain point, *even if the functions are not algebraic*.
- This is often the crucial step in proving $f(n) = O(g(n))$
- Big-O makes our intuition about one function being an “upper bound” for another precise, ignoring constant factors and small input sizes.
    - Big-O matches our (current) goals to be a tool to compare algorithms!

Extensions of Big-O: $\Omega()$ and $\Theta()$

More Precise Boundaries

- Currently we can express the concept of an upper bound:
  - $f()$ is below or at (≤) $g()$
    - It could be more specific.
  - With numbers we'd be pretty limited with just $x \leq y$, but not also $x \geq y$ or $x = y$
  - We'd like more precise statements, like $\geq$ and $=$

Definition of Ω ($\geq$)

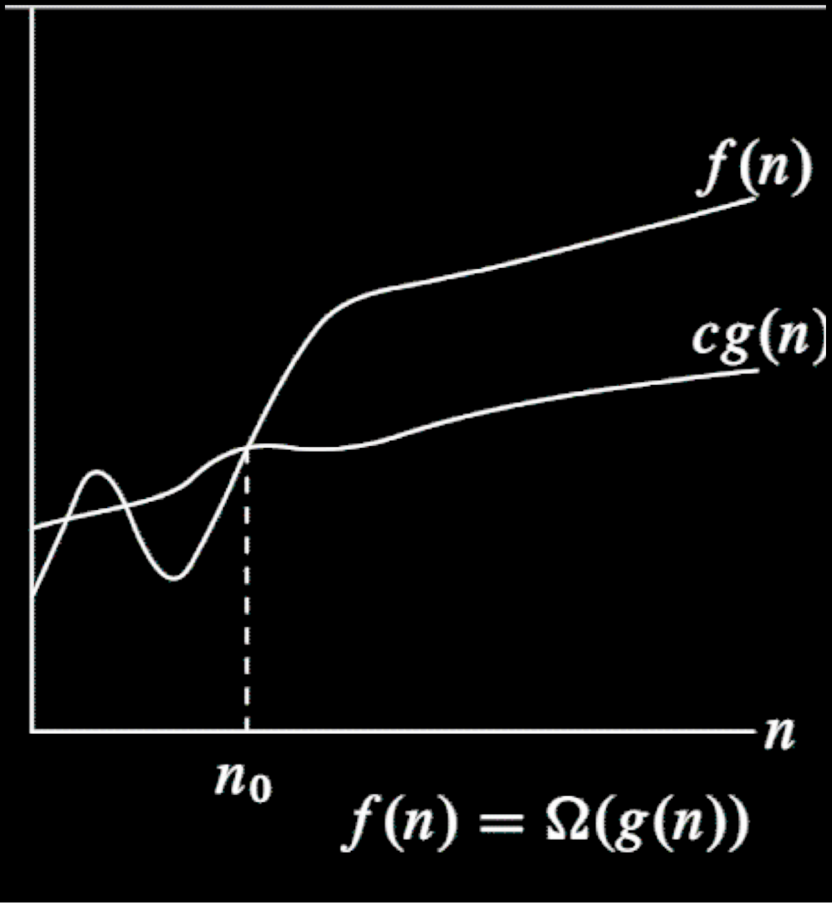
- Let $f(n)$ and $g(n)$ be non-negative functions for $n > 0$
  - Again, running times or memory
- $f(n) = \Omega(g(n))$ if there exists constants $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0$

$$f(n) \geq c \cdot g(n)$$

Definition of Ω ($\geq$)

- Let $f(n)$ and $g(n)$ be non-negative functions for $n > 0$
- $f(n) = \Omega(g(n))$ if there exists constants $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0$

$$f(n) \geq c \cdot g(n)$$



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

- Lemma: $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$
- So, if we want to prove: $n^2 = \Omega(n \cdot \log(n))$
  - We prove $n \cdot \log(n) = O(n^2)$

Proof of Lemma

$f(n) = O(g(n))$ **iff** $g(n) = \Omega(f(n))$

- if $f(n) = O(g(n))$, there are $c > 0$ and $n_0 > 0$ such that for $n \geq n_0, f(n) \leq c \cdot g(n)$
- Set $d = \frac{1}{c}$. Then for $n \geq n_0$, $g(n) \geq d \cdot f(n)$
- Conclude that with constants d and n_0 we have proved that $g(n) = \Omega(f(n))$
  - A similar argument works to prove the other direction of the iff. QED.

Definition of Θ (=)

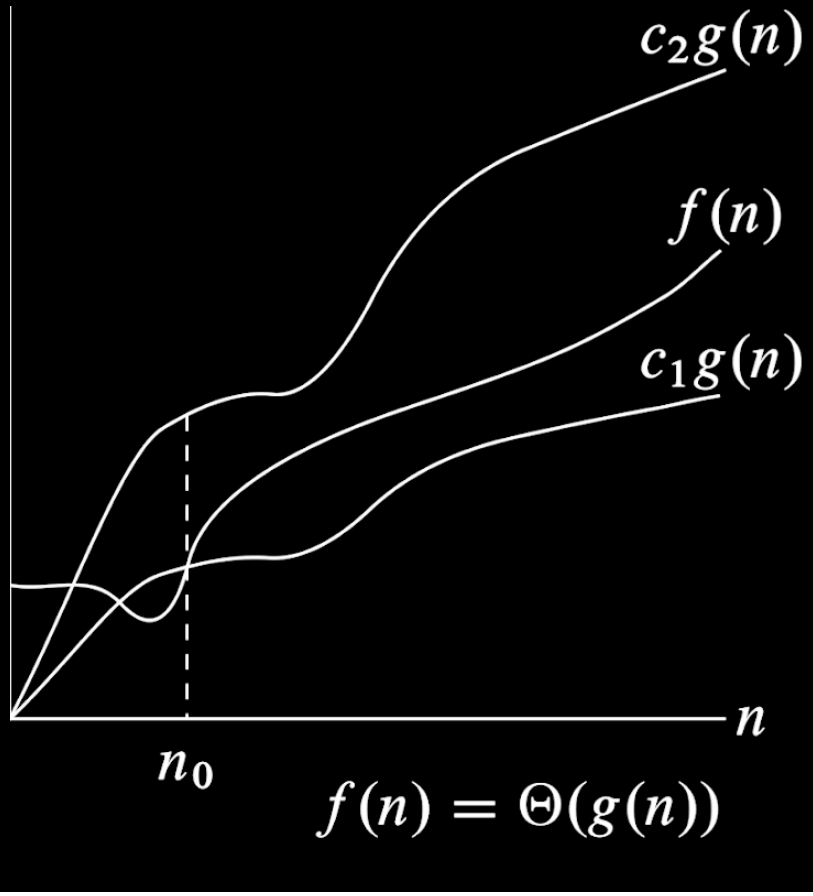
- Let $f(n)$ and $g(n)$ be non-negative functions for $n > 0$
  - Again, running times or memory
- $f(n) = \Theta(g(n))$ if there exists constants $c_1 > 0$, $c_2 > 0$, and $n_0 > 0$ such that for all $n \geq n_0$

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

Definition of Θ (=)

- Let $f(n)$ and $g(n)$ be non-negative functions for $n > 0$
- $f(n) = \Theta(g(n))$ if there exists constants $c_1 > 0$, $c_2 > 0$, and $n_0 > 0$ such that for all $n \geq n_0$

$$c_1 \cdot g(n) \leq f(n) \leq c_1 \cdot g(n)$$



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

- Lemma: $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ <u>and</u> $f(n) = \Omega(g(n))$
- So, we want to prove: $3n^2 + 11n = \Theta(n^2)$

You should be able to prove this from definitions of O, Ω, and Θ

Conclusions

- We have a *precise* way to bound behaviors of functions when $n$gets large, ignoring constant factors.
- We can replace ugly precise running times by much simpler expressions with the same asymptotic behavior!
- You will see O, Ω, and Θ frequently this semester!