Lecture 1: Asymptotic Complexity

These slides include material originally prepared by Dr.Ron Cytron, Dr. Jeremy Buhler, and Dr. Steve Cole.

Announcements

- TA office hours officially start this week – see web site.
- **Lab 1 released this Wednesday**
  - due **2/8** at **11:59 PM**
  - (***work on your own*** *– it's a lab*)
- There is no coding for this lab, just the written part.
- **Please review and follow the eHomework guidelines for this and future lab writeups. Read the Gradescope turn-in guide at the bottom of the eHomework guidelines.**

Announcements, Cont’d

- If you joined the class on or after last Thursday, 1/17, you must make up Studio 0 by showing your writeup to a TA in office hours by **1/31**
  - **See the website for office hours times and locations**

- **Please check that you have a Gradescope account.**
  - Those who joined by the first day of class should have gotten an invite email.
  - If you did not, or if you cannot see CSE 247, go to https://www.gradescope.com, create an account if needed, and register for class code

M7DRK3

Things You Saw in Studio 0

- “Ticks” are a useful way to measure complexity -- count # of times we reach a specific place in the code.

- Growing array by doubling takes time *linear in # of elements added.*

- (“Naïve approach” took quadratic time!)

- **We can reason about the number of ticks (≈ running time) of a program analytically, without actually running it.**

Today's Agenda

- Counting the number of ticks exactly
- Asymptotic complexity
- Big-O notation – *being sloppy, but in a* ***very precise*** *way*
- Big-Ω notation – the opposite (?) of big-O
- Big-Θ notation – how to say "about a constant times f(n)"

How Many Times Do We Tick?

- Let's take an example from the studio:

```
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();
    }
}
```

How many times do we call tick()?

How Many Times Do We Tick?

- Let’s take an example from the studio:

```
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();
    }
}
```

“Once for each value of i in the loop”

How Many Times Do We Tick?

- **Let's take an example from the studio:**

```java
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();
    }
}
```

So, for i = 0, 1, 2, … ???

How Many Times Do We Tick?

- **Let's take an example from the studio:**

```java
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();
    }
}
```

So, for i = 0, 1, 2, … n-1

How Many Times Do We Tick?

- **Let's take an example from the studio:**

```
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();
    }
}
```

So, for i = 0, 1, 2, … n-1 *(not n, because <)*

Accounting

- One tick per loop iteration.
- Total tick count is therefore
- $\sum_{i=0}^{n-1} (1)$

Greatest value of i in loop

Least value of i in loop

Accounting

- One tick per loop iteration.
- Total tick count is therefore
- $\sum_{i=0}^{n-1}(1) = (n-1) - 0 + 1 = n$

Accounting

- One tick per loop iteration.
- Total tick count is therefore
- $\sum_{i=0}^{n-1}(1) = (n-1) - 0 + 1 = n$

First rule of counting: a loop from i = LO to i = HI runs

HI – LO + 1 times

Let’s Try a Doubly-Nested Loop

- Now consider this code:

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

How many times do we call tick()?

Let's Work from the Inside Out

- Innermost loop runs for j from 0 to … ???

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

Let's Work from the Inside Out

- Inner loop runs for j from 0 to … i-1

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

Hence, we tick *(i-1) – 0 + 1* = *i* times each time we execute the inner loop.

Let's Work from the Inside Out

- Outer loop runs for i from 0 to … ???

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

        i ticks

    }
}
```

Let's Work from the Inside Out

- Outer loop runs for i from 0 to … n-1

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

        i ticks

    }
}
```

But this time, the number of ticks is different for each i!

Accounting

- i ticks per outer loop iteration
- Total tick count is therefore
- $\sum_{i=0}^{n-1} i$

Accounting

- i ticks per outer loop iteration.
- Total tick count is therefore
- $\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$

Remember this from last time? We'll use it a lot!

Accounting

- i ticks per outer loop iteration.
- Total tick count is therefore
- $\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$

Second rule of counting: when loops are nested,

Work inside-out and form a summation.

One More Time

- Instead of Java, let's do *pseudocode*.

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

One More Time…

- Instead of Java, let's do *pseudocode*.

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

"For j from 1 to n, ***inclusive***"

One More Time

- Instead of Java, let's do *pseudocode*.

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

Inner loop runs
for k from 0 to j
and ticks
3 times
per iteration

One More Time

- Instead of Java, let's do *pseudocode*.

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

Inner loop runs
j – 0 + 1 = j+1 times
and ticks
3 times
per iteration

One More Time

- Instead of Java, let's do *pseudocode*.

```
for j in 1 … n
    tick()
```

3(j+1) ticks

Inner loop runs
j – 0 + 1 = j+1 times
and ticks
3 times
per iteration

One More Time

- Instead of Java, let's do *pseudocode*.

```
for j in 1 … n
    tick()
```

3(j+1) ticks

Outer loop runs
for j from 1 to n
and ticks
??? times
on iteration j

One More Time

- Instead of Java, let's do *pseudocode*.

```
for j in 1 … n
    tick()
```

3(j+1) ticks

Outer loop runs
for j from 1 to n
and ticks
1 + 3(j+1) = 3j+4 times
on iteration j

Accounting

- **3j+4 ticks per outer loop iteration.**
- **Total tick count is therefore**
- $\sum_{j=1}^{n} (3j + 4)$

Accounting

- 3j+4 ticks per outer loop iteration.
- Total tick count is therefore
- 

(smite with the power of ~~Mjollnir~~ algebra)

Accounting

- 3j+4 ticks per outer loop iteration.
- Total tick count is therefore
- $\sum_{j=1}^{n} (3j + 4) = \frac{3n(n+1)}{2} + 4\mathbf{n} = \frac{3n^2+11n}{2}$

Do We Really Care?

- Seriously, $\frac{3n^2+11n}{2}$???

- Do we need this much detail to understand our code’s running time?

How Do We Actually Use Running Times?

- Predict exact time to complete a task

How Do We Actually Use Running Times?

- Predict exact time to complete a task
  (yeah, we need the precise count for this)

How Do We Actually Use Running Times?

- Predict exact time to complete a task
  (yeah, we need the precise count for this)

- Compare running times of different algorithms

How Do We Actually Use Running Times?

- Predict exact time to complete a task
  (yeah, we need the precise count for this)

- Compare running times of different algorithms

$1000\ n \log n$ n^2 $3n^2$

How Do We Actually Use Running Times?

- Predict exact time to complete a task
  (yeah, we need the precise count for this)

- Compare running times of different algorithms

1000 n log n **n^2** **$3n^2$**

Difference is a constant factor
*(solved by using a **bigger computer**)*

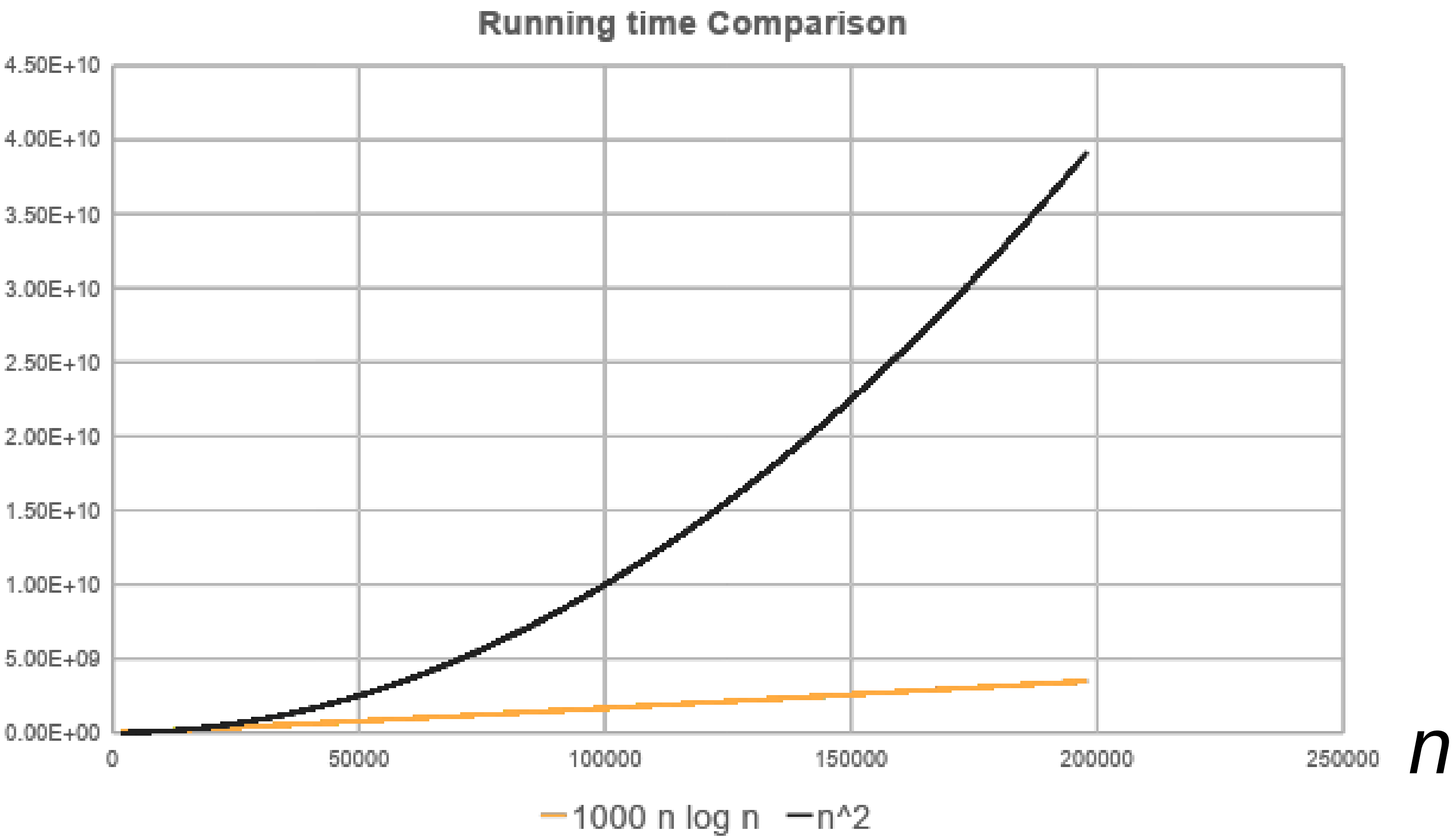
How Do We Actually Use Running Times?

- Predict exact time to complete a task
  (yeah, we need the precise count for this)

- Compare running times of different algorithms

$1000\ n \log n$ **n^2** **$3n^2$**

Qualitatively different!



Running time Comparison
4.50E+10
4.00E+10
3.50E+10
3.00E+10
2.50E+10
2.00E+10
1.50E+10
1.00E+10
5.00E+09
0.00E+00
0
50000
100000
150000
200000
250000
n
1000 n log n
n^2

Desirable Properties of Running Time Estimates

- Distinguish “get a bigger computer” vs “qualitatively different” → **order of growth matters** (constant factors don’t)

Desirable Properties of Running Time Estimates

- Distinguish “get a bigger computer” vs “qualitatively different”
  → **order of growth matters** (constant factors don’t)

- Ignore transient effects for small input sizes n


Running time Comparison
1.00E+09
9.00E+08
8.00E+08
7.00E+08
6.00E+08
5.00E+08
4.00E+08
3.00E+08
2.00E+08
1.00E+08
0.00E+00
0
5000
10000
15000
20000
25000
30000
35000
n
1000 n log n
n^2

Desirable Properties of Running Time Estimates

- Distinguish "get a bigger computer" vs "qualitatively different" → **order of growth matters** (constant factors don't)

- Ignore transient effects for small input sizes n
    - ***Standard assumption****: we care what happens as input becomes "large" (grows without bound)*
    - *In other words, we care about* ***asymptotic behavior*** *of an algorithm's running time!*


"BIG DATA"?
Oh, you must mean
asymptotic complexity.
imgflip.com

How do we reason about asymptotic behavior?

Time for Theory!

Definition of Big-O Notation

- Let f(n), g(n) be positive functions for $n > 0$.

Definition of Big-O Notation

- Let f(n), g(n) be positive functions for n > 0.
  [e.g. running times!]

Definition of Big-O Notation

- Let f(n), g(n) be positive functions for n > 0.
  [e.g. running times!]

- We say that **f(n) = O(g(n))** if there exist constants

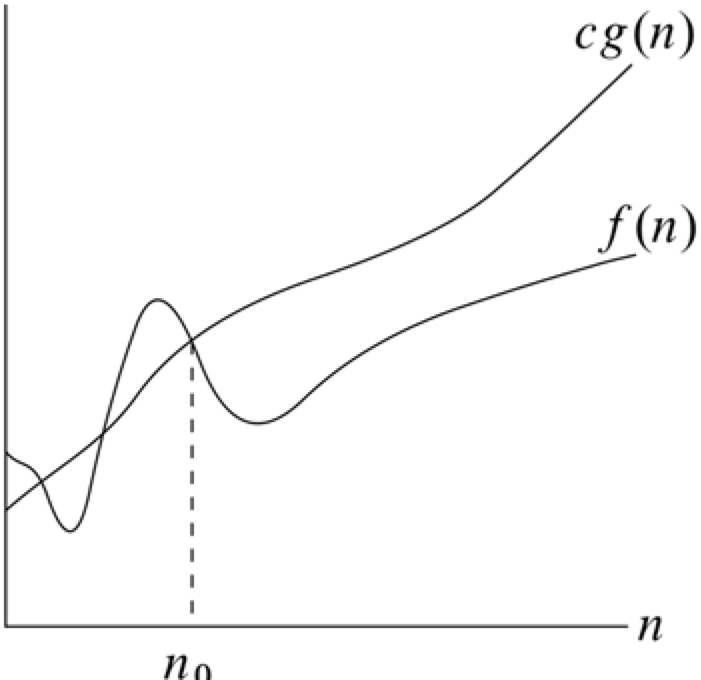
  $$c > 0, n_0 > 0$$

  such that for all $n \geq n_0$,

  $$f(n) \leq c \cdot g(n).$$

There exist constants $c > 0$, $n_0 > 0$ such that for all $n \geq n_0$,

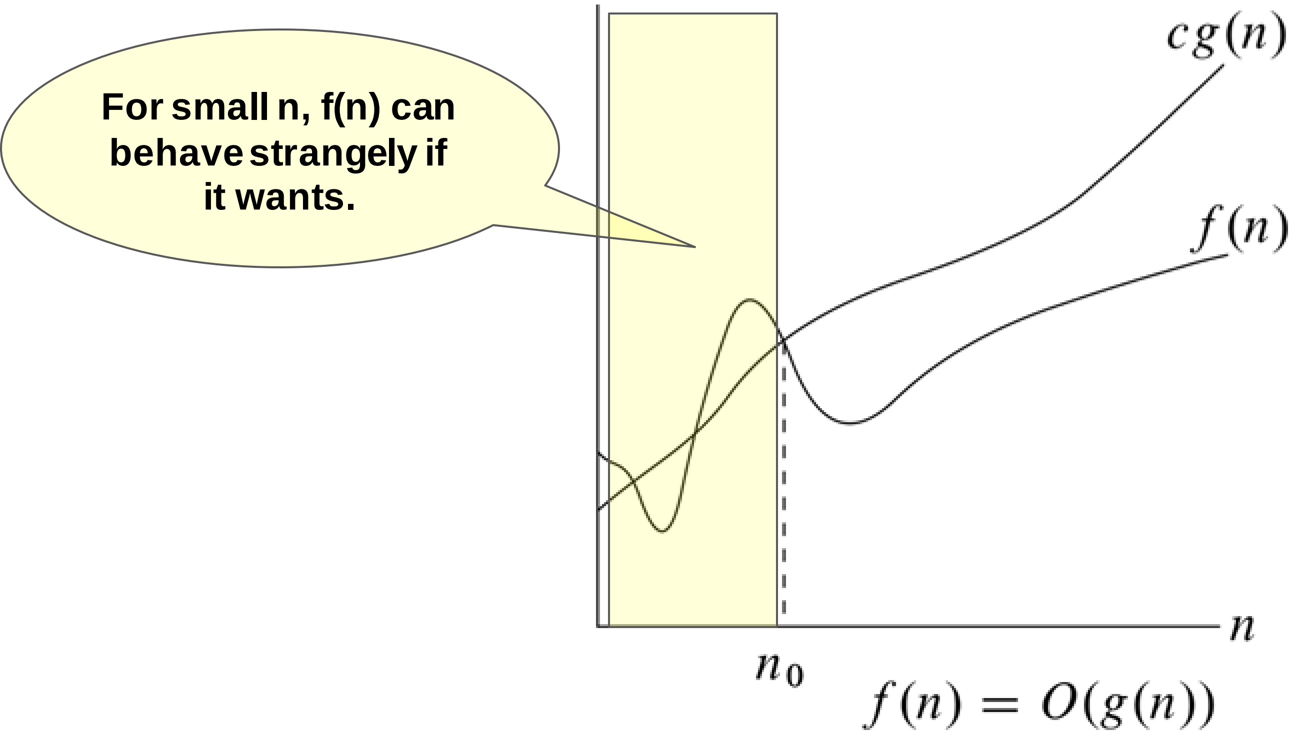
$f(n) \leq c \cdot g(n)$.



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

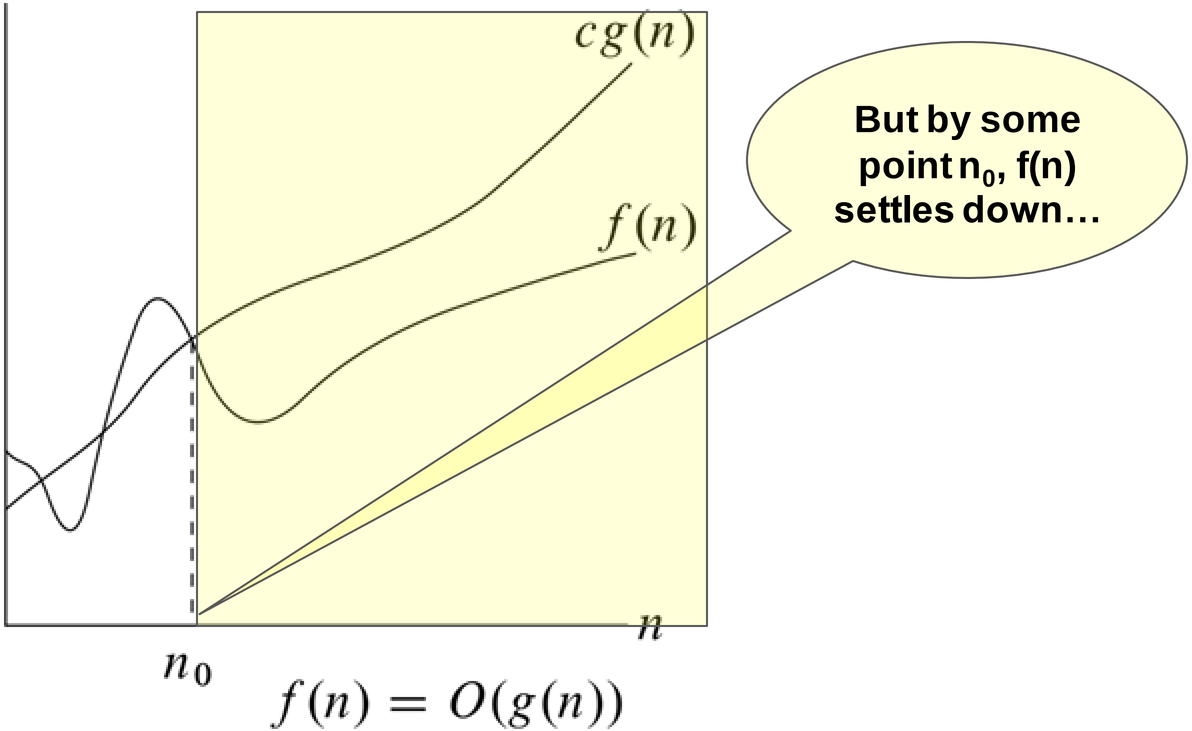
There exist constants $c > 0$, $n_0 > 0$ such that for all $n \geq n_0$,

$f(n) \leq c \cdot g(n)$.



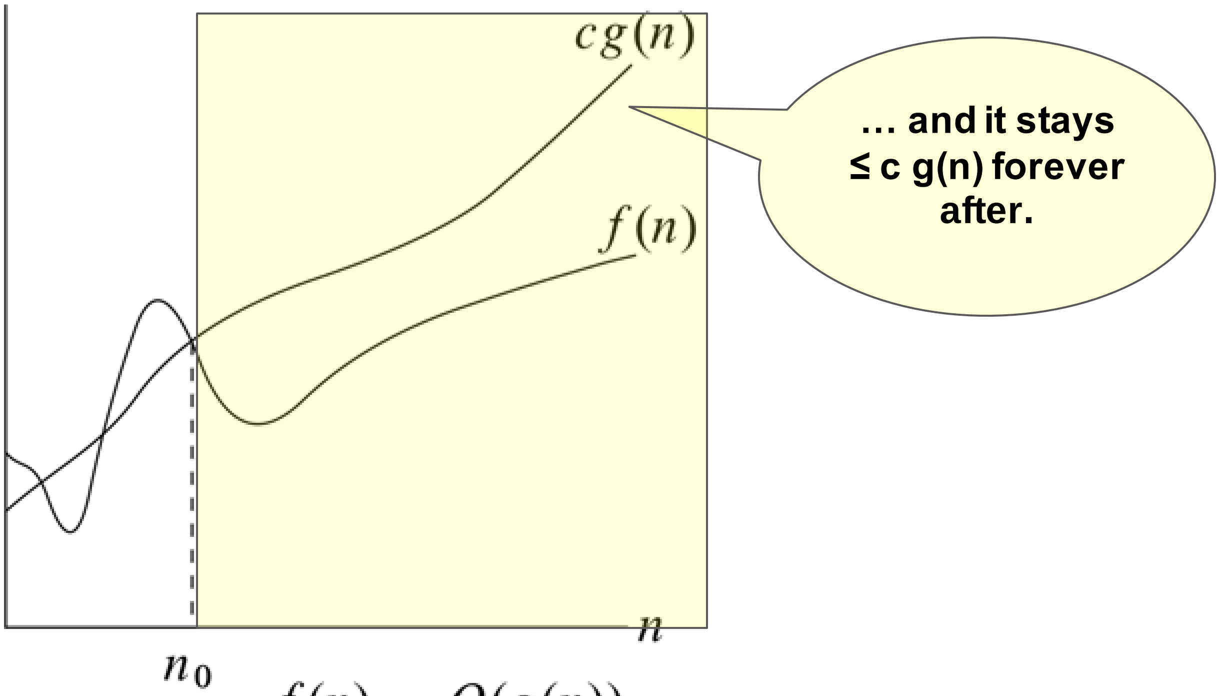
There exist constants $c > 0$, $n_0 > 0$ such that for all $n \geq n_0$,

$$\mathbf{f(n) \leq c \cdot g(n)}.$$



There exist constants $c > 0$, $n_0 > 0$ such that for all $n \geq n_0$,

$f(n) \leq c \bullet g(n)$.



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

Does Big-O Have the Properties We Desire?

- Explicitly ignores behavior of functions for small n *(we get to decide what "small" is).*

- Allows a constant *c* in front of g(n) for upper bound.

- *Does that make big-O insensitive to constants?*

Big-O Ignores Constants, as Desired

- **Lemma: If $f(n) = O(g(n))$, then $f(n) = O(a\, g(n))$ for *any* $a > 0$.**

- **Pf**: $f(n) = O(g(n))$ → for some $c > 0$, $n_0 > 0$, if $n \geq n_0$,

$$f(n) \leq c\, g(n).$$

- But then for $n \geq n_0$,

$$f(n) \leq \frac{c}{a} \cdot a\, g(n).$$

- Conclude that $f(n) = O(a\, g(n))$. QED

Big-O Ignores Constants, as Desired

- **Lemma: If f(n) = O(g(n)), then f(n) = O(a g(n)) for *any* a > 0 .**
- **Pf**: f(n) = O(g(n)) → for some $c > 0$, $n_0 > 0$, if $n \geq n_0$,

  $$f(n) \leq c\, g(n).$$

- But then for $n \geq n_0$,

  $$f(n) \leq \frac{c}{a}\, a\, g(n).$$

- Conclude that f(n) = O(a g(n)). QED

When specifying running times, never write a constant inside the O(). It is unnecessary.

Does big-O Match our Intuition?

- Which function grows faster, n or n^2?

Does big-O Match our Intuition?

- Which function grows faster, n or n^2? [quadratic beats linear]
- So does $n = O(n^2)$?
- Set c = ???, n_0 = ??? [many options here]

Does big-O Match our Intuition?

- Which function grows faster, n or n^2? [quadratic beats linear]
- So does $n = O(n^2)$?
- Set $c = 1$, $n_0 = 1$ [many options here]
- When $n \geq 1$, is $1 \cdot n^2 \geq n$?

Does big-O Match our Intuition?

- Which function grows faster, n or n^2? [quadratic beats linear]
- So does $n = O(n^2)$?
- Set $c = 1$, $n_0 = 1$ [many options here]
- When $n \geq 1$, is $1 \cdot n^2 \geq n$?
- Yes! – multiply both sides of "$n \geq 1$" by n. QED

General Strategy for Proving $f(n) = O(g(n))$

1. Pick $c > 0$, $n_0 > 0$. [*choose to make next steps easier*]
2. Write down desired inequality $f(n) \leq c\, g(n)$.
3. Prove that the inequality holds whenever $n \geq n_0$.

Another Example

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

Another Example

- Does $3n^2 + 11n = O(n^2)$? [what does your intuition say?]

- Let's prove it.

- Set c = ???, n_0 = ???

Another Example

- Does $3n^2 + 11n = O(n^2)$? [what does your intuition say?]
- Let's prove it.
- Set $c = 33$, $n_0 = 1$ [again, many possible choices]
- For $n \geq 1$, difference

$$33n^2 - (3n^2 + 11n) = (11n^2 - 3n^2) + (11n^2 - 11n) + (11n^2 - 0) > 0.$$

Conclude that the claim is true. *QED*

Generalization of Previous Proof

- **Thm**: any polynomial of the form $s(n) = \sum_{j=0}^{k} a_j \, n^j$ is O(n^k).
- **Pf**: pick c to be k+1 times the largest (most positive) a_j; pick n_0 = 1.
- Write $cn^k - s(n)$ as

$$\sum_{j=0}^{k} \left(\frac{c}{k+1} n^k - a_j n^j\right),$$

each term of which is ≥ 0 for n ≥ 1. QED

Generalization of Previous Proof

- **Thm**: any polynomial of the form $s(n) = \sum_{j=0}^{k} a_j \, n^j$ is O(n^k).

- **Pf**: pick c to be k+1 times the largest a_j and pick $n_0 = 1$

- Write $cn^k - s(n)$ as

$$\sum_{j=0}^{k}\left(\frac{c}{k+1} n^k - a_j n^j\right),$$

each term of which is ≥ 0 for n ≥ 1. QED

When specifying running times, never write lower-order terms inside the O(). It is unnecessary.

Generalization of Previous Proof

- **Thm**: any polynomial of the form $s(n) = \sum_{j=0}^{k} a_j\, n^j$ is O(n^k).
- **Pf**: pick c to be k+1 times the largest a_j and pick n_0 = 1.
- Write $cn^k - s(n)$ as

$$\sum_{j=0}^{k}\left(\frac{c}{k+1}n^k - a_j n^j\right),$$

each term of which is ≥ 0 for n ≥ 1. QED

Based on these two examples, we can *prove*

$$\frac{3n^2+11n}{2} = O(n^2)$$

Generalization of Previous Proof

- **Thm**: any polynomial of the form $s(n) = \sum_{j=0}^{k} a_j\, n^j$ is O(n^k).
- **Pf**: pick c to be k+1 times the largest a_j, and pick n_0 = 1.
- Write $cn^k - s(n)$ as

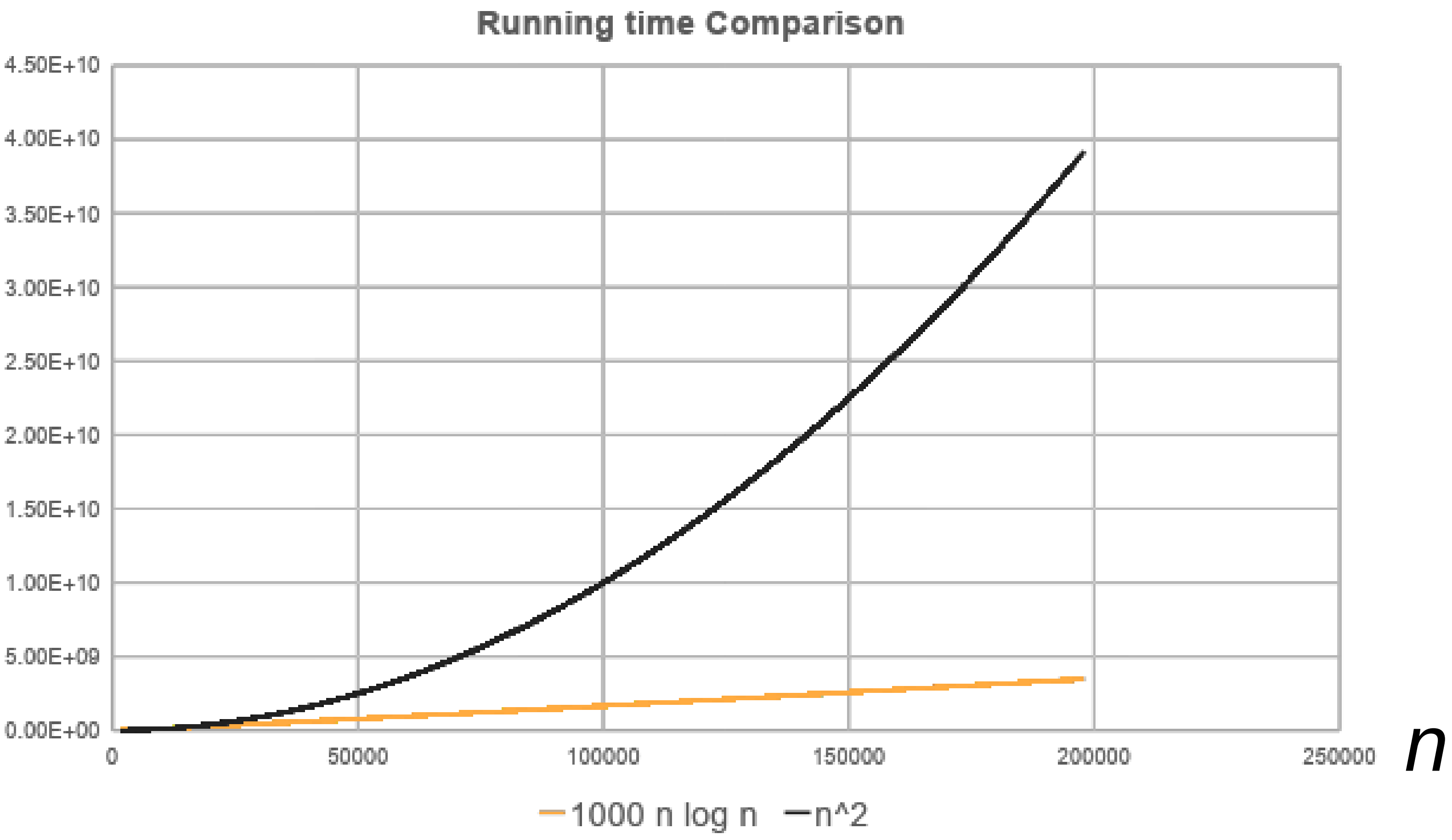
$$\sum_{j=0}^{k} \left(\frac{c}{k+1} n^k - a_j n^j\right),$$

each term of which is ≥ 0 for n ≥ 1. QED

Polynomial terms other than the highest do not impact asymptotic complexity!

One More Example

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


Running time Comparison
4.50E+10
4.00E+10
3.50E+10
3.00E+10
2.50E+10
2.00E+10
1.50E+10
1.00E+10
5.00E+09
0.00E+00
0
50000
100000
150000
200000
250000
n
1000 n log n
n^2

One More Example

- Does 1000 n log n = $O(n^2)$?
- Set c = ???, n_0 = ???

One More Example

- Does $1000 \, n \log n = O(n^2)$?
- Set $c = 1000$, $n_0 = 1$
- When $n = 1$, $1000 \, n^2 - 1000 \, n \log n = 1000 > 0$.
- Moreover, *this difference only grows with increasing $n > 1$*. QED

One More Example

- Does 1000 n log n = $O(n^2)$?
- Set c = 1000, n_0 = 1
- When n = 1, 1000 n^2 – 1000 n log n = 1000 > 0.
- Moreover, *this difference only grows with increasing n > 1*. QED

(Oh really? Are you sure?)

One More Example

- Well, the derivative of the difference

$$\frac{d}{dn}[1000\, n^2 - 1000\, n \log n] = 2000\, n\ - 1000\ - 1000 \log n,$$

which is > 0 for n = 1. *But does it stay that way for n > 1?*

One More Example

- Well, the derivative of the difference

$$\frac{d}{dn}[1000\, n^2 - 1000\, n \log n] = 2000\, n\ - 1000\ - 1000 \log n,$$

which is > 0 for n = 1. *But does it stay that way for n > 1?*

- Furthermore,

$$\frac{d^2}{dn^2}\ [1000n^2 - 1000\, n \log n] = 2000 - 1000/n,$$

which is > 0 for n ≥ 1. Hence, the derivative *remains* positive, and so the difference increases for n ≥ 1 as claimed.

Moral

- 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 a crucial step in proving f(n) = O(g(n)).
- (*Next time, we'll use this idea to derive a general test for comparing the asymptotic behavior of two functions.*)

Big-O makes precise our intuition about when one function effectively upper-bounds another, ignoring constant factors and small input sizes.

Extensions of Big-O Notation: Ω and Θ

More Ways to Bound Running Times

- When comparing numbers, we would *not* be happy if we could say

  "x ≤ y"

  but not

  "x ≥ y" or "x = y"

- Big-O is analogous to ≤ for functions [*upper bound on growth rate*]

- **What are statements analogous to ≥, =?**

More Ways to Bound Running Times

- When comparing numbers, we would *not* be happy if we could say

  “x ≤ y”

  but not

  “x ≥ y” or “x = y”

- Big-O is analogous to ≤ for functions [*upper bound on growth rate*]

- **What are statements analogous to ≥, =?**

  Ω, Θ

Definition of Big-Ω Notation

- Let f(n), g(n) be positive functions for $n > 0$.
  [e.g. running times!]

- We say that **f(n) = Ω(g(n))** if there exist constants

  $$c > 0,\ n_0 > 0$$

  such that for all $n \geq n_0$,

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

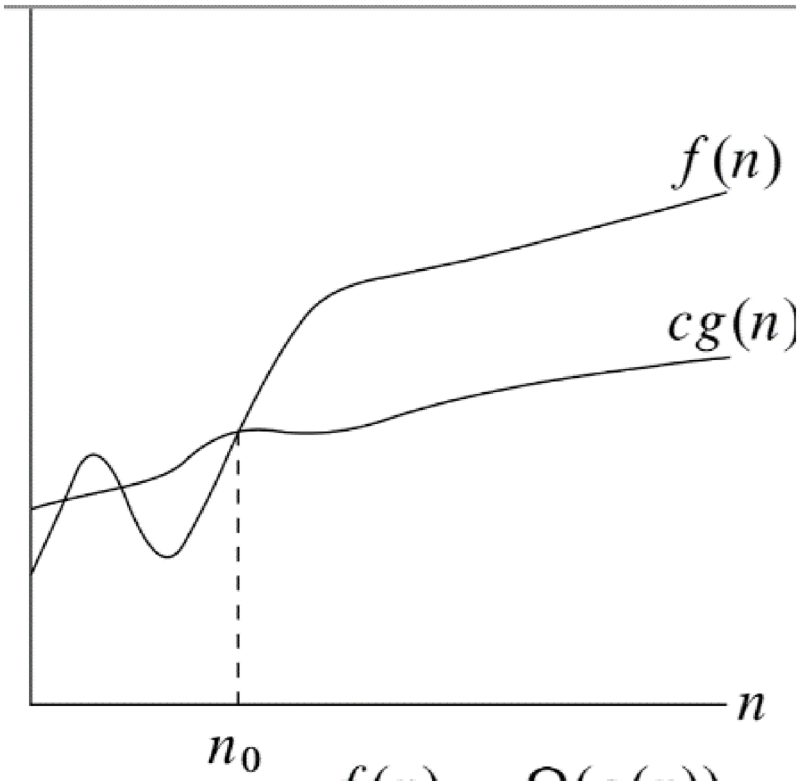
Definition of Big-Ω Notation

- Let f(n), g(n) be positive functions for $n > 0$.
  [e.g. running times!]

- We say that **f(n) = Ω(g(n))** if there exist constants
  $c > 0, n_0 > 0$
  such that for all $n \geq n_0$,
  $f(n) \geq c \cdot g(n)$.

There exist constants $c > 0$, $n_0 > 0$ such that for all $n \geq n_0$,

$\mathbf{f(n) \geq c \cdot g(n)}$.



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

How Do You Prove $f(n) = \Omega(g(n))$?

- Lemma:

$$f(n) = O(g(n)) \text{ iff } g(n) = \Omega(f(n))$$

- So if we want to prove, say,

$$n^2 = \Omega(n \log n),$$

we just prove

$$n \log n = O(n^2).$$

(Proof of Lemma)

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

Definition of Big-Θ Notation

- Let f(n), g(n) be positive functions for n > 0.
  [e.g. running times!]

- We say that **f(n) = Θ(g(n))** if there exist constants

$$c_1, c_2 > 0, n_0 > 0$$

such that for all $n \geq n_0$,

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

Definition of Big-Θ Notation

- Let f(n), g(n) be positive functions for $n > 0$.
  [e.g. running times!]

- We say that **f(n) = Θ(g(n))** if there exist constants

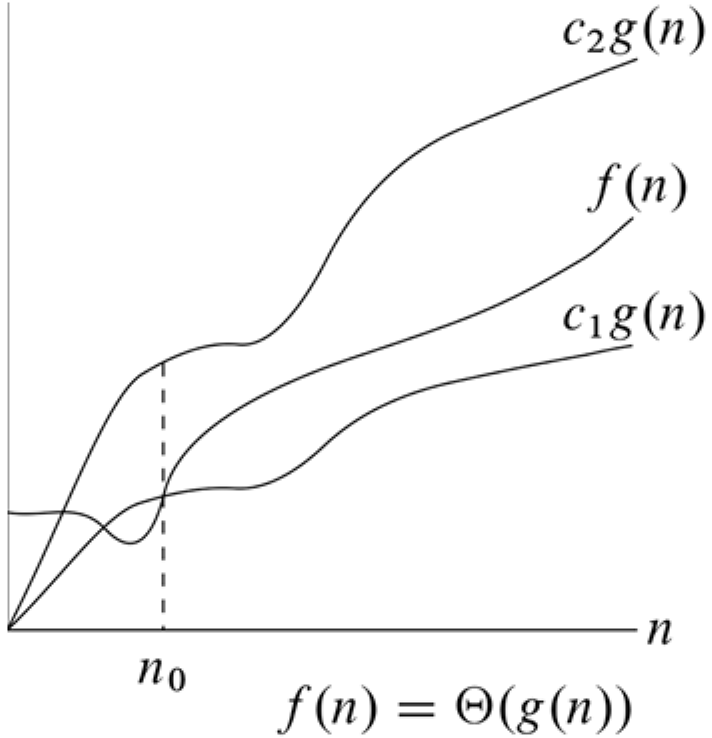
  $$c_1, c_2 > 0, n_0 > 0$$

  such that for all $n \geq n_0$,

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

There exist constants $c_1, c_2 > 0$, $n_0 > 0$ s.t. for all $n \geq n_0$,

$$\mathbf{c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)}.$$



Upper *and* lower bounds on f(n)
(might not be same constant)

How Do You Prove $f(n) = \Theta(g(n))$?

- Lemma:

$$f(n) = \Theta(g(n)) \text{ iff}$$
$$f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))$$

- So if we want to prove, say,

$$3n^2 + 11n = \Theta(n^2),$$
we just prove
$$3n^2 + 11n = O(n^2) \text{ and } 3n^2 + 11n = \Omega(n^2)$$

How Do You Prove $f(n) = \Theta(g(n))$?

- Lemma:

$$f(n) = \Theta(g(n)) \text{ iff}$$
$$f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))$$

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

Conclusion (so far)

- We now have **precise** way to bound behavior of fcns *when n gets large, ignoring constant factors*.

- We can replace ugly precise running times by much simpler expressions with same asymptotic behavior.

- You will see O, Ω, and Θ frequently for rest of 247!

Next Time…

- Quick, *uniform* proof strategy for O, Ω, and Θ statements
- Review of linked lists for Studio 2
- More practice applying asymptotic complexity

End of Asymptotic Complexity Part 1

continued next lecture