Announcements

- EX02 due Friday night
- P1 Checkpoint 1 due Tuesday
  - Fill out **BOTH** checkpoint form AND gradescope

A word on checkpoints and projects

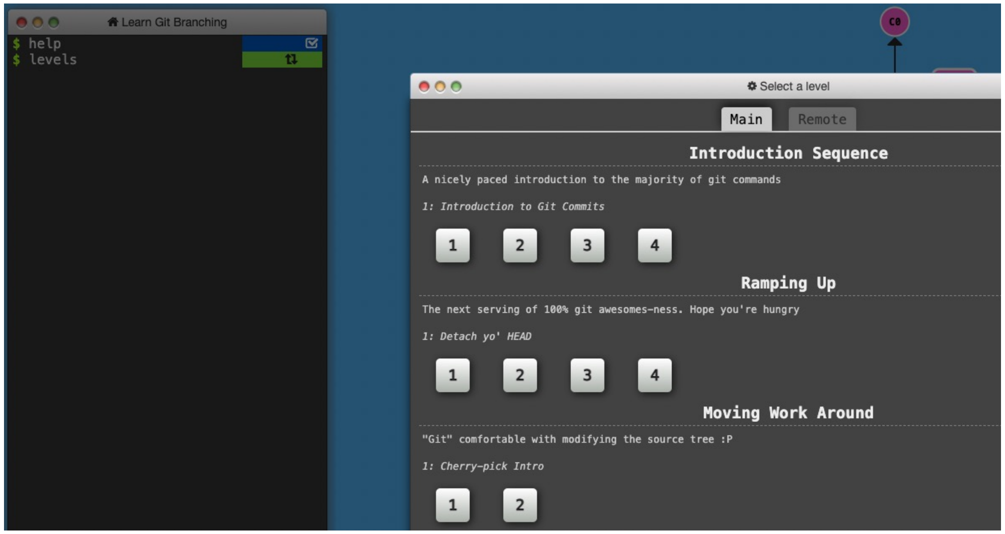
- Checkpoints are (somewhat) lenient
  - Don't expend all your energy staying up late if you can't find the bug
  - That being said, a strong signal that you are behind and need to spend extra time to catch up
    - Stay on top of things! Easier to catch the train then to catch it after it leaves

- Use the Ed board!
  - If possible, make questions public (can make anonymous)

- Read Javadoc
  - Read the comments in the repository!
  - IntelliJ has functionality to jump to the reference (the interface, implementation of method, etc. Super helpful!)

Git good at git

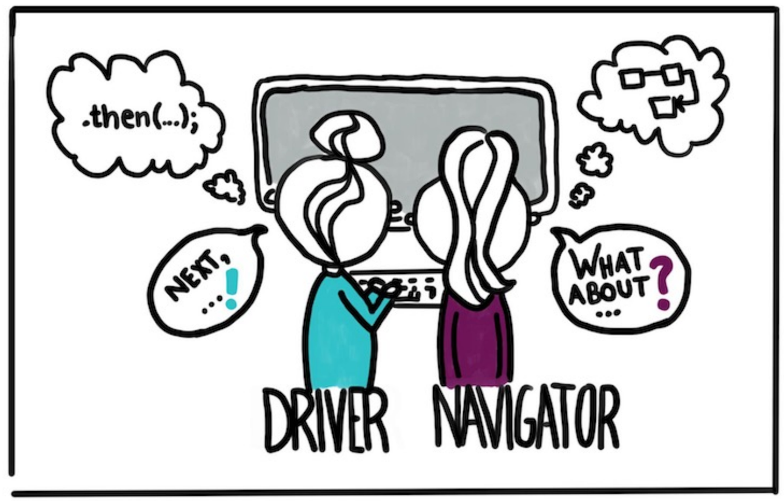
- Git is a really good resource
  - https://learngitbranching.js.org/
  - Make frequent commits and messages (helpful for debugging!)



Pair Programming

- Divide & Conquer
  - Boo :(
  - Good for algorithms, not so much for coding. Easy to miss subtle bugs… doubling your mistakes!

- Pair Programming
  - Yay :)
  - You are responsible for knowing your code!



Today

- Big-Oh Definition
  - And other friends
- Proofs
- Amortization

Recap:

1. Look at code and get a function of how long it runs based on input size
   - Heuristic: counting operations!
   - Too detailed, so… step 2…

2. Group it into a Big Oh set of functions
   - asymptotic behavior
   - Informally: “Drop” coefficients, lower-order terms
   - Formally: Find c and n_0

```
for (i=0; i<n;i++){
  ...
}
```

$n \cdot 5 + 1 + 1$

$3n + 4$

$100n + 30 \log n$

$O(1)$	$O(n)$	$O(n^2)$
	$3n+4$	$3n+4$

What you can drop

- Eliminate coefficients because we don't have units anyway
  - $3n^2$ versus $5n^2$ doesn't mean anything when we cannot count operations very accurately
- Eliminate low-order terms because they have vanishingly small impact as n grows
- Do NOT ignore constants that are not multipliers
  - n^3 is not $O(n^2)$
  - 3^n is not $O(2^n)$

(This all follows from the formal definition) (We can prove it!)

Big Oh: Common Categories

fastest

From fastest to slowest

$O(1)$	constant (same as $O(k)$ for constant k)
$O(\texttt{log}\ n)$	logarithmic
$O(n)$	linear
$O(n\ \texttt{log}\ n)$	"n **log** *n*"
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(n^k)$	polynomial (where is k is any constant > 1)
$O(k^n)$	exponential (where k is any constant > 1)

slowest

$\log \log(n)$

$\log^2(n)$

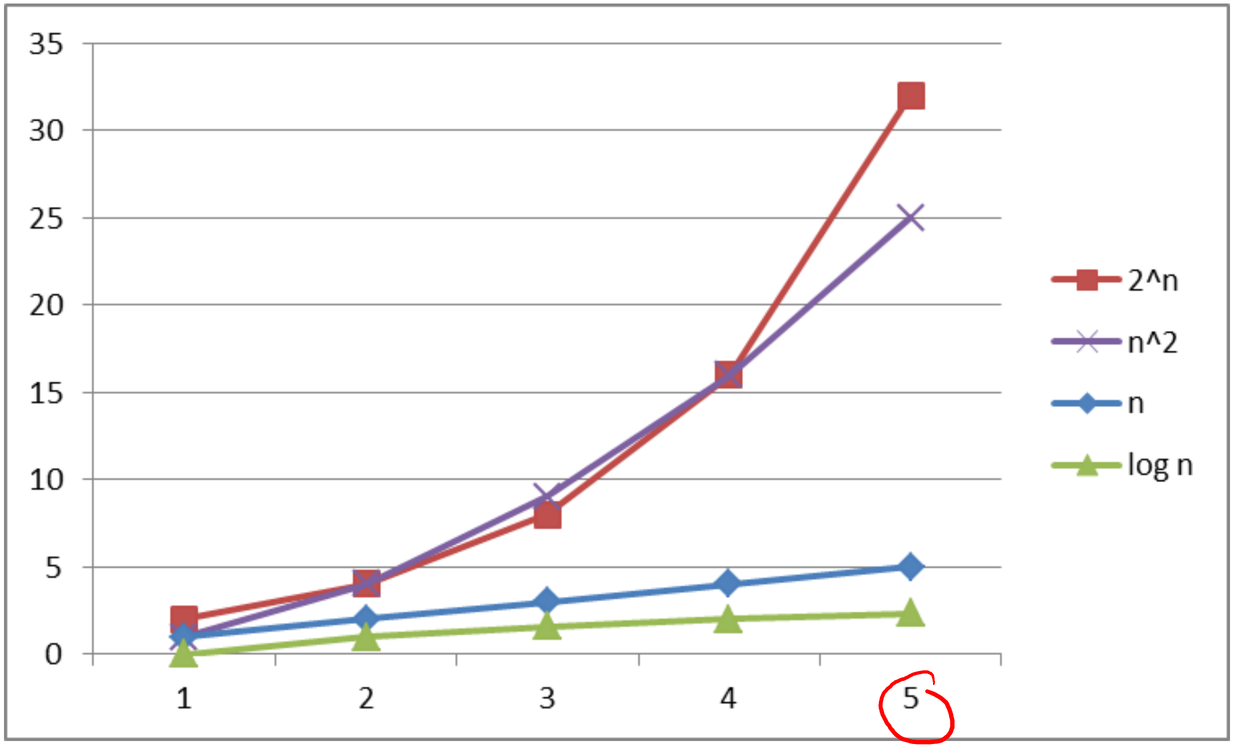
$\log^c(n)$

2^n

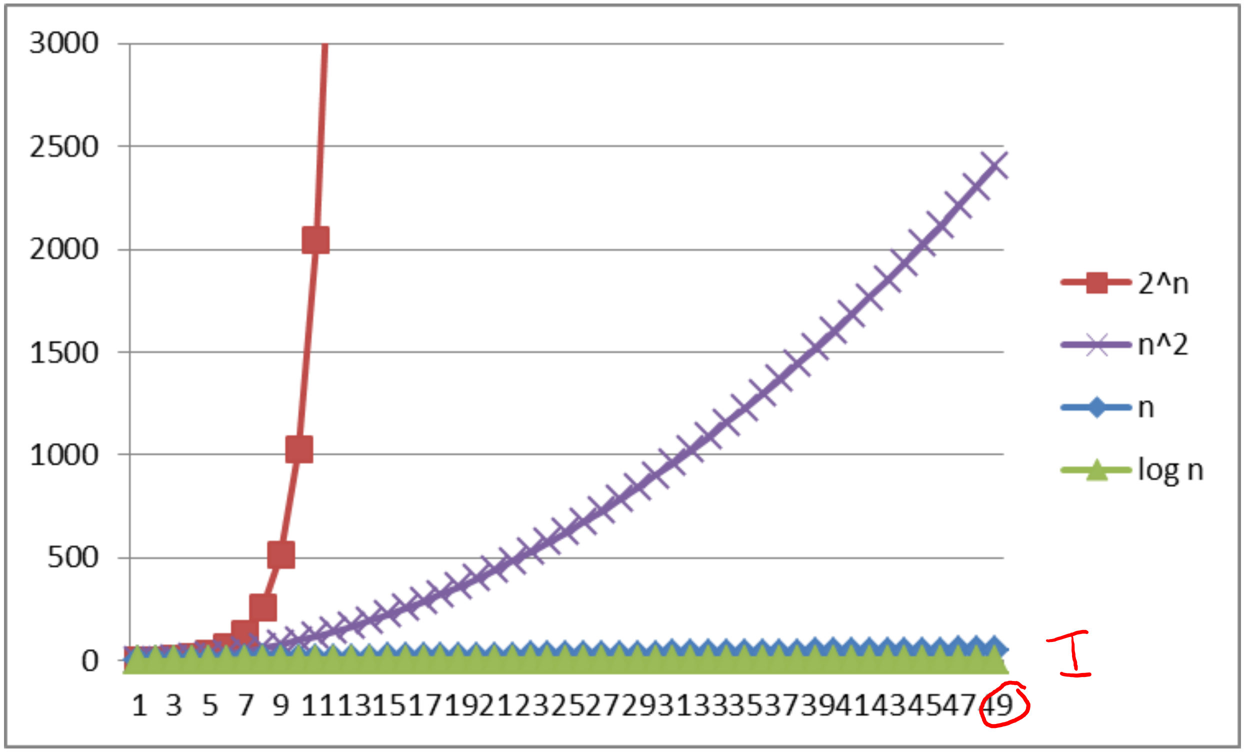
Note: *Don't write* $O(5n)$ instead of $O(n)$ – same thing! It's like writing 6/2 instead of 3. Looks weird

Usage note: "exponential" does not mean "grows really fast", it means "grows at rate proportional to k^n for some k>1"

Big Oh: Common Categories



Big Oh: Common Categories



Poll Everywhere pollev.com/artliu

True or false?

1. `4+3n` is in `O(n)` True $c=7$ $n_0=1$
2. `n+2logn` is in `O(logn)` False
3. `logn+2` is in `O(1)` False
4. n^{50} is in $O(1.1^n)$ True

Notes:

- Do NOT ignore constants that are not multipliers:
  - n^3 is $O(n^2)$: FALSE
  - 3^n is $O(2^n)$: FALSE
- When in doubt, refer to the definition

More asymptotic analysis

Upper bound: $O(\ f(n)\ )$

g(n) is in $O(\ f(n)\ )$ if there exist constants c and n_0 such that

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

Lower bound: $\Omega(\ f(n)\ )$

g(n) is in $\Omega(\ f(n)\ )$ if there exist constants c and n_0 such that

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

Tight bound: $\theta(\ f(n)\ )$

g(n) is in $\theta(\ f(n)\ )$ if it is in $O(\ f(n)\ )$ and it is in $\Omega(\ f(n)\ )$

Regarding use of terms

A common error is to say $O(\ f(n)\ )$ when you mean $\theta(\ f(n)\ )$

- People often say O() to mean a tight bound
- Say we have f(n)=n; we could say f(n) is in O(n), which is true, but only conveys the upper-bound
- Since f(n)=n is *also* $O(n^5)$, it's tempting to say "this algorithm is *exactly* $O(n)$"
- Somewhat incomplete; instead say it is $\theta(n)$
- That means that it is not, for example $O($**`log`** $n)$

$\Theta(N)$ $O(N)$ $\Omega(N)$

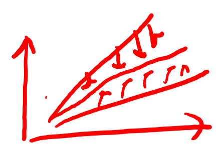
Less common notation:

- "little-oh": like "big-Oh" but strictly less than
  - Example: sum is $o(n^2)$ but not $o(n)$
- "little-omega": like "big-Omega" but strictly greater than
  - Example: sum is $\omega($**`log`** $n)$ but not $\omega(n)$

$\leq$ $<$ small θ $\boxed{o(N) \quad \omega(N)}$

$\geq$ $>$

~~little theta?~~



$=$ $>$ $<$

(Previously) Formally ***<u>Big-Oh</u>***

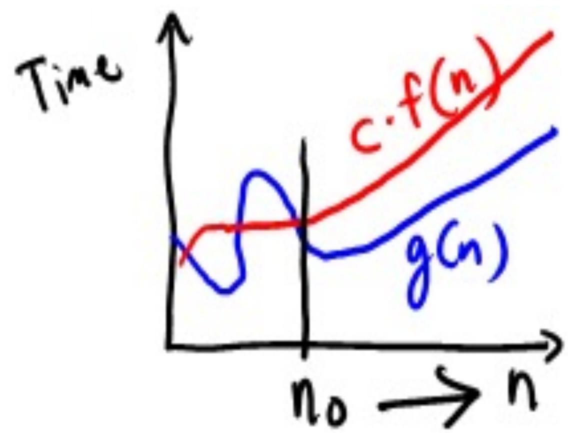
Definition: g(n) is in O(**f(n)**) iff there exist positive constants **c** and n_0 such that

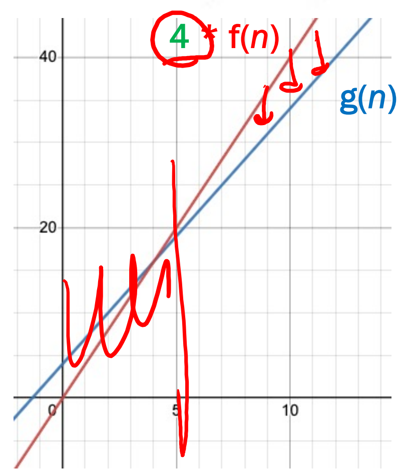
$$g(n) \leq c\ f(n) \quad \text{for all } n \geq n_0$$

Note: $n_0 \geq 1$ (and a natural number) and $c > 0$

Example: Let **g(n)** = 3n + 4 and **f(n)** = n

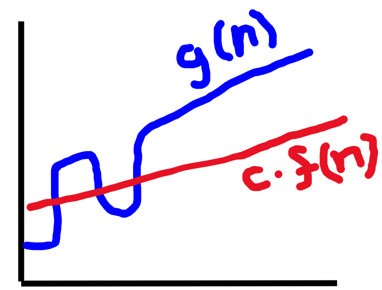
c = 4 and n_0 = 5 is one possibility





(NOW) Formally ***Big-Omega***

Now red is on bottom!



Definition: g(n) is in Ω(f(n)) iff there exist positive constants c and n_0 such that

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

Note: $n_0 \geq 1$ (and a natural number) and $c > 0$

Example: Let g(n) = 3n + 4 and f(n) = n

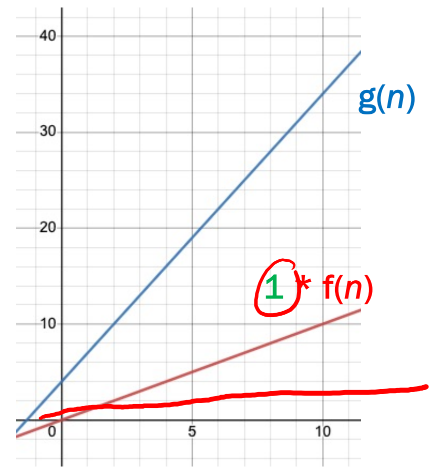
~~c = 1~~ and ~~n_0 = 1~~ is one possibility

$c = 3$ $n_0 = 1$

$c = 1$ $n_0 = 1$

$3n + 4 \geq 3n$

$c = 0.5$



Formally *Big-Theta*

if & only if

Definition: g(n) is in θ(f(n)) iff g(n) is in
O(f(n)) and it is in Ω(f(n))
Equivalently: iff there exist positive constants
c and n_0 such that
c_1 f(n) ≤ g(n) ≤ c_2 f(n) for all $n \geq n_0$

O Ω

n_0 n_0

max(n_0, n_0)

larger c

O(*n*) g(*n*) = 3*n* + 4

Ω(n)

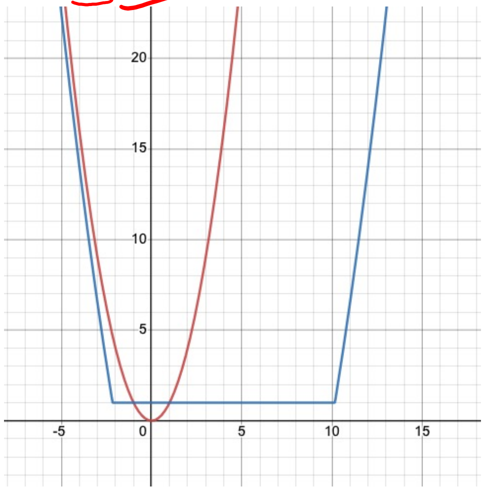
Notice: c (and n_0) constants can be different for proving Oh and Omega

Another example of Big Omega

Can we pick c >= 0.5?

- https://www.desmos.com/calculator/kmlqfe0lie

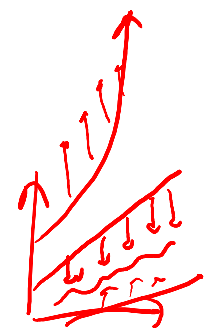
$f(n) = n^2$ $g(n) = \max(1, 0.5n^2 - 4n - 10)$

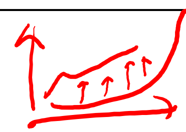


Big-O, Big-Theta, Big-Omega Relationships

If a function is in Big-Theta, what does it mean for its membership in Big-O and Big-Omega? Vice versa?

Mystery Function	Big-O	Big-Theta	Big-Omega
—	$O(N^4)$	$\Theta(N^4)$	$\Omega(N^4)$
	$O(N^3)$	$\Theta(N^3)$	$\Omega(N^3)$
	$O(N)$	??	?? but it must be equal to or less than $O(N)$
	must be equal to or higher than N^2	??	$\Omega(N^2)$





Big-O, Big-Theta, Big-Omega Relationships

If a function is in Big-Theta, what does it mean for its membership in Big-O and Big-Omega? Vice versa?

Mystery Function	Big-O	Big-Theta	Big-Omega
	$O(N^4)$	$\Theta(N^4)$	$\Omega(N^4)$
	$O(N^3)$	$\Theta(N^3)$	$\Omega(N^3)$
	$O(N)$	??	??, but cannot be $\Omega(N^2)$
	??, but cannot be $O(N)$	??	$\Omega(N^2)$

Theta, Oh, Omega != Worst-Case, Best-Case

≤ ≥

- These are independent!
  - We can analyze for both best-case and worst-case for all three

Example: what is the asymptotic analysis for Omega and Theta

Find an integer in a *sorted* array

```
// requires array is sorted
// returns whether k is in array
boolean find(int[]arr, int k){
  for(int i=0; i < arr.length; ++i)
    if(arr[i] == k)
      return true;
  return false;
}
```

Best case: 6 "ish" operations = O(1)

find(first) $\Omega(1)$ $\Theta(1)$

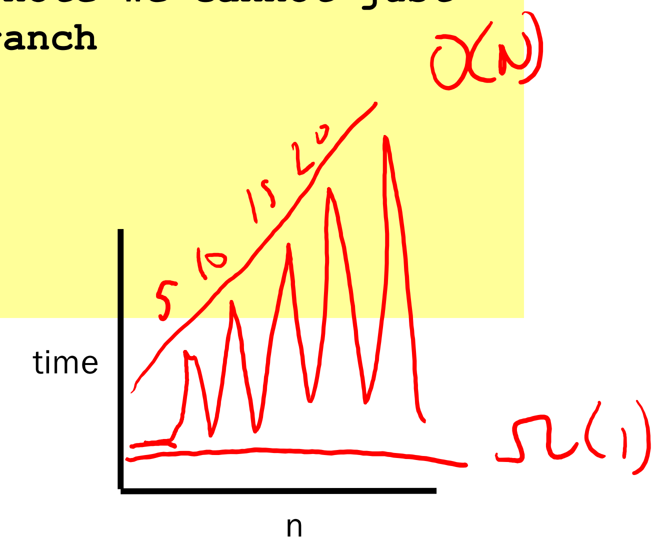
Worst case: 5 "ish" * (arr.length) = O(N)

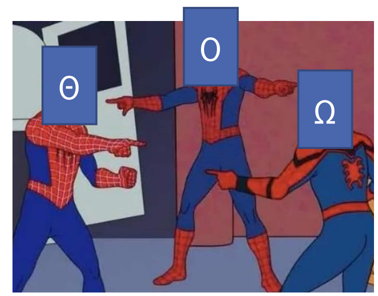
find(not in array) $\Omega(N)$ $\Theta(N)$

When are Oh and Omega (and theta) different?

- When doing worst-case analysis on code, ***tight*** Oh and ***tight*** Omega are *often* the same. When are they different?

```
// toggle function, note we cannot just
// analyze slower branch
if (n % 5 == 0) {
  // linear work
} else {
  // constant work
}
```





Today

- Big-Oh Definition
- Proofs
- Amortization

Proof Mistake: Backwards Reasoning

- Careful to not assume something is true and work backwards
  - Backwards reasoning only shows “consistency” NOT “truth”
- We need to start with something true and work towards the statement that we want to show is true
  - ***The last statement should be our claim!***

Example | Claim: Show that $4 \geq 5$

$4 \geq 5$

$4 \cdot 0 \geq 5 \cdot 0$ multiply by 0

$\boxed{0 \geq 0}$ ✓ tada

$0 \geq 0$ true

$0/0 \geq 0/0$ divide by 0 ??

$4 \geq 5$

Big-Oh Proofs

There is likely some “scratch work” – the insight isn’t explained in the final proof

- You just say “consider”

But don’t try to skip the scratch work when drafting your big-O proofs!

- But it won’t necessarily appear in your final version

Example

Let's show: `10n`^{2} `+ 15n is in O(n`^{2}`)`

Scratch work:

$10n^2 + 15n$

$25n^2$ $c = 25$

$n_0 = 1$

Let $c = 25$, $n_0 = 1$

$$10n^2 + 15n \leq 10n^2 + 15n^2 \quad \text{since } n \geq 1$$
$$\leq 25n^2$$
$$\leq c\,n^2$$

which is exactly what the def of Big Oh is

Example

Let's show: $10n^2 + 15n$ is in $O(n^2)$

Proof:

Let $c = 25$ and $n_0 = 1$

$10n^2 + 15n \le 10n^2 + 15n^2$ since $n \ge 1$

$\le 25n^2$ addition

Which is exactly what we wanted to show from the definition of Big-Oh.

COUNTER Example

Let's show: $10n^2 + 15n$ is in $O(n^2)$

BAD:

$$10n^2 + 15n \le 25n^2$$

$$15n \le 15n^2 \qquad \text{subtract } 10n^2$$

$$n \le n^2 \qquad \text{divide by 15}$$

$$1 \le n \qquad \text{divide by n, tada (WRONG)}$$

So, we can choose c = 25 and n_0 = 1 and thus we have shown it is in $O(n^2)$

AGAIN, START WITH TRUE STATEMENT and END AT OUR GOAL CLAIM!

Proving NOT in Big Oh

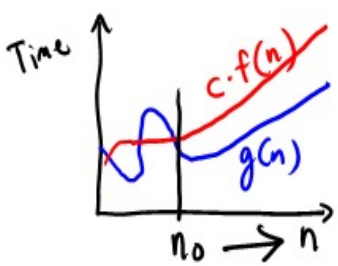
- Prove the negation is true
- Prove by contradiction
    - Assume true, prove impossible

Negating Big-Oh

Definition: g(n) is in O(f(n)) iff there exist positive constants c and n_0 such that

g(n) ≤ c f(n) for all $n \geq n_0$

$$\exists_{c,n_0} \forall_{n \geq n_0} \; g(n) \leq c \cdot f(n)$$

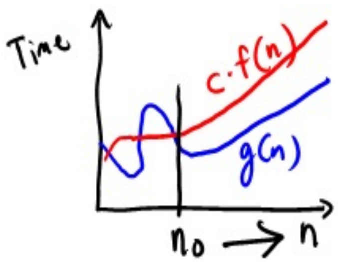


Negating Big-Oh

Definition: $g(n)$ is in $O(f(n))$ iff there exist positive constants c and n_0 such that

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

$$\exists_{c,n_0} \forall_{n \geq n_0} \; g(n) \leq c \cdot f(n)$$



$$\forall_{c,n_0} \exists_{n \geq n_0} \; g(n) > c \cdot f(n)$$

For any c or n_0 that you pick, there is a valid n where our function g(n) exceeds the Big Oh function f(n)

Proving NOT in Big Oh Example: Prove Negation

Let's show: `10n`^{2} `is NOT O(n)`

Not in Big OH:

$$\forall_{c,n_0} \exists_{n \geq n_0} \ g(n) > c \cdot f(n)$$

For any c or n_0 that you pick, there is a valid n where our function g(n) exceeds the Big Oh function f(n)

Proving NOT in Big Oh Example: Prove Negation

Let's show: $10n^2$ is NOT O(n)

Scratch work:

Need to find an n

$10n^2 > cn$

$10n > c$

$n > c/10$

Not in Big OH:

$$\forall_{c,n_0} \exists_{n \geq n_0} \; g(n) > c \cdot f(n)$$

For any c or n_0 that you pick, there is a valid n where our function g(n) exceeds the Big Oh function f(n)

Proof:

Let n = max(c/10+1, n0)

```
 n   > c/10    from def
10n > c        math
10n² > cn      multiply both sides by n (positive)
```

Which is exactly the inequality that we wanted to show. Since c and n0 are arbitrary, we have shown this for all c and n0 and shown that it is not in Big Oh of O(n).

Proving NOT in Big Oh Example: Contradiction

Show: $10n^2$ is NOT $O(n)$

For sake of contradiction, assume that $10n^2$ is O(n)

$10n^2 \le cn$ for some c and all $n \ge n_0$ *(definition)*

$10n \le c$ divide by n (positive)

$n \le c/10$ divide by 10

But this is true for all $n \ge n_0$ so $n = \max(c/10+1, n0)$ contradicts the last statement. Since c and n0 are arbitrary, and by contradiction we have shown that $10n^2$ is NOT $O(n)$

Today

- Big-Oh Definition
- Proofs
- Amortization

Amortization

How much does housing cost per day in Seattle?
Well, it depends on the day.

The day rent is due, it’s $1200.
Other days of the month, it’s free.

Amortization

Amortization is an **accounting trick.** It's a way to reflect the fact that the "first of the month" is really responsible for the other days of the week, and each day should be assigned it's "fair share."

Amortization

Amortized:

It costs $1200/month, and we pay one day of the 30 in a month

Cost per day is 1200/30 = 40

“What does my daily pay need to be to afford housing?”

Un-amortized:

On the first it costs $1200

Every other day, it costs $0

“How much do I need to keep in my bank account, so it doesn’t get overdrawn?”

Array Insertion Example

What's the worst case for insert into an array-based queue?

- O(n) when we need to resize, O(1) otherwise

Is O(n) a good description of the worst-case behavior?

Amortization

Amortized:

It takes O(n) time to resize once, after n-1 calls that take O(1) time

Cost per operation is

$$\frac{O(n)+[n-1]O(1)}{n} = O(1)$$

“What will happen when I do many insertions in a row?”

Un-amortized:

The resize takes O(n) time. That’s the worst case that could happen.

”How long might one (unlucky) user need to wait on a single insertion?”

Why double size?

The most common strategy for increasing array size is doubling. Why not just increase the size by 10,000 each time we fill up?

Let's say we did n insertions:

Costs of the unlucky insertions:

Costs of the other insertions:

Amortized insert cost:

Why double size?

The most common strategy for increasing array size is doubling. Why not just increase the size by 10,000 each time we fill up?

Let's say we did n insertions:

Costs of the unlucky insertions:

$$\sum_{i=0}^{n/10{,}000} 10{,}000i \approx 10{,}000 \cdot \frac{n^2}{10{,}000^2} = O(n^2)$$

Costs of the other insertions:

O(1) * n = O(n)

Amortized insert cost:

$$O\left(\frac{n^2+n}{n}\right) = O(n)$$

Way worse than O(1) with doubling!

Notes on Amortization

- Depends on the question you are asking
- Can customize your “rent pay-day” algorithm to the use-case
  - Pay all on one day?
  - Pay in 3 easy payments of 99.99?
- See Weiss chapter 11

What we are (often) analyzing in 332

- The most common thing to do is give an O or θ bound to the worst-case running time of an algorithm

- Example: True statements about binary-search algorithm
  - Common: $\theta($**`log`** $n)$ running-time in the worst-case
  - Less common: $\theta(1)$ in the best-case (item is in the middle)
  - Less common: Algorithm is $\Omega($**`log log`** $n)$ in the worst-case (it is not really, really, really fast asymptotically)
  - Less common (but very good to know): the find-in-sorted-array ***problem*** is $\Omega($**`log`** $n)$ in the worst-case
    - *No* algorithm can do better (without parallelism)

Problem vs. Algorithm Analysis

A ***problem*** cannot be $O(f(n))$ since you can always find a slower algorithm, so instead you can say ***there exists*** an algorithm that solves the problem in O(f(n))

A ***problem*** ***can*** be Ω(f(n)) which means that we cannot find an algorithm that solves the problem any faster!

Other things to analyze

- Space instead of time
  - Remember we can often use space to gain time
- Average case
  - Sometimes only if you assume something about the distribution of inputs
    - See CSE312 and STAT391
  - Sometimes uses randomization in the algorithm
    - Will see an example with sorting; also see CSE312
- Sometimes an *amortized guarantee*

Poll Everywhere pollev.com/artliu

Sample Exam Questions:

Assume domain and co-domain of all functions are the natural numbers (1, 2, 3...)

Decide: Always True Sometimes True Never True

1. $f(n)$ is in $O(f(n)^2)$
2. $f(n)$ is in $\Theta(f(n))$
3. $f(n)+g(n)$ is in $\Theta(\max(f(n),\ g(n)))$
4. $f(n) * n$ is in $O(f(n)^2)$

1. `f(n)` is in $O(f(n)^2)$ AT ST NT

2. `f(n)` is in $\Theta(f(n))$ AT ST NT

3. `f(n)+g(n)` is in $\Theta(\max(f(n), g(n)))$ AT ST NT

4. `f(n) * n` is in $O(f(n)^2)$ AT ST NT

Summary

Analysis can be about:

- The problem or the algorithm (usually algorithm)
- Time or space (usually time)
  - Or power or dollars or …
- Best-, worst-, or average-case (usually worst)
- Upper-, lower-, or tight-bound (usually upper or tight)
- Amortized or un-amortized

Big-Oh Caveats

- Asymptotic complexity (Big-Oh) focuses on behavior for **large *n*** and is independent of any computer / coding trick
  - But you can "abuse" it to be misled about trade-offs
  - Example: $n^{1/10}$ vs. `log` n
    - Asymptotically $n^{1/10}$ grows more quickly
    - But the "cross-over" point is around $5 * 10^{17}$
    - So if you have input size less than 2^{58}, prefer $n^{1/10}$
- Comparing O() for ***small n*** values can be misleading
  - Quicksort: O(nlogn) (expected)
  - Insertion Sort: $O(n^2)$ (expected)
  - Yet in reality Insertion Sort is faster for small n's
  - We'll learn about these sorts later

Addendum: Timing vs. Big-Oh?

- At the core of CS is a backbone of theory & mathematics
  - Examine the algorithm itself, mathematically, not the implementation
  - Reason about performance as a function of n
  - Be able to mathematically prove things about performance
- Yet, timing has its place
  - In the real world, we do want to know whether implementation A runs faster than implementation B on data set C
  - Ex: Benchmarking graphics cards
- Evaluating an algorithm? Use asymptotic analysis
- Evaluating an implementation of hardware/software? Timing can be useful