Algorithm Analysis

Data Structures and Algorithms

Logistics

Did you get the course list email?

- Went out yesterday, a reminder about the partnership survey and p1 spec is out

Section tomorrow

- Meet your TA
- Chance to practice problems about what we learned in lecture.
- Occasionally learn new material
- This week: Talk about concepts core to your first programming project.

Project 1 repos out tonight. Spec already out. Survey due by 3:00pm today

If you don't have a partner yet, meet up here at the end of class, and we'll pair you up.

Algorithm Analysis

I have some programming problem I need solved.

Siva and Alon have different ideas for how to solve the problem. How do we know which is better?

Easy. Have them both write the code and run it, and see which is faster.

THIS IS A TERRIBLE IDEA

How can we analyze their suggestions *before* they have to write the code, and in a way that is independent of their machines?

Comparing Algorithms

We want to know when one algorithm will be better than another

- What does "better" mean?

We really care about large inputs.

- If n=15, any algorithm will probably finish in less than a second anyway…

Want our answer to be independent of computer speed or programming language.

And we want an answer that's mathematically rigorous.

Analyzing Code

Assume basic operations take the same constant amount of time.

What's a basic operation?

- Adding ints or doubles
- Assignment
- Incrementing a variable
- A return statement
- Accessing an array index or an object field

This is a lie, but it's a useful lie.

Analyzing Code

What's not a basic operation?

Consecutive statements	Sum of time of each statement
Loops	Num iterations * time for loop body
Conditionals	Time of condition plus time of slower branch
Function Calls	Time of function's body
Recursion	Solve recurrence equation

What Are We Counting?

Worst case analysis

- For a given input size, what's the running time for the worst state our data structure we can be in or the worst input we can give?

Best case analysis

- What is the number of steps for the best state of our structure and the best question?

Average case analysis

- How are we doing on average over all possible inputs/states of our data structure?
- Have to ask this question very carefully to get a meaningful answer
- Nice to have, but hard to calculate

We usually do worst case analysis. Why?

Example

Linear search

```
int linearSearch(int[] A, int target){
   for(int i = 0; i < A.length; i++){
      if(A[i] == target)
         return i;
   }
   return -1;
}
```

What is the best case number of simple operations for this piece of code?
Let `A` have n entries.

What is the worst case number of simple operations for this piece of code?
Let `A` have n entries.

More Examples

```
 b = b + 5
 c = b / a
 b = c + 100
```

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

```
if (j < 5) {
  sum++;
} else {
 for (i = 0; i < n; i++) {
     sum++;
 }
}
```

One More Example

```
int coolFunction(int n, int sum) {
    int i, j;
    for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++) {
        sum++;
      }
    }
    print "This program is great!"
    for (i = 0; i < n; i++) {
       sum++;
    }
    return sum
}
```

Asymptotic Notation

Does everything in an estimate matter?

Multiplying by constant factors doesn't matter – let's just ignore them.

Lower order terms don't matter – delete them.

Gives us a "simplified big-O"

$10n \log n + 3n$	$O(n \log n)$
$5n^2 \log n + 13n^3$	$O(n^3)$
$20n \log \log n + 2\, n \log n$	$O(n \log n)$
2^{3n}	$O(8^n)$

Formally Big-O

We wanted to find an upper bound on our algorithm's running time, but

- We don't want to care about constant factors.
- We only care about what happens as n gets large.

The formal, mathematical definition is Big-O.

Big-O

$f(n)$ is $O(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$,

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

We also say that $g(n)$ "dominates" $f(n)$.

Why is that the definition?

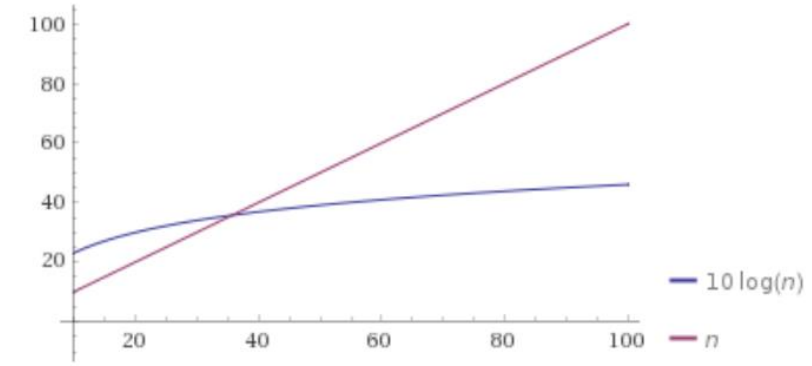
Big-O

$f(n)$ is $O(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$,

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

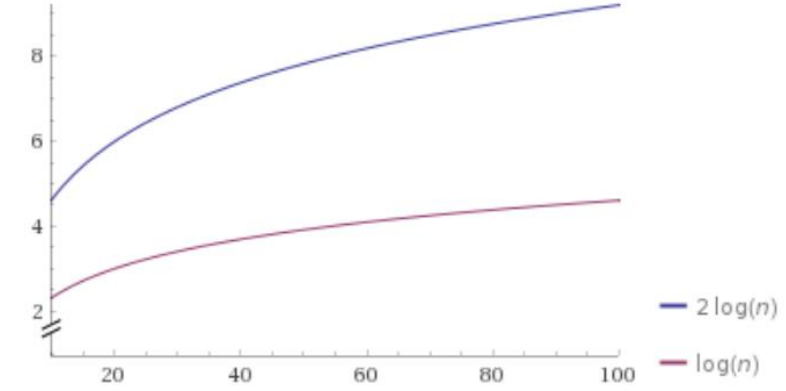
Why n_0?

Plot:



Why c?

Plot:



Why Are We Doing This?

You already intuitively understand what big-O means.

Who needs a formal definition anyway?

- We will.

Your intuitive definition and my intuitive definition might be different.

We're going to be making more subtle big-O statements in this class.

- We need a mathematical definition to be sure we're on the same page.

Once we have a mathematical definition, we can go back to intuitive thinking.

- But when a weird edge case, or subtle statement appears, we can figure out what's correct.

Edge Cases

True or False: $10n^2 + 15n$ is $O(n^3)$

It's true – it fits the definition.

Big-O is just an upper bound. It doesn't have to be a *good* upper bound.

If we want the best upper bound, we'll ask you for a **tight** big-O bound.

$O(n^2)$ is the tight bound for this example.

It is (almost always) technically correct to say your code runs in time $O(n!)$.

- DO NOT TRY TO PULL THIS TRICK ON AN EXAM. Or in an interview.

Another Edge Case

Algorithm A has n/10000 operations and is O(n)

Algorithm B has 2^{100} operations and is O(1)

B has the better asymptotic behavior and thus is clearly the better choice. *Is it though?*

Asymptotic behavior only guarantees something as n gets infinitely large. This is an extreme example, but if you aren't using an infinite amount of data as input, O notation may not dictate the best choice.

O, Omega, Theta

Big-O is an **upper bound**
- My code takes at most this long to run

Big-Omega is a lower bound

Big-Omega

$f(n)$ is $\Omega(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$,
$f(n) \geq c \cdot g(n)$

Big Theta is "equal to"

Big-Theta

$f(n)$ is $\Theta(g(n))$ if
$f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

Viewing O as a class

Sometimes you'll see big-O defined as a family or set of functions.

Big-O (alternative definition)

$O(g(n))$ is the set of all functions $f(n)$ such that there exist positive constants c, n_0 such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$

For that reason, some people write $f(n) \in O(g(n))$ where we wrote "$f(n)$ is $O(g(n))$".
Other people write "$f(n) = O(g(n))$" to mean the same thing.
The set of all functions that run in linear time (i.e. $O(n)$) is a "complexity class."
We'll talk more about complexity classes much later in the quarter.
We never write $O(5n)$ instead of $O(n)$ – they're the same thing!

It's like writing $\frac{6}{2}$ instead of 3. It just looks weird.

Useful Vocab

The most common running times all have fancy names:

$O(1)$ constant

$O(\log n)$ logarithmic

$O(n)$ linear

$O(n \log n)$ n log n (well, ok, except for this one)

$O(n^2)$ quadratic

$O(n^3)$ cubic

$O(n^c)$ polynomial (where c is a constant)

$O(c^n)$ exponential (where c is a constant)

Log Tidbits

If I write $\log n$, without specifying a base, I mean $\log_2 n$.

But does it matter for big-O?

Suppose we found an algorithm with running time $\log_5 n$ instead.

Is that different from $O(\log_2 n)$?

No!

$\log_c n = \frac{\log_2 n}{\log_2 c}$ If c is a constant, then $\log_2 c$ is just a constant, and we can hide it inside the $O()$.

Log Tidbits

log(log x) is written log log x

Grows as slowly as $2^{(2\text{^}y)}$ grows fast

Ex:

$$\log_2 \log_2 4billion \sim \log_2 \log_2 2^{32} = \log_2 32 = 5$$

(log x)(log x) is written $\log^2 x$

It is greater than log x for all x > 2

constant < log log x < log x < $\log^2 x$ < x

Writing Proofs

Claim: For every odd integer y, there exists an even integer x, such that $x > y$.

Proof:

Let y be an arbitrary odd integer. By definition, $y = 2z + 1$ for some integer z.

Consider $\mathrm{x} = \ 2(z + 1)$.

x is even (since it can be written as 2 times some integer) and

$x = 2(z + 1) = 2z + 2 > 2z + 1 = y$. □

Writing Proofs

Where did that $x = 2(z + 1)$ come from?

You probably came up with that even integer first, before you started writing the proof.

That was some "scratch work" – the insight isn't explained in the final proof

- You just say "Consider"

For this class, when you're writing big-O proofs, include both your scratch work and the final proof.

- If you make a mistake, it's much easier for us to diagnose.
- Which makes it easier for you to get partial credit.

Using the Definition

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

Scratch work:
$10n^2 \leq 10n^2$
$15n \leq 15n^2$ for $n \geq 1$
$10n^2 + 15n \leq 25n^2$ for $n \geq 1$

Proof:
Take $c = 25$ and $n_0 = 1$.
The inequality $10n^2 \leq 10n^2$ is always true. The inequality $15n \leq 15n^2$ is true for $n \geq 1$, as the right hand side is a factor of n more than the right hand side.
As long as both inequalities are true we can add them, thus
$10n^2 + 15n \leq 25n^2$ holds as long as $n \geq 1$.
This is exactly the inequality we needed to show.