Lecture 3

Big-O notation, more recurrences!!
Announcements!

• **HW1 is posted!** (Due Friday)

• See Piazza for a list of HW clarifications

• First recitation section was this morning, there’s another tomorrow (same material). *(These are optional, it’s a chance for TAs to go over more examples than we can get to in class).*
FAQ

• How rigorous do I need to be on my homework?
  • See our example HW solution online
  • In general, we are shooting for:
    
    You should be able to give a friend your solution 
    and they should be able to turn it into a rigorous proof 
    without much thought.
  
  • This is a delicate line to walk, and there’s no easy answer. Think of it like more like writing a good essay than “correctly” solving a math problem.

• What’s with the array bounds in pseudocode?
  • SORRY! I’m trying to match CLRS and this causes me to make mistakes sometimes. In this class, I’m trying to do:
    • Arrays are 1-indexed
    • A[1..n] is all entries between 1 and n, inclusive
    • I will also use A[1:n] (python notation) to mean the same thing (not python notation).
  
  • Please call me out when I mess up.
Last time....

- Sorting: InsertionSort and MergeSort
- Analyzing correctness of iterative + recursive algs
  - Via “loop invariant” and induction
- Analyzing running time of recursive algorithms
  - By writing out a tree and adding up all the work done.
Today

• How do we measure the runtime of an algorithm?
  • Worst-case analysis
  • Asymptotic Analysis

• Recurrence relations:
  • Integer Multiplication and MergeSort again

• The “Master Method” for solving recurrences.
Recall from last time...

- We analyzed **INSERTION SORT** and **MERGESORT**.
- They were both correct!
- **INSERTION SORT** took time about \( n^2 \)
- **MERGESORT** took time about \( n \log(n) \).

\( n \log(n) \) is way better!!!
A few reasons to be grumpy

• Sorting

should take zero steps...why nlog(n)??

• What’s with this $T(MERGE) < 2 + 4n \leq 6n$?
Analysis

\[ T(n) = \text{time to run MERGESORT on a list of size } n \]

This is called a recurrence relation: it describes the running time of a problem of size \( n \) in terms of the running time of smaller problems.

\[ T(n) = T(n/2) + T(n/2) + T(\text{MERGE}) = 2T(n/2) + 6n \]

T(\text{MERGE two lists of size } n/2) is the time to do:

- 3 variable assignments (counters \( \leftarrow 1 \))
- \( n \) comparisons
- \( n \) more assignments
- 2n counter increments

So that’s

\[ 2T(\text{assign}) + n T(\text{compare}) + n T(\text{assign}) + 2n T(\text{increment}) \]

or \( 4n + 2 \) operations

Or \( 4n + 3 \)...

We will see later how to analyse recurrence relations like these automagically...but today we’ll do it from first principles.
A few reasons to be grumpy

• Sorting

should take zero steps...why nlog(n)??

• What’s with this \( T(\text{MERGE}) < 2 + 4n \leq 6n \)?
  • The “2 + 4n” operations thing doesn’t even make sense. Different operations take different amounts of time!
  • We bounded 2 + 4n \( \leq 6n \). I guess that’s true, but that seems pretty dumb.
How we will deal with grumpiness

• Take a deep breath...
• Worst case analysis
• Asymptotic notation
Worst-case analysis

• In this class, we will focus on worst-case analysis

Pros: very strong guarantee
Cons: very strong guarantee

Algorithm designer

Algorithm:
Do the thing
Do the stuff
Return the answer

Here is my algorithm!

Sorting a sorted list should be fast!!
Big-O notation

• What do we mean when we measure runtime?
  • We probably care about wall time: how long does it take to solve the problem, in seconds or minutes or hours?

• This is heavily dependent on the programming language, architecture, etc.

• These things are very important, but are not the point of this class.

• We want a way to talk about the running time of an algorithm, independent of these considerations.
Remember this slide?

<table>
<thead>
<tr>
<th>n</th>
<th>n log(n)</th>
<th>n^2</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>24</td>
<td>64</td>
</tr>
<tr>
<td>16</td>
<td>64</td>
<td>256</td>
</tr>
<tr>
<td>32</td>
<td>160</td>
<td>1024</td>
</tr>
<tr>
<td>64</td>
<td>384</td>
<td>4096</td>
</tr>
<tr>
<td>128</td>
<td>896</td>
<td>16384</td>
</tr>
<tr>
<td>256</td>
<td>2048</td>
<td>65536</td>
</tr>
<tr>
<td>512</td>
<td>4608</td>
<td>262144</td>
</tr>
<tr>
<td>1024</td>
<td>10240</td>
<td>1048576</td>
</tr>
</tbody>
</table>
Change $n \log(n)$ to $5n \log(n)$...

<table>
<thead>
<tr>
<th>$n$</th>
<th>$5n \log(n)$</th>
<th>$n^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>120</td>
<td>64</td>
</tr>
<tr>
<td>16</td>
<td>320</td>
<td>256</td>
</tr>
<tr>
<td>32</td>
<td>800</td>
<td>1024</td>
</tr>
<tr>
<td>64</td>
<td>1920</td>
<td>4096</td>
</tr>
<tr>
<td>128</td>
<td>4480</td>
<td>16384</td>
</tr>
<tr>
<td>256</td>
<td>10240</td>
<td>65536</td>
</tr>
<tr>
<td>512</td>
<td>23040</td>
<td>262144</td>
</tr>
<tr>
<td>1024</td>
<td>51200</td>
<td>1048576</td>
</tr>
</tbody>
</table>

As $n$ gets large, I’d even take runtime 100 $n \log(n)$ over $n^2$...
Asymptotic Analysis
How does the running time scale as \( n \) gets large?

One algorithm is “faster” than another if its runtime grows more “slowly” as \( n \) gets large.

Pros:
- Abstracts away from hardware- and language-specific issues.
- Makes algorithm analysis much more tractable.

Cons:
- Only makes sense if \( n \) is large (compared to the constant factors).

This will provide a formal way of saying that \( n^2 \) is “worse” than 100 \( n \log(n) \).

This is especially relevant now, as data get bigger and bigger and bigger...

\[ 2^{1000000000000000} \text{ is “better” than } n^2?!?! \]
Now for some definitions...

• Quick reminders:
  • ∃: “There exists”
  • ∀: ”For all”
  • Example: ∀ students in CS161, ∃ an algorithms problem that really excites the student.
  • Much stronger statement: ∃ an algorithms problem so that, ∀ students in CS161, the student is excited by the problem.

• We’re going to formally define an upper bound:
  • “T(n) grows no faster than f(n)”
O(...) means an upper bound

• Let T(n), f(n) be functions of positive integers.
  • Think of T(n) as being a runtime: positive and increasing in n.
• We say "T(n) is O(f(n))" if f(n) grows at least as fast as T(n) as n gets large.

• Formally,

\[
T(n) = O(f(n)) \iff \exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0, \\
0 \leq T(n) \leq c \cdot f(n)
\]

pronounced “big-oh of ...” or sometimes “oh of ...”
$T(n) = O(f(n))$

$\Leftrightarrow$

$\exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0, \\
0 \leq T(n) \leq c \cdot f(n)$

$T(n) = O(f(n))$ means:
Eventually, (for large enough $n$) something that grows like $f(n)$ is always bigger than $T(n)$. 
Example 1

- $T(n) = n$, $f(n) = n^2$.
- $T(n) = O(f(n))$

\[ T(n) = O(f(n)) \iff \exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0, \quad 0 \leq T(n) \leq c \cdot f(n) \]
Examples 2 and 3

• All degree k polynomials with positive leading coefficients are $O(n^k)$.
• For any $k \geq 1$, $n^k$ is not $O(n^{k-1})$. 

(On the board)
Take-away from examples

• To prove $T(n) = O(f(n))$, you have to come up with $c$ and $n_0$ so that the definition is satisfied.

• To prove $T(n)$ is NOT $O(f(n))$, one way is by contradiction:
  • Suppose that someone gives you a $c$ and an $n_0$ so that the definition is satisfied.
  • Show that this someone must by lying to you by deriving a contradiction.
\( O(...) \) means an upper bound, and 
\( \Omega(...) \) means a lower bound

- We say “\( T(n) \) is \( \Omega(f(n)) \)” if \( f(n) \) grows at most as fast as \( T(n) \) as \( n \) gets large.

- Formally,

\[
T(n) = \Omega(f(n)) \\
\iff \\
\exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0, \\
0 \leq c \cdot f(n) \leq T(n)
\]

Switched these!!
Parsing that...

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

$$\iff$$

$$\exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0, 0 \leq c \cdot f(n) \leq T(n)$$
Θ(...) means both!

• We say “T(n) is Θ(f(n))” if:

\[ T(n) = O(f(n)) \]

-AND-

\[ T(n) = \Omega(f(n)) \]
Yet more examples

- $n^3 - n^2 + 3n = O(n^3)$
- $n^3 - n^2 + 3n = \Omega(n^3)$
- $n^3 - n^2 + 3n = \Theta(n^3)$
- $3^n$ is not $O(2^n)$
- $n \log(n) = \Omega(n)$
- $n \log(n)$ is not $\Theta(n)$.

Fun exercise: check all of these carefully!!
We’ll be using lots of asymptotic notation from here on out

• This makes both Plucky and Lucky happy.
  • Plucky the Pedantic Penguin is happy because there is a precise definition.
  • Lucky the Lackadaisical Lemur is happy because we don’t have to pay close attention to all those pesky constant factors like “4” or “6”.
• But we should always be careful not to abuse it.
• In the course, (almost) every algorithm we see will be actually practical, without needing to take \( n \geq n_0 = 2^{100000000} \).

Questions about asymptotic notation?
Back to recurrence relations

$T(n) = \text{time to solve a problem of size } n$.

We’ve seen three recursive algorithms so far.

• Needlessly recursive integer multiplication
  • $T(n) = 4 \cdot T(n/2) + O(n)$
  • $T(n) = O(n^2)$

• Karatsuba integer multiplication
  • $T(n) = 3 \cdot T(n/2) + O(n)$
  • $T(n) = O(n^{\log_2 3} \approx n^{1.6})$

• MergeSort
  • $T(n) = 2T(n/2) + O(n)$
  • $T(n) = O(n \log(n))$

What’s the pattern?!?!?!?
The master theorem

• A **formula** that solves recurrences when all of the sub-problems are the same size.

• (We’ll see an example Wednesday when not all problems are the same size).

A useful formula it is. Know why it works you should.
The master theorem

- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$. Then

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

Three parameters:
- $a$ : number of subproblems
- $b$ : factor by which input size shrinks
- $d$ : need to do $n^d$ work to create all the subproblems and combine their solutions.

Many symbols those are....
Examples
(details on board)

\[ T(n) = a \cdot T \left( \frac{n}{b} \right) + O(n^d). \]

\[ T(n) = \begin{cases} 
O(n^d \log(n)) & \text{if } a = b^d \\
O(n^d) & \text{if } a < b^d \\
O(n^{\log_b(a)}) & \text{if } a > b^d 
\end{cases} \]

• Needlessly recursive integer mult.
  \[ T(n) = 4 \cdot T(n/2) + O(n) \]
  \[ a = 4 \quad b = 2 \quad a > b^d \]
• \[ T(n) = O(n^2) \]
  \[ a = 3 \quad b = 2 \quad a > b^d \]

• Karatsuba integer multiplication
  \[ T(n) = 3 \cdot T(n/2) + O(n) \]
  \[ a = 3 \quad b = 2 \quad a > b^d \]
• \[ T(n) = O(n^{\log_2(3)} \approx n^{1.6}) \]
  \[ a = 3 \quad b = 2 \quad a > b^d \]

• MergeSort
  \[ T(n) = 2T(n/2) + O(n) \]
  \[ a = 2 \quad b = 2 \quad a = b^d \]
• \[ T(n) = O(n \log(n)) \]
  \[ a = 2 \quad b = 2 \quad a = b^d \]
Proof of the master theorem

• We’ll do the same recursion tree thing we did for **MergeSort**, but be more careful.

• Suppose that $T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n^d$.

Hang on! The hypothesis of the Master Theorem was the extra work at each level was $O(n^d)$. That’s NOT the same as work $\leq cn$ for some constant $c$.

That’s true ... we’ll actually prove a weaker statement that uses this hypothesis instead of the hypothesis that $T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$. It’s a good exercise try to make this proof work rigorously with the $O()$ notation.
Recursion tree

\[ T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n^d \]

<table>
<thead>
<tr>
<th>Level</th>
<th># problems</th>
<th>Size of each problem</th>
<th>Amount of work at this level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>n</td>
<td>( c \cdot n^d )</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>n/b</td>
<td>( ac \left(\frac{n}{b}\right)^d )</td>
</tr>
<tr>
<td>2</td>
<td>a^2</td>
<td>n/b^2</td>
<td>( a^2c \left(\frac{n}{b^2}\right)^d )</td>
</tr>
<tr>
<td>( t )</td>
<td>( a^t )</td>
<td>n/b^t</td>
<td>( a^t c \left(\frac{n}{b^t}\right)^d )</td>
</tr>
<tr>
<td>( \log_b(n) )</td>
<td>( a^{\log_b(n)} )</td>
<td>1</td>
<td>( a^{\log_b(n)} c )</td>
</tr>
</tbody>
</table>

(Size 1)
Recursion tree  

\[ T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n^d \]

<table>
<thead>
<tr>
<th>Level</th>
<th># problems</th>
<th>Size of each problem</th>
<th>Amount of work at this level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>n</td>
<td>(c \cdot n^d)</td>
</tr>
<tr>
<td>1</td>
<td>(a)</td>
<td>(n/b)</td>
<td>(ac \left(\frac{n}{b}\right)^d)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Total work (derivation on board) is at most:

\[
c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^t
\]
Now let’s check all the cases (on board)

\[ T(n) = \begin{cases} 
O(n^d \log(n)) & \text{if } a = b^d \\
O(n^d) & \text{if } a < b^d \\
O(n^{\log_b(a)}) & \text{if } a > b^d 
\end{cases} \]
Even more generally, for $T(n) = aT(n/b) + f(n)$...

**Theorem 3.2** (Master Theorem). Let $T(n) = a \cdot T \left( \frac{n}{b} \right) + f(n)$ be a recurrence where $a \geq 1$, $b > 1$. Then,

- If $f(n) = O \left( n^{\log_b a - \epsilon} \right)$ for some constant $\epsilon > 0$, $T(n) = \Theta \left( n^{\log_b a} \right)$.
- If $f(n) = \Theta \left( n^{\log_b a} \right)$, $T(n) = \Theta \left( n^{\log_b a \log n} \right)$.
- If $f(n) = \Omega \left( n^{\log_b a + \epsilon} \right)$ for some constant $\epsilon > 0$ and if $af(n/b) \leq cf(n)$ for $c < 1$ and all sufficiently large $n$, then $T(n) = \Theta(f(n))$. 
Recap

- $O()$ notation makes our lives easier.
- The ”Master Method” also make our lives easier.

Next time:

- What if the sub-problems are different sizes?
- And when might that happen?
Extra slides...
Some brainteasers

• Are there functions $f, g$ so that \textbf{NEITHER} $f = O(g)$ nor $f = \Omega(g)$?

• Are there \textit{non-decreasing} functions $f, g$ so that the above is true?

• Define the $n$’th fibonacci number by $F(0) = 1, F(1) = 1, F(n) = F(n-1) + F(n-2)$ for $n > 2$.
  • $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$

True or false:

• $F(n) = O(2^n)$
• $F(n) = \Omega(2^n)$
A few more $O()$ examples
Example A

• $g(n) = 2$, $f(n) = 1$.
• $g(n) = O(f(n))$ (and also $f(n) = O(g(n))$)

$T(n) = O(f(n)) \iff \exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0,$

\[ 0 \leq T(n) \leq c \cdot f(n) \]
Example B

- \( f(n) = 1 \), \( g(n) \) as below.
- \( g(n) = O(f(n)) \) (and also \( f(n) = O(g(n)) \))

\[
T(n) = O(f(n)) \iff \\
\exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0, \\
0 \leq T(n) \leq c \cdot f(n)
\]