CS 161 Section
W1: Asymptotic analysis, proof by induction.

Spring 2023
What is good pseudocode?
### Setup:
You have light bulbs of different sizes placed in order of their size, and have 1 socket. How do you match the socket to the light bulb?

<table>
<thead>
<tr>
<th>PSEUDOCODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>find_matching_light_bulb (LightBulbs, Socket)</td>
</tr>
<tr>
<td>search the light bulbs and return the matching lightbulb with the socket</td>
</tr>
</tbody>
</table>

- **Incorrect:**
  - `find_matching_light_bulb (lightBulbs, socket)`
  - `for lightBulb in lightBulbs: if lightBulb and socket match: return lightBulb return no matching light bulb found`

- **Correct:**
  - `find_matching_light_bulb (LightBulbs, Socket)`
  - `search the light bulbs and return the matching lightbulb with the socket`
Use algorithms covered in lectures without their pseudocode

```python
def find_matching_light_bulb(LightBulbs, Socket):
    ind = modified_binary_search(LightBulbs, Socket)
    if LightBulbs[ind] matches with socket:
        return LightBulbs[ind]
    else:
        return no matching light bulb found

modifed_binary_search is binary_search, but we change it [in this specific way].
```
**PSEUDOCODE**

- Simplify pseudocode as much as possible

```python
find_matching_light_bulb (lightBulbs, socket)
    if the 1st light bulb matches the socket:
        return the first light bulb
    if the 2nd light bulb matches the socket:
        return the 2nd light bulb
    if the 3rd light bulb matches the socket:
        return the 3rd light bulb
    ...
    if the nth light bulb matches the socket:
        return the nth light bulb
    return no matching light bulb found
```

```python
find_matching_light_bulb (lightBulbs, socket)
    for lightBulb in lightBulbs:
        if lightBulb and socket match:
            return lightBulb
    return no matching light bulb found
```
**PSEUDOCODE**

- Clear description of the steps
- Use algorithms covered in lectures without their pseudocode
- Simplify pseudocode as much as possible
BIG-O NOTATION
Let $T(n)$ & $f(n)$ be functions defined on the positive integers.

(In this class, we’ll typically write $T(n)$ to denote the worst case runtime of an algorithm)

**In English**

$T(n) = O(f(n))$ if and only if $T(n)$ is eventually **upper bounded** by a constant multiple of $f(n)$

**In Pictures**

- $T(n)$ as a function of $n$.
- $f(n)$ as a function of $n$.
- $c \cdot f(n)$ as a function of $n$.
- $n_0$ as a constant.

**In Math**

$T(n) = O(f(n))$ if and only if there exists positive **constants** $c$ and $n_0$ such that for all $n \geq n_0$

$$T(n) \leq c \cdot f(n)$$
Let \( T(n) \) & \( f(n) \) be functions defined on the positive integers. 

(In this class, we’ll typically write \( T(n) \) to denote the worst case runtime of an algorithm)

What do we mean when we say “\( T(n) \) is \( O(f(n)) \)”?

**In English**

\( T(n) = O(f(n)) \) if and only if \( T(n) \) is eventually **upper bounded** by a constant multiple of \( f(n) \)

**In Pictures**

![Diagram showing runtime vs input size](chart.png)

**In Math**

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

“if and only if” \( \iff \)

\[ \exists \ c, \ n_0 > 0 \ \text{s.t.} \ \forall \ n \geq n_0, \]

“there exists”

\[ T(n) \leq c \cdot f(n) \]

“such that”

“for all”
If you’re ever asked to formally prove that \( T(n) \) is \( O(f(n)) \), use the *MATH definition:

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

- **To prove** \( T(n) = O(f(n)) \), you need to announce your \( c \) & \( n_0 \) up front!
  - Play around with the expressions to find appropriate choices of \( c \) & \( n_0 \) (positive constants)
  - Then you can write the proof! Here how to structure the start of the proof:

  “Let \( c = \__ \) and \( n_0 = \__ \). We will show that \( T(n) \leq c \cdot f(n) \) for all \( n \geq n_0 \).”
PROVING BIG-O BOUNDS: EXAMPLE

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

Prove that \( 3n^2 + 5n = O(n^2) \).

*My thinking:* I want to find a \( c \) & \( n_0 \) such that for all \( n \geq n_0 \):

\[ 3n^2 + 5n \leq c \cdot n^2 \]

I can rearrange this inequality just to see things a bit more clearly:

\[ 5n \leq (c - 3) \cdot n^2 \]

Now let’s cancel out the \( n \):

\[ 5 \leq (c - 3) n \]

Let’s choose:

\[ c = 4 \]
\[ n_0 = 5 \]

(other choices work too! e.g. \( c = 10, n_0 = 10 \))
PROVING BIG-O BOUNDS: EXAMPLE

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

Prove that \(3n^2 + 5n = O(n^2)\).

Let \(c = 4\) and \(n_0 = 5\). We will now show that \(3n^2 + 5n \leq c \cdot n^2\) for all \(n \geq n_0\).

We know that for any \(n \geq n_0\), we have:

\[
\begin{align*}
5 &\leq n \\
5n &\leq n^2 \\
3n^2 + 5n &\leq 4n^2
\end{align*}
\]

Using our choice of \(c\) and \(n_0\), we have successfully shown that \(3n^2 + 5n \leq c \cdot n^2\) for all \(n \geq n_0\). From the definition of Big-O, this proves that \(3n^2 + 5n = O(n^2)\).
DISPROVING BIG-O BOUNDS

If you’re ever asked to formally disprove that $T(n)$ is $O(f(n))$, use **proof by contradiction**!

This means you need to show that **NO POSSIBLE CHOICE** of $c$ & $n_0$ exists such that the Big-O definition holds.
DISPROVING BIG-O BOUNDS

If you’re ever asked to formally disprove that T(n) is O(f(n)), use proof by contradiction!

For sake of contradiction, assume that T(n) is O(f(n)). In other words, assume there does indeed exist a choice of c & n₀ s.t. ∀ n ≥ n₀, T(n) ≤ c · f(n).

pretend you have a friend that comes up and says “I have a c & n₀ that will prove T(n) = O(f(n))!!!” and you say “ok fine, let’s assume your c & n₀ does prove T(n) = O(f(n))”

Treating c & n₀ as “variables”, derive a contradiction!

although you are skeptical, you’ll entertain your friend by saying: “let’s see what happens. [some math work... and then...] AHA! regardless of what your constants c & n₀, trusting you has led me to something impossible!!!”

Conclude that the original assumption must be false, so T(n) is not O(f(n)).

you have triumphantly proven your silly (or lying) friend wrong.
Prove that $3n^2 + 5n$ is \textit{not} $O(n)$.

For sake of contradiction, assume that $3n^2 + 5n$ is $O(n)$. This means that there exists positive constants $c$ & $n_0$ such that $3n^2 + 5n \leq c \cdot n$ for all $n \geq n_0$.

Then, we would have the following:
\begin{align*}
3n^2 + 5n &\leq c \cdot n \\
3n + 5 &\leq c \\
n &\leq \frac{(c - 5)}{3}
\end{align*}

However, since $(c - 5)/3$ is a constant, we’ve arrived at a contradiction since $n$ cannot be bounded above by a constant for all $n \geq n_0$. For instance, consider $n = n_0 + c$: we see that $n \geq n_0$, but $n > (c - 5)/3$. Thus, our original assumption was incorrect, which means that $3n^2 + 5n$ is not $O(n)$. 

\[ \Box \]
**BIG-O EXAMPLES**

**Polynomials**

Say \( p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 \) is a polynomial of degree \( k \geq 1 \).

Then:

i. \( p(n) = O(n^k) \)

ii. \( p(n) \) is **not** \( O(n^{k-1}) \)

---

\[ \log_2 n + 15 = O(\log_2 n) \]

\[ 3^n = O(4^n) \]

\[ 6n^3 + n \log_2 n = O(n^3) \]

\[ 25 = O(1) \]

[any constant] \( = O(1) \)

**lower order terms don’t matter!**

**remember, big-O is upper bound!**

**constant multipliers & lower order terms don’t matter**
AN ASIDE: $O(n \log n)$ vs. $O(n^2)$?

$log(n)$ grows very slowly! (Much more slowly than $n$)

$\begin{align*}
\log(2) &= 1 \\
\log(4) &= 2 \\
\log(64) &= 6 \\
\log(128) &= 7 \\
\log(4096) &= 12 \\
\log(\text{# particles in the universe}) &< 280
\end{align*}$

Logs are slow! In fact, $\log n = O(n^d)$ for any $d > 0$

$n \log n$ grows much more slowly than $n^2$

Punchline: A running time of $O(n \log n)$ is a LOT better than $O(n^2)$
Let $T(n)$ & $f(n)$ be functions defined on the positive integers.

(In this class, we'll typically write $T(n)$ to denote the worst case runtime of an algorithm)

What do we mean when we say “$T(n)$ is $\Omega(f(n))$”?  

**In English**  
$T(n) = \Omega(f(n))$ if and only if $T(n)$ is eventually **lower bounded** by a constant multiple of $f(n)$

**In Pictures**  

**In Math**  

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

Inequality switched directions!
We say “**T(n) is Θ(f(n))**” if and only if both

\[
T(n) = O(f(n)) \quad \text{and} \quad T(n) = Ω(f(n))
\]

\[
T(n) = Θ(f(n)) \iff \exists c_1, c_2, n_0 > 0 \text{ s.t. } \forall n \geq n_0, c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)
\]
### Asymptotic Notation Cheat Sheet

<table>
<thead>
<tr>
<th>Bound</th>
<th>Definition (How to Prove)</th>
<th>What it Represents</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T(n) = O(f(n)) )</td>
<td>( \exists \ c &gt; 0, \ \exists \ n_0 &gt; 0 \text{ s.t. } \forall \ n \geq n_0, \ T(n) \leq c \cdot f(n) )</td>
<td>upper bound</td>
</tr>
<tr>
<td>( T(n) = \Omega(f(n)) )</td>
<td>( \exists \ c &gt; 0, \ \exists \ n_0 &gt; 0 \text{ s.t. } \forall \ n \geq n_0, \ T(n) \geq c \cdot f(n) )</td>
<td>lower bound</td>
</tr>
<tr>
<td>( T(n) = \Theta(f(n)) )</td>
<td>( T(n) = O(f(n)) \text{ and } T(n) = \Omega(f(n)) )</td>
<td>tight bound</td>
</tr>
</tbody>
</table>
KARATSUBA INTEGER MULTIPLICATION

Three subproblems instead of four!
CHOOSING SUBPROBLEMS WISELY

\[
\begin{bmatrix}
x_1 & x_2 & \ldots & x_{n-1} & x_n \\
y_1 & y_2 & \ldots & y_{n-1} & y_n
\end{bmatrix}
\times
\begin{bmatrix}
x_1 & x_2 & \ldots & x_{n-1} & x_n \\
y_1 & y_2 & \ldots & y_{n-1} & y_n
\end{bmatrix}
= (a \times 10^{n/2} + b) \times (c \times 10^{n/2} + d)
= (a \times c)10^n + (a \times d + b \times c)10^{n/2} + (b \times d)
\]

The subproblems we choose to solve just need to provide these quantities:

- \(ac\)
- \(ad + bc\)
- \(bd\)

Originally, we assembled these quantities by computing FOUR things: \(ac\), \(ad\), \(bc\), and \(bd\).
KARATSUBA’S TRICK

\[
\text{end result} = (ac)10^n + (ad + bc)10^{n/2} + (bd)
\]

\[ac\] \& \[bd\] can be recursively computed as usual

\[ad + bc\] is equivalent to \[(a+b)(c+d) - ac - bd\]

\[= (ac + ad + bc + bd) - ac - bd\]
\[= ad + bc\]

So, instead of computing \[ad\] \& \[bc\] as two separate subproblems, let’s just compute \[(a+b)(c+d)\] instead!
OUR THREE SUBPROBLEMS

These *three* subproblems give us everything we need to compute our desired quantities:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ac</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>bd</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(a+b)(c+d)</td>
<td></td>
</tr>
</tbody>
</table>

(a+b) and (c+d) are both going to be $n/2$-digit numbers!

This means we still have half-sized subproblems!

Assemble our overall product by combining these three subproblems:

\[
\begin{align*}
(\text{ac})10^n + (\text{ad} + \text{bc})10^{n/2} + (\text{bd})
\end{align*}
\]
Karatsuba Multiplication Recursion Tree

Level 0: 1 problem of size n
Level 1: $3^1$ problems of size $n/2$
Level $t$: $3^t$ problems of size $n/2^t$
Level $\log_2 n$: $n^{1.6}$ problems of size 1

Thus, the runtime is $O(n^{1.6})$!
IT WORKS IN PRACTICE TOO!

\[ O(n^2) \]

\[ O(n^{1.6}) \]
REVIEW OF INDUCTION

How to write proof by induction.
4 INGREDIENTS OF INDUCTION

INDUCTIVE HYPOTHESIS (IH)
This is a statement that’s basically what you’re trying to prove, except it’s written in terms of some variable (e.g. \(i\)). We need to set up the inductive hypothesis clearly, and our goal in the next three steps is to prove that the IH holds for a whole range of values for \(i\).

BASE CASE
First establish that the inductive hypothesis holds for some base case value(s) of \(i\).

INDUCTIVE STEP (strong/complete induction version)
Next, assume that the IH holds when \(i\) takes on any value between [base case value(s)] and some number \(k\). Now prove that the IH holds as well when \(i\) takes on the value \(k\).

CONCLUSION
By induction, conclude that the IH holds across the range of \(i\) you’re dealing with.
PROVE CORRECTNESS w/ INDUCTION

ITERATIVE ALGORITHMS

1. **Inductive hypothesis**: some state/condition will always hold throughout your algorithm by any iteration $i$
2. **Base case**: show IH holds for iteration 0 (i.e. start of algorithm)
3. **Inductive step**: Assume IH holds for $k$ $\Rightarrow$ prove $k+1$
4. **Conclusion**: IH holds for $i = \#$ total iterations $\Rightarrow$ yay!

RECURSIVE ALGORITHMS

1. **Inductive hypothesis**: your algorithm is correct for sizes up to $i$
2. **Base case**: IH holds for $i < \text{small const.}$
3. **Inductive step**:  
   - assume IH holds for $k$ $\Rightarrow$ prove $k+1$, OR  
   - assume IH holds for $\{1,2,...,k-1\}$ $\Rightarrow$ prove $k$ (*it’s not important that I chose $k$ instead of $k+1$, using $k$ is can just be syntactically cleaner!)
4. **Conclusion**: IH holds for $i = n$ $\Rightarrow$ yay!
Example: MERGESORT

Algorithm, Proof of Correctness, Runtime
● **DIVIDE-AND-CONQUER:** an algorithm design paradigm

1. break up a problem into smaller subproblems
2. solve those subproblems *recursively*
3. combine the results of those subproblems to get the overall answer
MERGESORT: RECURSIVE CALLS

This is where we hit our base case!
MERGESORT: MERGE STEPS

We have a sorted sequence!
**Intuition:** Divide and Conquer. If you sort your left and right halves, it’s easier to “Merge” them into a sorted list.

**MERGESORT (A):**

```python
n = len(A)
if n <= 1:
    return A
L = MERGESORT(A[0:n/2])
R = MERGESORT(A[n/2:n])
return MERGE(L, R)
```

**MERGE (L, R):**

```python
result = length n array
i = 0, j = 0
for k in [0,...,n-1]:
    if L[i] < R[j]:
        result[k] = L[i]
        i += 1
    else:
        result[k] = R[j]
        j += 1
return result
```
THIS IS A JOB FOR: PROOF BY INDUCTION!
(This time, we perform induction on the length of input list, rather than # of iterations)
MERGESORT: INDUCTION PROOF

INDUCTIVE HYPOTHESIS (IH)
In every recursive call on an array of length at most \(i\), MERGESORT returns a sorted array.

BASE CASE
The IH holds for \(i = 1\): A 1-element array is always sorted.

INDUCTIVE STEP (strong/complete induction)
Let \(k\) be an integer, where \(1 < k \leq n\). Assume that the IH holds for \(i < k\), so MERGESORT correctly returns a sorted array when called on arrays of length less than \(k\). We want to show that the IH holds for \(i = k\), i.e. that MERGESORT returns a sorted array when called on an array of length \(k\).

[INSERT INDUCTION PROOF TO PROVE THE MERGE SUBROUTINE IS CORRECT WHEN GIVEN TWO SORTED ARRAYS]
Since the two “child” recursive calls are executed on arrays of length \(k/2\) (which is strictly less than \(k\)), then our inductive hypothesis tells us that MERGESORT will correctly sort the left and right halves of our length-\(k\) array. Then, since the MERGE subroutine is correct when given two sorted arrays, we know that MERGESORT will ultimately return a fully sorted array of length \(k\).

CONCLUSION
By induction, we conclude that the IH holds for all \(1 \leq i \leq n\). In particular, it holds for \(i = n\), so in the top recursive call, MERGESORT returns a sorted array.
Mergesort: Is it fast?

\[
\text{Mergesort}(A): \\
\quad n = \text{len}(A) \\
\quad \text{if } n \leq 1: \\
\quad \quad \text{return } A \\
\quad L = \text{Mergesort}(A[0:n/2]) \\
\quad R = \text{Mergesort}(A[n/2:n]) \\
\quad \text{return } \text{merge}(L,R)
\]

Claim: Mergesort runs in time $O(n \log n)$
If a subproblem is of size $n$, then the work done in that subproblem is $O(n)$.  
$\Rightarrow \text{Work} \leq c \cdot n$ (c is a constant)

---

<table>
<thead>
<tr>
<th>Level</th>
<th># of Problems</th>
<th>Size of each Problem</th>
<th>Work done per Problem $\leq$</th>
<th>Total work on this level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>$n$</td>
<td>$c \cdot n$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>1</td>
<td>$2^1$</td>
<td>$n/2$</td>
<td>$c \cdot (n/2)$</td>
<td>$2^1 \cdot c \cdot (n/2) = O(n)$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$t$</td>
<td>$2^t$</td>
<td>$n/2^t$</td>
<td>$c \cdot (n/2^t)$</td>
<td>$2^t \cdot c \cdot (n/2^t) = O(n)$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$\log_2 n$</td>
<td>$2^{\log_2 n} = n$</td>
<td>1</td>
<td>$c \cdot (1)$</td>
<td>$n \cdot c \cdot (1) = O(n)$</td>
</tr>
</tbody>
</table>

We have $(\log_2 n + 1)$ levels, each level has $O(n)$ work total  
$\Rightarrow \text{O}(n \log n)$ work overall!