BEFORE WE START
Announcements

• Project 0 (CSE 143 Review) due Wednesday 10/7 11:59pm

• Project 1 (Deques) comes out the same day
  - Three options for projects:
    - Choose a partner – someone you know or meet in the class
    - Join the partner pool – we’ll assign you a partner
    - Opt to work alone – not recommended, but available

• Exercise 1 (written, individual) released Friday

• Option to choose your breakout room for class sessions!
  - See Ed announcement for details on how to sign up! Can modify at any time.
  - Will still use random assignment for those in class who don’t sign up.
Learning Objectives

After this lecture, you should be able to...

1. Describe the difference between Code Modeling and Asymptotic Analysis (both components of Algorithmic Analysis)

2. Model a (simple) piece of code with a function describing its runtime

3. Explain why we can throw away constants when we compute Big-Oh bounds.
   - From a practical perspective and from the “definition” perspective.
Lecture Outline

• **Overview: Algorithmic Analysis**

• Code Modeling

• Asymptotic Analysis

• Big-O Definition
143 Review  Complexity Class

- **Complexity Class**: a category of algorithm efficiency based on the algorithm’s relationship to the input size N

<table>
<thead>
<tr>
<th>Complexity Class</th>
<th>Big-O</th>
<th>Runtime if you double N</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>$O(1)$</td>
<td>unchanged</td>
</tr>
<tr>
<td>logarithmic</td>
<td>$O(\log_2 N)$</td>
<td>increases slightly</td>
</tr>
<tr>
<td>linear</td>
<td>$O(N)$</td>
<td>doubles</td>
</tr>
<tr>
<td>log-linear</td>
<td>$O(N \log_2 N)$</td>
<td>slightly more than doubles</td>
</tr>
<tr>
<td>quadratic</td>
<td>$O(N^2)$</td>
<td>quadruples</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>exponential</td>
<td>$O(2^N)$</td>
<td>multiplies drastically</td>
</tr>
</tbody>
</table>
### Review

**Big-Oh Analysis: Why?**

<table>
<thead>
<tr>
<th>Operation</th>
<th>ArrayList</th>
<th>LinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td>add (front)</td>
<td>$O(n)$ linear</td>
<td>$O(1)$ constant</td>
</tr>
<tr>
<td>remove (front)</td>
<td>$O(n)$ linear</td>
<td>$O(1)$ constant</td>
</tr>
<tr>
<td>add (back)</td>
<td>$O(1)$ constant</td>
<td>$O(n)$ linear</td>
</tr>
<tr>
<td>remove (back)</td>
<td>$O(1)$ constant</td>
<td>$O(n)$ linear</td>
</tr>
<tr>
<td>get</td>
<td>$O(1)$ constant</td>
<td>$O(n)$ linear</td>
</tr>
<tr>
<td>insert (anywhere)</td>
<td>$O(n)$ linear</td>
<td>$O(n)$ linear</td>
</tr>
</tbody>
</table>

- Complexity classes help us differentiate between data structures
  - “Just change first node” vs. “Change every element” is clearly different
  - To *evaluate* data structures, need to understand impact of design decisions
Review  Big-Oh Analysis: Why?

• We need a tool to analyze code, and we want it to be:

  A  Simple
  We don’t care about tiny differences in implementation, want the big picture result

  B  Mathematically Rigorous
  Use mathematical functions as a precise, flexible basis

  C  Decisive
  Produce a clear comparison indicating which code takes “longer”
**Review**  
Big-Oh Analysis: ... How?!

143 general patterns: “O(1) constant is no loops, O(n) is one loop, O(n^2) is nested loops”
- This is still useful!
- But in 373 we’ll go much more in depth: we can explain more about why, and how to handle more complex cases when they arise (which they will!)
Overview: Algorithmic Analysis

- **Algorithmic Analysis**: The overall process of characterizing code with a complexity class, consisting of:
  - **Code Modeling**: Code $\rightarrow$ Function describing code’s runtime
  - **Asymptotic Analysis**: Function $\rightarrow$ Complexity class describing asymptotic behavior

```java
for (i = 0; i < n; i++) {
    a[i] = 1;
    b[i] = 2;
}
```

\[ f(n) = 2n \]

\[ O(n) \]
Lecture Outline

• Overview: Algorithmic Analysis

• **Code Modeling**

• Asymptotic Analysis

• Big-O Definition
Talking About Code

• **Cost Model**: An analysis mindset to express the resource whose growth rate is being measured

• For simplicity, we’ll discuss everything in terms of runtime today
  - But other cost models exist! For example, storage space is common
Code Modeling

- **Code Modeling** – the process of mathematically representing how many operations a piece of code will run in relation to the input size n.
  - Convert from code to a function representing its runtime
What is an operation?

• We don’t know exact runtime of every operation, but for now let’s try simplifying assumption: all basic operations take the same time

• Basics:
  - +, -, /, *, %, ==
  - Assignment
  - Returning
  - Variable/array access

• Function Calls
  - Total runtime in body
  - Remember: new calls a function (constructor)

• Conditionals
  - Test + time for the followed branch
    - Learn how to reason about branch later

• Loops
  - Number of iterations * total runtime in condition and body
Code Modeling Example I

```java
public void method1(int n) {
    int sum = 0;  // +1
    int i = 0;   // +1
    while (i < n) {  // +1
        sum = sum + (i * 3);  // +3
        i = i + 1;  // +2
    }
    return sum;  // +1
}
```

Loop runs $n$ times

$f(n) = 6n + 3$
public void method2(int n) {
    int sum = 0;  \+1
    int i = 0;   \+1
    while (i < n) {  \+1
        int j = 0;  \+1
        while (j < n) {  \+1
            if (j % 2 == 0) {  \+2
                // do nothing
            }  \+2
            sum = sum + (i * 3) + j; \+4
            j = j + 1; \+2
        } \+9
        i = i + 1; \+2
    } \+1
    return sum;
} \+1

This inner loop runs n times
This outer loop runs n times
f(n) = (9n+4)n + 3
Lecture Outline

• Overview: Algorithmic Analysis
• Code Modeling
• Asymptotic Analysis
• Big-O Definition
Where are we?

- We just turned a piece of code into a function!
  - We’ll look at better alternatives for code modeling later
- Now to focus on step 2, asymptotic analysis

```c
for (i = 0; i < n; i++) {
    a[i] = 1;
    b[i] = 2;
}
```

$\text{RUNTIME FUNCTION}$

$\text{Asymptotic Analysis}$

$O(n)$

$\text{COMPLEXITY CLASS}$

$\text{CODE}$

$\text{Code Modeling}$
Finding a Big-Oh

• We have an expression for $f(n)$. How do we get the $O()$ that we’ve been talking about?

1. Find the “dominating term” and delete all others.
   - The “dominating” term is the one that is largest as $n$ gets bigger. In this class, often the largest power of $n$.

2. Remove any constant factors.

$$f(n) = (9n+3)n + 3$$

$$= 9n^2 + 3n + 3$$

$$\approx 9n^2$$

$$\approx n^2$$

$$f(n) \text{ is } O(n^2)$$
Asymptotic Analysis

What is the complexity class for the following function?

\[ f(n) = 1,0000 + 400n + 0.00001n^3 + 20n^2 \]
Is it okay to throw away all that info?

- Big-Oh is like the “significant digits” of computer science
- **Asymptotic Analysis**: Analysis of function behavior as its input approaches infinity
  - We only care about what happens when $n$ approaches infinity
  - For small inputs, doesn’t really matter: all code is “fast enough”
  - Since we’re dealing with infinity, constants and lower-order terms don’t meaningfully add to the final result. The highest-order term is what drives growth!

Remember our goals:

- **Simple**: We don’t care about tiny differences in implementation, want the big picture result
- **Decisive**: Produce a clear comparison indicating which code takes “longer”
No seriously, this is really okay?

- There are tiny variations in these functions ($2n$ vs. $3n$ vs. $3n+1$)
  - But at infinity, will be clearly grouped together
  - We care about which group a function belongs in

- Let’s convince ourselves this is the right thing to do:
  - [https://www.desmos.com/calculator/t9qvn56yyb](https://www.desmos.com/calculator/t9qvn56yyb)
What *is* an operation, again?

- We could try being more precise, and count up individual operations
  - Then, sum the time each operation takes
  - But how long *do* they take? Some architectures are really fast at +, others faster at assignment
  - And when we compile it, our code gets expressed as lower-level operations anyway! *It’s almost impossible to stare at code and know the “true” constants.*

<table>
<thead>
<tr>
<th>Operation</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment</td>
<td>$2 + 2n$</td>
</tr>
<tr>
<td>$&lt;$</td>
<td>$n$</td>
</tr>
<tr>
<td>$+$</td>
<td>$2n$</td>
</tr>
<tr>
<td>$*$</td>
<td>$n$</td>
</tr>
<tr>
<td>Return</td>
<td>1</td>
</tr>
</tbody>
</table>

```java
public void method1(int n) {
    int sum = 0;
    int i = 0;
    while (i < n) {
        sum = sum + (i * 3);
        i = i + 1;
    }
    return sum;
}
```

```java
public static void method1(int[]); Code:
0: iconst_0
1: istore_1
2: iconst_0
3: istore_2
4: iload_2
5: iload_0
6: if_icmpge 22
7: iload_1
8: iload_0
9: iload_1
10: iload_2
11: iconst_3
12: imul
13: iadd
14: istore_1
15: iload_1
16: icmpge
17: ireturn
18: istore_2
19: goto 4
20: ireturn

public void method1(int n) {
    int sum = 0;
    int i = 0;
    while (i < n) {
        sum = sum + (i * 3);
        i = i + 1;
    }
    return sum;
}
Code Modeling Anticipating Asymptotic Analysis

• We can’t accurately model the constant factors just by staring at the code.
  - And the lower-order terms matter even less than the constant factors.

• Since they’re going to be thrown away anyway, you can anticipate which constants are unnecessary to count precisely during Code Modeling
  - e.g. a loop body containing a constant 2 vs. 10 operations is unimportant here

• This does not mean you shouldn’t care about constant factors ever – they are important in real code!
  - Asymptotic analysis is just one tool, but other perspectives that do consider constants are also valid and useful!
Big-Oh Analysis: Why?

• We need a tool to analyze code, and we want it to be:

  A  Simple
  We don’t care about tiny differences in implementation, want the big picture result

  B  Mathematically Rigorous
  Use mathematical functions as a precise, flexible basis

  C  Decisive
  Produce a clear comparison indicating which code takes “longer”
Lecture Outline

• Overview: Algorithmic Analysis

• Code Modeling

• Asymptotic Analysis

• Big-O Definition
Using Formal Definitions

• If analyzing simple or familiar functions, don’t bother with the formal definition. You can be comfortable using your intuition!

• We’re going to be making more subtle big-O statements in this class.
  - We need a mathematical definition to be sure we know exactly where we are.

• We’re going to teach you how to use the formal definition, so if you come across a weird edge case, you know how to get your bearings.

Mathematically Rigorous
Use mathematical functions as a precise, flexible basis
Big-Oh Definition

- We wanted to find an upper bound on our algorithm’s running time, but
  - We only care about what happens as $n$ gets large.
  - We don’t want to care about constant factors.

**Big-Oh**

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

Intuition: $g(n)$ “eventually dominates” $f(n)$

**Why $n_0$?**
- $f_1(n) = 0.01n^2$
- $f_2(n) = n$

**Why $c$?**
- $f(n) = 5n$
- $g(n) = n$
Big-Oh Proofs

Show that \( f(n) = 10n + 15 \) is \( O(n) \).

Apply definition term by term

\[ 10n \leq c \cdot n \text{ when } c = 10 \text{ for all values of } n. \text{ So } 10n \leq 10n, \text{ for all } n \]

\[ 15 \leq c \cdot n \text{ when } c = 15 \text{ for } n \geq 1. \text{ So } 15 \leq 15n, \text{ when } n \geq 1 \]

Add up all your truths

\[ 10n + 15 \leq 10n + 15n = 25n \text{ for } n \geq 1 \]

\[ 10n + 15 \leq 25n \text{ for } n \geq 1. \]

which is in the form of the definition

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

where \( c = 25 \) and \( n_0 = 1 \).
Big-Oh Doesn’t Have to be Tight

• True or False: $10n^2$ is $O(n^3)$

• It’s true – it fits the definition
  $10n^2 \leq c \cdot n^3$ when $c = 10$, for $n \geq 1$

• Big-O is just an upper bound that may be loose and not describe the function fully. For example, all of the following are true:

  10$n^2$ is $O(n^3)$
  10$n^2$ is $O(n^4)$
  10$n^2$ is $O(n^5)$
  10$n^2$ is $O(n^n)$
  10$n^2$ is $O(n!)$ ... and so on