Before we start

Which of the following is an invariant of our LinkedList implementation?

a) it can store any type inside
b) adding at the front is O(1)
c) the last node’s next field is null
Announcements

• Project 0 (CSE 143 Review) due Wednesday 7/1 11:59pm
• Project 1 (Deques) comes out the same day
  - Partner sign-up form due Tuesday 6/30 11:59pm
  - 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 (default)
    - Opt to work alone – not recommended, but available
• Friday (July 3\textsuperscript{rd}) is a holiday: Independence Day (observed)
  - No lecture
• Exercise 1 (written, individual) released Friday
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

4. Identify whether Big-Oh (and Big-Omega, Big-Theta) statements about a function are accurate
Lecture Outline

• **Overview: Algorithmic Analysis**

• Code Modeling

• Asymptotic Analysis

• Big-O, Big-Omega, Big-Theta
**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></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 usually</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!)

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

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

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

- $f(n) = 2n$
- $O(n)$
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

- This topic has a lot of details/relationships between concepts
  - We’ll try to introduce things one at a time, but might take until next week for a “full”/satisfying picture to emerge
Lecture Outline

• Overview: Algorithmic Analysis

• Code Modeling

• Asymptotic Analysis

• Big-O, Big-Omega, Big-Theta
• **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;
    int i = 0;
    while (i < n) {
        int j = 0;
        while (j < n) {
            if (j % 2 == 0) {
                // do nothing
            }
            sum = sum + (i * 3) + j;
            j = j + 1;
        }
        i = i + 1;
    }
    return sum;
}
Lecture Outline

• Overview: Algorithmic Analysis

• Code Modeling

• Asymptotic Analysis

• Big-O, Big-Omega, Big-Theta
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

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

\[ f(n) = 2n \]

\[ \mathcal{O}(n) \]
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.

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

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

$$\approx 9n^2$$

$$\approx n^2$$

$f(n)$ is $O(n^2)$
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: aload_0
1: arraylength
2: istore_1
3: aload_0
4: iload_1
5: iconst_1
6: isub
7: aload_0
8: iadd
9: iastore
10: aload_0
11: iconst_4
12: iaload
13: iadd
14: iastore
15: aload_0
16: iconst_0
17: dup2
18: iaload
19: aload_0
20: iconst_1
21: iaload
22: iadd
23: iastore
24: return
```
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!