LEC 04

# CSE 373

# Asymptotic Analysis

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


Instructor Aaron Johnston

TAs Timothy Akintilo, Brian Chan, Joyce Elauria, Eric Fan, Farrell Fileas, Leona Kazi, Keanu Vestil, Howard Xiao, Siddharth Vaidyanathan

# 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 3rd) 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

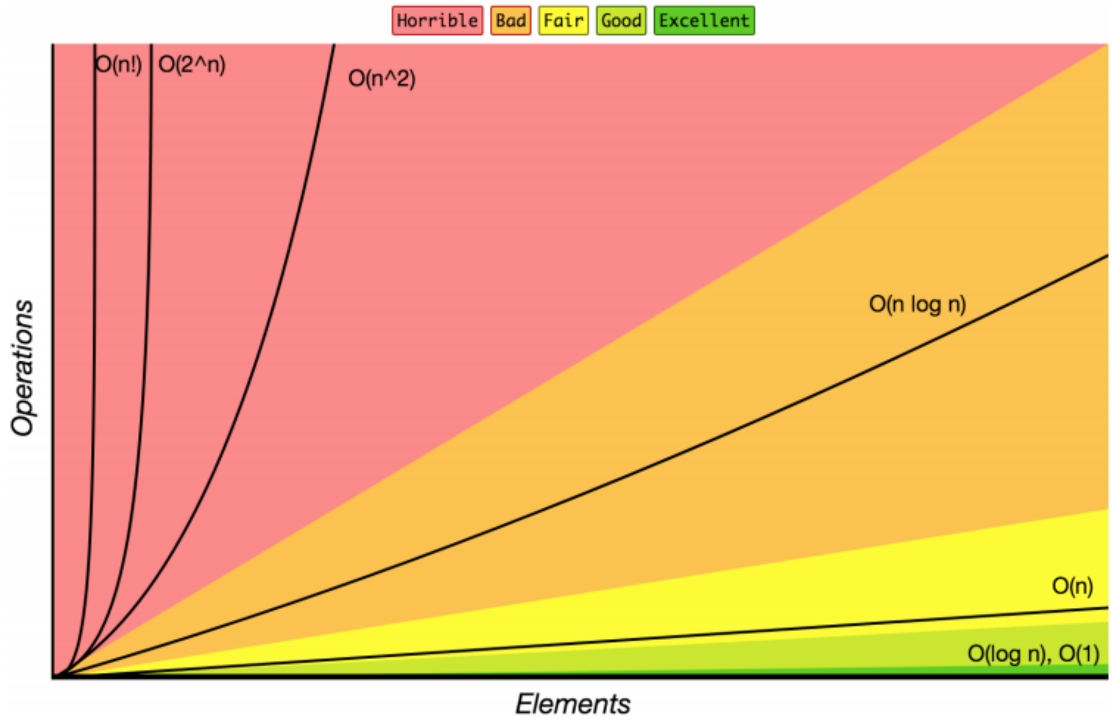
# Lecture Outline

- **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

| Complexity Class | Big-O | Runtime if you double N |
|---|---|---|
| **constant** | **O(1)** | **unchanged** |
| logarithmic | $O(\log_2 N)$ | increases slightly |
| **linear** | **O(N)** | **doubles** |
| log-linear | $O(N \log_2 N)$ | slightly more than doubles |
| **quadratic** | $\mathbf{O(N^2)}$ | **quadruples** |
| … | … | … |
| exponential | $O(2^N)$ | multiplies drastically |



# *Review* Big-Oh Analysis: Why?

| | ArrayList | LinkedList |
|---|---|---|
| `add (front)` | O(n) linear | O(1) constant |
| `remove (front)` | O(n) linear | O(1) constant |
| `add (back)` | O(1) constant usually | O(n) linear |
| `remove (back)` | O(1) constant | O(n) linear |
| `get` | O(1) constant | O(n) linear |
| `insert (anywhere)` | O(n) linear | O(n) linear |

- 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?!

CODE

BIG-OH

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

O(n)

- 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


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

$f(n) = 2n$

$O(n)$

- **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

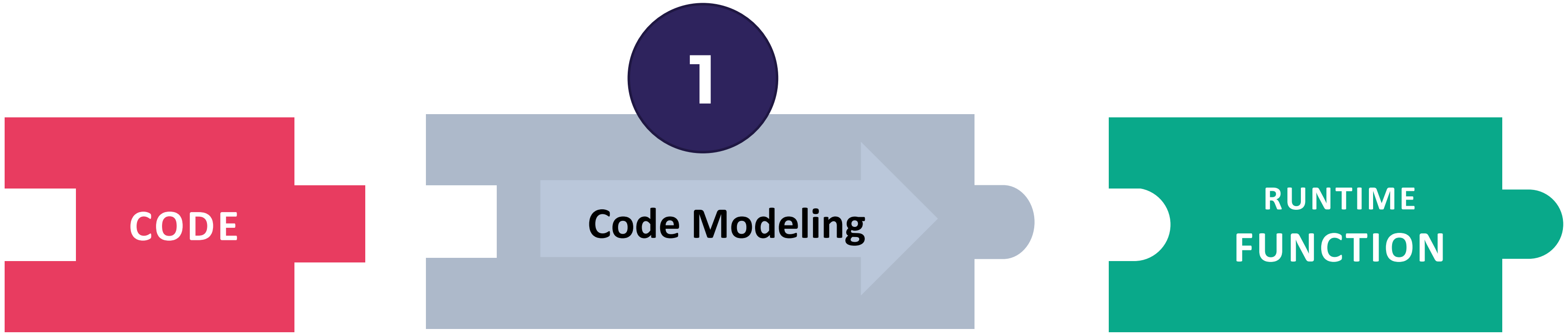
# Talking About Code

- **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




- **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

```
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

+6 *n

$f(n) = 6n + 3$

# Code Modeling Example II

```
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
            }
            sum = sum + (i * 3) + j;  +4
            j = j + 1;              +2
        }
        i = i + 1;                  +2
    } return sum;                   +1
}
```

This inner loop runs n times

+9 *n

This outer loop runs n times

9n + 4 *n

f(n) = (9n+4)n + 3

# Lecture Outline

- Overview: Algorithmic Analysis
- Code Modeling
- **Asymptotic Analysis**
- Big-O, Big-Omega, Big-Theta

# Where are we?

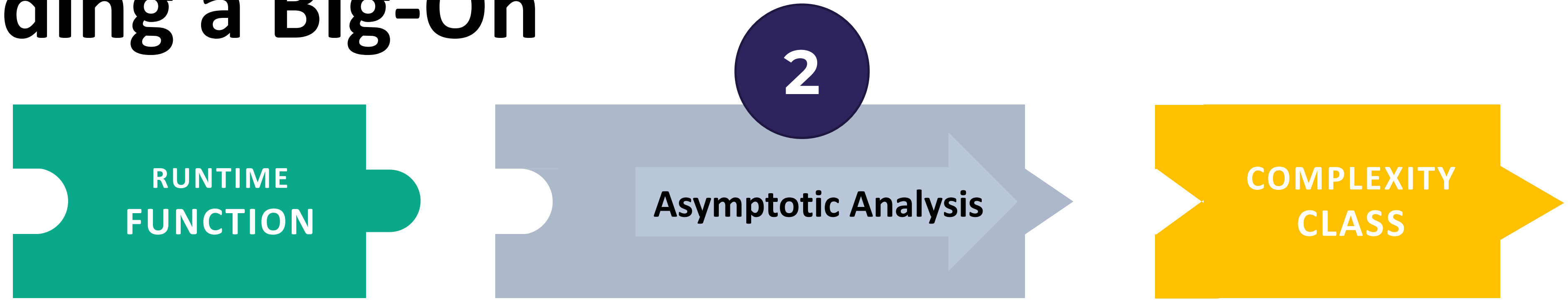

```
for (i = 0; i < n; i++) {
  a[i] = 1;
  b[i] = 2;
}
```

$f(n) = 2n$

$O(n)$

- 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

# 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?

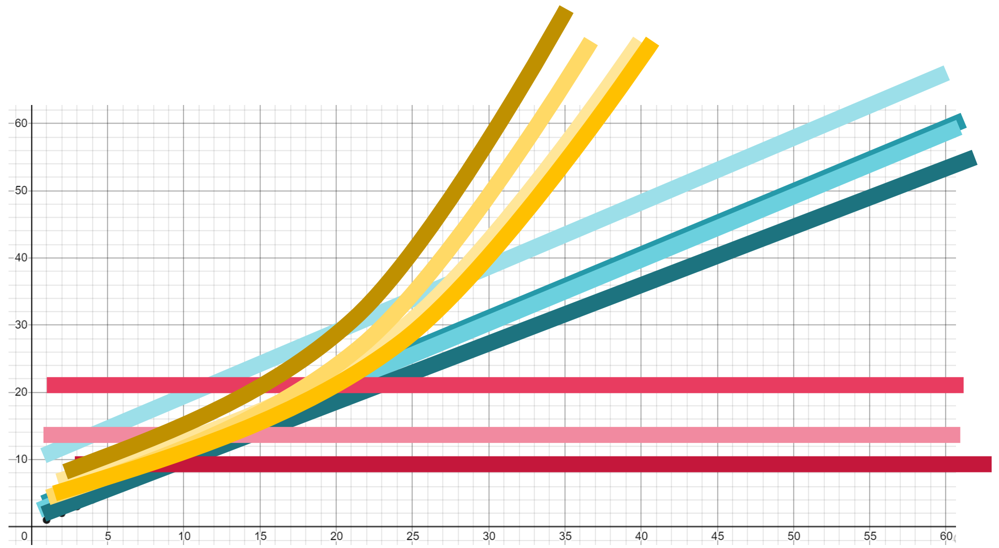
- Big-Oh is like the "significant digits" of computer science
- **Asymptotic Analysis**: Analysis of function behavior as its input approaches infinity
  - We only care about what happens when n approaches infinity
  - For small inputs, doesn't really matter: all code is "fast enough"
  - Since we're 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:

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

C
**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

# What *is* an operation, again?

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

| Operation | Count |
|---|---|
| Assignment | 2 + 2n |
| < | n |
| + | 2n |
| * | n |
| Return | 1 |

- 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.**

```
public static void method1(int[]); Code:
0: aload_0          10: aload_0      20: iconst_1
1: arraylength      11: iconst_4     21: iaload
2: istore_1         12: iaload       22: iadd
3: aload_0          13: iadd         23: iastore
4: iload_1          14: iastore      24: return
5: iconst_1         15: aload_0
6: isub             16: iconst_0
7: aload_0          17: dup2
8: iconst_3         18: iaload
9: iaload           19: aload_0
```

# 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!