Lecture 6: More Code Analysis, Testing

CSE 373: Data Structures and Algorithms

Warm Up - Code Modeling

Take 3 Minutes

```
public static boolean isPrime(int n) {
   for (int i = 2; i < n; i++){
      if (n % i == 0) { +2
         return false; +1
      }
   }
   return true; +1
}
```

n - 2

Approach

-> start with basic operations, work inside out for control structures

- Each basic operation = +1
- Conditionals = worst case test operations + branch
- Loop = iterations (loop body)

Extra Credit:

Go to PollEv.com/champk

Text CHAMPK to 22333 to join session, text “1” or “2” to select your answer

Answer:

code model: $3n-5$ or $C_1 n + C_2$

simplified tight-O bound: $O(n)$

Administrivia

Homework 2 is out!

- You should have a repo
- If you have issues submitting post on piazza

Modeling Complex Loops

Write a mathematical model of the following code

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        System.out.println("Hello!");
    }
}
```

+1 n n $f(n) = n^2$

Keep an eye on loop bounds!

Modeling Complex Loops

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        System.out.print("Hello! ");
    }
    Sysem.out.println();
}
```

+1

0 + 1 + 2 + 3 +…+ i-1

n

$$T(n) = \underbrace{(0 + 1 + 2 + 3 + \ldots + i-1)}_{\text{How do we model this part?}}$$

Summations!

$$1 + 2 + 3 + 4 + \ldots + n = \sum_{i=1}^{n} i$$

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} 1$$

What is the Big O?

Definition: Summation
$\sum_{i=a}^{b} f(i)$ = f(a) + f(a + 1) + f(a + 2) + … + f(b-2) + f(b-1) + f(b)

On the ith iteration of the outer loop	Output of ith iteration
0	""
1	"Hello! "
2	"Hello! Hello! "

Simplifying Summations

Find closed form using summation identities (given on exams)

```java
for (int i = 0; i < n; i++) {
   for (int j = 0; j < i; j++) {
      System.out.println("Hello!");
   }
}
```

→ $T(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} 1$ → closed form → simplified tight big O

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} 1 \quad = \sum_{i=0}^{n-1} 1 \cdot i \quad = 1 \sum_{i=0}^{n-1} i \quad = \frac{n(n-1)}{2} \quad = \frac{1}{2}n^2 - \frac{1}{2}n \quad = \boldsymbol{O(n^2)}$$

Summation of a constant

$$\sum_{i=0}^{n-1} c = cn$$

Factoring out a constant

$$\sum_{i=a}^{b} cf(i) = c \sum_{i=a}^{b} f(i)$$

Gauss's Identity

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$$

Practice – Worksheet #2

Take 3 Minutes

```
public static void primesUpToN(int n) {
    System.out.print("1 2 ");
    for (int i = 3; i <= n; i++) {
        for (int j = 2; j < i; j++){
            if (j != i && j % i == 0) {
                System.out.print(i + " ");
                break;
            }
        }
    }
    System.out.println();
}
```

+1

+4

+1

$\sum_{j=2}^{i-1} 5$

$\sum_{i=3}^{n} \sum_{j=2}^{i-1} 5$

+1

$T(n) = 1 + \sum_{i=3}^{n} \sum_{j=2}^{i-1} 5 = 1 + \sum_{i=0}^{n-3} \sum_{j=0}^{i-3} 5 = 1 + \sum_{i=0}^{n-3} 5(i-2) = 1 + 5(\sum_{i=0}^{n-3} i - \sum_{i=0}^{n-3} 2) - = 1 + 5(\frac{(n-2)(n-3)}{2}$ - (n-2)(2))

Adjusting summation bounds	Summation of a constant	Factoring out a constant	Gauss's identity

Definition: Big-O

We wanted to find an upper bound on our algorithm's running time, but

- We don't want to care about constant factors.
- We only care about what happens as n gets large.

Big-O

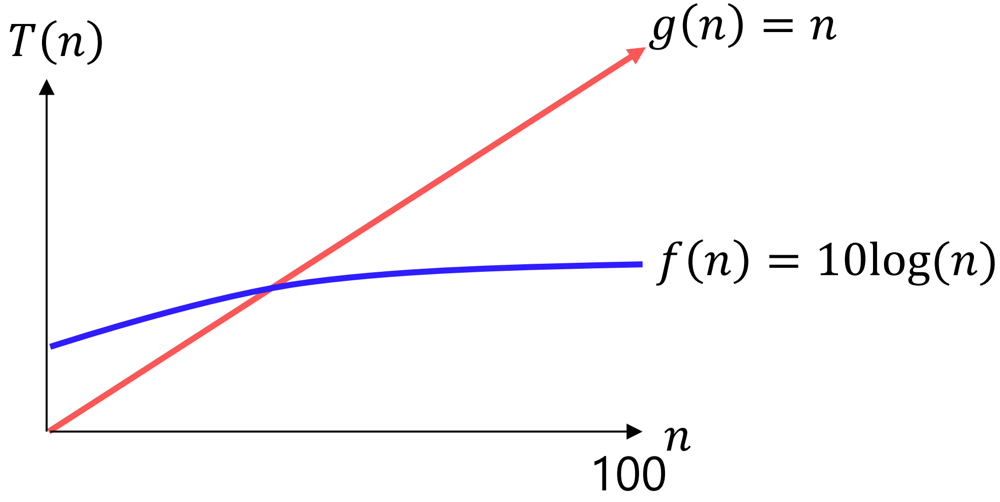
$f(n)$ is $O(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$,

$$f(n) \leq c \cdot g(n)$$

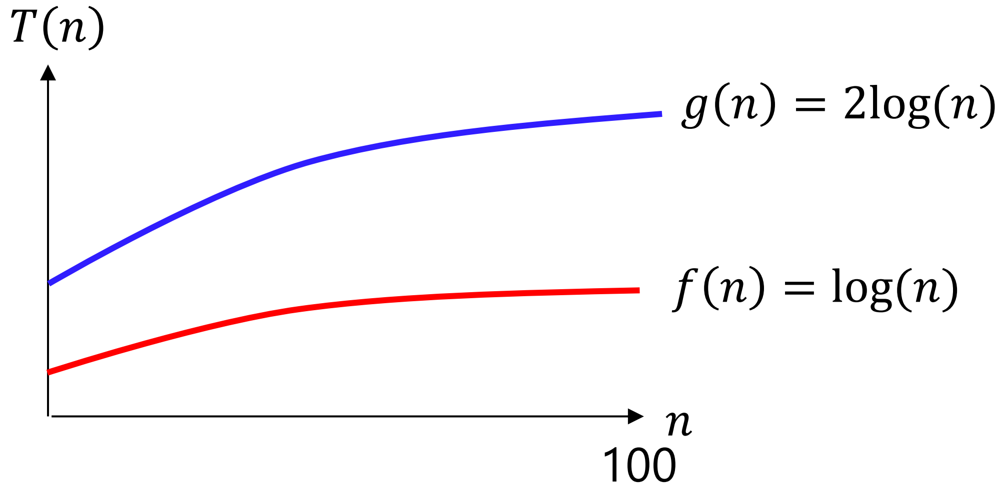
We also say that $g(n)$ "dominates" $f(n)$

O(g(n)) is the "family" or "set" of all functions that are dominated by g(n)

Why n_0?



Why c?



Practice – Worksheet #3

Take 3 Minutes

Prove the function $f(n) = \frac{n^2}{2} - \frac{3n}{2} \in O(n^2)$ by finding a c and n_0. Show your work

$$\frac{n^2}{2} \le c \cdot n^2 \ when\ c = \frac{1}{2}\ and\ n_0 = 1$$
$$-\frac{3n}{2} \le\ c \cdot n^2 \ when\ c = 1\ and\ n_0 = 1$$
$$combing\ it\ all\ together\ \ldots$$
$$\frac{n^2}{2} - \frac{3n}{2} \le \frac{1}{2}n^2 + n^2 \le \frac{3}{2}n^2 \ when\ n_0 = 1$$
$$c = \frac{3}{2}\ and\ n_0 = 1\ show\ that\ f(n) \le g(n))$$

O, Omega, Theta [oh my?]

Big-O is an **upper bound**

- My code takes at most this long to run

Big-Omega is a **lower bound**

Big-Omega

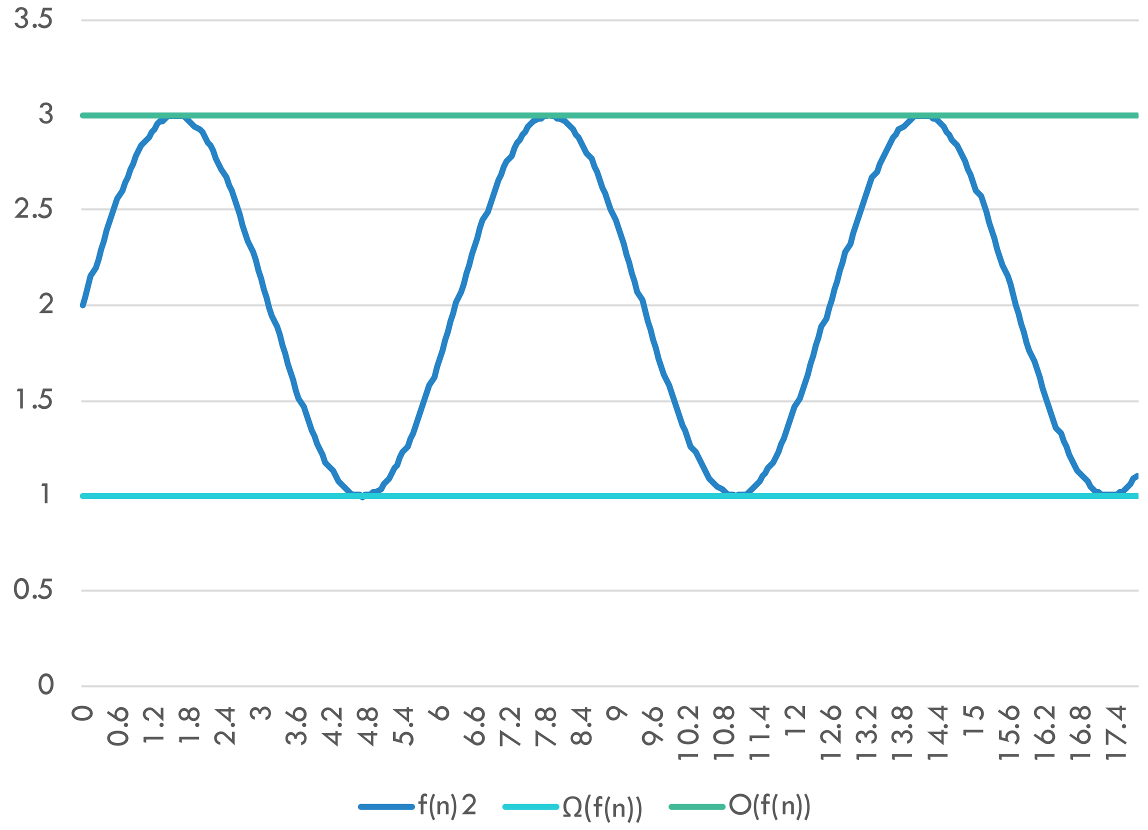
$f(n)$ is $\Omega(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$,

$$f(n) \geq c \cdot g(n)$$

Big Theta is **"equal to"**

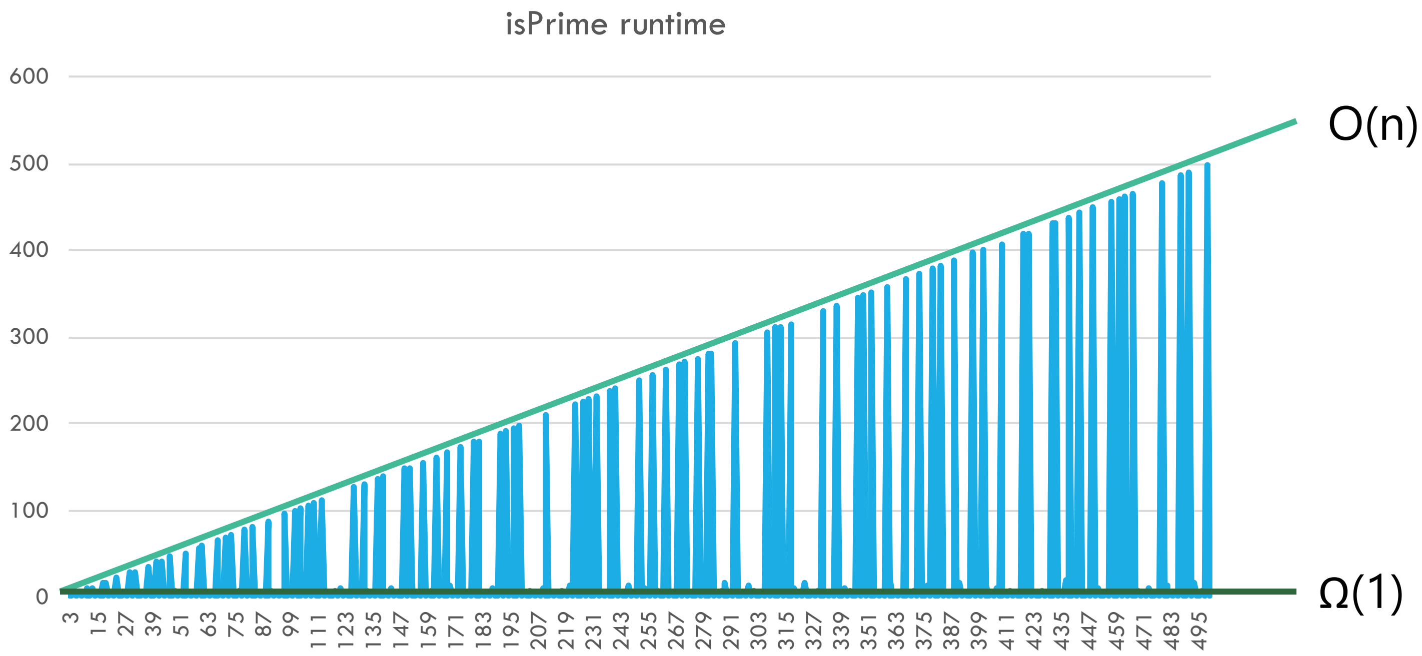
Big-Theta

$f(n)$ is $\Theta(g(n))$ if
$f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.



Practice – Worksheet #4

Take 2 Minutes



What is the tight big-O bound?

O(n)

What is the tight big-Ω bound?

Ω(1)

What is the big-ϴ bound?

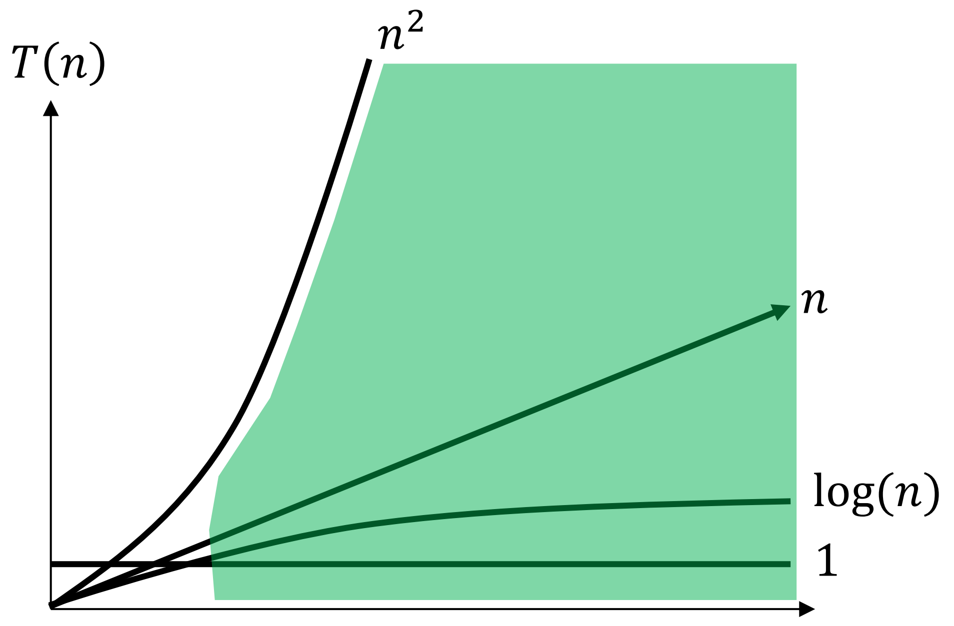
Doesn't exist :/

Viewing O as a class

Big-O can also be defined as a family or set of functions.

Big-O (alternative definition)

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



You can write $f(n) \in O\big(g(n)\big)$

Equivalent to "$f(n)$ is $O(g(n))$" or "$f(n) = O\big(g(n)\big)$"

The set of all functions that run in linear time (i.e. $O(n)$) is a "complexity class."

- We never write $O(5n)$ instead of $O(n)$ – they're the same thing!
- It's like writing $\frac{6}{2}$ instead of 3. It just looks weird.

3 Minutes

Practice

$5n + 3 \in O(n)$ True

$n \in O(5n + 3)$ True

$5n + 3 = O(n)$ True

$O(5n + 3) = O(n)$ True

$O(n^2) = O(n)$ False

$n^2 \in O(1)$ False

$n^2 \in O(n)$ False

$n^2 \in O(n^2)$ True

$n^2 \in O(n^3)$ True

$n^2 \in O(n^{100})$ True

Big-O

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

Big-Omega

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

Big-Theta

$f(n) \in \Theta(g(n))$ if
$f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

Examples

$4n^2 \in \Omega(1)$

true

$4n^2 \in \Omega(n)$

true

$4n^2 \in \Omega(n^2)$

true

$4n^2 \in \Omega(n^3)$

false

$4n^2 \in \Omega(n^4)$

false

$4n^2 \in O(1)$

false

$4n^2 \in O(n)$

false

$4n^2 \in O(n^2)$

true

$4n^2 \in O(n^3)$

true

$4n^2 \in O(n^4)$

true

Big-O

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

$$f(n) \leq c \cdot g(n)$$

Big-Omega

$f(n) \in \Omega(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$,

$$f(n) \geq c \cdot g(n)$$

Big-Theta

$f(n) \in \Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

Testing Your Code

Testing

Computers don’t make mistakes- people do!

“I’m almost done, I just need to make sure it works”
– Naive 14Xers

Software Test: a separate piece of code that exercises the code you are assessing by providing input to your code and finishes with an assertion of what the result should be.

1. Isolate - break your code into small modules
2. Build in increments - Make a plan from simplest to most complex cases
3. Test as you go - As your code grows, so should your tests

Types of Tests

Black Box

- Behavior only – ADT requirements
- From an outside point of view
- Does your code uphold its contracts with its users?
- Performance/efficiency

White Box

- Includes an understanding of the implementation
- Written by the author as they develop their code
- Break apart requirements into smaller steps
- “unit tests” break implementation into single assertions

What to test?

Expected behavior
- The main use case scenario
- Does your code do what it should given friendly conditions?

Forbidden Input
- What are all the ways the user can mess up?

Empty/Null
- Protect yourself!
- How do things get started?
- 0, -1, null, empty collections

Boundary/Edge Cases
- First items
- Last item
- Full collections

Scale
- Is there a difference between 10, 100, 1000, 10000 items?

Testing Strategies

You can't test everything

- Break inputs into categories
- What are the most important pieces of code?

Test behavior in combination

- Call multiple methods one after the other
- Call the same method multiple times

Trust no one!

- How can the user mess up?

If you messed up, someone else might

- Test the complex logic

5 Minutes

Thought Experiment

Discuss with your neighbors: Imagine you are writing an implementation of the List interface that stores integers in an Array. What are some ways you can assess your program's correctness in the following cases:

Expected Behavior
- Create a new list
- Add some amount of items to it
- Remove a couple of them

Forbidden Input
- Add a negative number
- Add duplicates
- Add extra large numbers

Empty/Null
- Call remove on an empty list
- Add to a null list
- Call size on an null list

Boundary/Edge Cases
- Add 1 item to an empty list
- Set an item at the front of the list
- Set an item at the back of the list

Scale
- Add 1000 items to the list
- Remove 100 items in a row
- Set the value of the same item 50 times

JUnit

JUnit: a testing framework that works with IDEs to give you a special GUI experience when testing your code

```
@Test
public void myTest() {
   Map<String, Integer> basicMap = new LinkedListDict<String, Integer>();
   basicMap.put("Kasey", 42);
   assertEquals(42, basicMap.get("Kasey"));
}
```

Assertions:

- `assertEquals(item1, item2)`
- `assertTrue(Boolean expression)`
- `assertFalse(bollean expression)`
- `assertNotNull(item)`

More: https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html