Time Complexity: **Big O Notation**

COMP128 Data Structures

Objectives

- Motivate the need for something like Big O notation
- Describe what Big O notation is
- Simplify Big O Expressions
- Define “time complexity” and “space complexity”
- Evaluate the time complexity of different algorithms using Big O notation

Imagine we have multiple implementations of the same method.

How can we determine which one is “best”?

Who cares?

- It’s important to have a precise vocabulary to talk about how our code performs
- Useful for discussing trade-offs between different approaches
- When your code slows down or crashes, identifying parts of the code that are inefficient can help us optimize our applications
- Less important: it comes up in software dev interviews

Example

Suppose we want to write a method that calculates the sum of all numbers from 1 up to and including some number n.

Example

Suppose we want to write a method that calculates the sum of all numbers from 1 up to and including some number n.

```
public int addUpTo(int n){
  int total = 0;
  for (int i = 1; i <= n; i++){
    total += i;
  }
  return total;
}
```

```
public int addUpTo(int n){
  return n * (n + 1) / 2.0;
}
```

Which one is better?

Example

$$\begin{array}{l}
\quad \text{addUpTo(int n)} = 1 + 2 + 3 \quad + \ldots + (n - 1) + n \\
+ \ \text{addUpTo(int n)} = n + (n - 1) + (n-2) + \ldots + 2 + 1 \\
\hline
2\ \text{addUpTo(int n)} = \underbrace{(n + 1) + (n + 1) + (n+1) + \ldots + (n+1) + (n+1)}_{\text{n copies}}
\end{array}$$

$$2\ \text{addUpTo(int n)} = n * (n+1)$$
$$\text{addUpTo(int n)} = n\ (n+1) / 2$$

What does better mean?

- Faster?
- Less memory/storage intensive?
- More readable?

Why not use timers?

Why not use timers?

- Different machines will record different times
- The same machine will record different times!
- For faster algorithms, speed measurements may not be precise enough
- Rather than counting seconds which are variable, let’s count the number of simple operations the computer has to perform

Counting Operations

```java
public int addUpTo(int n){
  return n * (n + 1) / 2.0;
}
```

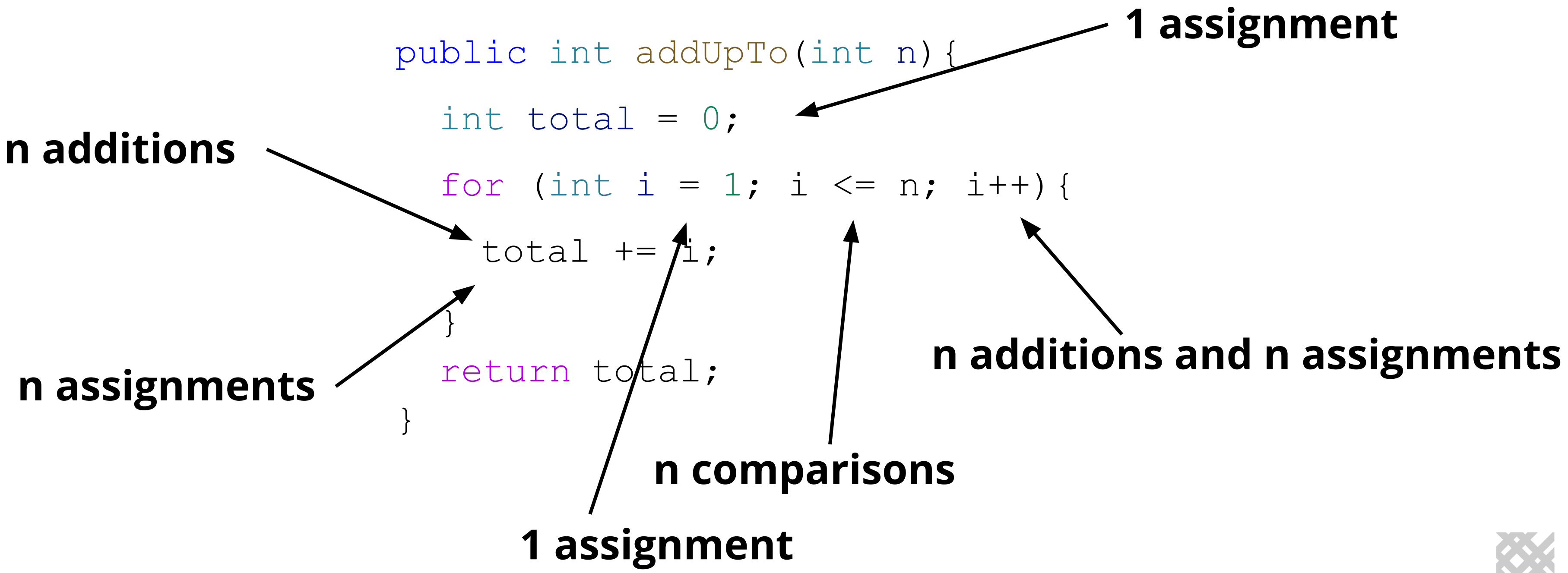
1 multiplication

1 addition

1 division

3 simple operations, regardless of the size of n

Counting Operations



```
public int addUpTo(int n){
  int total = 0;
  for (int i = 1; i <= n; i++){
    total += i;
  }
  return total;
}
```

Counting is hard

- Depending on what we count, the number of operations can be as low as 2n or as high as 5n + 2
- But regardless of the exact number, the number of operations grows roughly proportionally with n
- If n doubles, the number of operations will also roughly double.

Big O

Big O notation is a way to formalize fuzzy counting

It allows us to talk formally about how the runtime of an algorithm grows as the inputs grow

We won’t care about the details only the trends as the size of n increases

Big O Definition

We say that an algorithm is O(g(n)) if the number of simple operations the computer has to do is eventually less than a constant times g(n), as n increases

- g(n) could be linear, e.g. g(n) = n
- g(n) could be quadratic, e.g. g(n) = n2
- g(n) could be constant, e.g. g(n) = 1
- g(n) could be something entirely different

Example

```
public int addUpTo(int n){
  return n * (n + 1) / 2.0;
}
```

Always 3 operations
O(1)

Example

```
public int addUpTo(int n){

  return n * (n + 1) / 2.0;

}

public int addUpTo(int n){   int

  total = 0;

  for (int i = 1; i <= n; i++){

    total += i;

  }
  return total;
}
```

Always 3 operations
O(1)

Number of operations is (eventually) bounded by a multiple of n
O(n)

Practice Example

```
public void countUpAndDown(int n){
    System.out.println("Going up!");
    for(int i=0; i < n; i++){
        System.out.println(i);
    }
    System.out.println("Going down!");
    for(int i = n-1; i >= 0; i--){
        System.out.println(i);
    }
}
```

Practice Example

```
public void countUpAndDown(int n){
    System.out.println("Going up!");
    for(int i=0; i < n; i++){
      System.out.println(i);
    }

    System.out.println("Going down!");
    for(int i = n-1; i >= 0; i--){
      System.out.println(i);
    }

}
```

O(n)

O(n)

Number of operations is (eventually) bounded by a multiple of n

O(n)

Practice Example

```
public void printAllPairs(int n){
    for(int i=0; i < n; i++){
      for(int j=0; j < n; j++){

        System.out.print(i + ", ");
        System.out.println(j);

        }
      }
  }
```

Practice Example

```
public void printAllPairs(int n){
    for(int i=0; i < n; i++){
      for(int j=0; j < n; j++){
        System.out.print(i + ", ");
        System.out.println(j);
      }
    }
}
```

O(n)

O(n)

O(n) operation inside of a O(n) operation **$O(n*n) \longrightarrow O(n^2)$**

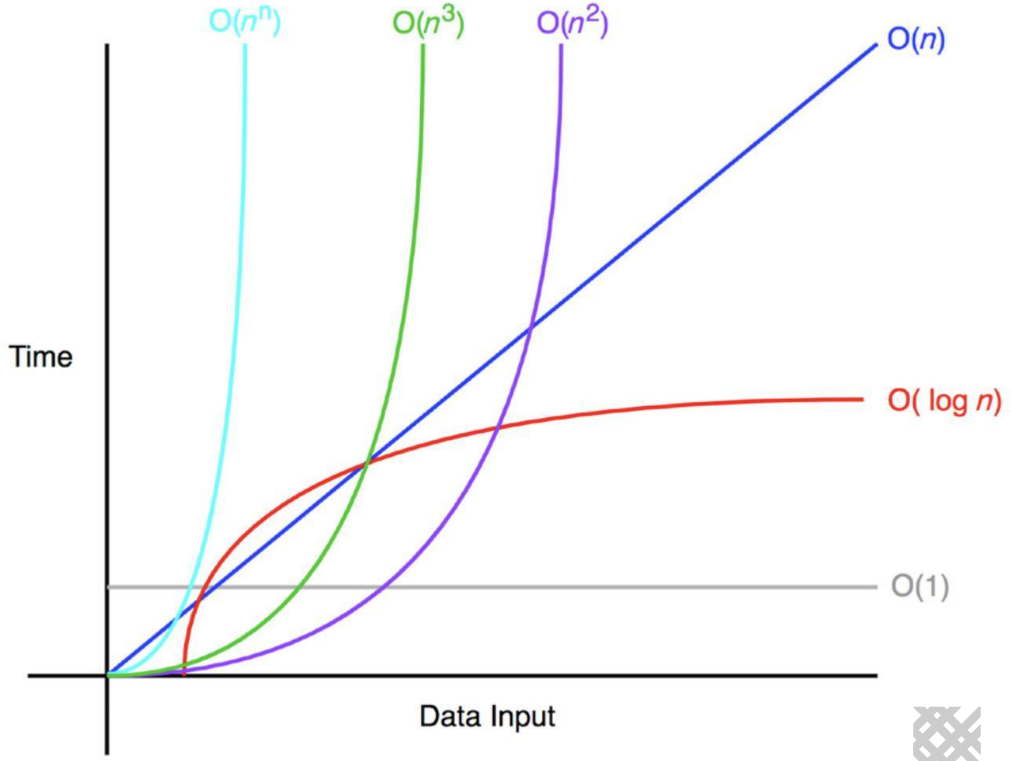
Simplifying Big O Expressions

Constants don’t matter

- O(2n) —> O(n)
- O(500) —> O(1)
- O(13n2) —> O(n2)

Smaller terms don’t matter

- O(n + 10) —> O(n)
- O(1000n + 50) —> O(n)
- O(n2 + 5n + 8) —> O(n2)



Simplifying Big O Expressions

Some helpful rules of thumb:

- Arithmetic operations are constant
- Variable assignment is constant
- Accessing elements in an array by index is constant
- In a loop, the complexity is the length of the loop times the complexity of whatever happens inside the loop
    - **Be careful with this! A loop isn’t always O(n)!**

Loop Examples

```
public void printAtLeast5(int n){
    for(int i=1; i <= Math.max(5, n); i++){
        System.out.println(i);
      }
```

O(n)

```
public void printAtMost5(int n){
    for(int i=1; i <= Math.min(5, n); i++){
        System.out.println(i);
      }
  }
```

O(1)

```
 public void printDivideBy2(int n){
     for(int i = n-1; i >= 0; i /= 2){
       System.out.println(i);
      }
   }
```

O(logn)

log == $\log_2$ for most of CS

A logarithm of a number roughly measures the number of times you can divide that number by 2 before you get a value that's less than or equal to one.

In-class Activity

Big O Activity