Announcements

1. HW2: How’s it going?
   A. I haven’t even read it
   B. I’ve read it, but haven’t started coding
   C. I’ve just started coding
   D. I’m well on my way
   E. I’m done!
Review quiz

Problems 1-4 in class Node<T> with member variables
- Node<T> next
- T data

Problem 5 in class SingleLinkedList<E> with member variables
- Node<E> head
- int size
Today’s Topics

1. Running time: How long does a program take to run
2. Best case and worst case analysis
3. Big-O, Big-Omega, Big-Theta
You have an unsorted array of integers, size 20.

You want to find the index where 15 is stored.

How many places do you have to look?

A. 1  
B. 10
C. 15
D. 20
E. Other/none/more than one
You have an unsorted array of integers, size 20: But I’m not telling you what they are

You want to find the index where 15 is stored

How many places do you have to look in the **BEST** case?

A. 1
B. 10
C. 15
D. 20
E. Other/none/more than one
You have an unsorted array of integers, size 20

You want to find the index where 15 is stored

How many places do you have to look in the WORST case?

A. 1
B. 10
C. 15
D. 20
E. Other/none/more than one

We also sometimes consider average case, but that’s somewhat tricky
Running time: What version of the problem are you analyzing

- One part of figuring out how long a program takes to run is figuring out how “lucky” you got in your input.
  - You might get lucky (best case), and require the least amount of time possible
  - You might get unlucky (worst case) and require the most amount of time possible
  - Or you might want to know “on average” (average case) if you are neither lucky or unlucky, how long does an algorithm take.

Almost always, what we care about is the WORST CASE or the AVERAGE CASE. Best case is usually not that interesting.

In CSE 12 when we do analysis, we are doing WORST CASE analysis unless otherwise specified.
Analyzing the worst case

```java
boolean find( int[] theList, int toFind ) {
    for ( int i = 0; i < theList.length; i++ ) {
        if ( theList[i] == toFind )
            return true;
    }
    return false;
}
```

How many instructions do you have to execute to find out if the element is in the list in the worst case, if \( n \) represents the length of the list?
Analyzing the worst case

```java
boolean find( int[] theList, int toFind ) {
    for ( int i = 0; i < theList.length; i++ ) {
        if ( theList[i] == toFind ) {
            return true;
        }
    }
    return false;
}
```

```java
boolean slowFind( int[] theList, int toFind ) {
    for ( int i = 0; i < theList.length; i++ ) {
        System.out.println( "Looking for " + toFind );
        if ( theList[i] == toFind ) {
            return true;
        }
    }
    return false;
}
```

Which method is faster?
A. `find`
B. `find2`
C. They are about the same
Analyzing the worst case

```java
boolean find( int[] theList, int toFind ) {
    for ( int i = 0; i < theList.length; i++ ) {
        if ( theList[i] == toFind )
            return true;
    }
    return false;
}

boolean fastFind( int[] theList, int toFind ) {
    return false;
}
```

Which method is faster?
A. find
B. fastFind
C. They are about the same
2n vs n vs 1

- slow
- find
- fast
- find

same order of growth

p(n)

2n

n

n
\( f_2 \text{ is} \ O(f_1) \)

A. TRUE  
B. FALSE

*Is there a \( c, n_0 \) such that \( f_2(n) \leq c \times f_1(n) \) for \( n > n_0 \)?

Why or why not?

* You can't actually tell if you don't know the function, because it could do something crazy just off the graph, but we'll assume it doesn't.
Big-O

We say a function $f(n)$ is “big-O” of another function $g(n)$, and write $f(n) = O(g(n))$, if there are positive constants $c$ and $n_0$ such that:

- $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

In other words, for large $n$, can you multiply $g(n)$ by a constant and have it always be bigger than or equal to $f(n)$.
f(n) = \(O(g(n))\), if there are positive constants \(c\) and \(n_0\) such that \(f(n) \leq c \times g(n)\) for all \(n \geq n_0\).

- Obviously \(f_2 = O(f_1)\) because \(f_1 > f_2\) (after about \(n=10\), so we set \(n_0 = 10\))
  - \(f_1\) is clearly an upper bound on \(f_2\) and that’s what big-O is all about
\( f_1 \) is \( \mathcal{O}(f_2) \)

**A.** TRUE

**B.** FALSE

Why or why not?

In other words, for large \( n \), can you multiply \( f_2 \) by a constant and have it always be bigger than \( f_1 \)?
f(n) = O(g(n)), if there are positive constants c and n₀ such that f(n) ≤ c * g(n) for all n ≥ n₀.

- Obviously f₂ = O(f₁) because f₁ > f₂ (after about n=10, so we set n₀ = 10)
  - f₁ is clearly an upper bound on f₂ and that’s what big-O is all about
- But f₁ = O(f₂) as well!
  - We just have to use the “c” to adjust so f₂ that it moves above f₁
f(n) = O(g(n)), if there are positive constants c and n₀ such that f(n) ≤ c * g(n) for all n ≥ n₀.

- Obviously f₂ = O(f₁) because f₁ > f₂ (after about n=10, so we set n₀ = 10)
  - f₁ is clearly an upper bound on f₂ and that’s what big-O is all about
- But f₁ = O(f₂) as well!
  - We just have to use the “c” to adjust so f₂ that it moves above f₁
Next time

- More running time analysis, and calculating it from code