CSE 12 – Basic Data Structures

[Slides borrowed/adapted from slides by Cynthia Lee]

Announcements

1. HW3 is out

Today's Topics

1. Running time: How long does a program take to run
2. Best case and worst case ananlysis
3. Big-O, Big-Omega, Big-Theta

Lists: One application

```
List<Songs> playlist
```

Lists: One application

```
List<Songs> playlist
```

Imagine that this list is implemented as an array and that there are currently 20 songs in my playlist. How many places do I have to look to determine whether “All About that Bass” is in my playlist?

A. 1
B. 3
C. 10
D. 20
E. Other

Lists: Running time

```
List<Songs> playlist
```

Imagine that this list is implemented as an array and that there are currently 20 songs in my playlist, but now you can't see what any of them are.
How many places do I have to look to determine whether "Riptide" is in my playlist in the BEST case?

A. 1
B. 10
C. 15
D. 20
E. Other

Lists: Running time

```
List<Songs> playlist
```

Imagine that this list is implemented as an array and that there are currently 20 songs in my playlist, but now you can't see what any of them are.
How many places do I have to look to determine whether "Riptide" is in my playlist in the WORST case?

A. 1
B. 10
C. 15
D. 20
E. Other

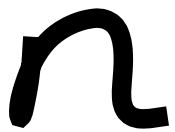
Running time: What version of the problem are you analyzing

- 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



```
boolean find( String[] theList, String toFind ) {
  for ( int i = 0; i < theList.length; i++ ) {
    if ( theList[i].equals( 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

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

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

Which method is faster?

A. find
B. find2
C. They are about the same

Analyzing the worst case

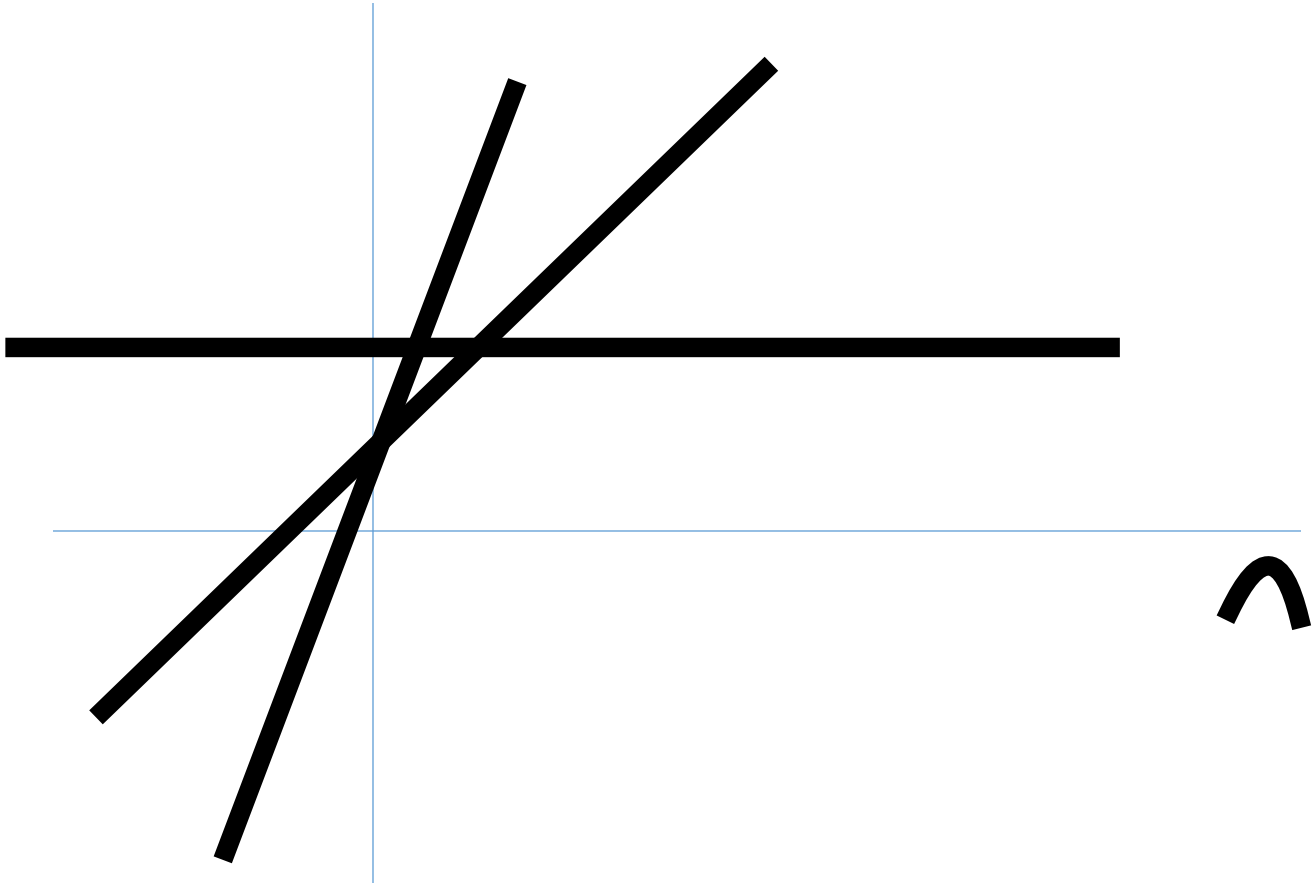
```
boolean find( String[] theList, String toFind ) {
  for ( int i = 0; i < theList.length; i++ ) {
    if ( theList[i].equals(toFind) )
      return true;
  }
  return false;
}

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

Which method is faster?

A. find
B. find2
C. They are about the same

2n vs n vs 1



(Play)lists typically have 1000s (or more!) elements, so we only care about very large n.

find and slowFind are “more similar” than find and fastFind.

We use Big-O notation to represent “classes” of running time—e.g. those that have similar behavior as N gets large, give or take a constant factor.

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:

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

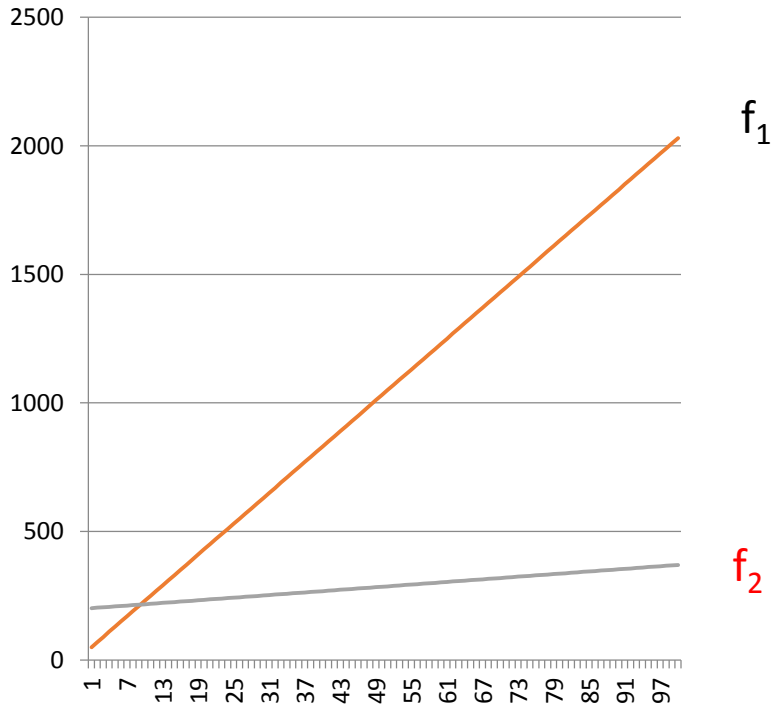
In other words, for large n, can you multiply g(n) by a constant and have it always be bigger than or equal to f(n)

f_2 is* $O(f_1)$

A. TRUE

B. FALSE

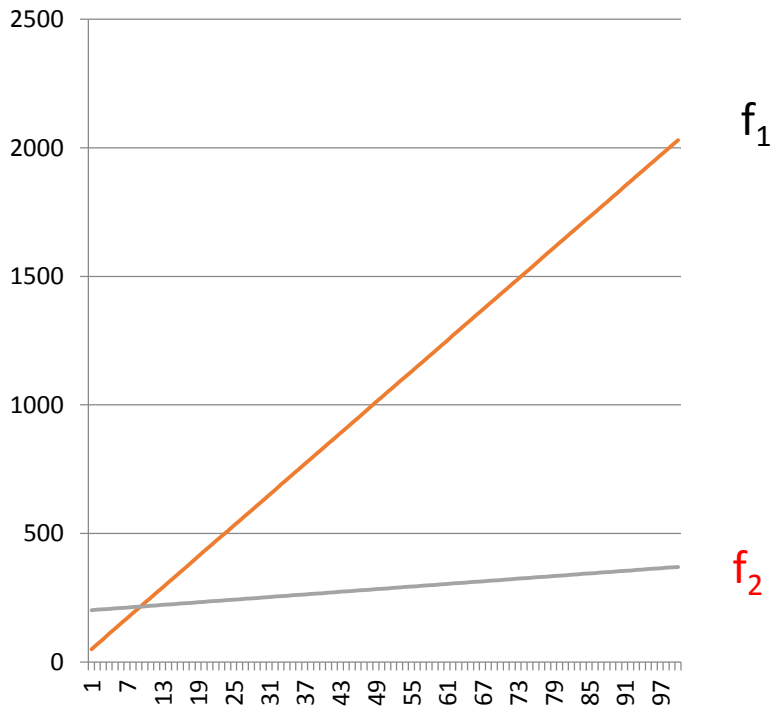
Why or why not?



* 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.

f(n) = O(g(n)), if there are positive constants c and n_0 such that f(n) ≤ **c** * g(n) for all n ≥ 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



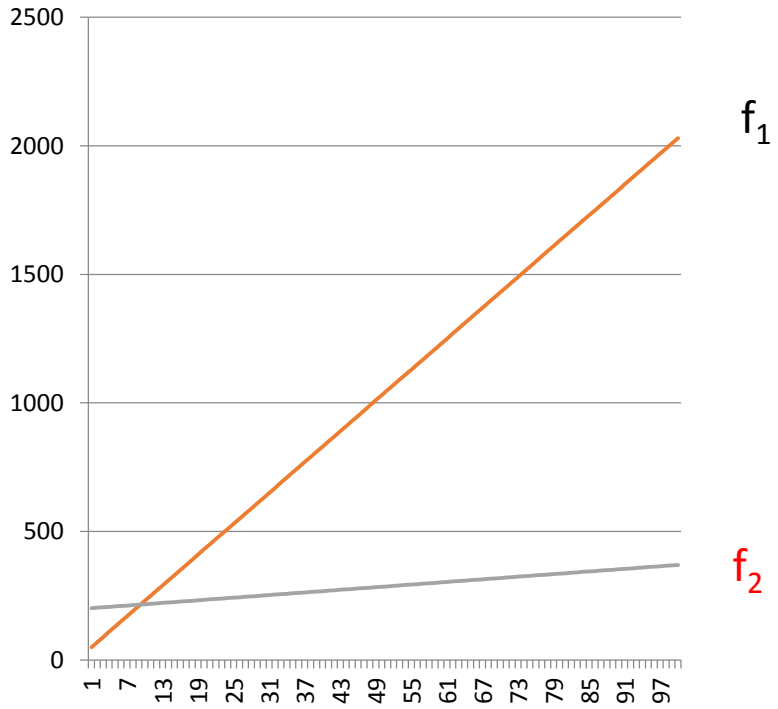
f_1 is $O(f_2)$

A. TRUE

B. FALSE

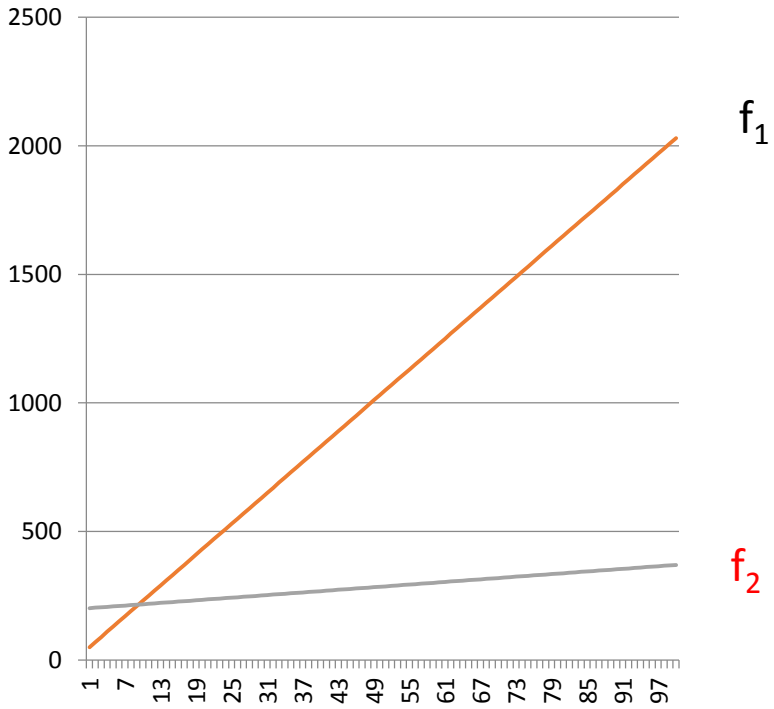
Why or why not?

In other words, for large n, can you multiply f_2 by a constant and have it always be bigger than f_1 for large enough n?



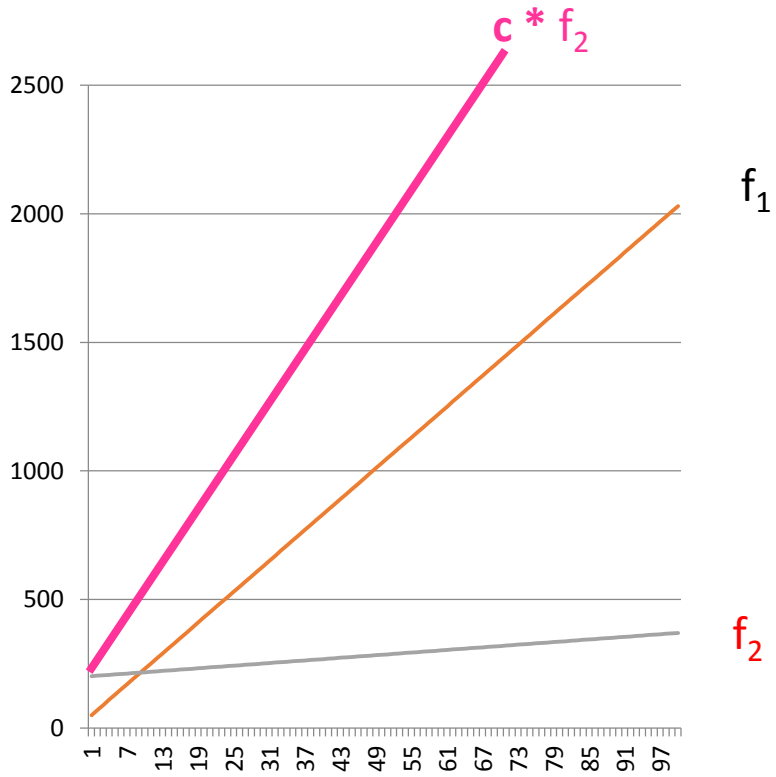
f(n) = O(g(n)), if there are positive constants c and n_0 such that f(n) ≤ **c** * g(n) for all n ≥ n_0.

- 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
- But $f_1 = O(f_2)$ as well!
  - We just have to use the "**c**" to adjust so f_2 that it moves above f_1



f(n) = O(g(n)), if there are positive constants c and n_0 such that f(n) ≤ **c** * g(n) for all n ≥ 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
- But $f_1 = O(f_2)$ as well!
  - We just have to use the "**c**" to adjust so f_2 that it moves above f_1



Shortcuts for calculating

Big-O analysis starting with a function characterizing the growth in cost of the algorithm

Let $f(n) = 3 \log_2 n + 4 n \log_2 n + n$

- Which of the following is true?

A. $f(n) = O(\log_2 n)$
B. $f(n) = O(n \log_2 n)$
C. $f(n) = O(n^2)$
D. $f(n) = O(n)$
E. Other/none/more

Let $f(n) = 546 + 34n + 2n^2$

- Which of the following is true?

A. $f(n) = O(2^n)$
B. $f(n) = O(n^2)$
C. $f(n) = O(n)$
D. $f(n) = O(n^3)$
E. Other/none/more

Let $f(n) = 2^n + 14n^2 + 4n^3$

- Which of the following is true? false

A. $f(n) = O(2^n)$

B. $f(n) = O(n^2)$

C. $f(n) = O(n)$

D. $f(n) = O(n^3)$

E. Other/none/more

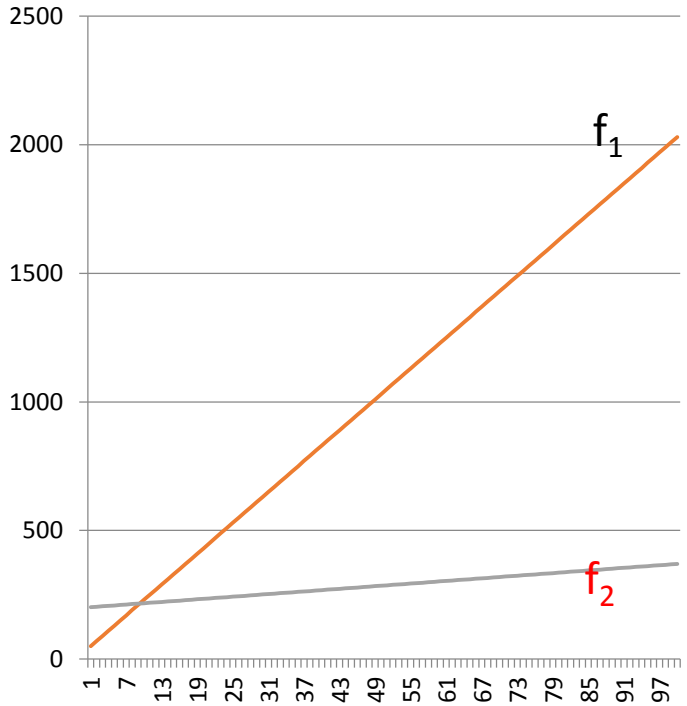
Let f(n) = 100

- Which of the following is true?

A. $f(n) = O(2^n)$
B. $f(n) = O(n^2)$
C. $f(n) = O(n)$
D. $f(n) = O(n^{100})$
E. Other/none/more

Common Big-O confusions when trying to argue that f_2 is $O(f_1)$:

- What if we multiply f_2 by a large constant, so that $c*f_2$ is larger than f_1 ? Doesn't that mean that f_2 is not $O(f_1)$?
  No, because we get to control the constants to our advantage, and only on f1.
- What about when n is less than 10? Isn't f2 larger than f1?
  Remember, we get to pick our n0, and only consider n larger than n0.



O is an upper bound
Ω is a *lower bound*

We say a function f(n) is **"big-omega"** of another function g(n), and write f(n) = **Ω**(g(n)), if there are positive constants c and n_0 such that:

- $f(n) \geq c\, g(n)$ 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 smaller than or equal to f(n)

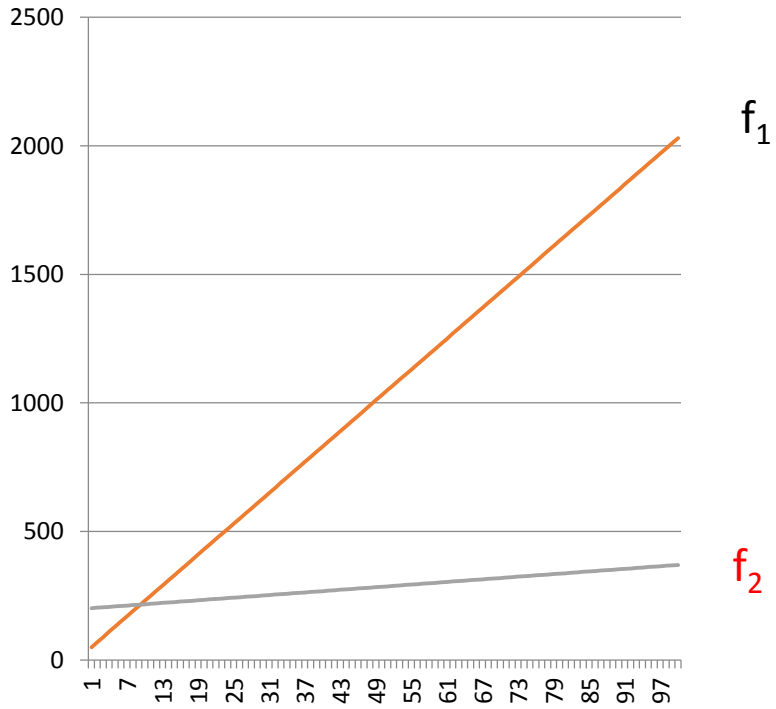
f_2 is $\Omega(f_1)$

A. TRUE

B. FALSE

Why or why not?

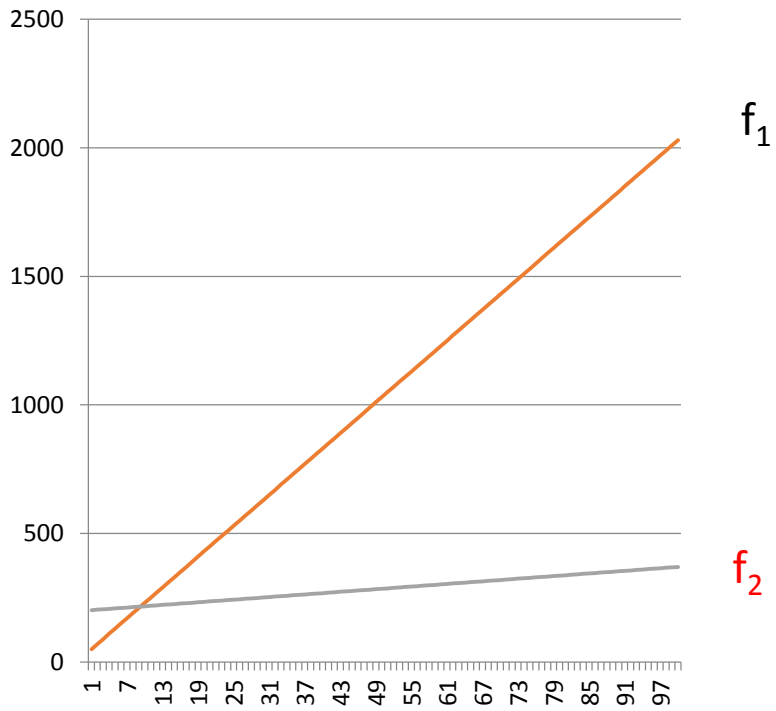
In other words, for large n, can you multiply f_1 by a positive constant and have it always be smaller than f_2



f(n) = **O**(g(n)), if there are positive constants c and n_0 such that f(n) **≤** **c** * g(n) for all n ≥ n_0.

f(n) = **Ω**(g(n)), if there are positive constants c and n_0 such that f(n) **≥** **c** * g(n) for all n ≥ n_0.

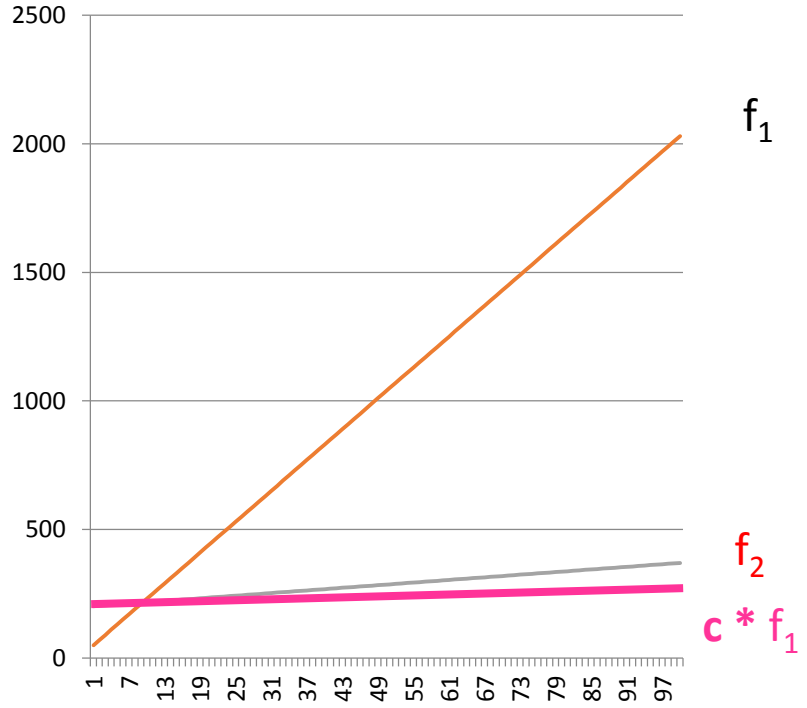
- Obviously $f_1 = \Omega(f_2)$ because $f_1 > f_2$ (after about n=10, so we set $n_0 = 10$)
  - f_2 is clearly a ***lower bound*** on f_1 and that's what big-Ω is all about
- But $f_2 = \Omega(f_1)$ as well!
  - We just have to use the "**c**" to adjust so f_1 that it moves below f_2



f(n) = **O**(g(n)), if there are positive constants c and n_0 such that f(n) **≤** **c** * g(n) for all n ≥ n_0.

f(n) = **Ω**(g(n)), if there are positive constants c and n_0 such that f(n) **≥** **c** * g(n) for all n ≥ n_0.

- Obviously $f_1 = \Omega(f_2)$ because $f_1 > f_2$ (after about n=10, so we set $n_0 = 10$)
  - f_2 is clearly a ***lower bound*** on f_1 and that's what big-Ω is all about
- But $f_2 = \Omega(f_1)$ as well!
  - We just have to use the "**c**" to adjust so f_1 that it moves below f_2



f(n) = **O**(g(n)), if there are positive constants c and n_0 such that f(n) **≤** **c** * g(n) for all n ≥ n_0.

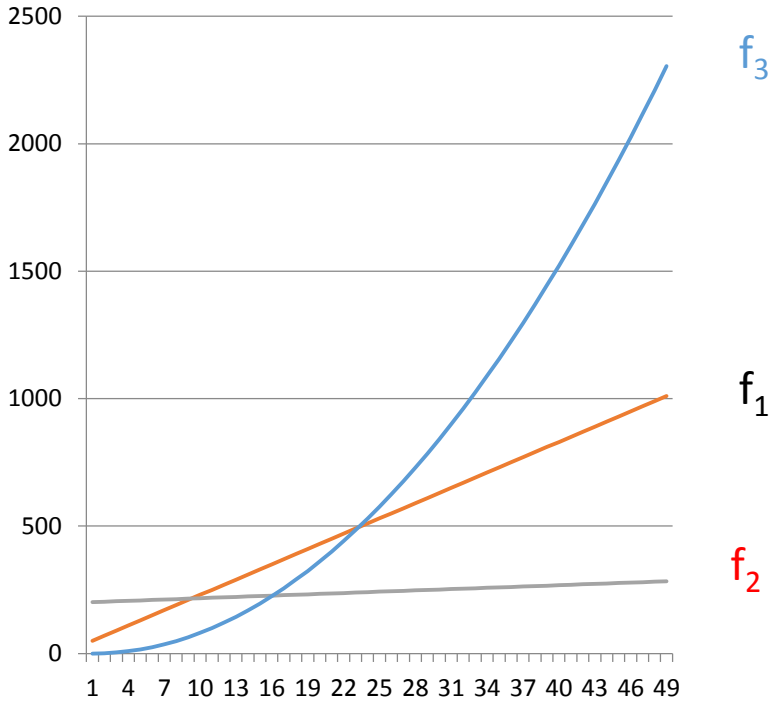
f(n) = **Ω**(g(n)), if there are positive constants c and n_0 such that f(n) **≥** **c** * g(n) for all n ≥ n_0.

f_1 is $O(f_3)$

A. TRUE

B. FALSE

Why or why not?



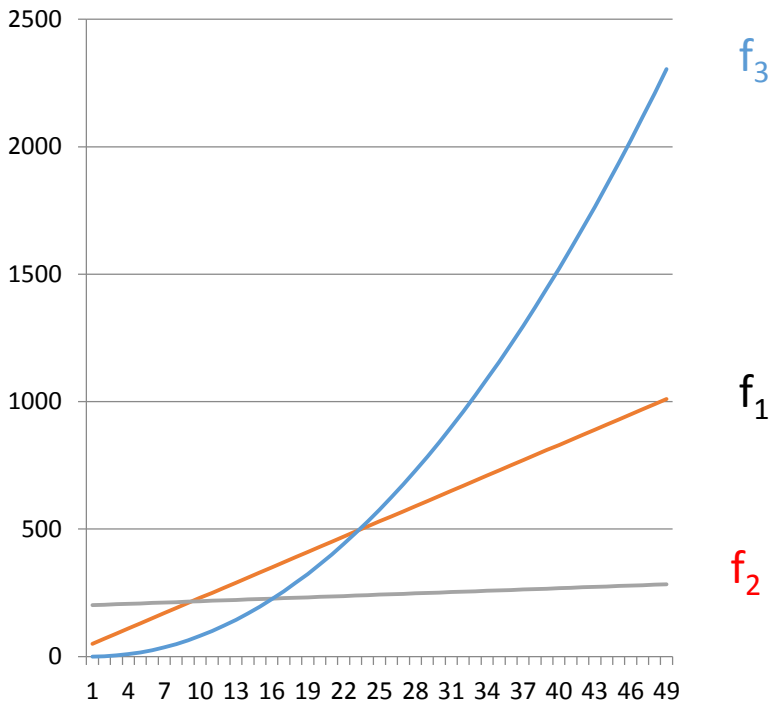
f(n) = **O**(g(n)), if there are positive constants c and n_0 such that f(n) **≤ c** * g(n) for all n ≥ n_0.
f(n) = **Ω**(g(n)), if there are positive constants c and n_0 such that f(n) **≥ c** * g(n) for all n ≥ n_0.

f_3 is $O(f_1)$

A. TRUE

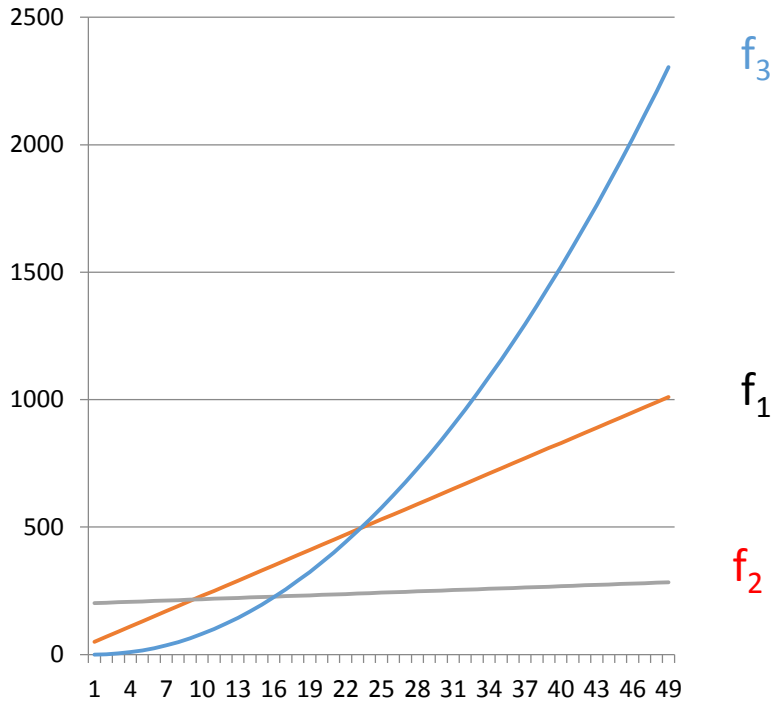
B. FALSE

Why or why not?



$f_1 = O(f_3)$ *but* $f_3 \neq O(f_1)$

- There is no way to pick a c that would make an O(n) function (f_1) stay above an $O(n^2)$ function (f_3).



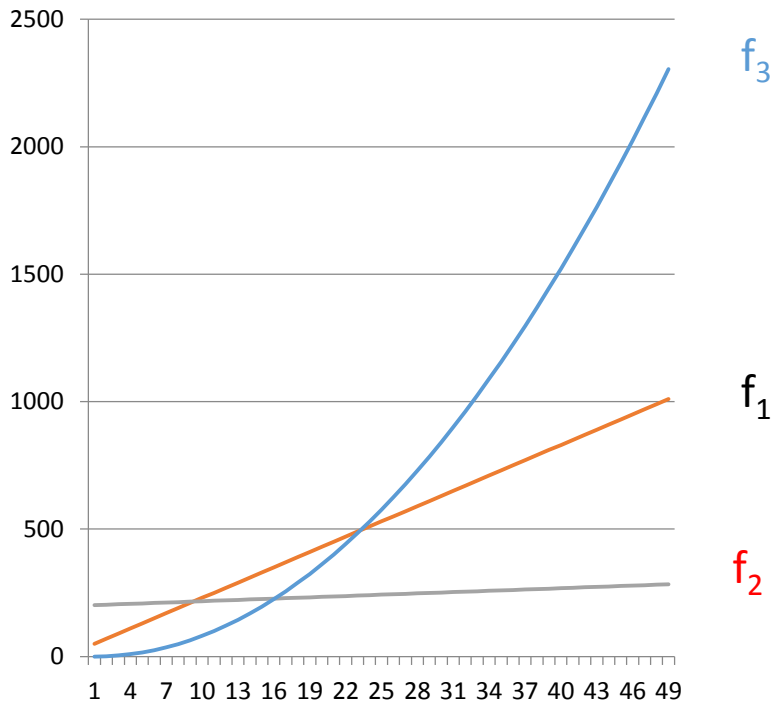
f(n) = **O**(g(n)), if there are positive constants c and n_0 such that f(n) **≤** **c** * g(n) for all n ≥ n_0.
f(n) = **Ω**(g(n)), if there are positive constants c and n_0 such that f(n) **≥** **c** * g(n) for all n ≥ n_0.

f_3 is $\Omega(f_1)$

A. TRUE

B. FALSE

Why or why not?



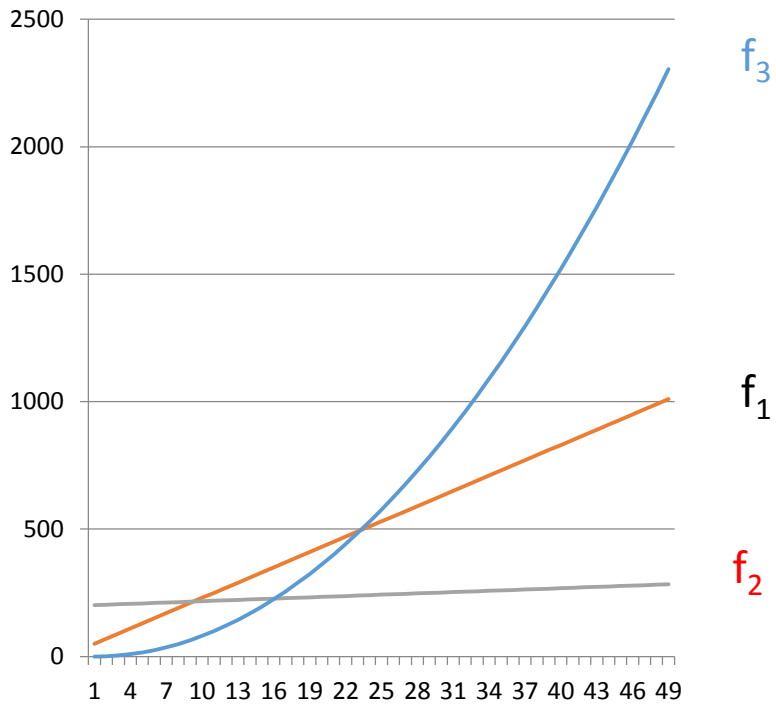
f(n) = **O**(g(n)), if there are positive constants c and n_0 such that f(n) **≤** **c** * g(n) for all n ≥ n_0.
f(n) = **Ω**(g(n)), if there are positive constants c and n_0 such that f(n) **≥** **c** * g(n) for all n ≥ n_0.

f_1 is $\Omega(f_3)$

A. TRUE

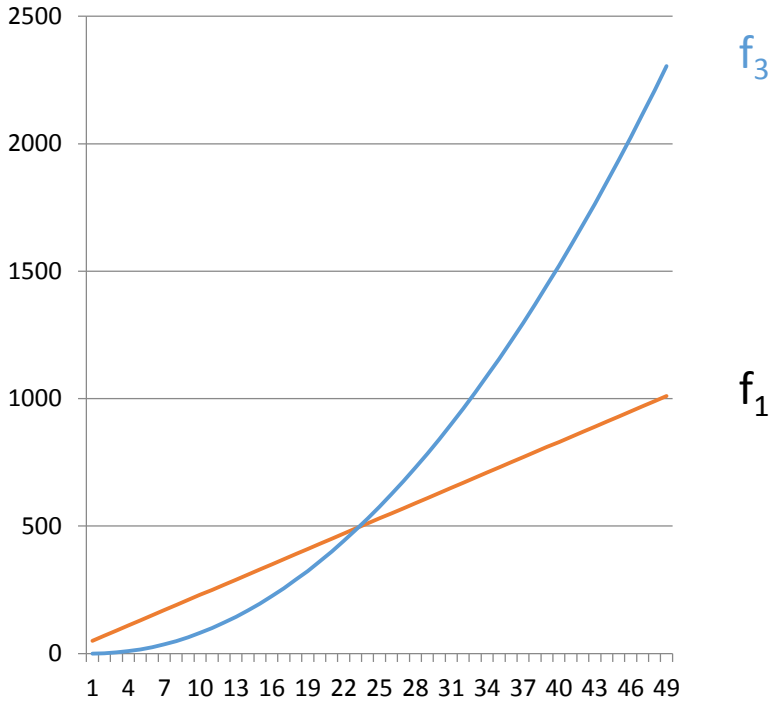
B. FALSE

Why or why not?



$f_3 = \Omega(f_1)$ *but* $f_1 \neq \Omega(f_3)$

- There is no way to pick a c that would make an $O(n^2)$ function (f_3) stay below an O(n) function (f_1).



Summary

Big-O

- **Upper bound** on a function
- f(n) = O(g(n)) means that we can expect f(n) will always be **under** the bound g(n)
  - But we don't count n up to some starting point n_0
  - And we can "cheat" a little bit by moving g(n) up by multiplying by some constant c

Big-Ω

- **Lower bound** on a function
- f(n) = Ω(g(n)) means that we can expect f(n) will always be **over** the bound g(n)
  - But we don't count n up to some starting point n_0
  - And we can "cheat" a little bit by moving g(n) down by multiplying by some constant c

Big-θ

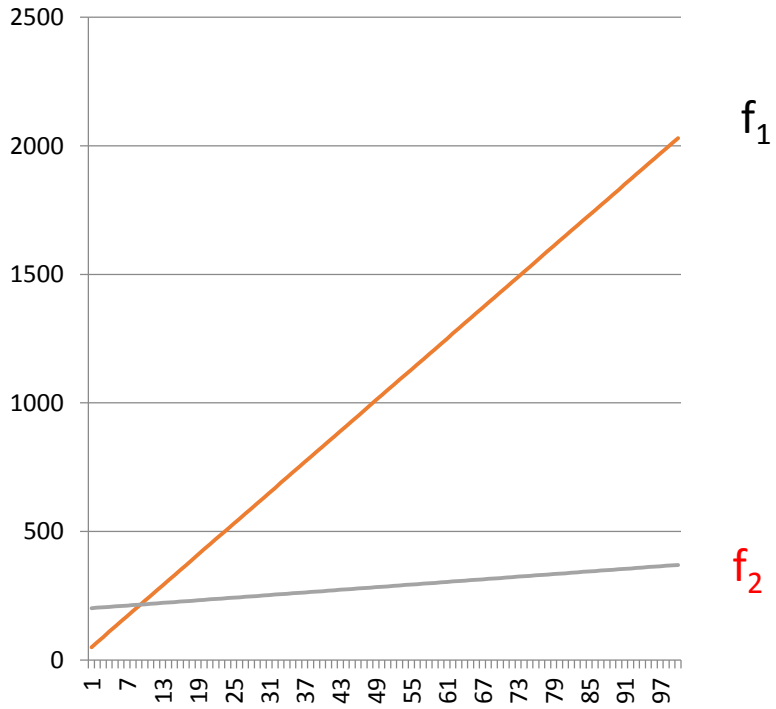
- **Tight bound** on a function.
- If f(n) = O(g(n)) *and* f(n) = Ω(g(n)), then f(n) = θ(g(n)).
- Basically it means that f(n) and g(n) are interchangeable
- Examples:
  - $3n+20 = \theta(10n+7)$
  - $5n^2 + 50n + 3 = \theta(5n^2 + 100)$

f_1 is $\Theta(f_2)$

A. TRUE

B. FALSE

Why or why not?



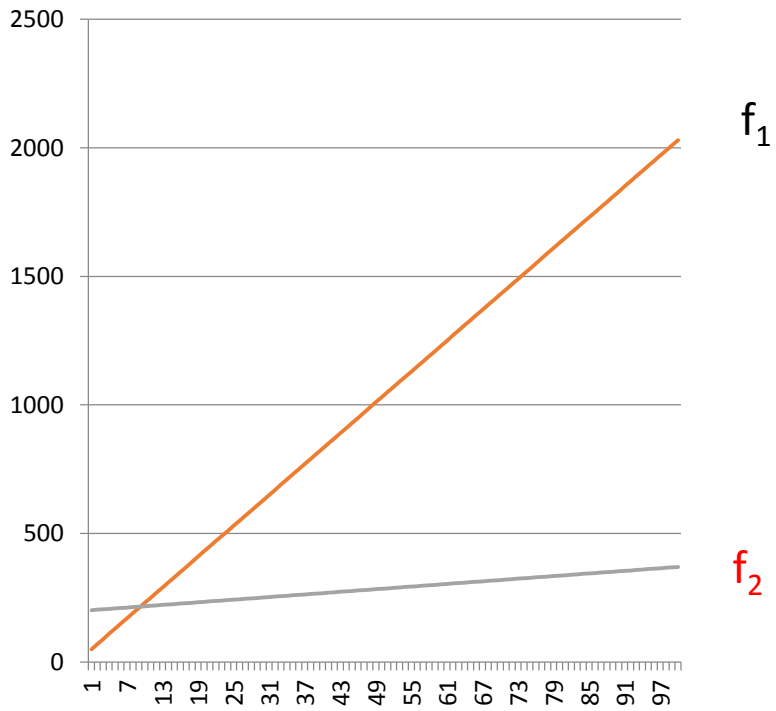
f_1 is $\Theta(f_2)$

A. TRUE

B. FALSE

Why or why not?

Since f_1 is $O(f_2)$ and $\Omega(f_2)$, it is also $\Theta(f_2)$ (this is the definition of big-Theta)

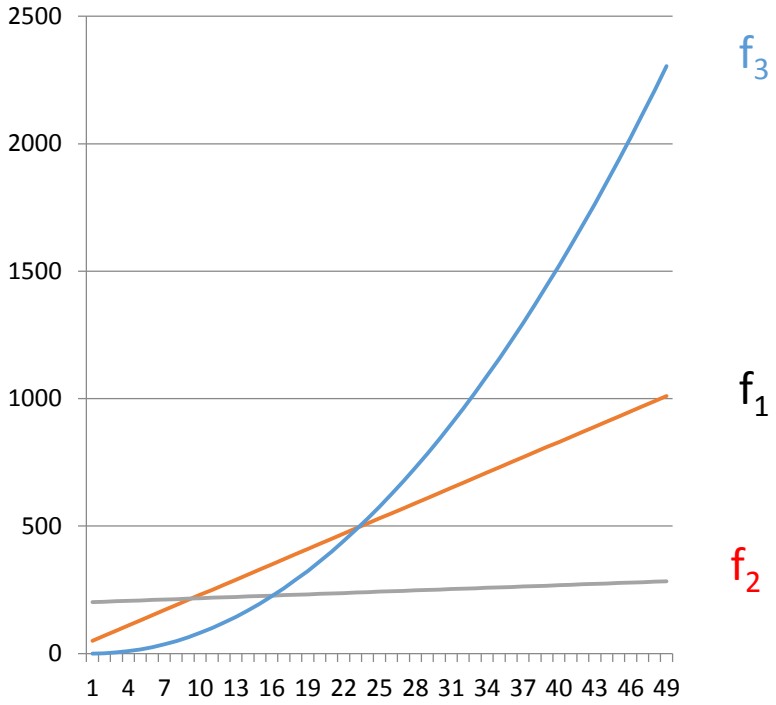


f_1 is $\Theta(f_3)$

A. TRUE

B. FALSE

Why or why not?



Big-θ and sloppy usage

- Sometimes people say, “This algorithm is $O(n^2)$” when it would be more precise to say that it is $\theta(n^2)$
  - They are intending to give a tight bound, but use the looser “big-O” term instead of the “big-θ” term that actually means tight bound
  - Not wrong, but not as precise
- I don’t know why, this is just a cultural thing you will encounter among computer scientists

Extracting time cost from example code

Algorithm analysis starting with the algorithm

Steps for calculating the Big O (Theta, Omega) bound on code

- Count the number of instructions in your code as precisely as possible as a function of n, which represents the size of your input (e.g. the length of the array). This is your f(n).
  - Make sure you know if you are counting best case, worst case or average case – could be any of these!
- Simplify your f(n) to find a simple g(n) such that f(n) = O(g(n)) (or Ω(g(n) or Θ(g(n))

A student has counted how many times we perform each line of code

Is the count 3n+5:

A. the best case?
B. the worst case?
C. the average case?
D. Other/none/more

	Statements	Cost
1	float findAvg (int []grades){	
2	float sum = 0;	1
3	int count = 0;	1
4	while (count < grades.length) {	$n + 1$
5	sum += grades[count];	n
6	count++;	n
7	}	
8	if (grades.length > 0)	1
9	return sum / grades.length;	
10	else	1
11	return 0.0f;	
12	}	
ALL		**3n+5**

Next time

- More code calculations of Big-O/etc
- Benchmarking code