CSE 12 – Basic Data Structures

[Slides borrowed/adapted from slides by Marina Langlois and Christine Alvarado]

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. fastFind
C. They are about the same

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, unless we can prove it's slow!

In CSE 12 when we do analysis, we are doing **WORST CASE** analysis unless otherwise specified.

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:

- $f(n) \le c\, g(n)$ for all $n \ge n_0$.

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)

n is the "size of your problem"

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

1. Identify the assumptions you're making about input and the case you're studying – best, worst, average?

2. Count the number of instructions in your code (or algorithm) as precisely as possible as a function the size of your input (e.g. the length of the array). Call this f(n)

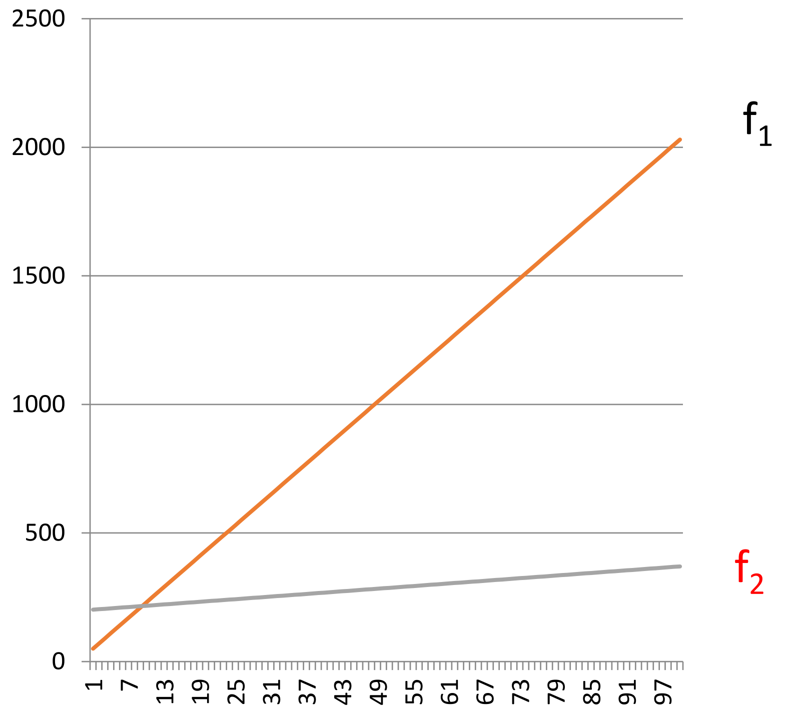
3. Summarize your findings by relating f(n) to a simpler g(n) such that f(n) = O(g(n))

f_2 is* $O(f_1)$

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.

A. TRUE
B. FALSE

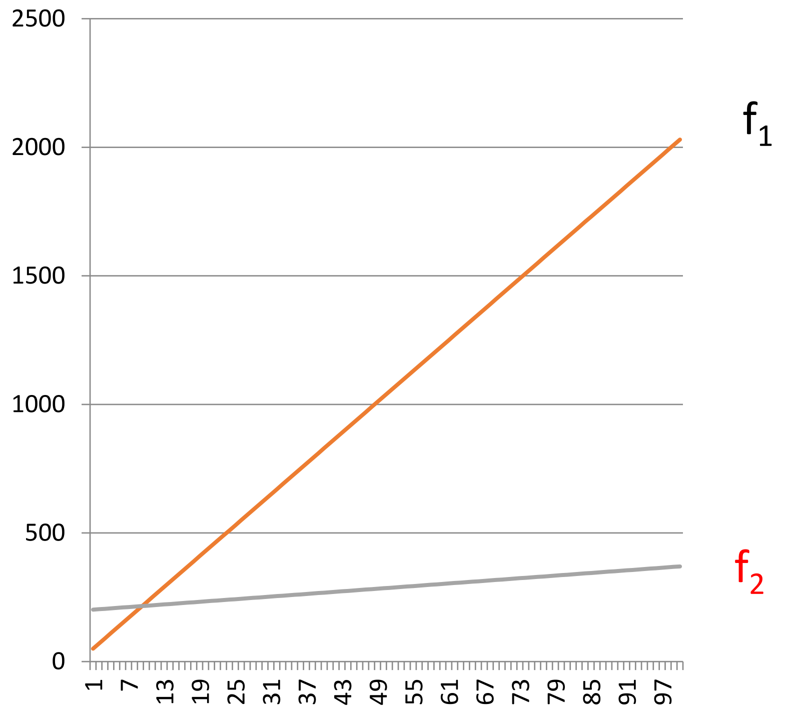
Why or why not?



* 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) is 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_2 is 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



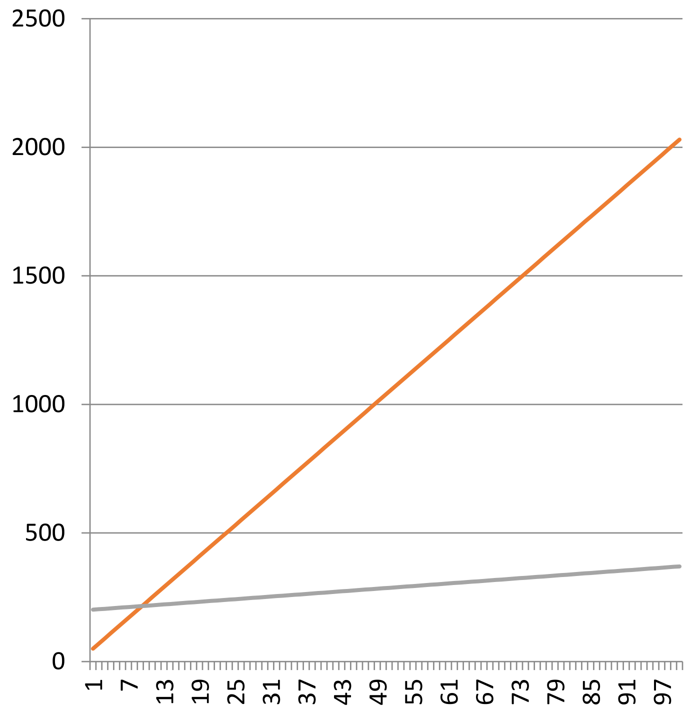
f_1 is $O(f_2)$

$f(n) = \mathbf{O}(g(n))$, if there are positive constants c and n_0 such that $f(n) \leq \mathbf{c} * g(n)$ for all $n \geq n_0$.

A. TRUE
B. FALSE

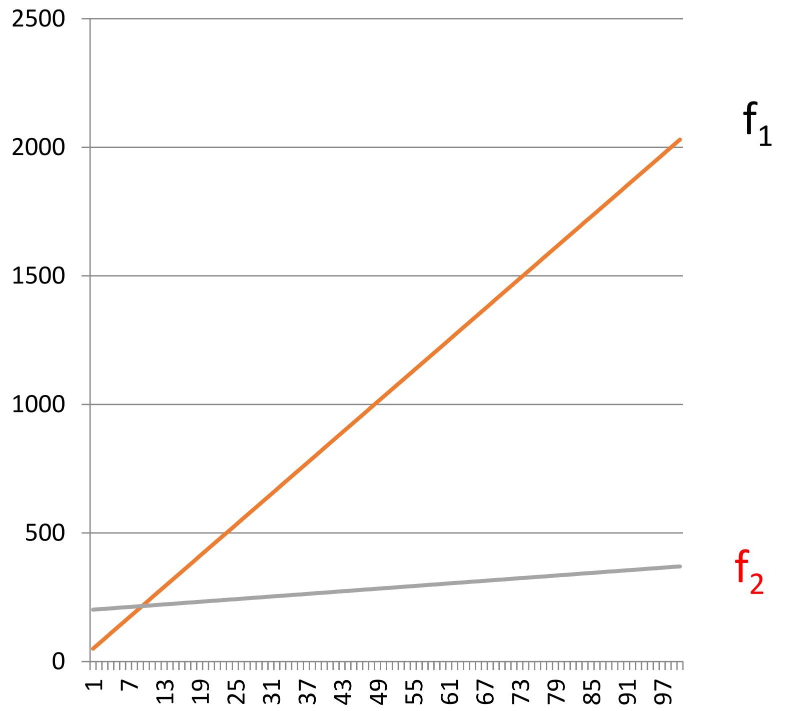
Why or why not?

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

- f_2 is $O(f_1)$ 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



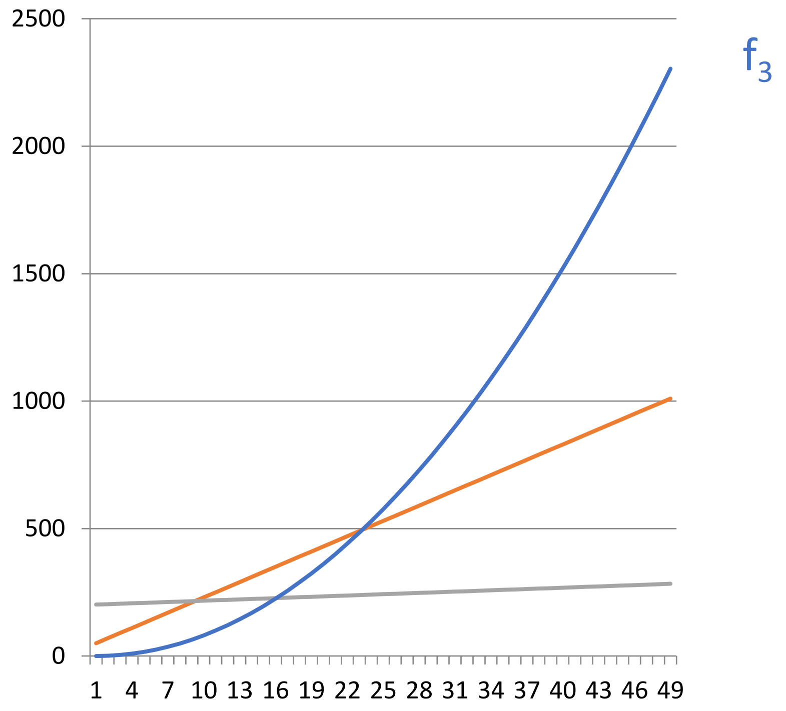
f(n) = **O**(g(n)), if there are positive constants c and n_0 such that f(n) ≤ **c** * g(n) for all n ≥ n_0.

f_1 is $O(f_3)$

A. TRUE

B. FALSE

Why or why not?



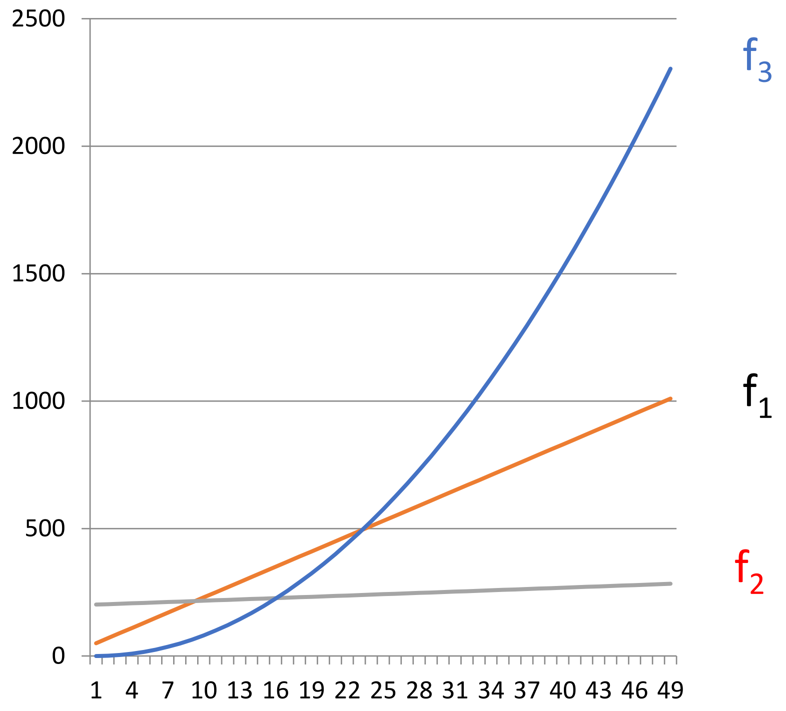
$f(n) = \mathbf{O}(g(n))$, if there are positive constants c and n_0 such that $f(n) \leq c * g(n)$ for all $n \geq n_0$.

f_3 is $O(f_1)$

A. TRUE

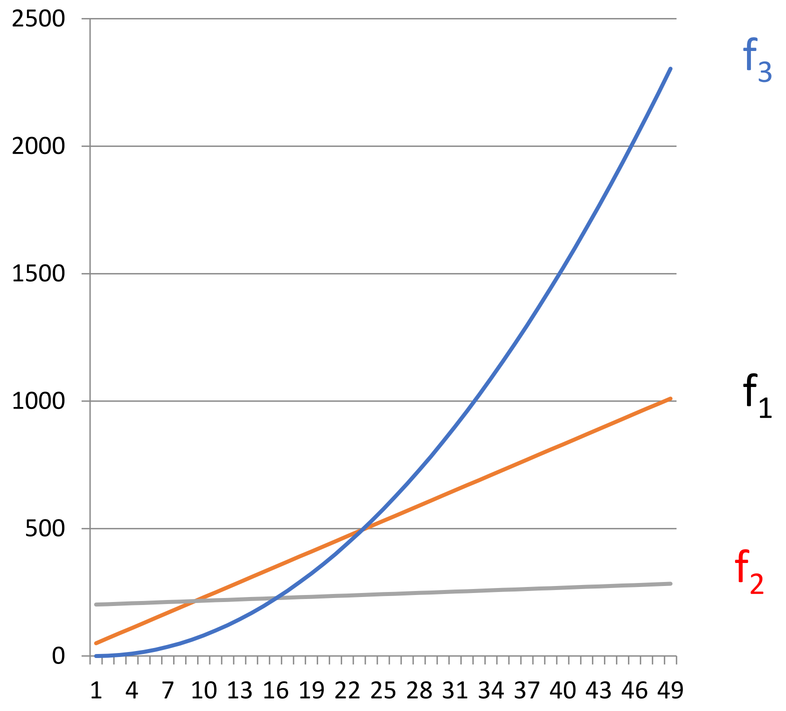
B. FALSE

Why or why not?



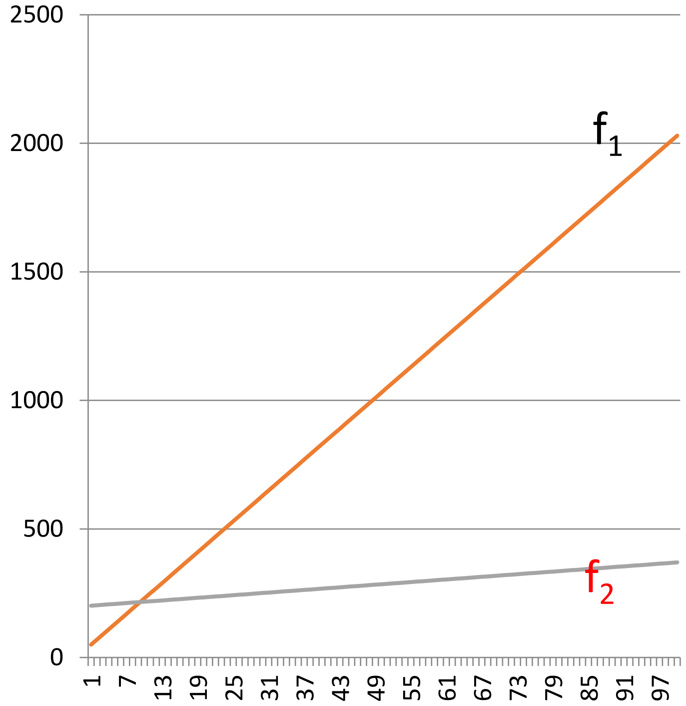
$f_1 = O(f_3)$ *but* $f_3 \neq O(f_1)$

- 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).



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 the definition asks us to find *some* constants that fit. Having some other constants that don't work is fine.
- 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.



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. $f(n)$ is $O(\log_2 n)$
B. $f(n)$ is $O(n \log_2 n)$
C. $f(n)$ is $O(n^2)$
D. $f(n)$ is $O(n)$
E. More than one of these

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

- Which of the following is NOT a correct bound?

A. $f(n)$ is $O(2^n)$
B. $f(n)$ is $O(n^2)$
C. $f(n)$ is $O(n)$
D. $f(n)$ is $O(n^3)$

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

- Which of the following is true?

A. f(n) is $O(2^n)$
B. f(n) is $O(n^2)$
C. f(n) is $O(n)$
D. f(n) is $O(n^3)$
E. More than one of these

Let $f(n) = 100$

- Which of the following is NOT a correct bound?

A. $f(n)$ is $O(2^n)$
B. $f(n)$ is $O(n^2)$
C. $f(n)$ is $O(n)$
D. $f(n)$ is $O(n^{100})$
E. None of these

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 $n \geq n_0$.

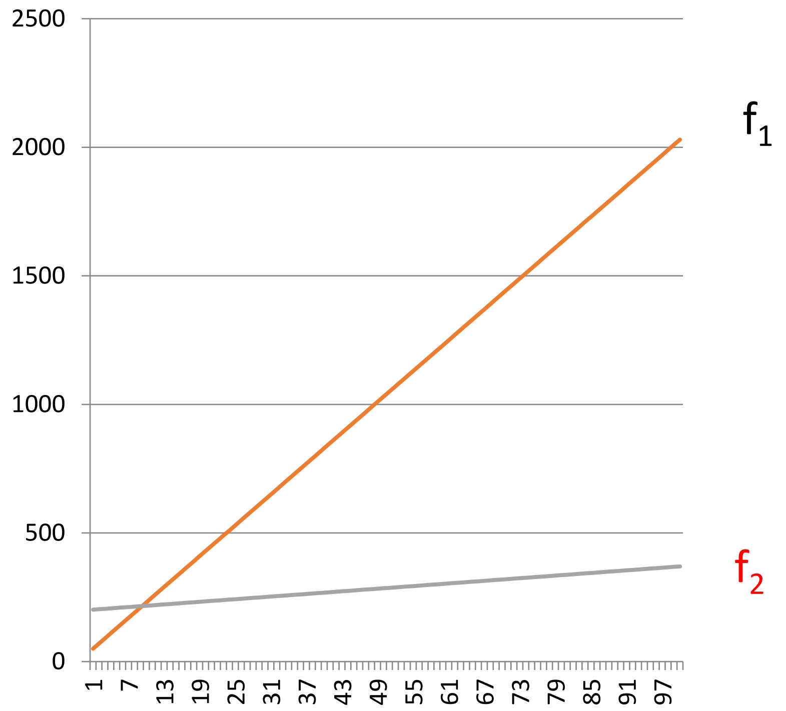
In other words, for large n, can you multiply g(n) by a constant and have it always be smaller than or equal to f(n)

f_2 is $\Omega(f_1)$

A. TRUE
B. FALSE

Why or why not?

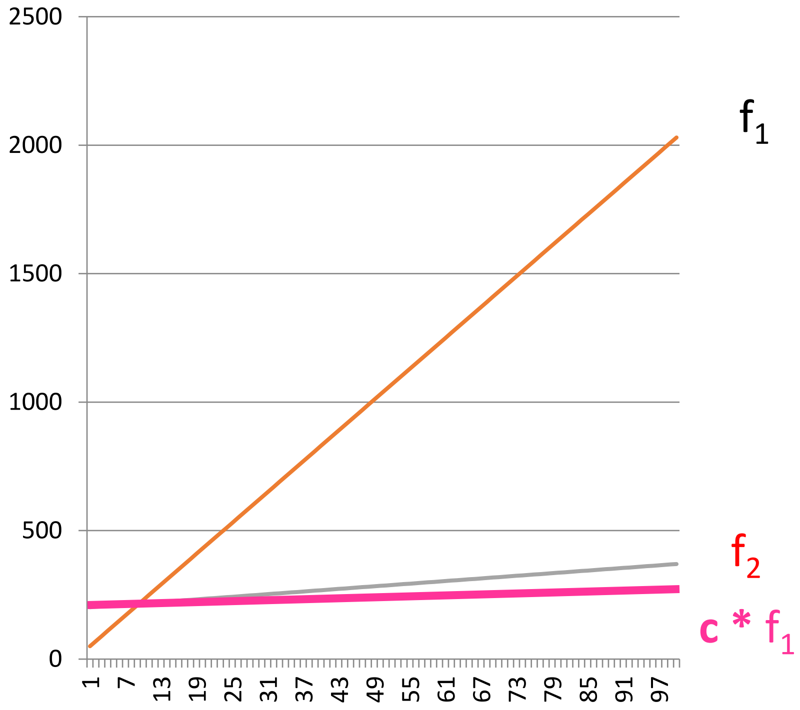
In other words, for large n, can you multiply f_1 by a positive constant and have it always be smaller than f_2



$f(n) = \mathbf{O}(g(n))$, if there are positive constants c and n_0 such that $f(n) \leq c * g(n)$ for all $n \geq n_0$.

$f(n) = \mathbf{\Omega}(g(n))$, if there are positive constants c and n_0 such that $f(n) \geq c * g(n)$ for all $n \geq n_0$.

- f_1 is $\Omega(f_2)$ 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 is $\Omega(f_1)$ as well!
  - We just have to use the "**c**" to adjust so f_1 that it moves below f_2

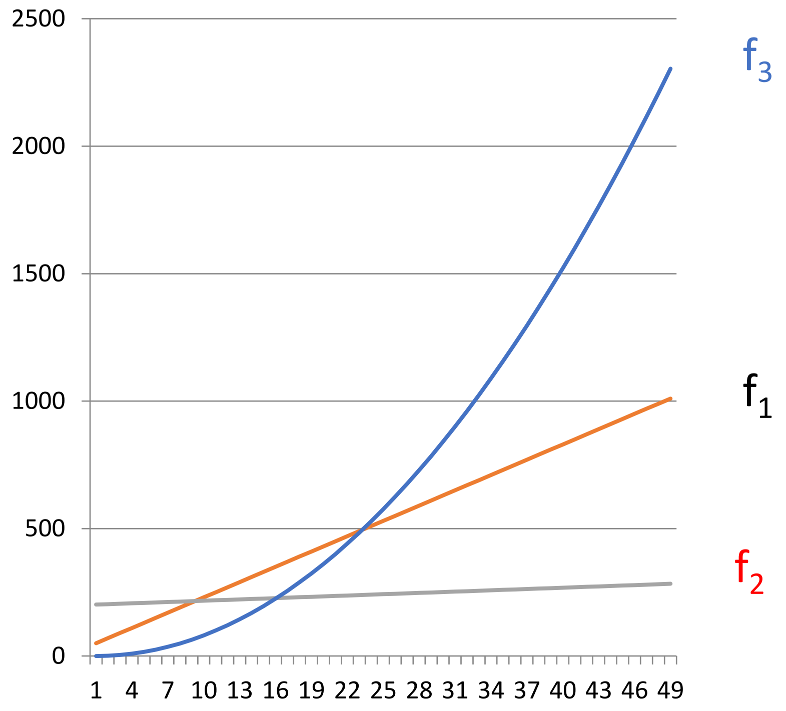


$f(n) = \mathbf{O}(g(n))$, if there are positive constants c and n_0 such that $f(n) \le c * g(n)$ for all $n \ge n_0$.
$f(n) = \mathbf{\Omega}(g(n))$, if there are positive constants c and n_0 such that $f(n) \ge c * g(n)$ for all $n \ge n_0$.

f_3 is $\Omega(f_1)$

A. TRUE

B. FALSE



Why or why not?

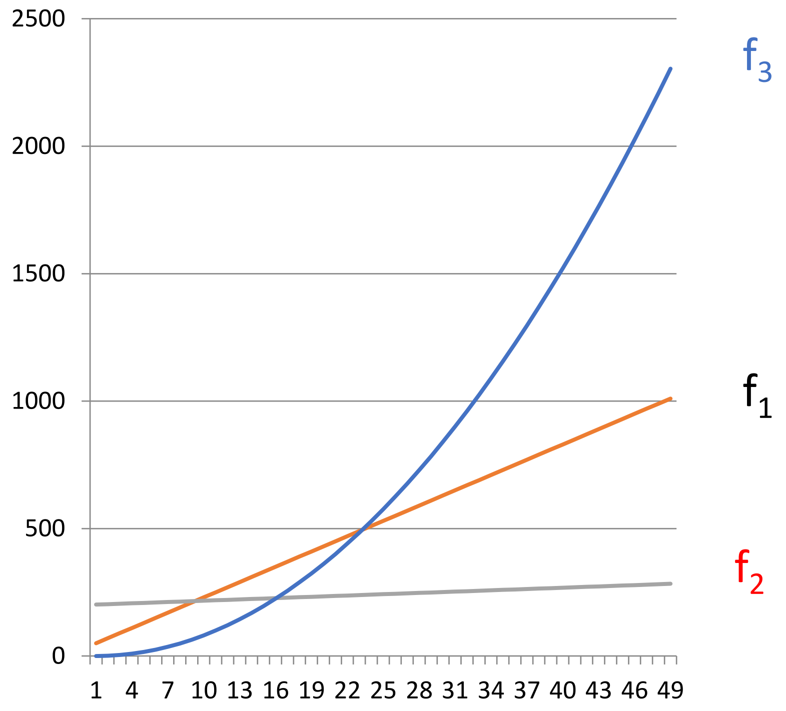
$f(n) = \mathbf{O}(g(n))$, if there are positive constants c and n_0 such that $f(n) \leq \mathbf{c} * g(n)$ for all $n \geq n_0$.
$f(n) = \mathbf{\Omega}(g(n))$, if there are positive constants c and n_0 such that $f(n) \geq \mathbf{c} * g(n)$ for all $n \geq n_0$.

f_1 is $\Omega(f_3)$

A. TRUE

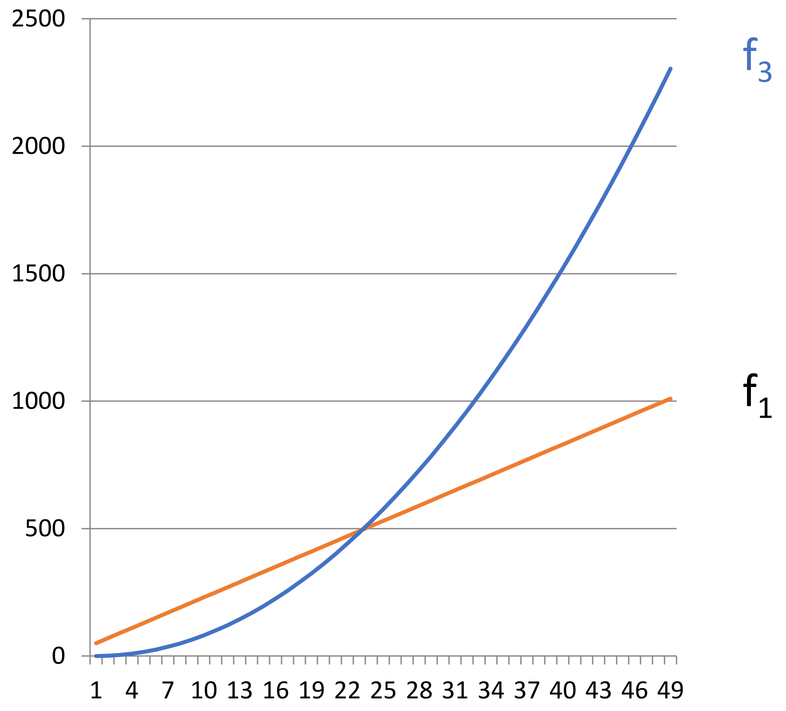
B. FALSE

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
- $f(n) = \Omega(g(n))$ 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

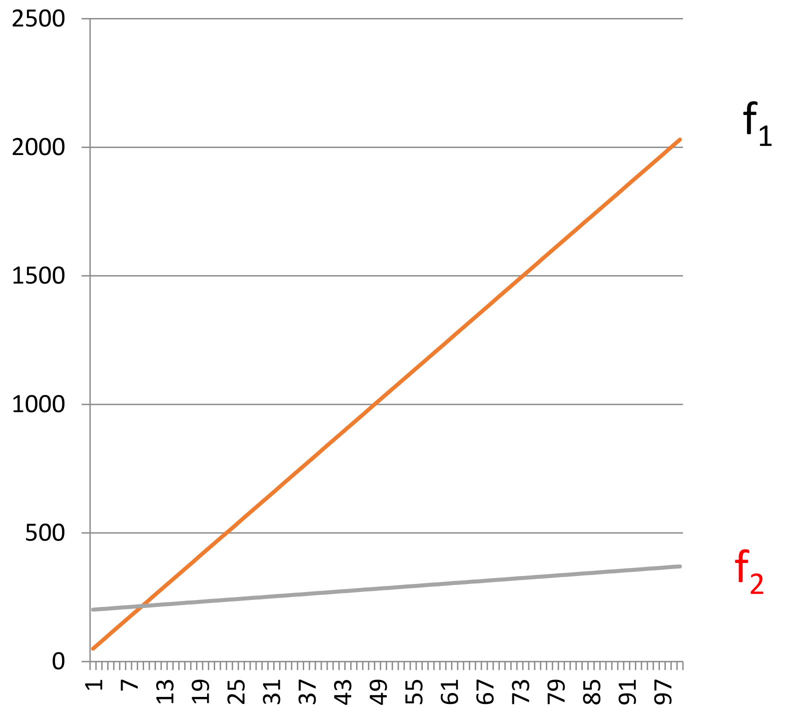
- **Tight bound** on a function.
- If f(n) = O(g(n)) *and* f(n) = Ω(g(n)), then f(n) = θ(g(n)).
- Basically it means that f(n) and g(n) are interchangeable
- Examples:
  - 3n+20 = θ(10n+7)
  - $5n^2 + 50n + 3 = \theta(5n^2 + 100)$

f_1 is $\Theta(f_2)$

A. TRUE

B. FALSE

Why or why not?

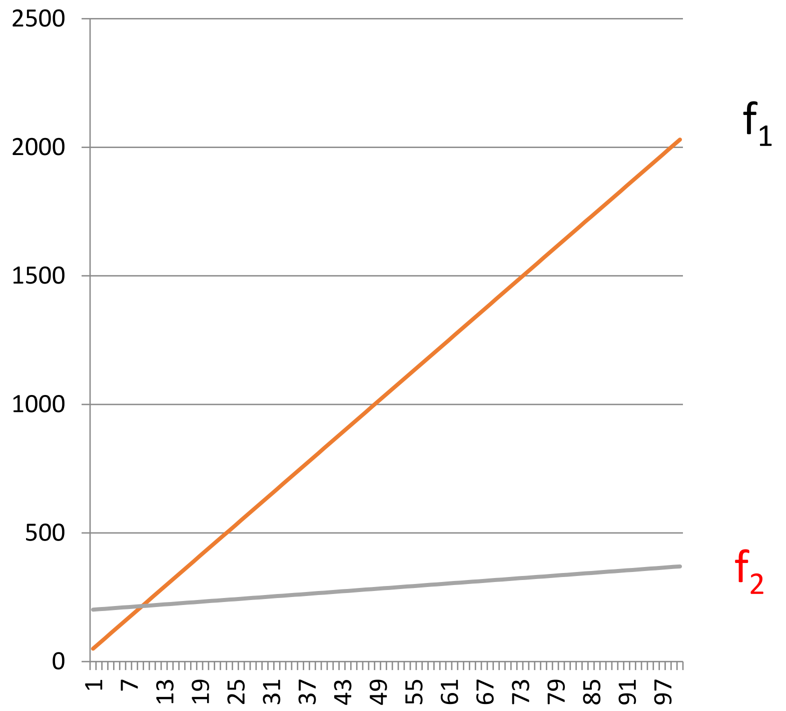


f_1 is $\Theta(f_2)$

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)

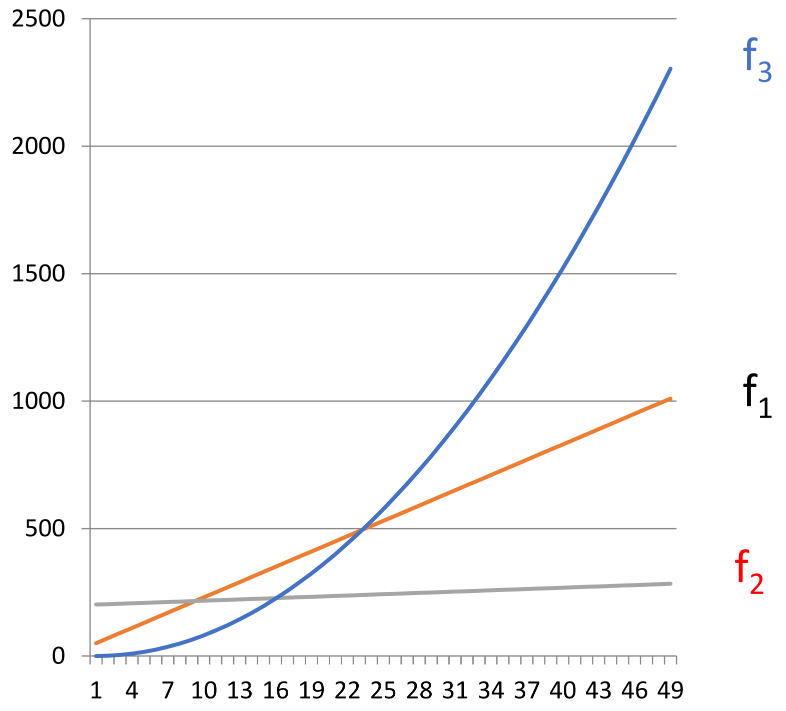


f_1 is $\Theta(f_3)$

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

Count how many times each line executes, then say which $\Theta(\ )$ **statement(s) is(are) true**.

```
int maxDifference(int[] arr){
   max = 0;
   for (int i=0; i<arr.length;  i++) {
      for (int j=0; j<arr.length; j++) {
         if (arr[i] – arr[j] > max)
            max = arr[i] – arr[j];
      }
   }
   return max;
}
```

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

(assume n = arr.length)

Count how many times each line executes, then say which $\Theta(\,)$ **statement(s) is(are) true**.

```
int sumTheMiddle(int[] arr){
   int range = 100;
   int start = arr.length/2 – range/2;
   int sum = 0;
   for (int i=start; i<start+range;  i++)
   {
      sum += arr[i];
   }
   return max;
}
```

A. $f(n) = \theta(2^n)$
B. $f(n) = \theta(n^2)$
C. $f(n) = \theta(n)$
D. $f(n) = \theta(1)$
E. None of these

(assume n = arr.length)

Count how many times each line executes, then say which $\Theta(\,)$ **statement(s) is(are) true**.

```
int sumTheMiddle(int[] arr){
   int range = arr.length/100;
   int start = arr.length/2 - range/2;
   int sum = 0;
   for (int i=start; start+range;  i++)
   {
      sum += arr[i];
   }
   return sum;
}
```

A. $f(n) = \theta(2^n)$
B. $f(n) = \theta(n^2)$
C. $f(n) = \theta(n)$
D. $f(n) = \theta(1)$
E. None of these

(assume n = arr.length)

With *worst case* analysis, which algorithm has the better (smaller) Big-Θ bound?

```
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;
}
```

A. find
B. fastFind
C. They are the same

With *best case* analysis, which algorithm has the better Big-Θ bound?

```
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;
}
```

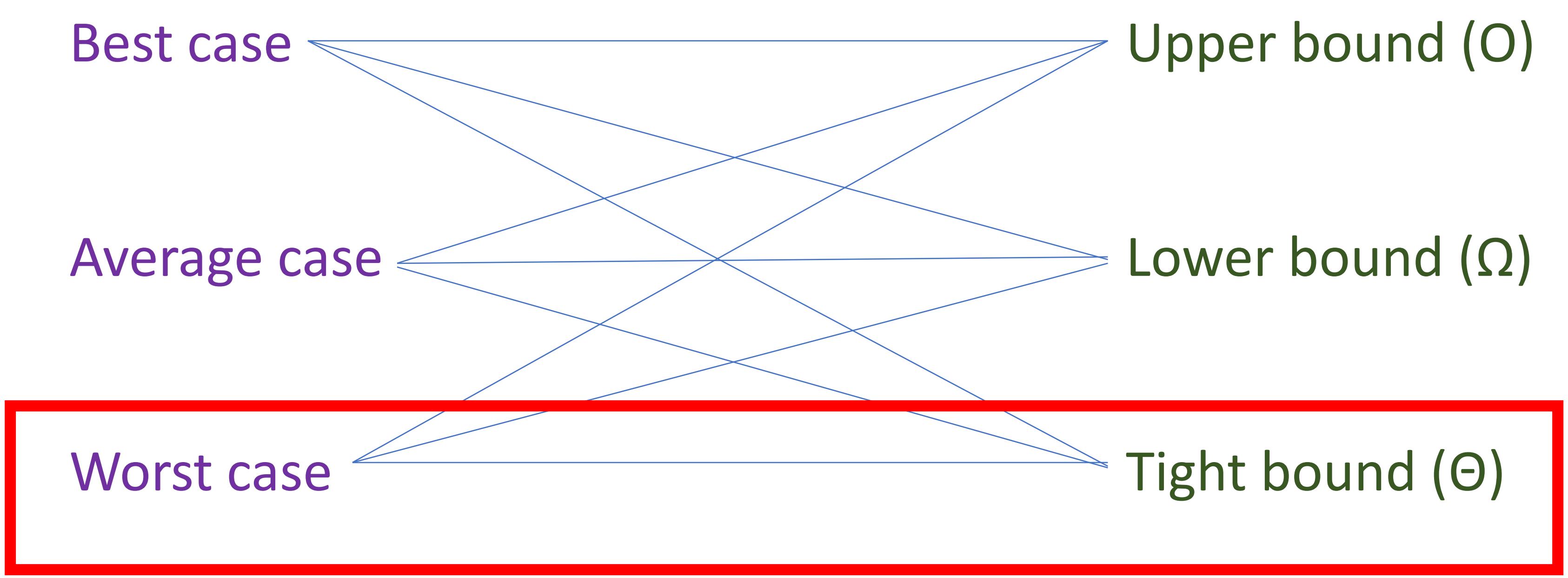
A. find
B. fastFind
C. They are the same

Big-O means *upper bound* NOT worst case

The many ways to do running time analysis
Any combination of purple and green OK!

Which version of the problem?

What kind of bound?



Most common combination