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:

- $f(n) \leq c\, g(n)$ 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)

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))

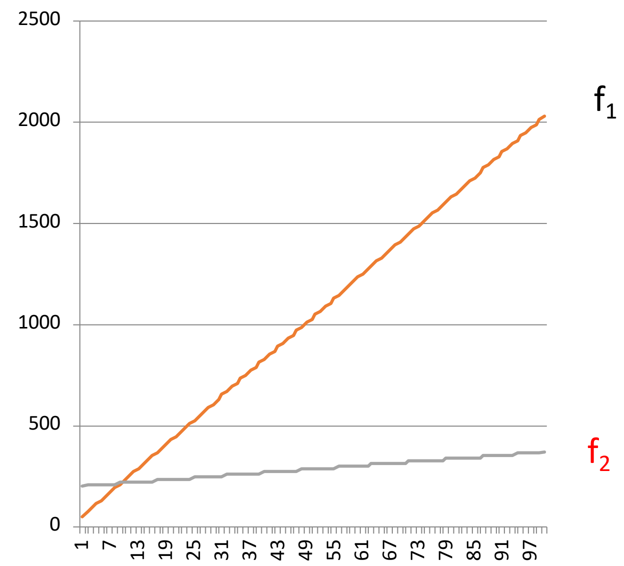
f_2 is* $O(f_1)$

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

A. TRUE

B. FALSE

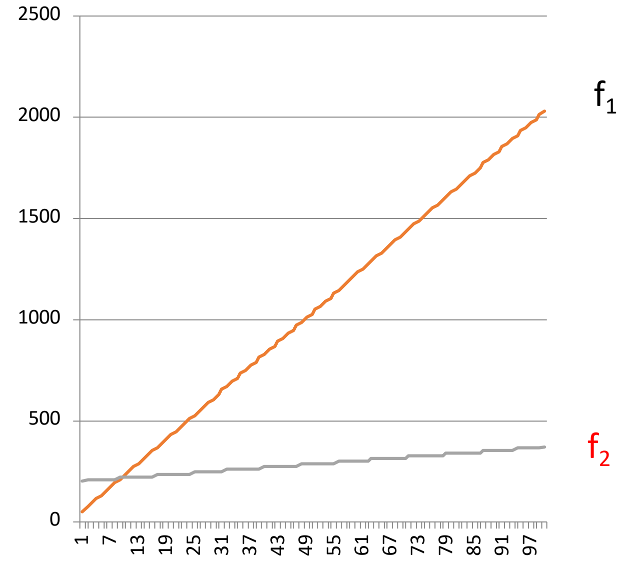
Why or why not?



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

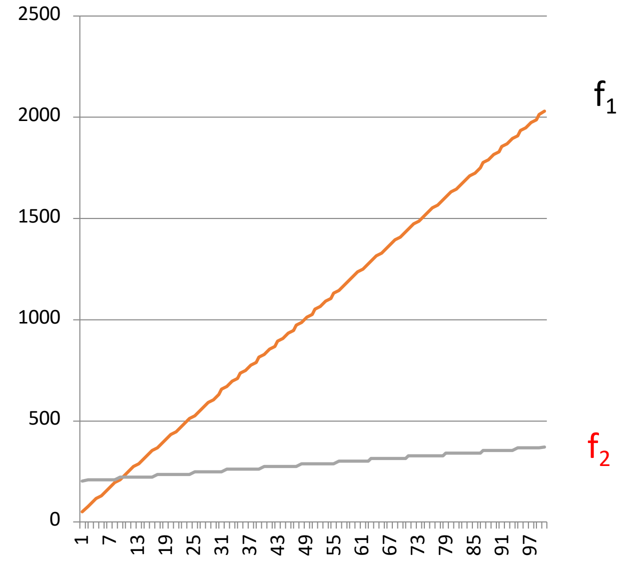
- Obviously $f_2 = O(f_1)$ because $f_1 > f_2$ (after about n=10, so we set n_0 = 10)
  - f_1 is clearly an *upper bound* on f_2 and that's what big-O is all about



f_1 is $O(f_2)$

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

A. TRUE
B. FALSE

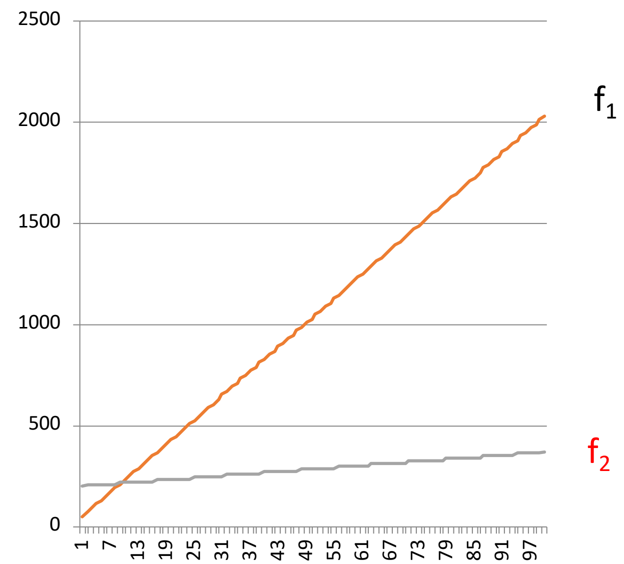


Why or why not?

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

- 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



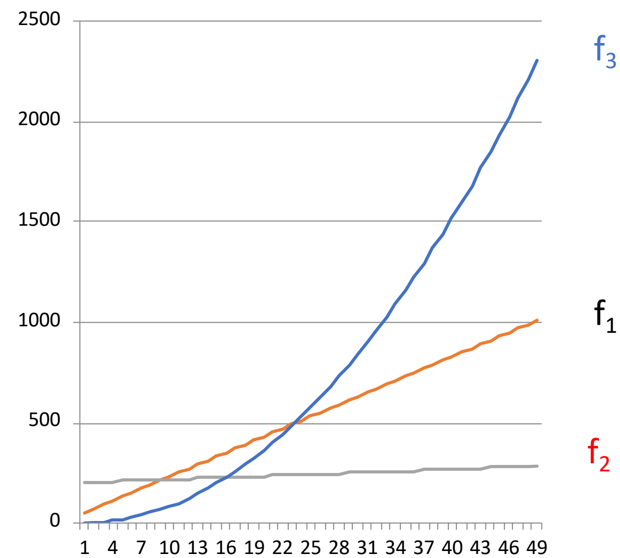
f(n) = **O**(g(n)), if there are positive constants c and n_0 such that f(n) **≤ c** * g(n) for all n ≥ n_0.

f_1 is $O(f_3)$

A. TRUE

B. FALSE

Why or why not?

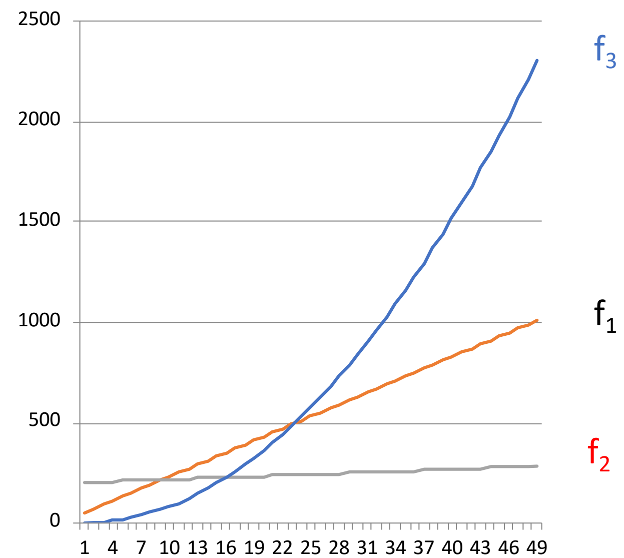


f(n) = **O**(g(n)), if there are positive constants c and n_0 such that f(n) **≤** c * g(n) for all n ≥ n_0.

f_3 is $O(f_1)$

A. TRUE

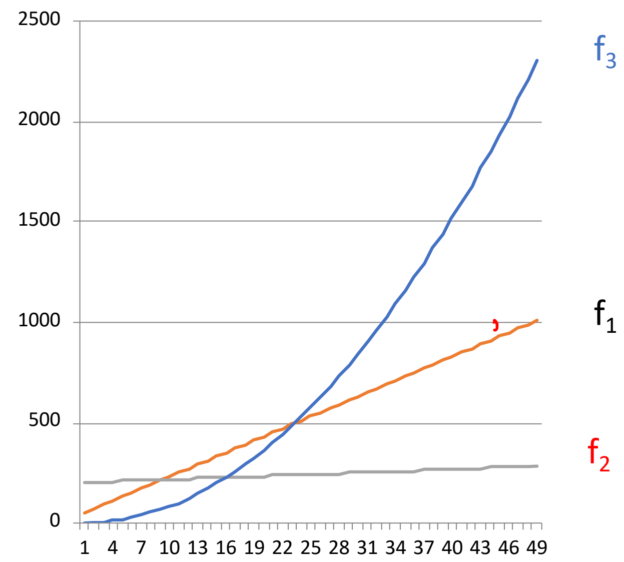
B. FALSE



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 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.