# CSE 12 – Basic Data Structures: Running Time Analysis

Prof. Joe Politz

[Slides borrowed/adapted from slides by

Christine Alvarado & Marina Langlois]

# Lists: One application

```
List<String> playlist
```

# Lists: One application

`List<String> 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<String> 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<String> 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, unless we can prove it's slow!

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

| Code | # of instr. |
|---|---|
| `boolean find( String[] theList, String toFind ) {` | N/A |
| `    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 ) {
  int count = 0;
  for ( int i = 0; i < theList.length; i++ ) {
    count = count + 1;
    if ( theList[i].equals(toFind) )
      return true;
  }
  return false;
}
```

Which method is faster?

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

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

1. Count the number of instructions in your code (or algorithm) 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!
2. Simplify your f(n) to find a simple g(n) such that f(n) = O(g(n)) (or $\Omega$(g(n) or $\Theta$(g(n))

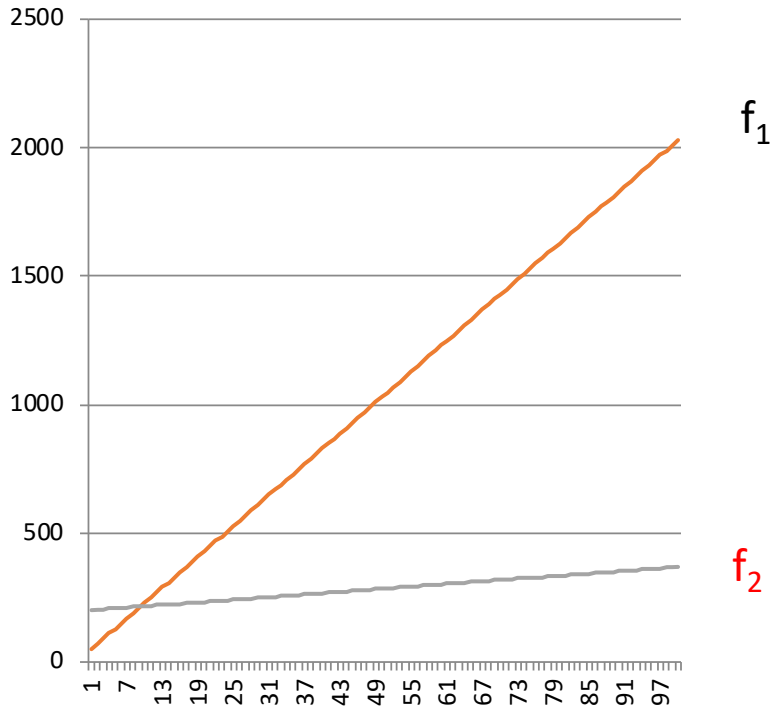
# $f_2$ is* $O(f_1)$

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 ≥ $n_0$.

A. TRUE

B. FALSE

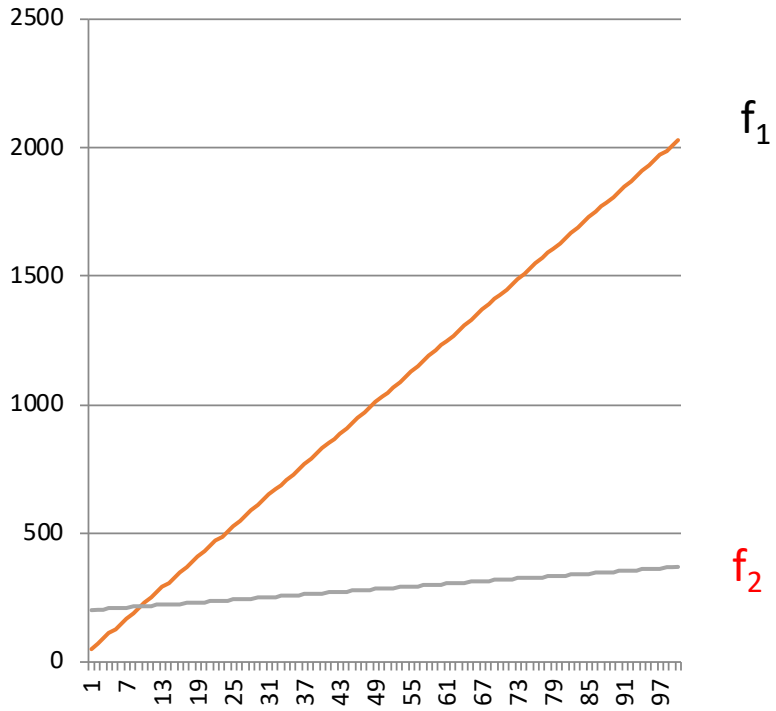
Why or why not?




* 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 ≥ $n_0$.

- Choose **n0 = 10, C = 1**
  - $f_1$ is clearly an *upper bound* on $f_2$ and that's what big-O is all about



# $f_1$ is $O(f_2)$

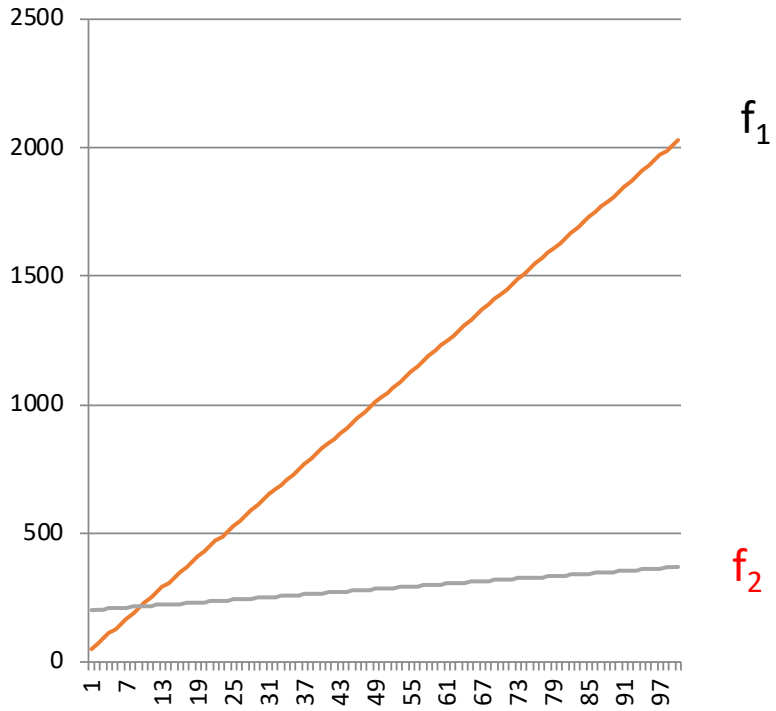
f(n) is **O**(g(n)), if there are positive constants c and $n_0$ such that f(n) **≤ c** * g(n) for all $n \geq n_0$.

A. TRUE

B. FALSE

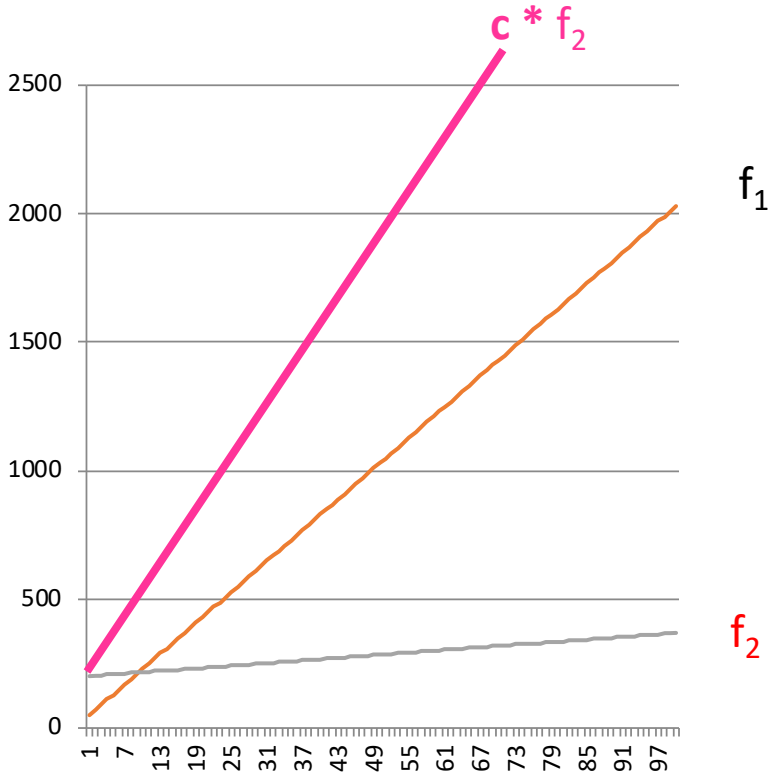
Why or why not?

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

- 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 $f_1$
  - Choose **n0 = 0, C = 1000** (something big enough to force the line above f1)



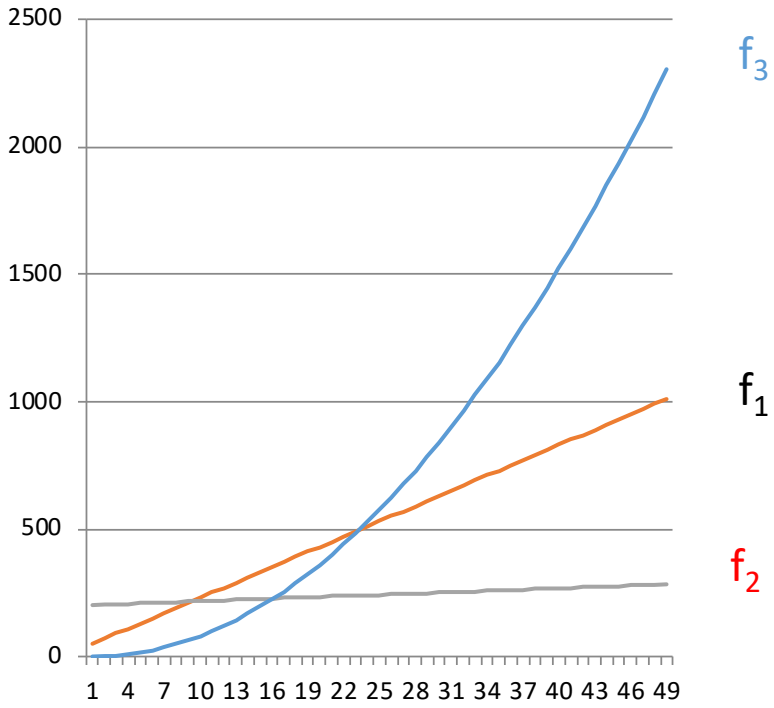
f(n) is **O**(g(n)), if there are positive constants c and $n_0$ such that f(n) ≤ **c** * g(n) for all n ≥ $n_0$.

# $f_1$ is O($f_3$)

A. TRUE

B. FALSE

Why or why not?

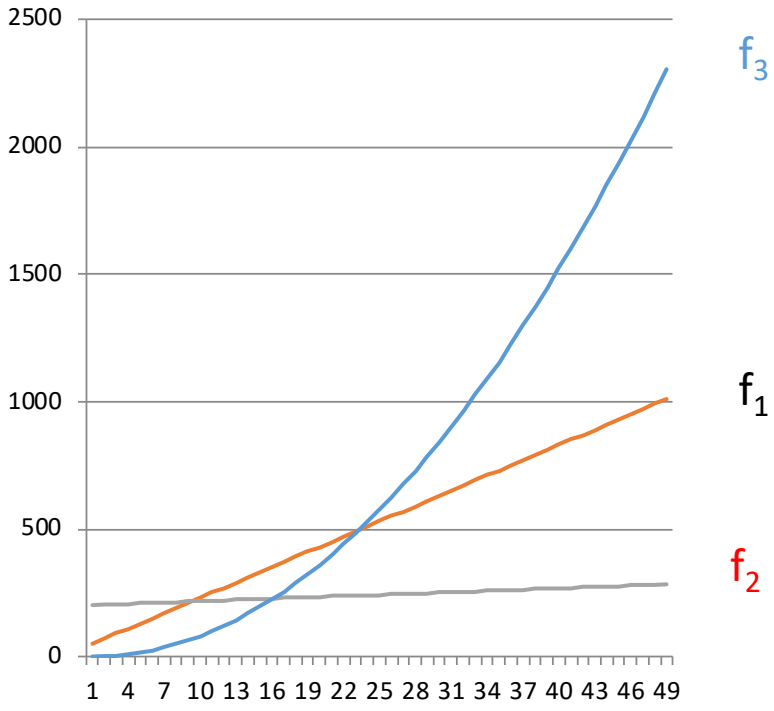


f(n) is **O**(g(n)), if there are positive constants c and $n_0$ such that f(n) **≤ c** * g(n) for all n ≥ $n_0$.

# $f_3$ is O($f_1$)

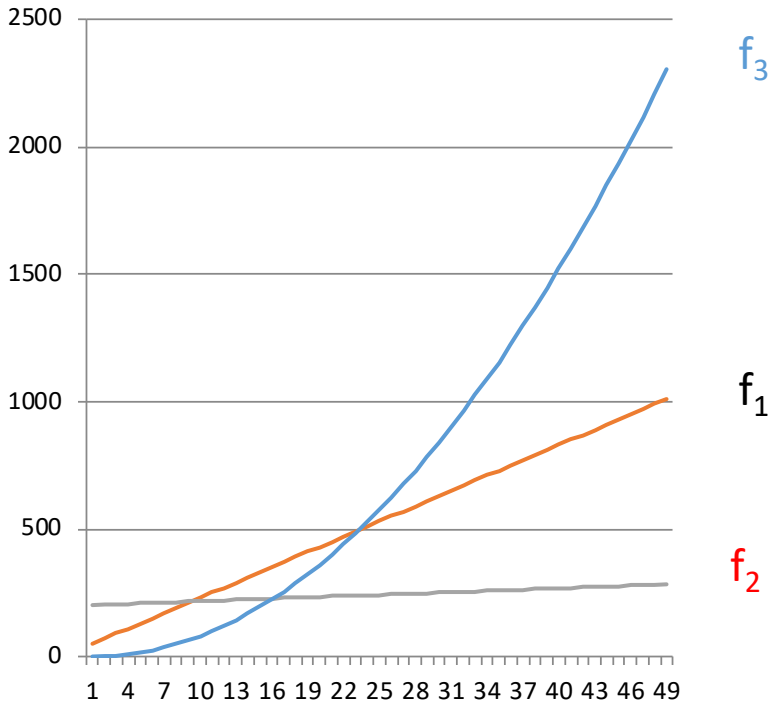
A. TRUE
B. FALSE

Why or why not?



# $f_1$ is $O(f_3)$ *but* $f_3$ **is not** $O(f_1)$

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



# 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 upper 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

Count how many times each line executes, then which O( ) **most tightly and correctly characterizes** the growth?

```
                                                    Line #
int maxDifference(int[] arr){                         1
   max = 0;                                           2
   for (int i=0; i<arr.length;  i+=1) {               3
      for (int j=0; j<arr.length; j+=1) {             4
         if (arr[i] - arr[j] > max)                   5
            max = arr[i] - arr[j];                    6
      }                                               7
   }                                                  8
   return max;                                        9
}
```

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

*(assume n = arr.length)*

Count how many times each line executes, then which O( ) **most tightly and correctly characterizes** the growth?

```
                                                    Line #
int maxDifference(int[] arr){                         1
   max = 0;                                           2
   for (int i=0; i<arr.length;  i+=1) {               3
      for (int j=0; j<arr.length; j+=2) {             4
         if (arr[i] - arr[j] > max)                   5
            max = arr[i] - arr[j];                    6
      }                                               7
   }                                                  8
   return max;                                        9
}
```

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

*(assume n = arr.length)*

Count how many times each line executes, then which O( ) **most tightly and correctly characterizes** the growth?

```
                                                    Line #
int maxDifference(int[] arr){                         1
   max = 0;                                           2
   for (int i=0; i<arr.length;  i+=1) {               3
      for (int j=0; j<100000; j+=2) {                 4
         if (arr[i] - arr[j] > max)                   5
            max = arr[i] - arr[j];                    6
      }                                               7
   }                                                  8
   return max;                                        9
}
```

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

*(assume n = arr.length)*

# 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 $n \geq n_0$.

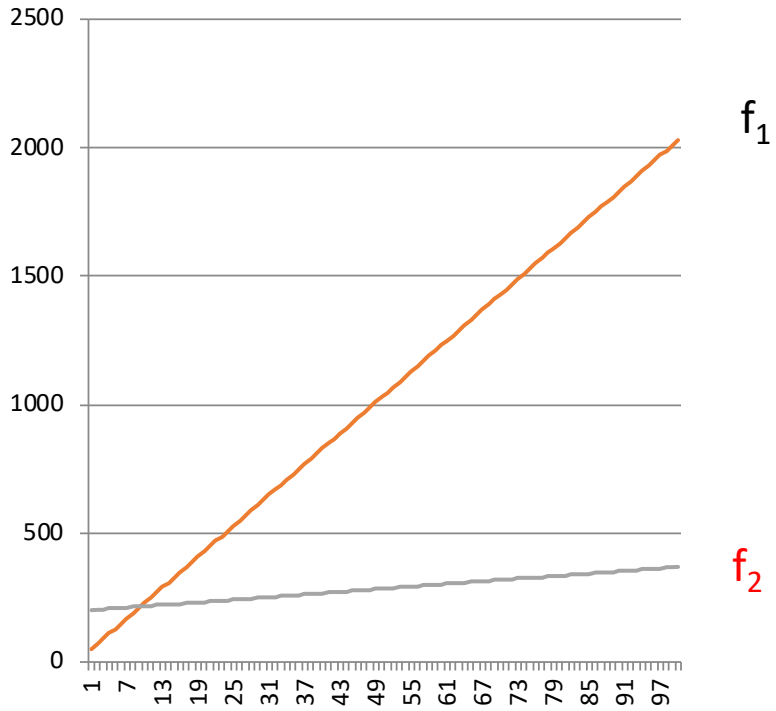
In other words, for large n, can you multiply g(n) by a constant and have it always be smaller than or equal to f(n)

# $f_2$ is $\Omega(f_1)$

A. TRUE
B. FALSE

Why or why not?

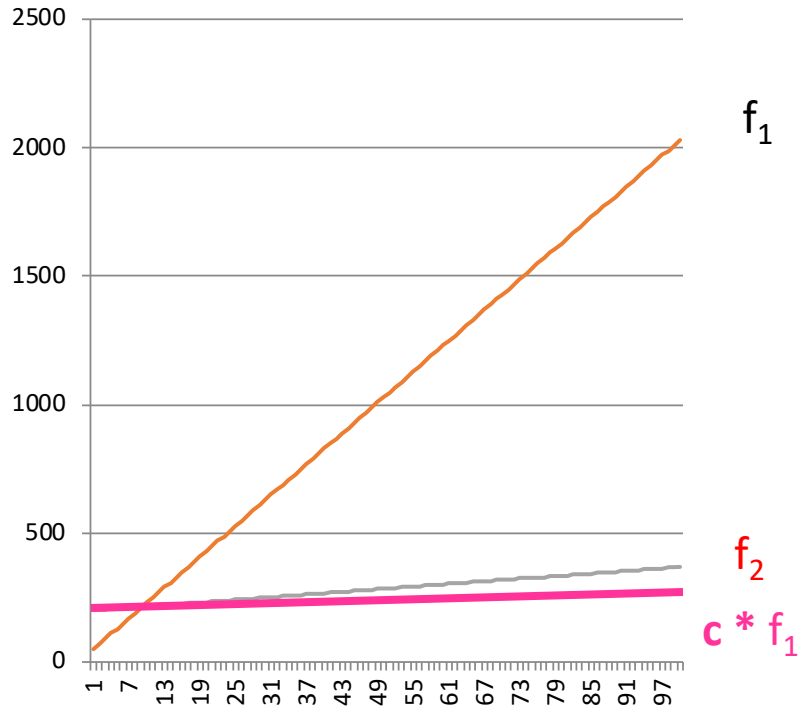
In other words, for large n, can you multiply $f_1$ by a positive constant and have it always be smaller than $f_2$



f(n) 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(n) is **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 $f_2$



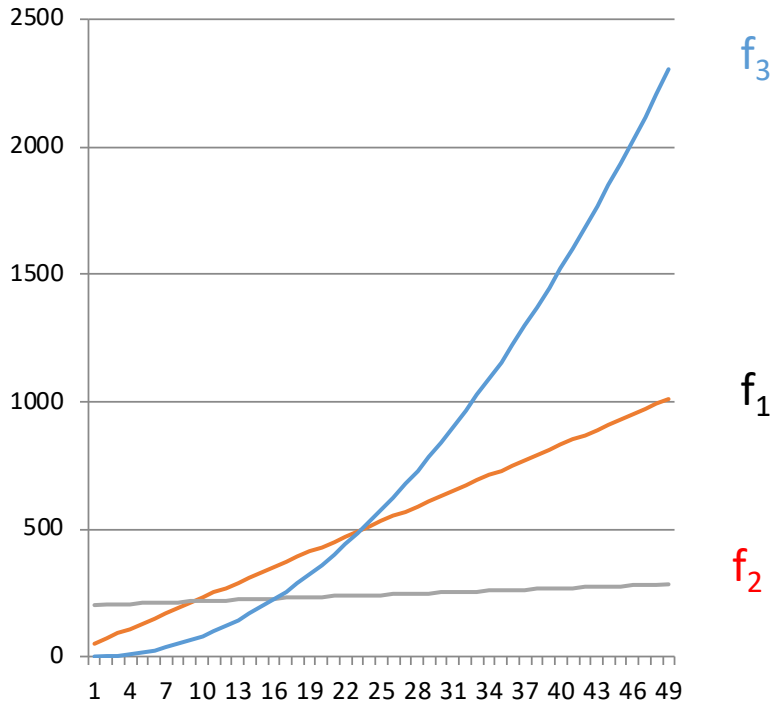
| |
|---|
| f(n) is **O**(g(n)), if there are positive constants c and $n_0$ such that f(n) **≤ c** * g(n) for all n ≥ $n_0$. |
| f(n) is **Ω**(g(n)), if there are positive constants c and $n_0$ such that f(n) **≥ c** * g(n) for all n ≥ $n_0$. |

# $f_3$ is $\Omega(f_1)$

A. TRUE

B. FALSE

Why or why not?



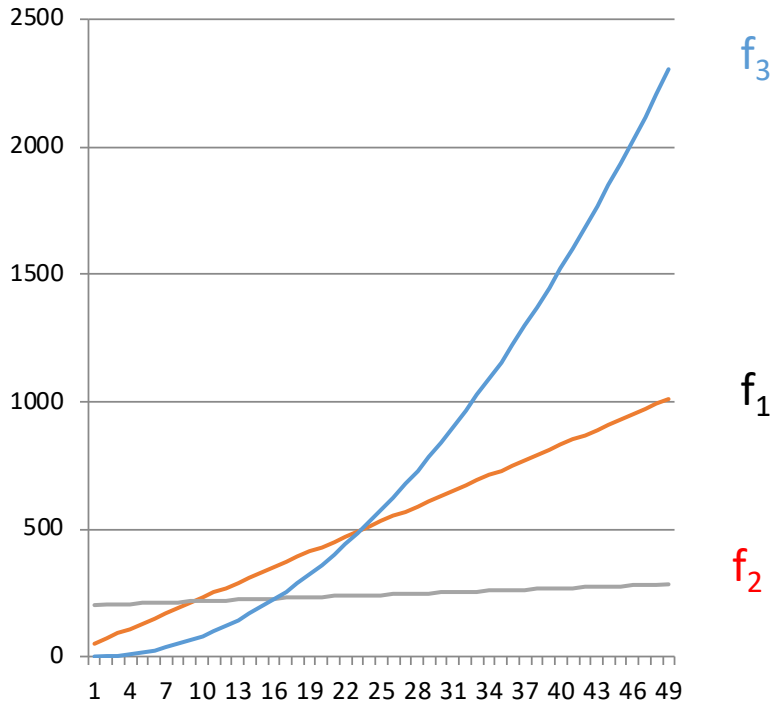
| |
|---|
| f(n) is **O**(g(n)), if there are positive constants c and $n_0$ such that f(n) **≤ c** * g(n) for all n ≥ $n_0$. |
| f(n) is **Ω**(g(n)), if there are positive constants c and $n_0$ such that f(n) **≥ c** * g(n) for all n ≥ $n_0$. |

# $f_1$ is $\Omega(f_3)$

A. TRUE

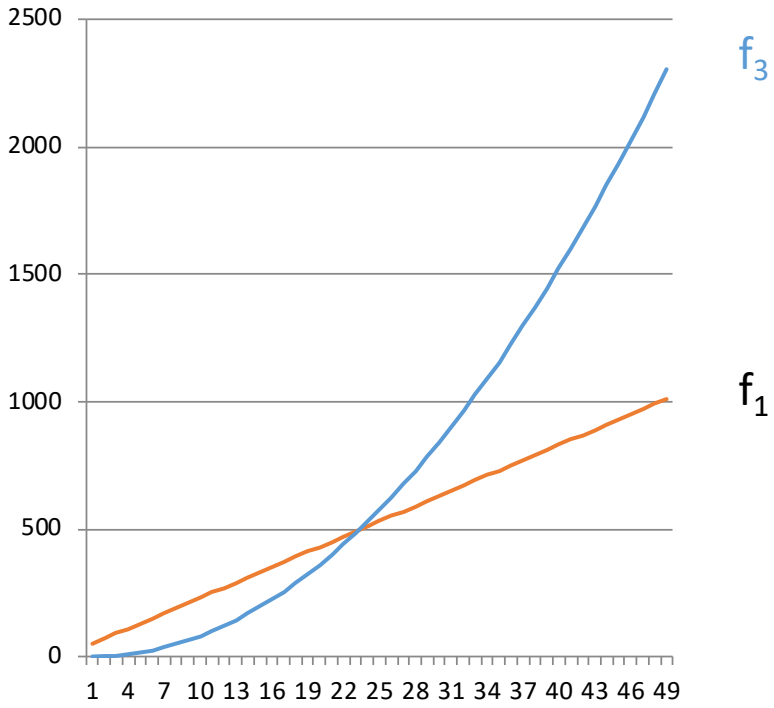
B. FALSE

Why or why not?



# $f_3$ is $\Omega(f_1)$ *but* $f_1$ **is not** $\Omega(f_3)$

- 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

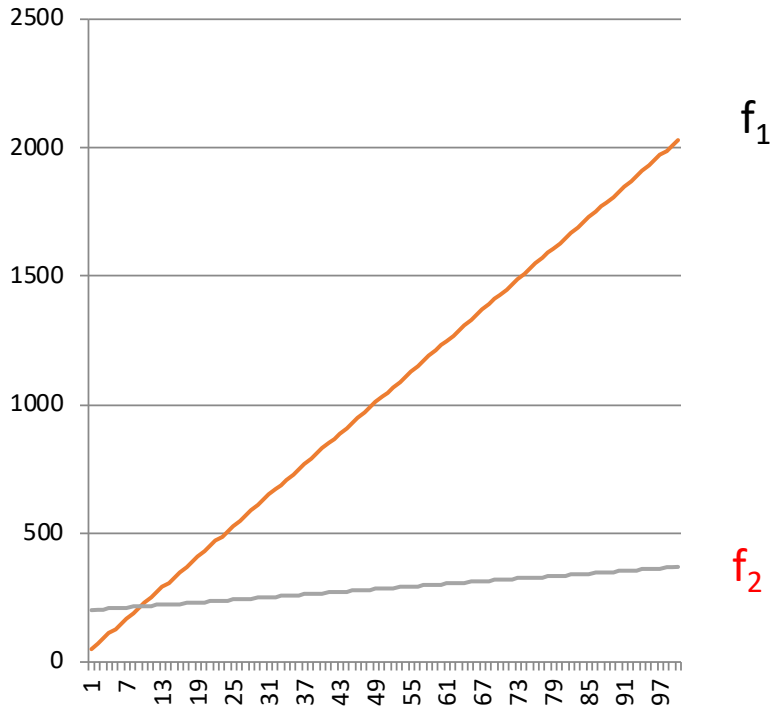
# Big-θ

- **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?

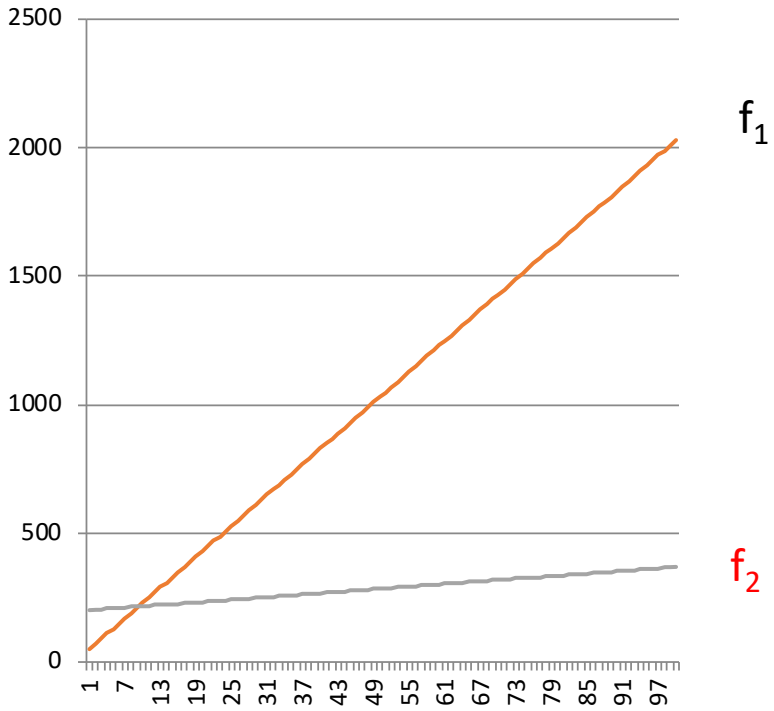


# $f_1$ is $\Theta(f_2)$

A. TRUE
B. FALSE

Why or why not?

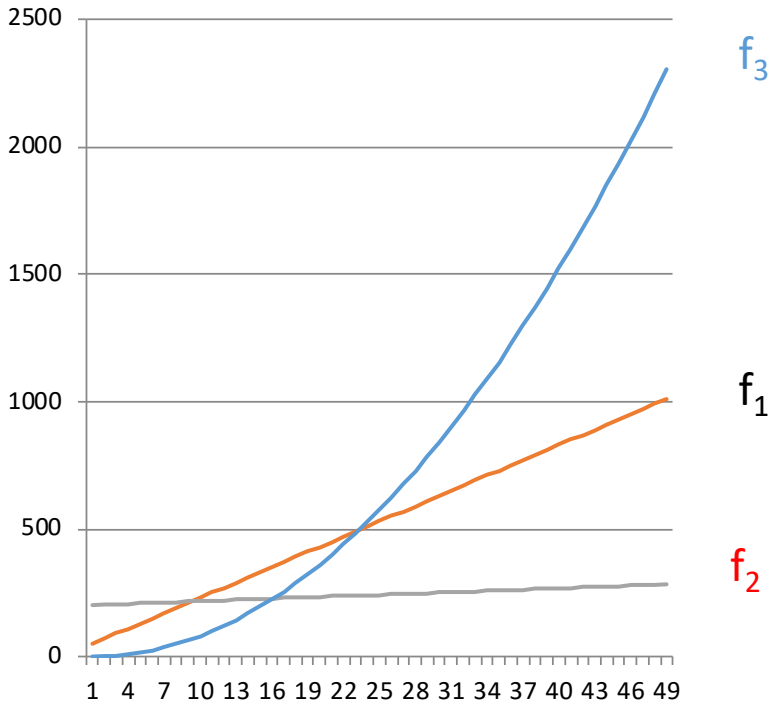
Since $f_1$ is $O(f_2)$ and $\Omega(f_2)$, it is also $\Theta(f_2)$ (this is the definition of big-Theta)



# $f_1$ 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

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

```
                                                    Line #
int maxDifference(int[] arr){                         1
   int max = 0;                                       2
   for (int i=0; i<arr.length;  i+=1) {               3
      for (int j=0; j<arr.length; j+=1) {             4
         if (arr[i] - arr[j] > max)                   5
            max = arr[i] - arr[j];                    6
      }                                               7
   }                                                  8
   return max;                                        9
}
```

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. Other/none/more

*(assume n = arr.length)*

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

```
                                                    Line #
int maxDifference(int[] arr){                         1
   int max = 0;                                       2
   for (int i=0; i<arr.length;  i+=1) {               3
      for (int j=0; j<arr.length; j+=1) {             4
         if (arr[i] – arr[j] > max)                   5
            max = arr[i] – arr[j];                    6
      }                                               7
   }                                                  8
   return max;                                        9
}
```

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

*(assume n = arr.length)*

Count how many times each line executes, then say which Θ( ) **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) is $\theta(2^n)$
B. f(n) is $\theta(n^2)$
C. f(n) is $\theta(n)$
D. f(n) is $\theta(1)$
E. None of these

*(assume n = arr.length)*

Count how many times each line executes, then say which Θ( ) **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) is $\theta(2^n)$
B. f(n) is $\theta(n^2)$
C. f(n) is $\theta(n)$
D. f(n) is $\theta(1)$
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?

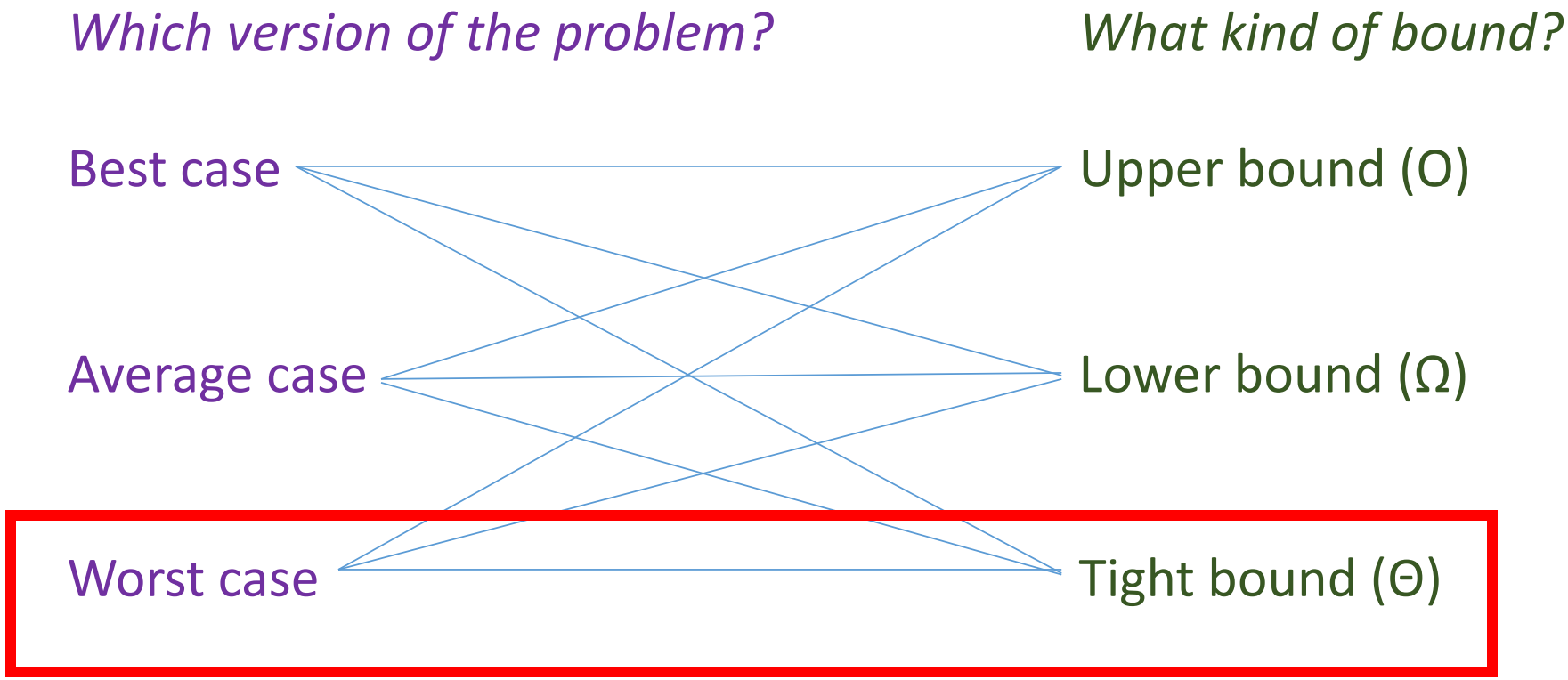
```
boolean find( String[] theList, String toFind ) {
  for ( int i = 0; i < theList.length; i++ ) {
    if ( theList[i].equals(toFind) )
      return true;
  }
  return false;
}

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 (left) and green (right) has meaning!




Most common combination