CSE 12 – Basic Data Structures and Object-Oriented Design
Lecture 11

Greg Miranda & Paul Cao, Winter 2021

This lecture is being recorded

Announcements

- Quiz 11 due Wednesday @ 8am
- Survey 5 due Friday @ 11:59pm
- PA4 due Wednesday @ 11:59pm

Topics

- Questions on Lecture 11?
- Big Theta
- Sorting

Questions on Lecture 11?

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 $f(n) \in \mathbf{\Omega}(g(n))$, if there are positive constants c and n_0 such that:

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

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)

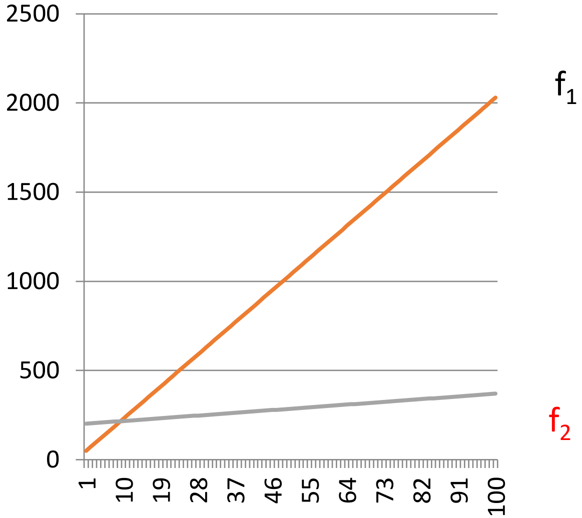
$f_2 \in \Omega(f_1)$

A. TRUE

B. FALSE

Why or why not?

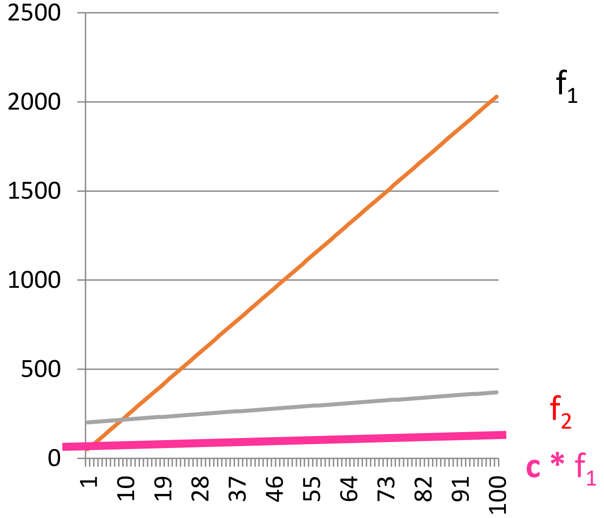
In other words, for large n, can you multiply f_1 by a positive constant and have it always be smaller than f_2



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



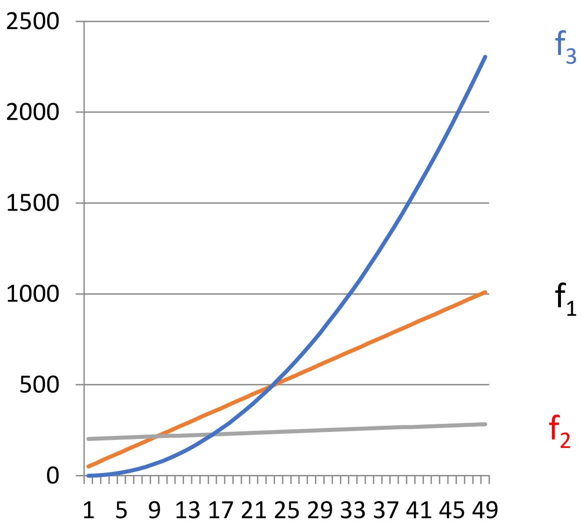
$f(n) \in \mathbf{O}(g(n))$, if there are positive constants c and n_0 such that $f(n) \le c * g(n)$ for all $n \ge n_0$.

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

$$f_3 \in \Omega(f_1)$$

A. TRUE

B. FALSE



Why or why not?

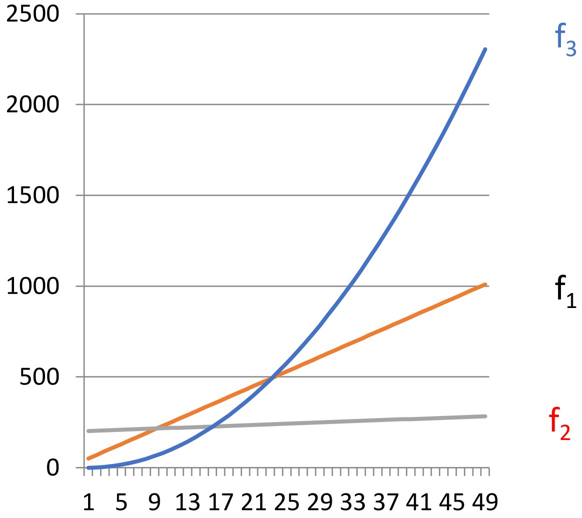
$f(n) \in \mathbf{O}(g(n))$, if there are positive constants c and n_0 such that $f(n) \leq c * g(n)$ for all $n \geq n_0$.

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

$f_1 \in \Omega(f_3)$

A. TRUE

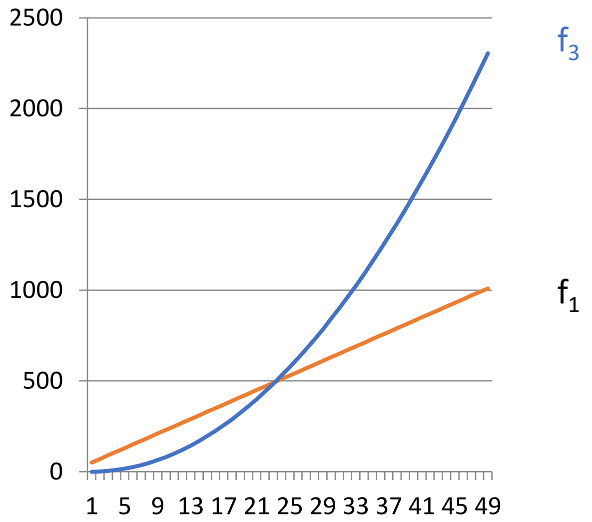
B. FALSE



Why or why not?

$f_3 \in \Omega(f_1)$ *but* f_1 not $\in \Omega(f_3)$

- 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
- $f(n) \in O(g(n))$ 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
- $f(n) \in \Omega(g(n))$ 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

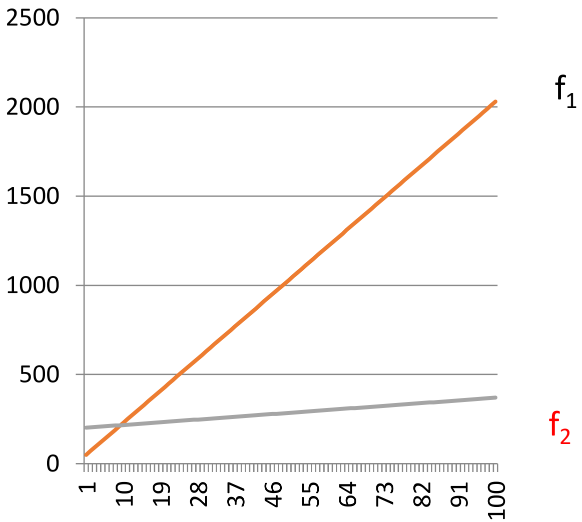
- **Tight bound** on a function.
- If $f(n) \in O(g(n))$ *and* $f(n) \in \Omega(g(n))$*,* then $f(n) \in \theta(g(n))$.
- Basically it means that f(n) and g(n) are interchangeable
- Examples:
  - $3n+20 = \theta(10n+7)$
  - $5n^2 + 50n + 3 = \theta(5n^2 + 100)$

$f_1 \in \Theta(f_2)$

A. TRUE

B. FALSE

Why or why not?



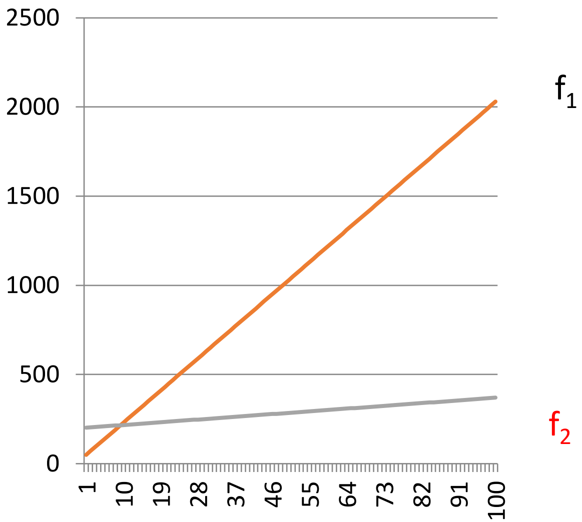
$f_1 \in \Theta(f_2)$

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)

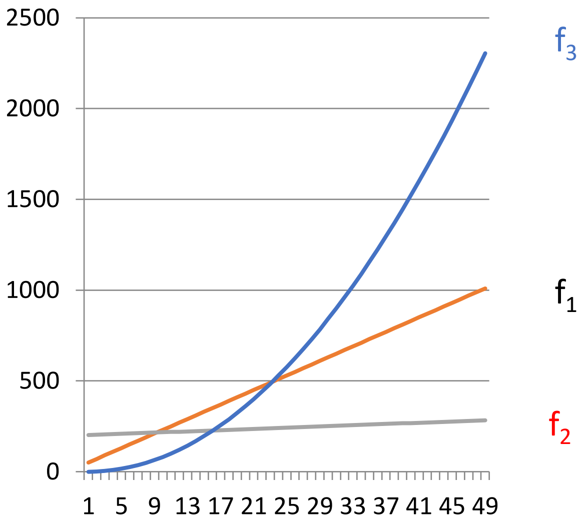


$f_1 \in \Theta(f_3)$

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 #

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

A. $f(n) \in O(2^n)$

B. $f(n) \in O(n^2)$

C. $f(n) \in O(n)$

D. $f(n) \in O(n^3)$

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 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. $f(n) \in \theta(2^n)$

B. $f(n) \in \theta(n^2)$

C. $f(n) \in \theta(n)$

D. $f(n) \in \theta(n^3)$

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. $f(n) \in \theta(2^n)$
B. $f(n) \in \theta(n^2)$
C. $f(n) \in \theta(n)$
D. $f(n) \in \theta(1)$
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. $f(n) \in \theta(2^n)$

B. $f(n) \in \theta(n^2)$

C. $f(n) \in \theta(n)$

D. $f(n) \in \theta(1)$

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

Practical sorting algorithms: Selection sort

<u>Pseudocode: selectionSort</u>

- *While the size of the unsorted part is greater than 1*
    - *find the position of the smallest element in the unsorted part*
    - *move this smallest element to the last position in the sorted part*
    - *increase the size of the sorted part and decrement the size of the unsorted part*

https://www.youtube.com/watch?v=Ns4TPTC8whw

12	4	9	3	15	8	19	2

sorted

unsorted

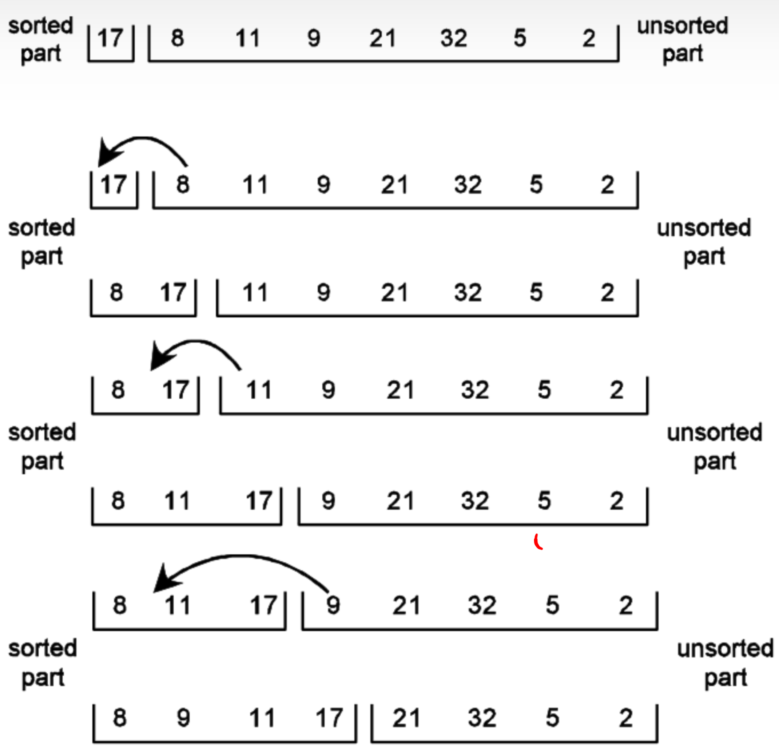
Insertion Sort: The Picture

(a) Initial configuration for insertion sort. The input array is logically split into a sorted part (initially containing one element) and an unsorted part.

(b) The array after the first value from the unsorted part has been inserted in its proper position in the sorted part (first pass).

(c) The array after the first value from the unsorted part has been inserted in its proper position in the sorted part (second pass).

(d) The array after the first value from the unsorted part has been inserted in its proper position in the sorted part (third pass).



https://www.youtube.com/watch?v=ROalU379l3U

Selection Sort – what does it print out?

```
import java.util.Arrays;
public class Sort {
public static void sortA(int[] arr) {
  for(int i = 0; i < arr.length; i += 1) {
    System.out.print(Arrays.toString(arr) + " -> ");
    int minIndex = i;
    for(int j = i; j < arr.length; j += 1) {
      if(arr[minIndex] > arr[j]) { minIndex = j; }
    }
    int temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
    System.out.println(Arrays.toString(arr));
  }
}
}
```

```
Sort.sortA(new int[]{ 53, 83, 15, 45, 49 });
```

```
[53, 83, 15, 45, 49] ->
```

Insertion Sort – what does it print out?

```
import java.util.Arrays;
public class Sort {
public static void sortB(int[] arr) {
  for(int i = 0; i < arr.length; i += 1) {
    System.out.print(Arrays.toString(arr) + " -> ");
    for(int j = i; j > 0; j -= 1) {
      if(arr[j] < arr[j-1]) {
        int temp = arr[j-1];
        arr[j-1] = arr[j];
        arr[j] = temp;
      }
    }
    System.out.println(Arrays.toString(arr));
  }
}
}
```

```
Sort.sortB(new int[]{ 53, 83, 15, 45, 49 });
[53, 83, 15, 45, 49] ->
```