CSE 12 – Basic Data Structures

Prof. Christine Alvarado
[Slides borrowed/adapted from slides by Cynthia Lee]
Announcements

1. HW3 posted, code available tomorrow morning, but there’s plenty to do in the meantime
Faculty Coffee Hour
Wed April 16th 1:30pm - 3pm
Jacobs Hall 4th Floor, Room 4309

Get a chance to network with your CS professors over coffee provided by WIC. Talk about classes, ask for industry advice, or just have a quick chat over a cup of joe!
\( f_1 \) is \( \mathcal{O}(f_2) \)

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 \( f_1 \)
\( f(n) = O(g(n)), \) if there are positive constants \( c \) and \( n_0 \) such that \( f(n) \leq c \cdot g(n) \) for all \( n \geq n_0 \).

- 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 n₀ such that \( f(n) \leq c \cdot g(n) \) for all \( n \geq n₀ \).

- Obviously \( f₂ = O(f₁) \) because \( f₁ > f₂ \) (after about \( n=10 \), so we set \( n₀ = 10 \))
  - \( f₁ \) is clearly an upper bound on \( f₂ \) and that’s what big-O is all about
- But \( f₁ = O(f₂) \) as well!
  - We just have to use the “c” to adjust so \( f₂ \) that it moves above \( f₁ \)
Common Big-O confusions:

- 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 \( f_1 \).

- What about when \( n \) is less than 10? Isn’t \( f_2 \) larger than \( f_1 \)?
  Remember, we get to pick our \( n_0 \), and only consider \( n \) larger than \( n_0 \).
$f_1$ is $\Omega(f_2)$

A. TRUE
B. FALSE

Why or why not?
We say a function $f(n)$ is "big-omega" of another function $g(n)$, and write $f(n) = \Omega(g(n))$, if there are positive constants $c$ and $n_0$ such that:

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

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

- Obviously \( f_1 = O(f_2) \) 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 \( f_2 = Ω(f_1) \) as well!
  - We just have to use the “\( c \)” to adjust so \( f_1 \) that it moves below \( f_2 \).
\[ f(n) = \mathcal{O}(g(n)), \text{ if there are positive constants } c \text{ and } n_0 \text{ such that } f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0. \]

\[ f(n) = \Omega(g(n)), \text{ if there are positive constants } c \text{ and } n_0 \text{ such that } f(n) \geq c \cdot g(n) \text{ for all } n \geq n_0. \]

- Obviously \( f_1 = \mathcal{O}(f_2) \) 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-\( \Omega \) 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 \( f_2 \)
\[ f(n) = \mathcal{O}(g(n)), \text{ if there are positive constants } c \text{ and } n_0 \text{ such that } f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0. \]

\[ f(n) = \Omega(g(n)), \text{ if there are positive constants } c \text{ and } n_0 \text{ such that } f(n) \geq c \cdot g(n) \text{ for all } n \geq n_0. \]

\textbf{f}_1 \text{ is } \mathcal{O}(f_3) \]

A. TRUE  
B. FALSE  

Why or why not?
\( f(n) = \mathcal{O}(g(n)) \), if there are positive constants \( c \) and \( n_0 \) such that \( f(n) \leq c \cdot g(n) \) for all \( n \geq n_0 \).

\( f(n) = \Omega(g(n)) \), if there are positive constants \( c \) and \( n_0 \) such that \( f(n) \geq c \cdot g(n) \) for all \( n \geq n_0 \).

**\( f_3 \) is \( \mathcal{O}(f_1) \)**

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 ($f_3$).
\[ f(n) = \Theta(g(n)), \text{ if there are positive constants } c \text{ and } n_0 \text{ such that } f(n) \leq c \times g(n) \text{ for all } n \geq n_0. \]

\[ f(n) = \Omega(g(n)), \text{ if there are positive constants } c \text{ and } n_0 \text{ such that } f(n) \geq c \times g(n) \text{ for all } n \geq n_0. \]

**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 n₀ such that f(n) ≤ c * g(n) for all n ≥ n₀.

f(n) = Ω(g(n)), if there are positive constants c and n₀ such that f(n) ≥ c * g(n) for all n ≥ n₀.

**f₁ is Ω(f₃)**

A. TRUE  
B. FALSE

Why or why not?
\[ f_3 = \Omega(f_1) \text{ but } f_1 \neq \Omega(f_3) \]

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* \( f(n) = \Omega(g(n)) \), then \( f(n) = \Theta(g(n)) \).
- Basically it means that \( f(n) \) and \( g(n) \) are interchangeable.
- Examples:
  - \( 3n + 20 = \Theta(10n + 7) \)
  - \( 5n^2 + 50n + 3 = \Theta(5n^2 + 100) \)
$f_1$ 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 θ(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
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?

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
Extracting time cost from example code

Algorithm analysis starting with the algorithm
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

<table>
<thead>
<tr>
<th>Statements</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  float findAvg ( int []grades ){</td>
<td></td>
</tr>
<tr>
<td>2  float sum = 0;</td>
<td>1</td>
</tr>
<tr>
<td>3  int count = 0;</td>
<td>1</td>
</tr>
<tr>
<td>4  while ( count &lt; grades.length ) {</td>
<td>$n + 1$</td>
</tr>
<tr>
<td>5   sum += grades[count];</td>
<td>$n$</td>
</tr>
<tr>
<td>6   count++;</td>
<td>$n$</td>
</tr>
<tr>
<td>7 }</td>
<td></td>
</tr>
<tr>
<td>8   if ( grades.length &gt; 0 )</td>
<td>1</td>
</tr>
<tr>
<td>9     return sum / grades.length;</td>
<td></td>
</tr>
<tr>
<td>10  else</td>
<td></td>
</tr>
<tr>
<td>11    return 0.0f;</td>
<td></td>
</tr>
<tr>
<td>12 }</td>
<td></td>
</tr>
<tr>
<td>ALL</td>
<td>$3n+5$</td>
</tr>
</tbody>
</table>
Count how many times each line executes, then which $O(\ )$ most tightly and correctly characterizes the growth?

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

(assume $n = arr.length$)
Count how many times each line executes, then say which $O(\ )$ statement(s) is(are) true.

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

(assume $n = arr.length$)
Count how many times each line executes, then say which $O(\ )$ statement(s) is(are) true.

```java
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. $f(n) = \Theta(2^n)$  D. $f(n) = \Theta(n^3)$
B. $f(n) = \Theta(n^2)$  E. Other/none/more
C. $f(n) = \Theta(n)$  \hspace{1cm} (assume $n = arr.length$)