CSE 373: Data Structures and Algorithms

Lecture 5: Finishing up Asymptotic Analysis
Big-O, Big-Ω, Big-θ, little-o, little-ω & Amortized Analysis

Instructor: Lilian de Greef
Quarter: Summer 2017
Today:

• Announcements
• Big-O and Cousins
  • Big-Omega
  • Big-Theta
  • little-o
  • little-omega
• Average running time: Amortized Analysis
News about Sections

Updated times:
• **10:50** – 11:50am
• 12:00 – **1:00**pm

Bigger room!
• 10:50am section now in **THO 101**

Which section to attend:
• Last week, section sizes were unbalanced (~40 vs ~10 people)
• If you can, I encourage you to choose the 12:00 section to rebalance sizes
  • Helps the 12:00 TA’s feel less lonely
  • More importantly: improves TA:student ratio in sections (better for tailoring section to your needs)
Homework 1

• Due today at 5:00pm!

• A note about grading methods:
  • Before we grade, we’ll run a script on your code to replace your name with ### anonymized ### so we won’t know who you are as we grade it (to address unconscious bias).
  • It’s still good practice to have your name and contact info in the comments!
Homework 2

• Written homework about asymptotic analysis (no Java this time)
• Will be out this evening
• Due Thursday, July 6\textsuperscript{th} at 5:00pm
  • Because July 4\textsuperscript{th} is a holiday

• A note for help on homework:
  • Note that holidays means fewer office hours
  • Remember: although you cannot share solutions, you can talk to classmates about concepts or work through non-homework examples (e.g. from section) together.
  • Give these classmates credit, write their names at the top of your homework.
Big-O: Formal Definition

(Finishing up from last time)
Formal Definition of Big-O

Definition: \( f(n) \) is in \( O(g(n)) \) if there exist constants \( c \) and \( n_0 \) such that \( f(n) \leq c g(n) \) for all \( n \geq n_0 \)
More Practice with the Definition of Big-O

Let \( a(n) = 10n + 3n^2 \) and \( b(n) = n^2 \)

What are some values of \( c \) and \( n_0 \) we can use to show \( a(n) \in O(b(n)) \)?

Definition: \( f(n) \) is in \( O(g(n)) \) if there exist constants \( c \) and \( n_0 \) such that \( f(n) \leq c \cdot g(n) \) for all \( n \geq n_0 \)
Constants and Lower Order Terms

• The constant multiplier $c$ is what allows functions that differ only in their largest coefficient to have the same asymptotic complexity.

  Example:

  $$2n^2 \leq O(n^2)$$

  $$16,000,000n^2 + n + 3 \leq O(n^2)$$

• Eliminate lower-order terms because they become negligible as $n \to \infty$.

• Eliminate coefficients because we don't have "units of execution".

  • $3n^2$ vs $5n^2$ is meaningless without the cost of constant-time operations.
  • Can always re-scale anyways.
  • Do not ignore constants that are not multipliers! $n^3$ is not $O(n^2)$, $3^n$ is not $O(2^n)$. 
The constant multiplier $c$ is what allows functions that differ only in their largest coefficient to have the same asymptotic complexity.

For example, $g(n) = 3n^2$ and $h(n) = 9999n^2 + 9999n + 2$ and $f(n) = n^2$, $g(n)$ and $h(n)$ are both in $O(f(n))$. 
Analyzing “Worst-Case” Cheat Sheet

Basic operations take “some amount of” **constant time**
- Arithmetic (fixed-width)
- Assignment
- Access one Java field or array index
- *etc.*

(This is an *approximation* of reality: a very useful “lie”)

<table>
<thead>
<tr>
<th>Control Flow</th>
<th>Time Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consecutive statements</td>
<td>Sum of time of statement</td>
</tr>
<tr>
<td>Conditionals</td>
<td>Time of test plus slower branch</td>
</tr>
<tr>
<td>Loops</td>
<td>Sum of iterations * time of body</td>
</tr>
<tr>
<td>Method calls</td>
<td>Time of call’s body</td>
</tr>
<tr>
<td>Recursion</td>
<td>Solve <em>recurrence relation</em></td>
</tr>
</tbody>
</table>
Cousins of Big-O

Big-O, Big-Omega, Big-Theta, little-o, little-omega
**Big-O & Big-Omega**

**Big-O:** Upper Bound

The function $f(n)$ is in $O(g(n))$ if there exist constants $c$ and $n_0$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

**Big-Ω:** Lower Bound

The function $f(n)$ is in $Ω(g(n))$ if there exist constants $c$ and $n_0$ such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.
**Big-Θ:** Tight Bound

$f(n)$ is in $\Theta(g(n))$ if $f(n)$ is in both $O(g(n))$ and $\Omega(g(n))$

Use two different $c's$ $(c_1, \& c_2)$
little-o & little-omega

**little-o:**
$f(n)$ is in $o(g(n))$ if constants $c > 0$ there exists an $n_0$ s.t. $f(n) < c \ g(n)$ for all $n \geq n_0$

**little-ω:**
$f(n)$ is in $ω(g(n))$ if constants $c > 0$ there exists an $n_0$ s.t. $f(n) > c \ g(n)$ for all $n \geq n_0$
Practice Time!

Let \( f(n) = 75n^3 + 2 \) and \( g(n) = n^3 + 6n + 2n^2 \)

Then \( f(n) \) is in... (choose all that apply)

A. Big-O(g)
B. Big-Ω(g)
C. \( \theta(g) \)
D. little-o(g)
E. little-\( \omega(g) \)
Second Practice Time!

Let \( f(n) = 3^n \) and \( g(n) = n^3 \)

Then \( f(n) \) is in... (choose all that apply)

A. Big-O(g)
B. Big-\( \Omega \)(g)
C. \( \theta \)(g)
D. little-o(g)
E. little-\( \omega \)(g)

\[
f(n) = 4n^2
\]

\[
O(n^2) \quad \checkmark
\]

\[
o(n^2) \quad \times
\]

\[
\text{no value of } c \text{ & } n_0 \text{ for } f(n) \leq cg(n) \text{ for } n \geq n_0
\]

If little-\( \omega \) is true
\rightarrow Big-\( \Omega \) is true

If little-o is true
\rightarrow Big-O
Big-O, Big-Omega, Big-Theta

- Which one is more useful to describe asymptotic behavior?
  * \( \text{Big-} \Theta \text{ is more specific} \)

- A common error is to say \( O(f(n)) \) when you mean \( \theta(f(n)) \)
  - A linear algorithm is in both \( O(n) \) and \( O(n^5) \).
  - Better to say it is \( \theta(n) \).
  - That means that it is not, for example \( O(\log n) \).
Comments on Asymptotic Analysis

• Is choosing the lowest Big-O or Big-Theta the best way to choose the fastest algorithm?
  
  No!

  Sometimes we care about worst-case. Sometimes we care about average case.

• Big-O can use other variables (e.g. can sum all of the elements of an n-by-m matrix in $O(nm)$).
Amortized Analysis

How we calculate the average time!
Case Study: the Array Stack

What’s the **worst-case** running time of \texttt{push()}?

\[ \Theta(n) \quad O(n) \]

What’s the **average** running time of \texttt{push()}?

\[ \Theta(1) \]

Calculating the average: not based off of running a \textit{single operation}, but running \textit{many operations in sequence}.

Technique: **Amortized Analysis**
Amortized Cost

The **amortized cost** of $n$ operations is the worst-case total cost of the operations divided by $n$.

$$\text{if } T(n) = \text{worst case/upper bound}$$

$$\text{for } n = \# \text{ operation}s$$

$$\Rightarrow \text{Amortized cost} = \frac{T(n)}{n}$$
Amortized Cost

The **amortized cost** of $n$ operations is the worst-case total cost of the operations divided by $n$.

Practice:

- $n$ operations taking $O(n) \rightarrow$ amortized cost = \frac{O(n)}{n} = O(1)
- $n$ operations taking $O(n^3) \rightarrow$ amortized cost = \frac{O(n^3)}{n} = O(n^2)
- $n$ operations taking $O(n f(n)) \rightarrow$ amortized cost = \frac{O(n f(n))}{n} = O(f(n))
Example: Array Stack

What’s the amortized cost of calling \texttt{push()} \( n \) times if we double the array size when it’s full?

\( n \) operations

\( n \) pushes @ \( \mathcal{O}(1) \) each \( \rightarrow \) cost is \( n \)

\( \text{cost of resizing} = n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} \ldots \)

\( \text{(doubling array when full)} \quad \text{upper bound} = 2n \)

\( \text{total cost} = 3n \)

\( \text{amortized cost} = \frac{3n}{n} = 3 \)

The \textit{amortized cost} of \( n \) operations is the worst-case total cost of the operations divided by \( n \).
Another Perspective: Paying and Saving “Currency”

1 operation costs us 1$ to the computer.
Another Perspective: Paying and Saving “Currency”

Use $2 for each push:
$1 to computer,
$1 to bank

Potential Function

Spend our savings in the bank to resize. That way it only costs $1 to push(E)!