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 6th at 5:00pm
  • Because July 4th 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.
Formal Definition of Big-O

Definition: \( f(n) \) is in \( O(g(n)) \) if there exist constants \( c \) and \( n_0 \) such that \( f(n) \leq c \cdot g(n) \) 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:

  • Eliminate lower-order terms because

  • Eliminate coefficients because
    • $3n^2$ 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 $O(2^n)$
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 recurrence relation</td>
</tr>
</tbody>
</table>
Big-O & Big-Omega

**Big-O:**
\[ f(n) \text{ is in } O(g(n)) \text{ if there exist constants } c \text{ and } n_0 \text{ such that } f(n) \leq c g(n) \text{ for all } n \geq n_0 \]

**Big-Ω:**
\[ f(n) \text{ is in } Ω(g(n)) \text{ if there exist constants } c \text{ and } n_0 \text{ such that } f(n) \geq c g(n) \text{ for all } n \geq n_0 \]
**Big-Θ:**

$f(n)$ is in $\Theta(g(n))$ if $f(n)$ is in both $O(g(n))$ and $\Omega(g(n))$.
little-o & little-omega

**little-o:**
\[ f(n) \text{ is in } o(g(n)) \text{ if } \]
constants \( c > 0 \) there exists an \( n_0 \)
s.t. \( f(n) \ll c \cdot g(n) \) for all \( n \geq n_0 \)

**little-\( \omega \):**
\[ f(n) \text{ is in } \omega(g(n)) \text{ if } \]
constants \( c > 0 \) there exists an \( n_0 \)
s.t. \( f(n) \gg c \cdot g(n) \) 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-ω(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-$\omega$($g$)
Big-O, Big-Omega, Big-Theta

• Which one is more useful to describe asymptotic behavior?

• 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?

• 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 \( \text{push}() \) ?

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

Calculating the average: not based off of running a *single operation*, but running *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 $n$. 
Amortized Cost

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

Practice:

- \( n \) operations taking \( O(n) \) → amortized cost =
- \( n \) operations taking \( O(n^3) \) → amortized cost =
- \( n \) operations taking \( O(n \ f(n)) \) → amortized cost =
Example: Array Stack

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

The 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 1$ to the computer

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

Spend our savings in the bank to resize. That way it only costs $1 extra to push(E)!
Example #2: Queue made of Stacks

A sneaky way to implement Queue using two Stacks

Example walk-through:

- enqueue A
- enqueue B
- enqueue C
- dequeue
- enqueue D
- enqueue E
- dequeue
- dequeue
- dequeue
Example #2: Queue made of Stacks

A sneaky way to implement Queue using two Stacks

class Queue<E> {
    Stack<E> in = new Stack<E>();
    Stack<E> out = new Stack<E>();
    void enqueue(E x) { in.push(x); }
    E dequeue() {
        if (!out.isEmpty()) {
            while (!in.isEmpty()) {
                out.push(in.pop());
            }
        }
        return out.pop();
    }
}

Wouldn’t it be nice to have a queue of t-shirts to wear instead of a stack (like in your dresser)? So have two stacks
- in: stack of t-shirts go after you wash them
- out: stack of t-shirts to wear
- if out is empty, reverse in into out
Example #2: Queue made of Stacks (Analysis)

```java
class Queue<E> {
    Stack<E> in = new Stack<E>();
    Stack<E> out = new Stack<E>();
    void enqueue(E x) { in.push(x); }
    E dequeue() {
        if (out.isEmpty()) {
            while (!in.isEmpty()) {
                out.push(in.pop());
            }
        }
        return out.pop();
    }
}
```

Assume stack operations are (amortized) O(1). What’s the worst-case for dequeue()?

What operations did we need to do to reach that condition (starting with an empty Queue)?

Hence, what is the amortized cost?

So the average time for dequeue() is:
Example #2: Using “Currency” Analogy

“Spend” 2 coins for every enqueue – 1 to the “computer”, 1 to the “bank”.

Example walk-through:
- enqueue A
- enqueue B
- enqueue C
- enqueue D
- enqueue E
- dequeue

Potential Function
Example #3: (Parody / Joke Example)

Lectures are 1 hour long, 3 times per week, so I’m supposed to lecture for 27 hours this quarter.

If I end the first 26 lectures 5 minutes early, then I’d have “saved up” 130 minutes worth of extra lecture time. Then I could spend it all on the last lecture and can keep you here for 3 hours (bwahahahahaha)!

(After all, each lecture would still be 1 hour amortized time)
Wrapping up Amortized Analysis

• In what cases do we care more about the average / amortized run time?

• In what cases do we care more about the worst-case run time?