CSE 373: Data Structures and Algorithms
Lecture 4: Asymptotic Analysis part 3
Code Style, Recurrence Relations, Formal Big-O & Cousins

Instructor: Lilian de Greef
Quarter: Summer 2017
Code Style

Why does code style matter?
<table>
<thead>
<tr>
<th>Code Style</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Do</strong></td>
</tr>
</tbody>
</table>
Code Style Critique

```java
import java.util.Arrays;
public boolean function(int n) {
    boolean[] p = new boolean[10000];
    Arrays.fill(p, true);
    p[0]=p[1]=false;
    for (int i=2;i<p.length;i++) {
        if(p[i]) {
            for (int j=2;i*j<p.length;j++) {
                p[i*j]=false;
            }
        }
    }
    return p[n];
}
```
// Tells you whether a number is prime.
public boolean isPrime(int n) {
    // Make an array.
    boolean[] primes = new boolean[10000];
    // Fill the array with the value "true"
    // except for the first two indices.
    Arrays.fill(primes, true);
    primes[0]=primes[1]=false;

    // Loop over the array. As you do, check
    // if the current array value is true.
    // If it is, loop over the rest of the array
    // in increments of that current value
    // and set those indices to "false".
    for (int i=2;i<primes.length;i++) {
        if (primes[i]) {
            for (int j=2;i*j<primes.length;j++) {
                primes[i*j]=false;
            }
        }
    }

    return primes[n];
}
// Returns whether a given number is prime.
// Assumes number is less than 10000.
public boolean isPrime(int n) {

    // Assume all numbers are prime.
    boolean[] primes = new boolean[10000];
    Arrays.fill(primes, true);

    // We know 0 and 1 are not prime.
    primes[0] = false;
    primes[1] = false;

    // Eliminate numbers that are not prime
    // using the Sieve of Eratosthenes.
    for (int i=2; i<primes.length; i++) {

        // If the current number is prime, flag
        // all of its multiples as not prime.
        if (primes[i]) {
            for (int j=2; i*j<primes.length; j++) {
                primes[i*j] = false;
            }
        }
    }

    return primes[n];
}
// Constants and data members
static final int MAX_PRIME = 10000;
private boolean[] primes = new boolean[MAX_PRIME];

// An implementation of the Sieve of Eratosthenes.
// Fills our array of primes with "true" or "false"
// to match whether the index is prime.
public void fillSieve() {
    // Assume all numbers are prime.
    Arrays.fill(primes, true);

    // We know 0 and 1 are not prime.
    primes[0] = false;
    primes[1] = false;

    // Eliminate numbers that are not prime.
    for (int i = 2; i < primes.length; i++) {
        // If the current number is prime, flag
        // all of its multiples as not prime.
        if (primes[i]) {
            for (int j = 2; i * j < primes.length; j++) {
                primes[i * j] = false;
            }
        }
    }
}

// Returns whether a given number is prime.
// Assumes number is less than the class's maximum.
public boolean isPrime(int n) {
    return primes[n];
}
Recurrence Relations

How to calculate Big-O for recursive functions!
(Continued from last lecture)
Example #1: Towers of Hanoi

// Prints instructions for moving disks from one pole to another, where the three poles are labeled with integers "from", "to", and "other". // Code from rosettacode.org
public void move(int n, int from, int to, int other) {
    if (n == 1) {
        System.out.println("Move disk from pole " + from + " to pole " + to);
    } else {
        move(n - 1, from, other, to);
        move(1, from, to, other);
        move(n - 1, other, to, from);
    }
}
Example #1: Towers of Hanoi

```java
if (n == 1) {
    System.out.println("Move disk from pole " + from + " to pole " + to);
}

else {
    move(n - 1, from, other, to);
    move(1, from, to, other);
    move(n - 1, other, to, from);
}
```
Base Case:

Recurrence Relation:

(Example #1 continued)
(Example #1 continued)
Example #2: Binary Search

Find an integer in a sorted array
(Can also be done non-recursively)

// Requires the array to be sorted.
// Returns whether k is in array.
public boolean find(int[] arr, int k) {
    return help(arr, k, 0, arr.length);
}

private boolean help(int[] arr, int k, int lo, int hi) {
    int mid = (hi+lo)/2; // i.e., lo+(hi-lo)/2
    if(lo==hi) return false;
    if(arr[mid]==k) return true;
    if(arr[mid]<k) return help(arr, k, mid+1, hi);
    else return help(arr, k, lo, mid);
What is the recurrence relation?

// Requires the array to be sorted.
// Returns whether k is in array.
public boolean find(int[] arr, int k){
    return help(arr,k,0,arr.length);
}

private boolean help(int[] arr, int k, int lo, int hi) {
    int mid = (hi+lo)/2; // i.e., lo+(hi-lo)/2
    if(lo==hi) return false;
    if(arr[mid]==k) return true;
    if(arr[mid]< k) return help(arr,k,mid+1,hi);
    else return help(arr,k,lo,mid);
}

A. $2T(n-1) + 3$
B. $T(n-1)*T(n-1) + 3$
C. $T(n/2) + 3$
D. $T(n/2) * T(n/2) + 3$
Base Case:

Recurrence Relation:

(Example #2 continued)
(Example #2 continued)
Recap: Solving Recurrence Relations

1. Determine the recurrence relation. What is the base case?
   • \( T(n) = 3 + T(n/2) \) \( T(1) = 3 \)

2. “Expand” the original relation to find an equivalent general expression in terms of the number of expansions.
   • \( T(n) = 3 + 3 + T(n/4) \)
     = \( 3 + 3 + 3 + T(n/8) \)
     = ...
     = \( 3k + T(n/(2^k)) \)

3. Find a closed-form expression by setting the number of expansions to a value which reduces the problem to a base case
   • \( n/(2^k) = 1 \) means \( n = 2^k \) means \( k = \log_2 n \)
   • So \( T(n) = 10 \log_2 n + 8 \) (get to base case and do it)
   • So \( T(n) is O(\log n) \)
Common Recurrence Relations

Should know how to solve recurrences but helps to recognize some common ones:

\[ T(n) = O(1) + T(n-1) \quad \text{linear} \]
\[ T(n) = O(1) + 2T(n/2) \quad \text{linear} \]
\[ T(n) = O(1) + T(n/2) \quad \text{logarithmic } O(\log n) \]
\[ T(n) = O(1) + 2T(n-1) \quad \text{exponential} \]
\[ T(n) = O(n) + T(n-1) \quad \text{quadratic} \]
\[ T(n) = O(n) + T(n/2) \quad \text{linear (why?)} \]
\[ T(n) = O(n) + 2T(n/2) \quad O(n \log n) \]
Big-O Big Picture

with its formal definition
In terms of Big-O, which function has the faster asymptotic running time?

The graph shows two functions, $f(n)$ and $g(n)$, plotted against $n$, the input size, and the worst-case running time. The graph indicates that $f(n)$ grows faster than $g(n)$ as $n$ increases.
In terms of Big-O, which function has the faster asymptotic running time?

Graph showing the comparison of functions $f(n)$ and $g(n)$ with respect to worst-case running time and input size $n$. The graph indicates that as $n$ increases, $g(n)$ grows faster than $f(n)$. Therefore, $g(n)$ has the faster asymptotic running time.
In terms of Big-O, which function has the faster asymptotic running time?

Take-away:
Formal Definition of Big-O

“General Idea” explanation from last week:
Mathematical upper bound describing the behavior of how long a function
takes to run in terms of \( N \). (The “shape” as \( N \to \infty \))

Formal definition of Big-O:
Formal Definition of Big-O
Using the Formal Definition of Big-O

Definition: \( f(n) \) is in \( \mathcal{O}(g(n)) \) if there exist constants \( c \) and \( n_0 \) such that \( f(n) \leq c \, g(n) \) for all \( n \geq n_0 \)

To show \( f(n) \) is in \( \mathcal{O}(g(n)) \), pick a \( c \) large enough to “cover the constant factors” and \( n_0 \) large enough to “cover the lower-order terms”

Example:
Let \( f(n) = 3n^2+18 \) and \( g(n) = n^2 \)

Example:
Let \( f(n) = 3n^2+18 \) and \( g(n) = n^5 \)
Practice with the Definition of Big-O

Let \( f(n) = 1000n \) and \( g(n) = n^2 \)

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

**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 $f(n) \leq c g(n)$ 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 $O(2^n)$
Cousins of Big-O

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

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

**Big-Ω:**
\( f(n) \) is in \( Ω(g(n)) \) if there exist constants \( c \) and \( n_0 \) such that \( f(n) \geq c g(n) \) for all \( n \geq n_0 \)
Big-Theta

**Big- θ:**
\[ f(n) \text{ is in } \theta(g(n)) \text{ if } f(n) \text{ is in both } O(g(n)) \text{ and } \Omega(g(n)) \]
little-o & little-omega

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

**little-\( \omega \):**
\( f(n) \) is in \( \omega(g(n)) \) if
constants \( c > 0 \) there exists an \( n_0 \)
s.t. \( f(n) \geq c g(n) \) for all \( n \geq n_0 \)
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 $O(\log n)$
Notes on Worst-Case Analysis
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>
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))