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
Today:

• Code Style
• Recurrence Relations
• Formal Definition of Big-O
• Cousins of Big-O
  • Big-Omega
  • Big-Theta
  • little-o
  • little-omega
Code Style
Why does code style matter?

Easier for others to read & understand
- "future you"
- to debug

Full credit on assignments (courses)
Code Style

**Do**
- Nice comments
- Good variable names
  - Descriptive
  - Concise
- Proper spacing
  - Indentation

**Don’t**
- Over comments
- Magic numbers
- Boolean Zen
  
  ```python
  if (---) 
  return true
  else 
  return false
  ```
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];
}
Code Style Critique #4

uses constants 😊 (instead magic Numbers!)

Improvement:
Instead of having constant max prime, use variable to update!

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

**Base Case:** @ n = 1

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

1 execution \( \Rightarrow H(1) = 1 \)

**Recursive Step:**

else {
    move(n - 1, from, other, to);
    move(1, from, to, other);
    move(n - 1, other, to, from);
}

All together: \( H(n) = H(n-1) + H(1) + H(n-1) \)

\[ H(n) = 1 + 2H(n-1) \]
Base Case: \( H(1) = 1 \)

Recurrence Relation: \( H(n) = 1 + 2H(n-1) \)

Expanding (plug in for \( H(n) \)): 

1\(^{st}\) \[ H(n) = 1 + 2H(n-1) = 1 + 2\left[1 + 2H(n-1) - 1\right] \]

2\(^{nd}\) \[ = 1 + 2 + 4H(n-2) \]

3\(^{rd}\) \[ = 1 + 2 + 4 + 8H(n-3) \]

\( \vdots \)

\( k^{th}\) \[ = 2^0 + 2^1 + 2^2 + \cdots + 2^{k-1} + 2^k H(n-k) \]

(Example #1 continued)
$k^{th}$ expansion

$$H(n) = 2^k - 1 + 2^k H(n-k)$$

Base case is at $H(1) = 1$

Let's solve

$$n-k = 1$$

$$\Rightarrow k = n-1$$

Plug it in:

$$H(n) = 2^k - 1 + 2^k H(n-k)$$

$$= 2^{n-1} - 1 + 2^{n-1} H(n-(n-1))$$

$$= 2^{n-1} - 1 + 2^{n-1} H(1)$$

$$= 2^{n-1} - 1 + 2^{n-1} (1)$$

$$= 2^n - 1$$

(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?

```java
// 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. $T(n/2) * T(n/2) + 3$
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);
}

T(n) = C + T(n/2)  \quad T(1) = C
Base Case: \[ T(1) = c \]

Recurrence Relation:

\[
T(n) = c + T\left(\frac{n}{2}\right)
\]

\[
= c + c + T\left(\frac{n}{4}\right)
\]

\[
= c + c + c + T\left(\frac{n}{8}\right)
\]

\[
\vdots
\]

\[
= c^k + T\left(\frac{n}{2^k}\right)
\]

Base case: \( n = 1 \) \( \frac{1}{2}^k = 1 \) \( \Rightarrow n = 2^k \) \( \Rightarrow k = \log_2 n \)

\( \Rightarrow T(n) = c \log_2 n + c \) \( \Rightarrow T(n) = O(\log_2 n) \)

(Example #2 continued)
Recap: Solving Recurrence Relations

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

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?

- $f(n)$
- $g(n)$

Graph showing the comparison of worst-case running time for $f(n)$ and $g(n)$ as $n$ increases.
In terms of Big-O, which function has the faster asymptotic running time?
In terms of Big-O, which function has the faster asymptotic running time?

Take-away: Can't base behavior on the first values (even 10,000,000)

That's why asymptotic analysis is about $N \to \infty$
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:

$f(n)$ is in $O(g(n))$ if there exists constants $c$ and $n_0$ s.t.
$f(n) \leq cg(n)$ for all $n \geq n_0$. 
Definition: \( f(n) \) is in \( \mathcal{O}(g(n)) \) if there exist constants \( c \) and \( n_0 \) such that \( f(n) \leq c \cdot g(n) \) for all \( n \geq n_0 \).
Using the 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 \( n \geq n_0 \)

To show \( f(n) \) is in \( O(g(n)) \), 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 \( g(n) = n^2 \)

\( c = 5 \) \( n_0 = 16 \)

\( 3n^2 + 18 \leq 5n^2 \) \( \forall n \geq n_0 \) ?

\( 18 \leq 2n^2 \) \( \forall n \geq 16 \) ?

**Example:**

Let \( f(n) = 3n^2 + 18 \) and \( g(n) = n^5 \)

\( c = 3 \) \( n_0 = 16 \)

\( 3n^2 + 18 \leq 3n^5 \) \( \forall n \geq 16 \) ?

\( 18 \leq 3(n^5 - n^2) \)

\( 6 \leq n^5 - n^2 \)