CS 310: Order Notation (aka Big-O and friends)

Chris Kauffman

Week 1-2
Logistics

At Home

- Read Weiss Ch 1-4: Java Review
- Read Weiss Ch 5: Big-O
- Get your java environment set up
- Compile/Run code for Max Subarray problem from first lecture

Goals

- Finish up Course Mechanics
- Basic understanding of Big O and friends
Announcement: UMD Diversity in Computing Summit

- http://mcwic.cs.umd.edu/events/diversity
- Keynote, talks, networking for current students
- Monday, November 7, 2016
- College Park Marriott Hotel, Hyattsville, MD
- $35 for students before 10/1, $50 after

Through informative workshops and dynamic speakers, the Summit will emphasize inclusive computing efforts that address the positive impact that underrepresented groups have and will continue to have on the future of technology.
Course Mechanics

Finish up course mechanics from last time (first slide deck)
Algorithmic time/space complexity depend on problem size

- Often have some input parameter like \( n \) or \( N \) or \( (M, N) \) which indicates problem size
- Talk about time and space complexity as functions of those parameters
- Example: For an input array of size \( N \), the maximum element can be found in \( 5 \times N + 3 \) operations while the array can be sorted in \( 2N^2 + 11N + 7 \) operations.
- Big-O notation: bounding how fast functions grow based on input
It’s Show Time!

Not *The* Big O

Just Big O

$T(n)$ is $O(F(n))$ if there are positive constants $c$ and $n_0$ such that

- When $n \geq n_0$
- $T(n) \leq cF(n)$

Bottom line:

- If $T(n)$ is $O(F(n))$
- Then $F(n)$ grows as fast or faster than $T(n)$
Show It

Show

\[ f(n) = 2n^2 + 3n + 2 \text{ is } O(n^3) \]

- Pick \( c = 0.5 \) and \( n_0 = 6 \)

<table>
<thead>
<tr>
<th>( n )</th>
<th>( f(n) )</th>
<th>( 0.5n^3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>29</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>46</td>
<td>32</td>
</tr>
<tr>
<td>5</td>
<td>67</td>
<td>62</td>
</tr>
<tr>
<td>6</td>
<td>92</td>
<td>108</td>
</tr>
<tr>
<td>7</td>
<td>121</td>
<td>171</td>
</tr>
</tbody>
</table>

How about the opposite? Show

\[ g(n) = n^3 \text{ is } O(2n^2 + 3n + 2) \]
Basic Rules

- **Constant additions disappear**
  - $N + 5$ is $O(N)$

- **Constant multiples disappear**
  - $0.5N + 2N + 7$ is $O(N)$

- **Non-constant multiples multiply**:
  - Doing a constant operation $2N$ times is $O(N)$
  - Doing a $O(N)$ operation $N/2$ times is $O(N^2)$
  - Need space for half an array with $N$ elements is $O(N)$ space overhead

- **Function calls are not free** (including library calls)
  - Call a function which performs 10 operations is $O(1)$
  - Call a function which performs $N/3$ operations is $O(N)$
  - Call a function which copies object of size $N$ takes $O(N)$ time and uses $O(N)$ space
Bounding Functions

- **Big O:** *Upper* bounded by ...
  
  \[ 2n^2 + 3n + 2 \text{ is } O(n^3) \text{ and } O(2^n) \text{ and } O(n^2) \]

- **Big Omega:** *Lower* bounded by ...
  
  \[ 2n^2 + 3n + 2 \text{ is } \Omega(n) \text{ and } \Omega(\log(n)) \text{ and } \Omega(n^2) \]

- **Big Theta:** *Upper* and *Lower* bounded by
  
  \[ 2n^2 + 3n + 2 \text{ is } \Theta(n^2) \]

- **Little O:** *Upper* bounded by *but not lower* bounded by...
  
  \[ 2n^2 + 3n + 2 \text{ is } o(n^3) \]
# Growth Ordering of Some Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Leading Term</th>
<th>Big-Oh</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>$1, 5, c$</td>
<td>$O(1)$</td>
<td>$2.5, 85, 2c$</td>
</tr>
<tr>
<td>Log-Log</td>
<td>$\log(\log(n))$</td>
<td>$O(\log \log n)$</td>
<td>$10 + (\log \log n + 5)$</td>
</tr>
<tr>
<td>Log</td>
<td>$\log(n)$</td>
<td>$O(\log(n))$</td>
<td>$5 \log n + 2 \log(n^2)$</td>
</tr>
<tr>
<td>Linear</td>
<td>$n$</td>
<td>$O(n)$</td>
<td>$2.4n + 10$</td>
</tr>
<tr>
<td>N-log-N</td>
<td>$n \log n$</td>
<td>$O(n \log n)$</td>
<td>$3.5n \log n + 10n + 8$</td>
</tr>
<tr>
<td>Super-linear</td>
<td>$n^{1.\times}$</td>
<td>$O(n^{1.\times})$</td>
<td>$2n^{1.2} + 3n \log n - n + 2$</td>
</tr>
<tr>
<td>Quadratic</td>
<td>$n^2$</td>
<td>$O(n^2)$</td>
<td>$0.5n^2 + 7n + 4$</td>
</tr>
<tr>
<td>Cubic</td>
<td>$n^3$</td>
<td>$O(n^3)$</td>
<td>$0.1n^3 + 8n^{1.5} + \log(n)$</td>
</tr>
<tr>
<td>Exponential</td>
<td>$a^n$</td>
<td>$O(2^n)$</td>
<td>$8(2^n) - n + 2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O(10^n)$</td>
<td>$100n^{500} + 2 + 10^n$</td>
</tr>
<tr>
<td>Factorial</td>
<td>$n!$</td>
<td>$O(n!)$</td>
<td>$0.25n! + 10n^{100} + 2n^2$</td>
</tr>
</tbody>
</table>
Constant Time Operations

The following take $O(1)$ Time

- Arithmetic operations (add, subtract, divide, modulo)
  - Integer ops usually practically faster than floating point
- Accessing a stack variable
- Accessing a field of an object
- Accessing a single element of an array
- Doing a primitive comparison (equals, less than, greater than)
- Calling a function/method but NOT waiting for it to finish

The following take more than $O(1)$ time (how much)?

- Raising an arbitrary number to arbitrary power
- Allocating an array
- Checking if two Strings are equal
- Determining if an array or ArrayList contains() an object
Common Patterns

- **Adjacent Loops Additive**: $2 \times n$ is $O(n)$
  
  ```java
  for(int i=0; i<N; i++){
      blah blah blah;
  }
  for(int j=0; j<N; j++){
      yakkety yack;
  }
  ```

- **Nested Loops Multiplicatively usually polynomial**
  
  - 1 loop, $O(n)$
  - 2 loops, $O(n^2)$
  - 3 loops, $O(n^3)$

- **Repeated halving usually involves a logarithm**
  
  - Binary search is $O(\log n)$
  - Fastest sorting algorithms are $O(n \log n)$
  - Proofs are harder, require solving recurrence relations

Lots of special cases so be careful
Practice

Two functions to reverse an array. Discuss
- Big-O estimates of runtime of both
- Big-O estimates of memory overhead of both
  - Memory overhead is the amount of memory in addition to the input required to complete the method
- Which is practically better?
- What are the exact operation counts for each method?

reverseE

```java
public static void reverseE(Integer a[]){
    int n = a.length;
    Integer b[] = new Integer[n];
    for(int i=0; i<n; i++){
        b[i] = a[n-1-i];
    }
    for(int i=0; i<n; i++){
        a[i] = b[i];
    }
}
```

reverseI

```java
public static void reverseI(Integer a[]){
    int n = a.length;
    for(int i=0; i<n/2; i++){
        int tmp = a[i];
        a[i] = a[n-1-i];
        a[n-1-i] = tmp;
    }
    return;
}
```
public static String toString( Object [ ] arr )
{
    String result = " [";
    for( String s : arr )
        result += s + " ";
    result += " ]";
    return result;
}

- Give a Big-O estimate for the runtime
- Give a Big-O estimate for the memory overhead
Multiple Input Size

What if "size" has two parameters?

- $m \times n$ matrix
- Graph with $m$ vertices and $n$ edges
- Network with $m$ computers and $n$ cables between them

Exercise: Sum of a Two-D Array
Give the runtime complexity of the following method.

```java
public int sum2D(int[][] A){
    int M = A.length;
    int N = A[0].length;
    int sum = 0;
    for(int i=0; i<M; i++){
        for(int j=0; j<N; j++){
            sum += A[i][j];
        }
    }
    return sum;
}
```
What if I have no idea?

Analyzing a complex algorithm is hard. More in CS 483.
  ▶ Most analyses in here will be straight-forward
  ▶ Mostly use the common patterns
If you haven’t got a clue looking at the code, *run it and check*
  ▶ This will give you a much better sense
### Observed Runtimes of Maximum Subarray

<table>
<thead>
<tr>
<th>$N$</th>
<th>$O(N^3)$</th>
<th>$O(N^2)$</th>
<th>$O(N \log N)$</th>
<th>$O(N)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.000001</td>
<td>0.000000</td>
<td>0.000001</td>
<td>0.000000</td>
</tr>
<tr>
<td>100</td>
<td>0.000288</td>
<td>0.000019</td>
<td>0.000014</td>
<td>0.000005</td>
</tr>
<tr>
<td>1,000</td>
<td>0.223111</td>
<td>0.001630</td>
<td>0.000154</td>
<td>0.000053</td>
</tr>
<tr>
<td>10,000</td>
<td>218</td>
<td>0.133064</td>
<td>0.001630</td>
<td>0.000533</td>
</tr>
<tr>
<td>100,000</td>
<td>NA</td>
<td>13.17</td>
<td>0.017467</td>
<td>0.005571</td>
</tr>
<tr>
<td>1,000,000</td>
<td>NA</td>
<td>NA</td>
<td>0.185363</td>
<td>0.056338</td>
</tr>
</tbody>
</table>

Weiss pg 203
Idealized Functions

Smallish Inputs

Larger Inputs
Where did this data come from?

Does this plot confirm our analysis?

How would we check?
Playing with MaxSumTestBetter.java

Let's generate part of the data, demo in w01-1-code/MaxSumTestBetter.java

- **Edit**: Running a main, \( n=100 \) to 100,000, multiply by 10
- Try in DrJava
- Demo interactive loop
Analysis

Linear

\[
\text{Coefficients:} \quad \begin{array}{ccc}
\text{Estim} & \text{Pr}(>|t|) \\
(\text{Intercept}) & 7.26 & <2e-16 \; *** \\
poly(N, 1) & 16.25 & <2e-16 \; *** \\
poly(N, 2) & -0.34 & 0.287 \\
poly(N, 3) & -0.01 & 0.962 \\
\end{array}
\]

Why these coefficients?

Quadratic

\[
\text{Coefficients:} \quad \begin{array}{ccc}
\text{Estim} & \text{Pr}(>|t|) \\
(\text{Intercept}) & 83.89 & <2e-16 \; *** \\
poly(N, 1) & 278.16 & <2e-16 \; *** \\
poly(N, 2) & 54.75 & <2e-16 \; *** \\
poly(N, 3) & -0.24 & 0.562 \\
\end{array}
\]
Take-Home

Today
Order Analysis captures big picture of algorithm complexity
- Different functions grow at different rates
- Big O upper bounds
- Big Theta tightly bounds

Next Time
- What are the limitations of Big-O?
- Reading: finish Ch 5, Ch 15 on ArrayList
- Suggested practice: Exercises 5.39 and 5.44 which explore string concatenation, why obvious approach is slow for lots of strings, alternatives