CSE332: Data Abstractions
Lecture 3: Asymptotic Analysis

Tyler Robison (covering for James Forgarty)
Winter 2012
Overview

• Asymptotic analysis
  – Why we care
  – Big Oh notation
  – Examples
  – Caveats & miscellany
  – Evaluating an algorithm
  – Big Oh’s family
  – Recurrence relations for analysis
What do we want to analyze?

• Correctness
• Performance: Algorithm’s speed or memory usage: our focus
  – Change in speed as the input grows
    • n increases by 1
    • n doubles
  – Comparison between 2 algorithms
• Security
• Reliability
• Sometimes other properties (‘stable’ sorts)
Gauging performance

• Uh, why not just run the program and time it?
  – Too much variability; not reliable:
    • Hardware: processor(s), memory, etc.
    • OS, version of Java, libraries, drivers
    • Choice of input
    • Programs running in the background, OS stuff, etc.: several executions on the same computer with the same settings may well yield different results
    • Implementation dependent
  – Timing doesn’t really evaluate the algorithm; it evaluates its implementation in one very specific scenario
  – As computer scientists, we are more interested in the algorithm itself
Gauging performance (cont.)

• At the core of CS is a backbone of theory & mathematics
  – Examine the algorithm itself, mathematically, not the implementation
  – Reason about performance as a function of n; not just ‘it runs fast on this particular test file’
  – Be able to mathematically prove things about performance

• Yet, timing has its place
  – In the real world, we do want to know whether implementation A runs faster than implementation B on data set C
  – Ex: Benchmarking graphics cards
  – May do some timing in projects

• Evaluating an algorithm? Use asymptotic analysis
• Evaluating an implementation of hardware/software? Timing can be useful
Overview

• Asymptotic analysis
  – Why we care
  – Big Oh notation
  – Examples
  – Caveats & miscellany
  – Evaluating an algorithm
  – Big Oh’s family
  – Recurrence relations for analysis
Big-Oh

- Say we’re given 2 run-time functions $f(n)$ & $g(n)$ for input $n$
- The Definition: $f(n)$ is in $O(g(n))$ iff there exist positive constants $c$ and $n_0$ such that
  $$f(n) \leq c \cdot g(n), \text{ for all } n \geq n_0.$$

- The Idea: Can we find an $n_0$ such that $g$ is always greater than $f$ from there on out?
  
  $c$: We are allowed to multiply $g$ by a constant value (say, 10) to make $g$ larger (more on why this is here in a moment)

$O(g(n))$ is really a set of functions whose asymptotic behavior is less than or equal that of $g(n)$

Think of ‘$f(n)$ is in $O(g(n))$’ as $f(n) \leq g(n)$ (sort of)
• **The Intuition:**
  – Take functions \( f(n) \) & \( g(n) \), consider only the most significant term and remove constant multipliers:
    - \( 5n+3 \rightarrow n \)
    - \( 7n+.5n^2+2000 \rightarrow n^2 \)
    - \( 300n+12+n\log n \rightarrow n\log n \)
    - \(-n \rightarrow ??? \) What does it mean to have a negative run-time?
  – Then compare the functions; if \( f(n) \leq g(n) \), then \( f(n) \) is in \( O(g(n)) \)
  – Do NOT ignore constants that are not additions or multipliers:
    - \( n^3 \) is \( O(n^2) \): FALSE
    - \( 3^n \) is \( O(2^n) \): FALSE
  – When in doubt, refer to the definition (examples in a moment)
Examples

• True or false?

1. 4+3n is O(n)  
   True
2. n+2logn is O(logn)  
   False
3. logn+2 is O(1)  
   False
4. n^{50} is O(1.01^n)  
   True
5. There exists $\alpha>1.0$ s.t. $\alpha^n$ is O($n^\beta$)  
   False
   For some finite $\beta$
Examples (cont.)

• For $f(n)=4n$ & $g(n)=n^2$, prove $f(n)$ is in $O(g(n))$
  – A valid proof (for our purposes) is to find valid $c$ & $n_0$
  – When $n=4$, $f=16$ & $g=16$; this is the crossing over point
  – Say $n_0 = 4$, and $c=1$
  – How many possible answers $(c,n_0)$ are there?
    • *Infinitely many:
      ex: $n_0 = 78$, and $c=42$

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

Examples (cont.)

• For $f(n) = n^3$ & $g(n) = 2^n$, prove $f(n)$ is in $O(g(n))$
  – Possible answer: $n_0 = 11$, $c = 1$

The Definition: $f(n)$ is in $O(g(n))$ iff there exist positive constants $c$ and $n_0$ such that

$$f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0.$$
What’s with the c?

• To capture this notion of similar asymptotic behavior, we allow a constant multiplier (called c)

• Consider:
  
  \[ f(n) = 7n + 5 \]
  
  \[ g(n) = n \]

• These have the same asymptotic behavior (linear), so \( f(n) \) is in \( O(g(n)) \) even though \( f \) is always larger

• There is no \( n_0 \) such that \( f(n) \leq g(n) \) for all \( n \geq n_0 \)

• The ‘c’ in the definition allows for that; it allows us to ‘throw out constant factors’

• To prove \( f(n) \) is in \( O(g(n)) \), have \( c = 12, n_0 = 1 \)
# Big Oh: Common Categories

*From fastest to slowest*

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>constant (same as $O(k)$ for constant $k$)</td>
</tr>
<tr>
<td>$O(\log n)$</td>
<td>logarithmic</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>linear</td>
</tr>
<tr>
<td>$O(n \log n)$</td>
<td>“$n \log n$”</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>quadratic</td>
</tr>
<tr>
<td>$O(n^3)$</td>
<td>cubic</td>
</tr>
<tr>
<td>$O(n^k)$</td>
<td>polynomial (where $k$ is a constant)</td>
</tr>
<tr>
<td>$O(k^n)$</td>
<td>exponential (where $k$ is any constant &gt; 1)</td>
</tr>
</tbody>
</table>

Usage note: “exponential” does not mean “grows really fast”, it means “grows at rate proportional to $k^n$ for some $k>1$”

- A savings account accrues interest exponentially ($k=1.01$?)

Where does $\log^2 n$ fit in?
Where does $\log\log n$ fit in?
Caveats

- Asymptotic complexity focuses on behavior of the algorithm for large $n$ and is independent of any computer/coding trick, but results can be misleading
  - Example: $n^{1/10}$ vs. $\log n$
    - Asymptotically $n^{1/10}$ grows more quickly
    - But the “cross-over” point is around $5 \times 10^{17}$
    - So if you have input size less than $2^{58}$, prefer $n^{1/10}$
More Caveats

- Even for more common functions, comparing $O()$ for small $n$ values can be misleading
  - Quicksort: $O(n \log n)$ (expected)
  - Insertion Sort: $O(n^2)$ (expected)
  - Yet in reality Insertion Sort is faster for small $n$’s
  - We’ll learn about these sorts later

- Usually talk about an algorithm being $O(n)$ or whatever
  - But you can also prove bounds for entire problems
  - Ex: Sorting cannot take place faster than $O(n \log n)$ in the worst case (assuming it’s sequential and comparison-based; more on these later)
Miscellaneous

• Not uncommon to evaluate for:
  – Best-case
  – Worst-case
  – ‘Expected case’

• What are the run-times for BST lookup?
  – Best \( O(1) \) – find at root
  – Worst \( O(n) \) – tree is 1 long branch
  – ‘Expected’ \( O(\log n) \) – complicated; see book
Notational Notes

• We say \((3n^2+17)\) is in \(O(n^2)\)
  – Confusingly, we also say/write:
    • \((3n^2+17)\) is \(O(n^2)\)
    • \((3n^2+17) = O(n^2)\) (very common; in the book)
      – But it’s not ‘=‘ as in ‘equality’:
      – We would never say \(O(n^2) = (3n^2+17)\)

• Perhaps the most accurate notation is
  \(f(n)\in O(g(n))\)
  – Because \(O(g(n))\) is a set of functions
Analyzing code (worst case)

Basic operations take “some amount of” constant time:
- Arithmetic (fixed-width)
- Assignment to a variable
- Access one Java field or array index
- Etc.

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

<table>
<thead>
<tr>
<th>Basic operations</th>
<th>Calculations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consecutive statements</td>
<td>Sum of times</td>
</tr>
<tr>
<td>Conditionals</td>
<td>Time of test plus slower branch</td>
</tr>
<tr>
<td>Loops</td>
<td>Sum of iterations</td>
</tr>
<tr>
<td>Calls</td>
<td>Time of call’s body</td>
</tr>
<tr>
<td>Recursion</td>
<td>Solve recurrence equation</td>
</tr>
</tbody>
</table>
Analyzing code

What are the run-times for the following code:

1. for(int i=0; i<n; i++) \(O(1)\) \(O(n)\)
2. for(int i=0; i<=n+100; i+=14) \(O(1)\) \(O(n)\)
3. for(int i=0; i<n; i++) for(int j=0; j<i; j++) \(O(1)\) \(O(n^2)\)
4. for(int i=0; i<n; i++) for(int j=0; j<n; j++) \(O(n)\) \(O(n^3)\)
5. for(int i=1; i<n; i*=2) \(O(1)\) \(O(\log n)\)
6. for(int i=0; i<n; i++) if(m(i)) \(O(n)\) else \(O(1)\) Depends on \(m()\); worst: \(O(n^2)\)
Big Oh’s Family

• Big Oh: Upper bound: $O( f(n) )$ is the set of all functions asymptotically less than or equal to $f(n)$: ‘≤’ of functions
  – $g(n)$ is in $O( f(n) )$ if there exist constants $c$ and $n_0$ such that $g(n) \leq c f(n)$ for all $n \geq n_0$

• Big Omega: Lower bound: $\Omega( f(n) )$ is the set of all functions asymptotically greater than or equal to $f(n)$: ‘≥’ of functions
  – $g(n)$ is in $\Omega( f(n) )$ if there exist constants $c$ and $n_0$ such that $g(n) \geq c f(n)$ for all $n \geq n_0$

• Big Theta: Tight bound: $\theta( f(n) )$ is the set of all functions asymptotically equal to $f(n)$: ‘=’ of functions
  – Intersection of $O( f(n) )$ and $\Omega( f(n) )$ (use different constants)
Regarding use of terms

Common error is to say $O(f(n))$ when you mean $\theta(f(n))$

– People often say $O()$ to mean a tight bound
– Say we have $f(n)=n$; we could say $f(n)$ is in $O(n)$, which is true, but only conveys the upper-bound
– Somewhat incomplete; instead say it is $\theta(n)$
– This gives us a tighter bound

Less common notation:

– “little-oh”: like “big-Oh” but strictly less than
  • Example: $n$ is $o(n^2)$ but not $o(n)$
– “little-omega”: like “big-Omega” but strictly greater than
  • Example: $n$ is $\omega(\log n)$ but not $\omega(n)$
Recurrence Relations

• Computing run-times gets interesting with recursion

• Say we want to perform some computation recursively on a list of size n
  – Conceptually, in each recursive call we:
    • Perform some amount of work, call it \( w(n) \)
    • Call the function recursively with a smaller portion of the list

So, if we do \( w(n) \) work per step, and reduce the \( n \) in the next recursive call by 1, we do total work:

\[
T(n) = w(n) + T(n-1)
\]

With some base case, like \( T(1) = 5 = O(1) \)
Recursive version of sum array

Recursive:
– Recurrence is 
  \( k + k + \ldots + k \)
  for \( n \) times

```java
int sum(int[] arr){
    return help(arr,0);
}
int help(int[] arr, int i) {
    if(i==arr.length)
        return 0;
    return arr[i] + help(arr, i+1);
}
```

Recurrence Relation: \( T(n) = O(1) + T(n-1) \)
Recurrence Relations (cont.)

Say we have the following recurrence relation:
\[ T(n) = 2 + T(n-1) \]
\[ T(1) = 5 \]
Now we just need to solve it; that is, reduce it to a closed form

Start by writing it out:
\[ T(n) = 2 + T(n-1) = 2 + 2 + T(n-2) = 2 + 2 + 2 + T(n-3) \]
\[ = 2 + 2 + 2 + \ldots + 2 + T(1) = 2 + 2 + 2 + \ldots + 2 + 5 \]
\[ = 2k + 5, \text{ where } k \text{ is the } \# \text{ of times we expanded } T() \]
We expanded it out \( n-1 \) times, so
\[ T(n) = 2(n-1) + 5 = 2n + 3 = O(n) \]
Example: Find k

Find an integer in a sorted array

```java
// requires array is sorted
// returns whether k is in array
boolean find(int[] arr, int k){
    ???
}
```
Linear search

Find an integer in a *sorted* array

```java
// requires array is sorted
// returns whether k is in array
boolean find(int[] arr, int k){
    for(int i=0; i < arr.length; ++i)
        if(arr[i] == k)
            return true;
    return false;
}
```

Best case: 6ish steps = $O(1)$
Worst case: 6ish*(arr.length) = $O(arr.length) = O(n)$
Binary search

Find an integer in a *sorted* array

- Can also be done non-recursively (same run-time)

```java
// requires array is sorted
// returns whether k is in array
boolean find(int[] arr, int k){
    return help(arr,k,0,arr.length);
}

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);
}
```
Binary search

Best case: 8ish steps = O(1)
Worst case:
\[ T(n) = 10ish + T(n/2) \text{ where } n \text{ is hi-lo} \]

```java
// requires array is sorted
// returns whether k is in array
boolean find(int[] arr, int k) {
    return help(arr, k, 0, arr.length);
}

boolean help(int[] arr, int k, int lo, int hi) {
    int mid = (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);
}
```
Solving Recurrence Relations

1. Determine the recurrence relation. What is the base case?
   - \( T(n) = 10 + T(n/2) \) \( T(1) = 8 \)

2. “Expand” the original relation to find an equivalent general expression in terms of the number of expansions.
   - \( T(n) = 10 + 10 + T(n/4) \)
     = \( 10 + 10 + 10 + T(n/8) \)
     = ...
     = \( 10k + T(n/(2^k)) \) where \( k \) is the # of expansions

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 \( O(\log n) \)
Linear vs Binary Search

• So binary search is $O(\log n)$ and linear is $O(n)$
  – Given the constants, linear search could still be faster for small values of $n$

Example w/ hypothetical constants:
What about a binary version of sum?

```c
int sum(int[] arr){
    return help(arr,0,arr.length);
}
int help(int[] arr, int lo, int hi) {
    if(lo==hi) return 0;
    if(lo==hi-1) return arr[lo];
    int mid = (hi+lo)/2;
    return help(arr,lo,mid) + help(arr,mid,hi);
}
```

Recurrence is \( T(n) = O(1) + 2T(n/2) = O(n) \)
(Proof left as an exercise)

“Obvious”: have to read the whole array
You can’t do better than \( O(n) \)
Or can you...

We’ll see a parallel version of this much later
With \( \infty \) processors, \( T(n) = O(1) + 1T(n/2) = O(\log n) \)
Another example

• $T(n) = n + 2T(n/2), \ T(1) = c$
  – Any guesses as to what algorithm(s) this represents?
    • Mergesort & quicksort (assuming good pivot selection)
  – Any guesses as to what the closed form for this is?
    • $O(n \log n)$
Really common recurrences

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

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

Note big-Oh can also use more than one variable (graphs: vertices & edges)

- Example: you can (and will in proj3!) sum all elements of an \( n \)-by-\( m \) matrix in \( O(nm) \)