CSE332: Data Abstractions

Lecture 3: Asymptotic Analysis

Tyler Robison

Summer 2010

Overview

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

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

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
    - Programs running in the background
    - Implementation dependent
    - Choice of input
  - Timing doesn't really evaluate the algorithm; it evaluates its implementation in one very specific scenario

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
  - 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
  - Will do some timing in project 3 (and in 2, a bit)
- Evaluating an algorithm? Use asymptotic analysis
- Evaluating an implementation of hardware/software? Timing can be useful

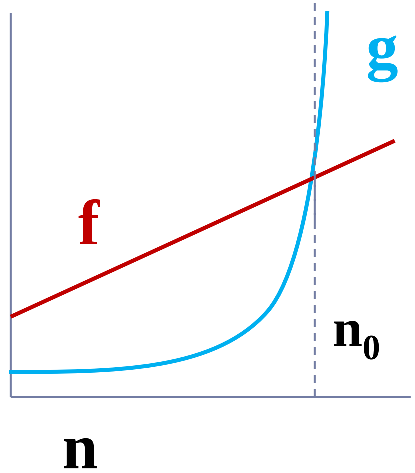
Big-Oh

- 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

  $$f(n) \leq c\, g(n), \text{ for all } n \geq n_0.$$

- The Idea: Can we find an n_0 such that g is always greater than f from there on out?

  We are allowed to multiply g by a constant value (say, 10) to make g larger



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) ≤ g(n) (sort of)

or 'f(n) is in O(g(n))' as g(n) is an upper-bound for f(n) (sort of)

Big Oh (cont.)

- 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 multipliers:
    - n^3 is $O(n^2)$: FALSE
    - 3^n is $O(2^n)$: FALSE
  - When in doubt, refer to the definition

Examples

- True or false?

1. $4+3n$ is $O(n)$ True
2. $n+2\log n$ is $O(\log n)$ False
3. $\log n+2$ is $O(1)$ False
4. n^{50} is $O(1.1^n)$ True

Examples (cont.)

- For $f(n)=4n$ & $g(n)=n^2$, prove $f(n)$ is in $O(g(n))$
  - A valid proof 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 $c=1$
  - (Infinitely) Many possible choices: ex: $n_0 = 78$, and $c=42$ works fine

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

$$f(n) \le c\, g(n) \text{ for all } n \ge n_0.$$

Examples (cont.)

- For $f(n)=n^4$ & $g(n)=2^n$, prove f(n) is in O(g(n))
  - Possible answer: $n_0=20$, $c=1$

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

$$f(n) \le c\, g(n) \text{ for all } n \ge n_0.$$

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 positive n_0 such that $f(n) \leq g(n)$ for all $n \geq n_0$
- The 'c' in the definition allows for that
- To prove f(n) is in O(g(n)), have $c=12$, $n_0=1$

Big Oh: Common Categories

From fastest to slowest

$O(1)$	constant (same as $O(k)$ for constant k)
$O(\texttt{log}\ n)$	logarithmic
$O(n)$	linear
$O(n\ \texttt{log}\ n)$	"n **`log`** *n*"
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(n^k)$	polynomial (where is k is an constant)
$O(k^n)$	exponential (where k is any constant > 1)

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?)

Caveats

- Asymptotic complexity focuses on behavior 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 $5 * 10^{17}$
    - So if you have input size less than 2^{58}, prefer $n^{1/10}$

Caveats

- Even for more common functions, comparing O() for small n values can be misleading
  - Quicksort: O(nlogn) (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 prove bounds for entire problems
    - Ex: No algorithm can do better than logn in the worst case for finding an element in a sorted array, without parallelism

Miscellaneous

- Not uncommon to evaluate for:
  - Best-case
  - Worst-case
  - 'Expected case'
- So we say $(3n^2+17)$ **is in** $O(n^2)$
  - Confusingly, we also say/write:
    - $(3n^2+17)$ **is** $O(n^2)$
    - $(3n^2+17)$ $=$ $O(n^2)$
  - But it's not '=' as in 'equality':
    - We would never say $O(n^2) = (3n^2+17)$

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".)

Consecutive statements	Sum of times
Conditionals	Time of test plus slower branch
Loops	Sum of iterations
Calls	Time of call's body
Recursion	Solve *recurrence equation (in a bit)*

Analyzing code

What is the run-time for the following code when

1. for(int i=0;i<n;i++) O(1) — $O(n)$
2. for(int i=0;i<n;i++) O(i) — $O(n^2)$
3. for(int i=0;i<n;i++) for(int j=0;j<n;j++) O(n) — $O(n^3)$

Big Oh's Family

- Big Oh: Upper bound: $O(\text{ f}(n)\text{ })$ is the set of all functions asymptotically less than or equal to $\text{f}(n)$
  - $\text{g}(n)$ is in $O(\text{ f}(n)\text{ })$ if there exist constants c and n_0 such that

$$\text{g}(n) \leq c\,\text{f}(n) \text{ for all } n \geq n_0$$

- Big Omega: Lower bound: $\Omega(\text{ f}(n)\text{ })$ is the set of all functions asymptotically greater than or equal to $\text{f}(n)$
  - $\text{g}(n)$ is in $\Omega(\text{ f}(n)\text{ })$ if there exist constants c and n_0 such that

$$\text{g}(n) \geq c\,\text{f}(n) \text{ for all } n \geq n_0$$

- Big Theta: Tight bound: $\theta(\text{ f}(n)\text{ })$ is the set of all functions asymptotically equal to $\text{f}(n)$
  - Intersection of $O(\text{ f}(n)\text{ })$ and $\Omega(\text{ f}(n)\text{ })$ (use *different* c values)

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 $\theta(n)$
- That means that it is not, for example $O(\mathtt{log}\ n)$

Less common notation:

- "little-oh": like "big-Oh" but strictly less than
  - Example: sum is $o(n^2)$ but not $o(n)$
- "little-omega": like "big-Omega" but strictly greater than
  - Example: sum is $\omega(\mathtt{log}\ n)$ 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

```
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: $T(n) = O(1) + T(n\text{-}1)$

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:

$T(n)=2+T(n-1)=2+2+T(n-2)=2+2+2+T(n-3)$

$=2+2+2+\ldots+2+T(1)=2+2+2+\ldots+2+5$

So it looks like

$T(n)=2(n-1)+5=2n+3=O(n)$

Example: Find k

2	**3**	**5**	**16**	**37**	**50**	**73**	**75**	**126**

Find an integer in a *sorted* array

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

Linear search

2	3	5	16	37	50	73	75	126

Find an integer in a *sorted* array

```
// 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

2	3	5	16	37	50	73	75	126

Find an integer in a *sorted* array

- Can also be done non-recursively but “doesn’t matter” here

```
// 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) where n is hi-lo

```
// 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);
}
```

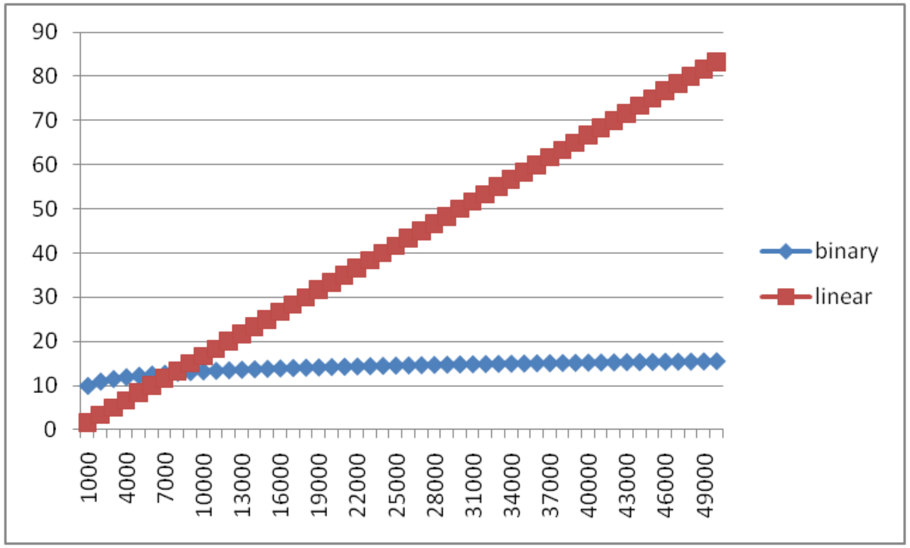
Solving Recurrence Relations

1. Determine the recurrence relation. What is the base case?
   - $T(n) = 10 + T(n/2)$ $\qquad$ $T(1) = 8$
2. "Expand" the original relation to find an equivalent general expression *in terms of the number of expansions*.
   - $T(n) = 10 + 10 + T(n/4)$
     $= 10 + 10 + 10 + T(n/8)$
     $= \ldots$
     $= 10k + T(n/(2^k))$ 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 $O(\texttt{log}\ n)$ 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?

```
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 ∞ processors, $T(n) = O(1) + 1T(n/2) = O(\log n)$

Really common recurrences

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

$T(n) = O(1) + T(n\text{-}1)$	linear
$T(n) = O(1) + 2T(n/2)$	linear
$T(n) = O(1) + T(n/2)$	logarithmic
$T(n) = O(1) + 2T(n\text{-}1)$	exponential
$T(n) = O(n) + T(n\text{-}1)$	quadratic
$T(n) = O(n) + T(n/2)$	linear
$T(n) = O(n) + 2T(n/2)$	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 $O(nm)$