CSE 332: Data Abstractions
Lecture 3: Asymptotic Analysis

Ruth Anderson
Spring 2013
Announcements

• **Project 1** – phase A due next Wed
• **Homework 1** – due next Friday at *beginning* of class

• Info sheets?
• Catalyst Survey
Today

- How to compare two algorithms?
- Analyzing code
- Big-Oh
Comparing Two Algorithms…
Gauging performance

• Uh, why not just run the program and time it
  – Too much *variability*, not reliable or *portable*:
    • Hardware: processor(s), memory, etc.
    • OS, Java version, libraries, drivers
    • Other programs running
    • Implementation dependent
  – Choice of input
    • Testing (inexhaustive) may *miss* worst-case input
    • Timing does not *explain* relative timing among inputs
      (what happens when \( n \) doubles in size)

• Often want to evaluate an *algorithm*, not an implementation
  – Even *before* creating the implementation (“coding it up”)
Comparing algorithms

When is one *algorithm* (not *implementation*) better than another?
  - Various possible answers (clarity, security, …)
  - But a big one is *performance*: for sufficiently large inputs, runs in less time (our focus) or less space

Large inputs (n) because probably any algorithm is “plenty good” for small inputs (if n is 10, probably anything is fast enough)

Answer will be *independent* of CPU speed, programming language, coding tricks, etc.

Answer is general and rigorous, complementary to “coding it up and timing it on some test cases”
  - Can do analysis before coding!

4/05/2013
Analyzing code ("worst case")

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”.)

Consecutive statements  
Sum of time of each statement
Conditionals  
Time of condition plus time of slower branch
Loops  
Num iterations * time for loop body
Function Calls  
Time of function’s body
Recursion  
Solve recurrence equation
Example

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

// 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:

Worst case:
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: 5ish*(arr.length) = \(O(arr.length)\)
Analyzing Recursive Code

• 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 problem size 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)
Example Recursive code: sum array

Recursive:
- Recurrence is some constant amount of work $O(1)$ done $n$ 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);
}
```

Each time `help` is called, it does that $O(1)$ amount of work, and then calls `help` again on a problem one less than previous problem size.

Recurrence Relation: $T(n) = O(1) + T(n-1)$
Solving Recurrence Relations

• Say we have the following recurrence relation:
  \[ T(n) = 3 + T(n-1) \]
  \[ T(1) = 5 \quad \text{← base case} \]

• Now we just need to solve it; that is, reduce it to a closed form.

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

• We expanded it out \( n-1 \) times, so
  \[ T(n) = 3k + T(n-k) \]
  \[ = 3(n-1) + T(1) = 3(n-1) + 5 \]
  \[ = 3n + 2 = O(n) \]

Or When does \( n-k=1? \)
Answer: when \( k=n-1 \)
Binary search

Find an integer in a sorted array

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

```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);
}
```

4/05/2013
Binary search

Best case: 9ish steps = $O(1)$
Worst case: $T(n) = 10ish + T(n/2)$ where $n$ is $hi-lo$
  • $O(\log n)$ where $n$ is `array.length`
  • Solve recurrence equation to know that...

```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?
   - $T(n) = 10 + T(n/2)$  \hspace{1cm} T(1) = 15

2. “Expand” the original relation to find an equivalent general expression in terms of the number 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
Solving Recurrence Relations

1. Determine the recurrence relation. What is the base case?
   - \( T(n) = 10 + T(n/2) \quad T(1) = 15 \)

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

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 \( T(n) = 10 \log_2 n + 15 \) (get to base case and do it)
   - So \( T(n) \) is \( O(\log n) \)
Ignoring constant factors

• So binary search is $O(\log n)$ and linear is $O(n)$
  – But which is faster?
  – Depending on constant factors and size of $n$, in a particular case, linear search could be faster…. 

• Could depend on constant factors
  – How many assignments, additions, etc. for each $n$
  – And could depend on size of $n$

• **But** there exists some $n_0$ such that for all $n > n_0$ binary search wins

• Let’s play with a couple plots to get some intuition…
Example

- Let’s try to “help” linear search
  - Run it on a computer 100x as fast (say 2010 model vs. 1990)
  - Use a new compiler/language that is 3x as fast
  - Be a clever programmer to eliminate half the work
  - So doing each iteration is 600x as fast as in binary search
- Note: 600x still helpful for problems without logarithmic algorithms!
Another example: sum array

Two “obviously” linear algorithms: \( T(n) = O(1) + T(n-1) \)

Iterative:

```java
int sum(int[] arr)
{
    int ans = 0;
    for(int i=0; i<arr.length; ++i)
        ans += arr[i];
    return ans;
}
```

Recursive:

- Recurrence is \( c + c + \ldots + c \) 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);
}
```
What about a binary version of sum?

```java
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 \( T(n) = O(1) + 2T(n/2) \)

- \( 1 + 2 + 4 + 8 + \ldots \) for \( \log n \) times
- \( 2^{(\log n)} - 1 \) which is proportional to \( n \) (by definition of logarithm)

Easier explanation: it adds each number once while doing little else

“Obvious”: You can’t do better than \( O(n) \) – have to read whole array
Parallelism teaser

• But suppose we could do two recursive calls at the same time
  – Like having a friend do half the work for you!

```java
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);
}
```

• If you have as many “friends of friends” as needed, the recurrence is now
  \( T(n) = O(1) + 1 \cdot T(n/2) \)
  – \( O(\log n) \) : same recurrence as for find
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

- Example: can sum all elements of an \( n \)-by-\( m \) matrix in \( O(nm) \)
Asymptotic notation

About to show formal definition, which amounts to saying:
1. Eliminate low-order terms
2. Eliminate coefficients

Examples:
- $4n + 5$
- $0.5n \log n + 2n + 7$
- $n^3 + 2^n + 3n$
- $n \log (10n^2)$
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**

Notes:
• 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)
**Big-Oh relates functions**

We use $O$ on a function $f(n)$ (for example $n^2$) to mean *the set of functions with asymptotic behavior less than or equal to* $f(n)$

So $(3n^2+17)$ **is in** $O(n^2)$
- $3n^2+17$ and $n^2$ have the same asymptotic behavior

Confusingly, we also say/write:
- $(3n^2+17)$ **is** $O(n^2)$
- $(3n^2+17) = O(n^2)$

But we would never say $O(n^2) = (3n^2+17)$
Formally Big-Oh

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

\[
g(n) \leq c f(n)
\]
for all \( n \geq n_0 \)

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

- Example: Let \( g(n) = 3n^2 + 17 \) and \( f(n) = n^2 \)
  - \( c = 5 \) and \( n_0 = 10 \) is more than good enough

This is “less than or equal to”
  - So \( 3n^2 + 17 \) is also \( O(n^5) \) and \( O(2^n) \) etc.
Using the definition of Big-Oh (Example 1)

For $g(n) = 4n$ & $f(n) = n^2$, prove $g(n)$ is in $O(f(n))$

– A valid proof is to find valid $c$ & $n_0$
– When $n=4$, $g(n) = 16$ & $f(n) = 16$; this is the crossing over point
– So we can choose $n_0 = 4$, and $c = 1$

– Note: There are many possible choices:
  ex: $n_0 = 78$, and $c = 42$ works fine

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

$$g(n) \leq c \cdot f(n) \text{ for all } n \geq n_0.$$
Using the definition of Big-Oh (Example 2)

For $g(n) = n^4$ & $f(n) = 2^n$, prove $g(n)$ is in $O(f(n))$

- A valid proof is to find valid $c$ & $n_0$
- One possible answer: $n_0 = 20$, and $c = 1$

The Definition: $g(n)$ is in $O(f(n))$ iff there exist positive constants $c$ and $n_0$ such that $g(n) \leq c \cdot f(n)$ 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:
  
  $g(n) = 7n + 5$
  $f(n) = n$

• These have the same asymptotic behavior (linear), so $g(n)$ is in $O(f(n))$ even though $g(n)$ is always larger
• There is no positive $n_0$ such that $g(n) \leq f(n)$ for all $n \geq n_0$
• The ‘$c$’ in the definition allows for that:
  
  $g(n) \leq c f(n)$  for all $n \geq n_0$
• To prove $g(n)$ is in $O(f(n))$, have $c = 12$, $n_0 = 1$
What you can drop

- Eliminate coefficients because we don’t have units anyway
  - $3n^2$ versus $5n^2$ doesn’t mean anything when we have not specified the cost of constant-time operations (can re-scale)

- Eliminate low-order terms because they have vanishingly small impact as $n$ grows

- Do NOT ignore constants that are not multipliers
  - $n^3$ is not $O(n^2)$
  - $3^n$ is not $O(2^n)$

(This all follows from the formal definition)
Big Oh: Common Categories

From fastest to slowest

$O(1)$ constant (same as $O(k)$ for constant $k$)

$O(\log n)$ logarithmic

$O(n)$ linear

$O(n \log n)$ “$n \log n$”

$O(n^2)$ 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 ($k=1.01$)
More Asymptotic Notation

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

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

• **Tight bound**: $\Theta( f(n) )$ is the set of all functions asymptotically equal to $f(n)$
  
  – Intersection of $O( f(n) )$ and $\Omega( f(n) )$ (use different $c$ values)
Regarding use of terms

A 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
– Since $f(n) = n$ is also $O(n^5)$, it’s tempting to say “this algorithm is exactly $O(n)$”
– Somewhat incomplete; instead say it is $\Theta(n)$
– That means that it is not, for example $O(\log n)$

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 $\omega(\log n)$ but not $\omega(n)$
What we are analyzing

- The most common thing to do is give an $O$ or $\theta$ bound to the worst-case running time of an algorithm

- Example: True statements about binary-search algorithm
  - Common: $\theta(\log n)$ running-time in the worst-case
  - Less common: $\theta(1)$ in the best-case (item is in the middle)
  - Less common: Algorithm is $\Omega(\log \log n)$ in the worst-case (it is not really, really, really fast asymptotically)
  - Less common (but very good to know): the find-in-sorted-array problem is $\Omega(\log n)$ in the worst-case
    - No algorithm can do better (without parallelism)
    - A problem cannot be $O(f(n))$ since you can always find a slower algorithm, but can mean there exists an algorithm
Other things to analyze

• Space instead of time
  – Remember we can often use space to gain time

• Average case
  – Sometimes only if you assume something about the distribution of inputs
    • See CSE312 and STAT391
  – Sometimes uses randomization in the algorithm
    • Will see an example with sorting; also see CSE312
  – Sometimes an amortized guarantee
    • Will discuss in a later lecture
Summary

Analysis can be about:

• The problem or the algorithm (usually algorithm)
• Time or space (usually time)
  – Or power or dollars or …
• Best-, worst-, or average-case (usually worst)
• Upper-, lower-, or tight-bound (usually upper or tight)
Big-Oh Caveats

- Asymptotic complexity (Big-Oh) focuses on behavior for large $n$ and is independent of any computer / coding trick
  - But you can “abuse” it to be misled about trade-offs
  - 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 $n^{1/10}$
- 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
Addendum: Timing vs. Big-Oh?

- 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
  - We 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