CSE 326: Data Structures

Asymptotic Analysis

Eric McCambridge
Spring 2009
Lecture 3 4

Algorithm Analysis

- Correctness:
  - Does the algorithm do what is intended.

- Performance:
  - Speed time complexity
  - Memory space complexity

- Why analyze?
  - To make good design decisions
  - Enable you to look at an algorithm (or code) and identify the bottlenecks, etc.

Administrivia

- Yes, there is class today (and Monday... and Wednesday)
- Project 1 – should be started
- HW1 – out today

Correctness

Correctness of an algorithm is established by proof. Common approaches:

- (Dis)proof by counterexample \( f(n) = 3^n \)
- Proof by contradiction
- Proof by induction
  - Especially useful in recursive algorithms

Proof by Induction

- **Base Case:** The algorithm is correct for a base case or two by inspection.

- **Inductive Hypothesis (n=k):** Assume that the algorithm works correctly for the first k cases.

- **Inductive Step (n=k+1):** Given the hypothesis above, show that the k+1 case will be calculated correctly.

Recursive algorithm for sum

- Write a recursive function to find the sum of the first n integers stored in array v.

```plaintext
sum(integer array v, integer n) returns integer
if n = 0 then
  sum = 0
else
  sum = nth number + sum(v, n-1)
return sum
```

Program Correctness by Induction

- **Basis Step**: \( n = 0 \)
  \[ 0 = 0 \]

- **Inductive Hypothesis \((n=k)\)**:
  \[ \text{sum}(v, k) = \text{sum of } \text{first } k \]

- **Inductive Step \((n=k+1)\)**:
  \[ \text{sum}(v, k+1) = \text{sum}(v, k) + \ell(v_{k+1}) \]
  \[ = v[k+1] + \ell(v_{k+1}) \]
  \[ = v[k+1] + v[k+1] \]
  \[ = v[k+1] \]

Algorithms vs Programs

- Proving correctness of an algorithm is very important
  - A well-designed algorithm is guaranteed to work correctly and its performance can be estimated

- Proving correctness of a program (an implementation) is fraught with weird bugs
  - Abstract Data Types are a way to bridge the gap between mathematical algorithms and programs

Comparing Two Algorithms

**GOAL**: Sort a list of names

"I'll buy a faster CPU"

"I'll use C++ instead of Java – wicked fast!"

"Ooh look, the -O4 flag!"

"Who cares how I do it, I'll add more memory!"

"Can't I just get the data pre-sorted??"

What we want:
- Rough Estimate
- Ignores Details

- Really, independent of details
  - Coding tricks, CPU speed, compiler optimizations, ...
  - These would help any algorithms equally
  - Don't just care about running time – not a good enough measure

How to measure performance?

- `count` # ops
- `memory` used
- `elapsed time`
  - `total`
    - `per operation`
  - `different # of elts`
  - `graphs, charts`

Analysis of Algorithms

- **Efficiency measure**
  - How long the program runs: **time complexity**
  - How much memory it uses: **space complexity**

- **Why analyze at all?**
  - Decide what algorithm to implement before actually doing it
  - Given code, get a sense for where bottlenecks must be, without actually measuring it
Big-O Analysis

- Ignores “details”
- What details?
  - CPU speed
  - Programming language used
  - Amount of memory
  - Compiler
  - Order of input
  - Size of input ... sorts.

Asymptotic Analysis

- Complexity as a function of input size $n$
  $T(n) = 4n + 5$
  $T(n) = 0.5 n \log n - 2n + 7$
  $T(n) = 2^n + n^2 + 3n$

- What happens as $n$ grows?

Why Asymptotic Analysis?

- Most algorithms are fast for small $n$
  - Time difference too small to be noticeable
  - External things dominate (OS, disk I/O, ...)
- BUT $n$ is often large in practice
  - Databases, internet, graphics, ...
- Difference really shows up as $n$ grows!

$2^n$ vs. $n$, only one will.

Analyzing Code

- Basic operations
- Consecutive statements
- Conditionals
- Loops
- Function calls
- Recursive functions

- Constant time
- Sum of times
- Larger branch plus best
- Sum of iterations
- Cost of function body
- Solve recurrence relation

Exercise - Searching

```c
bool ArrayFind(int array[], int n, int key)
// Insert your algorithm here
    - binary search
    - linear search
    - hashing
    - random search
    - false
```

What algorithm would you choose to implement this code snippet?

Linear Search Analysis

```c
bool linearArrayFind(int array[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (array[i] == key) {
            // Found it
            return true;
        }
    }
    return false;
}
```

Best Case: 4
Worst Case: 3n+2
(Not found)
Binary Search Analysis

```c
bool BinArrayFind(int array[], int low, int high, int key) {
    // The subarray is empty
    if (low > high) return false;

    // Search this subarray recursively
    int mid = (high + low) / 2;
    if (key == array[mid]) {
        return true;
    } else if (key < array[mid]) {
        return BinArrayFind(array, low, mid - 1, key);
    } else {
        return BinArrayFind(array, mid + 1, high, key);
    }
}
```

Best case: \( O(1) \)
Worst case: \( O(\log n) \) \( O(\log n + 1) \)

Linear Search vs Binary Search

<table>
<thead>
<tr>
<th></th>
<th>Linear Search</th>
<th>Binary Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best Case</td>
<td>( 4 )</td>
<td>( 4 )</td>
</tr>
<tr>
<td>Worst Case</td>
<td>( 3n + 2 )</td>
<td>( 4 \log n + 1 )</td>
</tr>
</tbody>
</table>

So, which algorithm is better? What tradeoffs can you make?

Solving Recurrence Relations

1. Determine the recurrence relation. What is/are the base case(s)?
   \( T(1) = 1 \), \( T(n) = 4 + \sqrt{n} / 2 \)

2. “Expand” the original relation to find an equivalent general expression in terms of the number of expansions:
   \[
   T(n) = 4 + \left( 4 + T\left( \frac{n}{4} \right) \right) = 4 \cdot 4 + (4 + T\left( \frac{n}{16} \right)) = 4 \cdot 4 \cdot 4 + T\left( \frac{n}{64} \right)
   \]
   \[
   T(n) = n \cdot 4 + T\left( \frac{n}{2^k} \right)
   \]

3. Find a closed-form expression by setting the number of expansions to a value which reduces the problem to a base case
   \[
   4^k n = 1 \rightarrow 4^k = \log_2 n \rightarrow k = \log_2 \log_2 n
   \]

Linear search—empirical analysis

Empirical comparison

Gives additional information
Fast Computer vs. Slow Computer

Fast Computer vs. Smart Programmer (round 2)

Asymptotic Analysis
- Asymptotic analysis looks at the order of the running time of the algorithm
- A valuable tool when the input gets "large"
- Ignores the effects of different machines or different implementations of an algorithm
- Intuitively, to find the asymptotic runtime, throw away the constants and low-order terms
  - Linear search is $T(n) = 3n + 2 \in O(n)$
  - Binary search is $T(n) = 4 \log n + 4 \in O(\log n)$

Properties of Logs
- $\log AB = \log A + \log B$
- Proof: $A = 2^{\log A}, B = 2^{\log B}$
  $AB = 2^{\log A \cdot \log B} = 2^{\log A + \log B}$
  $\therefore \log AB = \log A + \log B$
- Similarly:
  - $\log(A/B) = \log A - \log B$
  - $\log(A^n) = n \log A$
- Any log is equivalent to log-base-2
Properties of Logs

Basic:
- $A^{\log_B A} = B$
- $\log_B A = 1 \cdot \log_B A = \frac{1}{\log_B A} \cdot \log A$

Independent of base:
- $\log(AB) = \log A + \log B$
- $\log(A/B) = \log A - \log B$
- $\log(A^B) = B \log A$
- $\log((A^B)^C) = BC \log A$

Order Notation: Intuition

$f(n) = n^3 + 2n^2$
$g(n) = 100n^3 + 1000$

Although not yet apparent, as $n$ gets "sufficiently large", $f(n)$ will be "greater than or equal to" $g(n)$

Definition of Order Notation

$O(f(n))$: a set or class of functions

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

$g(n) \leq c \cdot f(n)$ for all $n \geq n_0$

Example:

$100n^2 + 1000 \leq 5(n^3 + 2n^2)$ for all $n \geq 19$

So $g(n) \in O(f(n))$

Order Notation: Example

$100n^2 + 1000 \leq 5(n^3 + 2n^2)$ for all $n \geq 19$

So $f(n) \in O(g(n))$
Some Notes on Notation

- Sometimes you'll see \( g(n) = O(f(n)) \)
- This is equivalent to \( g(n) \leq O(f(n)) \)
- What about the reverse? \( O(f(n)) = \{ g(n) \} \)

Big-O: Common Names

- constant: \( O(1) \)
- logarithmic: \( O(\log n) \)
- linear: \( O(n) \)
- log-linear: \( O(n \log n) \)
- quadratic: \( O(n^2) \)
- cubic: \( O(n^3) \)
- polynomial: \( O(n^k) \)
- exponential: \( \exp(c) \)

\[ n! = n \cdot (n-1) \cdot (n-2) \cdots 1 \]

Meet the Family

- \( O(f(n)) \) is the set of all functions asymptotically less than or equal to \( f(n) \)
- \( \omega(f(n)) \) is the set of all functions asymptotically strictly less than \( f(n) \)
- \( \Omega(f(n)) \) is the set of all functions asymptotically greater than or equal to \( f(n) \)
- \( \omega(f(n)) \) is the set of all functions asymptotically strictly greater than \( f(n) \)
- \( \Theta(f(n)) \) is the set of all functions asymptotically equal to \( f(n) \)

Big-Omega et al. Intuitively

<table>
<thead>
<tr>
<th>Asymptotic Notation</th>
<th>Mathematics Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( O )</td>
<td>( \leq )</td>
</tr>
<tr>
<td>( \Omega )</td>
<td>( \geq )</td>
</tr>
<tr>
<td>( \Theta )</td>
<td>( = )</td>
</tr>
<tr>
<td>( o )</td>
<td>( &lt; )</td>
</tr>
<tr>
<td>( \omega )</td>
<td>( &gt; )</td>
</tr>
</tbody>
</table>

Meet the Family, Formally

- \( g(n) \in O(f(n)) \) iff there exist \( c \) and \( n_0 \) such that \( g(n) \leq c f(n) \) for all \( n \geq n_0 \)
- \( g(n) \in \Omega(f(n)) \) iff there exist \( c \) and \( n_0 \) such that \( g(n) \geq c f(n) \) for all \( n \geq n_0 \)

- \( g(n) \in \Theta(f(n)) \) iff there exist \( c \) and \( n_0 \) such that \( g(n) \geq c f(n) \) for all \( n \geq n_0 \)
- \( g(n) \in \omega(f(n)) \) iff there exist \( c \) and \( n_0 \) such that \( g(n) \leq c f(n) \) for all \( n \geq n_0 \)

- \( g(n) \in o(f(n)) \) iff \( g(n) \in O(f(n)) \) and \( g(n) \leq O(f(n)) \)

Pros and Cons of Asymptotic Analysis

- **estimate**
- ignores architecture
- small data
- ignores constants
Perspective: Kinds of Analysis

- Running time may depend on actual data input, not just length of input
- Distinguish
  - Worst Case
    - Your worst enemy is choosing input
  - Amortized
    - Average time over many operations
  - Average Case
    - Assumes some probabilistic distribution of inputs
  - Best Case

Types of Analysis

Two orthogonal axes:

- Bound Flavor
  - Upper bound (O, o)
  - Lower bound (Ω, ω)
  - Asymptotically tight (Θ)

- Analysis Case
  - Worst Case (Adversary)
  - Average Case
  - Best Case
  - Amortized

16n^3 \log_8(10n^2) + 100n^2 = O(n^3 \log n)

- Eliminate low-order terms
- Eliminate constant coefficients