CSE 326: Data Structures

Asymptotic Analysis

Steve Seitz
Winter 2009

Project 1

• Soundblaster! Reverse a song
  – a.k.a., “backmasking”
• Implement a stack to make the “Reverse” program work
  – Implement as array and as linked list
• Read the website
  – Detailed description of assignment
  – Detailed description of how programming projects are graded
• Due by: 11:59pm PDT, Wed. Jan 14
  – Electronic submission

Algorithm Analysis

• Correctness:
  – Does the algorithm do what is intended.

• Performance:
  – Speed \( \text{time complexity} \)
  – Memory \( \text{space complexity} \)

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

Other announcements

• Homework requires you get the textbook (it’s a good book).

• Go to Thursdays sections

• Homework #1 is now assigned.
  – Due at the beginning of class next Friday (Jan 16).
Correctness

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

– (Dis)proof by counterexample
– 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.

```java
sum(int array v, int n) returns int
if n = 0 then
    sum = 0
else
    sum = nth number + sum of first n-1 numbers
return sum
```

Program Correctness by Induction

• **Base Case:**

• **Inductive Hypothesis (n=k):**

• **Inductive Step (n=k+1):**
How to measure performance?

We will focus on analyzing time complexity. First, we have some “rules” to help measure how long it takes to do things:

- **Basic operations**  Constant time
- **Consecutive statements**  Sum of times
- **Conditionals**  Test, plus larger branch cost
- **Loops**  Sum of iterations
- **Function calls**  Cost of function body
- **Recursive functions**  Solve recurrence relation...

Second, we will be interested in **best** and **worst** case performance.

Analyzing Performance

Exercise - Searching

```c
bool ArrayContains(int array[], int n, int key){
    // Insert your algorithm here
}
```

Complexity cases

We’ll start by focusing on two cases.

- **Worst-case complexity**: max # steps algorithm takes on “most challenging” input of size N
- **Best-case complexity**: min # steps algorithm takes on “easiest” input of size N

What algorithm would you choose to implement this code snippet?
Exercise - Searching

bool ArrayContains(int array[], int n, int key) {
    // Insert your algorithm here

What algorithm would you choose to implement this code snippet?

Linear Search Analysis

bool LinearArrayContains(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:

Worst Case:

Binary Search Analysis

bool BinArrayContains(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:

Worst case:

Solving Recurrence Relations

1. Determine the recurrence relation and base case(s).

2. “Expand” the original relation to find an equivalent expression in terms of the number of expansions (k).

3. Find a closed-form expression by setting k to a value which reduces the problem to a base case.
## 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>5 at [middle]</td>
</tr>
<tr>
<td>Worst Case</td>
<td>3n+3</td>
<td>7 ⌊ log n ⌋ + 9</td>
</tr>
</tbody>
</table>

### Linear search—empirical analysis

Each search produces a dot in above graph. Blue = less frequently occurring, Red = more frequent

### Binary search—empirical analysis

Each search produces a dot in above graph. Blue = less frequently occurring, Red = more frequent

### Empirical comparison

Gives additional information
Fast Computer vs. Slow Computer

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 same algorithm
- Comparing worst case search examples:
  \[ T_{\text{worst}}^{LS}(n) = 3n + 3 \quad \text{vs.} \quad T_{\text{worst}}^{BS}(n) = 7\lceil \log_2 n \rceil + 9 \]
Asymptotic Analysis

- Intuitively, to find the asymptotic runtime, throw away the constants and low-order terms
  - Linear search is $T_{\text{worst}}^{LS}(n) = 3n + 3 \in O(n)$
  - Binary search is $T_{\text{worst}}^{BS}(n) = 7\lceil\log_2 n\rceil + 9 \in O(\log n)$

Remember: the “fastest” algorithm has the slowest growing function for its runtime

Eliminate low order terms
- $-4n + 5 \Rightarrow -0.5n \log n + 2n + 7 \Rightarrow -n^3 + 3 \cdot 2^n + 8n \Rightarrow$

Eliminate coefficients
- $-4n \Rightarrow -0.5n \log n \Rightarrow -3 \cdot 2^n \Rightarrow$

Properties of Logs

Basic:
- $A^{\log_AB} = B$
- $\log_A A = 1$

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

Properties of Logs

$$\log_A n = \left(\frac{1}{\log_B A}\right) \log_B n$$
Another example

- Eliminate low-order terms
- Eliminate constant coefficients

$$16n^3 \log_8(10n^2) + 100n^2$$

Comparing functions

- $f(n)$ is an **upper bound** for $h(n)$ if $h(n) \leq f(n)$ for all $n$

This is too strict – we mostly care about large $n$

Still too strict if we want to ignore scale factors

Definition of Order Notation

- **Upper bound**: $h(n) \in O(f(n))$ Big-O “order”
  Exist positive constants $c$ and $n_0$ such that $h(n) \leq c f(n)$ for all $n \geq n_0$

- **Lower bound**: $h(n) \in \Omega(g(n))$ Omega
  Exist positive constants $c$ and $n_0$ such that $h(n) \geq c g(n)$ for all $n \geq n_0$

- **Tight bound**: $h(n) \in \theta(f(n))$ Theta
  When both hold:
  $h(n) \in O(f(n))$
  $h(n) \in \Omega(f(n))$

$O(f(n))$ defines a class (set) of functions

Order Notation: Intuition

$a(n) = n^3 + 2n^2$

$b(n) = 100n^2 + 1000$

Although not yet apparent, as $n$ gets “sufficiently large”, $a(n)$ will be “greater than or equal to” $b(n)$
Example

h(n) \in O(f(n)) \quad \text{iff there exist positive constants } c \text{ and } n_0 \text{ such that:}

h(n) \leq c f(n) \text{ for all } n \geq n_0

Example:

100n^2 + 1000 \leq \frac{1}{2} (n^3 + 2n^2) \text{ for all } n \geq 198

So \( b(n) \in O(a(n)) \)

Order Notation: Example

100n^2 + 1000 \leq \frac{1}{2} (n^3 + 2n^2) \text{ for all } n \geq 198

So \( b(n) \in O(a(n)) \)

Some Notes on Notation

Worst Case Binary Search

Sometimes you’ll see (e.g., in Weiss)

\( h(n) = O(f(n)) \)

or

\( h(n) \text{ is } O(f(n)) \)

These are equivalent to

\( h(n) \in O(f(n)) \)
Big-O: Common Names

- constant: \( O(1) \)
- logarithmic: \( O(\log n) \)  
  \((\log n, \log n^2 \in O(\log n))\)
- linear: \( O(n) \)
- log-linear: \( O(n \log n) \)
- quadratic: \( O(n^2) \)
- cubic: \( O(n^3) \)
- polynomial: \( O(n^k) \)  
  \((k \text{ is a constant})\)
- exponential: \( O(c^n) \)  
  \((c \text{ is a constant} > 1)\)

Meet the Family

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

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

- \( \theta(f(n)) \) is the set of all functions asymptotically equal to \( f(n) \)

Meet the Family, Formally

- \( h(n) \in O(f(n)) \) iff
  \( \exists c>0, n_0>0 \) such that \( h(n) \leq c f(n) \) for all \( n \geq n_0 \)

- \( h(n) \in o(f(n)) \) iff
  \( \exists n_0>0 \) such that \( h(n) < c f(n) \) for all \( c>0, n \geq n_0 \)
  - This is equivalent to: \( \lim_{n \to \infty} h(n)/f(n) = 0 \)

- \( h(n) \in \Omega(g(n)) \) iff
  \( \exists c>0, n_0>0 \) such that \( h(n) \geq c g(n) \) for all \( n \geq n_0 \)

- \( h(n) \in \omega(g(n)) \) iff
  \( \exists n_0>0 \) such that \( h(n) > c g(n) \) for all \( c>0, n \geq n_0 \)
  - This is equivalent to: \( \lim_{n \to \infty} h(n)/g(n) = \infty \)

- \( h(n) \in \theta(f(n)) \) iff
  \( h(n) \in O(f(n)) \) and \( h(n) \in \Omega(f(n)) \)
  - This is equivalent to: \( \lim_{n \to \infty} h(n)/f(n) = c \neq 0 \)

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>( \vartheta )</td>
<td>( = )</td>
</tr>
<tr>
<td>( o )</td>
<td>( &lt; )</td>
</tr>
<tr>
<td>( \omega )</td>
<td>( &gt; )</td>
</tr>
</tbody>
</table>

Mathematics Relation
Complexity cases (revisited)

Problem size \( N \)
- **Worst-case complexity**: \( \text{max} \) # steps algorithm takes on “most challenging” input of size \( N \)
- **Best-case complexity**: \( \text{min} \) # steps algorithm takes on “easiest” input of size \( N \)
- **Average-case complexity**: \( \text{avg} \) # steps algorithm takes on random inputs of size \( N \)
- **Amortized complexity**: \( \text{max} \) total # steps algorithm takes on \( M \) “most challenging” consecutive inputs of size \( N \), divided by \( M \) (i.e., divide the max total by \( M \)).

Bounds vs. Cases

Two orthogonal axes:
- **Bound Flavor**
  - Upper bound (\( O, o \))
  - Lower bound (\( \Omega, \omega \))
  - Asymptotically tight (\( \theta \))
- **Analysis Case**
  - Worst Case (Adversary), \( T_{\text{worst}}(n) \)
  - Average Case, \( T_{\text{avg}}(n) \)
  - Best Case, \( T_{\text{best}}(n) \)
  - Amortized, \( T_{\text{amort}}(n) \)

One can estimate the bounds for any given case.

Pros and Cons of Asymptotic Analysis