CSE 326: Data Structures

Asymptotic Analysis

Hal Perkins
Spring 2007
Lectures 2 & 3

Today’s Outline

- Admin: Project 1
- Asymptotic analysis

Office Hours, etc.

The plan so far…
Hal Perkins  MW 3:40-4:30  CSE 548
(except today)
Andy Sun  Tue 12:30-1:30  CSE 002 lab
Marius Nita  Thur 2:30-3:20  CSE 3rd floor breakout
Or by appointment.
(Comments? Conflicts?)

TODO: Important!
1. Subscribe to mailing list if you haven’t
2. Hand in info sheet

Project 1 – Sound Blaster!

Play your favorite song in reverse!

Aim:
1. Implement stack ADT two different ways (array, linked list)
2. Use to reverse a sound file

Due: Wed, April 4
    Electronic: at midnight, April 4
    Hardecopy: in sections Thursday
Comparing Two Algorithms

What we want

- Rough Estimate
- Ignores Details

- Characterize and compare algorithms independent of implementation details
  - (coding tricks, machine speed, compiler optimizations)

Analysis of Algorithms

- Efficiency measure
  - how long the program runs: time complexity
  - how much memory it uses: space complexity
    • For today, we’ll focus on time complexity only

- Analysis is in terms of the problem size
  - Size depends on problem being solved
  - Typical: size of data structure, magnitude of some numeric parameter, …

Asymptotic Analysis

- Complexity as a function of input size $n$
  \[
  T(n) = \begin{cases} 
  4n + 5 & \text{if } n \text{ is even} \\
  0.5n \log n - 2n + 7 & \text{if } n \text{ is odd} \\
  2^n + n^3 + 3n & \text{if } n \text{ is large}
  \end{cases}
  \]

- 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, …

- Time difference really shows up as \( n \) grows!

### Analyzing Code

<table>
<thead>
<tr>
<th>Basic Java operations</th>
<th>Constant time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consecutive statements</td>
<td>Sum of times</td>
</tr>
<tr>
<td>Conditionals</td>
<td>Larger branch plus test</td>
</tr>
<tr>
<td>Loops</td>
<td>Sum of iterations</td>
</tr>
<tr>
<td>Function calls</td>
<td>Cost of function body</td>
</tr>
<tr>
<td>Recursive functions</td>
<td>Solve recurrence relation</td>
</tr>
</tbody>
</table>

*Let’s try it!*

### Algorithm Analysis Examples

- Consider the following program segment:
  ```java
  x:= 0;
  for i = 1 to N do
    for j = 1 to i do
      x := x + 1;
  ```

- What is the value of \( x \) at the end?
  (equivalent: how many times is \( x := x+1 \) executed as a function of \( N \)?)

### Analyzing the Loop

- Total number of times \( x \) is incremented is executed:
  \[
  1+2+3+\ldots+\sum_{i=1}^{N} = \frac{N(N+1)}{2}
  \]

- Congratulations - You’ve just analyzed your first program!
  - Running time of the program is proportional to \( N(N+1)/2 \) for all \( N \)
Another Example: Nested Loops

for i = 1 to n do
    for j = 1 to n do
        sum = sum + 1
    end for
end for

And Another: Nested Loops

for i = 1 to n do
    for j = 1 to n do
        if (cond) {
            do_stuff(sum)
        } else {
            for k = 1 to n*n
                sum += 1
        }
    end for
end for

Exercise - Searching

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

    What algorithm would you choose to implement this code snippet?

Linear Search Analysis

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;
}
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:

Worst case:

Solving Recurrence Relations

1. Determine the recurrence relation. What is the base case(s)?
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.

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></td>
<td></td>
</tr>
<tr>
<td>Worst Case</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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 the same 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 $T(n) = 4 \log_2 n + 4 \in O(\log n)$

Remember: the fastest algorithm has the slowest growing function for its runtime
Although not yet apparent, as $n$ gets “sufficiently large”, $f(n)$ will be “greater than or equal to” $g(n)$.

**Order Notation: Intuition**

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

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

**Order Notation**

- **Upper bound:** $T(n) = O(f(n))$  
  Big-O

  Exist constants $c$ and $n'$ such that $T(n) \leq c f(n)$ for all $n \geq n'$

- **Lower bound:** $T(n) = \Omega(g(n))$  
  Omega

  Exist constants $c$ and $n'$ such that $T(n) \geq c g(n)$ for all $n \geq n'$

- **Tight bound:** $T(n) = \theta(f(n))$  
  Theta

  When both hold:
  
  $T(n) = O(f(n))$

  $T(n) = \Omega(f(n))$

**$O(f(n))$ Definition**

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

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

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

Example:

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

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

Sometimes, you’ll see the notation $g(n) = O(f(n))$. This is equivalent to $g(n) \in O(f(n))$.

Remember: notation $O(f(n)) = g(n)$ is meaningless!
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: \( O(n^k) \) (\( k \) is a constant)
- exponential: \( O(c^n) \) (\( c \) 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 \( f(n) \)
- \( \Omega( f(n) ) \) 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 \( f(n) \)

Meet the Family, Formally

- \( g(n) \in O( f(n) ) \) iff
  There exist \( c \) and \( n_0 \) such that \( g(n) \leq c \cdot f(n) \) for all \( n \geq n_0 \)
  - \( g(n) \in o( f(n) ) \) iff
  There exists a \( n_0 \) such that \( g(n) < c \cdot f(n) \) for all \( c \) and \( n \geq n_0 \)
- \( g(n) \in \Omega( f(n) ) \) iff
  Equivalent to: \( \lim_{n \to \infty} g(n)/f(n) = 0 \)
  There exist \( c \) and \( n_0 \) such that \( g(n) \geq c \cdot f(n) \) for all \( n \geq n_0 \)
  - \( g(n) \in \omega( f(n) ) \) iff
  There exists a \( n_0 \) such that \( g(n) > c \cdot f(n) \) for all \( c \) and \( n \geq n_0 \)
- \( g(n) \in \Theta( f(n) ) \) iff
  Equivalent to: \( \lim_{n \to \infty} g(n)/f(n) = \infty \)
  \( g(n) \in O( f(n) ) \) and \( g(n) \in \Omega( f(n) ) \)
Big-Omega et al. Intuitively

<table>
<thead>
<tr>
<th>Asymptotic Notation</th>
<th>Mathematics Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>≤</td>
</tr>
<tr>
<td>Ω</td>
<td>≥</td>
</tr>
<tr>
<td>θ</td>
<td>=</td>
</tr>
<tr>
<td>o</td>
<td>&lt;</td>
</tr>
<tr>
<td>ω</td>
<td>&gt;</td>
</tr>
</tbody>
</table>

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
  - best case
  - average case
    - assumes some probabilistic distribution of inputs
  - amortized
    - average time over many operations

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”

Pros and Cons of Asymptotic Analysis