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 (except today)	CSE 548
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
Hardcopy: 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

  $T(n) = 4n + 5$

  $T(n) = 0.5\, n \log n - 2n + 7$

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

- *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, …
- Time difference really shows up as *n* grows!

Analyzing Code

Basic Java operations	Constant time
Consecutive statements	Sum of times
Conditionals	Larger branch plus test
Loops	Sum of iterations
Function calls	Cost of function body
Recursive functions	Solve recurrence relation

Let's try it!

Algorithm Analysis Examples

- Consider the following program segment:

```
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 =

$$1+2+3+\ldots=\sum_{i=1}^{N} i=\frac{N(N+1)}{2}$$

- 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

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


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
```


Exercise - Searching

2	3	5	16	37	50	73	75	126

```
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;
}
```

Best Case:

Worst Case:

Binary Search Analysis

```
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

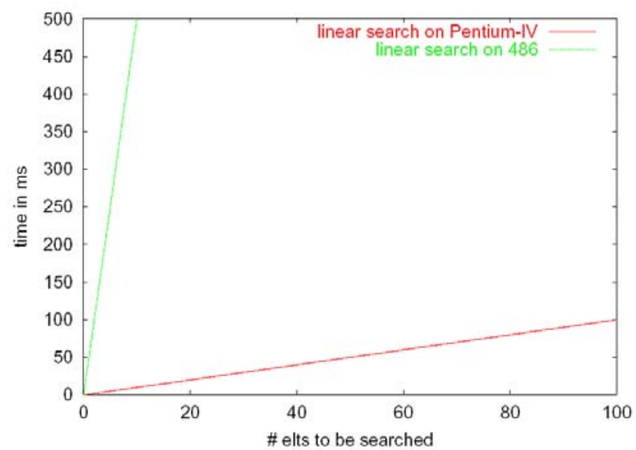
1. Determine the recurrence relation. What is the base case(s)?
2. "Expand" the original relation to find an equivalent general expression *in terms of the number of expansions*.
3. Find a closed-form expression by setting *the number of expansions* to a value which reduces the problem to a base case

Linear Search vs Binary Search

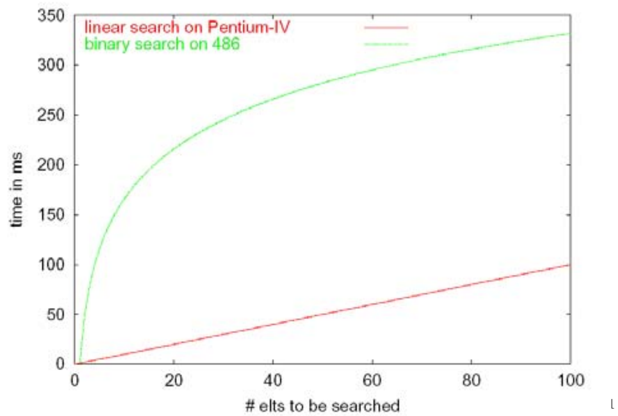
	Linear Search	Binary Search
Best Case		
Worst Case		

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

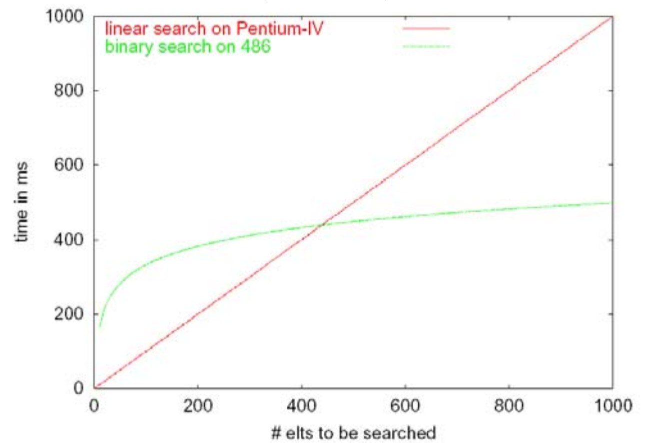
Fast Computer vs. Slow Computer



Fast Computer vs. Smart Programmer (round 1)



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 the same algorithm
- Intuitively, to find the asymptotic runtime, throw away the constants and low-order terms
  - Linear search is $T(n) = 3n + 2 \in \mathbf{O}(\boldsymbol{n})$
  - Binary search is $T(n) = 4 \log_2 n + 4 \in \mathbf{O}(\mathbf{log}\ \boldsymbol{n})$

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

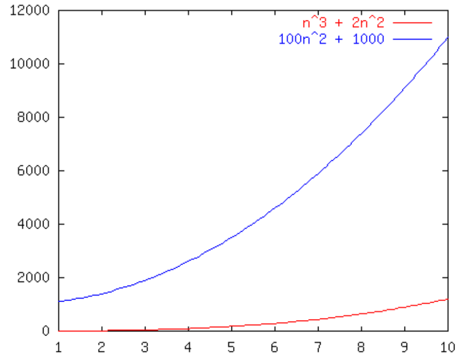
Asymptotic Analysis

- Eliminate low order terms
  - $4n + 5 \Rightarrow$
  - $0.5\ n \log n + 2n + 7 \Rightarrow$
  - $n^3 + 2^n + 3n \Rightarrow$
- Eliminate coefficients
  - $4n \Rightarrow$
  - $0.5\ n \log n \Rightarrow$
  - $n \log n^2$ =>

Order Notation: Intuition

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

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



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

Order Notation

- Upper bound: $T(n) = O(f(n))$ Big-O
  Exist constants c and n' such that
  $T(n) \le c\, f(n)$ for all $n \ge n'$
- Lower bound: $T(n) = \Omega(g(n))$ Omega
  Exist constants c and n' such that
  $T(n) \ge c\, g(n)$ for all $n \ge n'$
- 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*)) : <u>a set or class of functions</u>

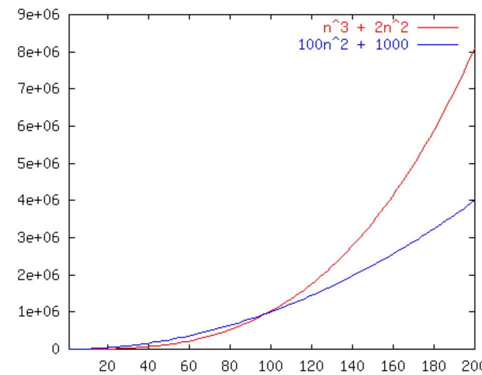
$g(n) \in O(f(n))$ iff there exist consts c and n_0 such that:

$g(n) \le c\, f(n)$ for all $n \ge n_0$

Example:
$100n^2 + 1000 \le 5\,(n^3 + 2n^2)$ for all $n \ge 19$
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!

Order Notation: Example



$100n^2 + 1000 \le 5\,(n^3 + 2n^2)$ for all $n \ge 19$
So $f(n) \in O(g(n))$

Big-O: Common Names

- constant: $O(1)$
- logarithmic: $O(\log n)$ ($\log_k n$, $\log n^2 \in O(\log n)$)
- 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)

Know Your Complexity Classes!

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)
- Ω(f(n)) is the set of all functions asymptotically greater than or equal to f(n)
  - ω(f(n)) is the set of all functions asymptotically strictly greater than f(n)
- θ(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 $g(n) \leq c\, f(n)$ for all $n \geq n_0$
  - $g(n) \in o(f(n))$ iff
    There exists a n_0 such that $g(n) < c\, f(n)$ 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 $g(n) \geq c\, f(n)$ for all $n \geq n_0$
  - $g(n) \in \omega(f(n))$ iff
    There exists a n_0 such that $g(n) > c\, f(n)$ 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

Asymptotic Notation	Mathematics Relation
O	$\le$
Ω	$\ge$
θ	$=$
o	$<$
ω	$>$

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