CSE 326: Data Structures

Asymptotic Analysis

Brian Curless
Spring 2008

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: Midnight (11:59 +ε PM PDT, ε<0:01), April 9
  - Electronic submission

Other announcements

- Both sections are now in EE 025.
- Homework requires you get the textbook (it's a good book).
- Laura rocks.
- Homework #1 is now assigned.
  - Due at the beginning of class next Friday (April 11).

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.

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**.

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


Program Correctness by Induction

- **Base Case:**
  `sum(v,0) = 0.` ✓
- **Inductive Hypothesis (n=k):**
  Assume **`sum(v,k)`** correctly returns sum of first k elements of v, i.e. **`v[0]+v[1]+…+v[k-1]`**
- **Inductive Step (n=k+1):**
  `sum(v,k+1)` returns
  `v[k]+sum(v,k)= (by inductive hyp.)`
  `v[k]+(v[0]+v[1]+…+v[k-1])=`
  `v[0]+v[1]+…+v[k-1]+v[k]` ✓

Analyzing 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.

Complexity cases

We'll start by focusing on two cases.

Problem size **N**

- **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**

Exercise - Searching

2	3	5	16	37	50	73	75

```
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

2	3	5	16	37	50	73	75

```
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/are 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

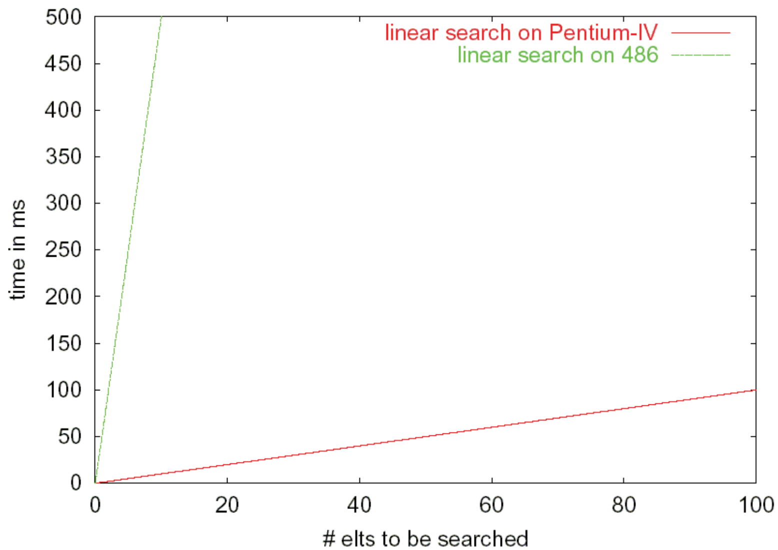
Linear Search vs Binary Search

	Linear Search	Binary Search
Best Case		
Worst Case		

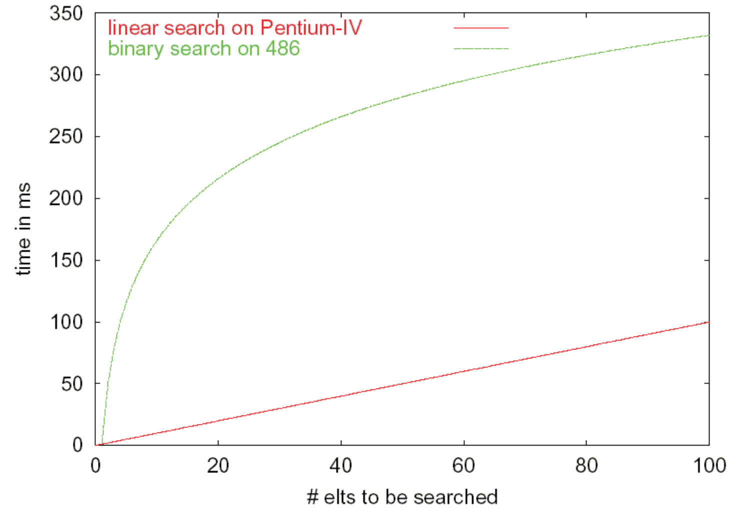
Linear Search vs Binary Search

	Linear Search	Binary Search
Best Case		
Worst Case		

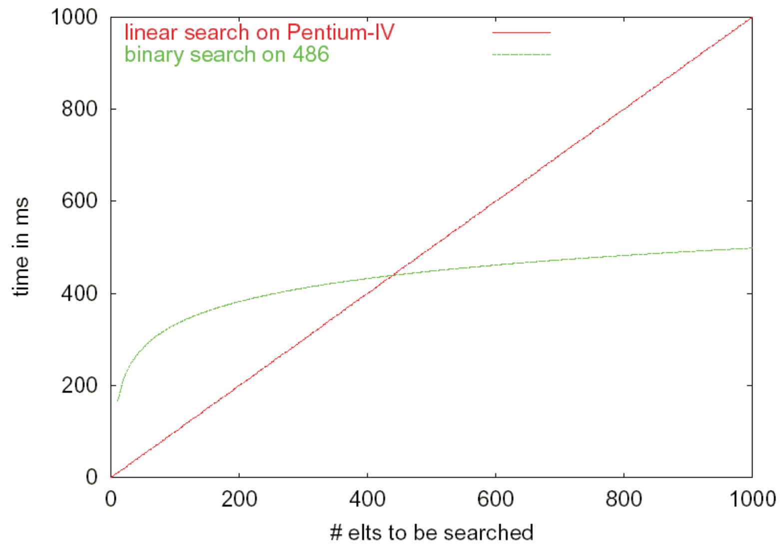
Fast Computer vs. Slow Computer



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 same algorithm
- Comparing worst case search examples:

$$T_{\text{worst}}^{\text{LS}}(n) = 3n + 3 \quad \text{vs.} \quad T_{\text{worst}}^{\text{BS}}(n) = 5\lfloor \log_2 n \rfloor + 7$$

Asymptotic Analysis

- Intuitively, to find the asymptotic runtime, throw away the constants and low-order terms
  - Linear search is $T_{\text{worst}}^{\text{LS}}(n) = 3n + 3 \in O(n)$
  - Binary search is $T_{\text{worst}}^{\text{BS}}(n) = 5\lfloor \log_2 n \rfloor + 7 \in O(\log n)$

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

Asymptotic Analysis

Eliminate low order terms

- $4n + 5 \Rightarrow$
- $0.5\ n \log n + 2n + 7 \Rightarrow$
- $n^3 + 3\ 2^n + 8n \Rightarrow$

Eliminate coefficients

- $4n \Rightarrow$
- $0.5\ n \log n \Rightarrow$
- $3\ 2^n$ =>

Properties of Logs

Basic:

- $A^{\log_A(B)} = B$
- $\log_A(A) =$

Independent of base:

- $\log(AB) =$
- $\log(A/B) =$
- $\log(A^B) =$
- $\log((A^B)^C) =$

Properties of Logs

$\log_A(B)$ vs. $\log_C(B)$?

Another example

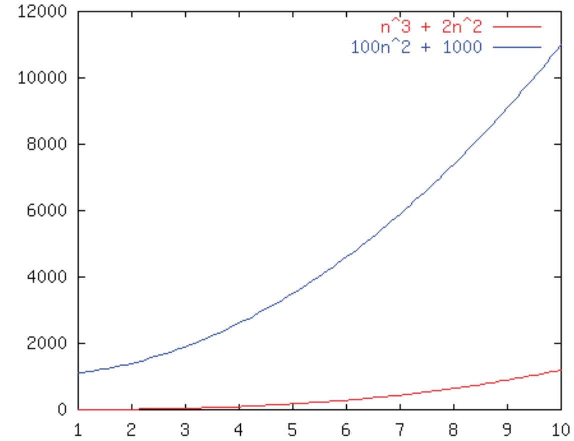
- Eliminate low-order terms
- Eliminate constant coefficients

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

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" $b(n)$

Definition of Order Notation

- Upper bound: $h(n) \in O(f(n))$ Big-O
  Exist positive constants c and n_0 such that
  $h(n) \le c\, f(n)$ for all $n \ge n_0$
- Lower bound: $h(n) \in \Omega(g(n))$ Omega
  Exist positive constants c and n_0 such that
  $h(n) \ge c\, g(n)$ for all $n \ge n_0$
- Tight bound: $h(n) \in \theta(f(n))$ Theta
  When both hold:
  $h(n) \in O(f(n))$
  $h(n) \in \Omega(f(n))$

Definition of Order Notation

O(*f*(*n*)) : <u>a set or class of functions</u>

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

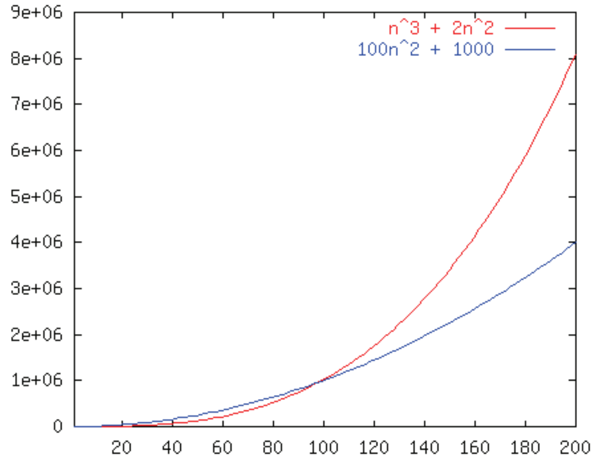
$h(n) \le c\, f(n)$ for all $n \ge n_0$

Example:

$100n^2 + 1000 \le 1/2\ (n^3 + 2n^2)$ for all $n \ge 198$

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

Order Notation: Example



$100n^2 + 1000 \le 1/2\ (n^3 + 2n^2)$ for all $n \ge 198$

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

Order Notation: Worst Case Binary Search

Some Notes on Notation

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

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($g(n)$) is the set of all functions asymptotically greater than or equal to $g(n)$
  - ω($g(n)$) is the set of all functions asymptotically strictly greater than $g(n)$
- θ($f(n)$) is the set of all functions asymptotically equal to $f(n)$

Meet the Family, Formally

- $h(n) \in O(f(n))$ iff
  There exist $c>0$ and $n_0>0$ such that $h(n) \leq c f(n)$ for all $n \geq n_0$
- $h(n) \in o(f(n))$ iff
  There exists an $n_0>0$ such that $h(n) < c f(n)$ for all $c>0$ and $n \geq n_0$
  - This is equivalent to: $\lim_{n\to\infty} h(n)/f(n) = 0$
- $h(n) \in \Omega(g(n))$ iff
  There exist $c>0$ and $n_0>0$ such that $h(n) \geq c\, g(n)$ for all $n \geq n_0$
- $h(n) \in \omega(g(n))$ iff
  There exists an $n_0>0$ such that $h(n) > c\, g(n)$ for all $c>0$ and $n \geq n_0$
  - 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

Asymptotic Notation	Mathematics Relation
O	$\leq$
Ω	$\geq$
θ	$=$
o	$<$
ω	$>$

Complexity cases (revisited)

Problem size **N**

- **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**
- **Average-case complexity**: **avg** # steps algorithm takes on *random* inputs of size **N**
- **Amortized complexity**: **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 (Ω, ω)
  - Asymptotically tight (θ)
- Analysis Case
  - Worst Case (Adversary), $T_{worst}(n)$
  - Average Case, $T_{avg}(n)$
  - Best Case, $T_{best}(n)$
  - Amortized, $T_{amort}(n)$

One can estimate the bounds for any given case.

Bounds vs. Cases