CSE 326: Data Structures

Topic 2: Asymptotic Analysis

Ashish Sabharwal
Autumn, 2003

Today's Outline

- Admin: Office hours, survey results, etc.
- Project 1 – Sound Blaster!
- **Asymptotic analysis**
- A little bit of math (review text for more)

2

Office Hours, etc.

Ashish	Mon, Thu	10:30-11:20	Allen 214
Ethan	Tue	11:00-12:00	Allen 216
Albert	Fri	12:30-1:20	Allen 002

Or by appointment.

<u>TODO</u> : *Important!*

1. Subscribe to mailing lists if you haven't
2. Mark errata in your copy of textbook

3

Survey Results: Where do you stand?

- Java: 75% used in 143
  60% have other experience
- Unix: 70% know the basics
- Big-O: 75% have seen the notation in basic form
- Solving recurrences : 65% know basics

- Data structures: linked lists, binary search tree
  25% have seen hash tables
- Sorting : 75%

4

Quick Review

What are the three things that define the stack ADT?

1.
2.
3.

5

Project 1 – Sound Blaster!

Play your favorite song in reverse!

<u>Aim</u>:

1. Get familiar with UNIX
2. Implement `DoubleStack` class
   (base code provided)
3. Write program to reverse a sound file

<u>Due</u>: Wed, Oct 8, 11:00 pm
(hardcopy in Section on Oct 9)

6

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
- *Why analyze at all?*
  - Confidence: algorithm will work well in practice
    : gives you boss a reason to pay you right away!
  - Insight   : alternative, better algorithms

7

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?*

8

Why do we care?

- 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!

9

Analysis: Simplifying assumptions

- Ideal single-processor machine
  (serialized operations)
- "Standard" instruction set
  (load, add, store, etc.)
- All operations take 1 time unit
  (including each Java or pseudocode statement)

10

Ashish Takes a Break

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?

11

ATaB: 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

Analyze your code!

12

ATaB: 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:

13

ATaB: Binary Search Analysis

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

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

- Best case:
- Worst case:

14

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

15

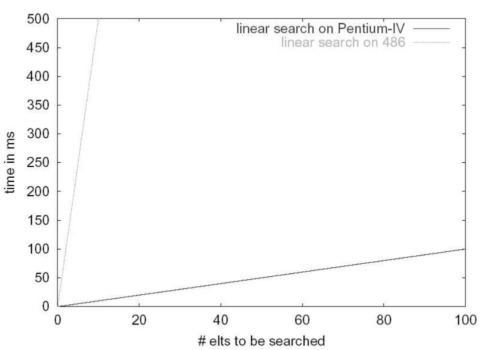
Linear Search vs Binary Search

	Linear Search	Binary Search
Best Case		
Worst Case		

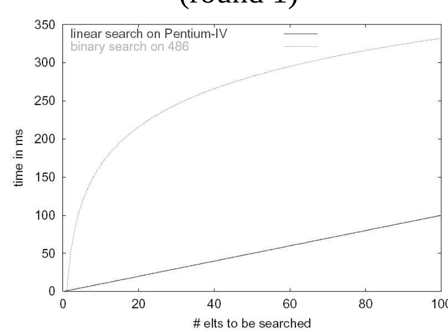
So … which algorithm is better?
What tradeoffs can you make?

16

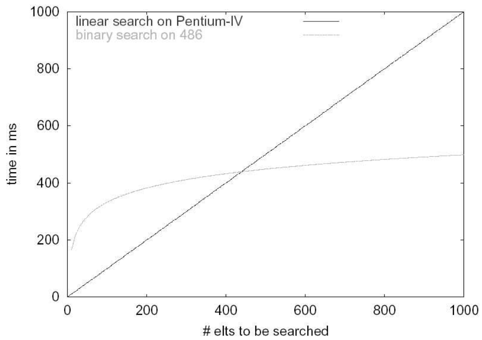
Fast Computer vs. Slow Computer



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 $T(n) = 2n + 1 \in \mathbf{O(n)}$
  - Binary search is $T(n) = 4 \log_2 n + 2 \in \mathbf{O(\log n)}$

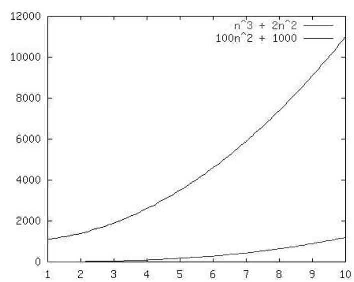
Remember: the fastest algorithm has the slowest growing function for its runtime

20

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" $g(n)$

21

Order Notation: Definition

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

$g(n) \in O(f(n))$ iff
There exist consts c and n_0 such that $g(n) \le c\, f(n)$ for all $n \ge n_0$

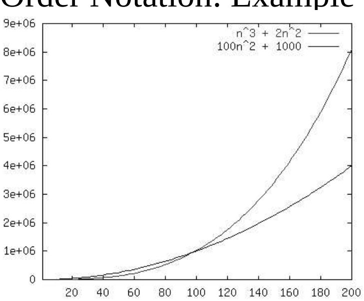
Example:
$100n^2 + 1000 \le 5(n^3 + 2n^2)$ for all $n \ge 19$
So $g(n) \in O(f(n))$

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!

22

Order Notation: Example



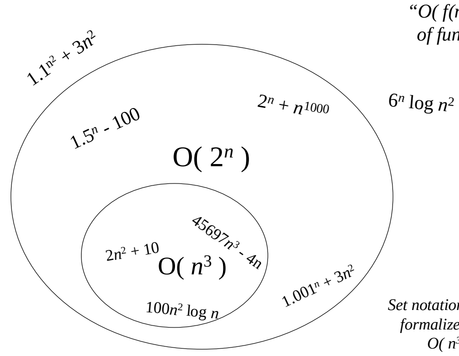
$100n^2 + 1000 \le 5(n^3 + 2n^2)$ for all $n \ge 19$

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

23

Set Notation

"O(f(n)) is a set of functions"



Set notation allows us to formalize our intuition $O(n^3)$ vs. $O(2^n)$

Big-O: Common Names

– constant:	O(1)	
– logarithmic:	O(log n)	($\log_k n$, $\log n^2 \in$ O(log n))
– poly-log:	O($\log^k$ n)	
– linear:	O(n)	
– log-linear:	O(n log n)	
– superlinear:	O(n^{1+c})	(c is a constant > 0)
– 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(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 $n \geq n_0$
    Equivalent to: $\lim_{n\to\infty} g(n)/f(n) = 0$
- $g(n) \in \Omega(f(n))$ iff
  There exist c and n_0 such that $g(n) \geq c\, f(n)$ for all $n \geq n_0$
  - $g(n) \in \omega(f(n))$ iff
    There exists a n_0 such that $g(n) > c\, f(n)$ for all c and $n \geq n_0$
    Equivalent to: $\lim_{n\to\infty} g(n)/f(n) = \infty$
- $g(n) \in \theta(f(n))$ iff
  $g(n) \in O(f(n))$ and $g(n) \in \Omega(f(n))$

Big-Omega et al. Intuitively

Asymptotic Notation	Mathematics Relation
O	≤
Ω	≥
θ	=
o	<
ω	>

True or False?

$10{,}000\, n^2 + 25n \in \theta(n^2)$	
$0.00000001\, n^2 \in \theta(n^2)$	
$n^3 + 4n \in \omega(n^3)$	
$n \log n \in O(2^n)$	
$n \log n \in \Omega(n^2)$	
$n^3 + 4 \in o(n^4)$	

Types of Analysis

Two <u>orthogonal</u> axes:

- bound flavor
  - upper bound (O, o)
  - lower bound (Ω, ω)
  - asymptotically tight (θ)
- analysis case
  - worst case (adversary)
  - average case
  - best case
  - "amortized"

ATaB: Pros and Cons of Asymptotic Analysis

31

Math background: Proof by...

- Counterexample
  - show an example which does not fit with the theorem
  - QED (the theorem is *disproven*)
- Contradiction
  - assume the opposite of the theorem
  - derive a contradiction
  - QED (the theorem is proven)
- Induction
  - prove for a base case (e.g., n = 1)
  - state hypothesis: assume true for a generic value (n = k)
  - inductive step: prove for the next value (n = k + 1)
  - QED (the theorem is proven)

32

Inductive Proof of Correctness

```
int sum(int v[], int n){
  if (n==0) return 0;
  else return v[n-1] + sum(v,n-1);
}
```

Theorem: sum(*v*,*n*) correctly returns sum of 1st *n* elements of array *v* for any *n*

Base case: Program is correct for *n*=0; returns 0. ➨

Inductive hypothesis (*n*=*k*): Assume sum(*v*,*k*) returns sum of first *k* elements of *v*.

Inductive step (*n*=*k*+1): sum(*v*,*k*+1) returns *v*[*k*]+sum(*v*,*k*), which is the same of the first *k*+1 elements of *v*. ➨

33

Inductive Proof of Complexity: Binary Search, Worst Case

If you <u>know</u> the closed form solution, you can validate it by ordinary induction

$T(1) = b + c\log 1 = b$ base case

Assume $\boxed{T(n) = b + c\log n}$ hypothesis

$T(2n) = T(n) + c$ definition of T(n)

$T(2n) = (b + c\log n) + c$ by induction hypothesis

$T(2n) = b + c((\log n) + 1)$

$T(2n) = b + c((\log n) + (\log 2))$

$T(2n) = b + c\log(2n)$ Q.E.D.

Thus: $T(n) = \theta(\log n)$

34

Asymptotic Analysis Summary

- Determine what characterizes a problem's size
- Express how much resources (time, memory, etc.) an algorithm requires as a function of input size using O( ), Ω( ), θ( ) [upper, lower, tight bounds]
  - worst case
  - best case
  - average case
  - common case
  - overall

35

To Do

- Get working on **Project 1**
  - Due Wed, Oct 8 at 11:00 PM sharp!
  - Bring questions to section tomorrow
- Sign up for 326 mailing list(s)
- Mark errata in your textbook
- Continue reading sections 1.1-1.3, 2 and 3
  - Also start/skim on next sections:
    4.1 introduction to trees
    6.1-6.4 priority queues and binary heaps

36