CSE 326: Data Structures

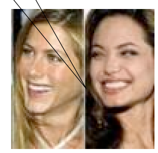
Asymptotic Analysis

Neva Cherniavsky
Summer 2006

Comparing Two Algorithms

Angelina: adds more memory
Jennifer: discovers optimizing flag on compiler
Angelina: uses C++ instead of Java
Jennifer: faster processor
Angelina: makes data set pre-sorted (that cheating wench!)

I want you to create an algorithm that sorts all the students in this class. Time your algorithm; whoever's runs fastest will get stock options



What we want

- Rough Estimate
- Ignores Details
  - Or really: *independent* of details
  - We could have been sneaky & used those methods; but both algorithms would benefit
  - Even without an adversary, those factors will improve over time
  - So running time, pure and simple, is not a good measure

Big-O Analysis

- Ignores "details"
- What are some details we should ignore?
  - Speed of machine
  - Programming language used
  - Amount of memory
  - Order of input
  - Size of input (we'll talk about this in a second)
  - Compiler

Analysis of Algorithms

- Efficiency measure
  - how long the program runs time complexity
  - how much memory it uses space complexity
- *Why analyze at all?*
  - Decide which one to implement before going to the trouble
  - Given code, idea of where bottlenecks will be – without running and timing

Asymptotic Analysis

One "detail" we won't ignore – problem size, # elements

- 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!

Big-O: Common Names

- constant: $O(1)$
- logarithmic: $O(\log n)$
- linear: $O(n)$
- 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)

Exercise

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?

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
	Number of calls * work for each call

Analyze your code!

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: 4

Worst Case: 3n + 2

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: 4

Worst case: log n?
We'll analyze this later

Solving Recurrence Relations

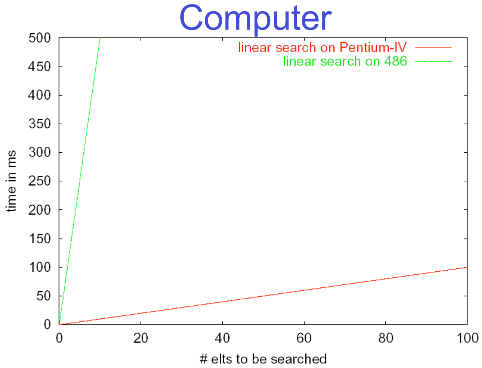
1. Determine the recurrence relation. What is the base case(s)?
2. "Expand" the original relation to find an equivalent general expression *in terms of the number of expansions*.
3. Find a closed-form expression by setting *the number of expansions* to a value which reduces the problem to a base case

Linear Search vs Binary Search

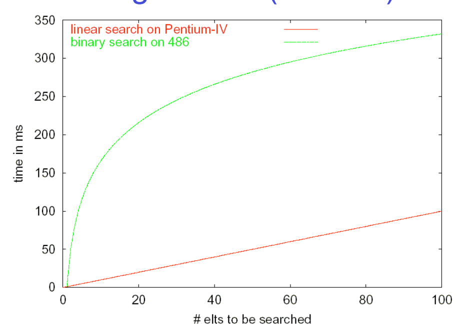
	Linear Search	Binary Search
Best Case	4 at [0]	4 at [mid]
Worst Case	3n+2	4 log n + 4

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

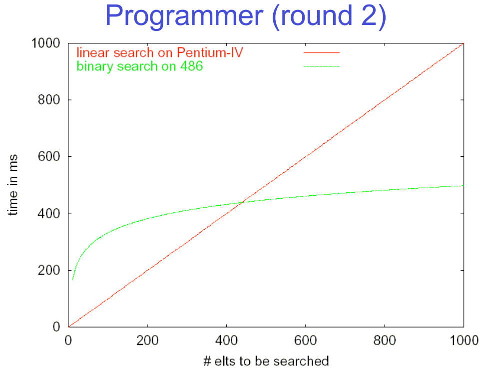
Fast Computer vs. Slow Computer



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 $T(n) = 3n + 2 \in \mathbf{O}(n)$
  - Binary search is $T(n) = 4 \log_2 n + 4 \in \mathbf{O}(\log n)$

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

Asymptotic Analysis

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

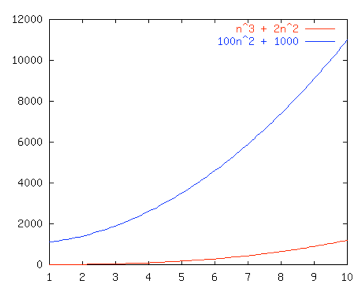
Properties of logs

- We will assume logs to base 2 unless specified otherwise
- log AB = log A + log B
- Proof:
  - $A=2^{\log_2 A}$ and $B=2^{\log_2 B}$
  - $AB = 2^{\log_2 A} \bullet 2^{\log_2 B} = 2^{\log_2 A+\log_2 B}$
  - so $\log_2 AB = \log_2 A + \log_2 B$
  - note: $\log AB \neq \log A \bullet \log B$
- log A/B = log A - log B
- $\log(A^B) = B \log A$
- Any base k log is equivalent to base 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*)

Definition of Order Notation

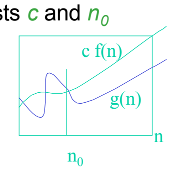
- Upper bound: $T(n) = O(f(n))$ Big-O
  Exist constants *c* and *n'* such that
  $T(n) \leq c\ f(n)$ for all $n \geq n'$
- Lower bound: $T(n) = \Omega(g(n))$ Omega
  Exist constants *c* and *n'* such that
  $T(n) \geq c\ g(n)$ for all $n \geq n'$
- Tight bound: $T(n) = \theta(f(n))$ Theta
  When both hold:
  $T(n) = O(f(n))$
  $T(n) = \Omega(f(n))$

Order Notation: 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:

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



Example: $g(n) = 1000n$ vs. $f(n) = n^2$

Is $g(n) \in O(f(n))$?

Pick: n0 = 1000, c = 1

$1000n \leq 1 * n^2$ for all $n \geq 1000$

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

Notation Notes

Note: Sometimes, you'll see the notation:

$g(n) = O(f(n))$.

This is equivalent to:

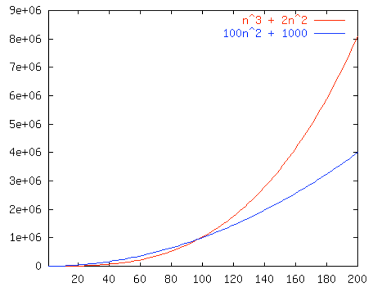
$g(n) \in O(f(n))$.

However: The notation

$O(f(n)) = g(n)$ is meaningless!

(in other words big-O is not symmetric)

Order Notation: Example



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

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

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 $g(n) \le c\, f(n)$ for all $n \ge n_0$
  - $g(n) \in o(\, f(n)\,)$ iff
    There exists a n_0 such that $g(n) < c\, f(n)$ for all c and $n \ge n_0$

    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 $g(n) \ge c\, f(n)$ for all $n \ge n_0$
  - $g(n) \in \omega(\, f(n)\,)$ iff
    There exists a n_0 such that $g(n) > c\, f(n)$ for all c and $n \ge n_0$

    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	>

Pros and Cons of Asymptotic Analysis

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 <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"

Algorithm Analysis Examples

- Consider the following program segment:

```
x:= 0;
for i = 1 to N do
  for j = 1 to N do
    x := x + 1;
```

- What is the value of x at the end?

Analyzing the Loop

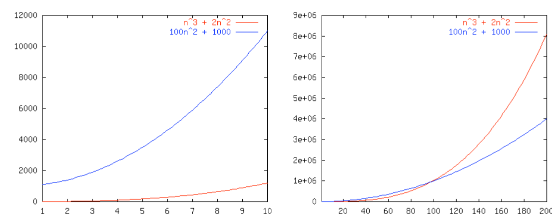
- Total number of times x is incremented is executed =

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

- Congratulations - You've just analyzed your first program!
  - Running time of the program is proportional to N(N+1)/2 for all N
  - Big-O ??

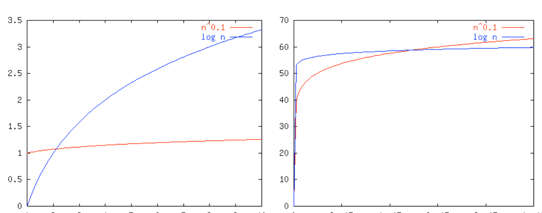
Which Function Grows Faster?

$n^3 + 2n^2$ vs. $100n^2 + 1000$



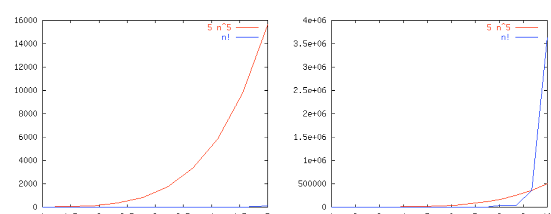
Which Function Grows Faster?

$n^{0.1}$ vs. $\log n$



Which Function Grows Faster?

$5n^5$ vs. $n!$



Nested Loops

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


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
```


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

- Eliminate low order terms
- Eliminate constant coefficients

$16n^3\log_8(10n^2) + 100n^2$
$\Rightarrow 16n^3\log_8(10n^2)$
$\Rightarrow n^3\log_8(10n^2)$
$\Rightarrow n^3[\log_8(10) + \log_8(n^2)]$
$\Rightarrow n^3\log_8(10) + n^3\log_8(n^2)$
$\Rightarrow n^3\log_8(n^2)$
$\Rightarrow 2n^3\log_8(n)$
$\Rightarrow n^3\log_8(n)$
$\Rightarrow n^3\log_8(2)\log(n)$
$\Rightarrow n^3\log(n)$