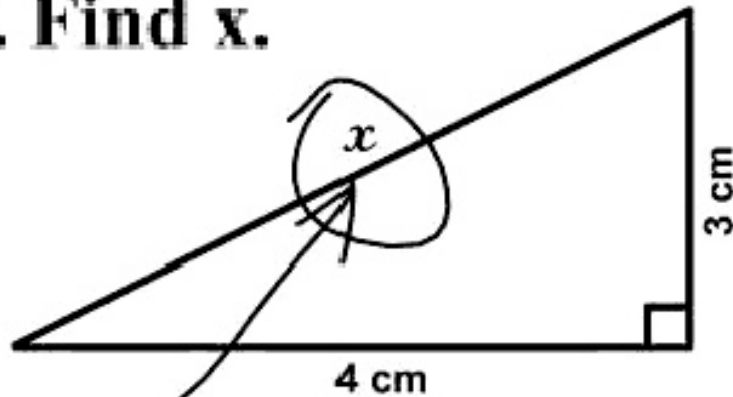


3. Find x .



Here it is

Searching, Sorting, and Asymptotic Complexity

Lecture 12
CS2110 – Fall 2008

Announcements

- Prelim 1
 - Thursday, October 16, 7:30 - 9pm, Uris Auditorium
 - Topics
 - ◆ all material up to (but not including) searching and sorting (this week's topics)
 - ◆ including interfaces & inheritance
- Exam conflicts
 - Email Kelly Patwell ASAP
- A3 due Friday, October 10, 11:59pm
- Prelim 1 review sessions
 - Wednesday 10/15, 7:30-9pm & 9-10:30pm, Upson B17 (sessions are identical)
 - See Exams on course website for more information
 - Individual appointments are available if you cannot attend the review sessions (email *one* TA to arrange appointment)
- Old exams are available for review on the course website

What Makes a Good Algorithm?

- Suppose you have two possible algorithms or data structures that basically do the same thing; which is *better*?
- Well... what do we mean by *better*?
 - Faster?
 - Less space?
 - Easier to code?
 - Easier to maintain?
 - Required for homework?
- How do we measure time and space for an algorithm?

Sample Problem: Searching

- Determine if a *sorted* array of integers contains a given integer
- First solution: Linear Search (check each element)

```
static boolean find(int[] a, int item) {  
    for (int i = 0; i < a.length; i++) {  
        if (a[i] == item) return true;  
    }  
    return false;  
}
```

Sample Problem: Searching

- Determine if a *sorted* array of integers contains a given integer
- First solution: Linear Search (check each element)

```
static boolean find(int[] a, int item) {
    for (int i = 0; i < a.length; i++) {
        if (a[i] == item) return true;
    }
    return false;
}
```

```
static boolean find(int[] a, int item) {
    for (int x : a) {
        if (x == item) return true;
    }
    return false;
}
```

Sample Problem: Searching

Second
solution:
Binary Search

```
static boolean find (int[] a, int item) {  
    int low = 0;  
    int high = a.length - 1;  
    while (low <= high) {  
        int mid = (low + high)/2;  
        if (a[mid] < item)  
            low = mid + 1;  
        else if (a[mid] > item)  
            high = mid - 1;  
        else return true;  
    }  
    return false;  
}
```

Linear Search vs Binary Search

- Which one is better?
 - Linear Search is easier to program
 - But Binary Search is faster... isn't it?
- How do we measure to show that one is faster than the other
 - Experiment?
 - Proof?
 - Which inputs do we use?
- Simplifying assumption #1: Use the *size* of the input rather than the input itself
 - For our sample search problem, the input size is $n+1$ where n is the array size
- Simplifying assumption #2: Count the number of “*basic steps*” rather than computing exact times

One Basic Step = One Time Unit

- Basic step:
 - input or output of a scalar value
 - accessing the value of a scalar variable, array element, or field of an object
 - assignment to a variable, array element, or field of an object
 - a single arithmetic or logical operation
 - method invocation (not counting argument evaluation and execution of the method body)
- For a conditional, count number of basic steps on the branch that is executed
- For a loop, count number of basic steps in loop body times the number of iterations
- For a method, count number of basic steps in method body (including steps needed to prepare stack-frame)

Runtime vs Number of Basic Steps

- But is this cheating?
 - The runtime is not the same as the number of basic steps
 - Time per basic step varies depending on computer, on compiler, on details of code...
- Well...yes, in a way
 - But the number of basic steps is *proportional* to the actual runtime
- Which is better?
 - n or n^2 time?
 - $100n$ or n^2 time?
 - $10,000n$ or n^2 time?
- As n gets large, multiplicative constants become less important
- **Simplifying assumption #3:**
Ignore multiplicative constants

Using Big-O to Hide Constants

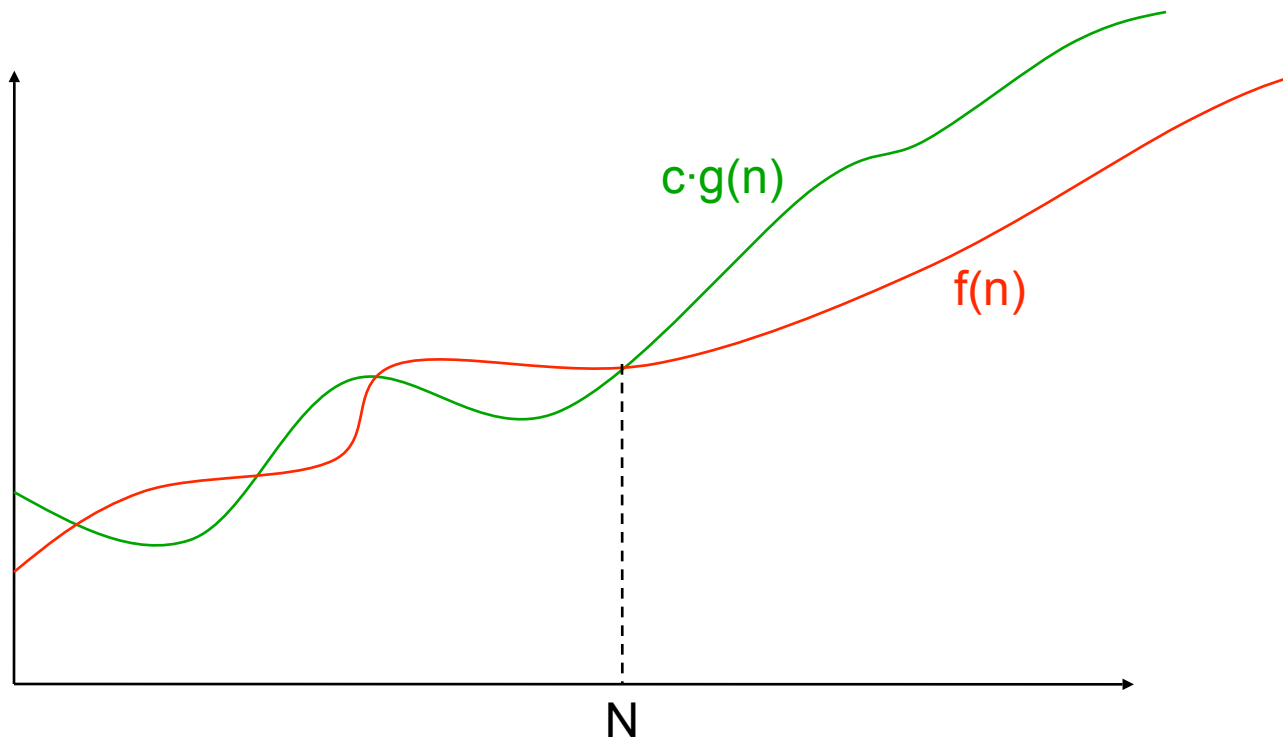
- We say $f(n)$ is *order of* $g(n)$ if $f(n)$ is bounded by a constant times $g(n)$
- Notation: $f(n)$ is $O(g(n))$
- Roughly, $f(n)$ is $O(g(n))$ means that $f(n)$ grows like $g(n)$ or slower, to within a constant factor
- "Constant" means fixed and independent of n

Example: $n^2 + n$ is $O(n^2)$

- We know $n \leq n^2$ for $n \geq 1$
- So $n^2 + n \leq 2n^2$ for $n \geq 1$
- So by definition, $n^2 + n$ is $O(n^2)$ for $c=2$ and $N=1$

Formal definition: $f(n)$ is $O(g(n))$ if there exist constants c and N such that for all $n \geq N$, $f(n) \leq c \cdot g(n)$

A Graphical View



- To prove that $f(n)$ is $O(g(n))$:
 - Find an N and c such that $f(n) \leq c g(n)$ for all $n \geq N$
 - We call the pair (c, N) a *witness pair* for proving that $f(n)$ is $O(g(n))$

Big-O Examples

Claim: $100n + \log n$ is $O(n)$

We know $\log n \leq n$ for $n \geq 1$

So $100n + \log n \leq 101n$
for $n \geq 1$

So by definition,

$100n + \log n$ is $O(n)$
for $c = 101$ and $N = 1$

Claim: $\log_B n$ is $O(\log_A n)$

since $\log_B n$ is $(\log_B A)(\log_A n)$

Question: Which grows faster:
 \sqrt{n} or $\log n$?

Big-O Examples

- Let $f(n) = 3n^2 + 6n - 7$
 - $f(n)$ is $O(n^2)$
 - $f(n)$ is $O(n^3)$
 - $f(n)$ is $O(n^4)$
 - ...
- $g(n) = 4n \log n + 34n - 89$
 - $g(n)$ is $O(n \log n)$
 - $g(n)$ is $O(n^2)$
- $h(n) = 20 \cdot 2^n + 40n$
 - $h(n)$ is $O(2^n)$
- $a(n) = 34$
 - $a(n)$ is $O(1)$
- Only the *leading* term (the term that grows most rapidly) matters

Problem-Size Examples

- Suppose we have a computing device that can execute 1000 operations per second; how large a problem can we solve?

	1 second	1 minute	1 hour
n	1000	60,000	3,600,000
$n \log n$	140	4893	200,000
n^2	31	244	1897
$3n^2$	18	144	1096
n^3	10	39	153
2^n	9	15	21

Commonly Seen Time Bounds

$O(1)$	constant	excellent
$O(\log n)$	logarithmic	excellent
$O(n)$	linear	good
$O(n \log n)$	$n \log n$	pretty good
$O(n^2)$	quadratic	OK
$O(n^3)$	cubic	maybe OK
$O(2^n)$	exponential	too slow

Worst-Case/Expected-Case Bounds

- We can't possibly determine time bounds for all possible inputs of size n
- **Simplifying assumption #4:** Determine number of steps for either
 - worst-case or
 - expected-case
- Worst-case
 - Determine how much time is needed for the *worst possible* input of size n
- Expected-case
 - Determine how much time is needed *on average* for all inputs of size n

Our Simplifying Assumptions

- Use the **size** of the input rather than the input itself – n
- Count the number of “basic steps” rather than computing exact times
- Multiplicative constants and small inputs ignored (order-of, big-O)
- Determine number of steps for either
 - worst-case
 - expected-case
- **These assumptions allow us to analyze algorithms effectively**

Worst-Case Analysis of Searching

Linear Search

```
static boolean find (int[] a, int item) {  
    for (int i = 0; i < a.length; i++) {  
        if (a[i] == item) return true;  
    }  
    return false;  
}
```

worst-case time = $O(n)$

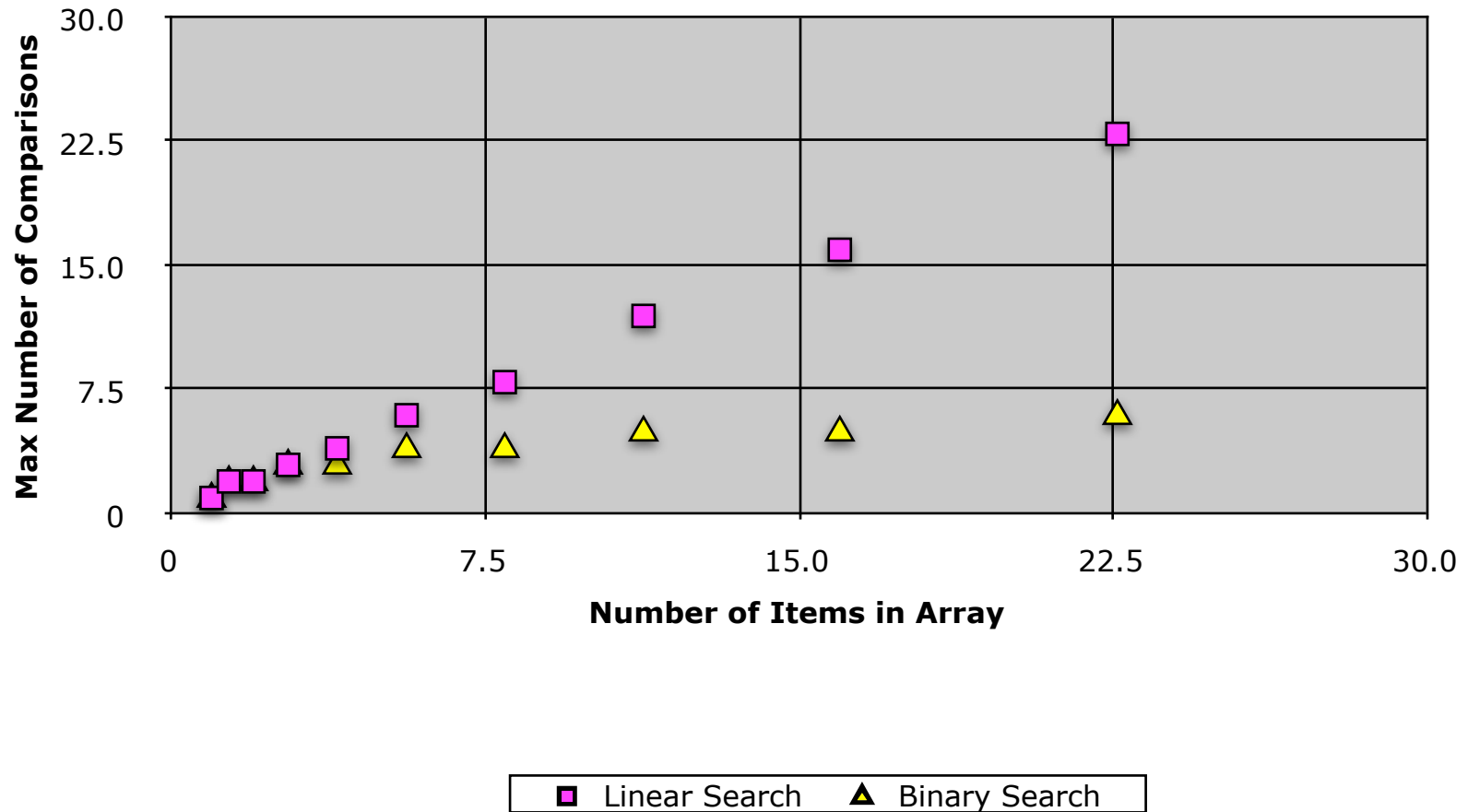
Binary Search

```
static boolean find (int[] a, int item) {  
    int low = 0;  
    int high = a.length - 1;  
    while (low <= high) {  
        int mid = (low + high)/2;  
        if (a[mid] < item)  
            low = mid+1;  
        else if (a[mid] > item)  
            high = mid - 1;  
        else return true;  
    }  
    return false;  
}
```

worst-case time = $O(\log n)$

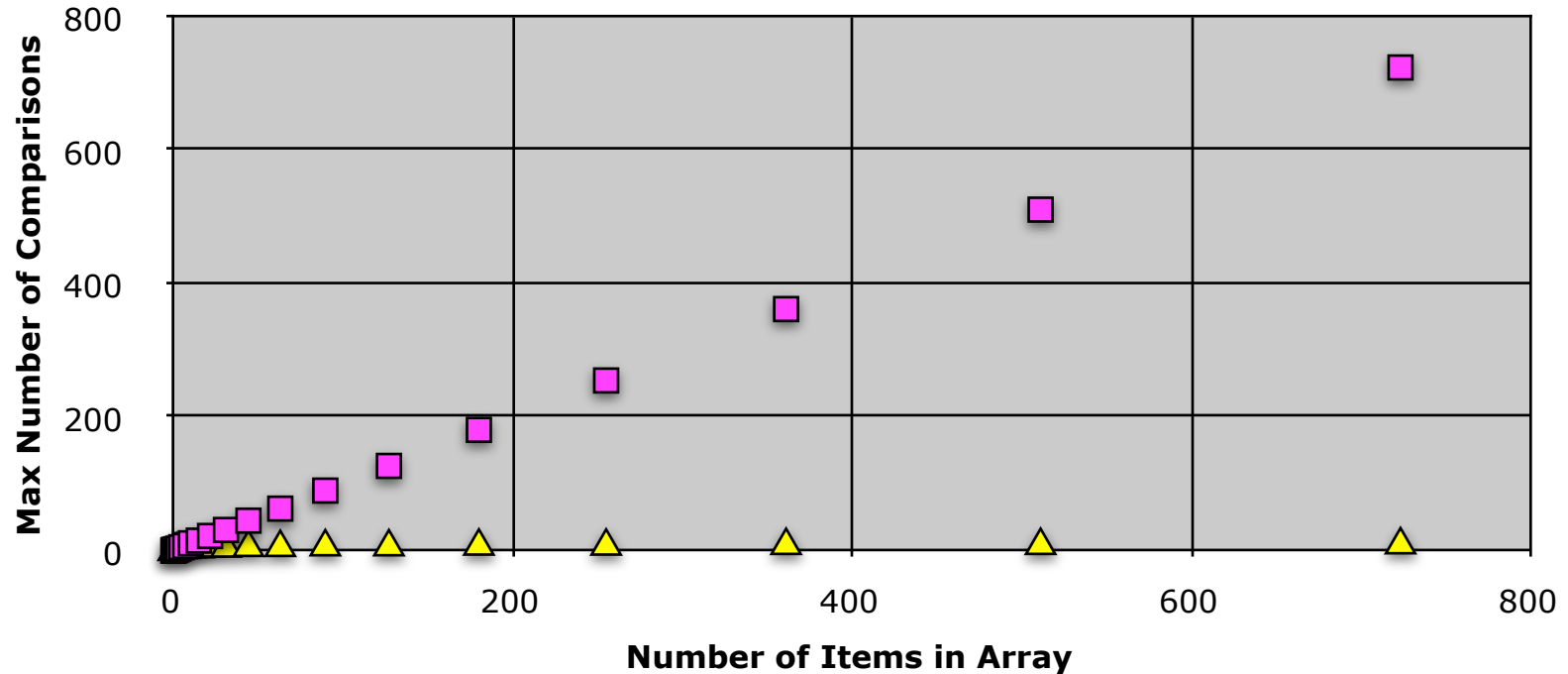
Comparison of Algorithms

Linear vs. Binary Search



Comparison of Algorithms

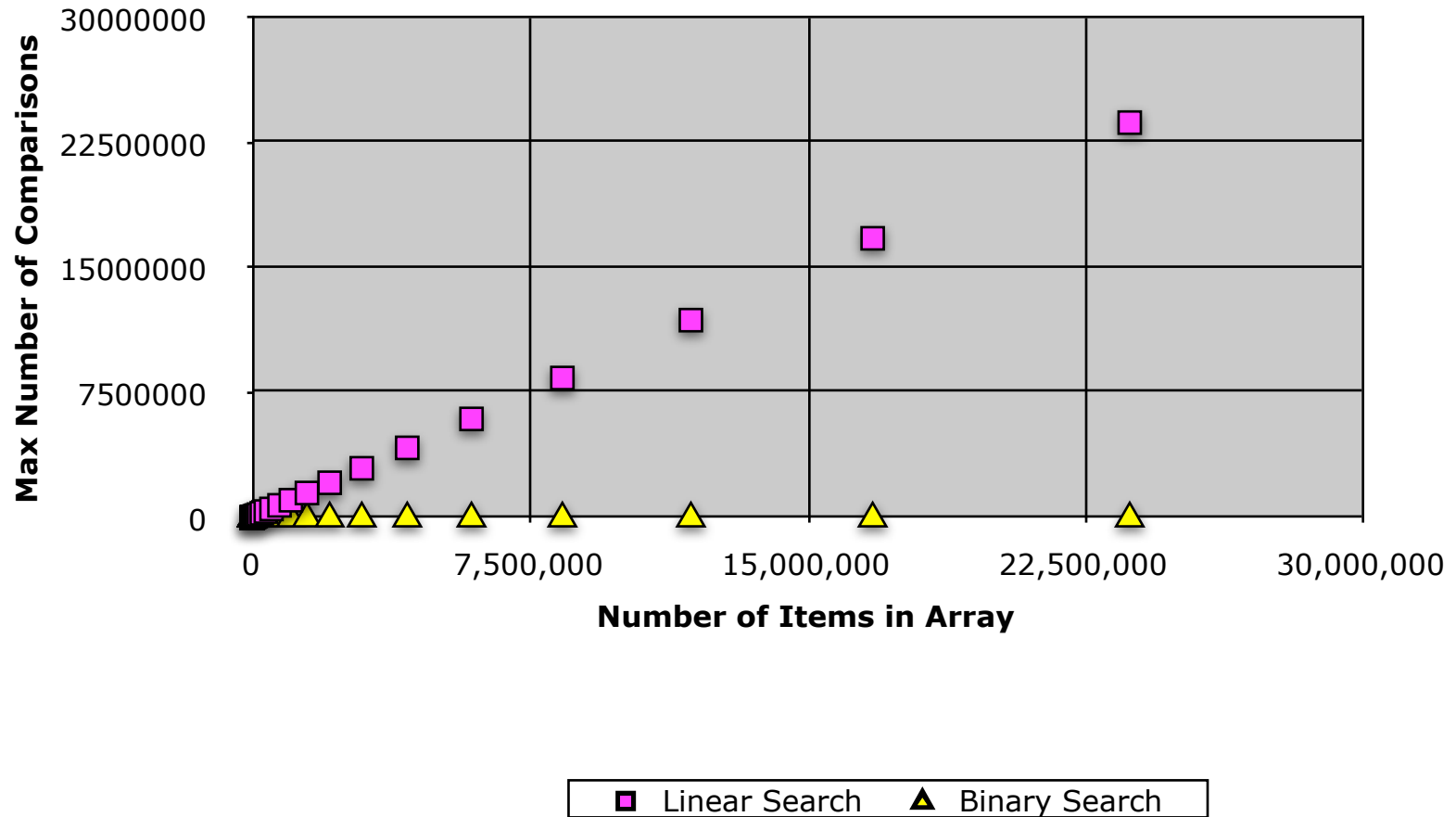
Linear vs. Binary Search



■ Linear Search ▲ Binary Search

Comparison of Algorithms

Linear vs. Binary Search



Analysis of Matrix Multiplication

By convention, matrix problems are measured in terms of n , the number of rows and columns

- Note that the input size is really $2n^2$, not n
- Worst-case time is $O(n^3)$
- Expected-case time is also $O(n^3)$

Code for multiplying n -by- n matrices A and B :

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++) {
    C[i][j] = 0;
    for (k = 0; k < n; k++)
      C[i][j] += A[i][k]*B[k][j];
  }
```

Remarks

- Once you get the hang of this, you can quickly zero in on what is relevant for determining asymptotic complexity
 - For example, you can usually ignore everything that is not in the innermost loop. Why?
- Main difficulty:
 - Determining runtime for recursive programs

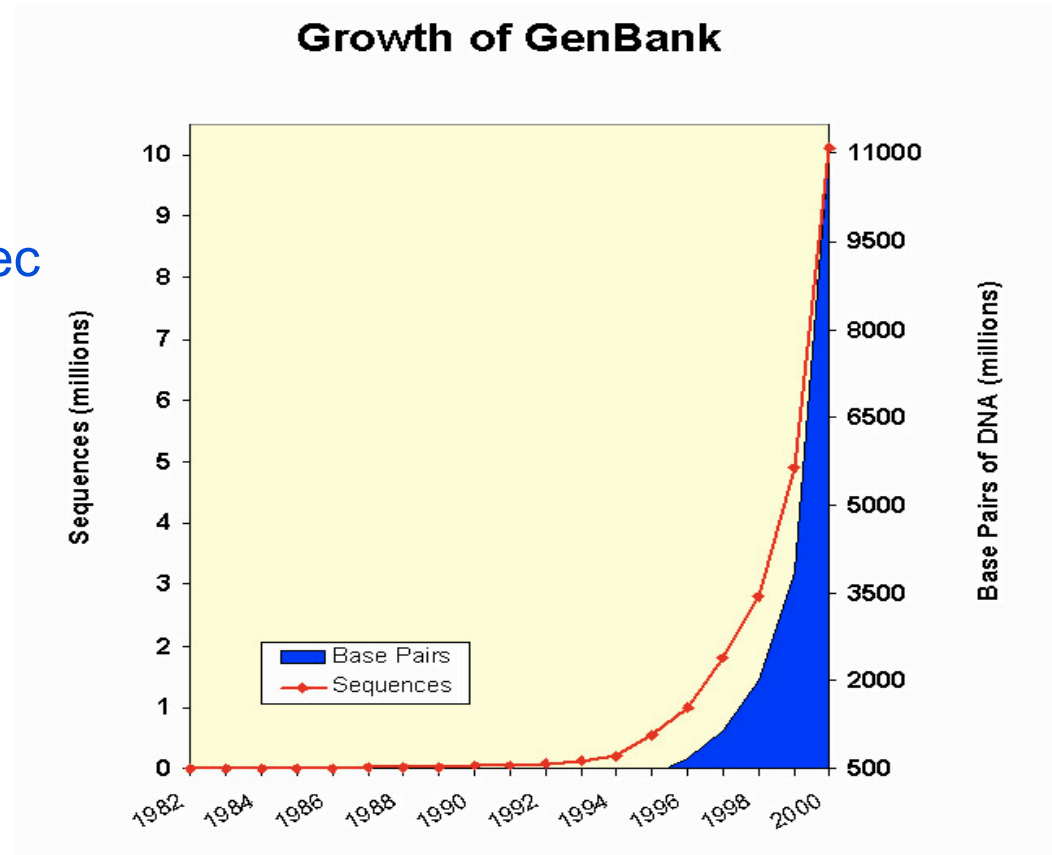
Why Bother with Runtime Analysis?

- Computers are so fast these days that we can do whatever we want using just simple algorithms and data structures, right?
- Well...not really – data-structure/algorithm improvements can be a *very big* win
- Scenario:
 - A runs in n^2 msec
 - A' runs in $n^2/10$ msec
 - B runs in $10 n \log n$ msec
- Problem of size $n=10^3$
 - A: 10^3 sec \approx 17 minutes
 - A': 10^2 sec \approx 1.7 minutes
 - B: 10^2 sec \approx 1.7 minutes
- Problem of size $n=10^6$
 - A: 10^9 sec \approx 30 years
 - A': 10^8 sec \approx 3 years
 - B: $2 \cdot 10^5$ sec \approx 2 days

1 day = 86,400 sec \approx 10^5 sec
1,000 days \approx 3 years

Algorithms for the Human Genome

- Human genome
= 3.5 billion nucleotides
~ 1 Gb
- @1 base-pair instruction/ μ sec
 - $n^2 \rightarrow 388445$ years
 - $n \log n \rightarrow 30.824$ hours
 - $n \rightarrow 1$ hour



Limitations of Runtime Analysis

- Big-O can hide a very large constant
 - Example: selection
 - Example: small problems
- The specific problem you want to solve may not be the worst case
 - Example: Simplex method for linear programming
- Your program may not be run often enough to make analysis worthwhile
 - Example: one-shot vs. every day
- You may be analyzing and improving the wrong part of the program
 - *Very common situation*
 - Should use *profiling* tools

Summary

- Asymptotic complexity
 - Used to measure of time (or space) required by an algorithm
 - Measure of the *algorithm*, not the *problem*
- Searching a sorted array
 - Linear search: $O(n)$ worst-case time
 - Binary search: $O(\log n)$ worst-case time
- Matrix operations:
 - Note: $n = \text{number-of-rows} = \text{number-of-columns}$
 - Matrix-vector product: $O(n^2)$ worst-case time
 - Matrix-matrix multiplication: $O(n^3)$ worst-case time
- More later with sorting and graph algorithms