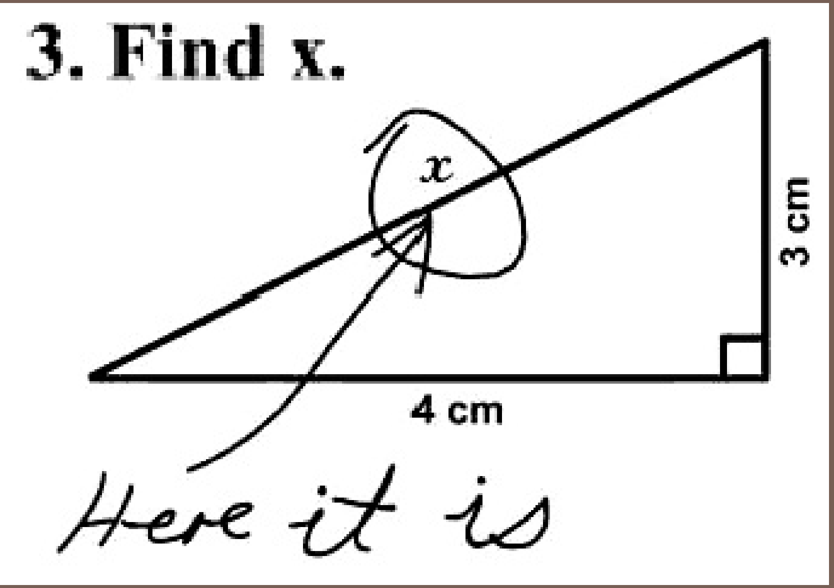


SEARCHING, SORTING, AND ASYMPTOTIC COMPLEXITY

Lecture 12
CS2110 – Fall 2009

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 *sorted* array a contains integer v
- First solution: Linear Search (check each element)

```
/** return true iff v is in a */
static boolean find(int[] a, int v) {
  for (int i = 0; i < a.length; i++) {
    if (a[i] == v) return true;
  }
  return false;
}
```

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

Sample Problem: Searching

Second solution:
Binary Search

Still returning true iff v is in a

Keep true: all occurrences of v are in b[low..high]

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

Linear Search vs Binary Search

Which one is better?

- Linear: easier to program
- Binary: faster… isn’t it?

How do we measure speed?

- Experiment?
- Proof?
- What inputs do we use?

- Simplifying assumption #1: Use *size* of input rather than input itself
  - For sample search problem, input size is n+1 where n is array size

- Simplifying assumption #2: Count number of "*basic steps*" rather than computing exact times

One Basic Step = One Time Unit

Basic step:

- Input/output of scalar value
- Access value of scalar variable, array element, or object field
- assign to variable, array element, or object field
- do one arithmetic or logical operation
- method invocation (not counting arg evaluation and execution of method body)

- For conditional: number of basic steps on branch that is executed
- For loop: (number of basic steps in loop body) * (number of iterations)
- For method: number of basic steps in method body (include steps needed to prepare stack-frame)

Runtime vs Number of Basic Steps

Is this cheating?

- The runtime is not the same as number of basic steps
- Time per basic step varies depending on computer, compiler, 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?
- 100 n or n^2 time?
- 10,000 n or n^2 time?

As n gets large, multiplicative constants become less important

Simplifying assumption #3: Ignore multiplicative constants

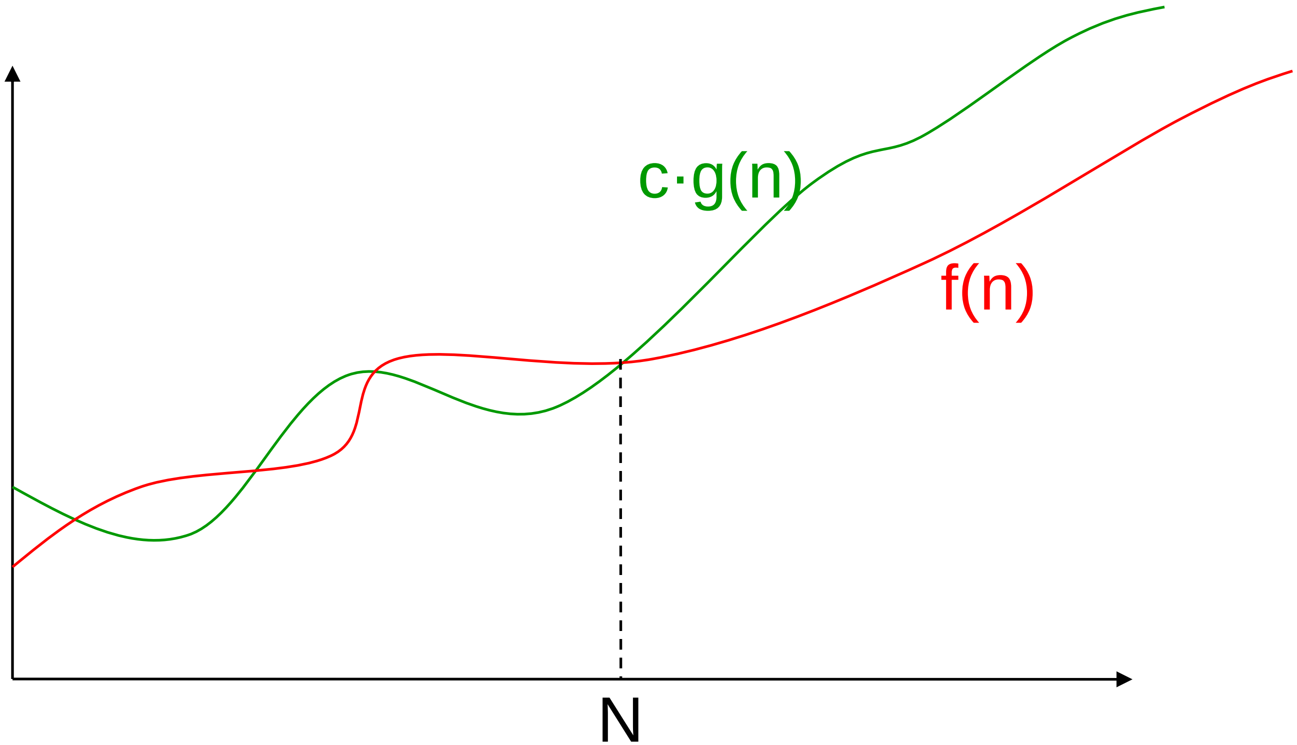
Using Big-O to Hide Constants

- 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 $n^2 + n \leq 2 n^2$ 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 N and c such that f(n) ≤ c g(n) for all n > N
- Pair (c, N) is a *witness pair* for proving that f(n) is O(g(n))

Big-O Examples

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

We know log n ≤ n for n ≥ 1

So 100 n + log n ≤ 101 n

for n ≥ 1

So by definition,

100 n + 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: 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 $O(n^4)$
- …

$g(n) = 4\, n \log n + 34\, n - 89$

- $g(n)$ 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

- Considier 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

Simplifying Assumptions

Use the size of the input rather than the input itself – n

Count the number of “basic steps” rather than computing exact time

Ignore multiplicative constants and small inputs (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

```
/** return true iff v is in a */
static bool find (int[] a, int v) {
    for (int x : a) {
      if (x == v) return true;
    }
   return false;
}
```

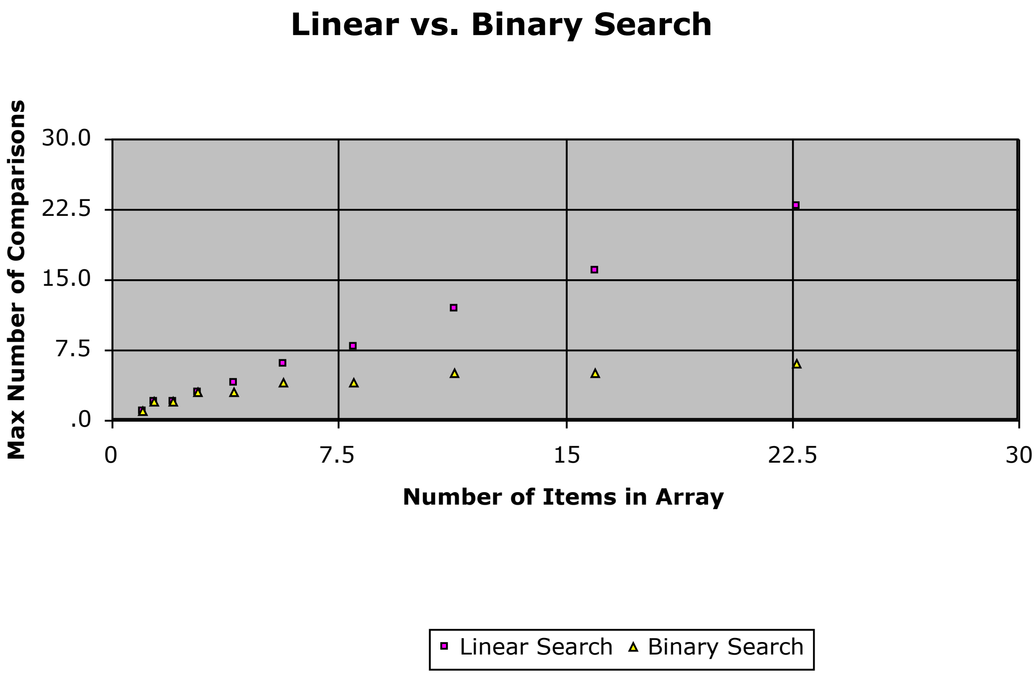
worst-case time: O(n)

Binary Search

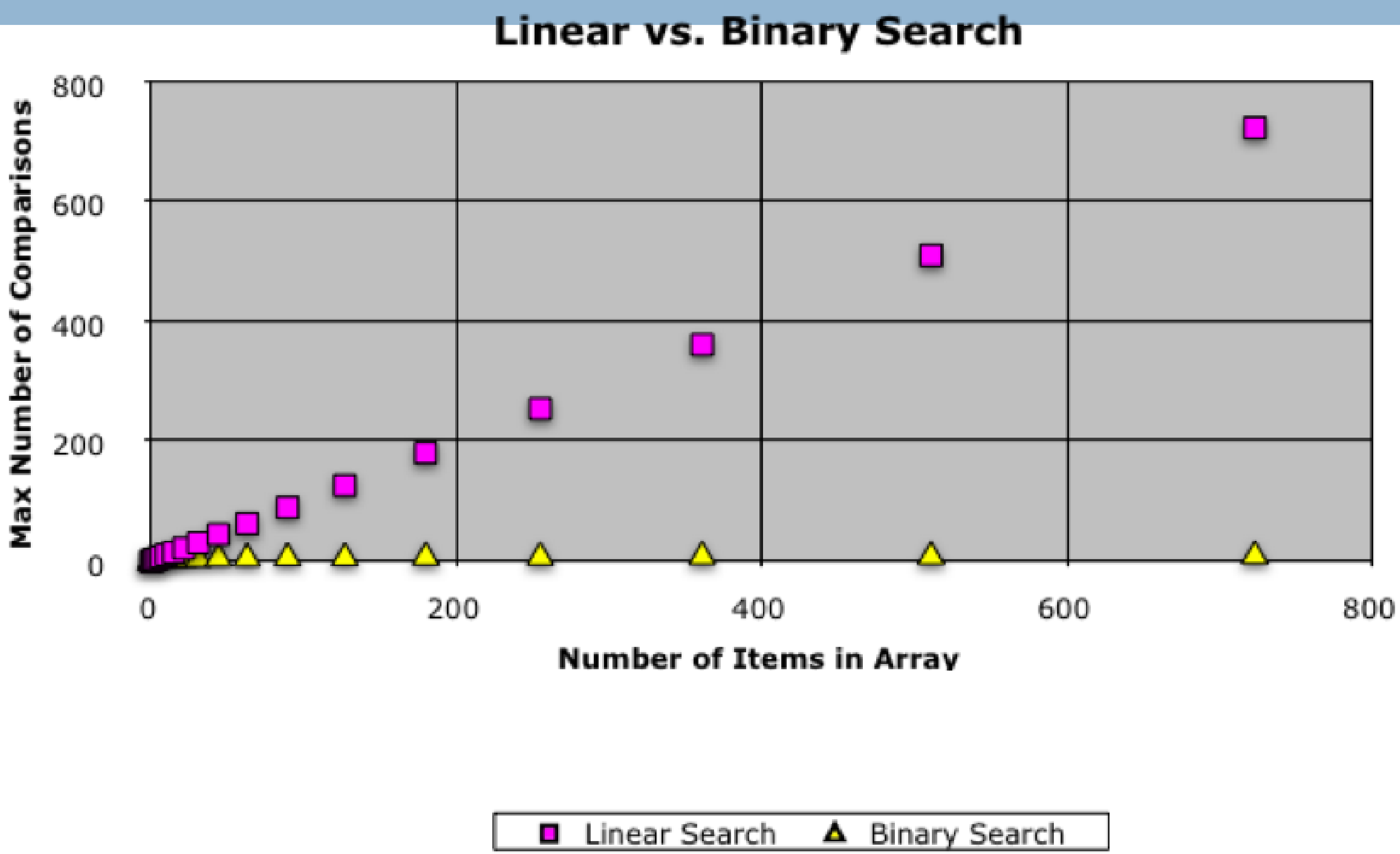
```
static bool find (int[] a, int v) {
   int low= 0;
   int high= a.length – 1;
   while (low <= high) {
      int mid = (low + high)/2;
      if (a[mid] == v) return true;
      if (a[mid] < v)
              low= mid + 1;
      else   high= mid - 1;
     }
    return false;
 }
```

worst-case time: O(log n)

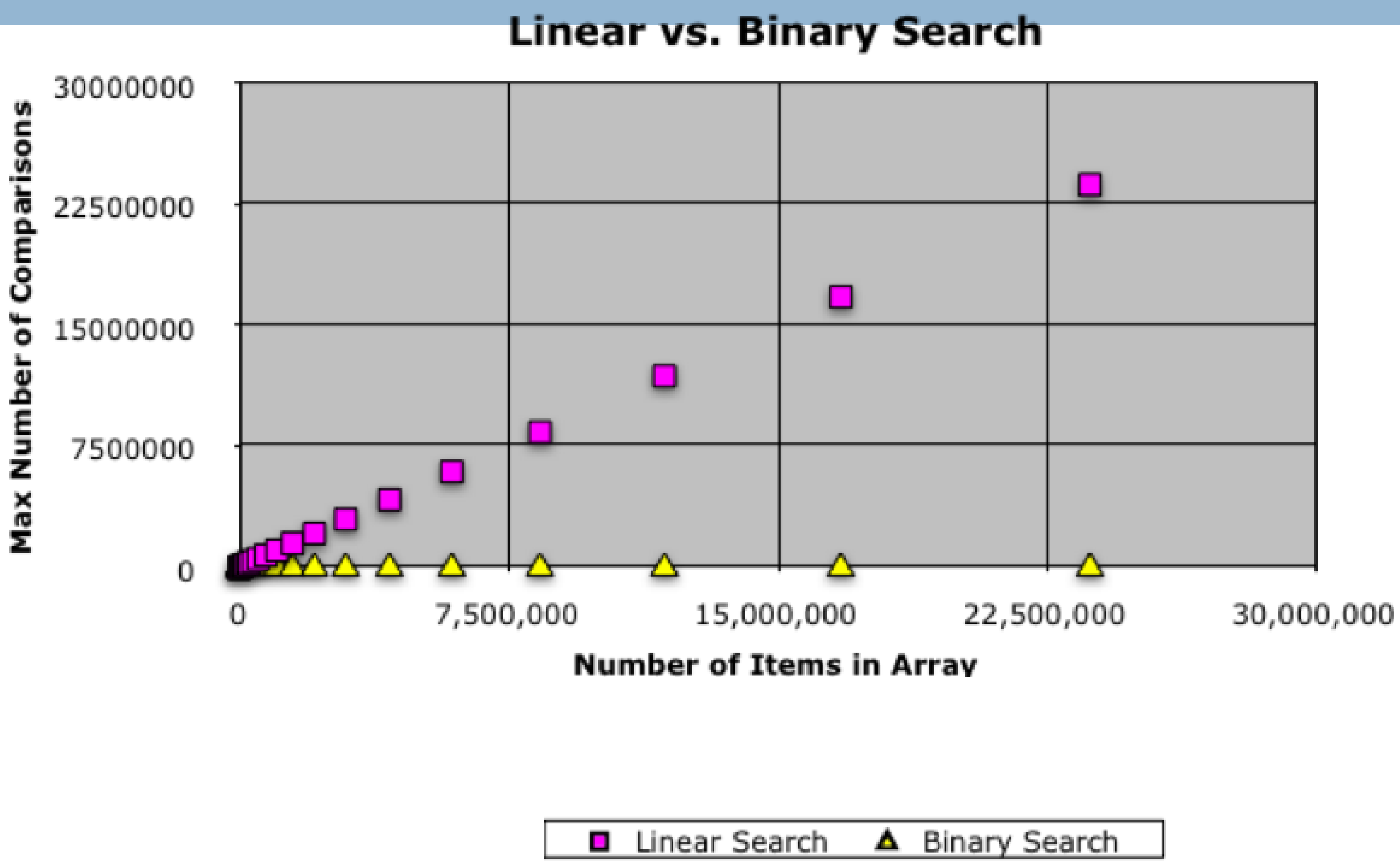
Comparison of Algorithms



Comparison of Algorithms



Comparison of Algorithms



Analysis of Matrix Multiplication

Multiply n-by-n matrices A and B:

Convention, matrix problems measured in terms of n, the number of rows, columns

- Input size is really $2n^2$, not n
- Worst-case time: $O(n^3)$
- Expected-case time: $O(n^3)$

```
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

- 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 so fast that we can do whatever we want using simple algorithms and data structures, right?

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 ≈ 17 minutes
- A': 10^2 sec ≈ 1.7 minutes
- B: 10^2 sec ≈ 1.7 minutes

Problem of size $n=10^6$

- A: 10^9 sec ≈ 30 years
- A': 10^8 sec ≈ 3 years
- B: $2 \cdot 10^5$ sec ≈ 2 days

1 day = 86,400 sec ≈ 10^5 sec

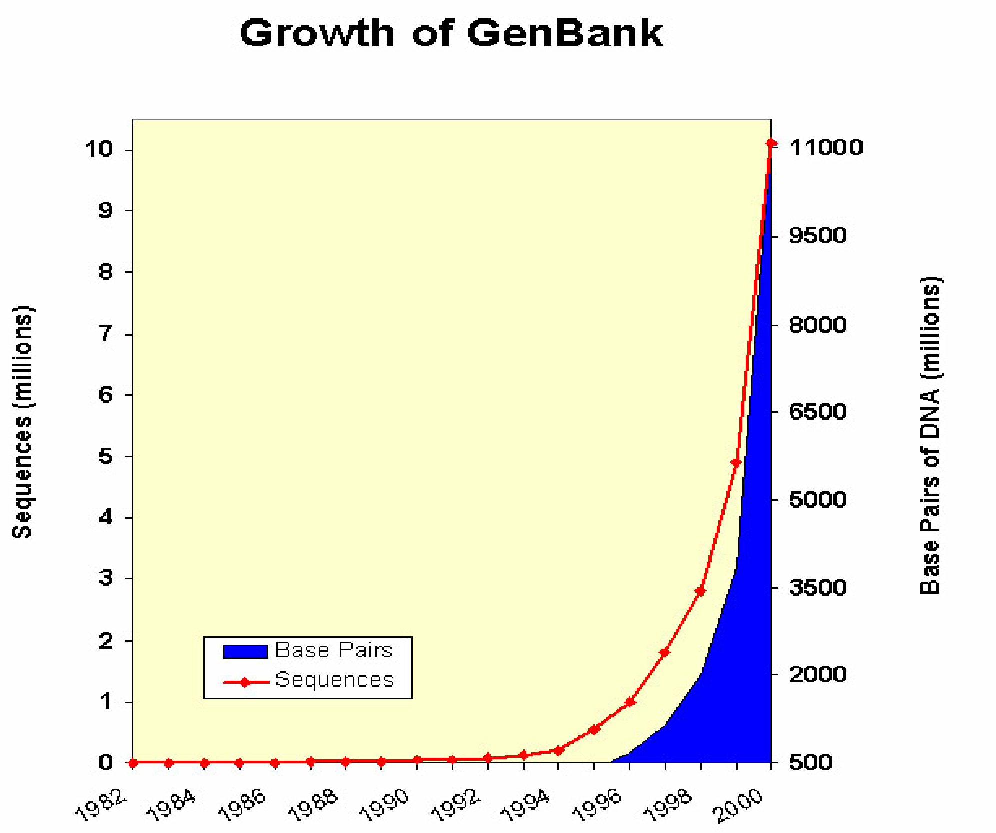
1,000 days ≈ 3 years

Algorithms for the Human Genome

Human genome
= 3.5 billion nucleotides
~ 1 Gb

@1 base-pair instruction/⌠ 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 = number-of-rows = 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