CSE 326: Data Structures

Asymptotic Analysis

James Fogarty
Autumn 2007
Lecture 2

Bring to Class on Wednesday:

- Name
- Email address
- Year (1,2,3,4)
- Major
- Hometown
- Interesting Fact or “What I did on my summer vacation”

Algorithm Analysis: Why?

- Correctness:
  - Does the algorithm do what is intended.
- Performance:
  - What is the running time of the algorithm.
  - How much storage does it consume.
- Different algorithms may be correct
  - Which should I use?

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
```

Proof by Induction

- **Basis Step:** 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.

Program Correctness by Induction

- **Basis Step:**
  `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]+v[k]`

- **Inductive Step (n=k+1):**
  `sum(v,n)` returns

```
 v[k]+sum(v,k-1)= (by inductive hyp.)
 v[k]+(v[0]+v[1]+…+v[k-1])=
 v[0]+v[1]+…+v[k-1]+v[k] ✓
```

Algorithms vs Programs

- Proving correctness of an algorithm is very important
  - a well designed algorithm is guaranteed to work correctly and its performance can be estimated

- Proving correctness of a program (an implementation) is fraught with weird bugs
  - Abstract Data Types are a way to bridge the gap between mathematical algorithms and programs

Comparing Two Algorithms

GOAL: Sort a list of names

“I’ll buy a faster CPU”

“I’ll use C++ instead of Java – wicked fast!”

“Ooh look, the –O4 flag!”

“Who cares how I do it, I’ll add more memory!”

“Can’t I just get the data pre-sorted??”

Comparing Two Algorithms

- What we want:
  - Rough Estimate
  - Ignores Details

- Really, *independent* of details
  - Coding tricks, CPU speed, compiler optimizations, ...
  - These would help any algorithms equally
  - Don't just care about running time – not a good enough measure

Big-O Analysis

- Ignores “details”
- What details?
  - CPU speed
  - Programming language used
  - Amount of memory
  - Compiler
  - Order of input
  - Size of input … sorta.

Analysis of Algorithms

- Efficiency measure
  - how long the program runs   time complexity
  - how much memory it uses   space complexity
- Why analyze at all?
  - Decide what algorithm to implement before actually doing it
  - Given code, get a sense for where bottlenecks must be, without actually measuring it

Asymptotic Analysis

One detail we won't ignore:
problem size, # of input 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, ...
- Difference really shows up as *n* grows!

Exercise - Searching

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

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:

3

Worst Case:

2n+1

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?

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*.

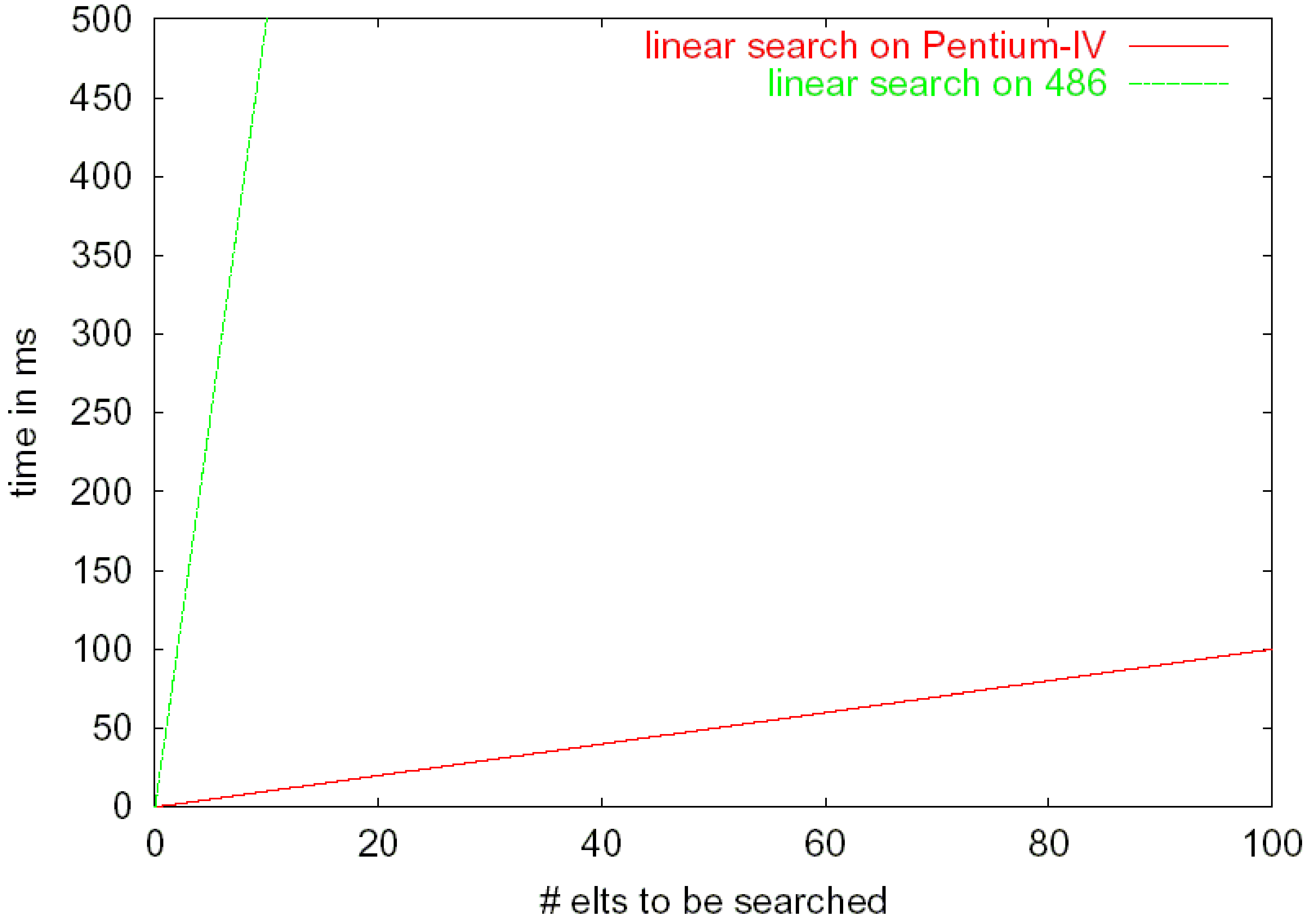
3. Find a closed-form expression by setting *the number of expansions* to a value which reduces the problem to a base case

Linear Search vs Binary Search

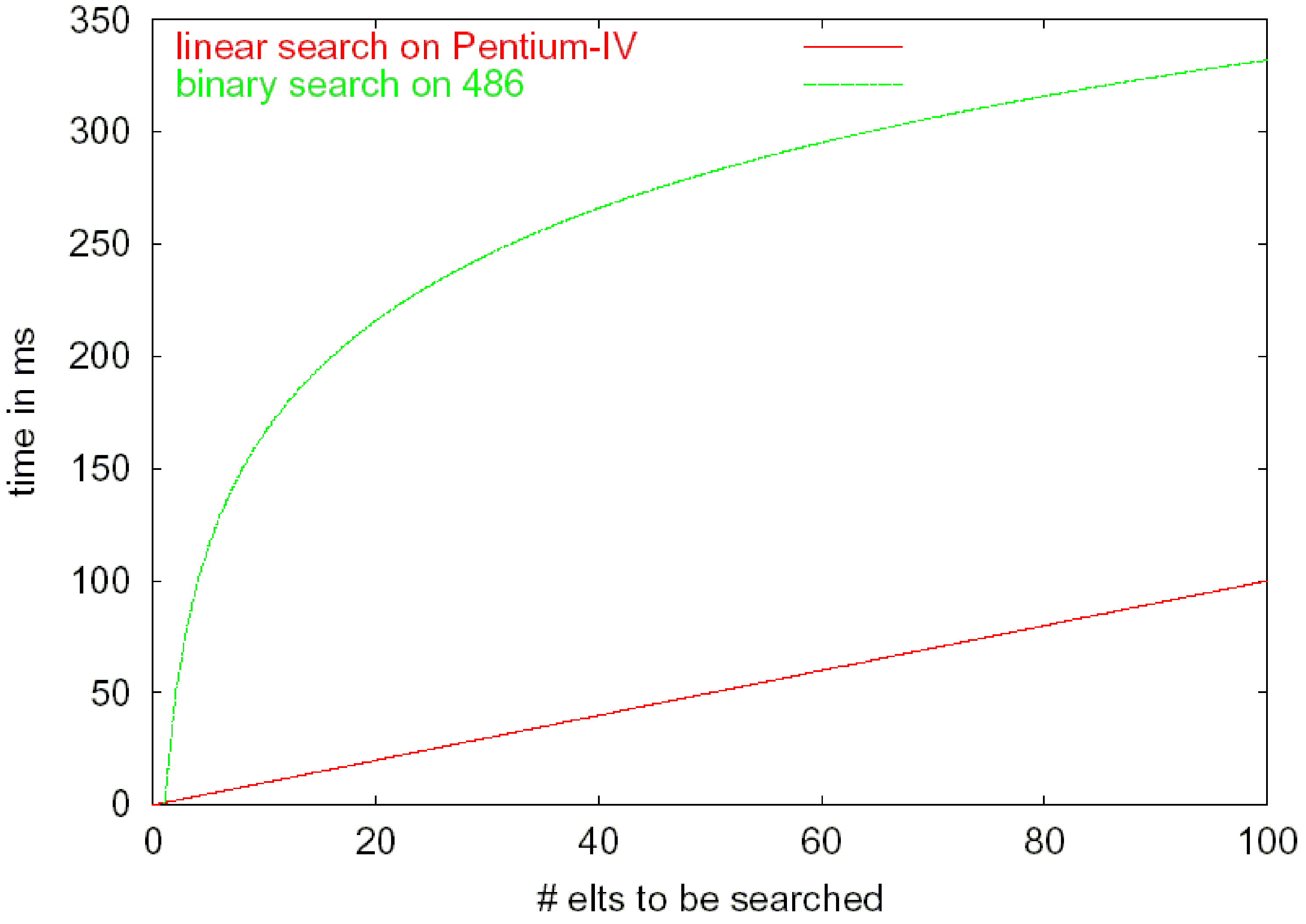
	Linear Search	Binary Search
Best Case		
Worst Case		

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

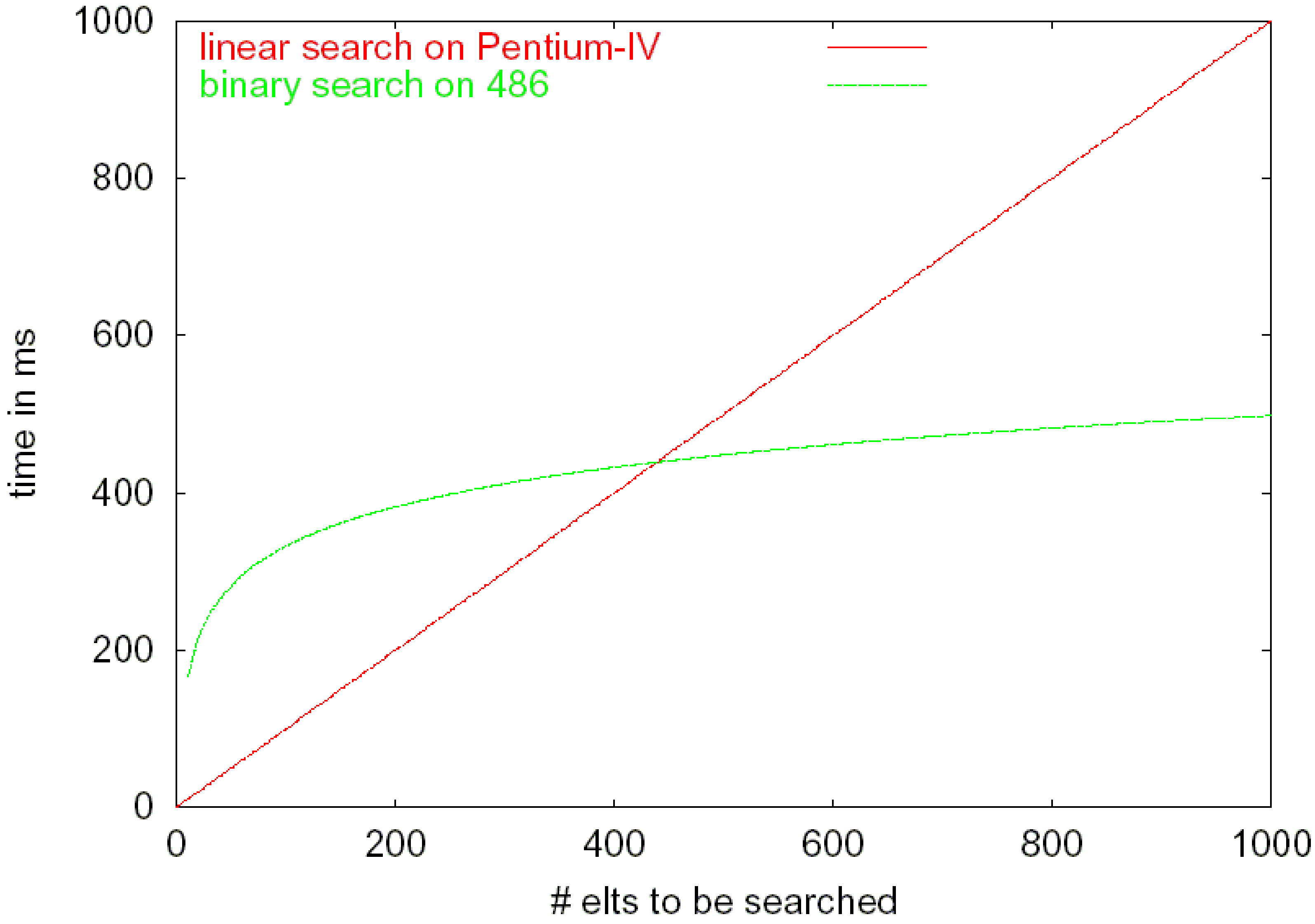
Fast Computer vs. Slow Computer



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

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