CSE 373
Data Structures & Algorithms
Guest Lecturer: VIBHOR RASTOGI

Lecture 03
Asymptotic Analysis

Algorithm Analysis

- Correctness:
  - Does the algorithm do what is intended.

- Performance:
  - Speed   time complexity
  - Memory   space complexity

- Why analyze ?

Correctness

Correctness of an algorithm is established by proof. Common approaches:

- (Dis)proof by counterexample
- Proof by contradiction
- Proof by induction
    - Especially useful in recursive algorithms

Proof by Induction

- **Base Case:** 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.

Recursive algorithm for *sum*

- Write a *recursive* function to find the sum of the first **n** integers stored in array **v**.

```
sum(int array v, int n) returns int
  if n = 0 then
    sum = 0
  else
    sum = nth number + sum of first n-1 numbers
  return sum
```

Program Correctness by Induction

- Base Case:

- Inductive Hypothesis (n=k):

- Inductive Step (n=k+1):

How to measure performance?

- Empirical approach:
  - Run the program many times
  - Caveats
    - Must implement every alg you want to evaluate
    - Results may be particular to type of machine used
    - need to test on many data
- Analytical approach:
  - Count operations, come up with (upper/lower) bounds on running time

Analyzing Performance

We will focus on analyzing time complexity. First, we have some "rules" to help measure how long it takes to do things:

Basic operations	Constant time
Consecutive statements	Sum of times
Conditionals	Test, plus larger branch cost
Loops	Sum of iterations
Function calls	Cost of function body
Recursive functions	Solve recurrence relation…

Second, we will be interested in **best** and **worst** case performance.

Complexity cases

We'll start by focusing on two cases.

Problem size **N**

- **Worst-case complexity**: **max** # steps algorithm takes on "most challenging" input of size **N**
- **Best-case complexity: min** # steps algorithm takes on "easiest" input of size **N**

Exercise - Searching

2	3	5	16	37	50	73	75

```
int ArrayContains(int array[], int n, int key){
    // Insert your algorithm here

}
```

What algorithm would you choose to implement this code snippet?

Linear Search Analysis

```
int LinearArrayContains(int array[], int n, int key ) {
   for( int i = 0; i < n; i++ ) {
       if( array[i] == key )
           // Found it!
           return i;
   }
   return -1;
}
```

Best case:

Worst case:

Binary Search Analysis

2	3	5	16	37	50	73	75

```
Int BinArrayContains( int array[], int low, int high, int key)
{
   // The subarray is empty
   if( low > high ) return -1;

   // Search this subarray recursively
   int mid = (high + low) / 2;
   if( key == array[mid] ) {
       return mid;
   } else if( key < array[mid] ) {
       return BinArrayContains( array, low, mid-1, key );
   } else {
       return BinArrayContains( array, mid+1, high, key );
}
```

Best case:

Worst case:

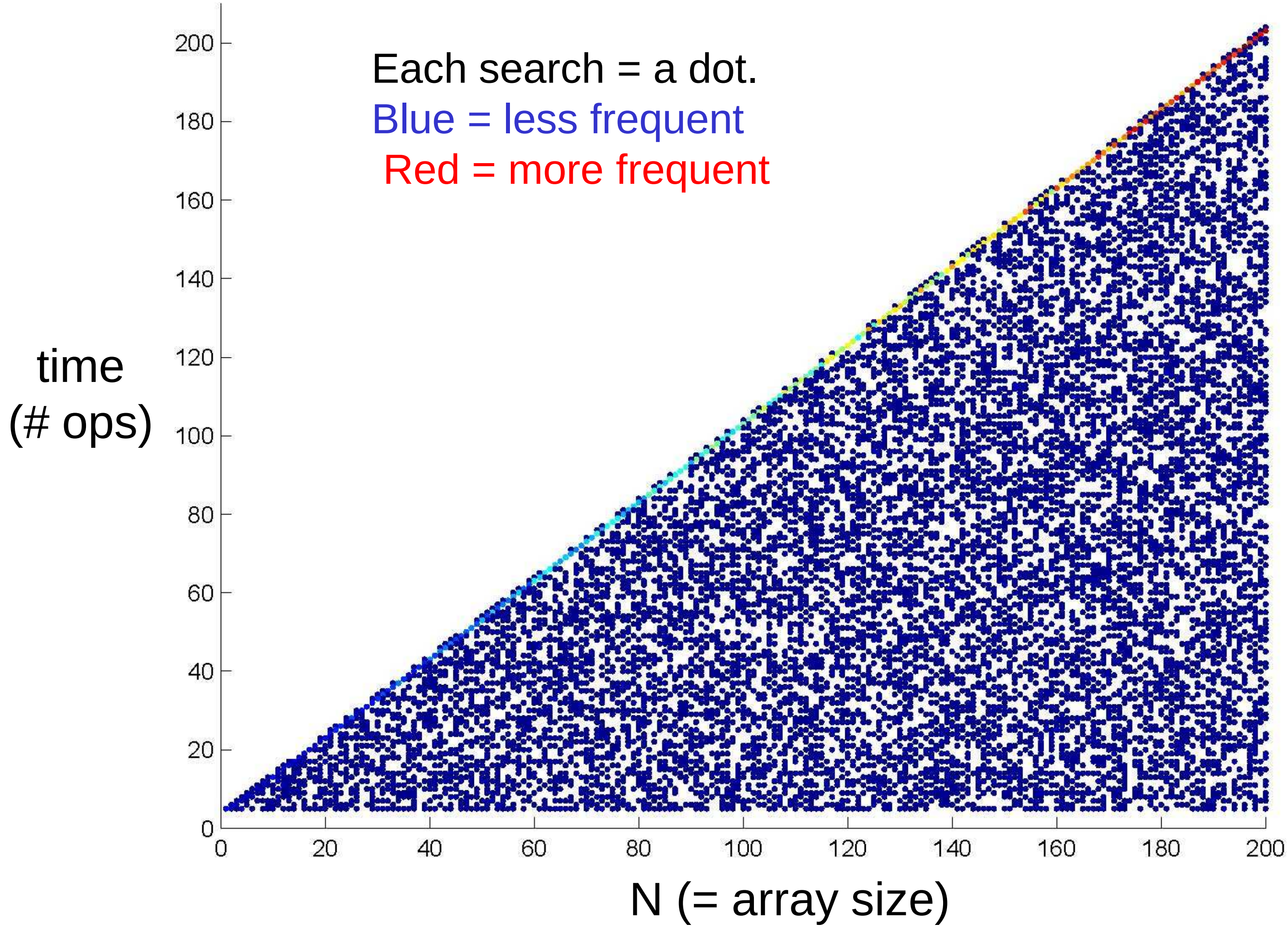
Solving Recurrence Relations

- Determine the recurrence relation and base case(s).
- “Expand” the original relation to find an equivalent expression in terms of the number of expansions (k).
- Find a closed-form expression by setting k to a value which reduces the problem to a base case

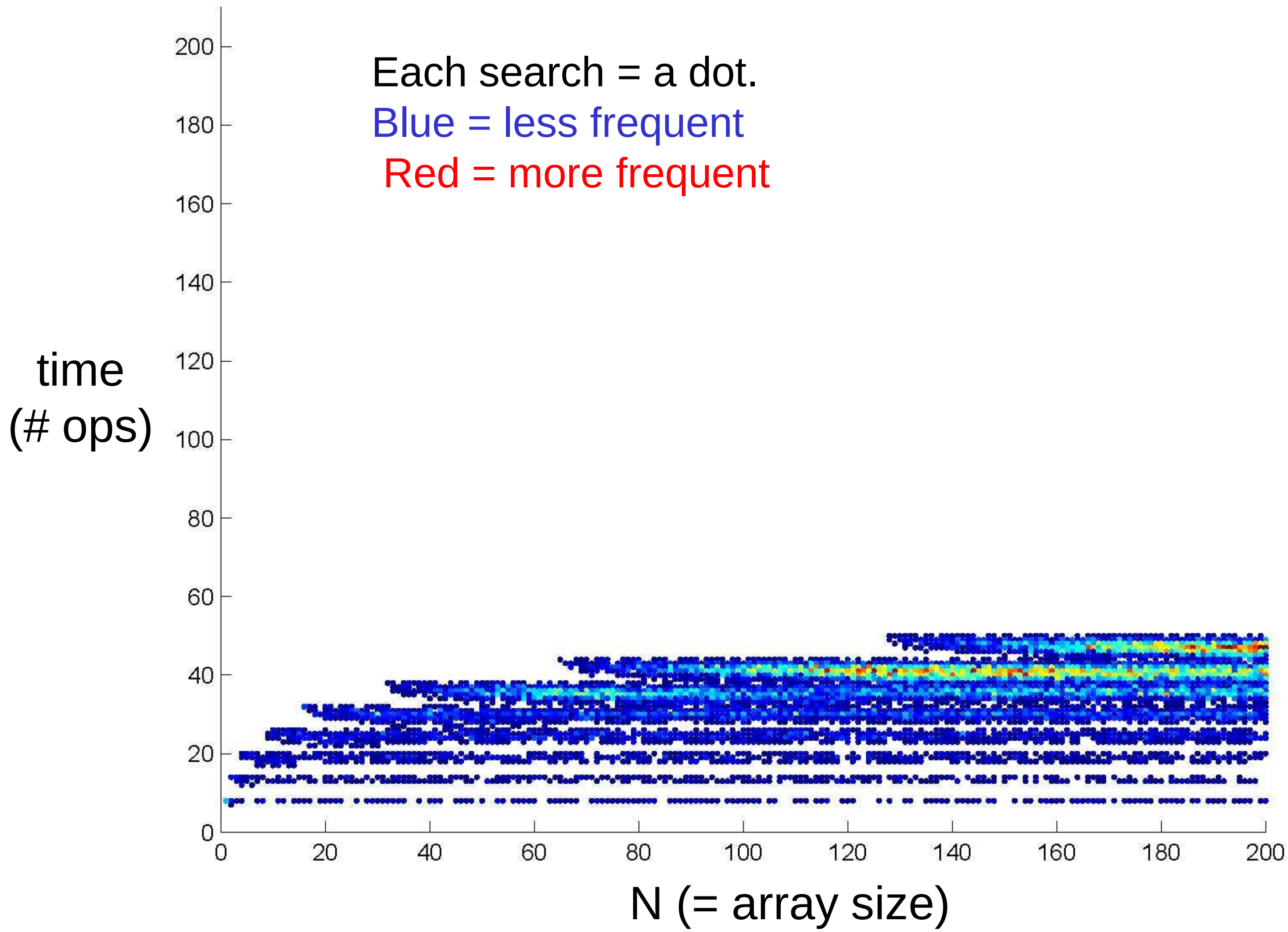
Linear Search vs Binary Search

	Linear Search	Binary Search
Best Case	4	5 at [middle]
Worst Case	3n+3	$4 \lfloor \log n \rfloor + 2$

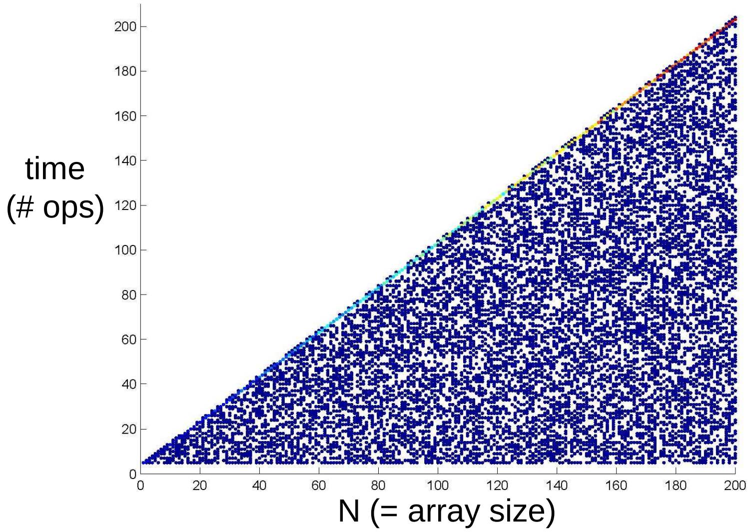
Linear search—empirical analysis



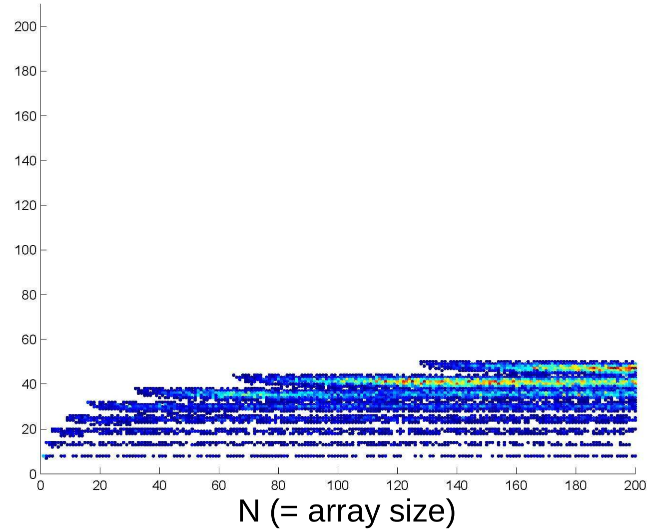
Binary search—empirical analysis



Empirical comparison



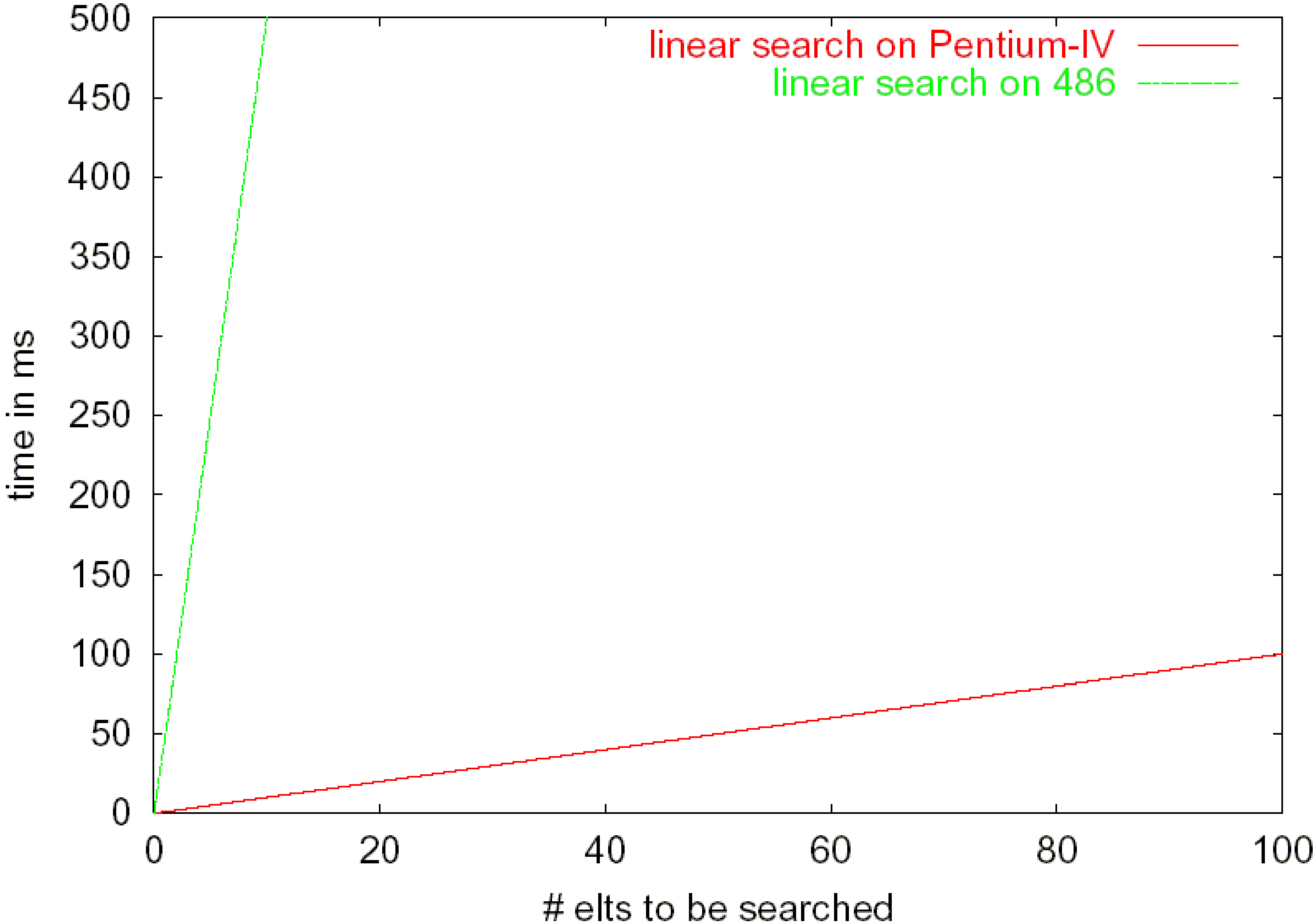
Linear search



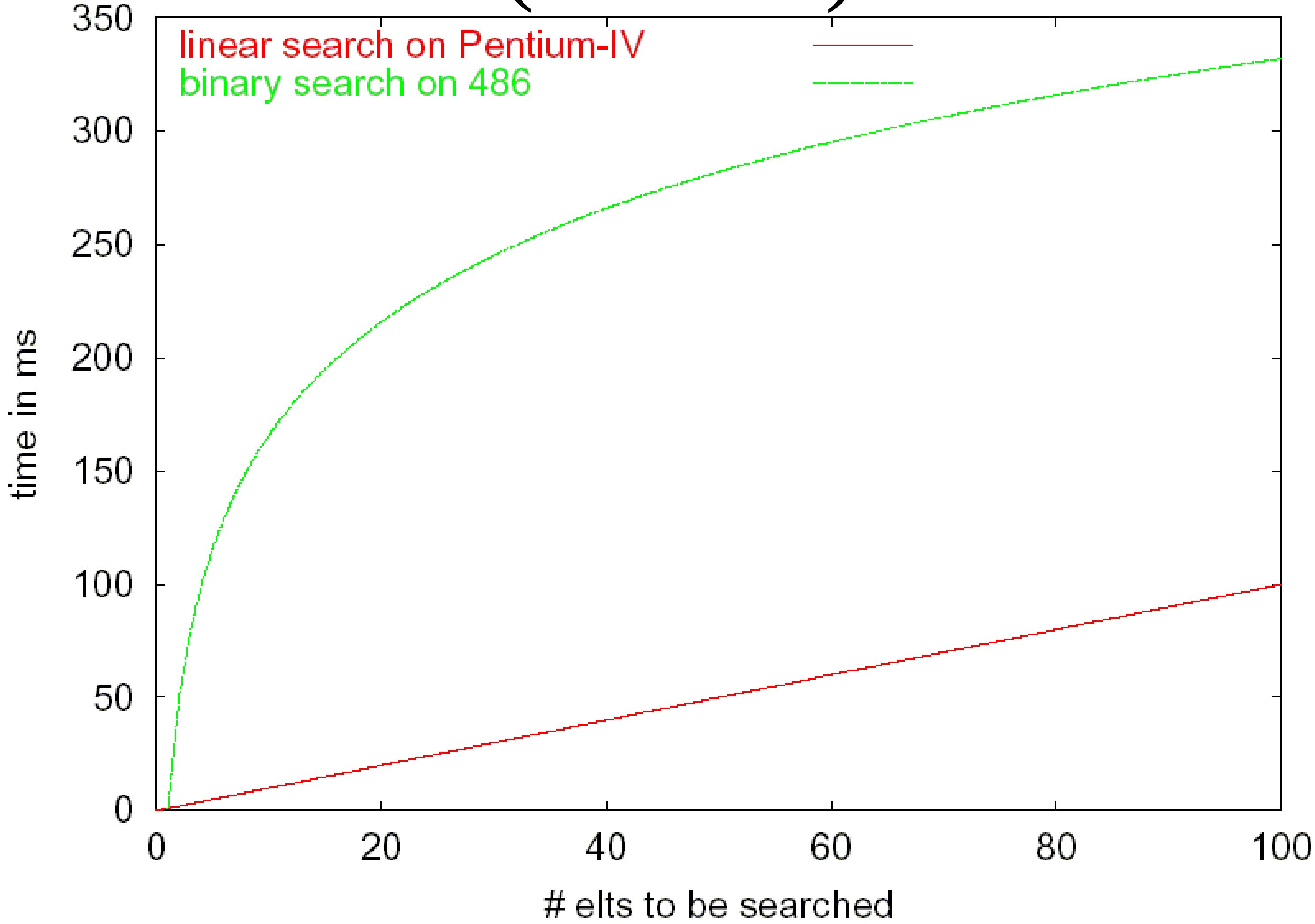
Binary search

Gives additional information

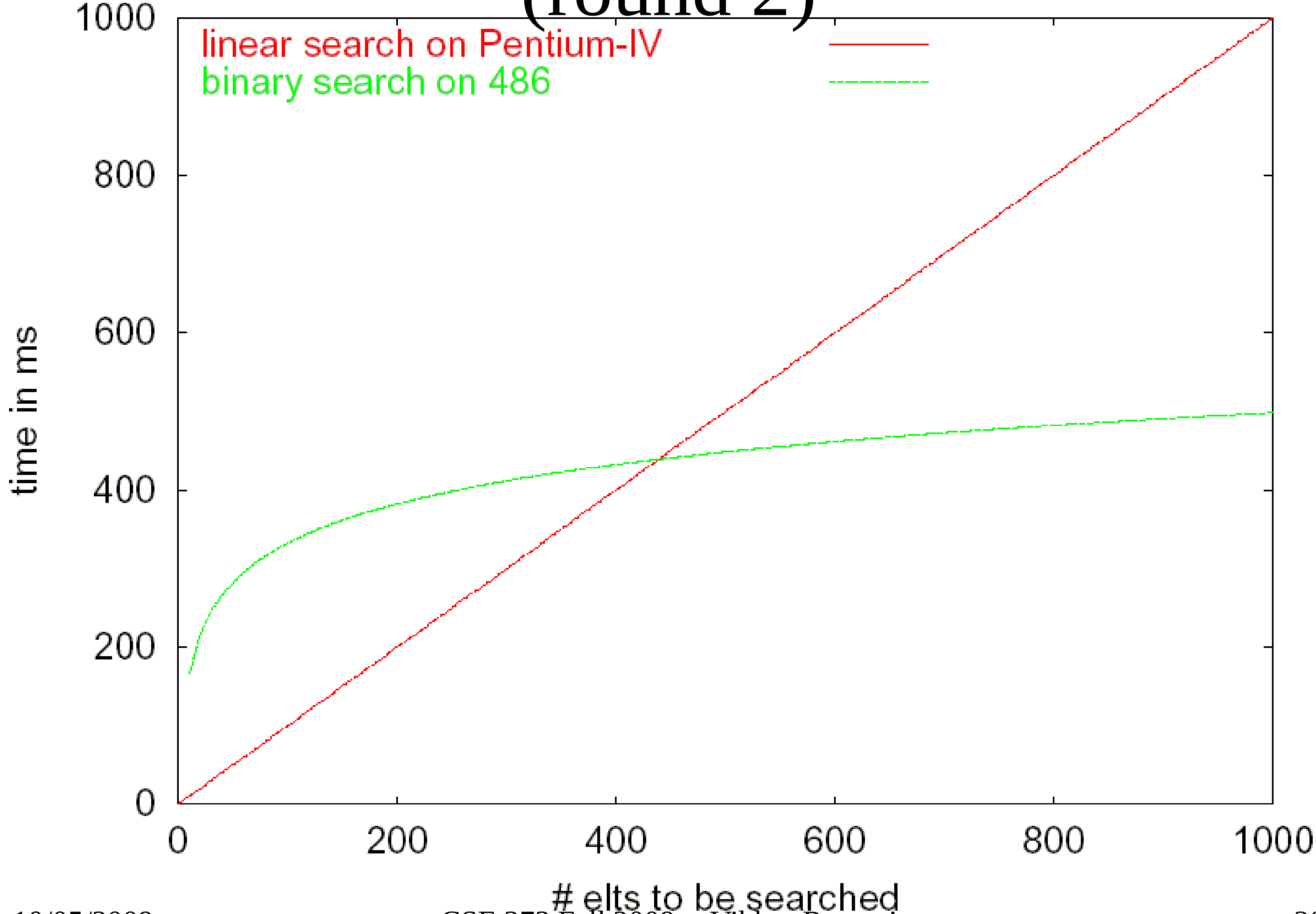
Fast Computer vs. Slow Computer



Fast Computer vs. Smart Programmer (round 1)



Fast Computer vs. Smart Programmer (round 2)



Review: Big O Notation

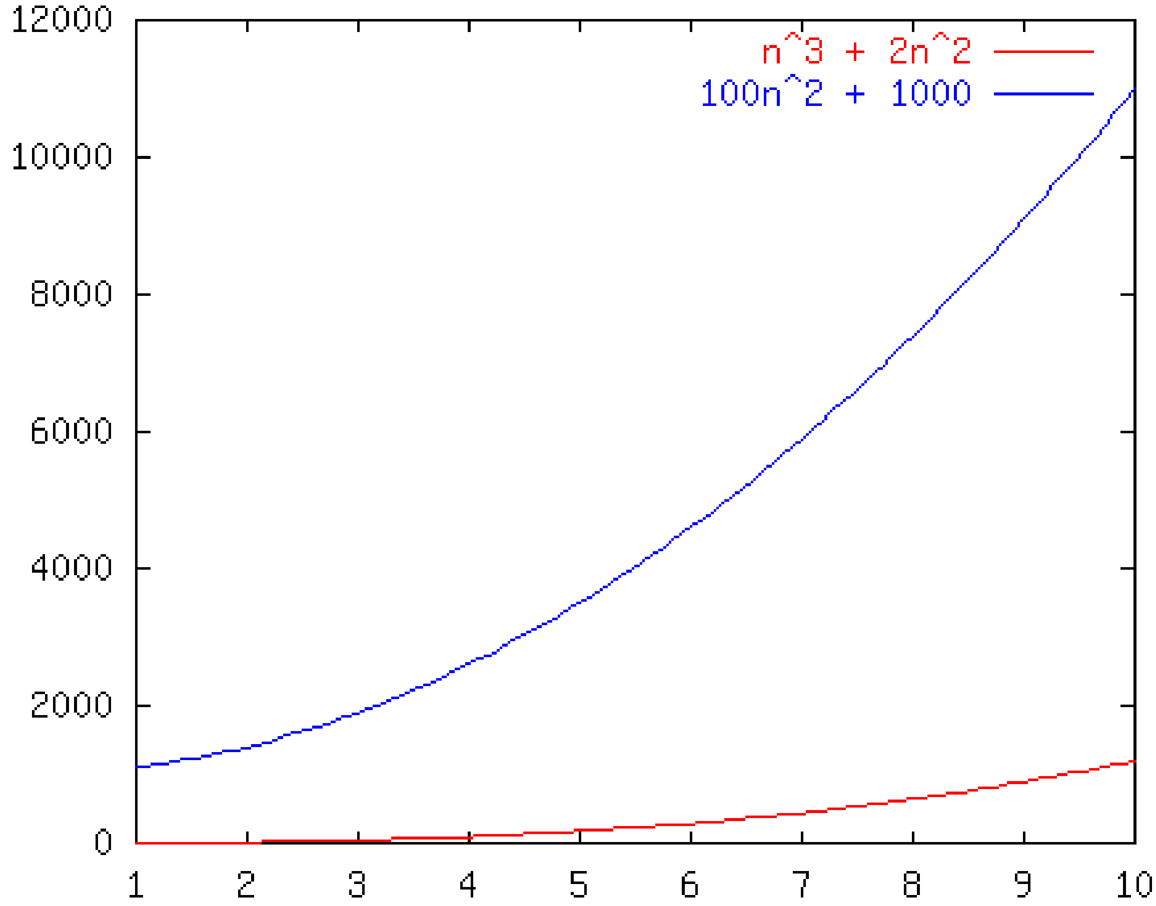
DEFINITION: The Big-O notation

$T(n) = O(f(n))$ if there exist constants c and n' such that: $T(n) \le c\, f(n)$ for all $n \ge n'$

Order Notation: Intuition

$a(n) = n^3 + 2n^2$

$b(n) = 100n^2 + 1000$



Although not yet apparent, as n gets "sufficiently large", $a(n)$ will be "greater than or equal to" $b(n)$

Example

$h(n) \in O(f(n))$ iff there exist positive constants c and n_0 such that:

$$h(n) \leq c\, f(n) \text{ for all } n \geq n_0$$

Example:

$$100n^2 + 1000 \leq 1/2\,(n^3 + 2n^2) \text{ for all } n \geq 198$$

So $b(n) \in O(a(n))$

Asymptotic Analysis

- Intuitively, to find the asymptotic runtime, throw away the constants and low-order terms
  - $a(n) = n^3 + 2n^2 = O(n^3)$
  - $b(n) = 100n^2 + 1000 = O(n^2)$

$$T_{worst}^{LS}(n) = 3n + 3 \in O(n) \qquad T_{worst}^{BS}(n) = 4\lfloor \log_2 n \rfloor + 2 \in O(\log n)$$

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

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 same algorithm
- Comparing worst case search examples:

$$T_{worst}^{LS}(n) = 3n + 3 \qquad \text{vs.} \qquad T_{worst}^{BS}(n) = 4\lfloor \log_2 n \rfloor + 2$$

Asymptotic Analysis

Eliminate low order terms

- $4n + 5 \Rightarrow$
- $0.5\ n \log n + 2n + 7 \Rightarrow$
- $n^3 + 3\ 2^n + 8n \Rightarrow$

Eliminate coefficients

- $4n \Rightarrow$
- $0.5\ n \log n \Rightarrow$
- $3\ 2^n$ =>

Properties of Logs

Basic:

- $A^{\log_A B} = B$
- $\log_A A =$

Independent of base:

- $\log(AB) =$
- $\log(A/B) =$
- $\log(A^B) =$
- $\log((A^B)^C) =$

Properties of Logs

$$\log_A n = \left(\frac{1}{\log_B A}\right)\log_B n$$

Another example

- Eliminate low-order terms
- Eliminate constant coefficients

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

Definition of Order Notation

- Upper bound: $h(n) \in O(f(n))$ Big-O "order"
  - Exist positive constants c and n0 such that
  - $h(n) \leq c\, f(n)$ for all $n \geq n0$
- Lower bound: $h(n) \in \Omega(g(n))$ Omega
  - Exist positive constants c and n0 such that
  - $h(n) \geq c\, g(n)$ for all $n \geq n0$

Definition of Order Notation

- Tight bound: $h(n) \in \theta(f(n))$ Theta
  - When both hold:
  - $h(n) \in O(f(n))$
  - $h(n) \in \Omega(f(n))$

- $O(f(n))$ defines a class (set) of functions

Thanks!

- Any questions:
  - vibhor@cs.washington.edu