

Data Structures

Introduction

Class Overview

- Introduction to many of the basic data structures used in computer software
 - Understand the data structures
 - Analyze the algorithms that use them
 - Know when to apply them
- Practice design and analysis of data structures.
- Practice using these data structures by writing programs.
- Make the transformation from programmer to computer scientist

Goals

- You will understand
 - what the tools are for storing and processing common data types
 - which tools are appropriate for which need
- So that you can
 - make good design choices as a developer, project manager, or system customer
- You will be able to
 - *Justify* your design decisions via formal reasoning
 - *Communicate* ideas about programs clearly and precisely

Goals

“I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships.”

Linus Torvalds, 2006

Data Structures

“Clever” ways to organize information in order to enable efficient computation

- What do we mean by clever?
- What do we mean by efficient?

Introduction to Data Structures

- Computer science can be defined as the study of data, its representation and transformation.
- Once data is stored in the form of bits, it has to be accessed and manipulated many times.
- To do so, there must be inbuilt mechanisms to access and store data.

Introduction to data structure

- To solve complex problems, the basic data types provided by programming languages are not enough.
- Thus, there is a need for "data structures", which gives more "structure" to data.
- These data structures may be a combination or collection of the basic data types with specific properties and operations.

NEED FOR DATA STRUCTURES

To identify a solution for a data processing problem, four things have to be identified.

1. The data elements that are concerned with the problem.
2. The operations that will be performed on these data elements.
3. Methods of representing the data elements in memory to retain the logical relationship between them.
4. The programming language which suits the current requirements.

Example: Calculating area of circle

1. The data elements are radius , pi, and area.
2. The operations performed on the above data elements are multiplication, assignment, etc.
3. They can be stored in memory as bits/bytes and the user can access them in different forms. Example : integer, float, etc.
4. Languages like Pascal , C, C++ can be used to solve the above problem.

The first three points deal mainly with the data elements and their representation.

Thus, a proper data organization is of prime importance to achieve the solution.

Advantages of data structure

- Structured data makes it easier to access and manipulate the information as compared to raw or unstructured data.
- A variety of operations can be performed on structured data.
- Related data can be stored together and in the required format.
- Better algorithms can be used on organized data which improves program efficiency.

Picking the best Data Structure for the job

- The data structure you pick needs to *support* the operations you need
- Ideally it supports the operations you will use most often in an *efficient* manner
- Examples of operations:
 - A **List** with operations **insert** and **delete**
 - A **Stack** with operations **push** and **pop**

Terminology

- **Data**

- A Piece of information
- Collection of numbers . Alphabets and symbols combined to represent information.
- A computer takes raw data as input and after processing of data it produces refined data as output.
- Computer science is therefore study of data

- **Data type**

- term used to describe the information type that can be processed by the computer and which is supported by the programming languages.
- It is also defined as a term, which refers to the kind of data that variables may “hold” in a programming language.
- Example : int , char , float , etc.
- Example : record in Pascal, structures and unions in C. (Derived data types)

Data object

- Data object refers to a set of elements (D) which may be finite or infinite.
- If the set is infinite, we have to devise some mechanisms to represent that set in memory, since available memory is limited.
- Example : A set of integer number is infinite.
- $D = \{ 0, 1, 2, 3, \dots \}$
- A set of alphabets is finite.
- $D = \{ 'a', 'b', \dots, 'z' \}$

Data Structure

- A data structure consists of data objects, their properties and the set of legal operations which may be applied to the elements of the data object.
- ***Definition:***
- A data structure is a set of domains D , a designated domain $d \in D$, a set of functions F and a set of axioms A . The triple (D, F, A) denotes the data structure.
- This definition is also called the Abstract Data Type.
- D – denotes the data objects
- F – denotes the set of operations that can be carried out on the data objects.
- A – describes the properties and rules of the operations.

ABSTRACT / LOGICAL DATA STRUCTURE

- The data structure defined above (ADT) is called the logical data structure because it simply specifies the “logical properties” of the data type. Hence, it is just a mathematical or abstract concept, which defines the data type.
- Once the ADT has been defined the next stage is the “Implementation stage” where the ADT is “implemented by hardware or software methods”. The abstract data type is implemented by using some physical data structures.

PHYSICAL DATA STRUCTURE

- When the ADT has to be implemented, thought has to be given to how the data objects have to be represented in computer memory so that their logical relationship is maintained and operations can be performed on them. This is the physical data structure.

Terminology

- Abstract Data Type (ADT)
 - Mathematical description of an object with set of operations on the object. Useful building block.
- Algorithm
 - A high level, language independent, description of a step-by-step process
- Data structure
 - A specific family of algorithms for implementing an abstract data type.
- Implementation of data structure
 - A specific implementation in a specific language

Terminology examples

- A stack is an *abstract data type* supporting push, pop and isEmpty operations
- A stack *data structure* could use an array, a linked list, or anything that can hold data
- One stack *implementation* is java.util.Stack; another is java.util.LinkedList

Describing the ADT for list / Arrays

**Sahni's approach
to representing
conceptual ADT**

AbstractDataType *LinearList*

{

instances

ordered finite collections of zero or more elements

operations

isEmpty(): return **true** iff the list is empty, **false** otherwise

size(): return the list size (i.e., number of elements in the list)

get(index): return the **index**th element of the list

indexOf(x): return the index of the first occurrence of **x** in
the list, return **-1** if **x** is not in the list

remove(index): remove and return the **index**th element,
elements with higher index have their index reduced by **1**

add(theIndex, x): insert x as the **index**th element, elements
with **theIndex** \geq **index** have their index increased by **1**

output(): output the list elements from left to right

}

Primitive operation on an array

- **Create** : Creating a new array : This needs the information regarding the type of data objects and the number of data objects to be stored.
- **Store(array, index, value)** : Store a value at particular position : This needs the name of the array, the position in the array and the value to be stored at that position.
- **Retrieve (array, index)** : Retrieve a value at a particular position : This operation requires the array name and an index and it returns the corresponding value.

Primitive and non-Primitive data structure

- **Primitive data structures** are the original data structures for a variety of machines and are available on most computers as inbuilt data types. These are the data structures that are directly operated upon by machine level instructions.

Example : Integers, Floating-point numbers, characters, and pointer information.

- **Non-Primitive data structures** are the user defined data structures which define a set of derived elements. User can select the number of memory locations needed to store particular data. Further non-primitive is divided into linear and non-linear data structure.

Example : Arrays, lists, stacks, queues, trees and graphs.

Linear and non-Linear data structure

- **Linear Data Structure** : consists of elements which are ordered i.e. in a line. The elements form a sequence such that there is a first element, second, and last element. Thus, the elements of a linear data structure have a one-one relationship.

Examples : array, list.

- **Non-Linear Data Structure** : Contains elements which exhibit a one-to-many relationship. Such data structures are useful in representing hierarchical information or complex relationship among elements.

Examples : Trees, graphs, generalized linked list.

Static and dynamic data structure

- A data structure is referred as static data structure if it is created before program execution begins (also called as during compilation time).

Example : arrays

- A data structure which is created at run time, is called as dynamic data structure. The variables of this type are not always referenced by a user defined names. These are accessed indirectly using their addresses through pointers.

Example : Linked list is a dynamic data structure when realized using dynamic memory management and pointers.

Persistent and Ephemeral data structure

- A data structure that supports operations on the most recent versions as well as previous version is termed as persistent data structure.
- An ephemeral data structure is the one which supports operations only on the most recent version.

Data Structures

Asymptotic Analysis

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:**

$$\text{sum}(v, 0) = 0. \checkmark$$

- **Inductive Hypothesis (n=k):**

Assume $\text{sum}(v, k)$ correctly returns sum of first k elements of v , i.e. $v[0] + v[1] + \dots + v[k-1] + v[k]$

- **Inductive Step (n=k+1):**

$\text{sum}(v, n)$ returns

$$v[k] + \text{sum}(v, k-1) = (\text{by inductive hyp.})$$

$$v[k] + (v[0] + v[1] + \dots + v[k-1]) =$$

$$v[0] + v[1] + \dots + v[k-1] + v[k] \quad \checkmark$$

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
Consecutive statements
Conditionals
Loops
Function calls
Recursive functions

Constant time
Sum of times
Larger branch plus test
Sum of iterations
Cost of function body
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

Data Structures

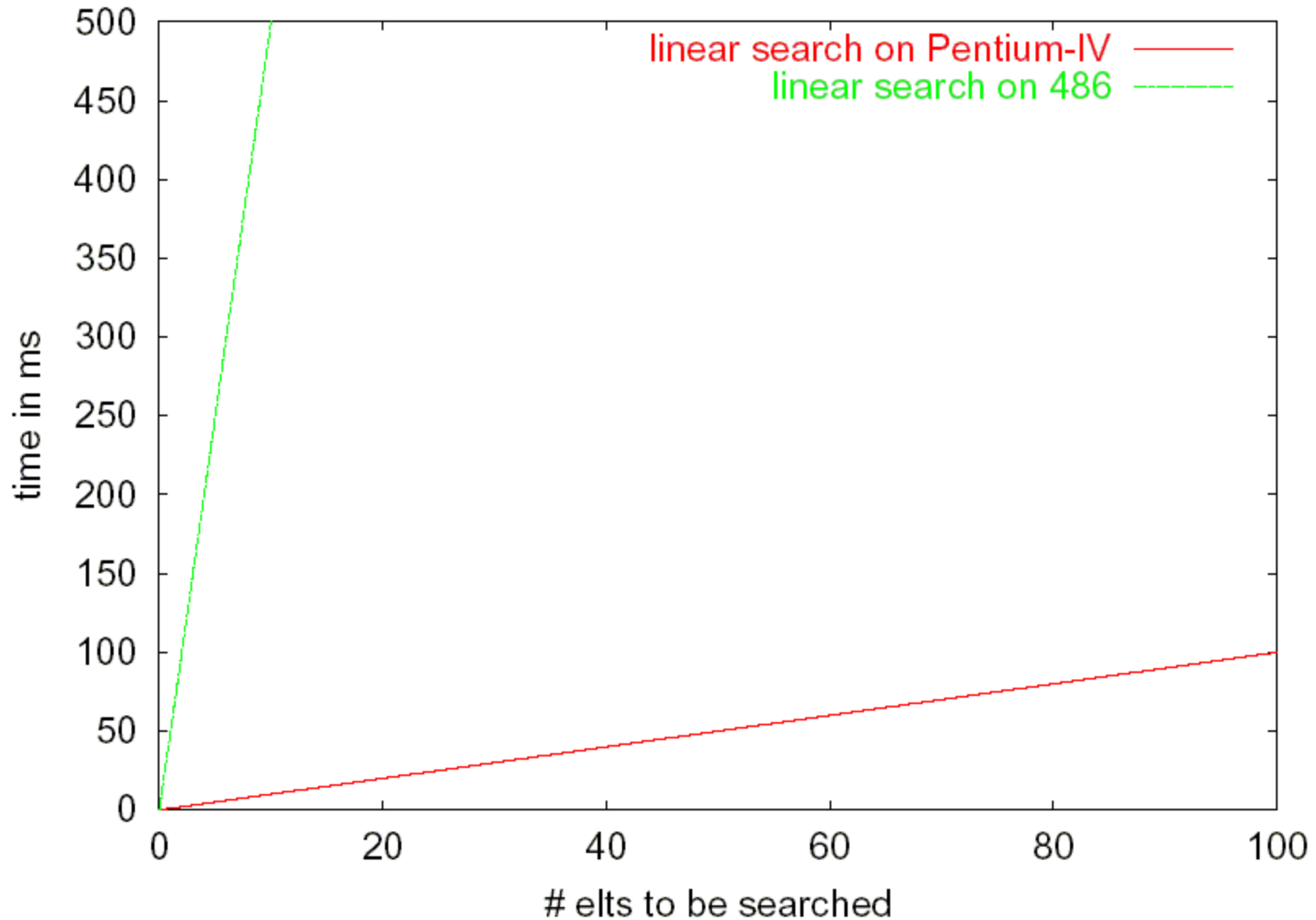
Asymptotic Analysis

Linear Search vs Binary Search

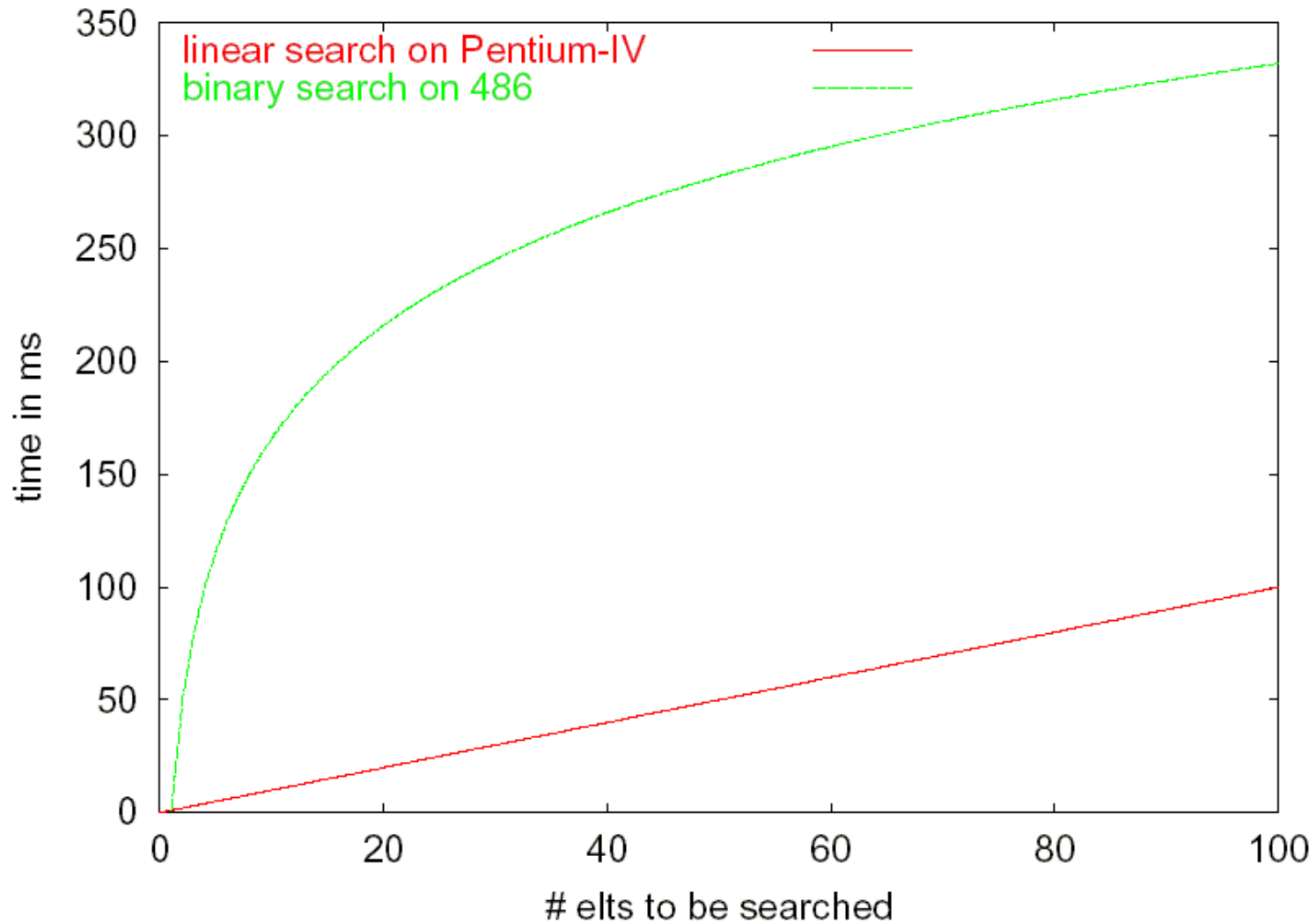
	Linear Search	Binary Search
Best Case	4 at [0]	4 at [middle]
Worst Case	$3n+2$	$4 \log n + 4$

*So ... which algorithm is better?
What tradeoffs can you make?*

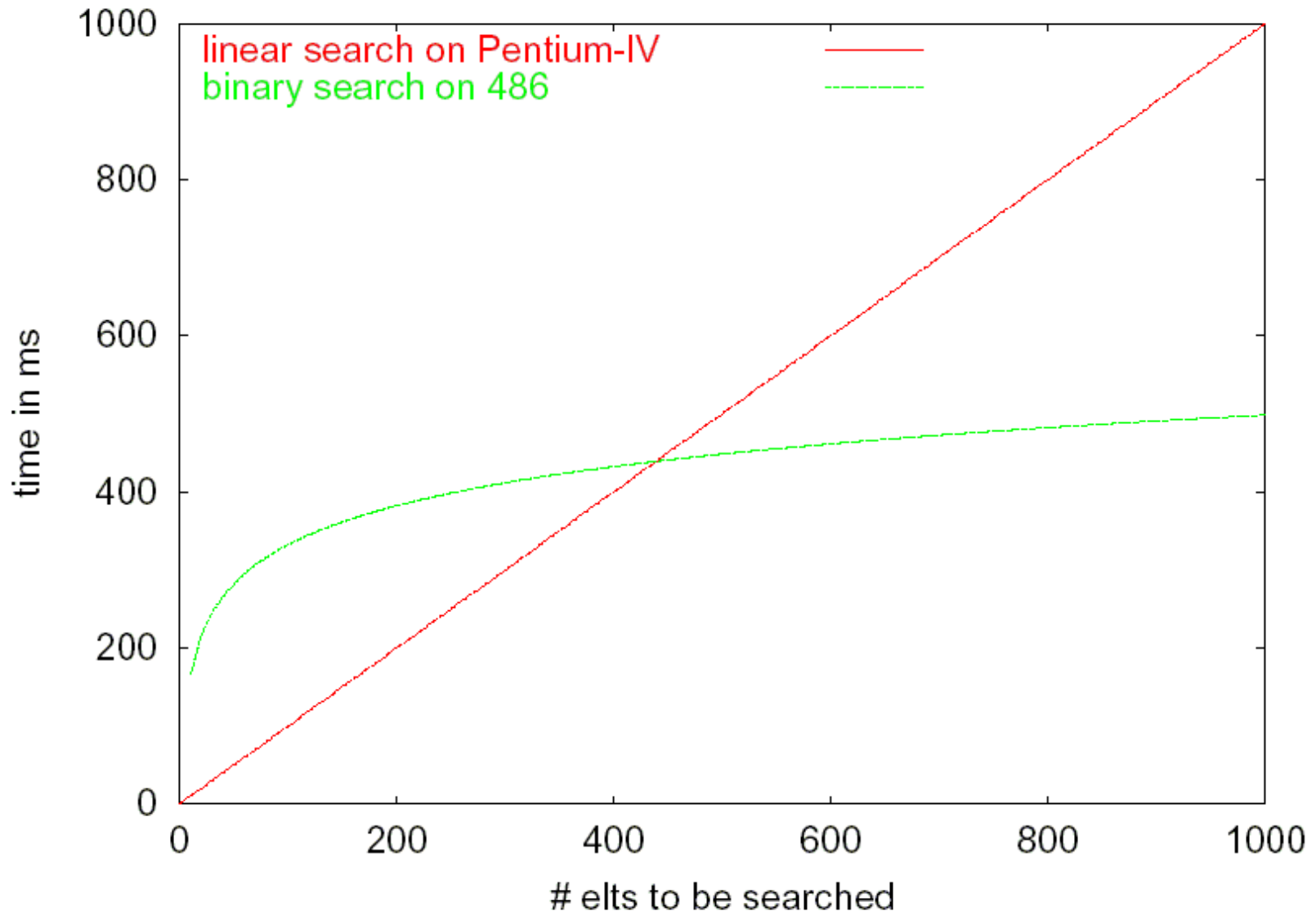
Fast Computer vs. Slow Computer



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 an 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
 - $4n \Rightarrow$
 - $0.5 n \log n \Rightarrow$
 - $n \log n^2 \Rightarrow$

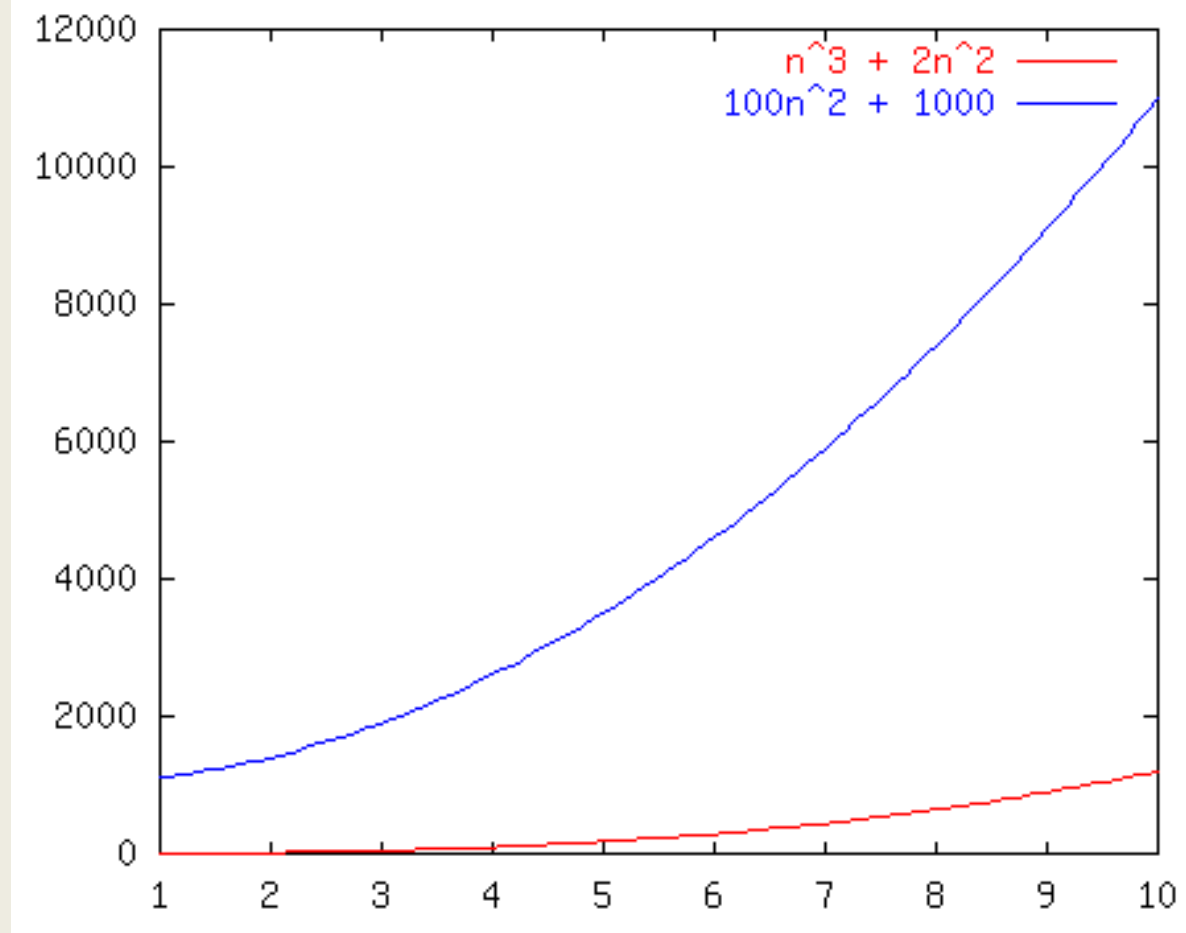
Properties of Logs

- $\log AB = \log A + \log B$
- Proof: $A = 2^{\log_2 A}, B = 2^{\log_2 B}$
 $AB = 2^{\log_2 A} \cdot 2^{\log_2 B} = 2^{(\log_2 A + \log_2 B)}$
 $\therefore \log AB = \log A + \log B$
- Similarly:
 - $\log(A/B) = \log A - \log B$
 - $\log(A^B) = B \log A$
- Any log is equivalent to log-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 positive constants c and n' such that
$$T(n) \leq c f(n) \quad \text{for all } n \geq n'$$
- Lower bound: $T(n) = \Omega(g(n))$ Omega
Exist positive constants c and n' such that
$$T(n) \geq c g(n) \quad \text{for all } n \geq n'$$
- Tight bound: $T(n) = \theta(f(n))$ Theta
When both hold:
$$T(n) = O(f(n))$$
$$T(n) = \Omega(f(n))$$

Definition of Order Notation

$O(f(n))$: a set or class of functions

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

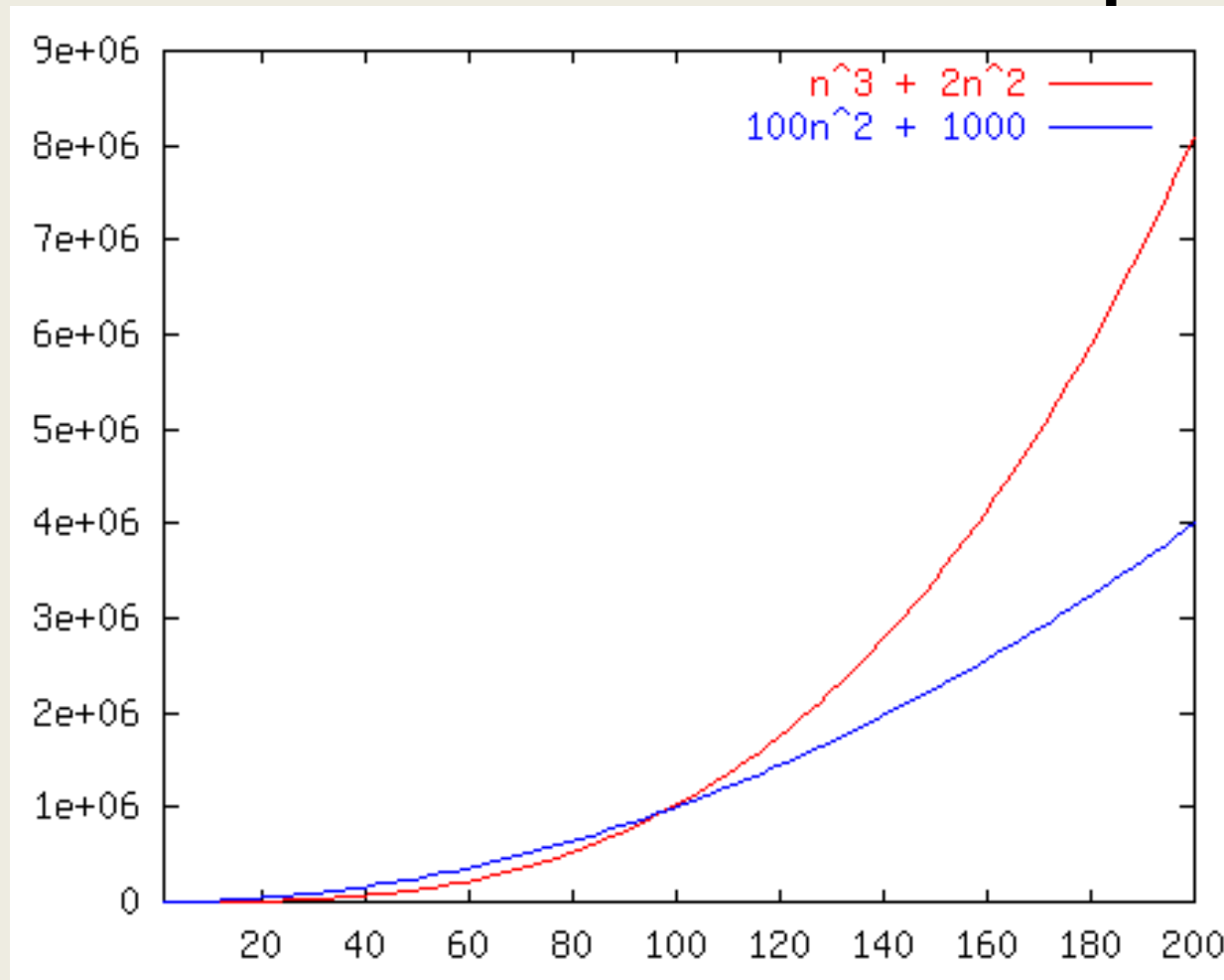
$$g(n) \leq c f(n) \text{ for all } n \geq n_0$$

Example:

$$100n^2 + 1000 \leq 5 (n^3 + 2n^2) \text{ for all } n \geq 19$$

$$\text{So } g(n) \in O(f(n))$$

Order Notation: Example



$$100n^2 + 1000 \leq 5(n^3 + 2n^2) \text{ for all } n \geq 19$$

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

Some Notes on Notation

- Sometimes you'll see

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


- This is equivalent to

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

- What about the reverse?

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

Big-O: Common Names

- 
- constant: $O(1)$
 - logarithmic: $O(\log n)$ ($\log_k n, \log n^2 \in O(\log n)$)
 - linear: $O(n)$
 - log-linear: $O(n \log n)$
 - 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)

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

- $g(n) \in o(f(n))$ iff

There exists a n_0 such that $g(n) < c f(n)$ for all c and $n \geq n_0$

Equivalent to: $\lim_{n \rightarrow \infty} g(n)/f(n) = 0$

- $g(n) \in \Omega(f(n))$ iff

There exist c and n_0 such that $g(n) \geq c f(n)$ for all $n \geq n_0$

- $g(n) \in \omega(f(n))$ iff

There exists a n_0 such that $g(n) > c f(n)$ for all c and $n \geq n_0$

Equivalent to: $\lim_{n \rightarrow \infty} g(n)/f(n) = \infty$

- $g(n) \in \Theta(f(n))$ iff

$g(n) \in O(f(n))$ and $g(n) \in \Omega(f(n))$

Big-Omega et al. Intuitively

Asymptotic Notation	Mathematics Relation
O	\leq
Ω	\geq
θ	$=$
o	$<$
ω	$>$

Pros and Cons of Asymptotic Analysis

Perspective: 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 orthogonal axes:

– Bound Flavor

- Upper bound (O , o)
- Lower bound (Ω , ω)
- Asymptotically tight (θ)

– Analysis Case

- Worst Case (Adversary)
- Average Case
- Best Case
- Amortized

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

- Eliminate low-order terms

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

$$\rightarrow 16n^3\log_8(10n^2)$$

$$\rightarrow n^3\log_8(10n^2)$$

$$\rightarrow n^3(\log_8(10) + \log_8(n^2))$$

$$\rightarrow n^3\log_8(10) + n^3\log_8(n^2)$$

$$\rightarrow n^3\log_8(n^2)$$

$$\rightarrow 2n^3\log_8(n)$$

$$\rightarrow n^3\log_8(n)$$

$$\rightarrow n^3\log_8(2)\log(n)$$

$$\rightarrow n^3\log(n)/3$$

$$\rightarrow n^3\log(n)$$

- Eliminate constant coefficients