CMSC 341
Lecture 5 Asymptotic Analysis

Based on slides from Prof; Gibson, previous iterations of CMSC 341, and textbook

Today's Topics

- Review
  - Mathematical properties
  - Proof by induction
- Program complexity
  - Growth functions
- Big O notation

Mathematical Properties

Why Review Mathematical Properties?

- You will be solving complex problems
  - That use division and power

- These mathematical properties will help you solve these problems more quickly
  - Exponents
  - Logarithms
  - Summations
  - Mathematical Series

Exponents

- Shorthand for multiplying a number by itself
  - Several times
- Used in identifying sizes of memory
- Help to determine the most efficient way to write a program

Exponent Identities

$x^a x^b =$

$x^a y^a =$

$(x^a)^b =$

$x^{(a-b)} =$

$x^{(-a)} =$

$x^{(a/b)} =$

Exponent Identities

$$x^a x^b = x^{(a+b)}$$

$$x^a y^a = (xy)^a$$

$$(x^a)^b = x^{(ab)}$$

$$x^{(a-b)} = (x^a)/(x^b)$$

$$x^{(-a)} = 1/(x^a)$$

$$x^{(a/b)} = (x^a)^{\frac{1}{b}} = \sqrt[b]{x^a}$$

Logarithms

- **ALWAYS** base 2 in Computer Science
  - Unless stated otherwise

- Used for:
  - Conversion between numbering systems
  - Determining the mathematical power needed

- Definition:
  - $n = \log_a x$ if and only if $a^n = x$

Logarithm Identities

$\log_b(1) =$

$\log_b(b) =$

$\log_b(x*y) =$

$\log_b(x/y) =$

$\log_b(x^n) =$

$\log_b(x) =$

$=$

Logarithm Identities

$\log_b(1) = 0$

$\log_b(b) = 1$

$\log_b(x*y) = \log_b(x) + \log_b(y)$

$\log_b(x/y) = \log_b(x) - \log_b(y)$

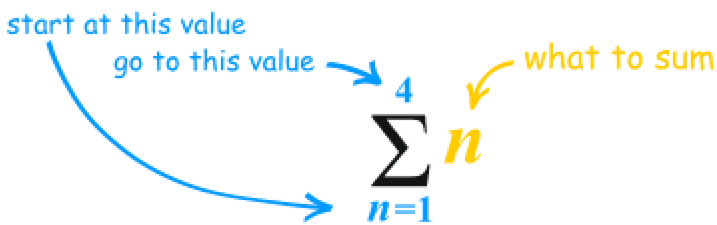
$\log_b(x^n) = n*\log_b(x)$

$\log_b(x) = \log_b(c) * \log_c(x)$

$= \log_c(x) / \log_c(b)$

Summations

- The addition of a sequence of numbers
  - Result is their sum or total



$$\sum_{n=1}^{6} 4n = 4(1)+4(2)+4(3)+4(4)+4(5)+4(6)$$
$$= 4+8+12+16+20+24$$
$$= 84$$

- Can break a function into several summations

$$\sum_{i=1}^{100}(4+3i) = \sum_{i=1}^{100} 4 + \sum_{i=1}^{100} 3i = \sum_{i=1}^{100} 4 + 3\left(\sum_{i=1}^{100} i\right)$$

Proof by Induction

Proof by Induction

- A proof by induction is just like an ordinary proof
  - In which every step must be justified
- However, it employs a neat trick:
  - You can prove a statement about an arbitrary number n by first proving
    - It is true when n is 1 and then
    - Assuming it is true for n=k and
    - Showing it is true for n=k+1

Proof by Induction Example

- Let’s say you want to show that you can climb to the nth floor of a fire escape

- With induction, need to show that:
  - They can climb the ladder up to the fire escape ($n = 0$)
  - They can climb the first flight of stairs ($n = 1$)
- Then we can show that you can climb the stairs from any level of the fire escape ($n = k$) to the next level ($n = k + 1$)

Program Complexity

What is Complexity?

- How many resources will it take to solve a problem of a given size?
  - Time (ms, seconds, minutes, years)
  - Space (kB, MB, GB, TB, PB)

- Expressed as a function of problem size (beyond some minimum size)

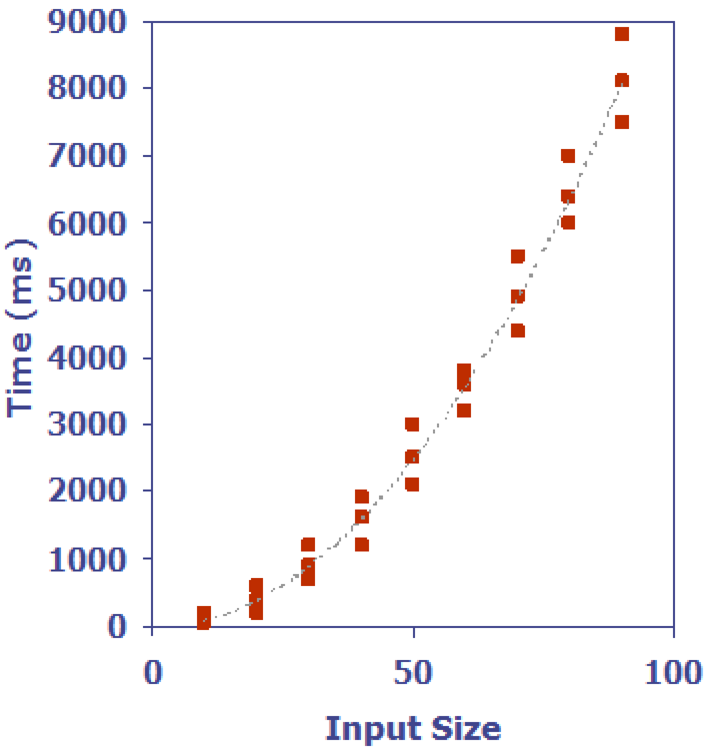
Increasing Complexity

- How do requirements grow as size grows?

- Size of the problem
  - Number of elements to be handled
  - Size of thing to be operated on

Determining Complexity: Experimental

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like clock() to get an accurate measure of the actual running time
- Plot the results



Limitations of Experimental Method

- What are some limitations of this approach?
- Must implement algorithm to be tested
  - May be difficult
- Results may not apply to all possible inputs
  - Only applies to inputs explicitly tested
- Comparing two algorithms is difficult
  - Requires same hardware and software

Determining Complexity: Analysis

- Theoretical analysis solves these problems

- Use a high-level description of the algorithm
  - Instead of an implementation
- Run time is a function of the input size, n
- Take into account all possible inputs
- Evaluation is independent of specific hardware or software
  - Including compiler optimization

Using Asymptotic Analysis

- For an algorithm:
  - With input size **`n`**
  - Define the run time as **`T(n)`**

- Purpose of asymptotic analysis is to examine:
  - The rate of growth of **`T(n)`**
  - As **`n`** grows larger and larger

Growth Functions

Seven Important Functions

- Constant ≈ $\boldsymbol{1}$
- Logarithmic ≈ log $\boldsymbol{n}$
- Linear ≈ $\boldsymbol{n}$
- N-Log-N ≈ $\boldsymbol{n}$ log $\boldsymbol{n}$
- Quadratic ≈ $\boldsymbol{n}^2$
- Cubic ≈ $\boldsymbol{n}^3$
- Exponential ≈ $\mathbf{2}^n$

Constant and Linear

- Constant
  - `T(n) = c`
  - Getting array element at known location
  - Any simple C++ statement (e.g. assignment)

"**c**" is a constant value, like 1

- Linear
  - `T(n) = cn   [+ any lower order terms]`
  - Finding particular element in array of size n
    - Sequential search
  - Trying on all of your n shirts

Quadratic and Polynomial

- Quadratic
  - **T(n) = cn^2 [+ any lower order terms]**
  - Sorting an array using bubble sort
  - Trying all your n shirts with all your n pants

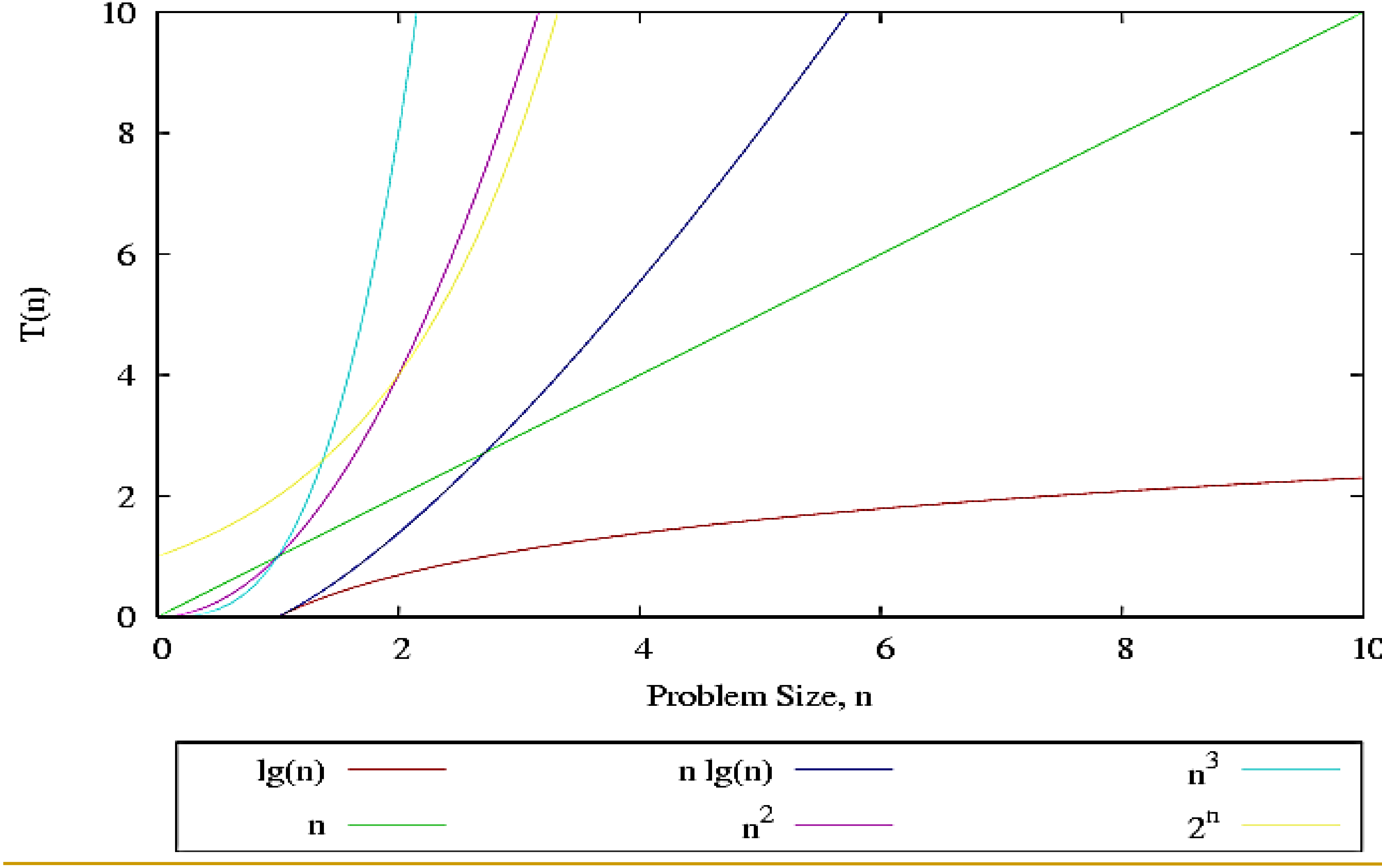
- Polynomial
  - **T(n) = cn^k [+ any lower order terms]**
  - Finding the largest element of a k-dimensional array
  - Looking for maximum substrings in array

Exponential and Logarithmic

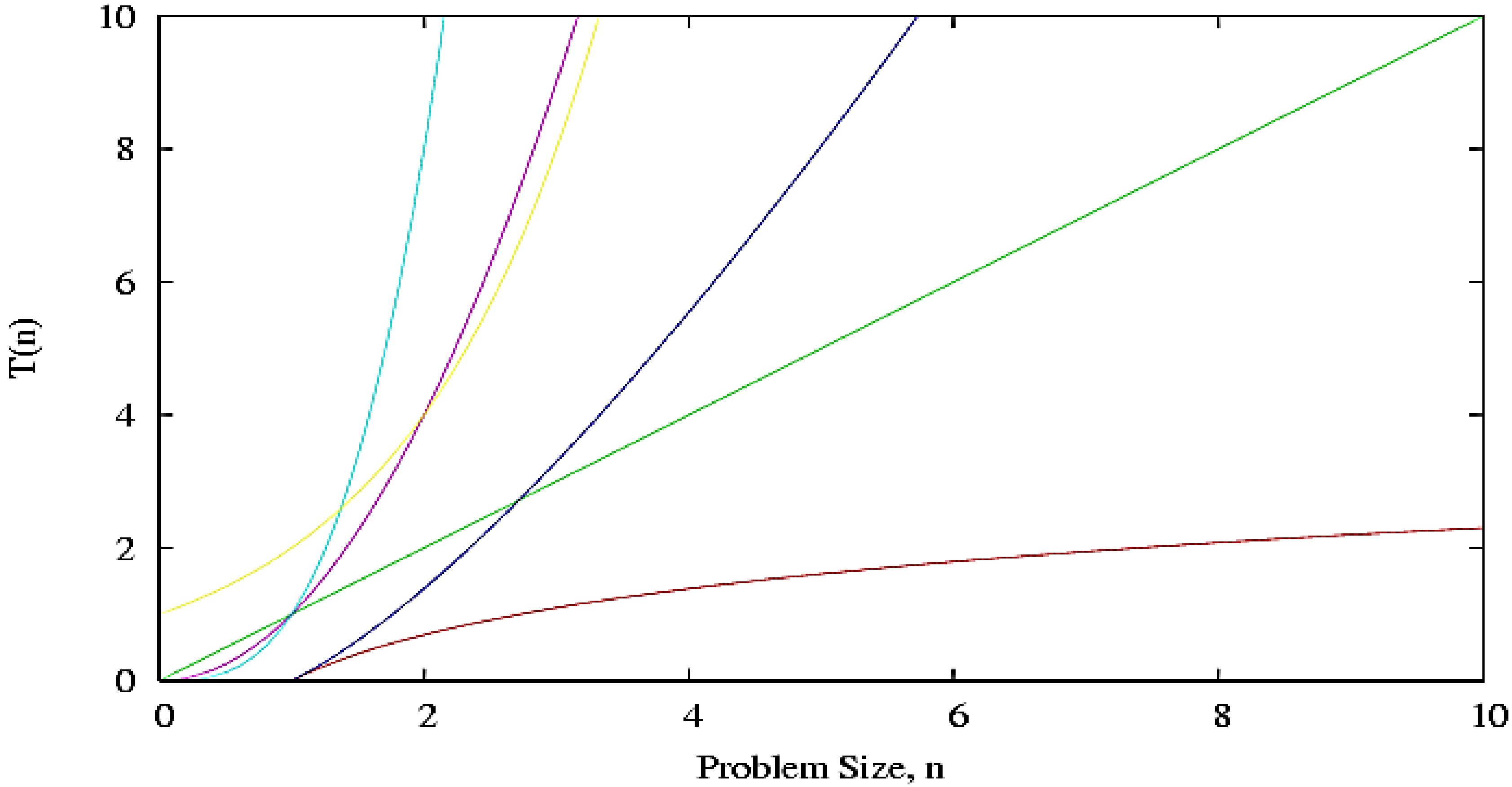
- Exponential
  - **`T(n) = c`^{n} `[ + any lower order terms]`**
  - Constructing all possible orders of array elements
  - Towers of Hanoi (2^n)
  - Recursively calculating nth Fibonacci number (2^n)

- Logarithmic
  - **`T(n) = lg n [ + any lower order terms]`**
  - Finding a particular array element (binary search)
  - Algorithms that continually divide a problem in half

Graph of Growth Functions

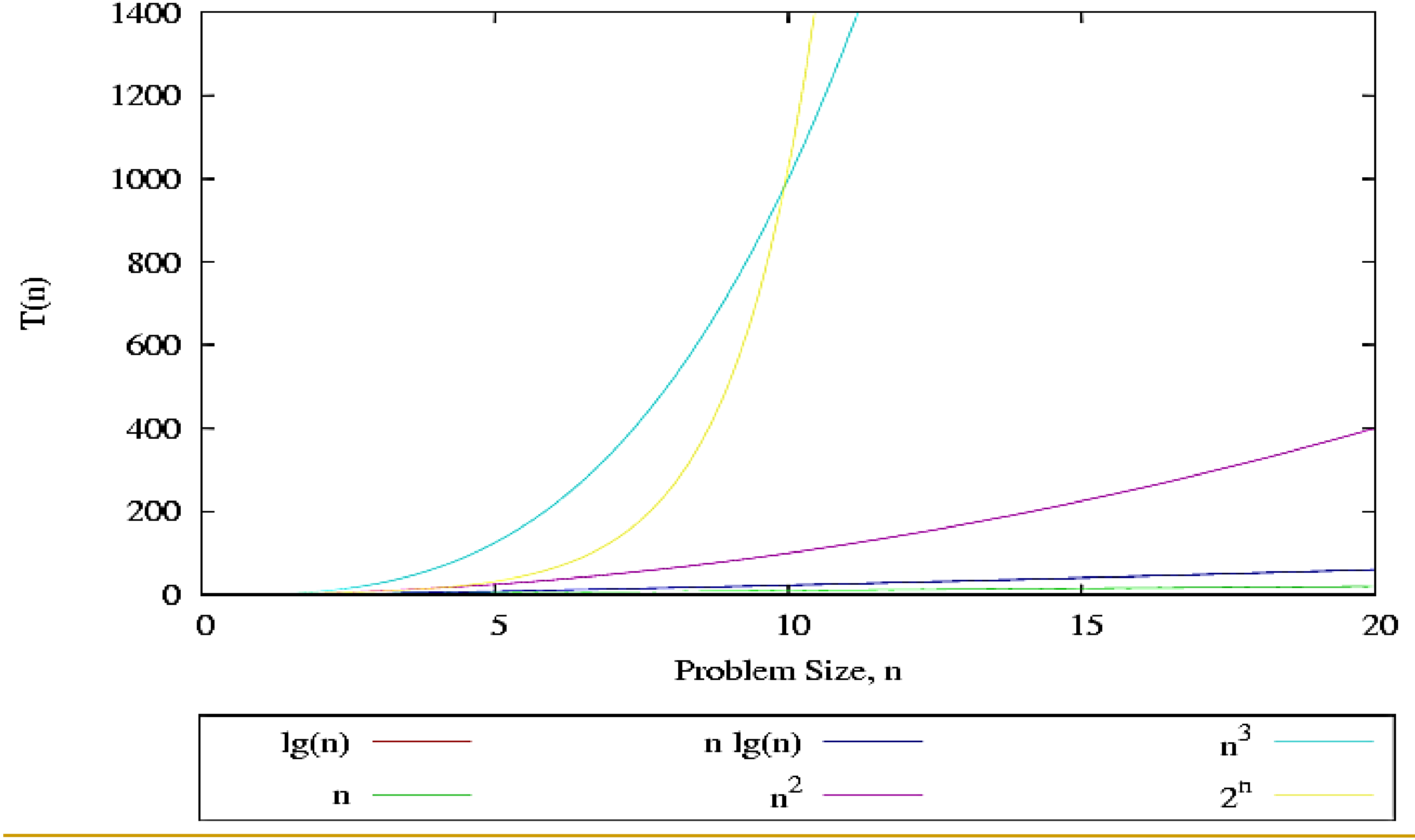


Graph of Growth Functions



logarithmic	n-log-n	cubic
linear	quadratic	exponential

Expanded Growth Functions Graph



Asymptotic Analysis

Simplification

- We are only interested in the growth rate as an "order of magnitude"
  - As the problem grows really, really, really large

- We are not concerned with the fine details
  - Constant multipliers are dropped
    - If **T(n) = c*2**^{n}, we reduce it to **T(n) = 2**^{n}
  - Lower order terms are dropped
    - If **T(n) = n**^{4} **+ n**^{2}, we reduce it to **T(n) = n**^{4}

Three Cases of Analysis

- Best case
  - When input data minimizes the run time
    - An array that needs to be sorted is already in order
- Average case
  - The “run time efficiency” over all possible inputs
- Worst case
  - When input data maximizes the run time
    - Most adversarial data possible

Analysis Example: Mileage

- How much gas does it take to go 20 miles?

- Best case
  - Straight downhill, wind at your back
- Average case
  - "Average" terrain
- Worst case
  - Winding uphill gravel road, inclement weather

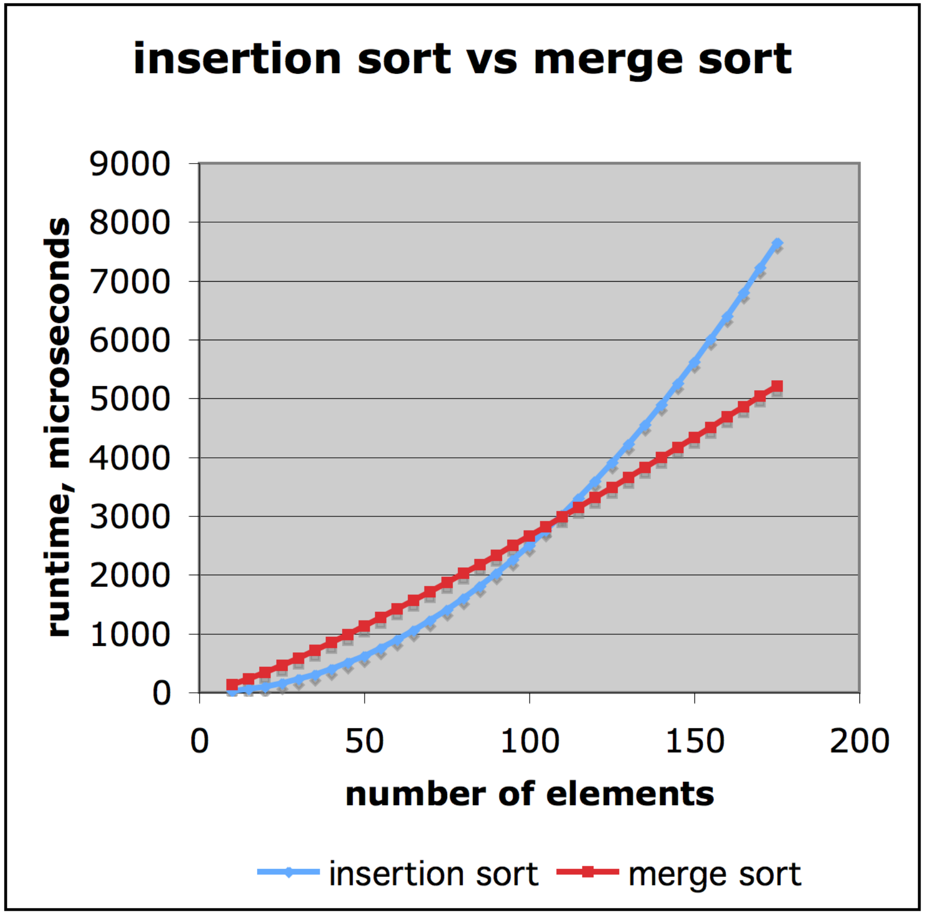
Analysis Example: Sequential Search

- Consider sequential search on an unsorted array of length n, what is the time complexity?
- Best case
- Worst case
- Average case

Comparison of Two Algorithms

- Insertion sort:
  - **(n²)/4**
- Merge sort:
  - **2*n*lg n**

- **n = 1,000,000**
- Million ops per second
  - Merge takes 40 secs
  - Insert takes 70 **hours**



Source: Matt Stallmann, Goodrich and Tamassia slides

Big O Notation

What is Big O Notation?

- Big O notation has a special meaning in Computer Science
  - Used to describe the complexity (or performance) of an algorithm

- Big O describes an upper-limit bound
  - Big Omega (Ω) describes a lower-limit bound
  - Big Theta (Θ) is used when the same bound *order* can be used to describe an upper and lower bound

Big O Definition

- We say that $f(n)$ is $O(g(n))$ if
  - There is a real constant $c > 0$
  - And an integer constant $n_0 \geq 1$
- Such that
  - $f(n) \leq c*g(n)$, for $n \geq n_0$
- Let’s do an example
  - Taken from https://youtu.be/ei-A_wy5Yxw

Big O: Example – n^4

- We have $f(n) = 4n^2 + 16n + 2$
- Let's test if $f(n)$ is $O(n^4)$
  - Remember, we want to see $f(n) \le c*g(n)$, for $n \ge n_0$
- We'll start with $c = 1$

n_0	$4n^2 + 16n + 2$	$\le$	$c*n^4$
0			
1			
2			
3			
4			

Big O: Example – n^4

- We have $f(n) = 4n^2 + 16n + 2$
- Let's test if $f(n)$ is $O(n^4)$
  - Remember, we want to see $f(n) \leq c*g(n)$, for $n \geq n_0$
- We'll start with $c = 1$

n_0	$4n^2 + 16n + 2$	≤	$c*n^4$
0	2	>	0
1	22	>	1
2	50	>	16
3	86	>	81
4	130	<	256

Big O: Example

- So we can say that
  - $f(n) = 4n^2 + 16n + 2$ is $O(n^4)$
- Big O is an upper bound
  - The worst the algorithm might perform
- Does n^4 seem high to you?

Big O: Example – n^2

- We have $f(n) = 4n^2 + 16n + 2$
- Let's test if $f(n)$ is $O(n^2)$
  - Remember, we want to see $f(n) \le c*g(n)$, for $n \ge n_0$
- Let's start with $c = 10$

n_0	$4n^2 + 16n + 2$	≤	$c*n^2$
0			
1			
2			
3			

Big O: Example – n^2

- We have $f(n) = 4n^2 + 16n + 2$
- Let's test if $f(n)$ is $O(n^2)$
  - Remember, we want to see $f(n) \le c*g(n)$, for $n \ge n_0$
- Let's start with $c = 10$

n_0	$4n^2 + 16n + 2$	≤	$c*n^2$
0	2	>	0
1	22	>	10
2	50	>	40
3	86	<	90

Big O: Example

- So we can more accurately say that
  - $f(n) = 4n^2 + 16n + 2$ is $O(n^2)$

- Could $f(n) = 4n^2 + 16n + 2$ is $O(n)$ ever be true?
  - Why not?

Big O:
Practice Examples

Big O: Example 1

- Code:

```
a = b;
++sum;
int y = Mystery( 42 );
```

- Complexity:
  - Constant – O(c)

Big O: Example 2

- Code:

```
sum = 0;
for (i = 1; i <= n; i++) {
  sum += n;
}
```

- Complexity:
  - Linear – O(n)

Big O: Example 3

- Code:

```
sum1 = 0;
for (i = 1; i <= n; i++) {
  for (j = 1; j <= n; j++) {
    sum1++;
  }
}
```

- Complexity:
  - Quadratic – $O(n^2)$

Big O: Example 4

- Code:

```
sum2 = 0;
for (i = 1; i <= n; i++) {
   for (j = 1; j <= i; j++) {
      sum2++;
   }
}
```

how many times do we execute this statement?

1 + 2 + 3 + 4 + … + n-2 + n-1 + n

- Complexity:
  - Quadratic – $O(n^2)$

Expressing as a summation

- Can we express this as a summation?
  - Yes!

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

- Does this have a known formula?
  - Yes!
- What does this formula multiply out to?
  - $(n^2 + n) / 2$
  - or $O(n^2)$

Other Geometric Formulas

- $O(n^3)$ $$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$
- $O(n^4)$ $$\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$$
- $O(c^n)$ $$\sum_{i=0}^{n} c^i = \frac{1-c^{(n+1)}}{1-c}, \text{ where } c \neq 1$$

Big O: Example 5

- Code:

```
sum3 = 0;
for (i = 1; i <= n; i++) {
  for (j = 1; j <= i; j++) {
    sum3++; }
}
for (k = 0; k < n; k++) {
  a[ k ] = k;
}
```

- Complexity:
  - Quadratic – $O(n^2)$

Big O: Example 6

- Code:

```
sum4 = 0;
for (k = 1; k <= n; k *= 2)
   for (j = 1; j <= n; j++) {
      sum4++;
}
```

- Complexity:
  - O(n log n)

Big O: More Examples

- Square each element of an N x N matrix
- Printing the first and last row of an N x N matrix
- Finding the smallest element in a sorted array of N integers
- Printing all permutations of N distinct elements

Big Omega (Ω) and Big Theta(Θ)

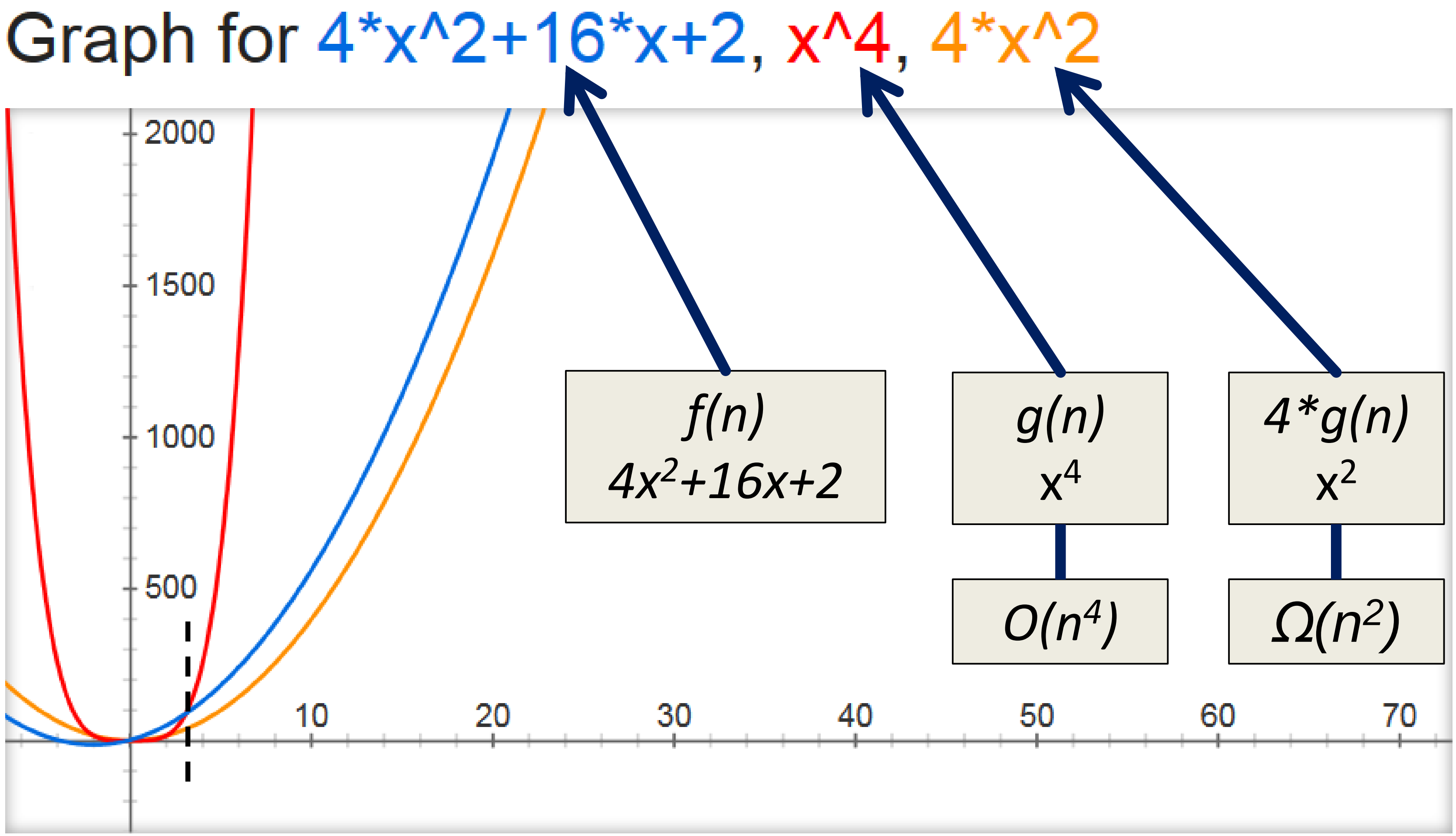
"Big" Notation (words)

- Big O describes an *asymptotic upper bound*
  - The worst possible performance we can expect
- Big Ω describes an *asymptotic lower bound*
  - The best possible performance we can expect
- Big Θ describes an *asymptotically tight bound*
  - The best and worst running times can be expressed with the same equation

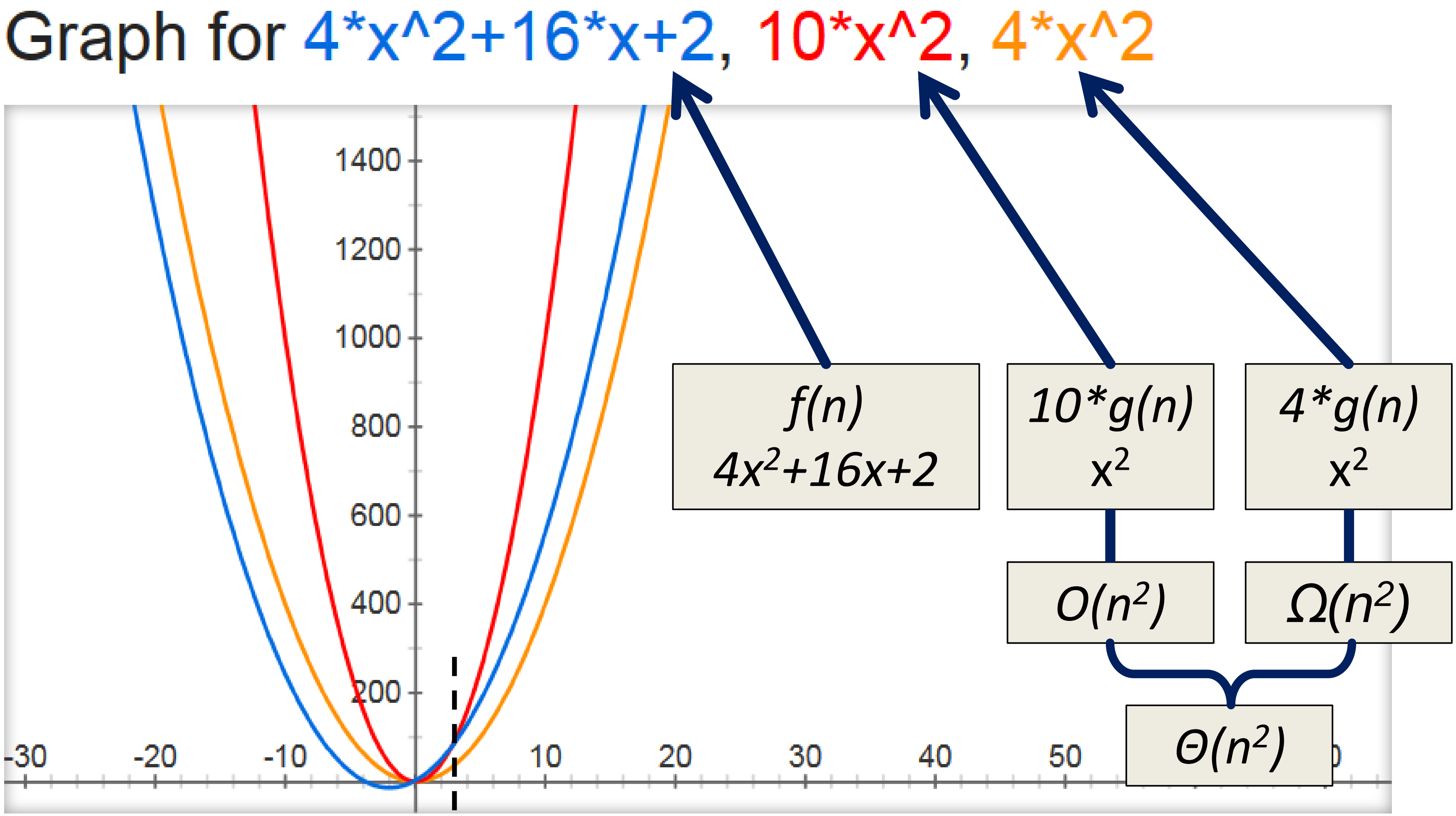
“Big” Notation (equations)

- Big O describes an *asymptotic upper bound*
  - *f(n)* is asymptotically **less than or equal to** *g(n)*
- Big Ω describes an *asymptotic lower bound*
  - *f(n)* is asymptotically **greater than or equal to** *g(n)*
- Big Θ describes an *asymptotically tight bound*
  - *f(n)* is asymptotically **equal to** *g(n)*

Big O and Big Omega Example



Big Theta Example



A Simple Example

- Say we write an algorithm that takes in an array of numbers and returns the highest one
  - What is the absolute fastest it can run?
    - Linear time – $\Omega(n)$
  - What is the absolute slowest it can run?
    - Linear time – $O(n)$
  - Can this algorithm be *tightly* asymptotically bound?
    - YES – so we can also say it's $\Theta(n)$

Proof by Induction

Proof by Induction

- The only way to prove that Big O will work
  - As n becomes larger and larger numbers

- To prove F(n) for any positive integer n
  1. Base case: prove F(1) is true
  2. Hypothesis: Assume F(k) is true for any k >= 1
  3. Inductive: Prove the if F(k) is true, then F(k+1) is true

Induction Example (Step 1)

- Show that for all $n \geq 1$: $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$

1. Base case:
   - $n = 1$
   - (This is our n_0)

$$\sum_{i=1}^{1} i^2 = \frac{1(1+1)(2(1)+1)}{6}$$

$$\sum_{i=1}^{1} i^2 = \frac{1(2)(3)}{6}$$

$$\sum_{i=1}^{1} i^2 = \frac{6}{6}$$

$$\sum_{i=1}^{1} i^2 = 1$$

Induction Example (Step 2)

- Show that for all $n \geq 1$: $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$

2. <u>Hypothesis</u>:
   - Assume that $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$

     holds for any $n \geq 1$

Induction Example (Step 3)

- Show that for all $n \geq 1$: $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$

3. <u>Inductive</u>:
   - Prove that if $F(k)$ is true (assumed), the $F(k+1)$ is also true
   - We've already proved $F(1)$ is true
   - So proving this step will prove $F(2)$ from $F(1)$, and $F(3)$ from $F(2)$, …, and $F(k+1)$ from $F(k)$

Induction Example (Step 3)

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=1}^{k+1} i^2 = \sum_{i=1}^{k} i^2 + (k+1)^2$$

$$\sum_{i=1}^{k+1} i^2 = \frac{k(k+1)(2k+1)}{6} + (k+1)^2$$

$$\sum_{i=1}^{k+1} i^2 = \frac{(k+1)(k(2k+1)+6(k+1))}{6}$$

$$\sum_{i=1}^{k+1} i^2 = \frac{(k+1)(2k^2+7k+6)}{6}$$

$$\sum_{i=1}^{k+1} i^2 = \frac{(k+1)(k+2)(2k+3)}{6}$$

$$\sum_{i=1}^{k+1} i^2 = \frac{(k+1)((k+1)+1)(2(k+1)+1)}{6}$$