CSE 3101: Introduction to the Design and Analysis of Algorithms

Instructor: Suprakash Datta (datta[at]cse.yorku.ca) ext 77875

Lectures: Tues (Tel 0016), 7–10 PM

Office hours (Las/CSEB 3043): Tue 6-7 pm, Wed 4-6 pm or by appointment.

Tutorials: TBA

TA: Xiwen Chen

Textbook: Cormen, Leiserson, Rivest, Stein. Introduction to Algorithms (3nd Edition)

Note: Some slides in this lecture are adopted from Jeff Edmonds' slides.

CSE 3101: Administrivia

Described in more detail on webpage

http://www.cse.yorku.ca/course/3101

<u>Grading:</u>

Quizzes: 2 X 10%

Final: 40%

Midterm (June 25): 20%

HW: 20% (At least 5 assignments)

<u>Notes:</u>

1. All assignments are individual.

Topics: Listed on webpage.

CSE 3101: More administrivia

Plagiarism: Will be dealt with very strictly. Read the detailed policies on the webpage.

Handouts (including solutions): in /cs/course/3101

Grades: will be on *ePost* [you need a cse account for this].

Slides: Will usually be on the web the morning of the class. The slides are for MY convenience and for helping you recollect the material covered. They are not a substitute for, or a comprehensive summary of, the book.

Webpage: All announcements/handouts will be published on the webpage -- check often for updates)

CSE 3101: resources

- We will follow the textbook closely.
- There are more resources than you can possibly read – including books, lecture slides and notes, online texts, video lectures, assignments.
- Jeff Edmonds' (www.cse.yorku.ca/~jeff) textbook has many, many worked examples.
- Andy Mirzaian (www.cse.yorku.ca/~andy) has very good notes and slides for this course
- The downloadable text by Parberry on Problems in Algorithms (http://www.eng.unt.edu/ian/books/free/) is an invaluable resource for testing your understanding

Recommended strategy

- Practice instead of reading.
- Try to get as much as possible from the lectures.
- Try to listen more and write less in class.
- If you need help, get in touch with me early.
- If at all possible, try to come to the class with a fresh mind.
- Keep the big picture in mind. ALWAYS.
- If you like challenging problems, and/or to improve your problem solving ability, try programming contest problems. http://www.cse.yorku.ca/acm

The Big Picture for CSE3101

- The design and analysis of algorithms is a FOUNDATIONAL skill -- needed in almost every field in Computer Science and Engineering.
- Programming and algorithm design go hand in hand.
- Coming up with a solution to a problem is not of much use if you cannot argue that the solution is
  - Correct, and
  - Efficient

Imagine - 0

- You take up a job at a bank. Your group leader defines the problem you need to solve. Your job is to design an algorithm for a financial application that you did not encounter in your classes.

- How do you go about this task?

Designing algorithms – knowledge of paradigms.

Imagine - 1

You want your team to implement your idea – one of them brings you this code and argues that anyone should see that this sorts an array of numbers correctly.

How can you be sure?

```
for j=2 to length(A)
  do key=A[j]
    i=j-1
    while i>0 and A[i]>key
      do A[i+1]=A[i]
        i--
    A[i+1]:=key
```

Correctness proofs – reasoning about algorithms

Imagine - 2

- Two members of your team have designed alternative solutions to the problem you wanted them to solve. Your job is to select the better solution and reward the designer. There are serious consequences for the company as well for the designer.

Efficiency of algorithms – algorithm analysis

Imagine - 3

- Your boss asks you to solve a problem – the best algorithm you can come up with is very slow. He wants to know why you cannot do better.

Intractability : reasoning about problems

Primary Objectives

Previous courses (1020,1030,2011): Given a problem,

1. Figure out an algorithm.
2. Code it, debug, test with “good” inputs.
3. Some idea of running time, asymptotic notation.
4. Study some well known algorithms:
   e.g. QuickSort, Prim(MST), Dijkstra’s algorithm
5. Possibly: some idea of lower bounds.

Primary objectives - continued

3101: Problem-solving, Reasoning about ALGORITHMS

1. Design of algorithms -- Some design paradigms.
   Divide-and-Conquer, Greedy, Dynamic Programming
2. Very simple data structures
   Heaps
3. Correctness proofs.
   Loop invariants
3. Efficiency analysis.

Rigorous

Machine-independent

4. Comparison of algorithms (Better? Best?)

Primary objectives - continued

Reasoning about PROBLEMS:

1. Lower bounds.
   "Is your algorithm the *best possible*?"
   "No comparison-based sorting algorithm can have running time better than θ(n log n)".
2. Intractability: "The problem seems to be hard – is it provably intractable?"
3. Complexity classes.
   "Are there inherently **hard** problems?"
   P vs NP

Secondary objectives

A new way of thinking -- abstracting out the algorithmic problem(s):

-- Extract the algorithmic problem and ignore the "irrelevant" details

-- Focuses your thinking, more efficient problem solving

-- Programming contest problems teach this skill more effectively than exercises in algorithms texts.

Role of mathematics

1. Needed for correctness proofs
   Pre-condition – post-condition framework; similar ideas used in program verification, Computer-aided design.

2. Needed for performance analysis
   Computation of running time

Specific topics

1. (Very) elementary logic.
2. (Occasionally) Elementary calculus.
3. Summation of series.
4. Simple counting techniques.
5. Simple proof techniques: Induction, proof by contradiction
6. Elementary graph theory

Why you should learn algorithms,
Or, Why this is a core course.

1. Fact: Algorithms are always crucial
   Applications: Computational Biology, Genomics
   Data compression
   Indexing and search engines
   Cryptography
   Web servers: placement, load balancing, mirroring
   Optimization (Linear programming, Integer Programming)

"Big data"

2. Fact: Real programmers may not need algorithms……
   but architects do!
3. Much more **important** fact: you must be able to REASON about algorithms designed by you and others. E.g., convincingly argue "my algorithm is correct", "my algorithm is fast", "my algorithm is better than the existing one", "my algorithm is the best possible", "our competitor cannot possibly have a fast algorithm for this problem",…

Some examples

1. Sorting a set of numbers (seen before)
2. Finding minimal spanning trees (seen before)
3. Matrix multiplication – compute $A_1A_2A_3A_4....A_n$ using the fewest number of multiplications

   e.g.: $A_1 = 20 \times 30$, $A_2 = 30 \times 60$, $A_3 = 60 \times 40$,

   $(A_1\ A_2)\ A_3 \Rightarrow 20 \times 30 \times 60 + 20 \times 60 \times 40 = 84000$

   $A_1\ (A_2\ A_3) \Rightarrow 20 \times 30 \times 40 + 30 \times 60 \times 40 = 96000$

4. Traveling Salesman Problem: Find the minimum weight cycle in an weighted undirected graph which visits each vertex exactly once and returns to the starting vertex

   Brute force: find all possible permutations of the vertices and compute cycle costs in each case. Find the maximum. Q: Can we do better?

Pseudocode

- Machine/language independent statements.
- Very simple commands: assignment, equality tests, branch statements, for/while loops, function calls.
- No objects/classes (usually).
- Comments, just like in real programs.
- Should be at a level that can be translated into a program very easily.
- **As precise as programs, without the syntax headaches**
- My notation can vary slightly from the book.

You can use pseudocode, English or a combination.

Reasoning (formally) about algorithms

1. I/O specs: Needed for correctness proofs, performance analysis.

   E.g. for sorting:

   INPUT: A[1..n] - an array of integers

   OUTPUT: a permutation B of A such that

   $B[1] \leq B[2] \leq \ldots. \leq B[n]$

2. CORRECTNESS: The algorithm satisfies the output specs for EVERY valid input.

3. ANALYSIS: Compute the performance of the algorithm, e.g., in terms of running time

Factors affecting algorithm performance

Importance of platform

- Hardware matters (memory hierarchy, processor speed and architecture, network bandwidth, disk speed,…..)
- Assembly language matters
- OS matters
- Programming language matters

Importance of input instance

Some instances are easier (algorithm dependent!)

Analysis of Algorithms

- Measures of efficiency:
  - –Running time
  - –Space used
  - – others, e.g., number of cache misses, disk accesses, network accesses,….
- Efficiency as a function of <u>input size</u> (NOT value!)
  - –Number of data elements (numbers, points)
  - –Number of bits in an input number

    e.g. Find the factors of a number n,

    Determine if an integer n is prime
- Machine Model What machine do we assume? Intel? Motorola? P4? Atom? GPU?

What is a machine-independent model?

- Need a generic model that models (approximately) all machines
- Modern computers are incredibly complex.
- Modeling the memory hierarchy and network connectivity generically is very difficult
- All modern computers are “similar” in that they provide the same basic operations.
- Most general-purpose processors today have at most eight processors. The vast majority have one or two. GPU’s have tens or hundreds.

The RAM model

- Generic abstraction of sequential computers
- RAM assumptions:
  - Instructions (each taking constant time), we usually choose one type of instruction as a **characteristic** operation that is counted:
    - Arithmetic (add, subtract, multiply, etc.)
    - Data movement (assign)
    - Control (branch, subroutine call, return)
    - Comparison
  - Data types – integers, characters, and floats
  - Ignores memory hierarchy, network!

Can we compute the running time on a RAM?

- Do we know the speed of this generic machine?
- If we did, will that say anything about the running time of the same program on a real machine?
- What simplifying assumptions can we make?

Idea: efficiency as a function of input size

- Want to make statements like, “the running time of an algorithm grows linearly with input size”.
- Captures the nature of growth of running times, NOT actual values
- Very useful for studying the behavior of algorithms for LARGE inputs
- Aptly named Asymptotic Analysis

Importance of input representation

Consider the problem of factoring an integer n

Note: Public key cryptosystems depend critically on hardness of factoring – if you have a fast algorithm to factor integers, most e-commerce sites will become insecure!!

Trivial algorithm: Divide by 1,2,…, n/2 (n/2 divisions)

aside: think of an improved algorithm

Representation affects efficiency expression:

Let input size = S.

Unary: 1111…..1 (n times) -- S/2 multiplications (linear)

Binary: $\log_2 n$ bits -- 2^{S-1} multiplications (exponential)

Decimal: $\log_{10} n$ digits -- $10^{S-1}/2$ multiplications (exponential)

A simple example

Q1. Find the max of n numbers (stored in array A)

Formal specs:

INPUT: A[1..n] - an array of integers

OUTPUT: an element m of A such that $A[j] \leq m$, $1 \leq j \leq length(A)$

Find-max (A)

```
1. max ← A[1]
2. for j ← 2 to length(A)
3.    do if (max < A[j])
4.            max ← A[j]
5. return max
```

How many comparisons?

Q2. Can you think of another algorithm? Take a minute….

How many comparisons does it take?

Aside

- How many nodes does a binary tree with n leaves have?

Finding the maximum – contd.

- Proposition: Every full binary tree with n leaves has n-1 internal nodes.

  Proof: done on the board.

- Corollary: Any "tournament algorithm" to find the maximum uses n-1 comparisons, as long it uses each element exactly once in comparisons.

- Later: ***Every*** correct algorithm must use at least n-1 comparisons (this is an example of a lower bound)

Analysis of Find-max

COUNT the number of cycles (**running time)** as a function of the **input size**

```
Find-max (A)
    1. max ← A[1]
    2. for j ← 2 to length(A)
    3.    do if (max < A[j])
    4.            max ← A[j]
    5. return max
```

cost	times
c_1	1
c_2	n
c_3	n-1
c_4	$0 \leq k \leq n-1$
c_5	1

Running time (upper bound): $c_1 + c_5 - c_3 - c_4 + (c_2 + c_3 + c_4)n$

Running time (lower bound): $c_1 + c_5 - c_3 - c_4 + (c_2 + c_3)n$

Q: What are the values of c_i?

Best/Worst/Average Case Analysis

- **Best case**: A[1] is the largest element.
- **Worst case**: elements are sorted in increasing order
- **Average case**: *?* Depends on the input characteristics

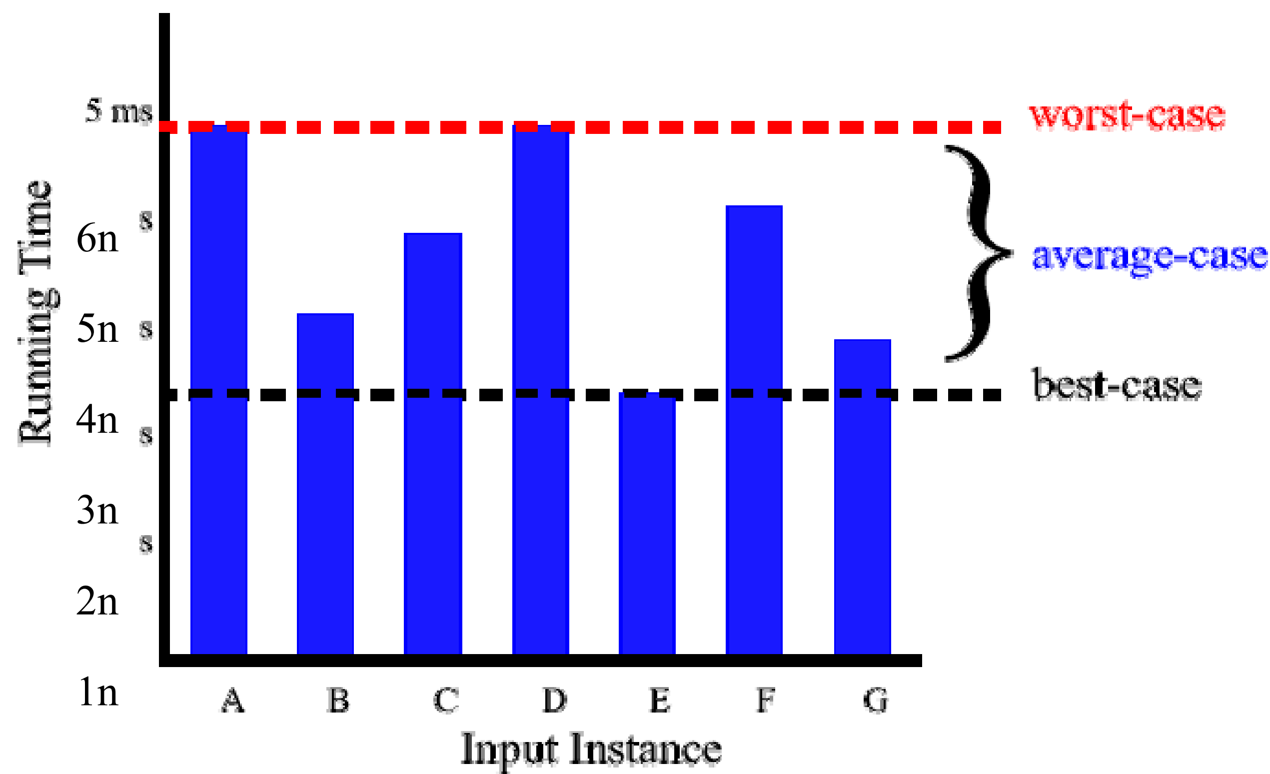
Q: What do we use?

A: Worst case or Average-case is usually used:

- Worst-case is an upper-bound; in certain application domains (e.g., air traffic control, surgery) knowing the **worst-case** time complexity is of crucial importance
- Finding the **average case** can be very difficult; needs knowledge of input distribution.
- Best-case is not very useful.

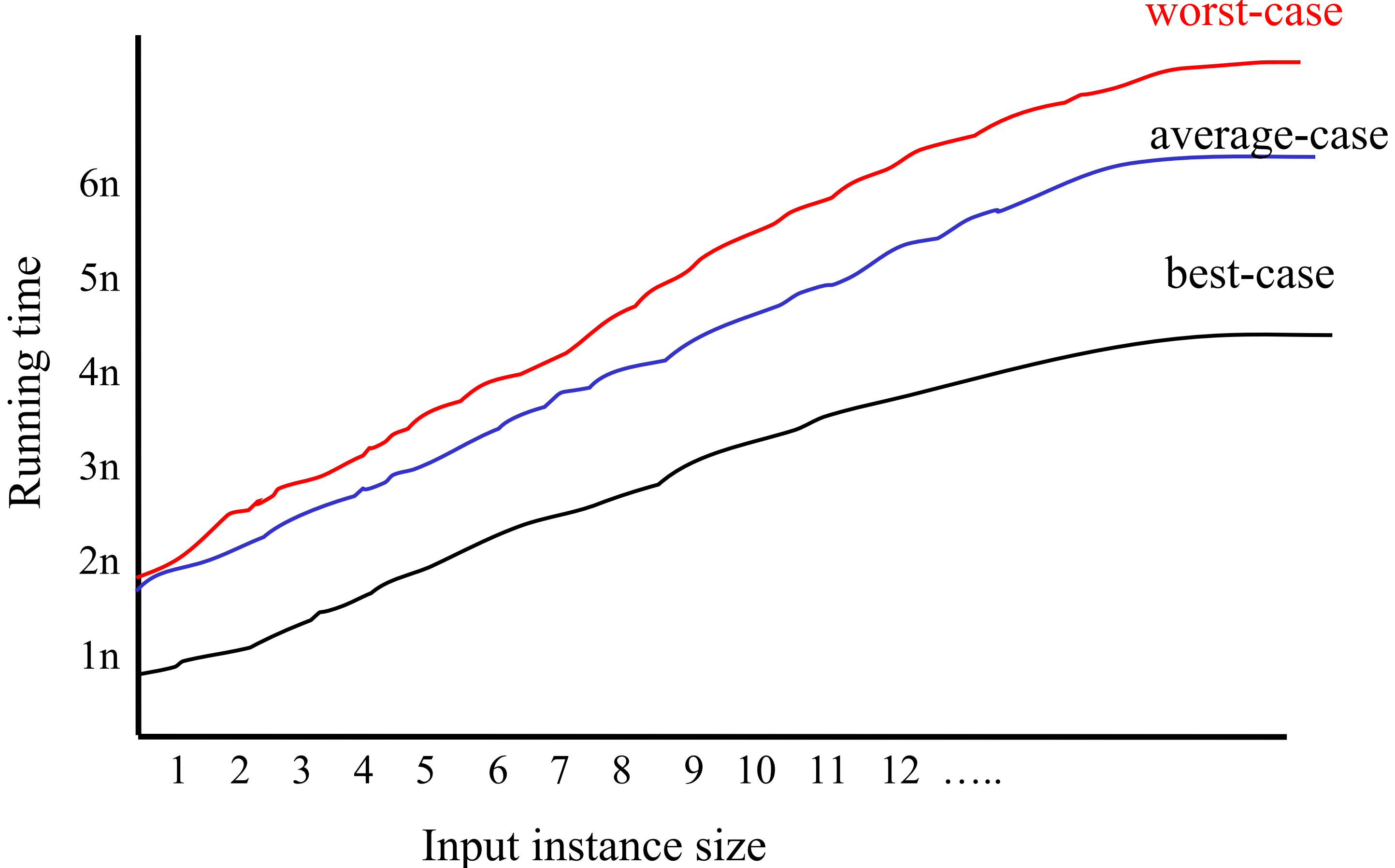
Best/Worst/Average Case (2)

– For a specific size of input *n*, investigate running times for different input instances:



Best/Worst/Average Case (3)

For inputs of all sizes:



Asymptotic notation : Intuition

Running time bound: $c_1 + c_5 - c_3 - c_4 + (c_2 + c_3 + c_4)n$
What are the values of c_i? machine-dependent

A simpler expression: $c_6 + c_7 n$ [**still complex].**

Q: Can we throw away the lower order terms?
A: Yes, if we do not worry about constants, and there exist constants c_8, c_9 such that $c_8 n \leq c_6 + c_7 n \leq c_9 n$, then we say that the running time is $\theta(n)$.

Need some mathematics to formalize this (LATER).

Q: Are we interested in small n or large?
A: Assume we are interested in large n – cleaner theory, usually realistic. BUT, remember the assumption when interpreting results!

What does asymptotic analysis not predict?

- Exact run times
- Comparison for small instances
- Small differences in performance

So far…

1. Covered basics of algorithm analysis (Ch. 1 of the text).
2. Next: Another example of algorithm analysis (Ch 2).
   More about asymptotic notation (Ch. 3).

Asymptotic notation - continued

Will do the relevant math later. For now, the intuition is:

1. O() is used for upper bounds "grows slower than"
2. Ω() used for lower bounds "grows faster than"
3. Θ() used for denoting matching upper and lower bounds. "grows as fast as"

   These are bounds on running time, not for the problem

The thumbrules for getting the running time are

1. Throw away all terms other than the most significant one -- Calculus may be needed

   e.g.: which is greater: n log n or $n^{1.001}$?
2. Throw away the constant factor.
3. The expression is Θ() of whatever's left.

   Asymptotic optimality – expression inside Θ() best possible.

A Harder Problem

INPUT: A[1..n] - an array of integers, k, $1 \leq k \leq length(A)$
OUTPUT: an element m of A such that m is the kth largest element in A.

Think for a minute

Brute Force: Find the maximum, remove it. Repeat k-1 times. Find maximum.

Q: How good is this algorithm?

A: Depends on k! Can show that the running time is $\Theta(nk)$. If k=1, asymptotically optimal.

Also true for any constant k.

If k = log n, running time is $\Theta(n \log n)$. Is this good?

If k = n/2 (MEDIAN), running time is $\Theta(n^2)$.

Definitely bad! Can sort in O(n log n)!

Q: Is there a better algorithm? YES!

Space complexity

INPUT: n distinct integers such that $n = 2^m-1$, each integer k satisfies $0 \leq k \leq 2^m-1$.
OUTPUT: a number j, $0 \leq j \leq 2^m-1$, such that j is not contained in the input.

Brute Force 1: Sort the numbers.
Analysis: $\Theta(n \log n)$ time, $\Theta(n \log n)$ space.

Brute Force 2: Use a table of size n, "tick off" each number as it is read.
Analysis: θ (n) time, θ (n) space.

Q: Can the running time be improved? No (why?)
Q: Can the space complexity be improved? YES!

Think for a minute

Space complexity – contd.

INPUT: n distinct integers such that $n = 2^m-1$, each integer k satisfies $0 \le k \le 2^m-1$.

OUTPUT: a number j, $0 \le j \le 2^m-1$, such that j is not contained in the input.

Observation:

```
  000
  001
  010
  011
  100
  101
  110
+ 111
--------------
  000
```

Keep a running bitwise sum (XOR) of the inputs. The final sum is the integer missing.

Q: How do we prove this?

Aside: When you see a new problem, ask…

1. Is it similar/identical/equivalent to an existing problem?
2. Has the problem been solved?
3. If a solution exists, is the solution the best possible?

   May be a hard question :

   Can answer NO by presenting a better algorithm.

   To answer YES need to prove that NO algorithm can do better!

   How do you reason about all possible algorithms?

   (there is an infinite set of correct algorithms)

4. If no solution exists, and it seems hard to design an efficient algorithm, is it ***intractable***?

Analysis example: Insertion sort

"We maintain a subset of elements sorted within a list. The remaining elements are off to the side somewhere. Initially, think of the first element in the array as a sorted list of length one. One at a time, we take one of the elements that is off to the side and we insert it into the sorted list where it belongs. This gives a sorted list that is one element longer than it was before. When the last element has been inserted, the array is completely sorted."

English descriptions:

- Easy, intuitive.
- Often imprecise, may leave out critical details.

Insertion sort: pseudocode

```
for j=2 to length(A)
  do key=A[j]
    i=j-1
    while i>0 and A[i]>key
      do A[i+1]=A[i]
         i--
    A[i+1]:=key
```

Can you understand
The algorithm?
I would not know
this is insertion sort!

Moral: document code!

We will prove this algorithm correct when we study writing correctness proofs

Analysis of Insertion Sort

Let's compute the **running time** as a function of the **input size**

code	cost	times
for j←2 **to** *n*	c_1	n
do key←A[j]	c_2	n-1
Insert A[j] into the sorted sequence A[1..j-1]	0	n-1
i←j-1	c_3	n-1
while i>0 **and** A[i]>key	c_4	$\sum_{j=2}^{n} t_j$
do A[i+1]←A[i]	c_5	$\sum_{j=2}^{n} (t_j - 1)$
i ← i-1	c_6	$\sum_{j=2}^{n} (t_j - 1)$
A[i+1] ← key	c_7	n-1

Analysis of Insertion Sort – contd.

- **Best case**: elements already sorted $\rightarrow$ $t_j=1$, running time = $\Theta(n)$, i.e., *linear* time.
- **Worst case**: elements are sorted in inverse order $\rightarrow$ $t_j=j$, running time = $\Theta(n^2)$, i.e., *quadratic* time
- **Average case**: $t_j=j/2$, running time = $\Theta(n^2)$, i.e., *quadratic* time

- We can see that insertion sort has a worst case running time $An^2 + Bn + C$, where $A = (c_5+c_6+c_7)/2$ etc.
- Q1: How useful are the details in this result?
- Q2: How can we simplify the expression?

Back to asymptotics……

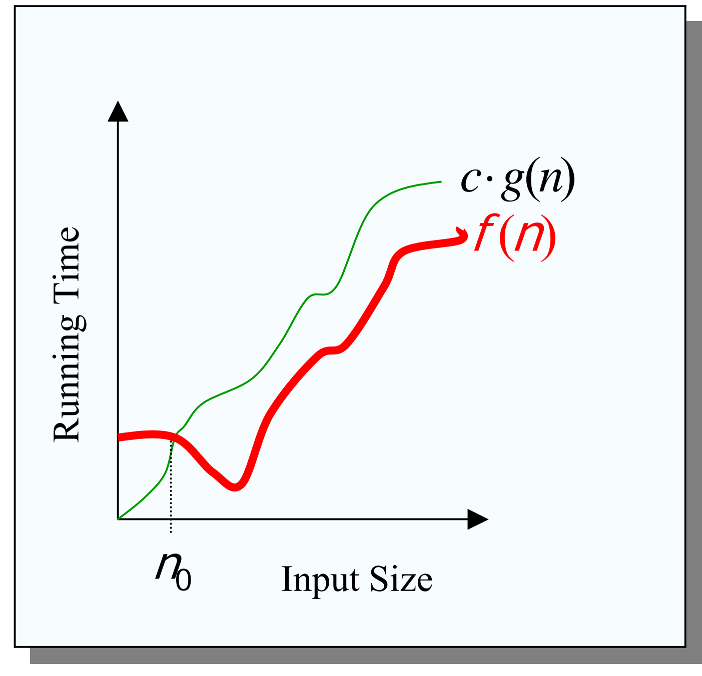
We will now look more formally at the process of simplifying running times and other measures of complexity.

Asymptotic analysis

- Goal: to simplify analysis of running time by getting rid of "details", which may be affected by specific implementation and hardware
  - like "rounding": $1{,}000{,}001 \approx 1{,}000{,}000$
  - $3n^2 \approx n^2$
- Capturing the essence: how the running time of an algorithm increases with the size of the input *in the limit*.
  - Asymptotically more efficient algorithms are best for all but small inputs

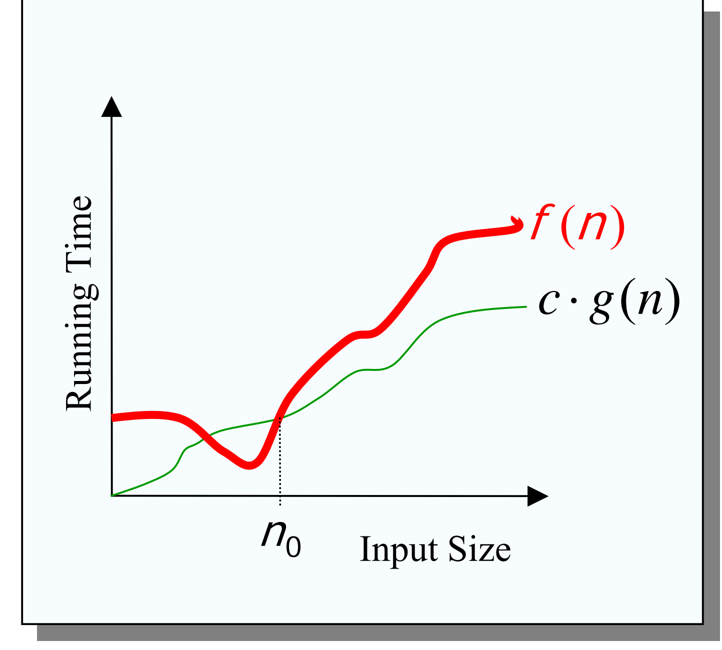
Asymptotic notation

- The "big-Oh" *O*-Notation
  - asymptotic upper bound
  - $f(n) \in O(g(n))$, if there exists constants c and n_0, $s.t.$ $\mathbf{f(n) \leq c\ g(n)}$ for $n \geq n_0$
  - $f(n)$ and $g(n)$ are functions over non-negative integers
- Used for *worst-case* analysis



Asymptotic notation – contd.

- The "big-Omega" Ω–Notation
  - asymptotic lower bound
  - $f(n) \in \Omega(g(n))$ if there exists constants *c* and n_0, *s.t.* $\mathbf{c\ g(n)} \leq \mathbf{f(n)}$ for $n \geq n_0$
- Used to describe *best-case* running times or lower bounds of algorithmic problems
  - E.g., lower-bound of searching in an unsorted array is $\Omega(n)$.



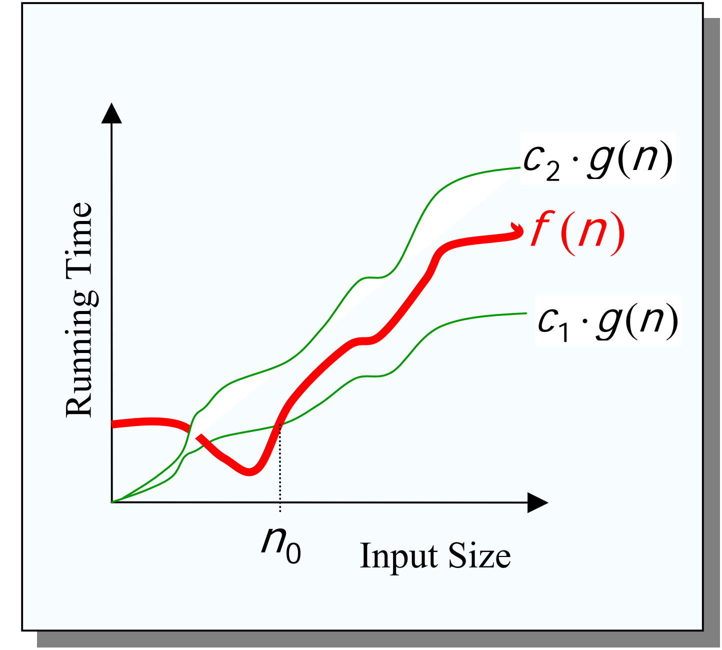
Asymptotic notation – contd.

- Simple Rule: Drop lower order terms and constant factors.
  - $50\, n \log n \in O(n \log n)$
  - $7n - 3 \in O(n)$
  - $8n^2 \log n + 5n^2 + n \in O(n^2 \log n)$

- Note: Even though $50\, n \log n \in O(n^5)$, we usually try to express a O() expression using as small an order as possible

Asymptotic notation – contd.

- The "big-Theta" Θ–Notation
  - asymptoticly tight bound
  - $f(n) \in \Theta(g(n))$ if there exists constants c_1, c_2, and n_0, $s.t.$ $\mathbf{c_1\ g(n) \leq f(n) \leq c_2\ g(n)}$ for $n \geq n_0$
- $f(n) \in \Theta(g(n))$ if and only if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$
- $O(f(n))$ is often misused instead of $\Theta(f(n))$



Asymptotic notation – contd.

- Two more asymptotic notations
  - "Little-Oh" notation $f(n)=o(g(n))$ non-tight analogue of Big-Oh
    - For every c, there should exist n_0, s.t. $\mathbf{f(n) \leq c\ g(n)}$ for $n \geq n_0$
    - Used for **comparisons** of running times.
      If $f(n) \in o(g(n))$, it is said that $g(n)$ *dominates* $f(n)$.
    - More useful defn:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

  - "Little-omega" notation $f(n) \in \omega(g(n))$ non-tight analogue of Big-Omega

Asymptotic notation – contd.

- (VERY CRUDE) Analogy with real numbers
  - $f(n) = O(g(n)) \quad \cong \quad f \leq g$
  - $f(n) = \Omega(g(n)) \quad \cong \quad f \geq g$
  - $f(n) = \Theta(g(n)) \quad \cong \quad f = g$
  - $f(n) = o(g(n)) \quad \cong \quad f < g$
  - $f(n) = \omega(g(n)) \quad \cong \quad f > g$
- Abuse of notation: $f(n) = O(g(n))$ actually means $f(n) \in O(g(n))$.

Points to ponder and lessons

Common "colloquial" uses:

$\Theta(1)$ – *constant.*

$n^{\Theta(1)}$ – *polynomial*

$2^{\Theta(n)}$ – *exponential*

Be careful!

$n^{\Theta(1)} \neq \Theta(n^1)$

$2^{\Theta(n)} \neq \Theta(2^n)$

- When is asymptotic analysis useful?
- When is it NOT useful?

Many, many abuses of asymptotic notation in Computer Science literature.

Lesson: Always remember the implicit assumptions…

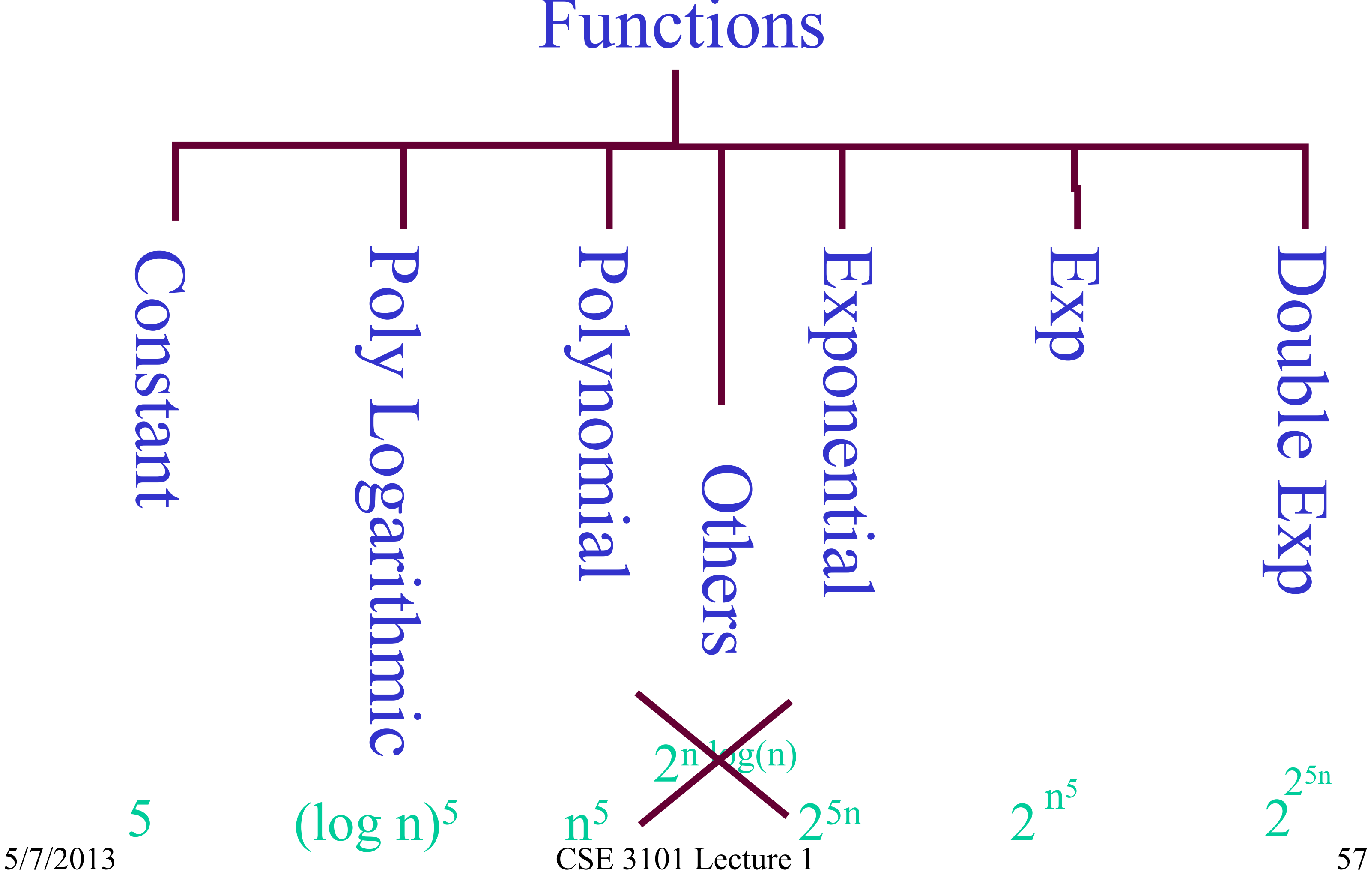
Comparison of Running Times

Running Time	Maximum problem size (n)		
	1 second	1 minute	1 hour
$400n$	2500	150000	9000000
$20n \log n$	4096	166666	7826087
$2n^2$	707	5477	42426
n^4	31	88	244
2^n	19	25	31

Classifying functions

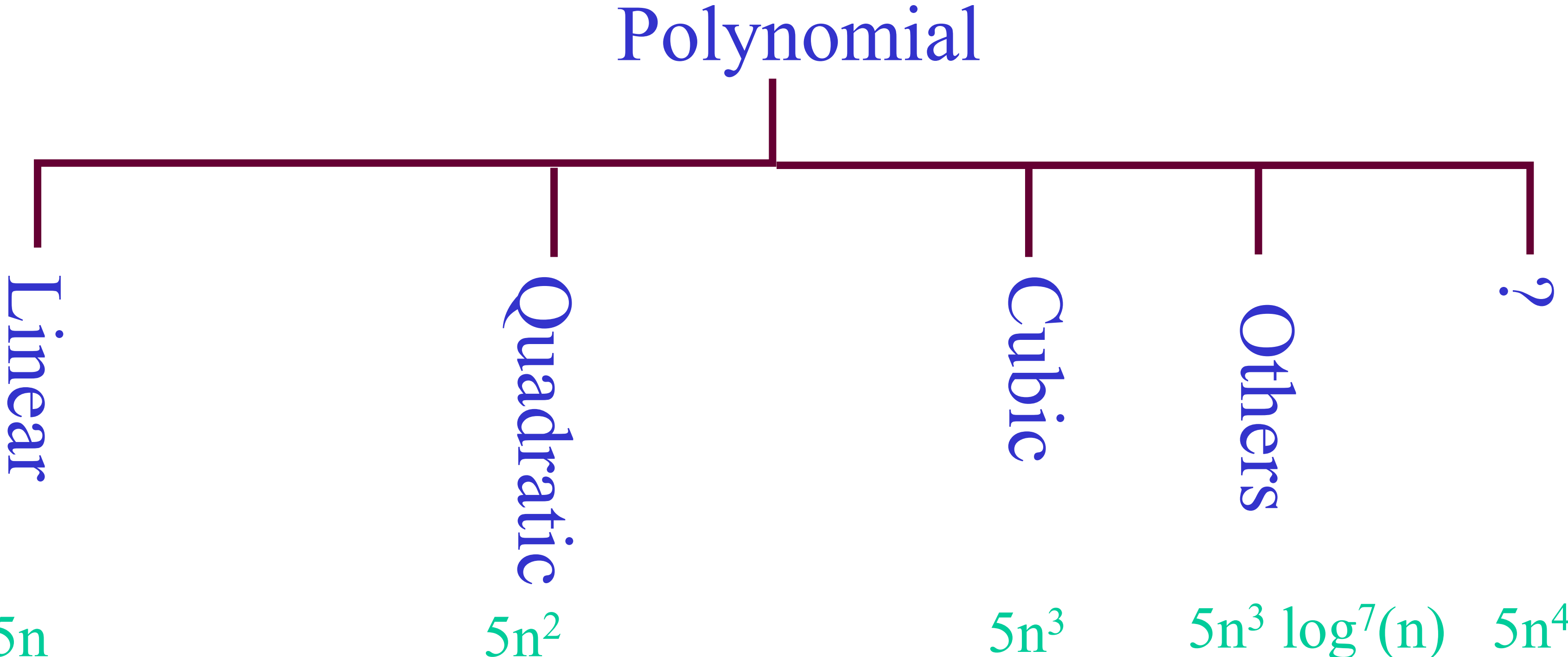
$T(n)$	10	100	1,000	10,000
$\log n$	3	6	9	13
$n^{1/2}$	3	10	31	100
n	10	100	1,000	10,000
$n \log n$	30	600	9,000	130,000
n^2	100	10,000	10^6	10^8
n^3	1,000	10^6	10^9	10^{12}
2^n	1,024	10^{30}	10^{300}	10^{3000}

Hierarchy of functions



Classifying Polynomials

Dominant term is of the form n^c



$5n$ $5n^2$ $5n^3$ $5n^3 \log^7(n)$ $5n^4$

Logarithmic functions

- $\log_{10} n$ = # digits to write n
- $\log_2 n$ = # bits to write n
  $= 3.32 \log_{10} n$
- $\log(n^{1000}) = 1000 \log(n)$

Differ only by a multiplicative constant.

Poly Logarithmic (a.k.a. polylog)

$$(\log n)^5 = \log^5 n$$

Crucial asymptotic facts

Logarithmic << Polynomial

$\log^{1000} n \ll n^{0.001}$ For sufficiently large n

Linear << Quadratic

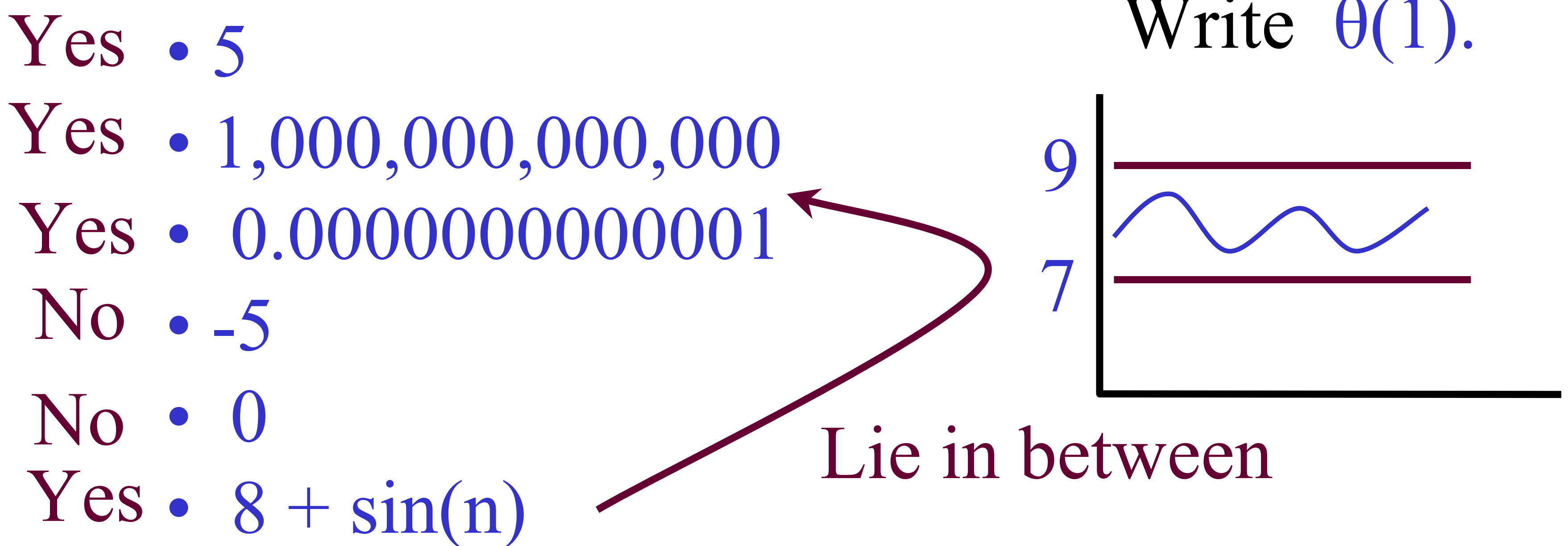
$10000\, n \ll 0.0001\, n^2$ For sufficiently large n

Polynomial << Exponential

$n^{1000} \ll 2^{0.001\, n}$ For sufficiently large n

Are constant functions constant?

The running time of the algorithm is a "constant"
It does not depend **significantly**
on the size of the input.



Polynomial Functions

Quadratic

- n^2
- $0.001\ n^2$
- $1000\ n^2$
- $5n^2 + 3000n + 2\log n$

Lie in between

Polynomial

- n^c
- $n^{0.0001}$
- n^{10000}
- $5n^2 + 8n + 2\log n$
- $5n^2 \log n$
- $5n^{2.5}$

Lie in between

Exponential functions

- 2^n
- $2^{0.0001\, n}$
- $2^{10000\, n}$
- $8^n \quad = 2^{3n}$
- $2^n / n^{100} \quad > 2^{0.5n}$
- $2^n \cdot n^{100} \quad < 2^{2n}$

$$2^{0.5n} > n^{100}$$

$$2^n = 2^{0.5n} \cdot 2^{0.5n} > n^{100} \cdot 2^{0.5n}$$

$$2^n / n^{100} > 2^{0.5n}$$

Proving asymptotic expressions

Use definitions!

e.g. $f(n) = 3n^2 + 7n + 8 = \theta(n^2)$

$f(n) \in \Theta(g(n))$ if there exists constants c_1, c_2, and n_0, $s.t.$

$\mathbf{c_1\, g(n) \leq f(n) \leq c_2\, g(n)}$ for $n \geq n_0$

Here $g(n) = n^2$

One direction $(f(n) = \Omega(g(n))$ is easy

$c_1\, g(n) \leq f(n)$ holds for $c_1 = 3$ and $n \geq 0$

The other direction $(f(n) = O(g(n))$ needs more care

$f(n) \leq c_2\, g(n)$ holds for $c_2 = 18$ and $n \geq 1$ (CHECK!)

So $n_0 = 1$

Proving asymptotic expressions – contd.

Caveats!

1. constants c_1, c_2 MUST BE POSITIVE.
2. Could have chosen $c_2 = 3 + \varepsilon$ for any $\varepsilon > 0$. WHY?

-- because $7n + 8 \leq \varepsilon n^2$ for $n \geq n_0$ for some sufficiently large n_0. Usually, the smaller the ε you choose, the harder it is to find n_0. So choosing a large ε is easier.

3. Order of quantifiers

$\exists c_1\ c_2\ \exists n_0\ \forall\ n \geq n_0,\ c_1 g(n) \leq f(n) \leq c_2 g(n)$

vs

$\exists n_0\ \forall\ n \geq n_0\ \exists c_1\ c_2,\ c_1 g(n) \leq f(n) \leq c_2 g(n)$

-- allows a different c_1 and c_2 for each n. Can choose $c_2 = 1/n$!! So we can "prove" $n^3 = \Theta(n^2)$.

Why polynomial vs exponential?

Philosophical/Mathematical reason – polynomials have different properties, grow much slower; mathematically natural distinction.

Practical reasons

1. almost every algorithm ever designed and every algorithm considered practical are very low degree polynomials with reasonable constants.
2. a large class of natural, practical problems seem to allow only exponential time algorithms. Most experts believe that there do not exist any polynomial time algorithms for any of these; i.e. $P \neq NP$.