CES 512 Theory of Software Systems

B. Ravikumar (Ravi)
Office: 116 I Darwin Hall
Phone: 664 3335
E-mail: ravi93@gmail.com

Course Web site:
http://ravi.cs.sonoma.edu/ces512sp08

Textbook for the course:

Introduction to Algorithms
(Second edition)

Cormen, Leiserson, Rivest and Stein

How will we use the book?

- Many chapters will be covered from the text. (Roughly 80% of the material will be from the text.)
- Roughly 80% of the home work problems will be from the text. (Most of the remaining 20% will be coding and implementing.)

Course Goals:

- develop critical thinking about problem solving in the context of software design
- systematic approach to problem solving
- ability to analyze the solution – both experimentally and by analytical techniques
- focus on models – understanding the context of a problem and create models to approach problem solving
- applications: case studies in which algorithmic model and solutions lead to efficient and elegant solutions

Goals for today's lecture

- Course outline
- Discuss Course work
  - homework
  - quiz
  - tests, final exam
  - course project
- Cover Chapters 1-3 of the text.

Context of the course in Master's program MS-CES

- I created this course mainly because I felt that the students who want to do a thesis in Software Design need to know this course material. (More specifically, the thesis topics I would like to supervise will require background similar to this.)

- Emphasis on theoretical ideas and models. But we will study only the models and techniques that have practical consequence for real-world software development.

- Other courses in this track:
  - Data Mining
  - High-Performance Computing
  - Advanced Computer Architecture
  - Data Compression
  - Intelligent System Design
  - Advanced Software Engineering
  - Hardware/Software Testing

Course Goals

- tools for algorithm design
  - divide and conquer, induction
  - efficient data structures
  - graph searching, network flow, matching
  - linear programming
  - dynamic programming
  - scheduling and other optimization problems
  - probabilistic techniques
- tools for analysis
  - recurrence relations, summation of series
  - approximations, inequalities
  - reductions between problems
  - experimental analysis

- Applications
  - image processing
  - string matching and document processing
  - data and image compression

What is an Algorithm?

Algorithm Design and Analysis is at the core of software design.

Definition of an algorithm:

An algorithm is a finite, definite, effective procedure, with some input and some output.

- finite: terminates for every input
- definite: each instruction is precisely stated.
- effective procedure: control flow is well defined, i.e. there are clear rules to determine what the next step is.
- input and output: algorithm is the process of transforming the input to the output.

Origin of the term "algorithm"

- Etymology from: [Don Knuth, The Art of Computer Programming]

  - *Algorism* = process of doing arithmetic using Arabic numerals.

  - True origin: Abu 'Abd Allah Muhammad ibn Musa al-Khwarizm was a famous 9th century Persian textbook author who wrote *Kitab al-jabr wa'l-muqabala*, which evolved into today's high school algebra text.

Taxonomy

- Broad classification
  - numerical algorithms
  - non-numerical algorithms
  - semi-numerical algorithms (that have a flavor of both)

Taxonomy

- Numerical algorithms
  - Polynomial computations
    - interpolation
    - Root finding, splines
  - matrix operations
    - matrix multiplication
    - inverse, determinant
    - system of linear equations, eigenvalue computation
  - Solving non-linear equations
    - Iterative method for solving f(x) = 0
    - Newton-Raphson method
    - Optimization problems
  - Numerical solution of differential equations, PDE etc.

- We will only cover a very small subset of these problems, if at all.

Non-numerical algorithms

- Search algorithms
  - sorting
  - search trees, red-black trees etc.
  - hashing
  - Other data structures and applications (priority queue etc.)
- Graph algorithms
  - graph exploration (depth-first and breadth-first search)
  - shortest path, spanning tree problems
  - matching, network flow
- Geometric algorithms
  - nearest neighbor, convex hull, visibility problems etc.
- String algorithms
- Number-theoretic algorithms
  - basic arithmetic operations (multiplication, Chinese remaindering)
  - testing for prime, factorization, integer exponentiation etc.

Central Themes

- algorithm design techniques: (general purpose techniques)
  - Hashing, priority queues, binary search trees
  - divide and conquer
  - induction
  - Backtracking
  - randomization
  - dynamic programming etc.
- standard problems: (problems that arise often)
  - sorting and searching
  - string matching
  - shortest path, connectivity testing
  - polynomial multiplication

Central Themes

- Converting one problem to another:
  - Consider a 2-player game: There are 9 cards with numbers 1, 2, …, 9 written on it. Players alternately pick a card. If a player has three cards adding to 15, he/she wins. What is the optimal strategy for playing?

- Algorithm Analysis:
  - time complexity (worst- and average case)
  - other requirements (optimality)
  - tools from Discrete Math (summation, inequalities, recurrence relations etc.)

Central Themes

- Heuristics: (will cover very briefly)
  - backtracking, branch and bound
  - simulated annealing, genetic algorithms, local search

- Applications
  - Every field of software design requires a good understanding of algorithms.

  - Major application areas: web programming, search engines, network optimization, bioinformatics, graphics etc.

Algorithm Design and Analysis

- Efficient algorithms are a crucial part of software design
  - implementation and coding details can improve the performance, but a more efficient algorithm would provide significantly greater gains.
  - optimization problems allow multiple "correct" solutions, but only one correct and optimal solution. A small improvement (say 1%) may translate into huge profit/gain.
  - correctness of algorithms can be tricky to establish.
  - estimating the resource requirements is a crucial part of software problem solving.

- Some standard approaches to systematic algorithm design
  - identify commonly occurring problems (such as sorting, searching, hashing, breadth-first search etc.) and design the most efficient algorithms for them.
  - devise general-purpose techniques (induction, divide and conquer, backtracking, dynamic programming etc.) and study when they are effective by looking at many examples.
  - create and study models/ framework (such as trees, graphs, probabilistic algorithms, grammars and automata) that can be used to represent problems and solutions.

Some models

Graphs:

- can represent road networks
- nodes -> cities, edges -> highways connecting them.

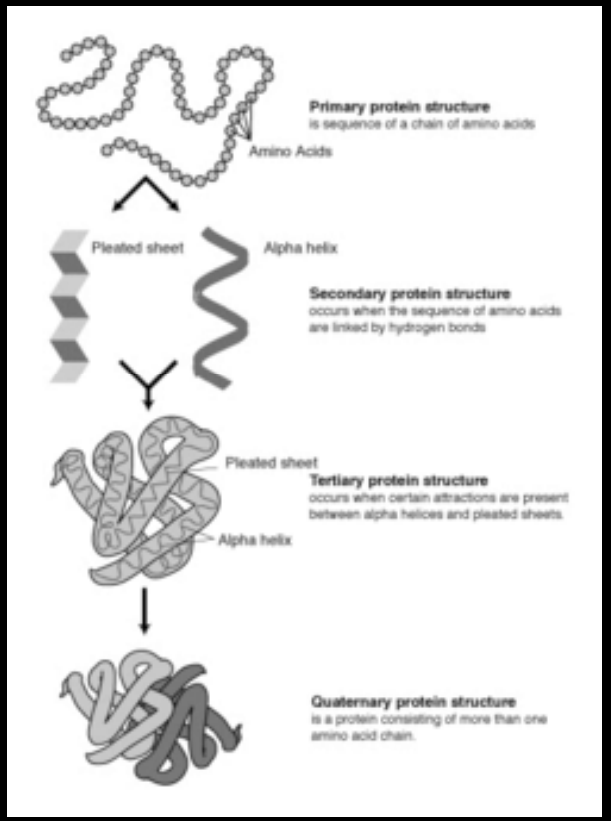
But graphs can represent many other relational data

- Example:
  scheduling and graph coloring

Another model

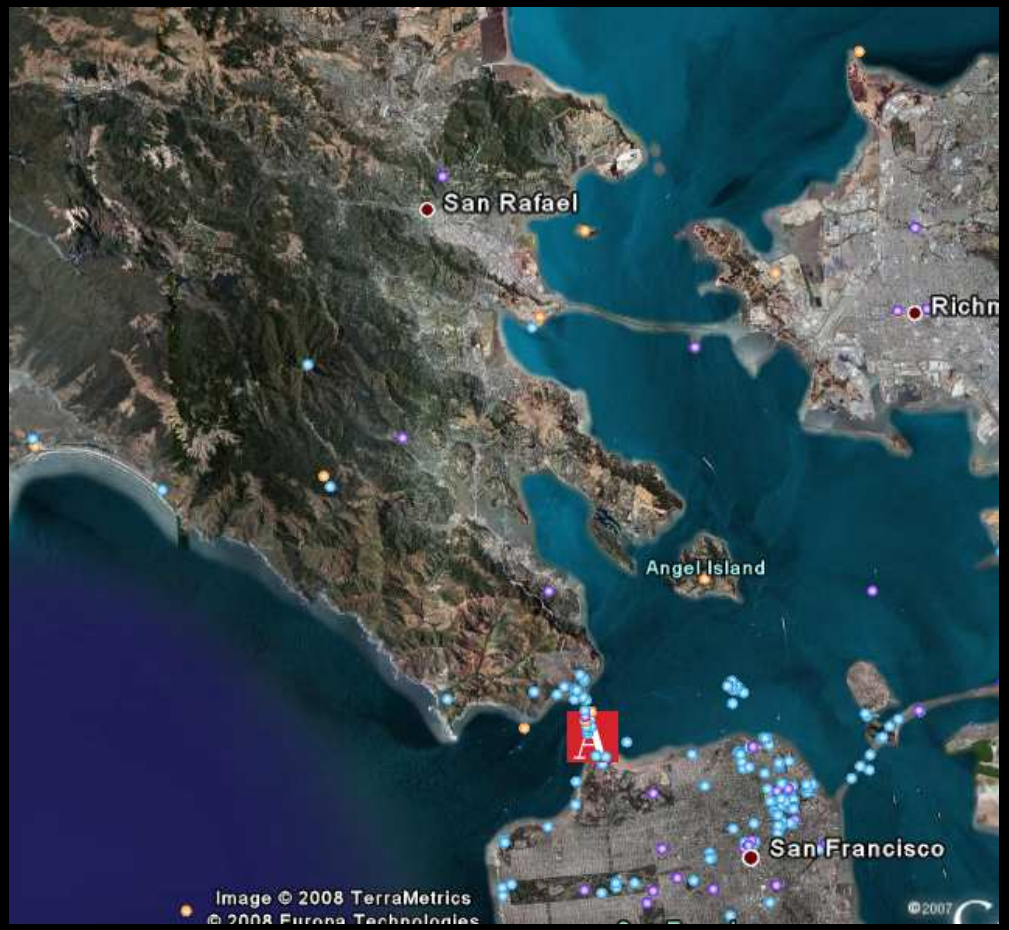
Proteins

- How to represent it as digital data?
  - Primary structure
  - Secondary and tertiary structure
- Computational problems
  - Classification of proteins
  - Function of proteins



An example from GIS

The entire landscape of the world is being digitized (there is a whole new branch that combines information technology and geography called GIS – Geographic Information System). What kind of data structure should be used to store all this information?



Snapshot from Google earth

Some general issues related to GIS

- How much memory do we need? Can this be stored in one computer? (or need a distributed data base?)

  Building the database is done in the background (off-line processing)

- How fast can the queries be answered?

  Response to query is called the on-line processing

- Suppose each square mile is represented by a 1024 by 1024 pixel image, how much storage do we need to store the terrain of United States?

Calculate the memory needed

Very rough estimate of the memory needed:

- Area of USA is 4×10^6 sq miles (roughly)
- Each square mile needs 10^6 pixels (roughly)
- Each pixel requires 32 bits usually.

Thus the total memory needed
$= 4 \times 10^6 \times 32 \times 10^6 = 168 \times 10^{12} = 168000$ Giga bits

(A standard desk top has ~ 200 Giga bits of memory.)

Need about 800 such computers to store the data

What data structure to store the images?

- each 1024 x 1024 image can be stored in a two-dimensional array. (standard way to store all kinds of images – bmp, jpg, png etc.) The actual images are stored in a secondary memory (hard disks on several servers either in a central location or distributed).

- The number of images would be roughly 4×10^6. A set of pointers to these images can be stored in a 1 (or 2) dimensional array.

- When you click on a point on the map, its index in the array is calculated.

- From that index, the image is accessed and sent by a network to the requesting client.

Analysis of algorithms

- time taken to solve a problem as a function of problem size usually behaves as a well-defined function (logarithmic, linear, quadratic, n log n, exponential, etc.)

- when the time complexity function is complicated, it can be approximated to a well-defined function.

- problems can be classified based on the time complexity into easy (polynomial time solvable), provably intractable (not polynomial time solvable) and hard (not known to be polynomial time solvable, and not likely to be so).

Course overview in detail

- Mathematical preliminaries (although assumed, we will cover them quickly.)
  - induction
  - summation, approximation, estimation
  - inequalities, upper and lower bounds
  - O, Ω and Θ notation
  - recurrence relations
  - review of discrete math
- sorting and searching problems (some of it covered others will be in HW)
  - heap sort and merge sort
  - quick sort
  - binary search tree
  - hashing
  - applications

- Design Techniques
  - divide and conquer
  - dynamic programming
  - greedy method
  - probabilistic method
  - backtracking
  - genetic algorithms, neural networks etc.

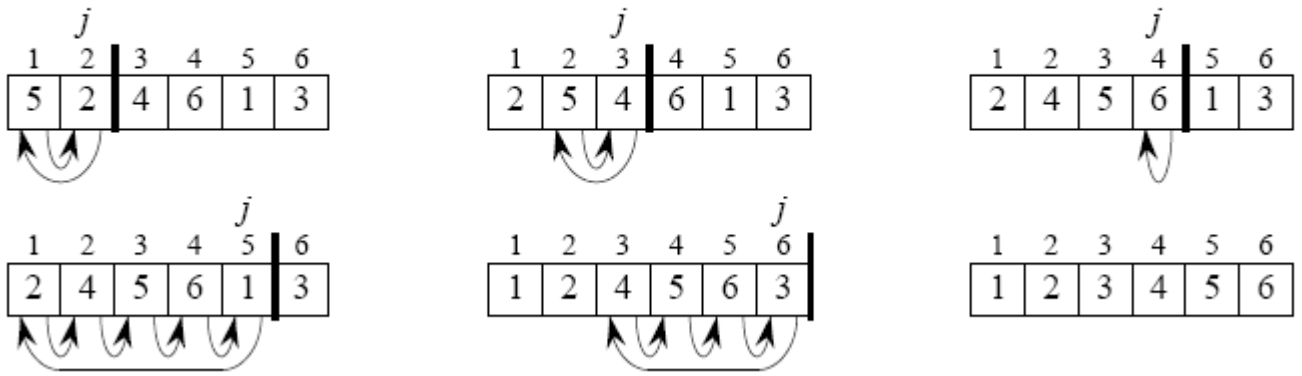
- Graph Problems
  - basic definitions and models
  - DFS, BFS and applications
  - minimum spanning tree
  - shortest path problem
  - matching in a bipartite graph
  - Network flow
- Linear Programming
  - simplex method
  - duality
  - applications

- Applications
  - string processing and biology
  - image processing
  - cryptography

Insertion Sorting (Chapter 2 of text)

INSERTION-SORT(A)	*cost*	*times*
for $j \leftarrow 2$ **to** n	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] \leftarrow key$	c_8	$n-1$

Example:



Analysis of insertion sort

[Now add statement costs and number of times executed to INSERTION-SORT *pseudocode.]*

- Assume that the ith line takes time c_i, which is a constant. (Since the third line is a comment, it takes no time.)
- For $j = 2, 3, \ldots, n$, let t_j be the number of times that the **while** loop test is executed for that value of j.
- Note that when a **for** or **while** loop exits in the usual way—due to the test in the loop header—the test is executed one time more than the loop body.

The running time of the algorithm is

$$\sum_{\text{all statements}} (\text{cost of statement}) \cdot (\text{number of times statement is executed}) .$$

Let $T(n) =$ running time of INSERTION-SORT.

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\ &\quad + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) . \end{aligned}$$

c_1, c_2 etc. depend on the architecture. t_j depends on the input.

Best case: The array is already sorted.

- Always find that $A[i] \leq key$ upon the first time the **while** loop test is run (when $i = j - 1$).
- All t_j are 1.
- Running time is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \ . \end{aligned}$$

- Can express $T(n)$ as $an + b$ for constants a and b (that depend on the statement costs c_i) $\Rightarrow$ $T(n)$ is a *linear function* of n.

Worst case: The array is in reverse sorted order.

- Always find that $A[i] > key$ in while loop test.
- Have to compare key with all elements to the left of the jth position $\Rightarrow$ compare with $j - 1$ elements.
- Since the while loop exits because i reaches 0, there's one additional test after the $j - 1$ tests $\Rightarrow t_j = j$.
- $\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j$ and $\sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} (j - 1)$.
- $\sum_{j=1}^{n} j$ is known as an ***arithmetic series***, and equation (A.1) shows that it equals $\frac{n(n+1)}{2}$.
- Since $\sum_{j=2}^{n} j = \left(\sum_{j=1}^{n} j\right) - 1$, it equals $\frac{n(n+1)}{2} - 1$.

- Running time is

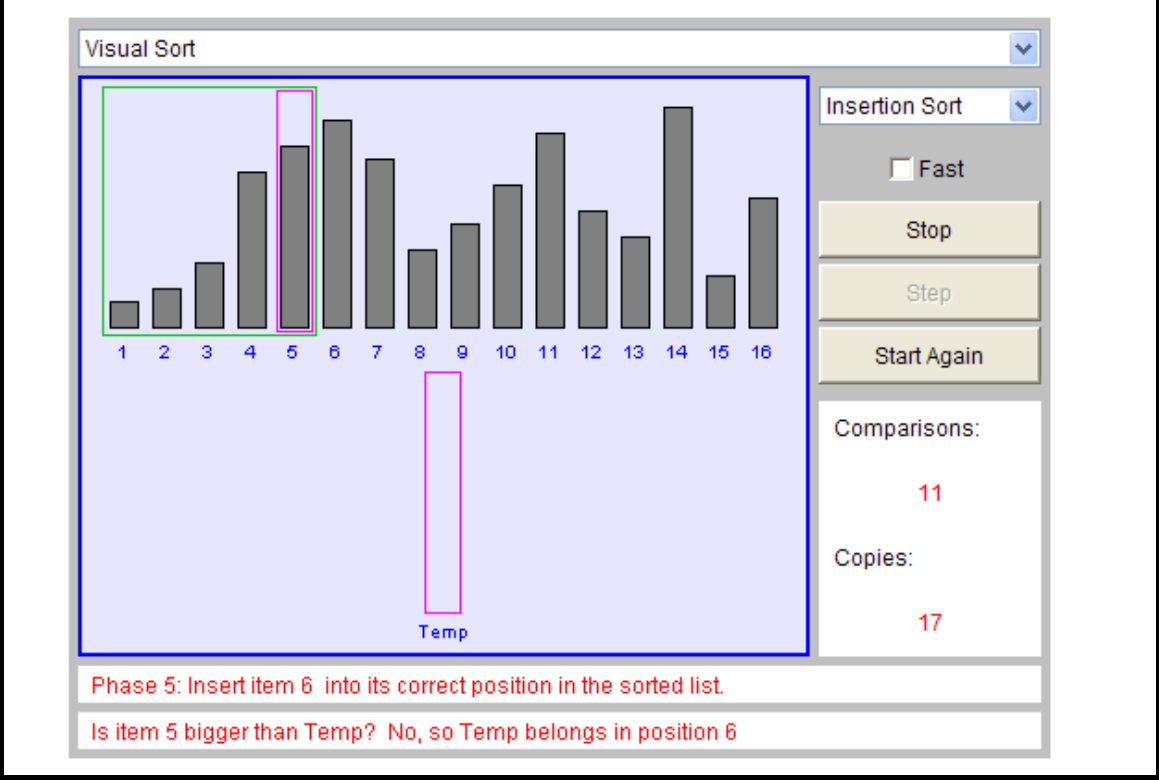
$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8) \ . \end{aligned}$$

- Can express $T(n)$ as $an^2 + bn + c$ for constants a, b, c (that again depend on statement costs) $\Rightarrow$ $T(n)$ is a *quadratic function* of n.

The following link contains an applet to animate various sorting (and other) algorithms:

http://math.hws.edu/TMCM/java/xSortLab/

A screen shot is shown below:



Another sorting animation can be found at:

http://www.cs.hope.edu/alganim/animator/Animator.html

Exercise 2.3.5

Write pseudo-code for binary search and argue that the worst-case running time is $\Theta(\log n)$.

Solution:

```
ITERATIVE-BINARY-SEARCH(A, v, low, high)
while low ≤ high
    do mid ← ⌊(low + high)/2⌋
       if v = A[mid]
          then return mid
       if v > A[mid]
          then low ← mid + 1
          else high ← mid − 1
return NIL
```

Each time the array size becomes half the original size, so the number of iterations is $\Theta(\log n)$.

Mathematical Preliminaries

- induction

- proof technique for assertion of the form " for all integer n, ... "

Example 1: *For all n, $n^3 + 5n$ is divisible by 6.*

Example 2.

$1 + r + \ldots + r^k = (r^{k+1} - 1)/(r-1)$

Chapter 3 – Standard functions

Most commonly occurring functions in algorithm studies:

- linear (of the form: $an + b$)
- quadratic ($an^2 + Bn + c$)
- more generally, polynomial functions
  ($a\, n^k + \ldots$)

- exponential (2^n more generally c^n)
- factorial ($n! = 1 \times 2 \ldots \times n$)

- logarithm

Recall basic properties of these functions

- laws of logarithms:
  - $\log_B (ab) = \log_B a + \log_B b$
  - $\log_B (a/b) = \log_B a - \log_B b$
  - $\log_B (a^c) = c \log_B a$ etc.

- laws of exponential function:

$a^b \times a^c = a^{b+c}$

$a^{-b} = 1/ a^b$

$(a^b)^c = a^{bc}$

More about log function

$$
\begin{aligned}
a &= b^{\log_b a}, \\
\log_c(ab) &= \log_c a + \log_c b, \\
\log_b a^n &= n \log_b a, \\
\log_b a &= \frac{\log_c a}{\log_c b}, \\
\log_b(1/a) &= -\log_b a, \\
\log_b a &= \frac{1}{\log_a b}, \\
a^{\log_b c} &= c^{\log_b a}.
\end{aligned}
$$

Inequalities, Chapter 3

A basic inequality:

For any positive real constants c and d where c > 1, we have: $c^n > n^d$ for all sufficiently large n.

Thus, $1.00001^n > n^{10000000}$ when n becomes sufficiently large.

This means that an algorithm whose time complexity is polynomial in (n = the size of the problem) is better than an algorithm whose time complexity is exponential in n. Thus, as a first step, we should seek polynomial time algorithms (over exponential time ones).

O, Ω and Θ notation

Definition: Let f(n) and g(n) be two functions defined on the set of integers. If there is a c > 0 such that f(n) <= c g(n) for all large enough n. Then, we say f(n) = O(g(n)).

- If f(n) = O(g(n)) then, we say that g(n) = Ω(g(n)).
- If f(n) = O(g(n)) and g(n) = O(f(n)) then we say that f(n) = Θ(g(n)).

One way to show that f(n) = O(g(n)) is to show that

f(n)/g(n) → c for some finite c as n goes to infinity.

Example: $n^2 + 2n - 15 = O(n^2)$

Some general facts for comparing functions

- fact 1: $n^r = \Omega(n^s)$ if $r > s$. (equivalently, $n^s = O(n^r)$ if $r > s$.
- fact 2: If $c > 1$ then $c^n = \Omega(n^d)$ (last slide).
- fact 3: For any c, d, $(\log n)^c = O(n^d)$.
- fact 4: If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$ then $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$
- fact 5: Fact 4 with + replaced by *

Some exercises from Chapter 3:

Exercise 3.2-3:
Show that log n! = Θ(n log n)

Proof: Note that we need to prove that log n! = O(n log n) and log n! = Ω(n log n)

Problem 3-3: order the following functions by growth rate

$n \cdot 2^n$ $(\lg n)!$ $(3/2)^n$ n^3