EECS 114:
Engineering Data Structures and Algorithms
Lecture 1

Instructor: Ryan Rusich
E-mail: rusichr@uci.edu
Office: EH 2204

The Henry Samueli School of Engineering
Electrical Engineering and Computer Science
University of California, Irvine

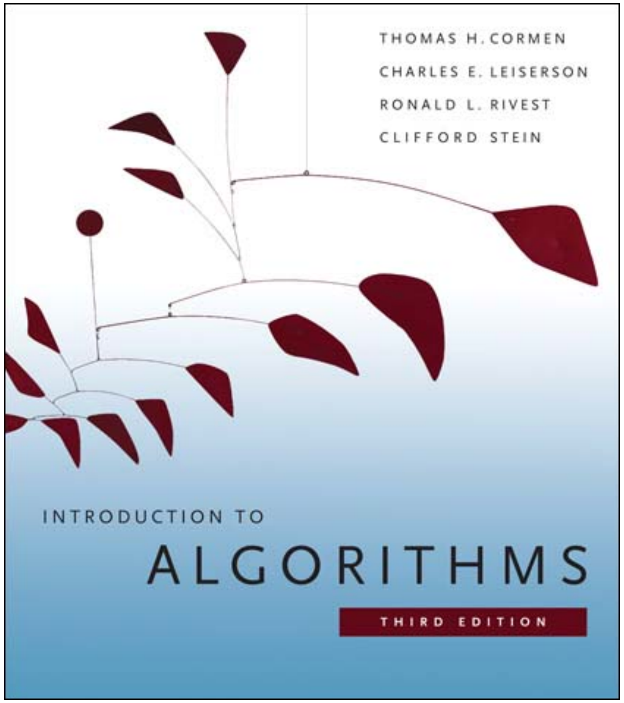
Course Administration

- Course web page
  https://piazza.com/uci/fall2015/eecs114/home
  - Syllabus
  - Class Forum
  - Assignments & Hws (password protected)
  - Labs
  - Submission Links
  - Lecture Slides
  - General info
- Course communication
  - Piazza (announcements and class forum)
  - Dropbox (confirmation e-mail sent)
  - EEE Mailing list (announcements sparingly)

Course Textbook

Introduction to Algorithms, 3rd Edition

by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.



Code Graded on EECS Servers

- Log into the server
  - Terminal with SSH protocol (secure shell)
  - EECS Linux servers
    - **`zuma.eecs.uci.edu`**
    - **`crystalcove.eecs.uci.edu`**
  - User name, password

NOTE: Programs that do not compile will NOT be graded.

Java Compilation

- Programming assignments should be completed in Java
- To compile a file into a class file

```
javac file.java
```

- To execute a class file

```
java file.class
```

- Java documentation available at http://java.sun.com/

What is an Algorithm?

- An **algorithm**
  - Takes an input (a value or set of values)
  - Produces an output (a value or set of values)
  - Terminates
  - Output satisfies some correctness property

    (e.g., the output of a sorting algorithm is sorted)

- An **algorithm** is a step-by-step procedure of unambiguous instructions for solving a problem in a finite amount of time.

Why take this class?

- Fundamental - cross cutting across all areas of computer science
- Analysis aspect - need to know how long an algorithm takes to execute (will your code work with 1 million entries, 1 billion?), how to classify the difficult of problems
- Provides many solutions for a given problems
- Many applications of a given solutions

Other Reasons

- When unemployment is UP, you need to be competitive.
- Interviewers for CS, CE, SE jobs typically ask algorithms questions.
- Why?
  - Easy to ask
  - Consider knowledge important in the work force
  - Common language to communicate in computer science

Example Algorithm: Sorting n integers

- Problem statement:
  - Input: An array $A=\{a_1, a_2, \ldots, a_n\}$
  - Output: An array $A'=\{a'_1, a'_2, \ldots, a'_n\}$ such that $a_i \leq a_{i+1}$ for $1 \leq i < n$.
- Many different possible algorithms to solve this problem
  - Different algorithms can have very different runtimes
  - Important to understand behavior of algorithm (can it handle large inputs)?

Analysis of Execution Time

- Use algorithm analysis to characterize behavior of algorithms
- Assumptions:
  - RAM (random access memory) model - all memory accesses are constant time
  - Sequential instruction execution (single processor)
  - Basic instructions are constant time (add, multiple, divide, subtract, compares, …)

Algorithm Runtime

- Could measure it, but want a formula $T(n)$ where n is the problem size so we can predict it
- Want to factor out machine details as scaling factors
- Worst case, best case, average case

Algorithm Runtime

```
search(A, key)
1.  for i ← 1 to length[A]
2.      if A[i]=key
3.              then return i
```

Searches for key in array A and returns the index of key

Best Case Algorithm Runtime

search(A, key)	cost	times
1. for i $\leftarrow$ 1 to length[A]	c_1	1
2. if A[i]=key	c_2	1
3. then return i	c_3	1

$T(n)=c_1+c_2+c_3$

Worst Case Algorithm Runtime

search(A, key)	cost	times
1. for i $\leftarrow$ 1 to length[A]	c_1	n
2. if A[i]=key	c_2	n
3. then return i	c_3	1

$T(n) = n(c_1+c_2)+c_3$

Average Case Algorithm Runtime

search(A, key)	cost	times
1. for i ← 1 to length[A]	c_1	n/2
2. if A[i]=key	c_2	n/2
3. then return i	c_3	1

$T(n) = n/2(c_1+c_2)+c_3$

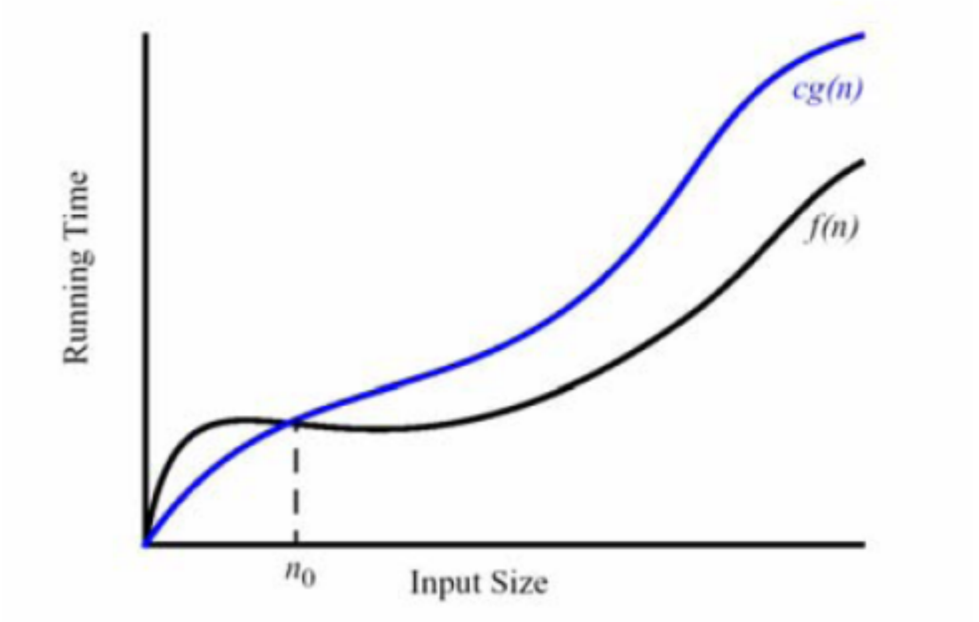
Asymptotic Notation

- The coefficients c_1, c_2,... depend on details of the machine
- Typically we just care about how fast the runtime grows with increasing input size
  - Coefficients aren't important
  - Lower order terms aren't important

Big-O Notation

- Given functions f(n) and g(n), we say that f(n) is O(g(n)) if there are positive constants c and n_0 such that $f(n) \leq cg(n)$ for $n \geq n_0$.
- Informally, if f(n) is O(g(n)), f(n) grows no faster than g(n)

Big-O Illustrated



- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that:

$$f(n) \leq cg(n) \quad \text{for } n \geq n_0$$

Big-O Notation for Polynomials

- If $f(n)$ is a polynomial, then $f(n)$ is $O(n^d)$ where d is the polynomial degree of $f(n)$
  - Drop lower-order terms
  - Drop constant factors
- Example
  - $3n^2+2n$ is $O(n^2)$

Other Notations

- big-Omega (lower bound)
  - f(n) is $\Omega(g(n))$ if there are constants $c>0$ and $n_0 \geq 1$ such that $f(n) \geq cg(n)$ for $n \geq n_0$
- big-Theta (tight bound)
  - f(n) is $\Theta(g(n))$ if there are constants $c>0$, $c'>0$, and $n_0 \geq 1$ such that $cg(n) \leq f(n) \leq c'g(n)$ for $n \geq n_0$
- little-oh (strict upper bound)
  - f(n) is $o(g(n))$ if for any constant $c>0$ there is a constant $n_0 \geq 0$ such that $f(n) \leq cg(n)$ for $n \geq n_0$
- little-omega (strict lower bound)
  - f(n) is $\omega(g(n))$ if for any constant $c>0$ there is a constant $n_0 \geq 0$ such that $f(n) \geq cg(n)$ for $n \geq n_0$