COMP 550: Algorithms & Analysis

Tues/Th 9:30am - 10:45am (FB 007)

On sakai.unc.edu, COMP 550

Stephen Pizer

SN 221-2, 590-6085

pizer@cs.unc.edu; preface subject line with "COMP 550"

Office Hours: Before/after class,
Weds 2:30-3:30, or by appointment

COMP 550 Admission Requirements

- The prerequisites of COMP 410 (Data Structures) and a course in Discrete Mathematics or Discrete Structures are real
- For credit students only
- If you are waiting for a for-credit slot to open up, put your name on the waiting list
  - If you are a Comp. Sci. grad student, see me to discuss

COMP 550 Details

- Homework: Around 7 assignments
  - You may work together on designs, but the final algorithms and the math must be done by yourself
- Quizzes: Around 4
- Midterm and final exams
- Honor code: Give credit where credit is due
  - Many of these slides are derived from those of MC Lin
- TA: TBA

Textbook & References

- *Introduction to Algorithms*, 3rd Ed. by Cormen, Leiserson, Rivest & Stein, MIT Press, 2009

OTHER REFERENCES --

- *The Design and Analysis of Computer Algorithms*, by Aho, Hopcroft and Ullman
- *Algorithms*, by Sedgewick
- *Algorithmics: Theory & Practice*, by Brassard & Bratley
- *Writing Efficient Programs*, by Bentley
- *The Science of Programming*, by Gries
- *The Craft of Programming*, by Reynolds

Solving a Computational Problem

- **Problem definition & specification**
  - **specify input, output and constraints**
- **Algorithm analysis & design**
  - **devise a correct & efficient algorithm**
  - **Analyze its efficiency**
- Implementation planning
- Coding, testing and verification

Primary Focus

Develop thinking ability

- **formal thinking (proof techniques & analysis)**
- **problem solving skills (algorithm design and application)**

What Will We Be Doing

- Devise algorithms for solving interesting problems!
- Analyze their runtime performance!
- Study core algorithms! and associated data structures
- Learn algorithmic design techniques!
- Applications in real-world problems

Goals

- Be very familiar with a collection of *core algorithms*
- Be fluent in *algorithm design paradigms*: divide&conquer, greedy algorithms, randomization, dynamic programming & approximation methods
- Be able to *analyze* correctness and runtime performance of a given algorithm
- Be familiar with *inherent complexity* (lower bounds & intractability) of some problems
- Be familiar with advanced data structures
- Be able to *apply* techniques in practical problems

My Assumptions About Your Background

- Comparison sort algorithms: merge, heap, quick
- Data structures: trees, graphs, lists, stacks, queues
- Definition of O(f(n))
- Can do proofs
  - of theorems
  - of program correctness
- If you do not know what a matrix and matrix multiplication is, learn it from the web, e.g., Wiki or http://www.mathsisfun.com/algebra/matrix-multiplying.html

Course Overview

Introduction to algorithm design, analysis and their applications

- Algorithm analysis
- Some advanced data structures
- Sorting & ordering without comparisons
- Algorithm design paradigms

Algorithm Analysis

- Asymptotic Notation
- Recurrence Relations
- Probability & Combinatorics
- Proof Techniques
- Inherent Complexity

Advanced Data Structures

- Balanced trees
- Graphs

Sorting & Ordering

- Comparison sorts you know
  - Merge sort
  - Heapsort
  - Quicksort
- Linear-Time Sort
  - bucket sort
  - counting sort
  - radix sort
- Selection

Algorithmic Design Paradigms

- Divide and Conquer
- Dynamic Programming
- Greedy Algorithms
- Graph Algorithms
- Randomized Algorithms
- Approximation Methods (if time allows)

Prerequisites

- The materials in the following chapters. Quick reviews, at most, in lecture may be given to refresh your memory.
  - Chapter 2.2, Merge-sort algorithm on pg. 34, chapter 3 through middle of pg 48, chapters 6 through 6.4, 7.1 10, 11.1-11.2

Course Roadmap

- Algorithmics basics (1.5)
- Divide and conquer (3)
- Efficiency analysis techniques
- Randomized algorithms (3)
- Sorting and selection (6)
- Balanced trees (3)
- Graph algorithms (3)
- Greedy algorithms (4)
- Dynamic programming (2)
- Special topics (1-2)

Algorithmics Basics (1.5)

- Introduction to algorithms, complexity and proof of correctness (Chapters 1 & 2)
- Asymptotic Notation (Chapter 3.1)

GOAL: Know how to write a problem spec(s), what a computational model is, and how to measure efficiency of an algorithm. Distinguish between upper / lower bounds for an algorithm & what they convey. Be able to prove the correctness of an algorithm and establish computational complexity.

Divide-and-Conquer (3)

- Designing algorithms (Chapter 2.3)
- Recurrences (Chapter 4)
- Merge-sort and quicksort analysis (Chapters 2.3 and 7)

GOAL: Know when the divide-and-conquer is an appropriate paradigm, the general structure of such an algorithm and its variants. Be able to characterize their complexity using techniques for solving recurrences. Memorize the common case solutions for recurrence relations.

Randomized Algorithms (3)

- Probability & Combinatorics (Chapter 5)
- Quicksort (Chapter 7)
- Hash Tables (Chapter 11)

GOAL: Know sample space, simple event, compound event, independent event, conditional probability, random variables, probability distribution function, expectation & variance well. Be able to apply them in the design and analysis of randomized algorithms and data structures. Understand the average case behavior vs. worst-case, esp. in sorting & hashing.

Sorting & Selection (6)

- Review merge-sort (Ch. 2), heapsort (Ch. 6), Quicksort (Ch. 7)
- Bucket Sort, Radix Sort, etc. (Chapter 8)
- Selection (Chapter 9)

GOAL: Understand the performance of sorting algorithms listed above and when to use them. Know why sorting is important, what one can do with binary heaps and why linear-time median finding is useful.

Balanced Trees (3)

- Binary Search Trees (Chapter 12)
- Red-Black Trees (Chapter 13)

GOAL: Know the fundamental ideas behind maintaining balance in insertions/deletion. Be able to use these ideas in other balanced tree data structures. Understand what they can represent and why they are useful. Know the special features of the trees listed above.

Graph Algorithms (3)

- **Basic Graph Algorithms (Chapter 22)**

GOAL: Know how graphs arise, their definition and implications. Be able to use adjacency matrix representation of a graph and the edge list representation appropriately. Understand and be able to use "cut-and-paste" proof techniques as seen in the basic algorithms (DFS, BFS, topological sort, connected components).

Greedy Algorithms (4)

- Greedy algorithms (Chapter 16)
- Minimum spanning trees (Chapter 23)
- Shortest paths (Chapter 24)

GOAL: Know when to use greedy algorithms and their essential characteristics. Be able to prove the correctness of an greedy algorithm in solving an optimization problem. Understand where minimum spanning trees arise in practice, how do union-by-rank and path compression heuristics for dynamic sets improve performance & make them effective.

Dynamic Programming (2)

- **Dynamic programming (Chapter 15)**

GOAL: *Know what problem characteristics make it appropriate to use dynamic programming and its difference from divide-and-conquer. Be able to move systematically from one to the other.*

Course Work & Grades

- **Homework: 25%**
  (total of 7, mostly design & analysis)
- **Quizzes: 20%**
  (total of 3-5, basic materials)
- **Exams: 55%**
  - **Midterm Exam: 1/3 of exam grade, but will be dropped in favor of final exam grade if final grade is higher**
  - **Final Exam: 2/3 of exam grade, but will count the same as the midterm grade if the midterm grade is higher**
- **Active Class Participation: 5% bonus**

Examinations

- **Quizzes: 3-5 throughout the semester**
  - **First will be in class on 26 September**
- **Midterm: In class, October 2013**
- **Final: December 10, 2013, 8am**

All closed book

Classes on Video

- Lectures for Th 9/19 and T 9/24 will be on video, available on the web, and also played in class

Homework Assignments

- Due at the beginning of each class on the due date given
- No late homework will be accepted
- Lowest score will be dropped
- Can discuss in group, but must write/formulate solutions alone
- Be neat, clear, precise, formal
  - you'll be graded on correctness, simplicity, elegance & clarity

Communication

- Visit instructor & TAs during office hours, by appointment, or email correspondence
- All lecture notes and most of handouts are posted at the course sakai website:
- Major messages are notified by email alias
- Student grades are posted periodically

Basic Courtesy

- Write your assignments neatly & formally
- Please do not read newspaper & other materials or surf the web in class
- When coming to the class late or leaving early, please take an aisle seat quietly
- Remain quiet, except asking questions or answering questions posed by instructors
  – no whispers or private conversation

THANK YOU!!!

How to Succeed in this Course

- Start early on *all* assignments. *DON'T* procrastinate.
- Complete all reading before class.
- Participate in class.
- Think in class.
- Review after each class.
- Be formal and precise on all problems sets and in-class exams

Weekly Reading Assignment

Chapters 1, 2 and 3

(Textbook: CLRS)

Review of Three Comparison Sorts (Assume n>1)

- Heapsort (A,n) invoked by heapsort (A,n)
  - Heapify(A,n) /* Heap has A[i]≥A[2i] and A[i] ≥ A[2i+1]
  - For i= n by -1 to 2
    - Swap A[1] and A[i]; Reheapify(A,i-1)
- Mergesort (A,i,j) invoked by mergesort(A,1,n)
  - If i=j, return
  - Mergesort(A,i,⌊(i+j)/2⌋); Mergesort(A,1+⌊(i+j)/2⌋); Merge(A,i,j)
- Quicksort (A,i,j) invoked by quicksort(A,1,n)
  - If i<j
    - q = partition slot of slots i through j of A under comparison with A[j] s.t. A[i] through A[q-1] are ≤ A[q] and A[q+1] through A[j] are ≥ A[q]
    - quicksort(A,i,q-1); quicksort(A,q+1,j)

Algorithms

- A tool for solving a well-specified computational problem

- Example: sorting

  input: A sequence of numbers

  output: An ordered permutation of the input

  issues: correctness, efficiency, storage, etc.

Analyzing Algorithms

- **Assumptions**
  - generic one processor, random access machine
  - running time (others: memory, communication, etc)
- **Worst Case Running Time:** the *longest* time for *any* input size of n
  - upper bound on the running time for any input
  - in some cases like searching, this is close
- **Average Case Behavior:** the *expected* performance averaged over *all* possible input
  - it is generally better than worst case behavior
  - sometimes it's roughly as bad as worst case

A Simple Example

INPUT: a sequence of n numbers

OUTPUT: the smallest number among them

1. $x \leftarrow$ T[1]
2. for $i \leftarrow$ 2 to n do
3. if T[i] < x then $x \leftarrow$ T[i]

* Performance of this algorithm is a function of n.

Runtime Analysis

- Elementary operation: an operation whose execution time can be bounded above by a constant depending on implementation used.
- Assume all elementary operations can be executed at unit cost. This is not true, but they are within some constant factors of each other. We are *mainly* concerned about the *order of growth*.

Order of Growth: Basic Ideas

- For very large input size, it is *order of growth* that matters asymptotically.
- We can ignore the *lower-order terms*, since they are relatively insignificant for very large n. We can also ignore *leading term's constant coefficients*, since they are not as important for the rate of growth in computational efficiency for very large n.
- Higher order functions of n are normally considered less efficient.

Comparisons of Algorithms

- **Multiplication**
  - classical technique: $O(nm)$
  - divide-and-conquer: $O(nm^{\ln 1.5}) \sim O(nm^{0.59})$

  For operands of size 1000, it takes 40 & 15 seconds respectively on Cyber 835.
- **Sorting**
  - insertion sort: $\Theta(n^2)$
  - merge sort: $\Theta(n \log n)$

  For 10^6 numbers, insertion sort takes 5.56 hrs on a supercomputer using machine language and 16.67 min on a PC using C/C++ with merge sort.

Why Order of Growth Matters

- Computer speeds double every two years, so why worry about algorithm speed?
- When speed doubles, what happens to the amount of work you can do?

Effect of faster machines

Ops/sec:	1M	2M	Gain
n*n alg	1000	1414	1.4
n log n alg	62700	118600	1.9

The number of items that can be sorted in one second using an algorithm taking exactly n^2 time as compared to one taking $n \lg n$ time, assuming 1 million and 2 million operations per second. Notice that, for the $n \lg n$ algorithm, doubling the speed almost doubles the number of items that can be sorted.