CSCI S-Q
Lecture #1
6/22/98

• Introduction
• Course Overview
• Logistics & Philosophy
• What is an Algorithm?
• big-O Notation
• Introduction to ADTs
Handouts

• Syllabus
• Coursebook, Chapters 1-5
• Assignments 1 and 2
• Questionnaire (return at the end of the break - if you don’t have your email address yet, send it to the course account ASAP)
• Excerpts from "Academic honesty and Rules and Regulations Concerning Computer Use"
• UNIX Tools for S-Q
What is the Goal of this Course?

• Working knowledge of Data Structures and Algorithms
• Exposure to many of the ideas behind good data structures and algorithms
• Techniques to analyze data structures and algorithms in terms of
  • performance - running time, memory usage, etc.
  • generality - useful for solving just one problem, or a class of problems
  • correctness
• Enhanced ability to “think algorithmically”

What is this course NOT about

• Programming
  • This course will not teach you C. We assume that you already know C.
• Software Engineering
  • The programs you will write will be small and structurally simple
Mechanics

• Friday June 26 - last day to drop/change courses

• Office hours
  • Monday night after class
  • This week only - Wednesday night after class

• Textbook: "Data Structures and Algorithm Analysis in C (2nd Edition)" by Mark Allen Weiss, available at the Coop, in the textbook section (open until 9:00pm this week). If they run out, either order it from them, or try:
  • The Coop - Main book building
  • Quantum Books - Kendall Square
  • SoftPro - Burlington
  • other technical bookstores

• Course Web page: http://www.fas.harvard.edu/~libsq

• Course Accounts - log in to fas.harvard.edu as user register (no password required). You must have an FAS account to do the coursework.
Assignments & Exams

• For Undergraduates
  • 5 Assignments, each worth 10% of your grade (50% total)
  • 2 Hourly exams, each worth 10% of your grade (20% total)
  • 1 Final exam (3 hours), worth 30% of your grade

• For Graduate students
  • 5 Assignments, each worth 10% of your grade (50% total)
  • 2 Hourly exams, each worth 9% of your grade (18% total)
  • 1 Final exam (3 hours), worth 27% of your grade
  • 1 Final paper worth 5% of your grade

• All grades are numeric
Assignments

• A mix of programming and written questions
• May require significant thought, so Don’t Wait Until the Last Minute!
• No collaboration is permitted. Do not discuss the assignments except with the teachers.
• Code must be written in ANSI C, and must compile and run on the course "ice" machines.
• Assignments are “submitted” on-line

Assignment Philosophy

• Engage your brain.
• Hard, but manageable.
• If too hard, tell us. Ask for help.
• If too easy, tell us. Ask for harder problems.
• Mail to libsq@fas.harvard.edu
Late Assignments

• Deduction of 10% per day (1 minute late is 1 day late)
• After drop-dead date, assignments will not be accepted without an extension.
• Extensions are granted for cases of unforeseen emergencies or other situations that would prevent you from doing class work.
  • Send email to libsq@fas.harvard.edu (or call) to request an extension.
• 3 “grace days” can be used on any assignment (but do not override the drop-dead dates).

Hourlies and the Final Exam

• Closed book - no books, notes, calculators, slide rules, etc…
• Focus on understanding of concepts - not “puzzles” like the homework.
• “practice exams” will be passed out a week or so before the exams.
Lecture Philosophy

- We prefer discussion
- Do the reading before class, and be prepared to talk about it! (At least skim it!)
What is an Algorithm?

• Weiss:

  An algorithm is a clearly specified set of simple instructions to be followed to solve a problem.

• Key attributes:
  a. **Precise** - Each step is simple, clear, unambiguous
  b. **Correct** - Solves the problem
  c. **Well-defined input and output**
    
    For example - addition of numbers
  d. **Finite** - the solution will be arrived at in finite time. (Also, the set of instructions is finite)

• The *specification* of an algorithm should include a description/limits of other effects - what does it do besides “solving the problem”?
What is a Data Structure?

• A way of structuring or organizing data, typically designed to allow/facilitate manipulation of the data

• Example - telephone book.

• Data Structures and Algorithms are sometimes difficult to separate - an algorithm may be designed to take advantage of a particular data structure, or vice versa.
Analysis of Algorithms

• Several important properties of algorithms -
  a. How long does it take to solve the problem?
  b. Does the length of time required to solve the problem depend upon the input, and if so, in what way?
  c. Similar questions about memory use, disk use, etc.

• The purpose of algorithm analysis is to find these properties, and help the designer choose among different algorithms that solve the same problem.

• Helps choose the right algorithm for the specific task.

Growth Rates

• How does the growth of the consumption of resources of an algorithm (a.k.a. the complexity of the algorithm) change as the “size” of the algorithm’s input (or perhaps output) increases?
  • time complexity - how much time is required.
  • space complexity - how much space.
What is the Size of a Problem?

• Examples:
  • Addition - the number of digits
  • Greeting - the number of people
  • Sorting - the number of elements

• As the size grows, the growth rate becomes the most important aspect of the complexity of an algorithm. For this reason a notation for expressing growth rates is used, called Big-O (or Big-Oh) notation.
Definitions

• Big-O

\[ T(N) = O(f(N)) \text{ if there are positive constants } c \text{ and } n_0 \text{ such that } T(N) \leq cf(N) \text{ when } N \geq n_0. \]

• Big-Omega

\[ T(N) = \Omega(f(N)) \text{ if there are positive constants } c \text{ and } n_0 \text{ such that } T(N) \geq cf(N) \text{ when } N \geq n_0. \]

• Big-Theta

\[ T(N) = \Theta(f(N)) \text{ if and only if } T(N) = O(f(N)) \text{ and } T(N) = \Omega(f(N)). \]

• There are others, but Big-O is where we'll concentrate.
Big-O Conventions

• Use the "smallest" function you can find

• Omit constant factors

• Omit all "lower order" terms. (only use the term that grows most quickly)

  e.g. Not $O(3n^2 + n)$, but $O(n^2)$

• Be careful with usage: remember that the big-O of a function is always is on the right-hand side of the =

  $5n^2 + 3n + 1 = O(n^2)$ \textbf{RIGHT}

  $O(n^2) = 5n^2 + 3n + 1$ \textbf{WRONG}

big-O notation is actually a form of set notation. It would be more proper to write big-O relations like

$$6n^2 \in O(n^2)$$

but for historical reasons, the = symbol is used instead.
Finding big-O

The definitions for big-O (and the other relations) can be used to find the big-O of any function, given enough time, energy, and scrap paper. For many common kinds of functions, however, much has already been discovered, and there are useful rules and guidelines.

Addition and multiplication:
If \( T_1(N) = O(f_1(N)) \) and \( T_2(N) = O(f_2(N)) \), then:

a. \( T_1(N) + T_2(N) = O(\max(f_1(N), f_2(N))) \)

b. \( T_1(N)T_2(N) = O(f_1(N)f_2(N)) \)

Transitive property:
If \( T_1(N) = O(T_2(N) \) and \( T_2(N) = O(T_3(N)) \), then it is also true that \( T_1(N) = O(T_3(N)) \)

Polynomials:
If \( T(N) \) is a polynomial of degree \( M \), then \( T(N) = \Theta(N^M) \)

There are more rules and examples in the Weiss text.

Computing the big-O of recursive functions can be complicated-- we won’t do much in this area, but we will study some of the important results. (Recursive functions have been investigated extensively, but some problems still remain unsolved...)
Abstract Data Types

• What is an Abstract Data Type (ADT)?

  • A set of well-defined operations, with an interface

  • The operations of an ATD are defined over a particular set of types. This set of types is part of the interface.

  • Sedgewick: "An ADT is a data type (a set of values and a collection of operations on those values) that is accessed only through an interface."

• The Abstraction:

  • What the operations do is well-defined. How the operations are performed (or how the mechanism is implemented) is not specified.

• The Types:

  • The set of types that the operations are defined over is important. Often we are not very concerned with the actual type, although it can be important, i.e. "numbers" vs. "arrays"
ADT Examples

Example - A Simple Array ADT

• Operations:
  • CREATE (length)
  • DESTROY
  • val = GETITH (index)
  • SETITH (index, val)

• Set of Types:
  • length - must be a positive integer
  • index - must be a positive integer
  • val - any type

Example - Simple Arithmetic on Numbers

• Operations: +, -, *, /

• Set of Types: numeric: int, double, rational numbers, etc.
Why Study ADTs?

• Weiss - goal of ADTs is modularity; implement once, and then reuse as necessary. There’s more to it than that however.

• From the perspective of the designer of the data structures and algorithms, the abstract interface allows us to reason about key properties without considering any details of the implementation.