CS 483 - Data Structures and Algorithm Analysis

Lecture I: Chapter 1

R. Paul Wiegand

George Mason University, Department of Computer Science

January 25, 2006

Outline

Personal & Course Introduction

- Personal Introduction:
  - Current position & Research interests
  - Industry experience
  - Personal expectations
- Course Introduction:
  - Course title & topic
  - Degree requirement & Pre-req's
  - Hand out info sheet
- Course Syllabus:
  - Office hours and contact info
  - Grading, projects, & homeworks
  - Cheating
  - Course schedule
- How to succeed:
  - Be curious & motivated
  - Read!! (BEFORE class)
  - Build good habits that work for you
  - Ask for help

Syllabus: `http://www.cs.gmu.edu/~pwiegand/cs483`

Motivating the Course

- Why this course matters:
  - Forrest for the trees
  - Making educated & informed decisions
  - Need as designer *AND* implementor
  - Engineer versus technician
- Personal reflections:
  - "Don't know Big-O stuff!"
  - "The JDK comes with a SORT routine..."
  - Etc.
- Key ideas (from Henry Hamburger)

Computational Problems

What is a *computational problem*?

Computational Problems

What is a *computational problem*?

- Problem statement
  - The *statement of a problem* specifies in general terms the relationship between input and output
  - Example Sort a set of numbers in non-decreasing order (*sorting problem*)
    Input: $\langle a_1, a_2, \ldots a_n \rangle$
    Output: $\langle a'_1, a'_2, \ldots a'_n \rangle : a'_1 \leq a'_2 \leq \cdots \leq a'_n$
- Problem instance
  - A *problem instance* consists of the input, satisfying whatever constraints are imposed by the problem statement) needed to compute a "solution" to the problem.
  - Example problem instance:
    Input: $\langle 4, 6, 7, 1, 9, 3, 8, 10, 5, 2 \rangle$
    Output(solution): $\langle 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 \rangle$
- Are problems inherently *hard* (or harder than others)?

Algorithms

What is an *algorithm*?

Algorithms

What is an *algorithm*?

- Algorithm
  - A recipe, a list of instructions , a transformation of data ... ?

Algorithms

What is an *algorithm*?

- Algorithm
  - A recipe, a list of instructions , a transformation of data ... ?
  - Cormen *et al.*: An *algorithm* is any well-defined computation procedure that takes some value, or set of values, as input and produces value, or set of values, as output.
  - Levitin: An *algorithm* is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.
  - In a sense, algorithms are *"procedural solutions to problems"*

Algorithms

What is an *algorithm*?

- Algorithm
  - A recipe, a list of instructions , a transformation of data ... ?
  - Cormen *et al.*: An *algorithm* is any well-defined computation procedure that takes some value, or set of values, as input and produces value, or set of values, as output.
  - Levitin: An *algorithm* is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.
  - In a sense, algorithms are *"procedural solutions to problems"*
- Important point about algorithms
  - Unambiguous instructions
  - Input range specified carefully
  - Multiple representations for same algorithm
  - Multiple algorithms for solving the same problem
  - Different alg. based on different ideas with different trade-offs

Example: Greatest Common Divisor

Input: $m, n \in \mathbb{N}$, where $(m \geq 0 \wedge n > 0) \vee (m > 0 \wedge n \geq 0)$
Output: Largest integer that divides both m and n evenly

EUCLID(m, n)

```
while n ≠ 0 do
    r ← m mod n
    m ← n
    n ← r
return m
```

CONSECUTIVEINTEGER(m, n):

```
step-1:  t ← min{m, n}
step-2:  if m/t ∈ ℕ⁺, goto step-4
step-3:  if n/t ∈ ℕ⁺, return t
step-4:  t ← t − 1, goto step-2
```

- Are these algorithms guaranteed to stop?
- Are there different input restrictions?
- Look over the "middle-school method" in the book ...

Steps for Designing Algorithms

- Understand the problem
- Assess computational resources (memory, speed, etc.)
- Decide between an *exact* or *approximate* algorithm
- Choose appropriate *data structures*
- Specify an algorithm in pseudo-code
- Prove *correctness*
- Analyze the algorithm
- Implement & test the algorithm

Issues Surrounding the Design of Algorithms

- An algorithm is *correct* if it produces the required result for *every* legitimate input
- An *exact* algorithm produces solutions to problems that are exactly correct.
- An *approximate* algorithm produces solutions to problems that are approximately correct.
- A *data structure* is a way to store and organize (related) information in order to facilitate access and modification.
- Algorithm analysis:
  - Efficiency (time, space): how algorithms *scale* wrt input size
  - Simplicity
  - Generality
    - Type of problems solved
    - Range of inputs accepted

Sorting

- Arrange a set of values in a total or partial ordering
- Often make use of a *key* for sorting more complicated data
- With key-comparison based sorts, cannot do better than $n \lg n$ time
- Sorting algorithms are *stable* if given two elements with equal key values at positions i and j such that $i < j$, after the sort they will appear in positions i' and j' such that $i' < j'$.
- Sorting algorithms are called *in place* sorts if they do not require more than a constant amount of memory beyond what is stored in the list.

Searching & String Processing

- Searching
  - Find a given value, called a *search key*, in a set of values
  - A variety of algorithms exist (sequential search, binary search, etc.)
  - Sometimes data are stored in data structures that make them more conducive for searching (hash maps, red-black trees, etc.)
  - Engineers have to pay attention to applications where the underlying data may change frequently relative to the number of searches.
- String Processing
  - A *string* is a sequence of characters from some well-defined alphabet (e.g., binary strings)
  - Large class of problems dealing with the handling of strings
  - An example problem is *string matching*: Find the positions of a substring in a master string.

Graph & Combinatorial Problems

- Graph Problems
  - A *graph* is a collection of vertices, some of which are connected by edges
  - Traditional examples: graph traversal, finding shortest-path, finding minimum spanning tree, etc.
  - Can be computationally very hard
  - Examples of hard graph problems:
    - Traveling salesperson problem
    - Graph coloring problem
- Combinatorial Problems
  - Problems in which one must find a combinatorial object that satisfies certain constraints and has some desired property
  - Tend to be the hardest types of computational problems
  - Many graph problems *are* combinatorial problems

Graph & Combinatorial Problems

- Graph Problems
  - A *graph* is a collection of vertices, some of which are connected by edges
  - Traditional examples: graph traversal, finding shortest-path, finding minimum spanning tree, etc.
  - Can be computationally very hard
  - Examples of hard graph problems:
    - Traveling salesperson problem
    - Graph coloring problem

Find the shortest tour that visits all connected vertices exactly once

- Combinatorial Problems
  - Problems in which one must find a combinatorial object that satisfies certain constraints and has some desired property
  - Tend to be the hardest types of computational problems
  - Many graph problems *are* combinatorial problems

Graph & Combinatorial Problems

- Graph Problems
  - A *graph* is a collection of vertices, some of which are connected by edges
  - Traditional examples: graph traversal, finding shortest-path, finding minimum spanning tree, etc.
  - Can be computationally very hard
  - Examples of hard graph problems:
    - Traveling salesperson problem
    - Graph coloring problem

Assign the smallest number of colors to vertices of a graph so that no two adjacent vertices are the same color

- Combinatorial Problems
  - Problems in which one must find a combinatorial object that satisfies certain constraints and has some desired property
  - Tend to be the hardest types of computational problems
  - Many graph problems *are* combinatorial problems

Geometric & Numerical Problems

- Geometric Problems
  - Geometric problems deal with geometric objects (e.g., points, lines, polygons, etc.)
  - For example:
    - Closest-pair problem
    - Convex hull problem
  - These are *different* than graph problems!

Geometric & Numerical Problems

- Geometric Problems
  - Geometric problems deal with geometric objects (e.g., points, lines, polygons, etc.)
  - For example:
    - Closest-pair problem
    - Convex hull problem
  - These are *different* than graph problems!

Given n points, find the pair of points with the minimum distance between them

Geometric & Numerical Problems

- Geometric Problems
  - Geometric problems deal with geometric objects (e.g., points, lines, polygons, etc.)
  - For example:
    - Closest-pair problem
    - Convex hull problem
  - These are *different* than graph problems!

Given n points in a set, find the smallest convex polygon that contains all these points.

Geometric & Numerical Problems

- Geometric Problems
  - Geometric problems deal with geometric objects (e.g., points, lines, polygons, etc.)
  - For example:
    - Closest-pair problem
    - Convex hull problem
  - These are *different* than graph problems!
- Numerical Problems
  - Problems involving continuous mathematical objects
  - For example:
    - Solving systems of equations
    - Computing derivatives & definite integrals
    - Optimizing numerical functions, etc.

Linear Data Structures: Elementary data structures

The following are two elementary data structures useful for produce more abstract linear data structures called *lists* (a finite sequence of data items)

array — A sequence of n items of the same data type stored contiguously in memory and accessible using an *index*

- Pre-established, fixed size
- Constant time access, insertion and deletion can be challenging
- Example: bit string, 1 0 0 1 1 0 1

linked list — A sequence of zero or more *nodes*, each containing data and *pointer*(s) to other node(s)

- Not necessarily fixed in size
- Linear time access, insertion and deletion are simpler
- Linked lists can be *single-linked* or *doubly-linked*
- Linked lists can have a *header*, which stores useful information (e.g., length)

Linear Data Structures: Advanced data structures

The following are two special types of lists.

stack — A list in which insertions and deletions can only be done at one end

- LIFO – last in, first out
- May be implemented by an array or a linked list
- Basic operations: PUSH,POP

queue — A list in which elements are accessed & deleted from one end (front) and inserted at the other end (rear)

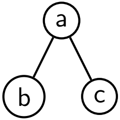
- FIFO – first in, first out
- May be implemented by an array or a linked list
- Basic operations: ENQUEUE, DEQUEUE
- Position in a queue can be determined using a *priority* (priority queues)

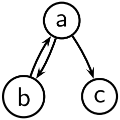
Graphs: Simple

- Graphs are collections of points called *vertices* and line segments, called *edges*, connecting (some of the) vertices
- Formally: $G := \langle V, E \rangle$, where V is a finite set of labels corresponding to vertices (e.g., $V := \{a, b, c\}$) and E is a finite set of pairs of these items (e.g., $E := \{(a, b), (a, c)\}$
- *Undirected graph:* Edges are unordered, i.e., $(a, b) = (b, a)$
- *Directed graph:* Edges are ordered and thus imply a direction

Graphs: Simple

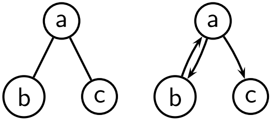
- Graphs are collections of points called *vertices* and line segments, called *edges*, connecting (some of the) vertices
- Formally: $G := \langle V, E \rangle$, where V is a finite set of labels corresponding to vertices (e.g., $V := \{a, b, c\}$) and E is a finite set of pairs of these items (e.g., $E := \{(a, b), (a, c)\}$
- *Undirected graph:* Edges are unordered, i.e., $(a, b) = (b, a)$
- *Directed graph:* Edges are ordered and thus imply a direction





Graphs: Simple

- Graphs are collections of points called *vertices* and line segments, called *edges*, connecting (some of the) vertices
- Formally: $G := \langle V, E \rangle$, where V is a finite set of labels corresponding to vertices (e.g., $V := \{a, b, c\}$) and E is a finite set of pairs of these items (e.g., $E := \{(a, b), (a, c)\}$
- *Undirected graph:* Edges are unordered, i.e., $(a, b) = (b, a)$
- *Directed graph:* Edges are ordered and thus imply a direction



- *Complete*—every pair of vertices is connected by an edge
- *Dense*—most vertices are connected
- *Sparse*—few vertices are connected

Graphs: Representation

adjacency matrix — Enumerate vertices in $\{1 \ldots n\}$, create an $n \times n$ matrix of boolean values indicating whether an edge exists between the specified vertices

	a	b	c
a	0	1	1
b	1	0	0
c	1	0	0

- Undirected graphs result in symmetric matrices
- Easily determine if an edge exists, requires space
- Good for dense graphs

Graphs: Representation

adjacency matrix — Enumerate vertices in $\{1 \ldots n\}$, create an $n \times n$ matrix of boolean values indicating whether an edge exists between the specified vertices

	a	b	c
a	0	1	1
b	1	0	0
c	1	0	0

- Undirected graphs result in symmetric matrices
- Easily determine if an edge exists, requires space
- Good for dense graphs

adjacency list — Create a linked list for each vertex containing the vertices to which that vertex is connected

$\boxed{a} \rightarrow b \rightarrow c$

$\boxed{b} \rightarrow a$

$\boxed{c} \rightarrow a$

- Somewhat more difficult to determine edge existence, more compact in space
- Good for sparse graphs

Graphs: Weights, Paths, & Cycles

- We refer to a *weighted graph* when there are costs or values associated with the edges in a graph
  - Adjacency matrix: Use numeric values in cells of the matrix, special character for no-edge (e.g., ∞)
  - Adjacency list: Attach values to nodes in the linked list

Graphs: Weights, Paths, & Cycles

- We refer to a *weighted graph* when there are costs or values associated with the edges in a graph
  - Adjacency matrix: Use numeric values in cells of the matrix, special character for no-edge (e.g., ∞)
  - Adjacency list: Attach values to nodes in the linked list
- Properties of graphs:
  - A *path* a sequence of adjacent vertices connected by an edge
  - A path is called *simple* if all edges are distinct
  - Path *length* is the total number of vertices in the sequence
  - A *directed path* is a sequence of vertices in which every consecutive pair of vertices is connected by an edge directed from the vertex listed first the next one
  - A *graph* is *connected* if a path exists for every pair of vertices
  - A *cycle* is a simple path of positive length that starts and ends with the same vertex
  - A graph is said to be *acyclic* if it admits no cycles

Graphs: Trees

What is a tree? What is a forrest?

Graphs: Trees

A (free) *tree* is a connected, acyclic graph. A *forest* is multiple trees, or an unconnected, acyclic graph.

Graphs: Trees

A (free) *tree* is a connected, acyclic graph. A *forest* is multiple trees, or an unconnected, acyclic graph.

- $|E| = |V| - 1$
- For every two vertices, there's always *exactly one* simple path between them
- $\therefore$ we can select an arbitrary vertex to be the *root*
- For any $v \in T$, all vertices on the path between the root and v are called *ancestors*
- The last edge on that path before v is called the *parent*, v is the *child* of that node, etc.
- A vertex with no children is called a *leaf*
- A vertex with all its descendants is called a *subtree*
- The *depth* v is the length of the simple path from the root to v
- The *height* of a tree is the length of the longest simple path from

Sets & Dictionaries

What is a set?

Sets & Dictionaries

A *set* is an unordered collection (possibly empty) of distinct items.

Sets & Dictionaries

A *set* is an unordered collection (possibly empty) of distinct items.

- We can implement a set as a bit vector over the *universal set*
- We can implement a set with a list structure (with insertion constraints)
- A *multiset* or *bag* is a set without the uniqueness constraint (an unordered collection of objects)
- Basic operations of a multiset: SEARCH, INSERT, DELETE
- A basic data structure that accomplishes these operations is a *dictionary*
- Sometimes we need to dynamically partition some n-element set into a collection of disjoint sets.
- Sometimes we need to take the union or intersection of sets

Assignments

- Section 1.1: Problems 5, 7, 9
- Section 1.2: Problems 4, 5, 7
- Section 1.3: Problems 1, 4, 8, 9*
- Section 1.4: Problems 2, 4, 6*, 9

*Challenge problem