Review

A selection of slides from throughout the term
Study guide for the final exam
The take-home final (and other projects) are due
1:00 PM Thursday June 12

Course Goals

- This course was an introduction to the field of computer science
  - what sorts of problems computer scientists study
  - some of the methods we use to solve those problems
- Although it wasn’t a “programming course” I wanted to have lab projects that
  would give people experience with a programming language
  - one of the best ways to understand a problem is to write a program that explores
    different ways of solving it
- The emphasis was on problems and their solutions
  - computer science is all about solving problems
  - it’s more than just “programming”

Course Topics

- Data Representation
- Algorithms
- “Big O” Notation and Performance
- Text Compression (Huffman Codes)
- Error Correction
- Regular Expressions
- Networks
- Encryption
- Relational Databases
- Software Engineering and Human-Computer Interaction
- Genetic Algorithms
- Limits of Computation

There will be no exam questions on voting or voting machines

Summary

- Information is stored in a computer in the form of bits
  - computer scientists are (usually) not concerned about the physical storage medium
  - we only care that something has two states that we can label 0 and 1
- A single piece of data is represented by a sequence of bits
  - \( n \) bits \( \Rightarrow 2^n \) different patterns
  - for integers, \( n \) bits can represent the numbers \( 0 \ldots 2^n - 1 \)
- Ruby has representations for numbers and text
  - Fixnum, Bignum, Float
  - Strings (where individual elements are ASCII characters)

Data is represented in a computer as strings of binary digits

This slide came from the lecture notes in the file named Data

Theme: Electronic Voting
Integers

- The number of bits in a word determines the range of numbers a processor can work on.
  - Integers are stored in words.
  - If a CPU has \( n \)-bit words, there are \( 2^n \) patterns.
- When a number is stored in memory, the bit pattern corresponds to the string of digits in the binary form of the number.
  - Example: the pattern 00100101 represents the number 37.
- The \( 2^n \) patterns in an \( n \)-bit word can represent the numbers from 0 to \( 2^n - 1 \).
  - In a 32-bit system, the numbers range from 0 to 4,294,967,295.

\[
\begin{array}{cccccccc}
2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
2^7 + 2^2 + 2^0 = 128 + 4 + 1 = 133
\end{array}
\]

Strings

- In computer science terminology, a string is a sequence of 0 or more characters.
- Two common representations for strings:
  - Use a special "end of text" marker (e.g., 0) to denote the last character.
  - Prefix the string with a byte that defines how many characters are in the string.

Strings in ASCII:

```
Hello, world!
```

Hexadecimal representation:

```
48 65 6C 6C 6F 2C 20 77 6F 72 6C 64 21
```

Data

See the text for representations of negative integers and real numbers.

Methods

- Most operations on strings are performed by methods.
- To apply a method to an object, write the object's name, a period, and the method name.

```
>> s => "Bonjour"
>> s.length => 7
>> s.reverse => "ruojnoB"
```

There are dozens of methods for the String class.

- Expect problems similar to those in Lab 1: binary representations of numbers and strings, simple operations on those items.

Elements of an Algorithm

- Review: An algorithm is a description of a procedure to solve a problem.
  - Algorithms have well-defined inputs (starting conditions) and outputs (goals).
  - Steps of the procedure must be:
    - Precise (simple and unambiguous).
    - Effective (make progress toward the solution).
    - Practical (can be completed).
  - The process must eventually terminate, otherwise the outputs are not defined.

The idea that problems can be solved by algorithms is the central concept in computer science.
Algorithm / Not an Algorithm

The Sieve of Eratosthenes (cont'd)

We looked at simple algorithms like the Sieve of Eratosthenes and Euclid’s GCD

See also: “sieve of eratosthenes” at Wikipedia (has an animation)

Example: Insertion Sort

Here is a pseudo-code version of insertion sort:

```
input: a list a₀, a₁, ... aₙ₋₁
for j = 1 to n-1
    key ← aⱼ
    i ← j-1
    while i ≥ 0 and aᵢ > key:
        aᵢ₊₁ ← aᵢ
        i ← i-1
        aᵢ ← key
```

... or in an abstract notation known as "pseudo-code"....

This specification is much more concise

Note this algorithm has two iterations ("loops"), one inside the other

these are called nested loops

Insertion Sort

Here is a version of insertion sort in Ruby

```
def isort(a)
    for i in 1..a.length-1
        key = a.slice!(i)
        j = i-1
        while j ≥ 0 and a[j] > key
            j = j-1
            a.insert(j+1,key)
        end
    end
    return a
end
```

Expect problems similar to those in Lab 2: do steps of a simple algorithm (e.g. gcd or sieve), answer questions about the algorithm

... or in a programming language like Ruby

This version takes advantage of Ruby String methods (slice and insert) to remove an item and re-insert it

we could also just translate the pseudo-code directly
Scalability

- Big O notation tells us something about the **scalability** of an algorithm.
  - It gives us an idea of how the algorithm will behave on larger inputs.
- As an example, suppose we have a choice of two algorithms for a problem.
  - Method A has a nested loop, and the key step inside the inner loop is very simple.
  - Method B does not have an inner loop, but it requires us to do some other steps before the loop and the key step inside the loop is 20 times more complicated.
- We do some algebra to come up with equations that tell us roughly how many steps are required for each algorithm:
  
  \[ f_A(n) = \frac{n(n-1)}{2} = n^2 - n \]
  
  \[ f_B(n) = 100 + 20n \]

Note: Algorithm A is \(O(n^2)\) and algorithm B is \(O(n)\).

“Big O” notation describes the scalability of an algorithm.

Performance

- We talked about various definitions of performance (wall-clock time, CPU time, ...) and tools for measuring performance.

\(O\) notation and performance measurement were part of Lab 3.
More Compact Codes

- A special code for protein sequences would need 5 bits per letter
  - there are 20 amino acid letters
  - \(2^4 = 16\), so 4 bits don’t provide enough combinations
  - \(2^5 = 32\) combinations
    \[ k = \lceil \log_2 20 \rceil = \lceil 4.3219 \rceil = 5 \]
- \(A = 00000, C = 00001, D = 00010\), etc

We looked at different methods for text compression (representations for letters that will reduce the size of a file)

Huffman Tree

- The codes on the previous slide were defined by a Huffman tree
- A Huffman tree is a type of binary tree
  - circles represent nodes
  - connections between nodes are labeled with bits
  - the root is at the top of the diagram (no nodes above it)
  - leaves are at the bottom (no nodes below them)
  - there is one leaf for each letter in the alphabet
- The path from the root to a leaf defines the code for that letter

A Huffman code saves space by using shorter codes for common letters

Generating a Huffman Tree (cont’d)

- Repeat until there is only one node left in the list:
  - remove the first two items (call them \(n_1\) and \(n_2\))
  - make an interior tree node \(n\):
    - \(n_1\) and \(n_2\) will be the children of \(n\)
    - the frequency of \(n\) is the sum of frequencies of \(n_1\) and \(n_2\)
    - label the connection to \(n_1\) with 0 and the connection to \(n_2\) with 1
  - insert \(n\) back into the list (keeping the list sorted by frequency)
- A simple algorithm builds the Huffman tree given an alphabet and letter frequencies

Review

- The main topics for today:
  - encoding translates symbols (letters, digits, bases, ...) into sequences of bits
  - decoding recovers symbols from bit sequences
  - ASCII codes (8 bits per character) are the default choice for text files
  - text can be compressed by using alternate encodings
  - special-purpose codes can be designed for an application (e.g., 2-bit code for DNA)
  - variable-length codes are based on letter frequencies
  - the Huffman tree algorithm can be used to generate variable-length codes
- You should be able to:
  - encode or decode a string using ASCII
  - encode or decode a string given a drawing of a Huffman tree
  - create a Huffman tree for a small alphabet given a table of letter frequencies (you’ll get some experience with this algorithm in this week’s lab)

Expect questions about codes, encoding and decoding strings, Huffman trees (Lab 3)
Parity

- The simplest method for error checking is to use a parity bit
  - add one extra bit to the end of the text
  - here “text” means any string, e.g. an entire message or a single character
- The value of the extra bit should make the total number of ‘1’ bits an even number
- Example: parity bits for ASCII characters
  - A = 01000001
  - there are two 1 bits, so the parity bit is 0 (the total remains two)
  - C = 01000011
  - there are three 1 bits, so the parity bit is 1 (bringing the total to four)

A parity bit appended to a message can help a receiver decide if an error occurred

Checksums vs. Parity Bits

- Adding the codes for the letters is the same as having 8 individual parity bits
  - think of 8 wires transmitting the bits in parallel
  - the receiver has 8 FSAs, one for each column

Using 8 parity bits (one per column) gives the receiver more information, helps it detect more errors

Calculating Parity: FSA

- One way to compute the value of a parity bit is to use a “finite state automaton”
  - the initial state is state 0
- As the message is being sent, the machine changes state according to bits being transmitted
- After the last bit in the message, send one additional bit, defined by the current state of the automaton
  - example: after sending 01000001 (ASCII 'A') send 0

A very simple algorithm computes the value of a parity bit

Distance Codes

- With more bits (larger cubes) we can “scatter” the valid code patterns around the cube
  - the receiver can detect up to $d - 1$ errors
  - the receiver can correct $\left\lfloor \frac{d}{2} \right\rfloor$ errors by choosing the “closest” correct code
- Example: 5-bit code with only two valid words (00000, 11111) and $d = 5$
  - suppose transmitter sends 00000
  - if 1 or 2 errors (any pattern with up to 2 1 bits)
Review (cont’d)

- Skills: you should be able to
  - add a parity bit to a word
  - identify whether an error occurred by checking a parity bit
  - add a checksum to a message
  - given a distance code find the closest code word to an erroneous bit pattern
  - e.g. given a 3D or 4D hypercube drawing understand the links and find a path from an error code to a valid code

Expect simple questions about parity bits and distance codes (e.g. what is the parity bit for ...?)

Regular expressions provide a “pattern language”
- implemented by Unix programs, Ruby methods, Internet sites, other applications
- used to search for substrings that match the pattern

Patterns may contain
- literal characters (i.e. “match this character”)
- place holders (e.g. “any character fits here”)
- sets (“any one of the following characters is allowed here”)
- anchors (“the match must start at a word boundary”)
- size ranges (“between 3 and 7 digits”)

The lab will give you some experience with regular expressions

Regular expressions allow us to search for instances of a pattern instead of an exact string

Expect simple questions about parity bits and distance codes

Expect questions on regular expressions using Ruby syntax (Lab 5)

I will provide a reference sheet (e.g. \w = “word char”, \d = “digit”)

Protocol Stacks

- Communication involves a layer of cooperating protocols
  - operations at one layer are implemented by operations at lower layers
  - An example from the textbook:
    - Network protocols (and other software) are organized as interacting modules
    - Processes at one layer trust jobs to processes at lower layers
    - They don’t worry about details that concern the lower level processes

Internet Protocols

- The Internet is based on a four-layer set of protocols commonly known as TCP/IP
  - TCP = transmission control protocol
  - IP = internet protocol

- The application layer defines interactions between user programs
  - HTTP (hypertext transfer protocol)
    - “get x”
    - “200 OK”
    - “404 not found”
  - SMTP (simple mail transfer protocol)
  - FTP (file transfer protocol)
  - many others

For more info: “internet protocol suite” at Wikipedia
HTML

- The special characters that mark the beginning and end of markup commands in HTML are `<` and `>`
  - `<b>` means “display xxx in bold”
  - `<a>` means “xxx is an anchor (hyperlink)”

HTML is a markup language

Review

- The important concepts introduced in these slides:
  - computer systems exchange information using **protocols**
  - we often talk about a “protocol stack” -- **layers** of software where programs at one layer trust lower levels to implement major operations
  - the Internet protocols are known as **TCP/IP**
  - the four layers in TCP/IP are application, transport, network, and data link
  - most modern local area networks use **ethernet** or **WIFI** for the physical connections
  - systems **broadcast** data to others in the local net
- Some terms to understand:
  - client, server
  - IP address, domain name, name server, DHCP
  - hypertext, markup language, HTML

Termiology

- A **cipher** is an algorithm for rewriting strings
  - the original message is called **plaintext**
  - the encoded message is the **ciphertext**
- In a substitution cipher each letter in the plaintext is replaced by a single letter in the ciphertext
  - the Caesar cipher is a substitution cipher
  - the “cryptoquote” puzzles in the newspaper are another example
- Messages encrypted with a substitution cipher:
  - letter frequency (e, t, a, o, i, n, ...)
  - letter pairs (th, sh, ... ee, oo but not ii, aa, uu)
  - common short words (a, i, in, of, the, ...)

Encryption algorithms scramble text so it is only readable by the recipient

Vigenère Square

- The Vigenère square illustrates two important ideas in traditional cryptography

1. The sender and receiver must share a **key**
   - the same key used to encrypt a message is used to decrypt a message
   - anyone who knows the key can decrypt a message

2. Kerckhoff’s Principle: the security of the scheme must depend on a secret **key but not a secret algorithm**
   - the method used to encrypt and decrypt messages must be straightforward and easy to implement
   - otherwise it would be easy to make a mistake while encrypting or decrypting

Historically the sender and receiver shared a secret key
Public Key Cryptography

- Diffie and Hellman’s method led to the idea of **public key cryptography**
  - each user maintains two keys, a **public key** and a **private key**
- Anyone can send A (who has public key $g^x$) a secure message by picking a random $y$ and sending a message encrypted with $k = g^{xy}$
  - B finds A’s public key $g^x$
  - message to A is encrypted with $g^{xy}$ (where $y$ is B’s private key)
  - A can decrypt the message by looking up B’s public key $g^y$ and computing $g^{yx}$ with their private key

State of the art methods use a public key system

- looking up B’s public key $g^x$ and computing $g^{xy}$ with their private key

RSA

- To send a message $m$ using the public key $c$:
  - turn $m$ into a number (as in the block cipher)
  - encrypt the message as $c = m^e \mod n$
- A receiver decrypts $c$ by computing $c^d \mod n$
  - $c^d = (m^e)^d = m^{ed} = m^{ed+1} \mod i$
  - $m^{ed+1} = m \times m^{ed}$
  - $m^{ed} \mod m = 1^i \times m$ (by definition of $i$)
  - $m^{ed} \mod m = m$

- If you want the gory details see Ferguson and Schneier and the Ruby program I wrote for this week’s lab project (`rsa.rb`)

Aside: Random Numbers

- A common technique: the “linear congruential method”
  - pick three constants named $a$, $c$, and $m$
  - for any value of $x$, the next $x$ in the sequence is calculated by:
    $$x = (x \times a + c) \mod m$$
  - The “mod” function (%) in Ruby is the important part of this equation
    - when $(x \times a + c)$ makes a value larger than $m$ the mod function “wraps around”
    - the value of $x$ seems to hop around at random in the range from 0 to $m-1$

Database Management Systems

- A database management system (DBMS) is a set of applications for organizing and accessing information
  - **client** programs provide the interface to the system
  - clients can connect to **servers** via the internet
  - similar to the way browsers connect to web servers
  - Users access information by submitting a **query**

- An administrator manages accounts and the system

- Database systems provide “persistent storage”
MySQL Syntax

- MySQL commands are very “English-like”
- All start with a verb telling the system what you want it to do
- Examples: “show tables”, “describe X”, “select a, b, c from X”

SQL is a standard language for relational databases

A list of columns in the table, showing their types and other information

Review

- You should be familiar with the basic definitions of relational databases
  - table, column, record, key (index)
- Given the description of the contents of one or more tables, know what the results of the basic operations would produce
  - select, project, join
- Be able to use MySQL queries to fetch information from a database
  - get a list of tables
  - get a description of a table
  - select rows from a single table
  - extra credit: write a query that does a join

Expect questions about tables, relations, MySQL (Lab 7)

Genetic Algorithms and Optimization

- We can use genetic algorithms in optimization problems
- For TSP:
  - create a “population” of possible tours
  - the “fitness” of a tour is the cost of the tour
  - use “natural selection” to keep best tours
  - replace bad tours with new ones derived from the survivors

A genetic algorithm is a different kind of search -- search an abstract space for solutions to a problem

Genetic Algorithm (cont’d)

- The key idea in a GA is that “individuals” represent problem solutions
- Generation of new solutions happens by:
  - mutation: make a copy of an existing solution and make a small change
  - cross-over: select two existing solutions, combine elements at random to produce a new solution
- In both cases the result must a complete solution
- A solution “evolves” from random starting points
Solutions (cont’d)

- A simple way to represent the tour is to use a string
  - if there are \( n \) cities there are \( n \) letters in the string
  - tours of more than 26 cities would use arrays of integers, but strings are useful for small demos (easy to understand, easy to display)
  - for the small graph shown below strings would have

Any string that is a **permutation** of these letters is a valid solution

Expect questions based on the TSP and genetic algorithms (Lab 8)

For the traveling salesman problem we start with random permutations and evolve better ones

Unsolvable Problems

- The problems on the previous slides are difficult because
  - it’s hard to specify exactly what the inputs, outputs, and steps are, or
  - they are unsolvable in a practical sense (they work for small inputs but not larger inputs)

- Problems in third category mentioned in the introduction are not just difficult, they are unsolvable
  - to a computer scientist unsolvable = **not computable**

- The remainder of these slides are concerned with noncomputable functions
  - these are functions for which no algorithm exists
  - we can go one step further: we can prove that **no algorithm will ever exist**

Universal Turing Machines

- Turing based his claim that any computable function could be computed by a Turing machine on the idea of a **universal Turing machine (UTM)**

  A UTM is a machine that simulates the actions of another machine

  The key idea:
  - the operation of a machine \( T \) is completely defined by its state transition table
  - the table can be written on a tape and read by another machine

  One can define a universal machine \( U \) that can carry out the operations defined by \( T \) and \( T \)'s input:

  \[
  \begin{array}{c|c}
  \text{State Transitions of } T & \text{Input to } T \\
  \hline
  0 & S 0 \\
  0 & S 1 \\
  1 & S 0 \\
  1 & S 1 \\
  1 & S 1 \\
  \text{U} & \text{S 1}
  \end{array}
  \]

  The “speed of light” for computation -- an absolute barrier that will never be exceeded -- is defined by what is computable with a universal Turing machine

Stored Program Computers

- The idea that a function can be represented as a string symbols and manipulated like any other data is the very foundation of modern computer science

  In a **stored-program computer** the program and data are stored in the same memory (RAM)
  - of all the “big ideas” in computer science we have studied this term this one is the biggest
  - it has profound implications on everything from abstract theoretical computer science to the more practical aspects of computer engineering

  The fact that machines (functions) can refer to other machines as data has huge implications

- Mechanical computers, dating back as far as the 1600's, used external signals to control the machine
  - these machines were (non-programmable) calculators

  [www.computerhistory.org](http://www.computerhistory.org)
Noncomputable Functions

- If the Church-Turing conjecture is true, the fact that no Turing machine can decide if another will halt means we cannot write a program, in any language, that will always decide if another program will halt.

- A straightforward proof shows functions that refer to other functions can get themselves in trouble...

- There may be simple questions based on Turing Machines (e.g. “what does the machine do if ...?”)

- Note again that these are statements about programs in general.
  - we cannot decide if a program will ever reach any given state, e.g. if it will ever execute any given statement or produce a certain output.
  - we cannot write a program that will tell us if any two functions are equivalent, i.e. that they will produce the same outputs given the same inputs.
  - we cannot write a general purpose “halt-checker” that will work on any program.

The Science of Computing

- The “science” in computer science includes:
  - algorithms: what are the most efficient methods for solving problems?
  - languages: what are the best ways to express algorithms?
  - software engineering: how can we build useful and reliable programs?
  - computer engineering: how can we build cost-effective computer systems?

- Computer science helps people solve problems:
  - science
  - engineering
  - medicine

- Computer science helps people build systems:
  - architecture
  - communications
  - music and the arts

- Instead of taking a broad survey of all these aspects of CS we took a more in-depth look at topics with a common theme.

- Another theme for the course: CS is having a major impact on other sciences and society in general.

What is “computer science”?

- Computer science is:
  - engineering
  - math
  - cognitive science
  - linguistics
  - business

Viewpoint

Jeannette M. Wing

Jeannette M. Wing (wing@cs.cmu.edu) is the President's Professor of Computer Science in and head of the Computer Science Department at Carnegie Mellon University, Pittsburgh, Pa.

Computational Thinking

It represents a universally applicable attitude and skill set everyone, not just computer scientists, would be eager to learn and use.

Computational thinking builds on the power and limits of computing processes, whether they are executed by a human or by a machine. Computational methods and models give us the courage to solve problems and design systems that no one of us would be capable of tackling alone. Computational thinking is reflec-
Professors of computer science should teach a course called “Ways to Think Like a Computer Scientist” to college freshmen, making it available to non-majors, not just to computer science majors. We should expose pre-college students to computational methods and models. Rather than bemoan the decline of interest in computer science or the decline in funding for research in computer science, we should look to inspire the public’s interest in the intellectual adventure of the field. We’ll thus spread the joy, awe, and power of computer science, aiming to make computational thinking commonplace.

Communications of the ACM, Mar. 2006

To help study for the final: read this article, find three examples of “computational thinking” that we looked at this term in CIS 170.