Review

A selection of slides from throughout the term
Study guide for the final exam

The final exam will be at
1:00 PM Wednesday March 18

Great Ideas in Computer Science

- This course was an introduction to the “great ideas” in the field of computer science
  - what sorts of problems computer scientists study
  - some of the methods used to solve those problems
- The emphasis was on problems and their solutions
  - computer science is all about techniques for solving problems
  - it’s more than just “programming”
- 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

Course Topics

- What is Computer Science?
- The Ruby Workbench
- Algorithms
  - Sieve of Eratosthenes
- Searching and Sorting
  - iterative algorithms
  - divide and conquer
- Hash Tables
- Natural Language Processing
  - ELIZA
  - regular expressions

Course Topics

- Scalability
  - Traveling Salesman
  - genetic algorithms
- Limits of Computation
  - the halting problem
- The next set of slides are all taken from earlier lectures
  - the key ideas or most important points from that week’s topic
Computation

- A **computation** is a sequence of well-defined operations that lead from an initial starting point to a desired final outcome.
  - note this definition does not include the word “computer”
  - a computation is a **process** that can be carried out by a person or a machine
  - the same computation might be carried out using any one of a number of different technologies

This slide came from the lecture named “Introduction”

What is Computer Science?

- Computer science is the **study of computation**
  - investigating problems that can be solved computationally
  - programming languages used to describe computations
  - machines that carry out computations
  - theoretical limits of computation (what is or is not computable)
  - computational solutions to problems in math, science, medicine, business, education, journalism, ...

- Computers play a key role
  - but (getting back to Dijkstra) computer science is not “about computers”

CIS 170 Labs

- Lab projects in CIS 170 are based on a programming language named Ruby
  - Ruby is a general-purpose language that can be used to write large applications
  - In CIS 170 we will use Ruby as a “workbench”
    - we will give you the programs and data
    - you will run programs, modify them, see what they do, learn how they work
  - Using a system like Ruby gives you a chance to **experiment with computations**

We used Ruby to set up and run experiments based on computations

Interactive Ruby

- Recall from the previous slides that we will be using Ruby as an interactive programming system
  - type an expression
  - Ruby evaluates the expression
  - Ruby prints the result

We used Ruby as an **interactive programming language**

Rub IRB using a terminal emulator program
Variables Are Labels for Objects

- Think of the memory of your computer (RAM) as being one big “object storage unit”
- When Ruby evaluates an assignment statement, it creates a new variable if it does not already exist
  ```ruby
class X
  def y
    x = 109
    y = x**2 - ((x/2)**2 / 2)
    return 'hello'
  end
end
```
- Variables can refer to any type of object
  - we’ll see examples of strings and other more complex objects later

Methods

- In Ruby (and other object-oriented languages) functions are known as **methods**
- Here is some important terminology:
  - when a method is used in an expression we call the method
  - a method call can include parameter values
  - we say the parameters are passed in to the method
  - methods return values that can be used in the original expression
- Example: when the sin method is passed the value 1 it returns 0.84...
  ```ruby
  sin(1.0)
  => 0.841470984807897
  ```

Abstraction

- One of the most important concepts in computer science is the idea of **abstraction**
  - when we define a method like countertop we’re making a small “package”
  - inside the package are all the details of how to compute the area of the counter
  - from outside we forget about the details -- all we care about is the fact that we can call this method and it will do a computation

The Sieve

- The basic idea is simple:
  ```ruby
  def countertop(x)
    x**2 - ((x/2)**2)/2
  end
  countertop(109)
  => 10423
  ```
  - You don’t need to know how a method is implemented
  - Example: add an item to a priority queue by calling `pq.insert(x)`

Sieve

- The first algorithm we looked at was the Sieve of Eratosthenes, which is used to make lists of prime numbers
Can we turn this method into an algorithm?

- detailed specification of starting and ending conditions
- What about the steps?
  - “cross off” and “next number” need to be defined if we’re going to use Ruby
  - when do we stop the iteration?

Isn’t it obvious there are no multiples of 11 in this list?

1. The sieve is a good example of a computation done by hand -- it is an old method that has been around for over 2000 years.

2. To do the computation on a computer we need to formalize the steps and write them symbolically.

Outline (cont’d)

- start
  - worklist: [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
  - primes: []
- copy
  - worklist: [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
  - primes: [2]
- delete
  - worklist: [3, 5, 7, 9, 11, 13, 15, 17, 19]
  - primes: [2]
- copy
  - worklist: [2, 3, 5, 7, 9, 11, 13, 15, 17, 19]
  - primes: [2, 3]
- delete
  - worklist: [5, 7, 11, 13, 17, 19]
  - primes: [2, 3]

This project introduced the ideas of arrays (aka lists) and iterators (methods that do something with each element in a collection).

Outline (cont’d)

- start
  - worklist: [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
  - primes: []
- copy
  - worklist: [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
  - primes: [2]
- delete
  - worklist: [3, 5, 7, 9, 11, 13, 15, 17, 19]
  - primes: [2]
- copy
  - worklist: [5, 7, 9, 11, 13, 15, 17, 19]
  - primes: [2, 3]
- delete
  - worklist: [5, 7, 11, 13, 17, 19]
  - primes: [2, 3]

Summary

- The Sieve of Eratosthenes is a process for making lists of prime numbers.
- For over 2000 years people used this process -- aided by paper and pencil, abacus, or whatever technology was available.
- In this unit we
  - used IRB to manage the lists, explore how the sieve works
  - created an algorithm that controls each step in the process
  - improved the algorithm after analyzing composite numbers

<table>
<thead>
<tr>
<th>Iterations</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>simple</td>
<td>sqrt</td>
</tr>
<tr>
<td>100</td>
<td>25</td>
<td>5</td>
</tr>
<tr>
<td>1000</td>
<td>168</td>
<td>12</td>
</tr>
<tr>
<td>10000</td>
<td>1229</td>
<td>26</td>
</tr>
<tr>
<td>100000</td>
<td>9592</td>
<td>66</td>
</tr>
<tr>
<td>1000000</td>
<td>78498</td>
<td>169</td>
</tr>
</tbody>
</table>
Linear Search

- The simplest search algorithm is known as linear search.
- As the name implies, the strategy is to start at the beginning of a collection and compare one by one.

Some terminology:
- The item we are looking for is known as the key.
- This type of search is also sometimes called a scan.
- If the key is not found, the search fails.

We looked at how iteration could be used to define straightforward searching and sorting algorithms.

Nested Loops

- At first glance, it might seem that insertion sort is a “linear” algorithm like search and max.
  - It has a for loop that progresses through the array from left to right.
- But it’s important to note what is happening inside the loop.
  - The step that finds the proper location for the current item is also a loop.
  - It scans left from location i, going all the way back to 0 if necessary.
- An algorithm with one loop inside another is said to have nested loops.

Linear Search

- The simplest search algorithm is known as linear search.
- As the name implies, the strategy is to start at the beginning of a collection and compare one by one.

Some terminology:
- The item we are looking for is known as the key.
- This type of search is also sometimes called a scan.
- If the key is not found, the search fails.

We looked at how iteration could be used to define straightforward searching and sorting algorithms.

Insertion Sort

- Here is a more precise statement of the insertion sort algorithm:
  1. The initial key is the second item in the array (the Q in this example).
  2. Use your left hand to pick up the key.
  3. Scan left until you find an item lower than the one in your left hand, or the front of the array, whichever comes first.
  4. Insert the key back into the array at this location.
  5. The new key is the item to the right of the location of the previous key.
  6. Go back to step 2.

- This new version is precise enough that we can organize a Ruby method that will implement this algorithm.

Nested Loops

- The diagram at right helps visualize how many steps the isort method takes:
  - A dot in a square indicates a potential comparison.
  - For any value of i, the inner loop might have to compare key to values from i - 1 all the way down to 0.
- The number of dots in this diagram is (6 * 5) / 2 = 15.
- In general, for an array with n items, the number of comparisons is
  \[(n \times (n - 1)) / 2 \approx n^2 / 2\]
What You Should Know

- The first part of the lab will be on the algorithms presented here
- The second half will be on new searching and sorting algorithms presented next week
- The tutorial project will have instructions for downloading and running the search, max, and isort methods
- The version of isort in this file will have options for printing the array as the algorithm progresses
- You will be able to see the sorted portion grow on each iteration
- Lab questions will test your understanding, e.g., ask you to write out what the sort method would print for a given input list

Expect questions that test your general understanding of linear search and insertion sort

Divide and Conquer

- The common theme for the previous slides: linear scan through every item in the list
- There is also a common theme for this set of slides: divide and conquer breaks a problem into smaller pieces and solve the smaller sub-problems, but the improvement can be dramatic

Divide and Conquer Sorting Algorithms

- We then looked at more sophisticated searching and sorting algorithms

<table>
<thead>
<tr>
<th>Search Method</th>
<th>n = 100</th>
<th>n = 1,000</th>
<th>Quadratic</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear</td>
<td>100</td>
<td>1,000</td>
<td>5,000</td>
</tr>
<tr>
<td>divide and conquer</td>
<td>7</td>
<td>10</td>
<td>500,000</td>
</tr>
</tbody>
</table>

Divide and Conquer Sorting Algorithms

- The divide and conquer strategy used to make a more efficient search algorithm can also be applied to sorting
- Two well-known sorting algorithms:
  - QuickSort
    - divide a list into big values and small values, then sort each part
  - Merge Sort
    - sort subgroups of size 2, merge them into sorted groups of size 4, merge those into sorted groups of size 8, ...
  - The remaining slides will have an overview of each algorithm, and a look at how Merge Sort can be implemented in Ruby
    - see the msort method in sortlab.rb

The same general strategy can be applied to sorting algorithms (and many other problems we did not look at)

Binary Search

- To search a list of n items, first look at the item in location n/2
  - then search the region from 0 to n/2-1 or from n/2+1 to n-1

Example: searching for 36 in a sorted list of 15 numbers

Real-world analogy: the process we use to find a word in a dictionary
The merge sort algorithm has been implemented in a method named `msort`:
- has the same parameters and options as `isort` and `qsort`.

```ruby
>> a = randoms(16, 100)
=> [1, 87, 0, 52, 12, 32, 44, 32, 35, 94, 55, 63, 17, 38, 86, 33]
>> msort(a, :trace)
[1 87] [0 52] [12 32] [32 44] [35 94] [55 63] [17 38] [33 86]
0 1 52 87] [12 32 32 44] [35 55 63 94] [17 33 38 86]
[0 1 12 32 32 44 52 87] [17 33 38 86]
[0 1 12 17 32 32 33 35 38 44] 52 87]
[0 1 msort(a, :count)
=> 38
```

Recursive Merge Sort divides its input list into small groups, then repeatedly combines small groups into larger groups.

### Comparisons in Merge Sort

- Is this new formula that much better than the \( n^2/2 \) comparisons made by `isort`?
  - not that big of a difference for small arrays
  - huge difference for larger arrays

```ruby
>> a = randoms(1000)
>> isort(a, :count)
=> 243995
>> msort(a, :count)
=> 8720
```

For small lists the new strategy might not seem very useful, but for bigger lists there is a dramatic improvement.

### Summary

- These slides introduced the **divide and conquer** strategy:
  - for searching: **binary search**
    - requires list to be sorted
  - for sorting: **QuickSort and merge sort**

- Binary search will find an item using at most \( \log_2 n \) comparisons.
- QuickSort and merge sort do at most \( n \times \log_2 n \) comparisons.

- An algorithm that uses divide and conquer can be written using **iteration** or **recursion**:
  - recursive = "self-similar"
  - a problem that can be divided into smaller subproblems of the same type
  - a recursive method calls itself

A recursive function is one that is defined in terms of a smaller (self-similar) pieces.

Expect questions that test your general understanding of binary search and merge sort.
Review

- A hash table is a technique for organizing information analogous to an index in a book.
- Use a hash function to determine a row in the table:
  - insert a word s in row h(s)
  - to see if s is in the table compute i = h(s), look in row i
- Trivial hash function: use only the first letter of a word
- Better hash functions: use more letters
- Collisions occur when h(s1) = h(s2)
  - far more frequent than we might guess (birthday paradox)
- A table can deal with collisions by using buckets

A hash table lets us organize information in a way that makes searching even more efficient than binary search

Summary

- A trivial hash function h0 uses the first letter of a word
  \[ s[0].ord \]
  - range of values = 0..25
- A slightly better function h1 uses the first two letters
  \[ s[0].ord \times 26 + h[1].ord \]
  - range = 0.675
- A general-purpose function hn uses all the letters, e.g.
  \[ s[0].ord \times 26^7 + s[1].ord \times 26^6 + \ldots \]
- After computing the product of the letter values find the remainder mod n
  - n is the number of rows in the table
  - the result is between 0 and n-1

In the best cases we can find an item with only one comparison

What You Should Know

- You should be able to compute the value of h0(s,n) or h1(s,n) for any word
  - don’t memorize the code — it will be given on an exam
  - don’t memorize ASCII or the ord method -- tables will be included with an exam
- You should be able to demonstrate where words are placed in a table using a simple hash function
- The sample data may include collisions -- understand how buckets work

1,000 empty rows for every full one, but still almost 5% chance of a collision!

<table>
<thead>
<tr>
<th>n</th>
<th>p(collision)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>99.29%</td>
</tr>
<tr>
<td>10,000</td>
<td>39.04%</td>
</tr>
<tr>
<td>100,000</td>
<td>4.83%</td>
</tr>
<tr>
<td>1,000,000</td>
<td>0.49%</td>
</tr>
</tbody>
</table>

A simple way to resolve collisions is to use “buckets” (lists of items that all hash to the same location)
Regular Expressions

- To help with pattern-based searches, Ruby has a special type of string called a **regular expression**
  - Often abbreviated as “regexp”
- To write a regular expression, use slash characters instead of quotes to indicate the start and end of the pattern.

```ruby
s = "How now, brown cow?"
puts s.index("ow")
# => 1
puts s.index(/ow/)  # A regular expression in Ruby is a special type of string that is used to defined a pattern
# => 1
```

- The simplest regular expressions are just like normal strings:
  - The pattern begins and ends with slash.

Regular Expressions in CIS 170

- **Why introduce such a complicated new topic?**
  - The next project will use pattern matching to break input sentences into smaller parts and find key words.
  - Patient: I don’t like cows.
  - Dr. Ruby: Why don’t you like cows? Response: “Why don’t you _____ ?”

```ruby
Pattern: "I don’t _____"
Dr. Ruby: Why don’t you like cows? Response: Why don’t you _____ ?
```

- Ruby will extract the part of the sentence after “don’t” and plug it into the blank part of the response.
- We’ll stick to the basics:
  - As with Ruby itself, the goal is “literacy.”
  - Understand that a search is using a pattern.
  - The more complicated patterns will be written for you already.

The ~= Operator

- Because regular expressions are used so often in programs, Ruby has a special operator:
  - `s =~ r` is equivalent to `s.index[r]`
  - This syntax was introduced in Perl, and adopted by Ruby.

```ruby
s =~ /ow/    # The simplest regular expressions are just like normal strings
# => 1
```

- Use an expression of the form `s =~ r` to ask Ruby if the string `s` matches the pattern `r`.

Review

- Regular expressions provide language for describing patterns.
  - Ruby methods use them 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”).
  - Character classes (“any character in this set 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.

```ruby
s = "Where in the world is Carmen Sandiego?"
puts s =~ /i\w/  # This operator is a two-character sequence with a tilde after an equal sign
# => 6
puts s.scan(/i\w/)  # Do not memorize all the special symbols -- there will be a “cheat sheet” at the end of the exam
# => ["in", "is", "ie"]
```
This Week’s Projects

- The topic this week: how information is stored inside a computer
- The kinds of things stored in a computer’s memory include:
  - numbers
  - text
  - drawings
  - photographs and other images
  - music and other audio
- Obviously a very big topic:
  - we’ll focus on text
  - even more specifically, on strings of letters (and not worry about formatting)

An Important Formula

- If a set has \( n \) items, the number of bits required to represent an element of the set is:
  \[ k = \lceil \log_2 n \rceil \]
- each item in the set can then be assigned a unique pattern of \( k \) bits
- Examples:
  - 4 nucleotides in DNA (A, C, G, or T): 2 bits per letter
  - a choice of 6 colors: 3 bits per color (two patterns unused)
  - a lower case letter: 5 bits (6 patterns unused)

More Compact Codes

- For most text ASCII is a good choice:
  - wide variety of letters and symbols
  - easily stored in computer memory (one letter = one byte)
- For sequence databases ASCII can be very inefficient:
  - Example: DNA sequences have only A, C, G, and T
  - a special-purpose code that uses just 2 bits for each letter would require 1/4 as much memory
  - e.g. 575,000 bytes instead of 4,600,000 for the *E. coli* genome

Huffman Tree

- The codes on the previous slide were defined by a Huffman 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

<table>
<thead>
<tr>
<th>Nucleotide</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>00</td>
</tr>
<tr>
<td>C</td>
<td>01</td>
</tr>
<tr>
<td>G</td>
<td>10</td>
</tr>
<tr>
<td>T</td>
<td>11</td>
</tr>
</tbody>
</table>
Generating a Huffman Tree (cont’d)

Repeat until there is only one node left in the list:
- remove the first two items (call them n1 and n2)
- make a new node (call it n) that will be an interior tree node:
  - n1 and n2 will be the children of n
  - the frequency of n is the sum of frequencies of n1 and n2
  - label the connection to n1 with 0 and the connection to n2 with 1
- insert n back into the list (keeping the list sorted by frequency)

In-Class Project

Let’s do one as a group, using the data from the strange RNA

Here’s the initial list of leaf nodes:

Note the list shrinks by one node on each step: remove two, insert one

Do we end up with the tree shown earlier?

Implementation

A class named Node captures all the information to represent a single node in a Huffman tree

```ruby
>> t1 = Node.new("T", 0.025)
=> T: 0.025
>> t2 = Node.new("C", 0.025)
=> C: 0.025
>> Node.combine(t2,t1)
=> ( 0.050 C: 0.025 T: 0.025 )
```

The lab project gave you a chance to create nodes and use a priority queue to build your own tree

Node.combine is a constructor that makes a new node from two existing nodes

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 on binary codes, text compression, and Huffman trees
**Overview**

- The last lecture showed how we can compress text files
  - letters are encoded as sequences of bits
  - one goal is to archive several large files on a CD
  - transferring large files over the internet is also easier if the files are compressed
- Files can be decoded without loss of information
  - \( \text{decode(encode}(F)) = F \)
- The topic for today goes in the opposite direction
  - we’ll see how adding extra bits to codes provides additional information
  - this information can be used to detect -- and maybe correct -- errors made when the file is stored or transmitted
  - the text will be longer, but the additional space may be a good investment if the data is critical and the cost of an error is high

**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)
- Example: parity bit for a piece of DNA (using ASCII)
  - \( \text{ATG} = 01000001 \ 01010100 \ 01000111 + 1 \)

**Back to Parity Bits**

- The receiver knows an error occurred somewhere in the bit stream if the total number of 1 bits is odd
- The receiver does not know where the error occurred...
  - it could have been the parity bit itself...
- An even number of 1 bits does not guarantee there were no errors...
  - if two bits changed the total number of 1 bits will still be even
- Example:
  - sender transmits \( 01000011 \)
  - receiver sees \( 00100011 \)
  - 2nd and 3rd bits changed, but the message looks OK to the receiver

**Distance Code Example**

- In the lecture on Huffman trees there was an example of a binary code for DNA letters
  - we can form \( 2^n \) different binary numbers using \( n \) bits
  - the four DNA letters use all four 2-bit codes
  - this is a **distance-1** code: a single change to any code will lead to another valid code
- The 3-bit code at right is a **distance-2** code
  - pick any code, and change one bit -- the result is not a code for any letter
  - you need to change two bits to get to another valid code
  - 8 patterns but only 4 **code words**

<table>
<thead>
<tr>
<th>Nucleotide</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>000</td>
</tr>
<tr>
<td>C</td>
<td>011</td>
</tr>
<tr>
<td>G</td>
<td>101</td>
</tr>
<tr>
<td>T</td>
<td>110</td>
</tr>
</tbody>
</table>
Distance-2 Code

- From this drawing it's clear why the 3-bit code for DNA letters shown on a previous slide is a distance-2 code
- Each link in a hypercube corresponds to a 1-bit difference between the codes at the ends of the link
- The valid words in this code are on nodes that are separated by two links

<table>
<thead>
<tr>
<th>Nucleotide</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>000</td>
</tr>
<tr>
<td>C</td>
<td>011</td>
</tr>
<tr>
<td>G</td>
<td>101</td>
</tr>
<tr>
<td>T</td>
<td>110</td>
</tr>
</tbody>
</table>

Correction

Expect questions on parity bits and distance codes

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 diagram understand the links and find a path from an error code to a valid code

The Turing Test

- Think of the test as a game with three players
  - Player A is a person typing questions on the terminal -- they win if they correctly guess which room has the computer
  - Player B is the human in one of the rooms -- he wins if he can fool A into thinking he is a computer
  - Player C is the computer -- it wins if it can fool A into thinking it is a human

- C passes the Turing Test if it wins the game... it would have to have a fairly high degree of “artificial intelligence” to win....

ELIZA

- One of the first programs to attempt to carry on a conversation was named ELIZA
  - Written by Joseph Weizenbaum at MIT in 1966

- Applied simple “transformation rules” to create a response to each sentence typed by a user

  - H: My father wouldn’t buy me a puppy.
  - C: Tell me more about your family.

- The program does not need to know about parents, or anything else implied by the input sentence
  - Just respond to any sentence with the word “father” by “tell me more....”

ELIZA won’t pass the Turing Test, but it’s a fun way to explore issues related to natural language processing

NLP is difficult because it is so hard to define it in the form of an algorithm

CAPTCHA = “Completely Automated Public Turing test to tell Computers and Humans Apart”

In these slides H and C indicate strings typed by the (human) user and the computer's response.

ELIZA won’t pass the Turing Test, but it’s a fun way to explore issues related to natural language processing.
**Doctor**

- The Doctor script is a set of rules that has the computer play the role of a non-judgmental psychotherapist.
- The doctor simply urges the patient to talk about what is bothering them.
- The simplest responses are statements like “tell me more” or “go on.”
- Slightly more complicated responses use part of the patient’s sentence.

  **H:** I need a hug.
  **C:** Why do you need a hug?

**Pattern Objects**

- After you make a Pattern object you can apply it to sentences:
  ```ruby
  >> p = Pattern.new( /I (want|need) (.*)/, "Why do you $1 $2?"
  >> /I (want|need) (.*)/: ["Why do you $1 $2?"
  >> p.class
  => Pattern
  >> p.apply("I want a new car")
  => "Why do you want a new car?"
  >> p.apply("I need a hug")
  => "Why do you need a hug?"
  ``

- The apply method simply combines all the things we’ve seen on previous slides into one convenient method.
- See if the sentence matches the pattern’s regular expression.
- If so collect the pieces in $1, … and plug them into the reassembly string.

**Postprocessing**

- Our sample pattern doesn’t do a very good job with this sentence:
  ```ruby
  >> p = Pattern.new( /I (want|need) (.*)/, "Why do you $1 $2?"
  >> p.apply("I need to take my pills")
  => "Why do you need to take my pills?"
  ```

- This sort of thing is easily fixed by **postprocessing**
  - pass a “dictionary” that defines words to change;
  - scan each piece of the match, substitute words defined in dictionary

  ```ruby
  >> p.apply("I need to take my pills", \{ "my" => "your" \})
  => "Why do you need to take your pills?"
  ```

**The ELIZA Algorithm**

- ELIZA breaks a sentence into words
- If there is a rule for a word the word is added to a list.

  ```ruby
  # simplified version; see eliza.rb
  # for the actual code
  line.scan(/\w+/) do |w|
    if r = rules[w]
      queue.insert(r)
      end
  end
  ```

- ELIZA breaks a sentence into words, then looks for Patterns in its script that are defined for those words.
**Script File**

- A simple rule just has a keyword and a list of responses
  - perhaps /
    - "You don't seem quite certain."
    - "Why the uncertain tone?"

- Put a number next to a key word to specify a priority
  - remember 5
    - /I remember (.+)/
      - "Do you often think of $1?"
      - "What is the connection between me and $1?"
    - /do you remember (.+)/
      - "Did you think I would forget $1?"
      - "Why do you think I should recall $1 now?"

**Inference**

- The responses on the previous slide are examples of a more general process known as *inference generation*
- We are constantly “filling in the blanks” and making inferences as we listen to sentences
- Consider this story:
  - “The cows walked across the pasture. George was waiting for them at the barn.”
  - How would you answer the question: “Where were the cows going?”
  - By “connecting the dots” between the two sentences it seems clear the cows were headed to the barn
    - this inference depends on knowing what “waiting” means
    - you might also make lots of other inferences (it was late in the day, George was getting ready to do the milking, ...)

**Real-World Knowledge**

- Simple answers to the question about George and the cows are based on *common sense knowledge*
  - if George was waiting for the cows the cows must have been going to where George was standing
- Other answers might be based on *expert knowledge* of farms and cows
  - cows walk to the barn when they know it’s time to be milked, cows are milked twice a day, ...
- Paradoxically, more progress has been made in creating programs that use expert knowledge
  - medical diagnosis
  - engineering (e.g. tracking down faults in systems)
  - business and finance

**A Hard Problem**

- Several hard problems were described in Ch 1 of the textbook
- Playing the perfect game of chess
  - a problem of *scalability*
  - we’ll see another one next week -- the traveling salesman
- The “halting problem”
  - will a program ever terminate?
  - a *mathematically impossible* problem
- Natural language processing
  - a problem that is very hard
  - will computers ever be as good as humans at understanding and using language?
This week’s lab explores a well-known problem known as the traveling salesman problem (TSP). Suppose we have a set of \( n \) cities, the goal is to define a tour that visits all \( n \) cities and returns to the starting place. Can we visit each city only one time? What is the lowest cost tour?

We used the TSP to explore problems that have simple solutions for small cases but are very difficult for larger cases.

This tour of 13,500 US cities was generated by an advanced algorithm that used several “tricks” to limit the number of possible tours. Required 5 “CPU-years.”

Real-Life Applications

- The idea that anyone would really plan a road trip to 13,000 cities is a bit silly.
- But the solution of several important “real world” problems is the same as finding a tour of a large number of cities.
  - transportation: school bus routes, service calls, delivering meals...
  - manufacturing: an industrial robot that drills holes in printed circuit boards
  - VLSI (microchip) layout
  - communication: planning new telecommunication networks.

The TSP is a good example of how computer scientists use abstraction. Consider only the essential aspects of a problem, develop an algorithm that applies to several different real-world situations.

For the TSP the goal is to find the minimum cost tour.

- each member of the population is a complete tour of all the cities
- start with a bunch of random tours
- evaluate the costs, toss the most expensive ones
- rebuild the population by making small variations of the surviving tours

If we can’t look at all solutions, is there a way to look at a small subset and find an answer that is probably the best? or at least close to the best?
Tours

A simple way to represent a tour is to use a string.
- If there are \( n \) cities there will be \( n \) 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 the letters "A" through "G".

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

These strings are the "DNA" of the population of solutions.

Point Mutations

One technique for defining mutations on paths: simply swap two cities at random.
- We could exchange the places of any two cities.
- To make a very small change exchange two adjacent cities.
- Pick a random location \( i \), exchange \( s[i] \) and \( s[i+1] \).

Crossovers

Defining cross-overs is a little more difficult.
One idea:
- Start with a random substring from one tour.
- Copy the remaining cities in order from the other tour.

Copying in order from the other string preserves any good subtours that may have evolved there.

Copy remaining letters (A,B,C,G) from other parent

Start with EDF from one parent

The Tour Class

Another new class defined in tsp.rb is named Tour.
- A Tour object has a unique id (so we can keep track of different tours), a string representing the path between cities, and the cost of that path.
- To make an object, just call new, passing it the matrix of distances.

<table>
<thead>
<tr>
<th>Object</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>7651</td>
</tr>
<tr>
<td>t2</td>
<td>8016</td>
</tr>
<tr>
<td>t3</td>
<td>9684</td>
</tr>
</tbody>
</table>

An object of the Tour class is a potential solution to the TSP.
The Tour Class

- Another way to make a new Tour object is to call a method named reproduce.
- If you pass this method a single Tour object, you get back a copy of that tour with a random point mutation added to it.

```ruby
>> t0 = Tour.new(m)
=> #3: EBCFDAGJHI / 6354
>> t1 = Tour.reproduce(t0)
=> #4: EBCFADGJHI / 6352
```

A method that creates a new object is called a constructor.
Every class has at least one constructor, named new.
The reproduce method is a second constructor for the Tour class.

A method that creates a new object is called a constructor.
Every class has at least one constructor, named new.
The reproduce method is a second constructor for the Tour class.

> Call Tour.reproduce(t) to make a new tour from an existing one.

- If you pass the reproduce method two Tour objects, the new object is a cross between the two.
  - the method copies a random chunk of the first parameter.
  - it then adds the missing cities by getting them in order from the second parameter.

```ruby
>> t0 = Tour.new(m, "ABCDEFGHIJ")
=> #5: ABCDEFGHIJ / 4112
>> t1 = Tour.new(m, "AEIBCDFGHJ")
=> #6: AEIBCDFGHJ / 6938
>> Tour.reproduce(t0, t1)
=> #8: EFGHAIBCDJ / 7528
```

- A fake tour with all cities in order (so we can more easily see how the crossover works).
- The second fake tour has all the vowels first, then the consonants.

- To make a “population” -- a collection of tours -- call initPopulation.
- Pass it the number of tours you want and the CityList object.
- You’ll get back an array of tours made from those cities:

```ruby
>> p = initPopulation(10, m)
=> [#0: FCHEADJIBG / 7916, #1: BDAGFEIHNJ / 5898, #2: CIFEGDJHAB / 7085, #3: EGFHDAIBEC / 9316, #4: ECDDHDFJAI / 7960, #5: CHABIFEDGJ / 8699, #6: EHGDFJAIUB / 8903, #7: HJGECRIADC / 6785, #8: BFHEGDDJCAI / 9294, #9: DFGAEBICHJ / 9616]
```

- Expect general questions on the TSP, genetic algorithms, and the Ruby implementation.

What You Should Know

- Understand the general idea behind a genetic algorithm.
  - each item in a “population” is a potential solution.
  - the best solution gradually evolves.
- Understand how an object of the Tour class represents a solution of the TSP.
- Be able to show how a point mutation or crossover would create a new Tour object from one or two existing tours.

- Start the genetic algorithm by making a “population” of random tours -- i.e. an array of Tour objects.

- Call Tour.combine(t1, t2) to make a cross of two existing tours.

- Populations
Unsolvable Problems

- The problems on the previous slides are difficult because:
  - it’s hard to specify exactly what the inputs, outputs, and steps are (e.g. NLP)
  - they are unsolvable in a practical sense -- they work for small inputs but not larger inputs (e.g. TSP)
- Problems in third category mentioned in the introduction are 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

Church-Turing Conjecture

- In the 1936 paper that introduced Turing machines Turing also made a startling claim:
  - if a function is computable at all there is a Turing machine that is capable of computing it
  - a paper by another logician at about the same time made a similar claim for another model of computation
  - today this assertion is known as the Church-Turing Conjecture
- So the claim that a function is not computable is now "the function is not computable by a Turing machine"
  - to explore the limits of computability we can ask about problems that cannot be solved by a Turing machine

Balanced Parentheses (cont’d)

- A high-level explanation of what the machine does:
  - state 0 is the "scanning right for [" state, state 1 is "scanning left for ["
  - start by scanning right
  - when a [ is found it replace it by an X and look left
  - if matching ] is found replace it by X and scan right
  - a scan in either direction just skips over X

Universal Turing Machines (cont’d)

- Think about the implications of this idea
- Before Turing’s paper, there was a widespread assumption that one would design a different computing machine for different "use the right tool for the job"
- build one machine for computing trajectories of rockets, another for computing tables of logarithms, ...
- Turing’s “universal machine” makes this unnecessary
  - build just one type of hardware: a UTM that simulates the actions of any other Turing machine
  - the “machines” that compute trajectories, tables, etc are "software" that is fed into the UTM
Proof

The proof that one cannot write an algorithm to decide if a Turing machine will halt is a proof by contradiction:
- first assume that such a machine exists
- then make a simple construction that modifies the original machine
- show the resulting machine leads to a paradox: there will be a statement about the modified machine that cannot be true if the original assumption is true.

The proof depends on the fact that a TM description can be put on a tape and fed into the UTM

The statement on the other side of this page is true
The statement on the other side of this page is false

Are these statements true or false?
Neither: they are undecidable

Summary

- Today we looked at another type of difficult problem:
- A noncomputable function is a problem for which there is no algorithm. Saying a function is noncomputable is more than saying “an algorithm has not yet been discovered”.
- It is based on a proof that there never will be an algorithm (at least for Turing-equivalent languages and computers).
- A noncomputable function is the CS equivalent of an undecidable problem in logic.

What happens when a UTM is fed a description of itself?

There may be extra credit questions on Turing machines and/or logical paradoxes

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 be more effective or creative:
  - architecture
  - communications
  - music and the arts

What is “Computer Science”?

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

Hopefully the topics we looked at this term gave you a good overview of the field of computer science

http://www.illustratori.it/Public
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 refines. Stating the difficulty of a problem accounts for the underlying power of the machine—the computing device that will run the solution. We must consider the machine's instruction set, its resource constraints, and its operating environment. In solving a problem efficiently, we might further ask whether an approximate solution is good enough, whether we can use randomization to our advantage, and whether false positives or false negatives are allowed. Computational thinking is reinforcing.