Mechanics

Location: CIT 368, J Hour (1-2:20 TuTh)
Instructor: John Savage, CIT 503, x3-7642
Office Hours: By appointment

Head TA:
   Laura Shea

UTAs:
   Tracy Chin (tchin) Arun Das (ad46)
   Francesco D’Amato (fdamato1) Peter Eastwood (peastwoo)
   Cole Hansen (chansen2) Bessie Jiang (bjiang)
   Lindy Le (lrle) Michael Mueller (mmuelle1)
   Giovanni Pittalis (gpittali) May Tomic (vtomic)
Textbooks

  – Available at no cost electronically at http://www.modelsofcomputation.org

  – Recommended but not required.
Course Questions

• Why are some problems impossible?
• Why are other problems hard?
• How do we approach these questions?
Approach

• We define basic **models** for **computers, languages, and computations**.
  – Why are models important?

• We present techniques to determine
  – What questions are/not answerable by computers.
  – Why some computational problems are inherently hard.
  – How to characterize the descriptive power of languages.

• We test knowledge through assignments and exams
More Mechanics

• Weekly homework
• Midterms
  – A take-home exam in lieu of a homework
  – An in-class exam
• Final exam in class
• Weekly labs
  – Provide opportunities to work together
  – Lab problem solutions are graded with homework
Defining Computational Tasks

• Problems are defined by functions/languages
  – A function $f$ maps one input $x$ to one output $y = f(x)$.
  – If the function $f$ has two values, 1 and 0, a language $L_f$ can be associated with $f$ that consists of strings $x$ such that $f(x) = 1$. We write this as $x \in L_f$.
  – Problems can also defined by languages

• Functions can be specified in many ways
  – By tables, formulas, circuits, or algorithms.
Examples of Function Definitions

• Formulas
  – $f(x) = x^2$

• Tables

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

• A function can be defined by an algorithm
  – sum = 0; for i = 1 to n sum = sum + i;
    % sum = n(n+1)/2

• Circuits
  – $z = x \land y$ (here $\land$ denotes the AND function)
Models of Computation

• Logic circuits
  – They connect gates computing AND, OR, NOT, XOR, etc.

• Finite state machines (FSMs)
  – An FSM has an initial state, takes an input, produces an output and moves to next state.

• Random access machine (RAM)
  – Models PC: Has CPU and random access memory (FSMs)

• Turing machines
  – These are FSMs augmented with a potentially infinite tape holding inputs and temporary results.
Logic Circuit Example – Full Adder

- Computes $g(x, y, z) = (c, s)$. Here $(c, s)$ is the binary representation for the number of 1s in $x$, $y$, $z$.

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>C</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

⊕ is Exclusive OR
∨ is OR, ∧ is AND
Logic Circuits

• Why study logic circuits?
  – Simplest, most fundamental computational model.
  – Take inputs and produce outputs, have no memory.

• We can simulate other computations by circuits.
  – Q: What is role of circuits in theory?
  – A: We can simulate time-bounded computations on memory-based machines by circuits.
Finite State Machines (FSMs)

- An FSM has state, takes input, produces output, and changes state. It is clocked.
- Example: Railroad crossing gate

![Finite State Machine Diagram]
Computing with an FSM

• An FSM computes the function defined by the mapping of input strings to output strings.
  – FSM has a fixed initial state.

• RR Crossing FSM
  – State is position of gate; Initial state is Open
  – Input is position of train. It is Here or Not Here
  – State is also the output: vehicles can observe state
    • FSM(No,No,No) = (Open,Open,Open)
    • FSM(Yes,Yes,No) = (Closed,Closed,Open)
Finite State Machines

• FSMs model computations in which the number of states is known in advance.
  – The state of a program is the set of all possible values that are assumed by program variables.
  – It is a clean conceptual way to design many algorithmic solutions.
Random Access Machine (RAM)

- Simple model for a personal computer
- Has CPU and random access memory
  - Each is an FSM
Random Access Machine (RAM)

- Implements fetch-execute cycle
  - Fetch an instruction from memory into CPU
  - Decode instruction and execute it
  - Repeat
Computing with a RAM

• RAM is traditional model for programming
  – Program loaded into memory
  – Program takes input from terminal, mouse, I/O port or file and produces output to these places.

• Functions describe algorithmic mappings, e.g.
  – fact(n) = n!
  – Draw(Mouse,KeyboardInputs) → CanvasPixels
The Turing Machine (TM)

• TM = Control Unit (an FSM) & infinite tape.
• Control Unit takes input from cell, changes state, and produces output which replaces cell contents and moves head by at most one cell.
Computing with a TM

• Tape is blank initially and FSM in initial state.
• **Input string** is left-adjusted on tape.
• TM takes input from cell. Its output replaces cell contents and moves head.
• **If it halts**, TM **output** is the non-blank string that is left-adjusted on its tape. **If it does not halt**, the output is not defined.
  – A TM computes the function from inputs on which it **halts** to the output strings it produces.
  – The function is not defined on non-halting inputs.
Computing with a TM

• Can you design a TM to
  – Add one to an integer placed on the tape?
  – Determine if an integer is a palindrome?
Church-Turing Thesis

• The TM is a formal model.
  – It is not meant to be efficient, but
  – No other known computational model is more powerful.
  – I.e., every function computed by every known computational model can be computed by a TM.

• **Church-Turing Thesis**: Every function that can be computed can be computed by a TM.
A Hard Computational Problem

• **Traveling Salesperson Problem**
  – Salesperson must visit \( n \) cities and return home.
  – Travel is expensive.
  – **Goal**: find shortest tour that visits all \( n \) cities

• **Input**: Distances between all pairs of \( n \) cities.

• **Output**: Shortest tour

• **Running time** \( T \) for fastest known algorithm satisfies \( T = O(k^n), k > 1 \). This is an exponential!
  – Is this the best possible?
Classifying Problems by Complexity

• This is one major objective of this course.
• Many important problems are hard to solve.
  – If a set of problems has no known polynomial time solution, try to group them according to difficulty.
  – \textbf{NP-complete} problems are common but thought to require exponential time to solve.
• Example: \textbf{Traveling Salesperson Problem}
  – The language version of it is NP-complete.
How to Classify Hard Problems?

• **We use reduction** (reuse the code!)
  – Efficiently reduce a new problem to a previously solved one.

• **Example, reduce squaring to multiplication**
  – Square $x$ by multiplying $x$ by itself. $sq(x) = mult(x,x)$
  – Thus, if best algorithm for $sq$ known to be slow, $mult$ can’t be fast!

• If best algorithm for $mult$ were known to be slow (say exponential time in $|x|$), could $sq$ be fast?
  – Reduce it to $mult$. That is compute $mult$ on $x$ and $y$ using $sq$. (Use inputs $x$ and $y$ to prepare input $z$ to $sq$. Find $xy$ in the output.) Thus, if $sq$ has a fast algorithm, $mult$ must also have a fast algorithm. But **If $mult$ is slow, $sq$ cannot be fast.**
  – This type of reasoning is used throughout the course.
Reducing Multiplication to Squaring

• Multiplication via squaring is harder than it looks
  – Use base 2. Let $x, y \leq 2^{n-1}$ for $n \geq 2$ and $a = 2^{2n-1}$.
  – In base 2, multiplication by $2^k$ obtained by shifting!
  – Form $z = x + ay$ and square.
  – Result is $z^2 = x^2 + 2axy + (ay)^2$.
  – But $x^2 < 2axy < (ay)^2$ if $x, y \geq 1$.
  – Bits representing $x^2$, $2axy$, and $(ay)^2$ don’t overlap

• $xy$ obtained by extracting middle bits $2axy$ of $z^2$. 

© John E. Savage
Generalizing Notion of Reduction

• If problem A is hard to solve and we can reduce (transform) it efficiently to problem B, then B is not easy to solve.
  – If B has an efficient solution, let’s solve hard problem A quickly by reducing it efficiently to B and then solving B.
  – This provides an efficient algorithm for A.
  – But since A is hard, B can’t have an efficient solution.

• Thus, if we can show A is hard, another problem B can be shown hard by efficiently reducing A to B.
The Role of Reductions

• Reductions are used to show that problems are
  – NP-complete
    • Thought hard to compute efficiently on serial computers.
  – P-complete
    • Thought hard to parallelize efficiently
  – Not decidable by any TM (e.g. the Halting Problem)

• The Halting Problem
  – No TM algorithm exists that can decide if another TM will halt on its input or not.
  – Once we show Halting Problem is impossible, we can do the same for other problems using reductions.
Languages and Computation

• We can model many tasks with languages
  – A language is a subset of the set of all strings over an alphabet.
  – E.g. Pascal, C, and \{01, 0011, 000111, \ldots\} are languages

• Example
  – Instances of TSP are inter-city distances + a tour length \( L \)
    • (\( d(1,2),d(1,3),d(1,4),d(2,3),d(2,4),d(3,4); L \))
  – Yes Instances: min\_tour\_length \( \leq L \)
  – No Instances: min\_tour\_length > L

• TSP language is the set of Yes Instances.
  – It is an NP-complete language
Chomsky Hierarchy of Languages

• Chomsky languages:
  – Recursively enumerable (RE) languages
  – Context-sensitive (CS) language
  – Context-free (CF) languages
  – Regular (REG) languages

• REG $\subseteq$ CF $\subseteq$ CS $\subseteq$ RE

• RE are most general languages known.
Recognition of Languages

• A computer **accepts** a **string** if it causes machine to move to an **accept state**.

• A computer **recognizes** a **language** if it accepts all strings in the language and nothing else.
  – Every RE language recognized by a TM
  – Every CF language recognized by a pushdown automaton
  – Every REG language recognized by an FSM

• We study **grammars** for these languages.
Nondeterministic Computation

• Standard computational models are deterministic
  – There is only one next state

• **Nondeterminism** allows for multiple next states
  – A string is accepted if there’s a path to an accept state
  – The language recognized is the set of accepted strings

• **Nondeterministic computations** are **not** natural but they do simplify problem descriptions.

• They describe the **NP**-complete languages
Complexity Measures

• Measures used in the course:
  – **Circuit size** (number of gates)
  – **Circuit depth** (length of longest path)
  – **Time** and **space** on serial machines with memory

• Measures are used for both deterministic and nondeterministic computations.
The Role of Simulation

• A simulation is an algorithm that results from the reduction of one computation to another.  
  – We show that a T-step FSM computation is easily simulated by a logic circuit of size $O(T)$.
  – If the simulation can be done efficiently, so can the simulated problem.

• T-step TM is an FSM and has circuit simulation

• Our first $\mathbf{NP}$-complete problem is obtained by simulating a nondeterministic TM computation.
Review of Instructional Material

- Problems defined by functions and languages
- Models of computation
  - Circuit, FSM, RAM, TM
  - Deterministic and nondeterministic
  - Functions and languages computed by machines
- Hard and impossible problems
- Classifying problems by simulation & reduction
- Complexity measures
Assignments

• Eleven homework assignments
  – Coordinated with the lectures
  – Labs help you understand assignments, develop confidence with proofs, and help complete homeworks.

• Exams
  – Two midterms and a final