Final Homework Assignment

• Due Wednesday, March 18 11:00 pm
• Submit in Gradescope: no grinch
  – Worth > regular homework and < midterm

• For individual questions for me or the CSE 311 staff between now and then use the Ed discussion board.
  – Mark the “Private” checkbox near the bottom of the New Thread creation page

• Because previous assignments will now end up being worth more than they were:
  – We will compute your “Best 7 of 8” for those grades
  – The Final Homework is not part of this.
Review: Countability vs Uncountability

• To prove a set $A$ countable you must show
  – There exists a listing $x_1, x_2, x_3, \ldots$ such that every element of $A$ is in the list.

• To prove a set $B$ uncountable you must show
  – For every listing $x_1, x_2, x_3, \ldots$ there exists some element in $B$ that is not in the list.

  – The diagonalization proof shows how to describe a missing element $d$ in $B$ based on the listing $x_1, x_2, x_3, \ldots$.

  *Important*: the proof produces a $d$ no matter what the listing is.
Last time: Undecidability of the Halting Problem

\text{\texttt{CODE(}}P\text{\texttt{)}} means “the code of the program \texttt{P}”

\begin{center}
\textbf{The Halting Problem}
\end{center}

\textbf{Given:} - \texttt{CODE(P)} for any program \texttt{P}
- input \texttt{x}

\textbf{Output:} \texttt{true} if \texttt{P} halts on input \texttt{x}
\texttt{false} if \texttt{P} does not halt on input \texttt{x}

\textbf{Theorem [Turing]:} There is no program that solves the Halting Problem

\textbf{Proof:} By contradiction.

Assume that a program \texttt{H} solving the Halting program does exist. Then program \texttt{D} must exist
Does $D(\text{CODE}(D))$ halt?

$H$ solves the halting problem implies that $H(\text{CODE}(D), x)$ is true iff $D(x)$ halts, $H(\text{CODE}(D), x)$ is false iff not $D(x)$ halts.

Suppose that $D(\text{CODE}(D))$ halts.
Then, by definition of $H$ it must be that $H(\text{CODE}(D), \text{CODE}(D))$ is true.
Which by the definition of $D$ means $D(\text{CODE}(D))$ doesn’t halt.

Suppose that $D(\text{CODE}(D))$ doesn’t halt.
Then, by definition of $H$ it must be that $H(\text{CODE}(D), \text{CODE}(D))$ is false.
Which by the definition of $D$ means $D(\text{CODE}(D))$ halts.

The ONLY assumption was the program $H$ exists so that assumption must have been false.

Contradiction!

```java
public static void D(x) {
    if (H(x,x) == true) {
        while (true); /* don’t halt */
    }
    else {
        return; /* halt */
    }
}
```
The Halting Problem isn’t the only hard problem

• Can use the fact that the Halting Problem is undecidable to show that other problems are undecidable

General method:
Prove that if there were a program deciding B then there would be a way to build a program deciding the Halting Problem.

“B decidable → Halting Problem decidable”

Contrapositive:

“Halting Problem undecidable → B undecidable”

Therefore B is undecidable
Last time: A CSE 141 assignment

Students should write a Java program that:

– Prints “Hello” to the console
– Eventually exits

Gradelt, Practicelt, etc. need to grade the students.

How do we write that grading program?

WE CAN’T: THIS IS IMPOSSIBLE!
Last Time: A related undecidable problem

• HelloWorldTesting Problem:
  – Input: CODE(Q) and x
  – Output:
    True if Q outputs “HELLO WORLD” on input x
    False if Q does not output “HELLO WORLD” on input x

• Theorem: The HelloWorldTesting Problem is undecidable.
• Proof idea: Show that if there is a program T to decide HelloWorldTesting then there is a program H to decide the Halting Problem for code(P) and x.
Last time: The HaltsNoInput Problem

- Input: CODE(R) for program R
- Output: True if R halts without reading input
  
  False otherwise.

**Theorem:** HaltsNoInput is undecidable

General idea “hard-coding the input”:
- Show how to use CODE(P) and x to build CODE(R) so
  
P halts on input x \iff R halts without reading input
Last time

- The impossibility of writing the CSE 141 grading program follows by combining the ideas from the undecidability of HaltsNoInput and HelloWorld.
More Reductions

- Can use undecidability of these problems to show that other problems are undecidable.

- For instance:

  \[
  \text{EQUIV}(P, Q) : \begin{cases}
  \text{True} & \text{if } P(x) \text{ and } Q(x) \text{ have the same I/O behavior for every input } x \\
  \text{False} & \text{otherwise}
  \end{cases}
  \]
Rice’s theorem

Not every problem on programs is undecidable!

Which of these is decidable?

- Input CODE(P) and x
  Output: true if P prints “ERROR” on input x after less than 100 steps
  false otherwise

- Input CODE(P) and x
  Output: true if P prints “ERROR” on input x after more than 100 steps
  false otherwise

Rice’s Theorem (a.k.a. Compilers Suck Theorem - informal):
Any “non-trivial” property of the input-output behavior of Java programs is undecidable.
CFGs are complicated

We know can answer almost any question about

• Regular Expressions, DFAs, NFAs, FSMs

But many problems about CFGs are undecidable!

• Is there any string that two CFGs both accept?
• Do two CFGs accept the same language?
• Does a CFG accept every string?
Computers and algorithms

• Does Java (or any programming language) cover all possible computation? Every possible algorithm?

• There was a time when computers were people who did calculations on sheets paper to solve computational problems

• Computers as we known them arose from trying to understand everything these people could do.
Before Java

1930’s:

How can we formalize what algorithms are possible?

• **Turing machines** (Turing, Post)
  – basis of modern computers

• **Lambda Calculus** (Church)
  – basis for functional programming, LISP

• **µ-recursive functions** (Kleene)
  – alternative functional programming basis
Turing machines

**Church-Turing Thesis:**
Any reasonable model of computation that includes all possible algorithms is equivalent in power to a Turing machine.

**Evidence**
- Intuitive justification
- Huge numbers of models based on radically different ideas turned out to be equivalent to TMs
Turing machines

• **Finite Control**
  – Brain/CPU that has only a finite # of possible “states of mind”

• **Recording medium**
  – An unlimited supply of blank “scratch paper” on which to write & read symbols, each chosen from a finite set of possibilities
  – Input also supplied on the scratch paper

• **Focus of attention**
  – Finite control can only focus on a small portion of the recording medium at once
  – Focus of attention can only shift a small amount at a time
Turing machines

- **Recording medium**
  - An infinite read/write “tape” marked off into cells
  - Each cell can store one symbol or be “blank”
  - Tape is initially all blank except a few cells of the tape containing the input string
  - Read/write head can scan one cell of the tape - starts on input

- **In each step, a Turing machine**
  1. Reads the currently scanned cell
  2. Based on current state and scanned symbol
     i. Overwrites symbol in scanned cell
     ii. Moves read/write head left or right one cell
     iii. Changes to a new state

- Each Turing Machine is specified by its finite set of rules
### Turing machines

<table>
<thead>
<tr>
<th>Symbol</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td>$(1, L, s_3)$</td>
<td>$(0, R, s_2)$</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$(0, R, s_1)$</td>
<td>$(1, R, s_1)$</td>
</tr>
<tr>
<td>$s_3$</td>
<td>$(1, L, s_4)$</td>
<td>$(0, R, s_1)$</td>
</tr>
<tr>
<td>$s_4$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

- DFA: All R
  - $s_1$: 100111
  - $s_2$: 00011
  - $s_3$: 01011
  - $s_4$: 11
UW CSE’s Steam-Powered Turing Machine

Original in Sieg Hall stairwell
Turing machines

Ideal Java/C programs:
- Just like the Java/C you’re used to programming with, except you never run out of memory
  - Constructor methods always succeed
  - `malloc` in C never fails

Equivalent to Turing machines except a lot easier to program:
- Turing machine definition is useful for breaking computation down into simplest steps
- We only care about high level so we use programs
Turing’s big idea part 1: Machines as data

Original Turing machine definition:

– A different “machine” $M$ for each task
– Each machine $M$ is defined by a finite set of possible operations on a finite set of symbols
– So... $M$ has a finite description as a sequence of symbols, its “code”, which we denote $\langle M \rangle$

You already are used to this idea with the notion of the program code or text but this was a new idea in Turing’s time.
Turing’s big idea part 2: A Universal TM

• A Turing machine interpreter $U$
  – On input $<M>$ and its input $x$, $U$ outputs the same thing as $M$ does on input $x$
  – At each step it decodes which operation $M$ would have performed and simulates it.

• One Turing machine is enough
  – Basis for modern stored-program computer
    Von Neumann studied Turing’s UTM design

\[
\begin{align*}
\text{input} & \quad x \quad \text{output} \\
M & \quad \rightarrow & \quad M(x) \\
\text{input} & \quad <M> \quad \text{output} \\
U & \quad \rightarrow & \quad M(x)
\end{align*}
\]
Takeaway from undecidability

• You can’t rely on the idea of improved compilers and programming languages to eliminate major programming errors
  – truly safe languages can’t possibly do general computation

• Document your code
  – there is no way you can expect someone else to figure out what your program does with just your code; since in general it is provably impossible to do this!
We’ve come a long way!

- Propositional Logic.
- Boolean logic and circuits.
- Boolean algebra.
- Predicates, quantifiers and predicate logic.
- Inference rules and formal proofs for propositional and predicate logic.
- English proofs.
- Set theory.
- Modular arithmetic.
- Prime numbers.
- GCD, Euclid's algorithm and modular inverse
We’ve come a long way!

• Induction and Strong Induction.
• Recursively defined functions and sets.
• Structural induction.
• Regular expressions.
• Context-free grammars and languages.
• Relations and composition.
• Transitive-reflexive closure.
• Graph representation of relations and their closures.
We’ve come a long way!

• DFAs, NFAs and language recognition.
• Product construction for DFAs.
• Finite state machines with outputs at states.
• Minimization algorithm for finite state machines
• Conversion of regular expressions to NFAs.
• Subset construction to convert NFAs to DFAs.
• Equivalence of DFAs, NFAs, Regular Expressions
• Method to prove languages not accepted by DFAs.
• Cardinality, countability and diagonalization
• Undecidability: Halting problem and evaluating properties of programs.
What’s next? ...after the final homework...

• **Foundations II (CSE 312)**
  - Fundamentals of counting, discrete probability, applications of randomness to computing, statistical algorithms and analysis
  - Ideas critical for machine learning, algorithms

• **Data Abstractions (CSE 332)**
  - Data structures, a few key algorithms, parallelism
  - Brings programming and theory together
  - Makes heavy use of induction and recursive defns
Complexity Theory  (in CSE 431 and beyond)

Not just what can be computed at all...

How about what can be computed *efficiently*?

A rich, interesting, and important topic.
Thank you!

• For being a great class this quarter!

• For bearing with me/us during this trying time!

• Stay healthy!

• Come by my office and say “Hello” when all this is over...
Thank you!

That's all Folks!