Please fill out course evals online! I/We really want your feedback!
Final exam Monday, Review session Sunday

• **Monday** at either **2:30-4:20** or **4:30-6:20**
  
  – JHN 102
  
  – **Must select your exam time by Saturday**
    
    No changes permitted after that
  
  – Bring your **UW ID**

• **Comprehensive:** Full probs only on topics that were covered in homework. May have small probs on other topics.
  
  – May includes pre-midterm topics, e.g., formal proofs.
  
  – Reference sheets will be included. Closed book. No notes.

• **Review session:** *Sunday starting at 1 pm on Zoom*
  
  – Bring your questions !!
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.
The Halting Problem

Given:  - CODE(P) for any program P  
      - input x

Output: true if P halts on input x
        false if P does not halt on input x

Theorem [Turing]: There is no program that solves the Halting Problem

Proof: By contradiction.
Assume that a program H solving the Halting program does exist. Then program D must exist
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.

Contradiction!

The ONLY assumption was the program $H$ exists so that assumption must have been false.
No general procedure for bug checks succeeds. Now, I won’t just assert that, I’ll show where it leads: I will prove that although you might work till you drop, you cannot tell if computation will stop.

For imagine we have a procedure called $P$ that for specified input permits you to see whether specified source code, with all of its faults, defines a routine that eventually halts.

You feed in your program, with suitable data, and $P$ gets to work, and a little while later (in finite compute time) correctly infers whether infinite looping behavior occurs...
Here’s the trick that I’ll use – and it’s simple to do. I’ll define a procedure, which I will call $Q$, that will use $P$’s predictions of halting success to stir up a terrible logical mess.

And this program called $Q$ wouldn’t stay on the shelf; I would ask it to forecast its run on *itself*. When it reads its own source code, just what will it do? What’s the looping behavior of $Q$ run on $Q$?

Full poem at:
http://www.lel.ed.ac.uk/~gpullum/loopsnoop.html
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 you can use it to build a program deciding the Halting Problem.

1. “\( B \) decidable \( \rightarrow \) Halting Problem decidable” \( \text{Shown by general method} \)
2. “Halting problem undecidable” \( \text{Turing} \)
3. “Halting Problem undecidable \( \rightarrow \) \( B \) undecidable” \( \text{Contrapositive from 1} \)
4. “\( B \) undecidable” \( \text{Modus Ponens 2 & 3} \)
A CSE 121 assignment

Students should write a Java program that:

– Prints “Hello” to the console
– Eventually exits

Our auto-grading program needs to grade the students.

How do we write that grading program?

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

- Suppose there is a program $T$ that solves the \texttt{HelloWorldTesting} problem. Define program $H$ that takes input $\text{CODE}(P)$ and $x$ and does the following:
  - Creates $\text{CODE}(Q)$ from $\text{CODE}(P)$ by
    1. removing all output statements from $\text{CODE}(P)$, and
    2. adding a `System.out.println("HELLO WORLD")` immediately before any spot where $P$ could halt
  Then runs $T$ on input $\text{CODE}(Q)$ and $x$.

- If $P$ halts on input $x$ then $Q$ prints HELLO WORLD and halts and so $H$ outputs \texttt{true} (because $T$ outputs true on input $\text{CODE}(Q)$)
- If $P$ doesn’t halt on input $x$ then $Q$ won’t print anything since we removed any other print statement from $\text{CODE}(Q)$ so $H$ outputs \texttt{false}

We know that such an $H$ cannot exist. Therefore $T$ cannot exist.
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 ⇔ R halts without reading input
The HaltsNoInput Problem

“Hard-coding the input”:

• Show how to use CODE(P) and x to build CODE(R) so P halts on input x ⇔ R halts without reading input

• Replace input statement in CODE(P) that reads input x into variable var, by a hard-coded assignment statement:

  var = x

  to produce CODE(R).

• So if we have a program N to decide HaltsNoInput then we can use it as a subroutine as follows to decide the Halting Problem, which we know is impossible:
  – On input CODE(P) and x, produce CODE(R). Then run N on input CODE(R) and output the answer that N gives.
• The impossibility of writing the **CSE 121** 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:
  \[ \text{EQUIV}(P, Q) : \]
  \[
  \begin{align*}
  \text{True} & \quad \text{if } P(x) \text{ and } Q(x) \text{ have the same behavior for every input } x \\
  \text{False} & \quad \text{otherwise}
  \end{align*}
  \]
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.
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>s₁</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, L, s₃)</td>
<td>(1, L, s₄)</td>
<td>(0, R, s₂)</td>
</tr>
<tr>
<td>s₂</td>
<td>(0, R, s₁)</td>
<td>(1, R, s₁)</td>
</tr>
</tbody>
</table>

- | - | 1 | 1 | 0 | 1 | 1 | - | - |
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” $\mathbf{M}$ for each task
– Each machine $\mathbf{M}$ is defined by a finite set of possible operations on finite set of symbols
– So... $\mathbf{M}$ has a finite description as a sequence of symbols, its “code”, which we denote $\langle \mathbf{M} \rangle$

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

\[
\begin{align*}
\text{input} & \quad x \quad \rightarrow \quad M \quad \rightarrow \quad M(x) \\
\text{output} & \\
\text{input} & \quad <M> \quad \rightarrow \quad U \quad \rightarrow \quad M(x)
\end{align*}
\]
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, modular inverse, and exponentiation.
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
- Finite automata for pattern matching.
- 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 exam...

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

• **Data Abstractions** (332)
  – Data structures, a few key algorithms, parallelism
  – Brings programming and theory together
  – Makes heavy use of induction and recursive defns
Course Evaluation Online

• Fill this out by Sunday night!
  – Your ability to fill it out will disappear at 11:59 p.m. on Sunday.
  – We really value your feedback!
Final exam Monday, Review session Sunday

• **Monday** at either **2:30-4:20** or **4:30-6:20**
  – JHN 102
  – Must select your exam time by Saturday
    No changes permitted after that
  – Bring your **UW ID**

• **Comprehensive:** Full probs only on topics that were covered in homework. May have small probs on other topics.
  – May includes pre-midterm topics, e.g., formal proofs.
  – Reference sheets will be included. Closed book. No notes.

• **Review session:** *Sunday starting at 1 pm on Zoom*
  – Bring your questions !!