CSE 311: Foundations of Computing

Lecture 28: Undecidability and Reductions

```
DEFINE DOESITHALT(PROGRAM):
{
    RETURN TRUE;
}

THE BIG PICTURE SOLUTION
TO THE HALTING PROBLEM

I have regraded midterms
for you to pick up.

Changed location
for review session.
EEB 125

```
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

**CODE(P)** means “the code of the program P”

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

We use the term “undecidable” since it is a true/false question that is not computable.

Why, intuitively, is this hard?
This really is a legal Java program...

What does this program do?

```java
_(_,___,____){___/___<=1?_(___,___+1,___):!
(___%___)?(_,___+1,0):___%___==___/___
&&!_____?(printf("\d\t",___/___),(_,_,____+1,0)):___%___>1&&___%___<___/
?(_,1+_____,____+!(___/__%(___%___))):___<___*___?
(_,___+1,____):0;}main(){_(100,0,0);}
```
public static void collatz(n) {
    if (n == 1) {
        return 1;
    }
    if (n % 2 == 0) {
        return collatz(n/2)
    } else {
        return collatz(3*n + 1)
    }
}

What does this program do?
  ... on n=11?
  ... on n=10000000000000000001?
public static void collatz(n) {
    if (n == 1) {
        return 1;
    }
    if (n % 2 == 0) {
        return collatz(n/2)
    }
    else {
        return collatz(3*n + 1)
    }
}

Nobody knows whether or not this program halts on all inputs!

What does this program do? Trying to solve this has been called a “mathematical disease”.

... on n=11?
... on n=10000000000000000001?
Last time: Undecidability of the Halting Problem

CODE(P) means “the code of the program P”

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

Note: Even though the program \( D \) has a
\textbf{while}(true), that doesn’t mean that the
program \( D \) actually goes into an infinite
loop on input \( x \), which is what \( H \) has to
determine
Does \texttt{D(CODE(D))} halt?

\texttt{public static void D(x) \{ if (H(x,x) == true) \{ while (true); /* don't halt */ \} else \{ return; /* halt */ \} \}}

\texttt{H} solves the halting problem implies that \texttt{H(CODE(D),x)} is \texttt{true} iff \texttt{D(x)} halts, \texttt{H(CODE(D),x)} is \texttt{false} iff \texttt{D(x)} doesn't halt.

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

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

The ONLY assumption was the program \texttt{H} exists so that assumption must have been false.
Contradiction!
Where did the idea for creating D come from?
Connection to diagonalization

Write \(<P>\) for CODE(P)

Some possible inputs x

\[\begin{array}{cccccc}
< P_1 > & < P_2 > & < P_3 > & < P_4 > & < P_5 > & < P_6 > & \ldots \\
P_1 & P_2 & P_3 & P_4 & P_5 & P_6 & \ldots \\
P_3 & P_4 & P_5 & P_6 & P_7 & P_8 & \ldots \\
P_7 & P_8 & P_9 & \ldots & \ldots & \ldots & \ldots \\
\end{array}\]

This listing of all programs really does exist since the set of all Java programs is countable.

The goal of this “diagonal” argument is not to show that the listing is incomplete but rather to show that a “flipped” diagonal element is not in the listing.
Connection to diagonalization  

<table>
<thead>
<tr>
<th>P</th>
<th>&lt;P₁&gt;</th>
<th>&lt;P₂&gt;</th>
<th>&lt;P₃&gt;</th>
<th>&lt;P₄&gt;</th>
<th>&lt;P₅&gt;</th>
<th>&lt;P₆&gt;</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P₂</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P₃</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P₄</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>P₅</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P₆</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P₇</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P₈</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
| P₉  | .    | .    | .    | .    | .    | .    | .   | .   | .    | .    | .    | .    | ...

(P,x) entry is 1 if program P halts on input x and 0 if it runs forever

Write <P> for CODE(P)
Connection to diagonalization

Write \(<P>\) for CODE(P)

<table>
<thead>
<tr>
<th>All programs P</th>
<th>(&lt;P_1&gt;)</th>
<th>(&lt;P_2&gt;)</th>
<th>(&lt;P_3&gt;)</th>
<th>(&lt;P_4&gt;)</th>
<th>(&lt;P_5&gt;)</th>
<th>(&lt;P_6&gt;)</th>
<th>(&lt;P_7&gt;)</th>
<th>(\ldots)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P_1)</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(P_2)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(P_3)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(P_4)</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(P_5)</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(P_6)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(P_7)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(P_8)</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(P_9)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Some possible inputs \(x\)

Want behavior of program \(D\) to be like the flipped diagonal, so it can’t be in the list of all programs.

\((P,x)\) entry is 1 if program \(P\) halts on input \(x\) and 0 if it runs forever.
Where did the idea for creating D come from?

public static void D(x) {
    if (H(x, x) == true) {
        while (true); /* don’t halt */
    } else {
        return; /* halt */
    }
}

D halts on input code(P) iff H(code(P), code(P)) doesn’t halt
iff P doesn’t halt on input code(P)

Therefore for any program P, D differs from P on input code(P)
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 \rightarrow \text{Halting Problem decidable}”

Contrapositive:

  “\text{Halting Problem undecidable} \rightarrow B \text{ undecidable}”

Therefore B is undecidable
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!
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 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 $\text{CODE}(Q)$ prints HELLO WORLD and halts and so $H$ outputs \text{true} (because $T$ outputs \text{true} on input $\text{CODE}(Q)$)

• If $P$ doesn’t halt on input $x$ then $\text{CODE}(Q)$ won’t print anything since we removed any other print statement from $\text{CODE}(Q)$ so $H$ outputs \text{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 \text{CODE}(P) and \text{x} to build \text{CODE}(R) so \text{P} halts on input \text{x} ⇔ \text{R} halts without reading input.

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

\[
\text{var} = \text{x}
\]

to produce \text{CODE}(R).

• So if we have a program \text{N} to decide \text{HaltsNoInput} then we can use it as a subroutine as follows to decide the Halting Problem, which we know is impossible:

– On input \text{CODE}(P) and \text{x}, produce \text{CODE}(R). Then run \text{N} on input \text{CODE}(\text{R}) and output the answer that \text{N} gives.
• 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:

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

Not every problem on programs is undecidable!

Which of these is decidable?

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

• Input \text{CODE}(P) \text{ 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

• **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 equivalent models to TM’s based on radically different ideas
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**
  – Reads the currently scanned symbol
  – Based on current state and scanned symbol
    Overwrites symbol in scanned cell
    Moves read/write head left or right one cell
    Changes to a new state

• **Each Turing Machine is specified by its finite set of rules**