DEFINE DOES IT HALT (PROGRAM):
{
    RETURN TRUE;
}

THE BIG PICTURE SOLUTION TO THE HALTING PROBLEM
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

\textbf{CODE}(P)\textsuperscript{(*)} means "the code of the program P"

\begin{tcolorbox}
\textbf{The Halting Problem}

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

\textbf{Output:} \textsf{true} if \(P\) halts on input \(x\)
\textsf{false} if \(P\) does not halt on input \(x\)
\end{tcolorbox}

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

We use the term "\textit{undecidable}" since it is a true/false question that is not computable.

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

What does this program do?

```c
__(_,___,____){___/__ <=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?
11 34 17 52 26 13 40 20 10 516 8421

... on n=10000000000000000001?
A “Simple” Program

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?

... on n=11?

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

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

The Halting Problem

**Given:**
- \( \text{CODE}(P) \) for any program \( P \)
- input \( x \)

**Output:**
- \text{true} if \( P \) halts on input \( x \)
- \text{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
Does $D(CODE(D))$ halt?

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

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

Note: Even though the program $D$ has a `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
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

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

Contradiction!

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

<table>
<thead>
<tr>
<th>All programs P</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>P₂</td>
<td>P₃</td>
<td>P₄</td>
<td>P₅</td>
<td>P₆</td>
<td>P₇</td>
</tr>
<tr>
<td>P₈</td>
<td>P₉</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>

Some possible inputs x

Write $<P>$ for CODE(P)

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

Write $<P>$ for $\text{CODE}(P)$

<table>
<thead>
<tr>
<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>$&lt;P_8&gt;$</th>
<th>$&lt;P_9&gt;$</th>
</tr>
</thead>
</table>
| P1      | 0       | 1       | 1       | 0       | 1       | 1       | 1       | 0       | 0       | 0       | 0       | 1       |...
| P2      | 1       | 1       | 0       | 1       | 0       | 1       | 1       | 0       | 1       | 1       | 1       |...
| P3      | 1       | 0       | 1       | 0       | 0       | 0       | 0       | 0       | 0       | 0       | 0       | 1       |...
| P4      | 0       | 1       | 1       | 0       | 1       | 0       | 1       | 1       | 0       | 1       | 1       | 1       |...
| P5      | 0       | 1       | 1       | 1       | 1       | 1       | 1       | 0       | 0       | 0       | 0       | 1       |...
| P6      | 1       | 1       | 0       | 0       | 0       | 1       | 1       | 0       | 1       | 1       | 1       | 1       |...
| P7      | 1       | 0       | 1       | 1       | 0       | 0       | 0       | 0       | 0       | 0       | 0       | 0       | 1       |...
| P8      | 0       | 1       | 1       | 1       | 1       | 0       | 1       | 1       | 0       | 1       | 0       |...
| P9      | .       | .       | .       | .       | .       | .       | .       | .       | .       | .       | .       | .       |...

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

Some possible inputs $x$
**Connection to diagonalization**

<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>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>01</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_2$</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_3$</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_4$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>01</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$P_5$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>10</td>
<td>11</td>
<td>00</td>
</tr>
<tr>
<td>$P_6$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>00</td>
<td>10</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>$P_7$</td>
<td>1</td>
<td>0</td>
<td>11</td>
<td>10</td>
<td>00</td>
<td>01</td>
<td>00</td>
</tr>
<tr>
<td>$P_8$</td>
<td>0</td>
<td>1</td>
<td>11</td>
<td>11</td>
<td>01</td>
<td>10</td>
<td>01</td>
</tr>
<tr>
<td>$P_9$</td>
<td>. . . . . . . . . . . . . . . . . . . . . . . .</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

*Write $<P>$ for CODE($P$)*

Some possible inputs $x$

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

Want behavior of program $D$ to be like the flipped diagonal, so it can’t be in the list of all programs.
Where did the idea for creating D come from?

```java
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 \( H(\text{code}(P), \text{code}(P)) \) 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 $\rightarrow$ Halting Problem decidable”

Contrapositive:

“Halting Problem undecidable $\rightarrow$ $B$ 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 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 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:

  \[ \text{var} = x \]

  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(Q) and output the answer that 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:

  $\text{EQUIV}(P, Q)$:  
  
  **True**  if $P(x)$ and $Q(x)$ have the same behavior for every input $x$
  
  **False**  otherwise
Rice’s theorem

Not every problem on programs is undecidable!

Which of these is decidable?

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

- **Input** \( \text{CODE}(P) \) 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.