Unsolvable Problems
Part One
What problems can we solve with a computer?

What does it mean to solve a problem?
Equivalence of TMs and Programs

Here's a sample program we might use to model a Turing machine for \( \{ w \in \{a, b\}^* \mid w \text{ has the same number of } a's \text{ and } b's \} \):

```c
int main() {
    string input = getInput();
    int difference = 0;

    for (char ch: input) {
        if (ch == 'a') difference++;
        else if (ch == 'b') difference--;
        else reject();
    }

    if (difference == 0) accept();
    else reject();
}
```
Equivalence of TMs and Programs

- As mentioned before, it's always possible to build a method `mySource()` into a program, which returns the source code of the program.
- For example, here's a narcissistic program:

```cpp
int main() {
    string me = mySource();
    string input = getInput();

    if (input == me) accept();
    else reject();
}
```
Self-Referential : Danger:

• Last time, we saw examples of self-reference that were fine, and some that created paradoxes:

True or False?

"This string is 34 characters long."
"This sentence is false."
Self-Reference : Danger:

• Last time, we saw examples of self-reference that were fine, and some that created paradoxes:

True or False?

"This string is 34 characters long."
"This sentence is false."
"This sentence is true."
Self-Reference in Set Theory

Now that we know that self-reference is dangerous (i.e., can lead to paradoxes) in propositions (e.g., “This sentence is false”), we can look at it in other domains we’ve studied, like Set Theory.

- We know that sets can contain other sets.
- They can even contain themselves (“The set of all sets” and “The set of all sets with infinite cardinality” both contain themselves).
- Since we know that self-reference is dangerous, we might want to make a set called SAFE_LIST that is the set of all sets that do not contain themselves, since any set on that list is safe from paradoxes.
True or False?

"The set of all sets that do not contain themselves contains itself."

Answer at PollEv.com/cs103 or text CS103 to 22333 once to join, then your answer.
Proofs by Contradiction in Number Theory

• One way to think about proofs by contradiction is that they lead to a kind of “impossible” situation that is similar to the paradoxes. Here is a simple example:

  • **Thm.** There is no greatest integer.
  • **Proof, by contradiction.** Assume for the sake of contradiction that there is a greatest integer, call it $g$.

    • *Now we will use $g$ to write a mathematical expression that is a syntactically valid mathematical expression that should be fine to write, if $g$ were real.]*
    • Let $x = g + 1$.
    • We see that $x > g$.
    • But this is a contradiction, because $g$ is the greatest integer.
    • So the assumption is false and the theorem is true. ■
One way to think about proofs by contradiction is that they lead to a kind of “impossible” situation that is similar to the paradoxes. Here is a simple example:

**Thm.** There is no greatest integer.

**Proof, by contradiction.** Assume for the sake of contradiction that there is a greatest integer, call it $g$.

[Now we will use $g$ to write a syntactically valid mathematical expression that should be fine to write, if $g$ were real.]

Let $x = g + 1$.

We see that $x > g$.

But this is a contradiction, because $g$ is the greatest integer.

So the assumption is false and the theorem is true. ■
More Self-Reference!

(this time with Turing Machines)
The Problem of Looping TMs

- Suppose we have a TM $M$ and a string $w$.
- If we run $M$ on $w$, we may never find out whether $w \in \mathcal{L}(M)$ because $M$ might loop on $w$.
- Is there some algorithm we can use to determine whether $M$ is eventually going to accept $w$?
A Decider for $A_{TM}$?

- **Recall**: $A_{TM}$ is the language of the universal Turing machine.
- We know that $\langle M, w \rangle \in A_{TM}$ if and only if $M$ accepts $w$.
- The universal Turing machine $U_{TM}$ is a recognizer for $A_{TM}$. Could we build a decider for $A_{TM}$?
A Decider for $A_{\text{TM}}$?

- Suppose that $A_{\text{TM}} \in \mathbb{R}$.
- Formally, this means that there is a TM that decides $A_{\text{TM}}$.
- Intuitively, this means that there is a TM that takes as input a TM $M$ and string $w$, then
  - accepts if $M$ accepts $w$, and
  - rejects if $M$ does not accept $w$. 
A Decider for `willAccept`?

- To make the previous discussion more concrete, let's explore the analog for computer programs.
- If $A_{TM}$ is decidable, we could construct a function
  
  ```
  bool willAccept(string program, string input)
  ```

  that takes in as input a program and a string, then returns true if the program will accept the input and false otherwise.

- What could we do with this?
• If \( A_{TM} \) is decidable, we could construct a function

\[
\text{bool willAccept(string program, string input)}
\]

that takes in as input a program and a string, then returns true if the program will accept the input and false otherwise.

How many of the following statements are true?

- willAccept("\texttt{int main() \{ accept(); \}}", "Emu") returns true.
- willAccept("\texttt{int main() \{ reject(); \}}", "Yak") returns false.
- willAccept("\texttt{int main() \{ while (true) \{ \}} \}"", "Cow") loops forever.

Answer at PollEv.com/cs103 or text \texttt{CS103} to \texttt{22333} once to join, then a number.
What does this program do?

```cpp
bool willAccept(string program, string input) {
    /* ... some implementation ... */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```
What does this program do?

```c
bool willAccept(string program, string input) {
    /* ... some implementation ... */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```

How many of the following statements are true?

- This program accepts at least one input.
- This program rejects at least one input.
- This program loops on at least one input.

Answer at PollEv.com/cs103 or text CS103 to 22333 once to join, then a number.
What does this program do?

```cpp
bool willAccept(string program, string input) {
    /* ... some implementation ... */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}

Try running this program on any input. What happens if

... this program accepts its input?
What does this program do?

```c
bool willAccept(string program, string input) {
    /* ... some implementation ... */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```

Try running this program on any input.

What happens if...

... this program accepts its input?

It rejects the input!

... this program doesn't accept its input?
What does this program do?

```cpp
bool willAccept(string program, string input) {
    /* ... some implementation ... */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```

Try running this program on any input. What happens if

... this program accepts its input? It rejects the input!

... this program doesn't accept its input? It accepts the input!
Knowing the Future

- This TM is analogous to a classical philosophical/logical paradox:

  *If you know what you are fated to do, can you avoid your fate?*

- If $A_{TM}$ is decidable, we can construct a TM that determines what it's going to do in the future (whether it will accept its input), then actively chooses to do the opposite.

- This leads to an impossible situation with only one resolution: $A_{TM}$ must not be decidable!
Theorem: $A_{TM} \notin R$. 

Proof: By contradiction; assume that $A_{TM} \in R$. Then there is some decider $D$ for $A_{TM}$, which we can represent in software as a method $\text{willAccept}$ that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise. Given this, we could then construct this program $P$:

```cpp
int main() {
    string me = mySource();
    string input = getInput();
    if (willAccept(me, input)) reject();
    else accept();
}
```

Choose any string $w$ and trace through the execution of program $P$ on input $w$, focusing on the answer given back by the $\text{willAccept}$ method. If $\text{willAccept(me, input)}$ returns true, then $P$ must accept its input $w$. However, in this case $P$ proceeds to reject its input $w$. Otherwise, if $\text{willAccept(me, input)}$ returns false, then $P$ must not accept its input $w$. However, in this case $P$ proceeds to accept its input $w$. In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, $A_{TM} \notin R$. ■
Theorem: $A_{TM} \notin R$.

Proof: By contradiction; assume that $A_{TM} \in R$.
**Theorem:** $A_{TM} \notin R$.

**Proof:** By contradiction; assume that $A_{TM} \in R$. Then there is some decider $D$ for $A_{TM}$, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

```c
int main() {
    string me = mySource();
    string input = getInput();
    if (willAccept(me, input)) reject();
    else accept();
}
```

Choose any string $w$ and trace through the execution of program $P$ on input $w$, focusing on the answer given back by the `willAccept` method. If `willAccept(me, input)` returns true, then $P$ must accept its input $w$. However, in this case $P$ proceeds to reject its input $w$. Otherwise, if `willAccept(me, input)` returns false, then $P$ must not accept its input $w$. However, in this case $P$ proceeds to accept its input $w$. In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, $A_{TM} \notin R$. ■
**Theorem:** $A_{TM} \notin R$.

**Proof:** By contradiction; assume that $A_{TM} \in R$. Then there is some decider $D$ for $A_{TM}$, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Given this, we could then construct this program $P$:

```c
int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) reject();
    else accept();
}
```
**Theorem:** $A_{TM} \notin R$.

**Proof:** By contradiction; assume that $A_{TM} \in R$. Then there is some decider $D$ for $A_{TM}$, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Given this, we could then construct this program $P$:

```java
int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) reject();
    else accept();
}
```

Choose any string $w$ and trace through the execution of program $P$ on input $w$, focusing on the answer given back by the `willAccept` method.
**Theorem:** $A_{TM} \not\in R$.

**Proof:** By contradiction; assume that $A_{TM} \in R$. Then there is some decider $D$ for $A_{TM}$, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Given this, we could then construct this program $P$:

```java
int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) reject();
    else accept();
}
```

Choose any string $w$ and trace through the execution of program $P$ on input $w$, focusing on the answer given back by the `willAccept` method. If `willAccept(me, input)` returns true, then $P$ must accept its input $w$.
**Theorem:** $A_{TM} \notin R$.

**Proof:** By contradiction; assume that $A_{TM} \in R$. Then there is some decider $D$ for $A_{TM}$, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Given this, we could then construct this program $P$:

```c
int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) reject();
    else accept();
}
```

Choose any string $w$ and trace through the execution of program $P$ on input $w$, focusing on the answer given back by the `willAccept` method. If `willAccept(me, input)` returns true, then $P$ must accept its input $w$. However, in this case $P$ proceeds to reject its input $w$. 
**Theorem:** \( A_{TM} \notin \mathbb{R} \).

**Proof:** By contradiction; assume that \( A_{TM} \in \mathbb{R} \). Then there is some decider \( D \) for \( A_{TM} \), which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Given this, we could then construct this program \( P \):

```plaintext
int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) reject();
    else accept();
}
```

Choose any string \( w \) and trace through the execution of program \( P \) on input \( w \), focusing on the answer given back by the `willAccept` method. If `willAccept(me, input)` returns true, then \( P \) must accept its input \( w \). However, in this case \( P \) proceeds to reject its input \( w \). Otherwise, if `willAccept(me, input)` returns false, then \( P \) must not accept its input \( w \).
**Theorem:** $A_{TM} \notin R$.

**Proof:** By contradiction; assume that $A_{TM} \in R$. Then there is some decider $D$ for $A_{TM}$, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Given this, we could then construct this program $P$:

```c
int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) reject();
    else accept();
}
```

Choose any string $w$ and trace through the execution of program $P$ on input $w$, focusing on the answer given back by the `willAccept` method. If `willAccept(me, input)` returns true, then $P$ must accept its input $w$. However, in this case $P$ proceeds to reject its input $w$. Otherwise, if `willAccept(me, input)` returns false, then $P$ must not accept its input $w$. However, in this case $P$ proceeds to accept its input $w$.
**Theorem:** $A_{TM} \notin R$.

**Proof:** By contradiction; assume that $A_{TM} \in R$. Then there is some decider $D$ for $A_{TM}$, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Given this, we could then construct this program $P$:

```c
int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) reject();
    else accept();
}
```

Choose any string $w$ and trace through the execution of program $P$ on input $w$, focusing on the answer given back by the `willAccept` method. If `willAccept(me, input)` returns true, then $P$ must accept its input $w$. However, in this case $P$ proceeds to reject its input $w$. Otherwise, if `willAccept(me, input)` returns false, then $P$ must not accept its input $w$. However, in this case $P$ proceeds to accept its input $w$.

In both cases we reach a contradiction, so our assumption must have been wrong.
**Theorem:** $A_{TM} \notin R$.

**Proof:** By contradiction; assume that $A_{TM} \in R$. Then there is some decider $D$ for $A_{TM}$, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Given this, we could then construct this program $P$:

```cpp
int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) reject();
    else accept();
}
```

Choose any string $w$ and trace through the execution of program $P$ on input $w$, focusing on the answer given back by the `willAccept` method. If `willAccept(me, input)` returns true, then $P$ must accept its input $w$. However, in this case $P$ proceeds to reject its input $w$. Otherwise, if `willAccept(me, input)` returns false, then $P$ must not accept its input $w$. However, in this case $P$ proceeds to accept its input $w$.

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, $A_{TM} \notin R$. 

$\blacksquare$
**Theorem:** $A_{TM} \not\in R$.

**Proof:** By contradiction; assume that $A_{TM} \in R$. Then there is some decider $D$ for $A_{TM}$, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Given this, we could then construct this program $P$:

```cpp
int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) reject();
    else accept();
}
```

Choose any string $w$ and trace through the execution of program $P$ on input $w$, focusing on the answer given back by the `willAccept` method. If `willAccept(me, input)` returns true, then $P$ must accept its input $w$. However, in this case $P$ proceeds to reject its input $w$. Otherwise, if `willAccept(me, input)` returns false, then $P$ must not accept its input $w$. However, in this case $P$ proceeds to accept its input $w$.

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, $A_{TM} \not\in R$. ■
Regular Languages
CFLs
All Languages
R
RE
A_{TM}
All Languages
What Does This Mean?

• In one fell swoop, we've proven that
  • $A_{TM}$ is undecidable; there is no general algorithm that can determine whether a TM will accept a string.
  • $R \neq RE$, because $A_{TM} \notin R$ but $A_{TM} \in RE$.
• What do these two statements really mean? As in, why should you care?
\[ A_{TM} \notin R \]

- The proof we've done says that
  
  \textit{There is no possible way to design an algorithm that will determine whether a program will accept an input.}

- Notice that our proof just assumed there was some decider for \( A_{TM} \) and didn't assume anything about how that decider worked. In other words, no matter how you try to implement a decider for \( A_{TM} \), you can never succeed!
At a more fundamental level, the existence of undecidable problems tells us the following:

There is a difference between what is true and what we can discover is true.

Given an TM and any string w, either the TM accepts the string or it doesn't – but there is no algorithm we can follow that will always tell us which it is!
$A_{TM} \notin R$

- What exactly does it mean for $A_{TM}$ to be undecidable?

  **Intuition:** The only general way to find out what a program will do is to run it.

- As you'll see, this means that it's provably impossible for computers to be able to answer questions about what a program will do.
\[ R \neq RE \]

- Because \( R \neq RE \), there are some problems where “yes” answers can be checked, but there is no algorithm for deciding what the answer is.

- *In some sense, it is fundamentally harder to solve a problem than it is to check an answer.*
More Impossibility Results
The Halting Problem

• The most famous undecidable problem is the **halting problem**, which asks:

  **Given a TM** $M$ **and a string** $w$, **will** $M$ **halt when run on** $w$?

• As a formal language, this problem would be expressed as

  $HALT = \{ \langle M, w \rangle \mid M \text{ is a TM that halts on } w \}$

• How hard is this problem to solve?
• How do we know?
\[\text{HALT} \in \text{RE}\]

- **Claim:** \(\text{HALT} \in \text{RE}\).
- **Idea:** If you were certain that a TM \(M\) halted on a string \(w\), could you convince me of that?
- Yes – just run \(M\) on \(w\) and see what happens!

```c
int main() {
    TM M = getInputTM();
    string w = getInputString();

    feed w into M;
    while (true) {
        if (M is in an accepting state) accept();
        else if (M is in a rejecting state) accept();
        else simulate one more step of M running on w;
    }
}
```
Claim: $HALT \notin R$.  

If $HALT$ is decidable, we could write some function

```python
bool willHalt(string program, string input)
```

that accepts as input a program and a string input, then reports whether the program will halt when run on the given input.

Then, we could do this...
What does this program do?

```cpp
bool willHalt(string program, string input) {
    /* ... some implementation ... */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willHalt(me, input)) {
        while (true) {
            // loop infinitely
        }
    } else {
        accept();
    }
}
```
What does this program do?

Imagine running this program on some input. What happens if...

... this program halts on that input?

... this program loops on this input?

It halts on the input!

It loops on the input!
What does this program do?

```c
bool willHalt(string program, string input) {
    /* ... some implementation ... */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willHalt(me, input)) {
        while (true) {
            // loop infinitely
        }
    } else {
        accept();
    }
}
```

Imagine running this program on some input. What happens if...

... this program halts on that input? It loops on the input!
What does this program do?

```cpp
bool willHalt(string program, string input) {
    /* ... some implementation ... */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willHalt(me, input)) {
        while (true) {
            // loop infinitely
        }
    } else {
        accept();
    }
}
```

Imagine running this program on some input. What happens if...

...this program halts on that input?
It loops on the input!

...this program loops on this input?
What does this program do?

```c++
bool willHalt(string program, string input) {
    /* ... some implementation ... */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willHalt(me, input)) {
        while (true) {
            // loop infinitely
        }
    }
    else {
        accept();
    }
}
```

Imagine running this program on some input. What happens if...

... this program halts on that input? It loops on the input!

... this program loops on this input? It halts on the input!
**Theorem:** \(\text{HALT} \notin R\).

**Proof:** By contradiction; assume that \(\text{HALT} \in R\). Then there’s a decider \(D\) for \(\text{HALT}\), which we can represent in software as a method \(\text{willHalt}\) that takes as input the source code of a program and an input, then returns true if the program halts on the input and false otherwise.

Given this, we could then construct this program \(P\):

```c
int main() {
    string me = mySource();
    string input = getInput();

    if (willHalt(me, input)) while (true) { /* loop! */ }
    else accept();
}
```

Choose any string \(w\) and trace through the execution of program \(P\) on input \(w\), focusing on the answer given back by the \(\text{willHalt}\) method. If \(\text{willHalt}(\text{me}, \text{input})\) returns true, then \(P\) must halt on its input \(w\). However, in this case \(P\) proceeds to loop infinitely on \(w\). Otherwise, if \(\text{willHalt}(\text{me}, \text{input})\) returns false, then \(P\) must not halt its input \(w\). However, in this case \(P\) proceeds to accept its input \(w\).

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, \(\text{HALT} \notin R\). ■
So What?

- These problems might not seem all that exciting, so who cares if we can't solve them?
- Turns out, this same line of reasoning can be used to show that some very important problems are impossible to solve.
Next Time

- *Intuiting RE*
  - What exactly is the class \textbf{RE} all about?
- *Verifiers*
  - A totally different perspective on problem solving.
- *Beyond RE*
  - Finding an impossible problem using very familiar techniques.