Unsolvable Problems
Part One
Outline for Today

- **Self-Reference Revisited**
  - Programs that compute on themselves.
- **Self-Defeating Objects**
  - Objects “too powerful” to exist.
- **The Fortune Teller**
  - Can you escape the future?
- **Why Do Programs Loop?**
  - ... and can we eliminate loops?
- **Undecidable Problems**
  - Something beyond the reach of algorithms.
Recap from Last Time
R and RE

• A language $L$ is **recognizable** if there is a TM $M$ with the following property:

  $\forall w \in \Sigma^*. (M \text{ accepts } w \leftrightarrow w \in L)$.

• That is, for any string $w$:
  - If $w \in L$, then $M$ accepts $w$.
  - If $w \notin L$, then $M$ does not accept $w$.
    - It might reject $w$, or it might loop on $w$.
• This is a “weak” notion of solving a problem.
• The class $\text{RE}$ consists of all the recognizable languages.
R and RE

• A language $L$ is **decidable** if there is a TM $M$ with the following properties:

$$\forall w \in \Sigma^*. (M \text{ accepts } w \leftrightarrow w \in L).$$

$M$ halts on all inputs.

• That is, for any string $w$:
  • If $w \in L$, then $M$ accepts $w$.
  • If $w \notin L$, then $M$ rejects $w$.

• This is a “strong” notion of solving a problem.

• The class $\mathbf{R}$ consists of all the decidable languages.
The Universal TM

- The *universal Turing machine*, denoted $U_{\text{TM}}$, is a TM with the following behavior: when run on a string $\langle M, w \rangle$, where $M$ is a TM and $w$ is a string, $U_{\text{TM}}$ will
  
  - ... accept $\langle M, w \rangle$ if $M$ accepts $w$,
  - ... reject $\langle M, w \rangle$ if $M$ rejects $w$, and
  - ... loop on $\langle M, w \rangle$ if $M$ loops on $w$.

- $A_{\text{TM}}$ is the language recognized by the universal TM. This is the language

  $$A_{\text{TM}} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}$$
Self-Referential Programs

• Computing devices can compute on their own source code:

  \textbf{Theorem:} It is possible to construct TMs that perform arbitrary computations on their own source code.

• This allows us to write programs that work on their own source code.
What do each of these pieces of code do?
New Stuff!
Part One: Self-Defeating Objects
A self-defeating object is an object whose essential properties ensure it doesn’t exist.
**Question:** Why is there no largest integer?

**Answer:** Because if $n$ is the largest integer, what happens when we look at $n+1$?
**Theorem:** There is no largest integer.

**Proof sketch:** Suppose for the sake of contradiction that there is a largest integer. Call that integer \( n \).

Consider the integer \( n+1 \).

Notice that \( n < n+1 \).

But then \( n \) isn’t the largest integer.

Contradiction! ■-ish

We’re using \( n \) to construct something that undermines \( n \), hence the term “self-defeating.”

**Self-Defeating Objects**
An Important Detail
**Claim:** There is a largest integer.

**Proof:** Assume $x$ is the largest integer.

Notice that $x > x - 1$.

So there’s no contradiction. ■-ish

---

Careful – we’re assuming what we’re trying to prove!

How do we know there’s no contradiction? We just checked one case.
Self-Defeating Objects

• If you can show

\[ x \text{ exists} \rightarrow \bot \]

then you know that \( x \) doesn’t exist. (This is a proof by contradiction.)

• If you can show

\[ x \text{ exists} \rightarrow \top \]

you cannot conclude that \( x \) exists. (This is not a valid proof technique.)
Part Two: The Fortune Teller
The Fortune Teller

- A fortune teller appears who claims they can see into anyone’s future.
- For a nominal fee, the fortune teller will tell you anything you want to know about the future.
One day, a trickster arrives. The trickster thinks the fortune teller is lying and can’t really see the future.

The trickster says the following:

“I have a yes/no question about the future. But before I ask my question, let’s talk payment.

If you answer yes, then I’ll pay you $137.

If you answer no, then I’ll pay you $42.

The fortune teller thinks for a moment, then agrees.

Trickster pays $137 if the fortune teller answers “yes.”

Trickster pays $42 if the fortune teller answers “no.”
The Fortune Teller

• The trickster then asks this question: “Am I going to pay you $42?”

• The fortune teller is trapped!

• Talk to your neighbor – why?

Trickster pays $137 if the fortune teller answers “yes.”

Trickster pays $42 if the fortune teller answers “no.”
The Fortune Teller

- The payment scheme the fortune teller agreed to means
  \[
  \text{Fortune Teller Says Yes} \leftrightarrow \text{Trickster Pays $137}.
  \]
- The trickster’s question to the fortune teller means
  \[
  \text{Fortune Teller Says Yes} \leftrightarrow \text{Trickster Pays $42}.
  \]
- Putting this together, we get
  \[
  \text{Trickster Pays $42} \leftrightarrow \text{Trickster Pays $137}.
  \]
- This is impossible!

Trickster pays $137 if the fortune teller answers “yes.”

Trickster pays $42 if the fortune teller answers “no.”
The Fortune Teller

• The fortune teller is a self-defeating object.
• The trickster’s strategy is to couple the fortune teller’s behavior to what the future holds.
  • The trickster’s behavior is chosen in advance to make the fortune teller’s answer wrong.
• Therefore, the fortune teller can’t answer all questions about all people in the future.

Trickster pays $137 if the fortune teller answers “yes.”

Trickster pays $42 if the fortune teller answers “no.”
Part Three: Why Do Programs Loop?
Thoughts on Loops

- In practice, the programs we write sometimes go into infinite loops.
- In Theoryland, Turing machines are allowed to loop. This happens if they don’t accept and don’t reject.

**Question:** Why are infinite loops possible?

- Or rather: are infinite loops an inherent part of computation, or are they some weird sort of “accident” in how we program computers?
Thoughts on Loops

- **Theorem:** The language $A_{TM}$ is recognizable, but undecidable.
  - There’s a recognizer for $A_{TM}$ (specifically, the universal Turing machine $U_{TM}$).
  - It is impossible to build a decider for this language.
- Stated differently, there’s a program we can write (a universal TM) that has to loop infinitely on some inputs.
- **Goal:** Prove this theorem, and explore its theoretical and philosophical implications.
A_\text{TM} Revisited

- As a refresher, the language A_\text{TM} is

  \[ A_\text{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \} \].

- The universal TM U_\text{TM} has the following behavior when given as input a TM M and a string w:
  - If M accepts w, then U_\text{TM} accepts \langle M, w \rangle.
  - If M rejects w, then U_\text{TM} rejects \langle M, w \rangle.
  - If M loops on w, then U_\text{TM} loops on \langle M, w \rangle.

- U_\text{TM} is a recognizer for A_\text{TM}, but because of that last case it’s not a decider for A_\text{TM}.
A\text{TM} Revisited

- As a refresher, the language $A_{\text{TM}}$ is
  
  $A_{\text{TM}} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}$. 

- Given a TM $M$ and a string $w$, a decider $D$ for $A_{\text{TM}}$ would need to have this behavior:
  - If $M$ accepts $w$, then $D$ accepts $\langle M, w \rangle$.
  - If $M$ rejects $w$, then $D$ rejects $\langle M, w \rangle$.
  - If $M$ loops on $w$, then $D$ rejects $\langle M, w \rangle$.

- This is basically the same set of requirements as $U_{\text{TM}}$, except for what happens if $M$ loops on $w$.

- Our goal is to prove that there is no way to build a program that meets these requirements.
A\textsuperscript{TM} Revisited

• We can envision a decider for A\textsuperscript{TM} as a function

\begin{verbatim}
bool willAccept(string fn, string input)
\end{verbatim}

that takes as input the source code of a function (fn) and a string representing an input to that function (input).

• It then does the following:
  • If fn(input) returns true, willAccept(fn, input) returns true.
  • If fn(input) returns false, willAccept(fn, input) returns false.
  • If fn(input) loops, then willAccept(fn, input) returns false.

• We’re going to show it’s impossible to write a function that actually does this. But for now, let’s just explore what such a decider would do.
For each of these instances, what does `willAccept(function, input)` return?

```c
function = "bool f(string input) {
    if (input == "") return false;
    return input[0] == 'a';
}";
input = "abbababba";
willAccept(function, input) = ?
```

```c
function = "bool g(string input) {
    while (true) {
        input += input;
    }
}";
input = "yay! ";
willAccept(function, input) = ?
```

```c
function = "bool h(string input) {
    for (char c: input) {
        if (c != input[0]) return true;
    }
    return false;
}";
input = "aaaaaa";
willAccept(function, input) = ?
```

```c
function = "bool j(string input) {
    int n = input.length();
    while (n > 1) {
        if (n % 2 == 0) n /= 2;
        else n = 3*n + 1;
    }
    return true;
}";
input = /* 10^{137} a's */;
willAccept(function, input) = ?
```
Deciding $A_{\text{TM}}$

- Earlier this quarter you explored sums of four squares. Now, let’s talk about sums of three cubes.
- Are there integers $x$, $y$, and $z$ where...
  - $x^3 + y^3 + z^3 = 10$?
  - $x^3 + y^3 + z^3 = 11$?
  - $x^3 + y^3 + z^3 = 12$?
  - $x^3 + y^3 + z^3 = 13$?
Deciding $A_{TM}$

- Surprising fact: until 2019, no one knew whether there were integers $x$, $y$, and $z$ where
  \[ x^3 + y^3 + z^3 = 33. \]
- A heavily optimized computer search found this answer:
  \[
  x = 8,866,128,975,287,528 \\
  y = -8,778,405,442,862,239 \\
  z = -2,736,111,468,807,040
  \]
- As of November 2021, no one knows whether there are integers $x$, $y$, and $z$ where
  \[ x^3 + y^3 + z^3 = 114. \]
Deciding $A_{TM}$

- Consider the language
  \[ L = \{ a^n \mid \exists x \in \mathbb{Z}. \exists y \in \mathbb{Z}. \exists z \in \mathbb{Z}. x^3 + y^3 + z^3 = n \} \]
- Here’s code for a recognizer to see whether such a triple exists:

```cpp
bool hasTriple(int n) {
    for (int max = 0; ; max++)
        for (int x = -max; x <= max; x++)
            for (int y = -max; y <= max; y++)
                for (int z = -max; z <= max; z++)
                    if (x*x*x + y*y*y + z*z*z == n)
                        return true;
}
```
- Imagine calling `willAccept(/* hasTriple code */, 114)`.
  - If such a triple exists, `willAccept` returns `true`.
  - If no such triple exists, `willAccept` returns `false`.
- **Key Intuition:** However `willAccept` is implemented, it has to be clever enough to resolve open problems in mathematics!
Why is $A^\text{TM}$ Hard?

• **Intuition:** A decider for $A^\text{TM}$ would be able to...
  
  • ... determine whether the hailstone sequence terminates for any input. (Write a recognizer that runs the hailstone sequence, then feed it into the decider for $A^\text{TM}$.)
  
  • ... see if any number is the sum of three cubes. (Write a recognizer that tries all infinitely many triples of integers, then feed it into the decider for $A^\text{TM}$.)
  
  • ... and much, much more.

• In other words, this seemingly simple problem of “is this program going to terminate?” accidentally scoops up a bunch of other seemingly harder problems.
Time-Out for Announcements!
Info session tomorrow (Thursday) at 2:30PM. RSVP using this link.
On Rigor and Formalism in Math

- Terry Tao, considered by many to be the greatest living mathematician, has an essay about learning mathematics.
- It explains why formal proofs and rigorous arguments are an important part of learning math – and why it can be a bit tricky at times.
- You can read it online here.
Your Questions
“What should my roommate and I name our plants?”
Back to CS103!
Part Four: Putting It All Together
To Recap

• We’re assuming that, somehow, someone wrote a function
  
  ```
  bool willAccept(string function, string input);
  ```
  that takes the code of a function and an input to that function, then

  • returns true if function(input) returns true, and
  • returns false if function(input) doesn’t return true.

• **Goal:** Show that this decider is “self-defeating;” its power is so great that it undermines itself.

• **Idea:** Convert the fortune teller story into a program.
Trickster pays $137 if the fortune teller answers "yes."

Trickster pays $42 if the fortune teller answers "no."

Am I going to pay you $42?
```c
bool willAccept(string function, string input) {
    // Returns true if function(input) returns true. Returns false otherwise.
}

bool trickster(string input) {
    string me = /* source code of trickster */;
    return !willAccept(me, input);
}
```

If `willAccept` says `trickster` will return true, then `trickster` returns false.

If `willAccept` says `trickster` will not return true, then `trickster` returns true.
bool willAccept(string function, string input) {
    // Returns true if function(input) returns true. Returns false otherwise.
}

bool trickster(string input) {
    string me = /* source code of trickster */;
    return !willAccept(me, input);
}
Theorem: There is no largest integer.

Proof sketch: Suppose for the sake of contradiction that there is a largest integer. Call that integer $n$.

Consider the integer $n+1$.

Notice that $n < n+1$.

But then $n$ isn’t the largest integer.

Contradiction! ■-ish
**Theorem:** \( A_{TM} \notin R. \)

**Proof:** By contradiction; assume that \( A_{TM} \in R. \) Then there is a decider \( D \) for \( A_{TM} \). We can represent \( D \) as a function

\[
\text{bool willAccept(string function, string w);}\
\]

that takes in the source code of a function \( \text{function} \) and a string \( w \), then returns true if \( \text{function}(w) \) returns true and returns false otherwise.

Given this, consider this function \( \text{trickster} \):

\[
\begin{array}{l}
\text{bool trickster(string input) } \\
\quad \text{string me } = /* \text{source code of trickster} */; \\
\quad \text{return !willAccept(me, input);} \\
\end{array}
\]

Choose a string \( w \). We consider two cases:

**Case 1:** \( \text{willAccept(me, input)} \) returns true. Since \( \text{willAccept} \) decides \( A_{TM} \), this means \( \text{trickster}(w) \) returns true. However, given how \( \text{trickster} \) is written, in this case \( \text{trickster}(w) \) returns false.

**Case 2:** \( \text{willAccept(me, input)} \) returns false. Since \( \text{willAccept} \) decides \( A_{TM} \), this means \( \text{trickster}(w) \) doesn’t return true. However, given how \( \text{trickster} \) is written, in this case \( \text{trickster}(w) \) returns true.

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

CFLs

R

RE

A_{TM}

All Languages
What Does This Mean?

- In one fell swoop, we've proven that
  - $A_{\text{TM}}$ is **undecidable**; there is no general algorithm that can determine whether a TM will accept a string.
  - $R \neq \text{RE}$, because $A_{\text{TM}} \notin R$ but $A_{\text{TM}} \in \text{RE}$.
- What do these three statements really mean? As in, why should you care?
What exactly does it mean for $A_{TM} \notin R$ 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 most questions about what a program will do.
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 a TM $M$ and a string $w$, one of these two statements is true:

- $M$ accepts $w$
- $M$ does not accept $w$

But since $A_{TM}$ is undecidable, there is no algorithm that can always determine which of these statements is true!
Because $R \neq RE$, there is a difference between decidability and recognizability:

**In some sense, it is fundamentally harder to solve a problem than it is to check an answer.**

There are problems where, when the answer is “yes,” you can confirm it (run a recognizer), but where if you don’t have the answer, you can’t come up with it in a mechanical way (build a decider).
Next Time

- **Why All This Matters**
  - Important, practical, undecidable problems.
- **Intuiting RE**
  - What exactly is the class RE all about?
- **Verifiers**
  - A totally different perspective on problem solving.
- **Beyond RE**
  - Finding an impossible problem using very familiar techniques.