The Limitations of Computation

11 May 2021
Where are we?
We’ve introduced our final model of a computer, the Turing machine.

We designed Turing machines at the level of individual states and transitions.

In your reading (and on Assignment 9), you’ve seen that you can also describe a Turing machine more abstractly, like writing pseudo-code.
The *Church–Turing thesis* states:

Every effective method of computation is either equivalent to or weaker than a Turing machine.
A language $L$ is **Turing-recognizable** if there is a TM $M$ such that

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

This is a *weak* notion of solving a problem:

- If $w \in L$, then $M$ accepts $w$.
- If $w \notin L$, then $M$ does not accept $w$.
  - It might reject or it might loop forever.

The class $\textbf{RE}$ consists of all Turing-recognizable languages.
A language $L$ is **Turing-decidable** if there is a TM $M$ such that

$$\forall w \in \Sigma^*. \ (M \text{ accepts } w \iff w \in L) \land \ (M \text{ halts on all inputs}).$$

This is a **strong** notion of solving a problem:

- If $w \in L$, then $M$ accepts $w$.
- If $w \notin L$, then $M$ rejects $w$.

The class $\mathbf{R}$ consists of all Turing-decidable languages.
If \( \text{Obj} \) is an object, then \( \langle \text{Obj} \rangle \) denotes some string representing \( \text{Obj} \), similar to how it might be stored on disk or in memory on a real computer.

We can encode multiple objects as a single string. For example, if \( M \) is a TM and \( w \) is a string, then \( \langle M, w \rangle \) is a string representing the pair of \( M \) and \( w \).
There is a TM named $U$ that is a universal Turing machine.

$U$ takes as input a pair $\langle M, w \rangle$, where $M$ is a TM and $w$ is a string. $U$ does to $\langle M, w \rangle$ whatever $M$ does to $w$. 
The language of $U$ is called $A_{TM}$:

$$A_{TM} = L(U)$$
$$= \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}$$

$A_{TM}$ is the *acceptance language for Turing machines*. Because there is a Turing machine, $U$, that recognizes $A_{TM}$, we know $A_{TM} \in \text{RE}$. 
A *self-defeating object* is an object whose essential properties ensure it doesn’t exist.
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!

We’re using $n$ to construct something that undermines $n$, hence the term “self-defeating”.
Teaser: Certain Turing machines can’t exist, as they’d be self-defeating objects.
Undecidability
Suppose $M$ is a recognizer for some language.

We have a string $w$, and we want to know if $w \in L(M)$.

How could we check this?
If you want to know whether this is true...

$w \in L(M)$ if and only if

$M$ accepts $w$

...you can try to determine whether this is true.
Option 1: Run $M$ on $w$.

What could happen?

- $M$ could accept $w$. Great! We know $w \in L(M)$.
- $M$ could reject $w$. Great! We know $w \notin L(M)$.
- $M$ could loop on $w$. In this case we've learned nothing. 😞

So, this won't always tell us whether $w \in L(M)$.

We'll need a different strategy.
If you want to know whether this is true...

\[ w \in L(M) \]

if and only if

\[ M \text{ accepts } w \]

if and only if

\[ \langle M, w \rangle \in A_{TM} \]

...you can try to determine whether this is true.
Option 2: Use the universal Turing machine, which is a recognizer for $A_{TM}$!

Specifically, run $U$ on $\langle M, w \rangle$.

What could happen?

- $U$ could accept $\langle M, w \rangle$. Great! Then $w \in L(M)$.
- $U$ could reject $\langle M, w \rangle$. Great! Then $w \notin L(M)$.
- $U$ could loop on $\langle M, w \rangle$. In this case we’ve learned nothing. 😞

This won’t always tell us whether $w \in L(M)$ either. We’ll need a different strategy.
**Option 2**: Use the universal Turing machine, which is a recognizer for $A_{\text{TM}}$!

Specifically, run $U$ on $\langle M, w \rangle$.

What could happen?

- $U$ could accept $\langle M, w \rangle$. Great! Then $w \in L(M)$.
- $U$ could reject $\langle M, w \rangle$. Great! Then $w \notin L(M)$.
- $U$ could loop on $\langle M, w \rangle$. In this case we’ve learned nothing. 😞

This won’t always tell us whether $w \in L(M)$ either. We’ll need a different strategy.

*What if we had a decider for $A_{\text{TM}}$ instead of a recognizer?*
Option 3: Build a **decider** for $A_{TM}$ rather than just a recognizer.

What could happen?

The decider could accept $\langle M, w \rangle$. Great! Then $w \in L(M)$.

The decider could reject $\langle M, w \rangle$. Great! Then $w \not\in L(M)$.

How do we build this decider?
CLAIM. A decider for $A_{TM}$ is a self-defeating object. It therefore doesn’t exist.
Assume $A_{TM}$ is decidable, i.e., $A_{TM} \in R$.

This means there’s a decider for $A_{TM}$. Let’s call it $D$:

![Diagram](image)

What can we do with $D$?
We've seen the idea that Turing machines can run other Turing machines as subroutines. This means we can write programs that use $D$ as a subroutine.

Since Turing machines are like programs, we can imagine that $D$ is a helper method like this:

```python
def will_accept(program, input):
    ...some implementation...
```
We've seen the idea that Turing machines can run other Turing machines as subroutines. This means we can write programs that use $D$ as a subroutine.

Since Turing machines are like programs, we can imagine that $D$ is a helper method like this:

```python
def will_accept(program, input):
    ...some implementation...
```
We've seen the idea that Turing machines can run other Turing machines as subroutines. This means we can write programs that use $D$ as a subroutine.

Since Turing machines are like programs, we can imagine that $D$ is a helper method like this:

```
def will_accept(program, input):
    ...some implementation...
```
What can we do with the subroutine `will_accept`?

Ultimately, we’re trying to get a contradiction.

Specifically, we’re going to build a program that has some really broken behavior:

*It will accept its input if and only if it doesn’t accept its input!*
def will_accept(program, input):
    ...some implementation...

def main(my_input):
    uh_oh = my_source()
    if will_accept(uh_oh, my_input):
        return False
    else:
        return True
def will_accept(program, input):
    ...some implementation...

def main(my_input):
    uh_oh = my_source()
    if will_accept(uh_oh, my_input):
        return False
    else:
        return True

Try running this on any input.
def will_accept(program, input):
    ...some implementation...

def main(my_input):
    uh_oh = my_source()

    if will_accept(uh_oh, my_input):
        return False
    else:
        return True
What happens if the program accepts its input?

uh-oh.py:

```python
def will_accept(program, input):
    ...some implementation...

def main(my_input):
    uh_oh = my_source()
    if will_accept(uh_oh, my_input):
        return False
    else:
        return True
```

It rejects the input!
What happens if the program rejects its input?

**uh-oh.py:**

```python
def will_accept(program, input):
    ...
some implementation ...

def main(my_input):
    uh_oh = my_source()

    if will_accept(uh_oh, my_input):
        return False
    else:
        return True
```
What happens if the program **rejects** its input?

**uh-oh.py:**

```python
def will_accept(program, input):
    ...some implementation...

def main(my_input):
    uh_oh = my_source()
    if will_accept(uh_oh, my_input):
        return False
    else:
        return True
```

*It accepts the input!*
def will_accept(program, input):
    ...some implementation...

def main(my_input):
    uh_oh = my_source()

    if will_accept(uh_oh, my_input):
        return False
    else:
        return True

uh-oh.py:
The self-defeating object

Using that object against itself
def will_accept(program, input):
    ...
some implementation...

def main(my_input):
    uh_oh = my_source()
    if will_accept(uh_oh, my_input):
        return False
    else:
        return True

*uh-oh.py:*

"The largest integer, n"

"The number n + 1."
"I cannot – yet I must! How do you calculate that? At what point on the graph do 'must' and 'cannot' meet?"

– Ro-Man in *Robot Monster*, 1953
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!

```
<uh-oh.py>

def will_accept(program, input):
    ...
    ...some implementation...

def main(my_input):
    uh_oh = my_source()
    if will_accept(uh_oh, my_input):
        return False
    else:
        return True
```
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!

uh-oh.py:

```python
def will_accept(program, input):
    ...some implementation...

def main(my_input):
    uh_oh = my_source()
    if will_accept(uh_oh, my_input):
        return False
    else:
        return True
```

Assume there exists this object $x$ which has these properties that are too powerful to actually work.
**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!

---

**uh-oh.py:**

```python
def will_accept(program, input):
    ...some implementation...

def main(my_input):
    uh_oh = my_source()
    if will_accept(uh_oh, my_input):
        return False
    else:
        return True
```

*Use the purported properties of $x$ against itself to create a contradiction.*
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!


```
def will_accept(program, input):
    ...
some implementation...

def main(my_input):
    uh_oh = my_source()
    if will_accept(uh_oh, my_input):
        return False
    else:
        return True
```

Thus the object $x$ cannot exist!
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 procedure `will_accept` that takes as input the source code of a program and an input, and returns true if the program accepts the input and false otherwise.

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

```python
if will_accept(my_source, my_input):
    return False
else:
    return True
```

Choose any string $w$ and trace through the execution of program $P$ on input $w$: If `will_accept(my_source, my_input)` returns true, this means that $P$ must accept its input $w$, but instead it rejects it. Otherwise, if `will_accept(my_source, my_input)` returns false, this means that $P$ must not accept its input $w$, but instead it accepts it.

In both cases, we reach a contradiction, so our assumption must have been wrong. Therefore, $A_{TM} \not\in R$. $\blacksquare$
THEOREM. $A_{\text{TM}} \notin \mathbb{R}$.

PROOF. By contradiction; assume that $A_{\text{TM}} \in \mathbb{R}$. Then there is some decider $D$ for $A_{\text{TM}}$. If this machine is given a TM/string pair, it will determine whether the TM accepts the string and report back the answer.

Given this, we could construct the following TM:

$$M = \text{“On input } w:\text{ “}
\begin{align*}
1. & \text{ Have } M \text{ obtain its own description } \langle M \rangle. \\
2. & \text{ Run } D \text{ on } \langle M, w \rangle \text{ and see what it says.} \\
3. & \text{ If } D \text{ says that } M \text{ will accept } w, \text{ reject.} \\
4. & \text{ If } D \text{ says that } M \text{ will not accept } w, \text{ accept.”}
\end{align*}$$

Choose any string $w$ and trace through the execution of the machine, focusing on the answer given back by the machine $D$. If $D$ says that $M$ will accept $w$, notice that $M$ then proceeds to reject $w$, contradicting what $D$ says. Otherwise, if $D$ says that $M$ will not accept $w$, notice that $M$ then proceeds to accept $w$, contradicting what $D$ says.

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

Same proof, without using the equivalence of TMs and code
Regular languages

Context-free languages

All languages
What does this mean?

In one fell swoop, we’ve proven that

A decider for $A_{TM}$ is a self-defeating object.

$A_{TM}$ is undecidable, i.e., there is no general algorithm that can determine whether a Turing machine will accept a string.

$R \neq RE$, because $A_{TM} \notin R$ but $A_{TM} \in RE$

What do these three statements really mean? Why should you care?
1 Self-defeating objects

The fact that a decider for $A_{TM}$ is a self-defeating object is analogous to the philosophical question,

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

If we have a decider for $A_{TM}$, we could use it to build a TM that determines what it’s supposed to do and then chooses to do the opposite!
The proof we’ve done says that there is no algorithm that can determine whether a program will accept an input.

Our proof just assumed that there was some decider for $A_{TM}$ and didn’t assume anything about how the decider worked.

No matter how you try to implement a decider for $A_{TM}$, you can never succeed!
What exactly does it mean for $A_{TM}$ to be undecidable?

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

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

Given a Turing machine $M$ 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’s no algorithm that can always determine *which* of these statements is true!
This tell us *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 (build 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).
The Halting Problem
The most famous undecidable problem is the **Halting Problem**, which asks:

> *Given a Turing machine $M$ and a string $w$, will $M$ halt when run on $w$?*

or, as a language,

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

This is an **RE** language. *We’ll see why later.*

How do we know that it’s undecidable?
CLAIM: A decider for $HALT_{TM}$ is a self-defeating object. Therefore it doesn’t exist.
A decider for $\text{HALT}_{\text{TM}}$

Suppose we managed to build a decider for $\text{HALT}_{\text{TM}} = \{\langle M, w \rangle \mid M \text{ is a TM that halts on } w \}$:

We could represent this in software as a procedure `will_halt(program, input)`
def will_halt(program, input):
    ...some implementation...

def main(my_input):
    halt = open("halt.py").read()
    if will_halt(halt, my_input):
        while True:  # Infinite loop
            pass
    else:
        return True
def will_halt(program, input):
    ...some implementation...

def main(my_input):
    halt = open("halt.py").read()
    if will_halt(halt, my_input):
        while True:  # Infinite loop
            pass
    else:
        return True

halt.py:

Try running this on any input.
What happens if the program halts on its input?

**halt.py:**

def will_halt(program, input):
  ...some implementation...

def main(my_input):
  halt = open("halt.py").read()

  if will_halt(halt, my_input):
    while True:
      # Infinite loop
      pass
  else:
    return True
What happens if the program *halts on its input*?

**halt.py:**

```python
def will_halt(program, input):
    ...some implementation...

def main(my_input):
    halt = open("halt.py").read()

    if will_halt(halt, my_input):
        while True:  # Infinite loop
            pass
    else:
        return True
```

*It loops on the input!*
```python
# halt.py:
def will_halt(program, input):
    # ...some implementation...

def main(my_input):
    halt = open("halt.py").read()

    if will_halt(halt, my_input):
        while True:  # Infinite loop
            pass
    else:
        return True
```

What happens if the program loops on its input?
def will_halt(program, input):
    ...some implementation...

def main(my_input):
    halt = open("halt.py").read()
    if will_halt(halt, my_input):
        while True:  # Infinite loop
            pass
    else:
        return True

It halts on the input!
def will_halt (program, input):
...some implementation...

def main (my_input):
    halt = open ("halt.py").read ()
    if will_halt (halt, my_input):
        while True:  # Infinite loop
            pass
    else:
        return True
THEOREM. $HALT_{TM} \not\in R$.

PROOF. By contradiction; assume that $HALT_{TM} \in R$. Then there is some decider $D$ for $HALT_{TM}$, which we can represent in software as a procedure `will_halt` that takes as input the source code of a program and an input, and returns true if the program halts on the input and false otherwise.

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

```python
if will_halt(my_source, my_input):
    while True: pass
else:
    return True
```

Choose any string $w$ and trace through the execution of program $P$ on input $w$: If `will_halt(my_source, my_input)` returns true, this means that $P$ must halt on its input $w$, but instead it loops on it. Otherwise, if `will_halt(my_source, my_input)` returns false, this means that $P$ must not halt on its input $w$, but instead it halts and accepts it.

In both cases, we reach a contradiction, so our assumption must have been wrong. Therefore, $HALT_{TM} \not\in R$. ■
**THEOREM.** \( \text{HALT}_{TM} \not\in \mathbb{R} \).

**PROOF.** By contradiction; assume that \( \text{HALT}_{TM} \in \mathbb{R} \). Then there is some decider \( D \) for \( \text{HALT}_{TM} \). If this machine is given any TM/string pair, it will then determine whether the TM halts on the string and report back the answer.

Given this, we could construct the following TM:

\[ M = \text{“On input } w: \]

\[ 1. \text{ Have } M \text{ obtain its own description } \langle M \rangle. \]
\[ 2. \text{ Run } D \text{ on } \langle M, w \rangle \text{ and see what it says.} \]
\[ 3. \text{ If } D \text{ says that } M \text{ halts on } w, \text{ go into an infinite loop.} \]
\[ 4. \text{ If } D \text{ says that } M \text{ loops on } w, \text{ accept.”} \]

Choose any string \( w \) and trace through the execution of the machine, focusing on the answer given back by machine \( D \). If \( D \) says that \( M \) will halt on \( w \), notice that \( M \) then proceeds to loop on \( w \), contradicting what \( D \) says. Otherwise, if \( D \) says that \( M \) will loop on \( w \), notice that \( M \) then proceeds to accept \( w \), so \( M \) halts on \( w \), contradicting what \( D \) says.

In both cases, we reach a contradiction, so our assumption must have been wrong. Therefore, \( \text{HALT}_{TM} \not\in \mathbb{R} \). \( \blacksquare \)
CLAIM. $\text{HALT}_\text{TM} \in \text{RE}$.

IDEA. If you were certain that a Turing machine $M$ halted on a string $w$, could you convince me of that?

Yes – just run $M$ on $w$ and see what happens!
Moral: This isn’t necessarily Microsoft’s fault.
Ramifications, or: So what?
These problems might seem really convoluted and not very exciting, so who cares if we can’t solve them?

The same line of reasoning we used to show that it’s undecidable whether a Turing machine will accept its input ($A_{TM}$) or whether it will halt on its input ($HALT$) can be used to show many important, practical problems are impossible to solve.
Secure voting

Suppose you want to make a voting machine for use in an imaginary election between two parties.

Let $\Sigma = \{r, d\}$, for no particular reason.

A string consists of a series of votes for the candidates.

For example, $rrddrd$ means “two people voted for $r$, then three people voted for $d$, then one more person voted for $r$, then one more person voted for $d$.”
A voting machine is a program that takes as input a string of \( r \)s and \( d \)s and then reports whether person \( r \) won the election.

It would be equivalent to ask “did \( d \) win”?

For simplicity, this model assumes centralized voting, e.g., done online.

**Question**: Given a Turing machine that someone claims is a secure voting machine, could we automatically check whether it’s really a secure voting machine?
WATCH WHAT HAPPENS WHEN I VOTE FOR BOBBY NEWPORT.

BUT WATCH WHAT HAPPENS WHEN YOU VOTE FOR ME.

GOOD CHOICE!
Enjoy a voucher for a complimentary Sweetums candy bar.

ARE YOU SURE?
Take a second and think it over...

[ buzzer sounds ]
def main(w):
    r_votes = count_rs(w)
    d_votes = count_ds(w)
    if r_votes > d_votes:
        return True  # R won
    else:
        return False  # R lost

A (simple) secure voting machine

def main(w):
    if w[0] == "r" :
        return True  # R won
    else:
        return False  # R lost

A (simple) insecure voting machine
An (evil) insecure voting machine

def main(w):
    r_votes = count_rs(w)
    d_votes = count_ds(w)
    if r_votes == d_votes:
        # Tied; R lost.
        return False
    if r_votes < d_votes:
        # D won; R lost.
        return False
    else:
        # R won
        return True
def main(w):
    r_votes = count_rs(w)
    d_votes = count_ds(w)
    if r_votes == d_votes:
        # Tied; R lost.
        return False
    if r_votes < d_votes:
        # D won; R lost.
        return False
    else:
        # R won
        return True

This is assignment; == is equality testing. Python won’t actually let you do this, but other languages like C or Java will!

An (evil) insecure voting machine
def main(w):
    n = len(w)
    while n > 1:
        if n % 2 == 0:
            n /= 2
        else:
            n = 3 * n + 1
    r_votes = count_rs(w)
    d_votes = count_ds(w)
    if r_votes > d_votes:
        return True  # R won
    else:
        return False  # R lost
def main(w):
    n = len(w)
    while n > 1:
        if n % 2 == 0:
            n /= 2
        else:
            n = 3 * n + 1
    r_votes = count_rs(w)
    d_votes = count_ds(w)
    if r_votes > d_votes:
        return True  # R won
    else:
        return False  # R lost

No one knows!
Secure voting

The **secure voting problem** is the following:

Given a TM $M$, is the language of $M$

$\{w \in \Sigma^* \mid w \text{ has more } rs \text{ than } ds\}$?

**Claim**: This problem is not decidable; there is no algorithm that can check an arbitrary TM to verify that it’s a secure voting machine!

A program that decides whether arbitrary input programs are secure voting machines is self-defeating. It therefore doesn’t exist.
A decider for secure voting

Suppose that, somehow, we managed to build a decider for the secure voting problem:

We could represent this in software as a procedure

\[ \text{is\_secure\_vm}(\text{program}) \]
def is_secure_vm(program):
    ...some implementation...

def main(w):
    vm = my_source()
    answer = count_rs(w) > count_ds(w)
    if is_secure_vm(vm):
        return not answer
    else:
        return answer
What happens if this program is a secure voting machine?

vm.py:

```python
def is_secure_vm(program):
    ...some implementation...

def main(w):
    vm = my_source()
    answer = count_rs(w) > count_ds(w)
    if is_secure_vm(vm):
        return not answer
    else:
        return answer
```

Then it’s not a secure voting machine!
What happens if this program is not a secure voting machine?

`vm.py`:

```python
def is_secure_vm(program):
    ...some implementation...

def main(w):
    vm = my_source()
    answer = count_rs(w) > count_ds(w)
    if is_secure_vm(vm):
        return not answer
    else:
        return answer
```

Then it’s a secure voting machine!
Interpreting this result

This tells us that there is *no* general algorithm that we can follow to determine whether a program is a secure voting machine.

In other words, any general algorithm to check voting machines will always be wrong on *at least one input*.

The previous example might seem contrived, but it’s not. This is a problem we really would like to be able to solve – but it’s provably impossible!
What can we do?

Design algorithms that work in *many* but not *all* cases.

  This is often done in practice!

Fall back on human verification of voting machines.

  We do this too!

Carry a healthy degree of skepticism about electronic voting machines.

  We were born skeptical.
Wrapping up undecidability
We’ve seen a general pattern in proving undecidability (i.e., non-membership in $\mathbb{R}$):

Assume the language in question – usually a language about TMs – is decidable.

Build a machine that decides whether it has the property, then chooses to do something contrary to that property.

Conclude that something is terribly wrong, meaning that the original language wasn't decidable.
Two intuitions

The *avoid your fate* intuition:

Construct a machine so that it learns its fate (i.e., *decides* what to do next), then actively chooses to do the opposite.

The *impossible bind* intuition:

Imagine the TM in conversation with the decider that can allegedly predict what happens next.

Have the TM tell the decider that it’s going to do the opposite of whatever the decider says.

The decider’s in an impossible bind – anything it says must be wrong!
Acknowledgments

This lecture incorporates material from:

Nancy Ide, Vassar College
Keith Schwarz, Stanford University
Michael Sipser, *Introduction to the Theory of Computation*