Undecidability

Lecture 23

April 22, 2025

Part I

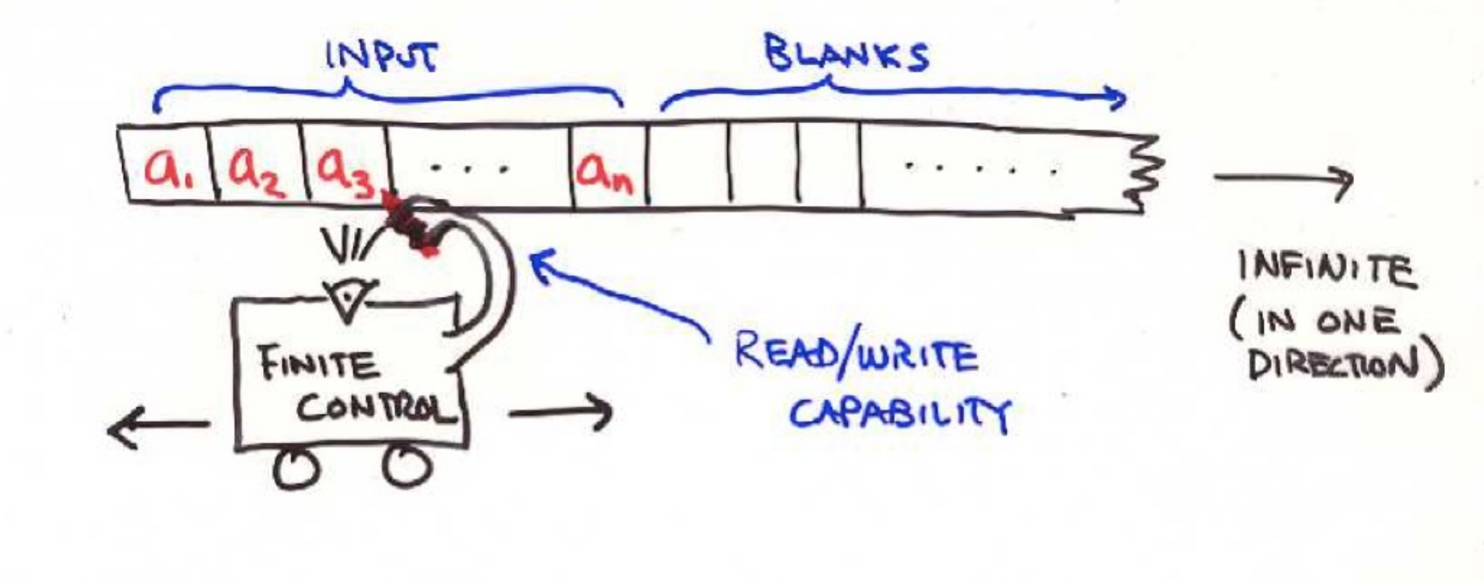
Formal Definitions

What is Computation?

Recall: our most general form of computation was that performed by Turing Machines.

What is Computation?

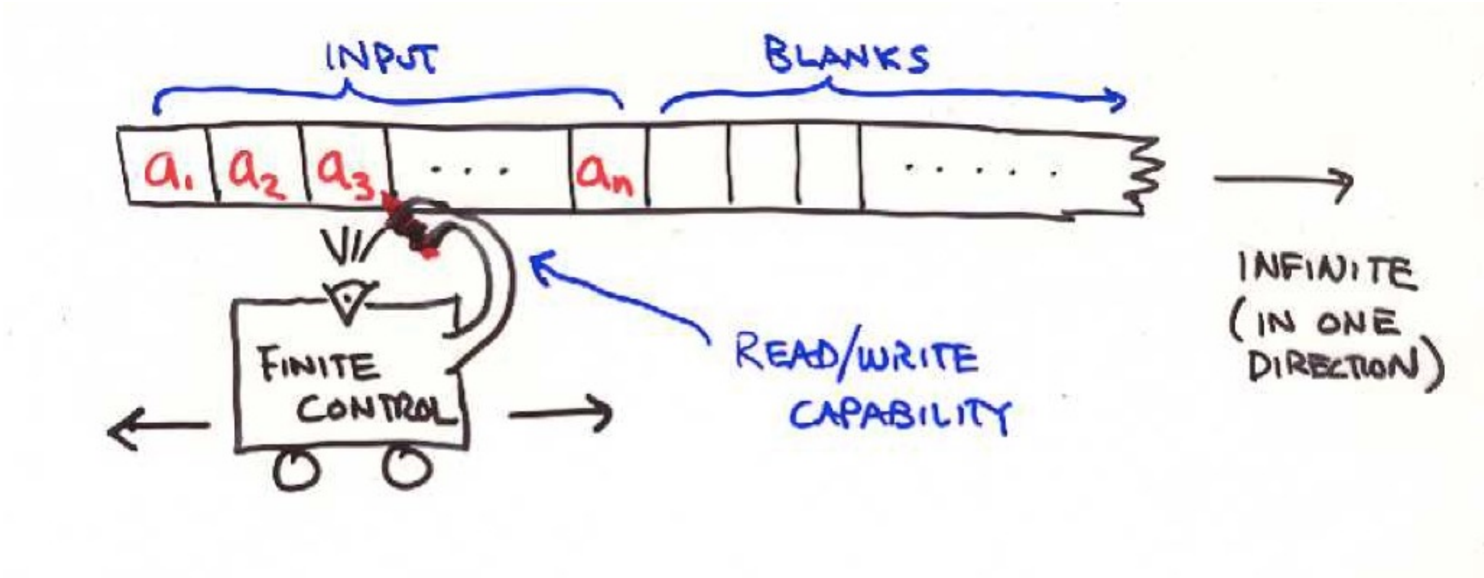
Recall: our most general form of computation was that performed by Turing Machines.



Each step: read the current character, write a new character, move one space, update (finite) state of head.

What is Computation?

Recall: our most general form of computation was that performed by Turing Machines.



Each step: read the current character, write a new character, move one space, update (finite) state of head.

Extremely tedious to work with TMs, so we'll just extract the properties we need and work with those.

TM Properties: Behavior

On input w, a TM either accepts w, rejects w, or runs forever.

TM Properties: Behavior

On input w, a TM either accepts w, rejects w, or runs forever.

For a TM M, we define the following languages:

- $L(M) = \text{Accept}(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$
- $\text{Reject}(M) = \{w \in \Sigma^* \mid M \text{ rejects } w\}$
- $\text{Diverge}(M) = \{w \in \Sigma^* \mid M \text{ runs forever on } w\}$
- $\text{Halt}(M) = \{w \in \Sigma^* \mid M \text{ halts on } w\}$

TM Properties: Behavior

On input w, a TM either accepts w, rejects w, or runs forever.

For a TM M, we define the following languages:

- $L(M) = \text{Accept}(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$
- $\text{Reject}(M) = \{w \in \Sigma^* \mid M \text{ rejects } w\}$
- $\text{Diverge}(M) = \{w \in \Sigma^* \mid M \text{ runs forever on } w\}$
- $\text{Halt}(M) = \{w \in \Sigma^* \mid M \text{ halts on } w\}$

In order for a language L to be computable (aka decidable aka recursive), we require that there is a TM M such that $\text{Accept}(M) = L$ *and* $\text{Diverge}(M) = \emptyset$.

TM Properties: Behavior

On input w, a TM either accepts w, rejects w, or runs forever.

For a TM M, we define the following languages:

- $L(M) = \text{Accept}(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$
- $\text{Reject}(M) = \{w \in \Sigma^* \mid M \text{ rejects } w\}$
- $\text{Diverge}(M) = \{w \in \Sigma^* \mid M \text{ runs forever on } w\}$
- $\text{Halt}(M) = \{w \in \Sigma^* \mid M \text{ halts on } w\}$

In order for a language L to be computable (aka decidable aka recursive), we require that there is a TM M such that $\text{Accept}(M) = L$ *and* $\text{Diverge}(M) = \emptyset$.

- If we know that $\text{Accept}(M) = L$, we call L "acceptable" or "recursively enumerable".

TM Properties: Encoding

Every TM can be encoded as a finite-length bit string.

- Each part of $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$ is finite, so use whatever (reasonable) encoding you want!
- We will often denote the *encoding* of M by $\langle M \rangle$.

TM Properties: Encoding

Every TM can be encoded as a finite-length bit string.

- Each part of $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$ is finite, so use whatever (reasonable) encoding you want!
- We will often denote the *encoding* of M by $\langle M \rangle$.

Properties we need from our encoding:

- **Unique**: different TMs have different encodings. (Bit strings that don't correspond to any TM can be mapped to a "dummy" machine.)

TM Properties: Encoding

Every TM can be encoded as a finite-length bit string.

- Each part of $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$ is finite, so use whatever (reasonable) encoding you want!
- We will often denote the *encoding* of M by $\langle M \rangle$.

Properties we need from our encoding:

- **Unique**: different TMs have different encodings. (Bit strings that don't correspond to any TM can be mapped to a "dummy" machine.)
- **Modifiable**: we can (algorithmically) modify an encoding to change the behavior of its TM. (For example, switch accepting with rejecting, add pre- or post-processing, etc.)

TM Properties: Encoding

Every TM can be encoded as a finite-length bit string.

- Each part of $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$ is finite, so use whatever (reasonable) encoding you want!
- We will often denote the *encoding* of M by $\langle M \rangle$.

Properties we need from our encoding:

- **Unique**: different TMs have different encodings. (Bit strings that don't correspond to any TM can be mapped to a "dummy" machine.)
- **Modifiable**: we can (algorithmically) modify an encoding to change the behavior of its TM. (For example, switch accepting with rejecting, add pre- or post-processing, etc.)
- **Executable**: There is a universal TM U that on input $\langle M, w \rangle$ simulates $M(w)$. (So $L(U) = \{\langle M, w \rangle \mid M \text{ accepts } w\}$.)

Properties of Computability

Lemma

Let L and L' be decidable languages. Then $L \cup L'$, $L \cap L'$, $L - L'$, and $L' - L$ are decidable.

Alg: run the alg for each of L and L', accept accordingly

Properties of Computability

Lemma

Let L and L' be decidable languages. Then $L \cup L'$, $L \cap L'$, $L - L'$, and $L' - L$ are decidable.

Lemma

Let L and L' be acceptable languages. Then $L \cup L'$ and $L \cap L'$ are acceptable. *($L - L'$ and $L' - L$ may or may not be!)*

Alg: run algs for L and L' "in parallel"

Properties of Computability

Lemma

Let L and L' be decidable languages. Then $L \cup L'$, $L \cap L'$, $L - L'$, and $L' - L$ are decidable.

Lemma

Let L and L' be acceptable languages. Then $L \cup L'$ and $L \cap L'$ are acceptable. ($L - L'$ and $L' - L$ may or may not be!)

Lemma

Let L be an acceptable language. Then L is decidable if and only if $\Sigma^* - L$ is acceptable.

Alg: run accepters for L and $\Sigma^* - L$ "in parallel"
eventually, one of them has to halt

Part II

The Halting Problem

Undecidable Problems

TMs are our “most general” form of computation—is there anything they can’t compute?

Undecidable Problems

TMs are our “most general” form of computation—is there anything they can’t compute?

Claim

There exists an undecidable language.

Undecidable Problems

TMs are our “most general” form of computation—is there anything they can’t compute?

Claim

There exists an undecidable language.

- There are uncountably many languages.

Undecidable Problems

TMs are our “most general” form of computation—is there anything they can’t compute?

Claim

There exists an undecidable language.

- There are uncountably many languages.
- Each decidable language requires its own TM—of which there are only countably many!

Undecidable Problems

TMs are our "most general" form of computation—is there anything they can't compute?

Claim

There exists an undecidable language.

- There are uncountably many languages.
- Each decidable language requires its own TM—of which there are only countably many!

Are there "natural" or "interesting" undecidable languages?

“Interesting” Undecidable Problems

Many problems throughout math and CS turn out to be undecidable:

- Do two CFGs generate the same language?
- What is the shortest program that outputs a given string?
- Does a multivariate polynomial have an integer root?
- Are two groups isomorphic?
- Can Conway’s Game of Life get from one pattern to another?
- Can a given set of tiles fill the plane?

“Interesting” Undecidable Problems

Many problems throughout math and CS turn out to be undecidable:

- Do two CFGs generate the same language?
- What is the shortest program that outputs a given string?
- Does a multivariate polynomial have an integer root?
- Are two groups isomorphic?
- Can Conway’s Game of Life get from one pattern to another?
- Can a given set of tiles fill the plane?

Our focus: problems that deal with computation / the behavior of programs.

The Halting Problem

Let $L_{HALT} = \{\langle M, w \rangle \mid M(w) \text{ halts}\}$.

How might you try to approach solving this problem?

Idea 1: just run $M(w)$

if $M(w)$ does not halt, we don't know when to "give up" on it.

Idea 2: look at the "code" of M and see if there's an infinite loop

code can be quite complicated...

Halting and Goldbach

```
CheckGoldbach():
    n = 4
    while True:
        flag = False
        for i from 2 to n − 2:
            if i and (n − i) are both prime:
                flag = True
        If flag is False:
            return False
        n = n + 2
```

Does `CheckGoldbach` halt?

Halting and Goldbach

```
CheckGoldbach():
    n = 4
    while True:
        flag = False
        for i from 2 to n − 2:
            if i and (n − i) are both prime:
                flag = True
        If flag is False:
            return False
        n = n + 2
```

Does `CheckGoldbach` halt?

Goldbach's Conjecture: Every even integer above 4 can be written as the sum of two primes. (Open problem since 1742)

Halting and Goldbach

```
CheckGoldbach():
    n = 4
    while True:
        flag = False
        for i from 2 to n − 2:
            if i and (n − i) are both prime:
                flag = True
        If flag is False:
            return False
        n = n + 2
```

Does `CheckGoldbach` halt?

Goldbach's Conjecture: Every even integer above 4 can be written as the sum of two primes. (Open problem since 1742)

- If we could check if `CheckGoldbach` halts, we would resolve Goldbach's conjecture!

Halting is Undecidable

Theorem

$L_{HALT} = \{\langle M, w \rangle \mid M(w) \textit{ halts}\}$ *is undecidable.*

Halting is Undecidable

Theorem

$L_{HALT} = \{\langle M, w\rangle \mid M(w) \text{ halts}\}$ *is undecidable.*

Suppose for contradiction that there is some program `TestHalt` that decides L_{HALT}. We construct the following program:

```
Turing(x):
    Interpret x as the encoding of some TM M_x
    if TestHalt(⟨M_x, x⟩) returns True:
        Enter an infinte loop
    else
        return ‘‘Done!’’
```

Halting is Undecidable

Theorem

$\boldsymbol{L_{HALT}} = \{\langle \boldsymbol{M}, \boldsymbol{w}\rangle \mid \boldsymbol{M}(\boldsymbol{w})$ *halts*$\}$ *is undecidable.*

Suppose for contradiction that there is some program `TestHalt` that decides $\boldsymbol{L_{HALT}}$. We construct the following program:

> **Turing**$(\boldsymbol{x})$:
>   `Interpret` $\boldsymbol{x}$ `as the encoding of some TM` $\boldsymbol{M_x}$
>   **if** `TestHalt`$(\langle \boldsymbol{M_x}, \boldsymbol{x}\rangle)$ `returns True:`
>   `Enter an infinte loop`
>   **else**
>   `return ''Done!''`

What does `Turing`$(\langle$`Turing`$\rangle)$ do?

if TestHalt returns True: Turing infinite loops ✗
if TestHalt returns False: Turing halts ✗

Part III

Undecidability Reductions

Halting with No Input

Our proof that L_{HALT} is undecidable relied on the ability to pass `Turing` its own source code—what happens if we don't allow inputs?

Halting with No Input

Our proof that L_{HALT} is undecidable relied on the ability to pass `Turing` its own source code—what happens if we don't allow inputs?

Theorem

$L_{HNI} = \{\langle M \rangle \mid M \text{ halts given no input}\}$ is undecidable.

Halting with No Input

Our proof that L_{HALT} is undecidable relied on the ability to pass `Turing` its own source code—what happens if we don't allow inputs?

Theorem

$L_{HNI} = \{\langle M \rangle \mid M \text{ halts given no input}\}$ *is undecidable.*

Idea: show that if we can decide L_{HNI}, we can decide L_{HALT}.

`from magic import TestHNI`

TestHalt($\langle M, w \rangle$):

Construct a new program P() that:

1. runs M(w)
2. halts

return TestHNI(<P>)

Halting with No Input

Our proof that L_{HALT} is undecidable relied on the ability to pass `Turing` its own source code—what happens if we don't allow inputs?

Theorem

$L_{HNI} = \{\langle M\rangle \mid M$ *halts given no input*$\}$ *is undecidable.*

Idea: show that if we can decide L_{HNI}, we can decide L_{HALT}.

```
from magic import TestHNI
TestHalt(⟨M, w⟩):
    Construct a program (machine) P() that runs M(w)
    return TestHNI(⟨P⟩)
```

Since L_{HALT} is undecidable, L_{HNI} must be as well.

- From last lecture, this also means that writing a 124 autograder is impossible!

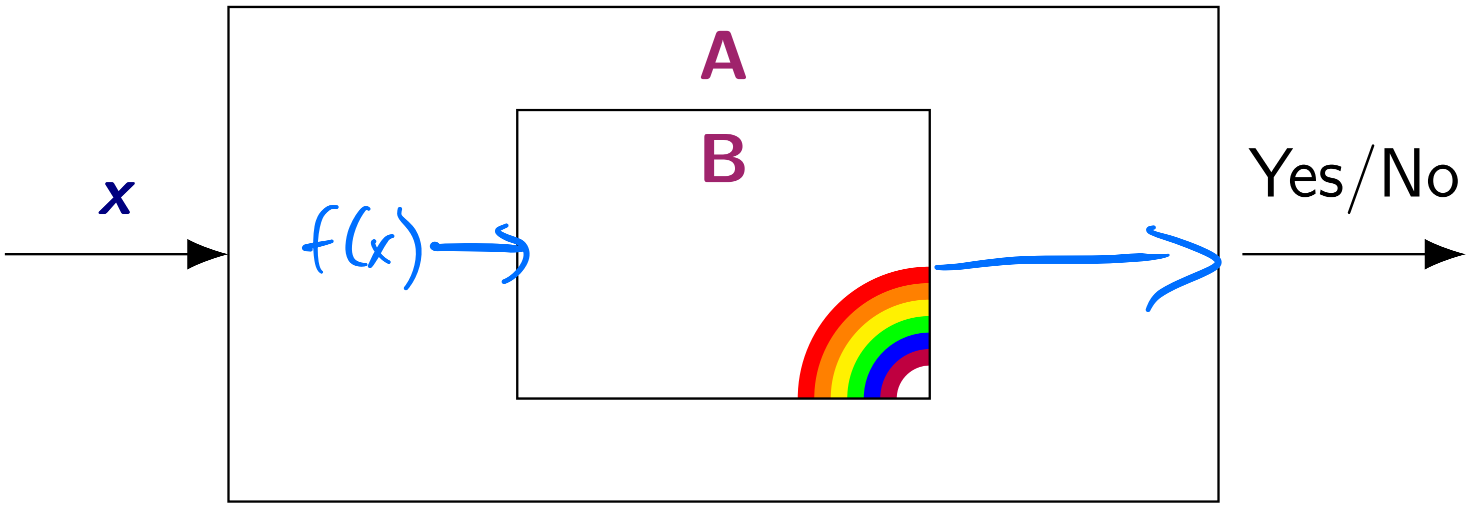
Reducing Decision Problems

Note the nice form of the previous reduction: given an input $\langle M, w \rangle$, we were able to convert it into an input we could pass to `TestHNI`.

Reducing Decision Problems

Note the nice form of the previous reduction: given an input $\langle M, w \rangle$, we were able to convert it into an input we could pass to `TestHNI`.

General pattern: given two *decision* problems A and B, we can reduce A to B by giving a (computable) function f such that $x \in L_A$ if and only if $f(x) \in L_B$.



Reducing Decision Problems

Note the nice form of the previous reduction: given an input $\langle M, w \rangle$, we were able to convert it into an input we could pass to `TestHNI`.

General pattern: given two *decision* problems A and B, we can reduce A to B by giving a (computable) function f such that $x \in L_A$ if and only if $f(x) \in L_B$.

When using this type of reduction, you just have to define (a computable) f and prove that $x \in L_A$ iff $f(x) \in L_B$!

- This is not the *only* way to do a reduction, but it is generally the simplest, and so the one we will use the most.

Accepting $\mathbf{0}^{374}$

Let $L_{374} = \{\langle M \rangle \mid M \text{ accepts } \mathbf{0}^{374}\}$.

Accepting 0^{374}

Let $L_{374} = \{\langle M \rangle \mid M \text{ accepts } 0^{374}\}$.

We can prove L_{374} is undecidable by reducing L_{HALT} to it!

Accepting 0^{374}

Let $L'_{374} = \{\langle M \rangle \mid M \text{ accepts } \text{only } 0^{374}\}$.

We can prove L_{374} is undecidable by reducing L_{HALT} to it!

- What f has $\langle M, w \rangle \in L_{HALT}$ iff $f(\langle M, w \rangle) \in L_{374}$?

$f(\langle M, w \rangle)$ = a program $P(x)$ that:

1. running $M(w)$
2. accept x (iff $x = 0^{374}$)

$\Rightarrow$: if $M(w)$ halts, $P(0^{374})$ reaches line 2 & accepts

$\Leftarrow$: if $M(w)$ does <u>not</u> halt, $P(0^{374})$ gets stuck at 1 & so does <u>not</u> accept

Accepting Regular Language

Let $L_{AccReg} = \{\langle M \rangle \mid \text{Accept}(M) \text{ is regular}\}$.

Accepting Regular Language

Let $L_{AccReg} = \{\langle M \rangle \mid \text{Accept}(M) \text{ is regular}\}$.

We can prove L_{AccReg} is undecidable by reducing L_{HNI} to it.

- What f has $\langle M \rangle \in L_{HNI}$ iff $f(\langle M \rangle) \in L_{AccReg}$?

Accepting The Same Language

Let $L_{AccSame} = \{\langle M_1, M_2\rangle \mid \text{Accept}(M_1) = \text{Accept}(M_2)\}$.

Accepting The Same Language

Let $L_{AccSame} = \{\langle M_1, M_2\rangle \mid \text{Accept}(M_1) = \text{Accept}(M_2)\}$.

What language should we reduce to $L_{AccSame}$? What function f should our reduction use?

Takeaway Points

Determining if a program halts is *impossible* to do algorithmically.

- If such an algorithm existed, we could use it to construct a program that the algorithm is wrong about!

We can use this to prove that "most" problems about the behavior of a program are undecidable.

- If we can reduce an undecidable problem (eg, L_{HALT}) to our problem, we know that our problem is also undecidable.
- Simplest type of reduction: give a (computable) function f such that $\langle M, w \rangle \in L_{HALT}$ iff $f(\langle M, w \rangle)$ is in our language.