

01204211 Discrete Mathematics

Lecture 12b: Undecidability (2)

Jittat Fakcharoenphol

September 21, 2023

Decision problems

- ▶ Given an integer x , is x odd?
- ▶ Given a string w , is w palindrome?
- ▶ Given a string w , is $w \in \{0^n 1^n \mid n \geq 0\}$?
- ▶ Given a map, a starting position s , a destination t , and an integer k , does there exist a path from s to t with distance at most k ?
- ▶ Given a program P and input string w , when running P with w as an input, does P terminate?

Decision problems and languages

For this problem:

Given an integer x , is x odd?

we can define a corresponding language

$$L_E = \{\dots, -6, -4, -2, 0, 2, 4, 6, \dots\}.$$

To solve this problem, given x , we can ask if $x \in L_E$.

Deciders

We say that a python program P **decides** the language L if for any input string x , P when running with x as an input,

- ▶ P always terminates,
- ▶ P outputs **yes**, if $x \in L$, and
- ▶ P outputs **no**, if $x \notin L$.

Deciders

We say that a python program P **decides** the language L if for any input string x , P when running with x as an input,

- ▶ P always terminates,
- ▶ P outputs **yes**, if $x \in L$, and
- ▶ P outputs **no**, if $x \notin L$.

If we believe that anything that a computer can do can be written as a python program, and there is no python program that decides a language A , then we say that

A is undecidable.

Language HALTA and HALT

Let \mathbb{P} be the set of all python programs. Let the language HALTA be

$$\{P \in \mathbb{P} \mid \text{when running } P \text{ with } P \text{ as an input, } P \text{ terminates}\}$$

Or with a more concise notation:

$$\text{HALTA} = \{P \in \mathbb{P} \mid P(P) \text{ terminates}\}.$$

We also have another related language

$$\text{HALT} = \{(P, w) \mid P \text{ is a python program such that } P(w) \text{ terminates}\}$$

Lemma 1

HALT_A and HALT are undecidable.

Reduction: proving undecidability of HALT

- ▶ We show that if HALT is decidable, then HALT A is also decidable.
- ▶ However, HALT A IS UNDECIDABLE.
- ▶ We conclude that HALT is also undecidable.

$P \Rightarrow Q$

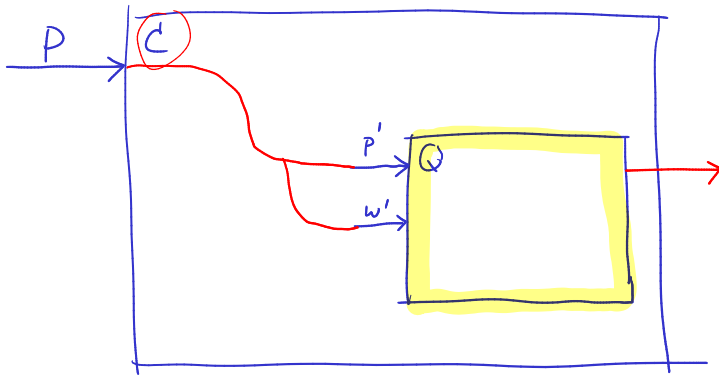
$\neg P$

Reduction in picture

► If HALT is decidable, HALT_A is decidable.

su P', w'
miss $P(w)$ term

||
su w term P
miss $P(P)$
term?



► HALT_A is undecidable

⇒ HALT is undecidable.

— "P accepts w"

If Accept dec~
⇒ xxx decide

Let $\text{ACCEPT} = \{(P, w) \mid P \in \mathbb{P} \text{ and } P(w) \text{ terminates with acceptance}\}$.

Lemma 2

ACCEPT is undecidable.

Proof.

We prove the lemma by contradiction. Assume that there is a python program Q that decides ACCEPT.

Let $\text{ACCEPT} = \{(P, w) \mid P \in \mathbb{P} \text{ and } P(w) \text{ terminates with acceptance}\}$.

Lemma 2

ACCEPT is undecidable.

Proof.

We prove the lemma by contradiction. Assume that there is a python program Q that decides ACCEPT . We construct a program C that decides HALT as follows

Let $\text{ACCEPT} = \{(P, w) \mid P \in \mathbb{P} \text{ and } P(w) \text{ terminates with acceptance}\}$.

Lemma 2

ACCEPT is undecidable.

Proof.

We prove the lemma by contradiction. Assume that there is a python program Q that decides ACCEPT . We construct a program C that decides HALT as follows

Program C

Input P, w

1. Replace every "print('no')" statement in P with "print('yes')"
1. if $Q(P, w) == \text{'yes'}$:
2. print('yes')
3. else
4. print('no')

Proof (cont.)

Program C

Input P, w

1. Replace every "print('no')" statement in P with "print('yes')"
1. if $Q(P, w) == \text{'yes'}$:
2. print('yes')
3. else
4. print('no')

Version C decides HALT

We have to make sure that our reduction is correct by considering two cases.

Case 1: when $P(w)$ halts.

P' accepts w , Q accepts P, w
 \Rightarrow if C accepts (P, w) then

Proof (cont.)

Program C

Input P, w

1. Replace every "print('no')" statement in P with "print('yes')"
1. if $Q(P, w) == \text{'yes'}$:
2. print('yes')
3. else
4. print('no')

We have to make sure that our reduction is correct by considering two cases.

Case 1: when $P(w)$ halts.

Case 2: when $P(w)$ does not halt.

$P'(w)$ halts, Q reject, C reject (P, w) mismatches

Proof (cont.)

Program C

Input P, w

```
1.  Replace every "print('no')" statement in P with "print('yes')"  
1.  if  $Q(P, w) == \text{'yes'}$ :  
2.      print('yes')  
3.  else  
4.      print('no')
```

We have to make sure that our reduction is correct by considering two cases.

Case 1: when $P(w)$ halts.

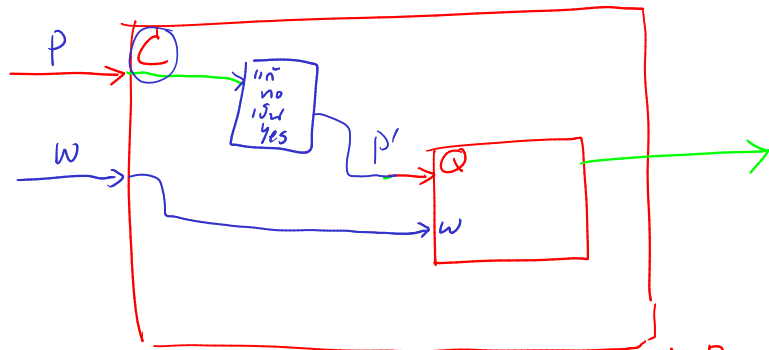
Case 2: when $P(w)$ does not halt.

Since in both cases, C answers correctly, we know that given program Q deciding ACCEPT, we can construct a program C that decides HALT. However, we know that HALT is undecidable; thus, we reach a contradiction. We conclude that ACCEPT is also undecidable. \square

Reduction from HALT to ACCEPT in picture

if program Q decides ACCEPT

if \exists Turing C that decides HALT



if P accepts $w \Rightarrow Q$ accepts ✓

~~if P rejects $w \Rightarrow Q$ rejects ✗~~

because P

if $(P, w) \in \text{HALT}$, C accepts (P, w) ✗

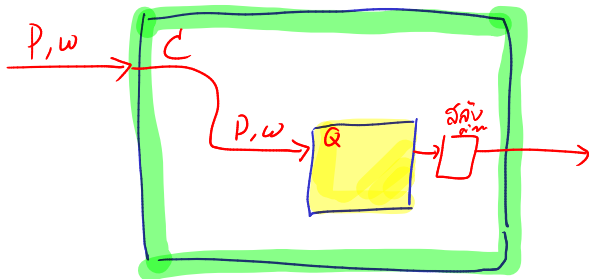
\rightarrow if $(P, w) \notin \text{HALT}$, C rejects (P, w) ✓

How about REJECT?

Let

$$\text{REJECT} = \{(P, w) \mid P \in \mathbb{P} \text{ and } P \text{ rejects } w\}.$$

How about NOTHALT?



Let

$$\text{NOTHALT} = \{(P, w) \mid P \in \mathbb{P} \text{ and } P(w) \text{ does not terminate}\}.$$

If NOTHALT is decidable \Rightarrow HALT is decidable

& program Q n^o
decide NOTHALT

& Turing C n^o
decides HALT

Language of program P

(q : reduction in
of HALT)

- Assume if ALL is decidable,
then there exists a TM Q which decides ALL
- if we know HALT is decidable then
there exists a TM C which decides HALT.

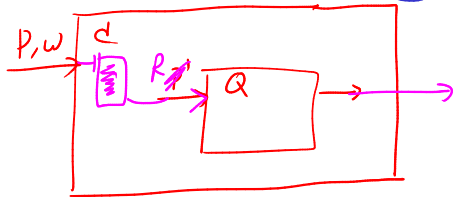
For a python program P , let $L(P)$ be the set of all strings that P accepts, i.e.,

$$L(P) = \{w \in \Sigma^* \mid P(w) = \text{yes}\}.$$

Let

$$\underline{\text{ALL}} = \{P \in \mathbb{P} \mid L(P) = \Sigma^*\}.$$

Set of all python



Lemma 3

ALL is undecidable.

Proof.

We prove by reduction from HALT. Assume that ALL is decidable, i.e., there is a python program Q that decides ALL. We construct program C that decides HALT as follows

Program C (input: P, w)

1. Construct another program R from P and w :
 - | Program R (input: x)
 - | 1. Run program P on input w , suppressing any output from P
 - | 2. Accept x
2. if $Q(R) == \text{'yes'}$:
3. return 'yes'
4. else: return 'no'

Proof (cont.)

Program C (input: P, w)

1. Construct another program R from P and w:
 - | Program R (input: x)
 - | 1. Run program P on input w , suppressing any output from P
 - | 2. Accept x
2. if $Q(R) == \text{'yes'}$:
3. return 'yes'
4. else: return 'no'

To ensure the correctness, we have to consider two cases.

Case 1: when $P(w)$ halts.

R runs on string x , $R \in ALL$, $Q(R) == \text{yes}$
C must yes.

Proof (cont.)

Program C (input: P, w)

1. Construct another program R from P and w:

 | Program R (input: x)

 | 1. Run program P on input w, suppressing any output from P \times

 | 2. Accept x

2. if $Q(R) == \text{'yes'}$:

3. return 'yes'

4. else: return 'no'

To ensure the correctness, we have to consider two cases.

Case 1: when $P(w)$ halts.

Case 2: when $P(w)$ does not halt. $L(R) = \emptyset$, $R \notin ALL$, $Q(R) = \text{'no'}$, C reject P, w -

Proof (cont.)

Program C (input: P, w)

1. Construct another program R from P and w:

| Program R (input: x)

| 1. Run program P on input w, suppressing any output from P ✕

| 2. Accept x

2. if $Q(R) == \text{'yes'}$:

3. return 'yes'

4. else: return 'no'

To ensure the correctness, we have to consider two cases.

Case 1: when $P(w)$ halts.

Case 2: when $P(w)$ does not halt.

Since we can construct a program for HALT using a program that decides ALL, but HALT is undecidable; therefore, we conclude that ALL is undecidable. □

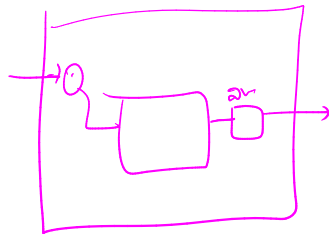
EMPTY

P, w သော R
 A' သာ $(P, w) \in \text{HALT}$
 $L(R) = \Sigma^*$
သာ $(P, w) \notin \text{HALT}$
 $L(R) = \emptyset$

Let

$$\text{EMPTY} = \{P \in \mathbb{P} \mid L(P) = \emptyset\}.$$

- Assume သာ EMPTY is decidable.
သို့သော်လည်းကောင်း Q သည် decides EMPTY
- ဒါ့ကြောင့် HALT is decidable လော
အားဖြင့် C သည် decides HALT.



Lemma 4

EMPTY is undecidable.

Proof.

We prove by reduction from HALT. Assume that EMPTY is decidable, i.e., there is a python program Q that decides EMPTY. We construct program C that decides HALT as follows

Program C (input: P,w)

1. Construct another program R from P and w:

| Program R (input: x)

| 1. Run program P on input w, suppressing any output from P ↓

| 2. Accept x ✗

2. if Q(R) == 'yes':

3. return 'no'

4. else: return 'yes'

. $(P,w) \in \text{HALT} \Rightarrow R \notin \text{EMPTY}$

. $(P,w) \notin \text{HALT} \Rightarrow R \in \text{EMPTY}$

Q ✗ decides no EMPTY

Since we can construct a program for HALT using a program that decides EMPTY, but HALT is undecidable; therefore, we conclude that EMPTY is undecidable. □

Lemma 5

Let $\text{EQ} = \{(P_1, P_2) \mid P_1, P_2 \in \mathbb{P} \text{ and } L(P_1) = L(P_2)\}$. EQ is undecidable.

Proof.

We prove by reduction from ALL. Assume that EQ is decidable, i.e., there is a python program Q that decides EQ. We construct program C that decides ALL as follows

Program C (input: P)

1. Construct another program R:

 Program R (input: x)

 1. Accept x

2. if $Q(P, R) == \text{'yes'}$:

3. return 'yes'

4. else: return 'no'

$$L(R) = \Sigma^*$$

$$\text{or } P \in \text{ALL},$$

$$P \notin \text{ALL},$$

$$L(P) = \Sigma^* = L(R)$$

$$L(P) \neq \Sigma^* = L(R)$$

Since we can construct a program for ALL using a program that decides EQ, but ALL is undecidable; therefore, we conclude that EQ is undecidable. □

Another proof for EQ

Proof.

We prove by reduction from `EMPTY`. Assume that `EQ` is decidable, i.e., there is a python program `Q` that decides `EQ`. We construct program `C` that decides `EMPTY` as follows

Program `C` (input: `P`)

1. Construct another program `R`:
 - | Program `R` (input: `x`)
 - | 1. Reject `x`
2. if `Q(P,R) == 'yes'`:
3. return 'yes'
4. else: return 'no'

Since we can construct a program for `EMPTY` using a program that decides `EQ`, but `EMPTY` is undecidable; therefore, we conclude that `EQ` is undecidable. □

Lemma 6

Let $\text{HELLO} = \{P \in \mathbb{P} \mid L(P) = \{\text{hello}\}\}$. HELLO is undecidable.

INCORRECT PROOF.

We prove by reduction from EQ. Assume that EQ is decidable, i.e., there is a python program Q that decides EQ. We construct program C that decides HELLO as follows

Program C (input: P)

1. Construct another program R:

| Program R (input: x)

| 1. if $x == \text{'hello'}$:

| 2. print('yes') # accept x

| 3. else

| 4. print('no') # reject x

2. if $Q(P, R) == \text{'yes'}$:

3. return 'yes'

4. else: return 'no'

- Assume if HELLO is decidable, then there exists a program Q that decides HELLO
- And since if EQ is decidable then there exists a program C that decides EQ.

$P \Rightarrow Q$

IF EQ is decidable \Rightarrow HELLO is decidable.

$\neg P$



Lemma 7

Let $\text{HELLO} = \{P \in \mathbb{P} \mid L(P) = \{\text{hello}\}\}$. *HELLO is undecidable.*

Proof

We prove by reduction from HALT. Assume that HELLO is decidable, i.e., there is a python program Q that decides HELLO. We construct program C that decides HALT as follows

Program C (input: P, w)

1. Construct another program R:

| Program R (input: x)

| 1. if $x \neq \text{'hello'}$:

| 2. print('no') # reject x

| 3. Replace any output statements from P

| 4. Run the modified P on w

| 5. print('yes') # accept x

2. if $Q(R) == \text{'yes'}$:

3. return 'yes'

4. else: return 'no'

Proof (cont.)

Program C (input: P,w)

1. Construct another program R:
 - | Program R (input: x)
 - | 1. if x != 'hello':
 - | 2. print('no') # reject x
 - | 3. Replace any output statements from P
 - | 4. Run the modified P on w
 - | 5. print('yes') # accept x
2. if Q(~~R~~) == 'yes':
3. return 'yes'
4. else: return 'no'

We consider two cases:

Case 1: $P(w)$ halts. $L(R) = \{ \text{hello} \}$

Proof (cont.)

Program C (input: P,w)

1. Construct another program R:
 - | Program R (input: x)
 - | 1. if $x \neq \text{'hello'}$:
 - | 2. print('no') # reject x
 - | 3. Replace any output statements from P
 - | 4. Run the modified P on w
 - | 5. print('yes') # accept x
2. if $Q(R) == \text{'yes'}$:
3. return 'yes'
4. else: return 'no'

We consider two cases:

Case 1: $P(w)$ halts.

Case 2: $P(w)$ does not halt. $L(R) \neq \{\text{hello}\}$

Proof (cont.)

Program C (input: P,w)

1. Construct another program R:
 - | Program R (input: x)
 - | 1. if $x \neq \text{'hello'}$:
 - | 2. `print('no')` # reject x
 - | 3. Replace any output statements from P
 - | 4. Run the modified P on w
 - | 5. `print('yes')` # accept x
2. if $Q(R) == \text{'yes'}$:
3. `return 'yes'`
4. else: `return 'no'`

We consider two cases:

Case 1: $P(w)$ halts.

Case 2: $P(w)$ does not halt.

Since we can construct a program that decides HALT given a program that decides HELLO, but HALT is undecidable. We conclude that HELLO is also undecidable. □

Python as computation

Do you believe in this assumption:

Anything that a computer can do can be written as a python program.

Turing machines

Anything that a computer can do can be carried out using Turing machines.

Turing machines

Anything that a computer can do can be carried out using Turing machines.

Any possible computation can be performed by Turing machines.

Turing Machines

Turing Machines

- ▶ Proposed by Alan Turing in 1936.

Turing Machines

- ▶ Proposed by Alan Turing in 1936.
- ▶ A finite automaton with an **unlimited** memory with **unrestricted** access.

Turing Machines

- ▶ Proposed by Alan Turing in 1936.
- ▶ A finite automaton with an **unlimited** memory with **unrestricted** access.
- ▶ Can perform any tasks that a computer can. (we'll see)

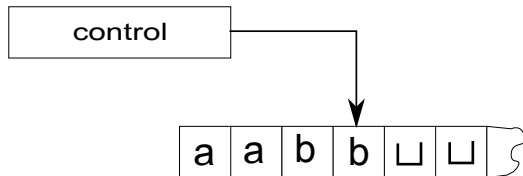
Turing Machines

- ▶ Proposed by Alan Turing in 1936.
- ▶ A finite automaton with an **unlimited** memory with **unrestricted** access.
- ▶ Can perform any tasks that a computer can. (we'll see)
- ▶ However, there are problems that TM can't solve. These problems are beyond the limit of computation.

Components

- ▶ An infinite **tape**.
- ▶ A tape head that can
 - ▶ **read and write** to the tape, and
 - ▶ **move** around the tape.

Schematic



How Turing machines work

- ▶ The tape initially contains an input string.

How Turing machines work

- ▶ The tape initially contains an input string.
- ▶ The rest of the tape is blank (denoted by \sqcup).

How Turing machines work

- ▶ The tape initially contains an input string.
- ▶ The rest of the tape is blank (denoted by \sqcup).
- ▶ The machine reads a symbol from of the tape where its head is at.

How Turing machines work

- ▶ The tape initially contains an input string.
- ▶ The rest of the tape is blank (denoted by \sqcup).
- ▶ The machine reads a symbol from of the tape where its head is at.
- ▶ It can write a symbol back and move **left** or **right**.

How Turing machines work

- ▶ The tape initially contains an input string.
- ▶ The rest of the tape is blank (denoted by \sqcup).
- ▶ The machine reads a symbol from of the tape where its head is at.
- ▶ It can write a symbol back and move **left** or **right**.
- ▶ At the end of the computation, the machine outputs **accept** or **reject**, by entering accept state of reject state. (After changing, it halts.)

How Turing machines work

- ▶ The tape initially contains an input string.
- ▶ The rest of the tape is blank (denoted by \sqcup).
- ▶ The machine reads a symbol from of the tape where its head is at.
- ▶ It can write a symbol back and move **left** or **right**.
- ▶ At the end of the computation, the machine outputs **accept** or **reject**, by entering accept state of reject state. (After changing, it halts.)
- ▶ It can go on forever (not entering any accept or reject states).

Examples

Formal definition of a Turing Machine

- ▶ Again, the important part is the definition of the transition function.

Formal definition of a Turing Machine

- ▶ Again, the important part is the definition of the transition function.
- ▶ The machine look at the tape **symbol** and consider its **current state**, then makes a move by **writing some symbol** on the tape and **moving its head left or right**.

Formal definition of a Turing Machine

- ▶ Again, the important part is the definition of the transition function.
- ▶ The machine look at the tape **symbol** and consider its **current state**, then makes a move by **writing some symbol** on the tape and **moving its head left or right**.
- ▶ Thus,
 - ▶ **Input:** current state and the symbol on the tape

Formal definition of a Turing Machine

- ▶ Again, the important part is the definition of the transition function.
- ▶ The machine look at the tape **symbol** and consider its **current state**, then makes a move by **writing some symbol** on the tape and **moving its head left or right**.
- ▶ Thus,
 - ▶ **Input:** current state and the symbol on the tape
 - ▶ **Output:** next state, a symbol to be written to the tape, and the new state.

Formal definition of a Turing Machine

- ▶ Again, the important part is the definition of the transition function.
- ▶ The machine look at the tape **symbol** and consider its **current state**, then makes a move by **writing some symbol** on the tape and **moving its head left or right**.
- ▶ Thus,
 - ▶ **Input:** current state and the symbol on the tape
 - ▶ **Output:** next state, a symbol to be written to the tape, and the new state.

Formal definition of a Turing Machine

- ▶ Again, the important part is the definition of the transition function.
- ▶ The machine look at the tape **symbol** and consider its **current state**, then makes a move by **writing some symbol** on the tape and **moving its head left or right**.
- ▶ Thus,
 - ▶ **Input:** current state and the symbol on the tape
 - ▶ **Output:** next state, a symbol to be written to the tape, and the new state.
- ▶ So, δ is in the form: $Q \times \Gamma \rightarrow Q \times \Gamma \times \{\text{LEFT}, \text{RIGHT}\}$

Formal definition of a Turing Machine

- ▶ Again, the important part is the definition of the transition function.
- ▶ The machine look at the tape **symbol** and consider its **current state**, then makes a move by **writing some symbol** on the tape and **moving its head left or right**.
- ▶ Thus,
 - ▶ **Input:** current state and the symbol on the tape
 - ▶ **Output:** next state, a symbol to be written to the tape, and the new state.
- ▶ So, δ is in the form: $Q \times \Gamma \rightarrow Q \times \Gamma \times \{\text{LEFT}, \text{RIGHT}\}$
- ▶ E.g., if $\delta(q, a) = (r, b, \text{LEFT})$, then if the machine is in state q and reads a , it will change its state to r , write b to the tape and move to the left.

Definition

Definition (Turing Machine)

A **Turing machine** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where Q, Σ, Γ are finite sets and

1. Q is the set of states,
2. Σ is the input alphabet not containing the **blank symbol** \sqcup ,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subset \Gamma$,
4. $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{\text{LEFT}, \text{RIGHT}\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{accept} \in Q$ is the accept state, and
7. $q_{reject} \in Q$ is the reject state, where $q_{accept} \neq q_{reject}$.

Variants of Turing Machines

- ▶ There are many alternative definitions of TM's.

Variants of Turing Machines

- ▶ There are many alternative definitions of TM's.
- ▶ They are called **variants** of TM's.

Variants of Turing Machines

- ▶ There are many alternative definitions of TM's.
- ▶ They are called **variants** of TM's.
- ▶ They all have the same power. This demonstrates the **robustness** in the definition of TM's.

Variants of Turing Machines

- ▶ There are many alternative definitions of TM's.
- ▶ They are called **variants** of TM's.
- ▶ They all have the same power. This demonstrates the **robustness** in the definition of TM's. Also, this is an evidence that TM's “capture” the idea of computation (because whatever computing machine we can think of they are all equivalent to TM's).

TM with “stay put”

- ▶ Let's start with an easy variant. Suppose we allow additional head movement: “stay put (S)”.

TM with “stay put”

- ▶ Let's start with an easy variant. Suppose we allow additional head movement: “stay put (S)”.
- ▶ The transition function will be of the form

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{\text{LEFT}, \text{RIGHT}, \text{S}\}.$$

- ▶ Does this give TM's more power?

TM with “stay put”

- ▶ Let's start with an easy variant. Suppose we allow additional head movement: “stay put (S)”.
- ▶ The transition function will be of the form

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{\text{LEFT}, \text{RIGHT}, \text{S}\}.$$

- ▶ Does this give TM's more power?
- ▶ Not really. We can convert a TM with “stay put” to a standard TM as follows.

TM with “stay put”

- ▶ Let's start with an easy variant. Suppose we allow additional head movement: “stay put (S)”.
- ▶ The transition function will be of the form

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{\text{LEFT}, \text{RIGHT}, \text{S}\}.$$

- ▶ Does this give TM's more power?
- ▶ Not really. We can convert a TM with “stay put” to a standard TM as follows.
 - ▶ For any “stay put” transition, we replace with two transitions: “right” and “left”.

Multitape Turing Machines

- ▶ A **multitape Turing machines** has many tapes.

Multitape Turing Machines

- ▶ A **multitape Turing machines** has many tapes.
- ▶ For each tape, the machine has a head for reading and writing it.

Multitape Turing Machines

- ▶ A **multitape Turing machines** has many tapes.
- ▶ For each tape, the machine has a head for reading and writing it.
- ▶ The input appears on tape 1; all other tapes contain blanks.

Multitape Turing Machines

- ▶ A **multitape Turing machines** has many tapes.
- ▶ For each tape, the machine has a head for reading and writing it.
- ▶ The input appears on tape 1; all other tapes contain blanks.
- ▶ Let k be the number of tapes. The transition function can be defined as

$$\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{\text{LEFT}, \text{RIGHT}, \text{STAY}\}^k.$$

Multitape Turing Machines

- ▶ A **multitape Turing machines** has many tapes.
- ▶ For each tape, the machine has a head for reading and writing it.
- ▶ The input appears on tape 1; all other tapes contain blanks.
- ▶ Let k be the number of tapes. The transition function can be defined as

$$\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{\text{LEFT}, \text{RIGHT}, \text{STAY}\}^k.$$

- ▶ E.g., if $\delta(q_i, a_1, \dots, a_k) = (q_j, b_1, \dots, b_k, \text{LEFT}, \text{RIGHT}, \dots, \text{LEFT})$ then if the machine is at state q_i and each head on tape i reads symbol a_i , it'll write b_i on each tape i , change state to q_j and move each head accordingly.

Nondeterministic Turing Machines

- ▶ A nondeterministic Turing machine can make “nondeterministic” move.

Nondeterministic Turing Machines

- ▶ A nondeterministic Turing machine can make “nondeterministic” move.
- ▶ As expected, its transition function has the form

$$\delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{\text{LEFT}, \text{RIGHT}\}).$$

Nondeterministic Turing Machines

- ▶ A nondeterministic Turing machine can make “nondeterministic” move.
- ▶ As expected, its transition function has the form

$$\delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{\text{LEFT}, \text{RIGHT}\}).$$

- ▶ Again, we view the computation of a nondeterministic Turing machine as a tree, where each branching corresponds to the place where the TM can make different moves.

Nondeterministic Turing Machines

- ▶ A nondeterministic Turing machine can make “nondeterministic” move.
- ▶ As expected, its transition function has the form

$$\delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{\text{LEFT}, \text{RIGHT}\}).$$

- ▶ Again, we view the computation of a nondeterministic Turing machine as a tree, where each branching corresponds to the place where the TM can make different moves.
- ▶ Can nondeterminism help?

Equivalence in Power

- ▶ **Should I write programs in C or Pascal?**

Equivalence in Power

- ▶ **Should I write programs in C or Pascal?**
- ▶ **Should I write programs in Python or Prolog?**

Equivalence in Power

- ▶ **Should I write programs in C or Pascal?**
- ▶ **Should I write programs in Python or Prolog?**
- ▶ **Should I write programs in Ruby or LISP?**

They are all the same, in terms of computability

Since you can write a C interpreter in Pascal and Pascal interpreter in C,

They are all the same, in terms of computability

Since you can write a C interpreter in Pascal and Pascal interpreter in C, what you can do in C, you can do in Pascal.

Turing machine

If you believe that Turing machines are ultimate model of computing, all those programming languages are equivalent because they all can simulate Turing machines (and they runs on Turing machines).

Two sides of a coin

- ▶ Computers are powerful

Two sides of a coin

- ▶ Computers are powerful (???)

Two sides of a coin

- ▶ Computers are powerful (???)
- ▶ How powerful?

Two sides of a coin

- ▶ Computers are powerful (???)
- ▶ How powerful?
- ▶ To understand that, we want to see samples of tasks that computers **can** do, and samples of tasks that they **can't** do.

Two sides of a coin

- ▶ Computers are powerful (???)
- ▶ How powerful?
- ▶ To understand that, we want to see samples of tasks that computers **can** do, and samples of tasks that they **can't** do.
- ▶ It's one story to show that computers can do something.

Two sides of a coin

- ▶ Computers are powerful (???)
- ▶ How powerful?
- ▶ To understand that, we want to see samples of tasks that computers **can** do, and samples of tasks that they **can't** do.
- ▶ It's one story to show that computers can do something. It's another to show that computers can't do something.
 - ▶ Maybe there's a limitation with “this” computer, but “other” computers might be able to do that thing.
 - ▶ We want to be able to say that **for all** computers.

Two sides of a coin

- ▶ Computers are powerful (???)
- ▶ How powerful?
- ▶ To understand that, we want to see samples of tasks that computers **can** do, and samples of tasks that they **can't** do.
- ▶ It's one story to show that computers can do something. It's another to show that computers can't do something.
 - ▶ Maybe there's a limitation with “this” computer, but “other” computers might be able to do that thing.
 - ▶ We want to be able to say that **for all** computers. In fact, **for any “thinkable”** computers.

What is a computer?

What is a computer?
Something that computes?

What is a computer?
Something that computes?
What is computation?

What is a computer?

Something that computes?

What is computation?

An act of following some instructions?

What is a computer?

Something that computes?

What is computation?

An act of following some instructions?

An act of following some algorithm?

What is a computer?

Something that computes?

What is computation?

An act of following some instructions?

An act of following some algorithm?

What is an algorithm?

Hilbert's problems

Mathematician David Hilbert asked:

"Find a process according to which it can be determined by a finite number of operations if a given polynomial has integral root"

To say NO

We need an argument (a mathematical proof) that covers all possible “processes” or all “computations”.

Possible definitions

- ▶ Church's λ -calculus
- ▶ Turing's machines

Possible definitions

- ▶ Church's λ -calculus
- ▶ Turing's machines

They both turned out to be **equivalent**.

Church-Turing thesis

Turing machine algorithms = intuitive notion of algorithms

Final answer to Hilbert

No, there doesn't exist any algorithm for determining if a polynomial has integral root.