Turing Machine III & Undecidability

Zeyu Mi

2022-12-21

Adapted From:

IALC 9.1, 9.2

Stanford CS103 Turing Machine

Universal Turing Machine

通用图灵机

An Observation

- When we've been discussing Turing machines, we've talked about designing specific TMs to solve specific problems.
  - TM for 0^n1^n
- Does this match your real-world experiences? Do you have one computing device for each task you need to perform?
- Or can we make a "**reprogrammable** Turing machine?"

A TM Simulator

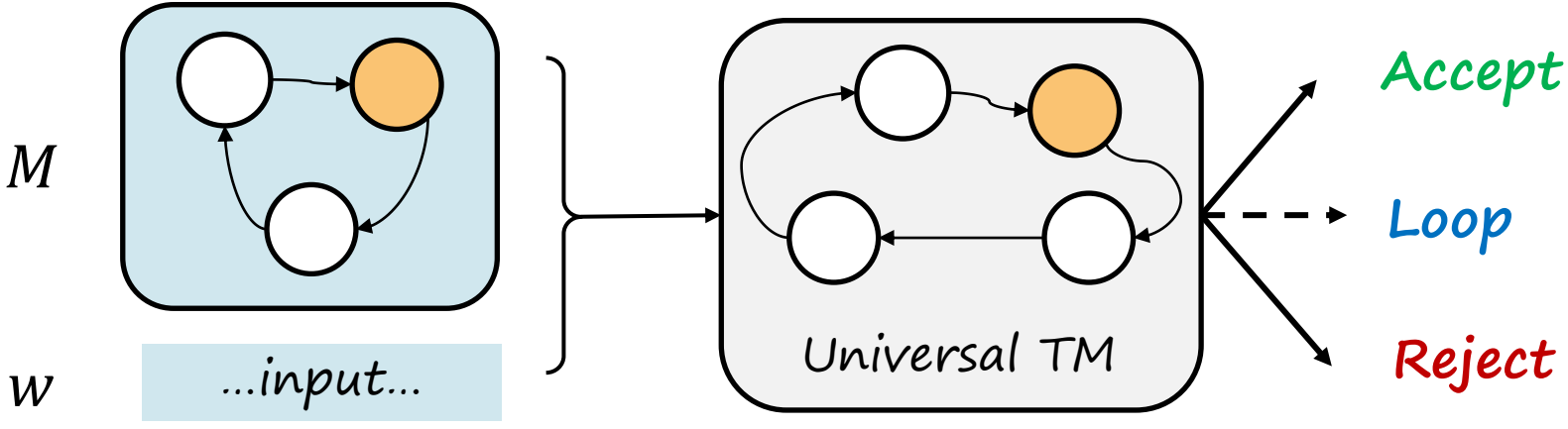
- We've known it is possible to program a TM simulator on an unbounded-memory computer.
- We could imagine it as a method

  boolean simulateTM(TM M, string w)

- with the following behavior:
  - If M accepts w, then *simulateTM(M, w)* returns true.
  - If M rejects w, then *simulateTM(M, w)* returns false.
  - If M loops on w, then *simulateTM(M, w)* loops infinitely.

A TM Simulator

- It is also known that anything that can be done with an unbounded-memory computer can be done with a TM
- This means that there must be some TM that has the behavior of this *simulateTM* method.



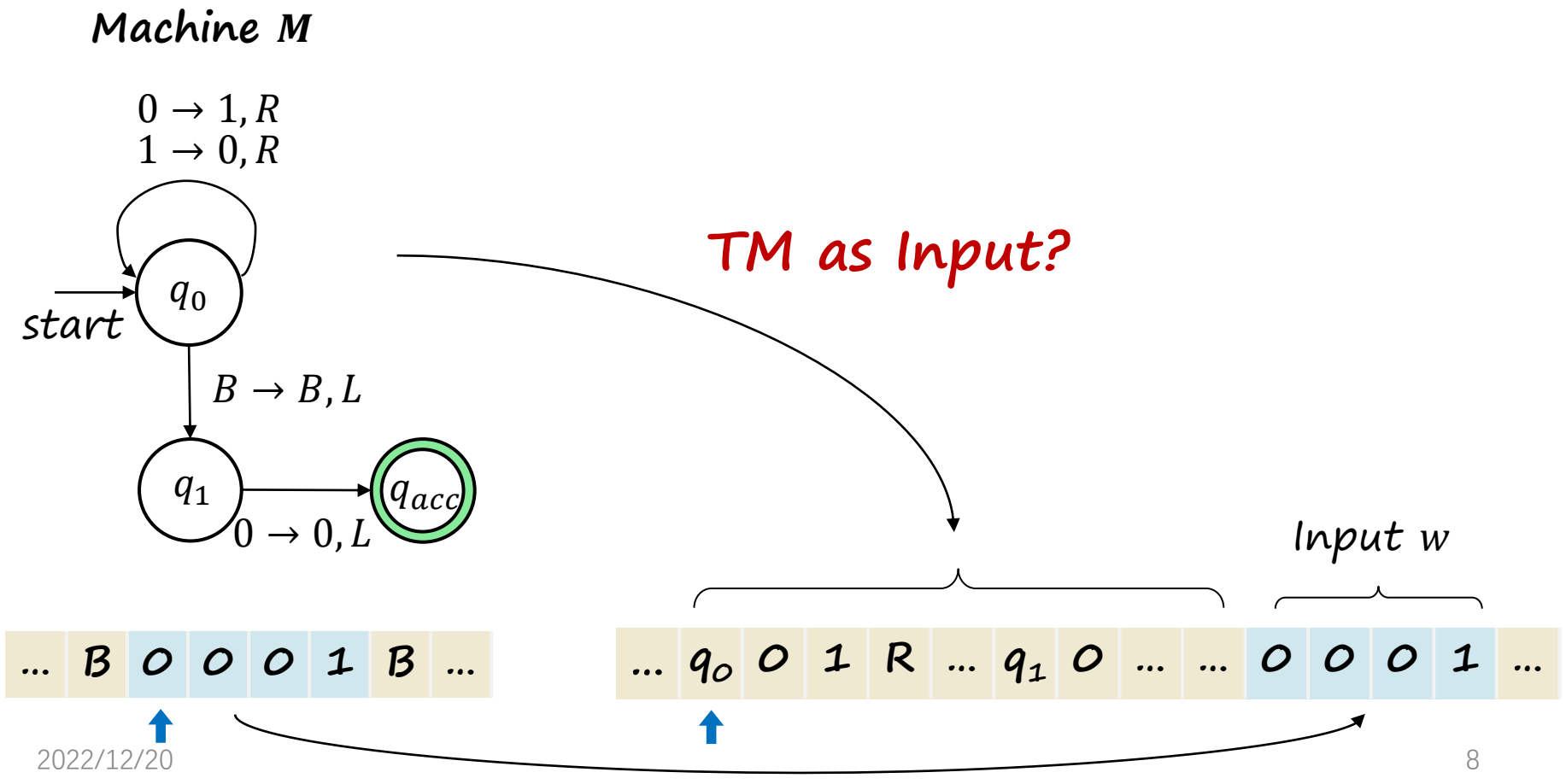
The Universal Turing Machine

- **Theorem (Turing, 1936):** There is a Turing machine U_{TM} called **the Universal Turing Machine** that, when run on an input of the form $\langle M, w \rangle$, where M is a Turing machine and w is a string, simulates M running on w and **does whatever** M does on w (accepts, rejects, or loops).
- U_{TM} behaves as follows:
  - If M accepts w, then U_{TM} accepts $\langle M, w \rangle$
  - If M rejects w, then U_{TM} rejects $\langle M, w \rangle$
  - If M loops on w, then U_{TM} loops on $\langle M, w \rangle$

Encoding Input with binary

- Input string may contains any possible character in the input alphabet of M
- But we know everything on your computer is a string over $\{0, 1\}$
- We can let the input alphabet to be $\{0,1\}$
- It not necessary to limit the alphabet as $\{0,1\}$, but only for simplicity.

The Universal Turing Machine



Encoding TM

- In order to take a Turing Machine as an input, we need to encoding the TM. Similarly, we shall encode TM with binary.
- We first assign integers to the states, tape symbols and directions
  - We assume the states are $q_1, q_2, \dots, q_k$ for some k. $\boldsymbol{q_1}$ is the **input state** and $\boldsymbol{q_2}$ is the **only accept state**.(Is it right?)
  - The tape symbols are $X_1, X_2, \dots, X_m$ for some m. $\boldsymbol{X_1, X_2, X_3}$ **are** $\boldsymbol{0, 1, B}$ respectively.
  - Refer to direction $\boldsymbol{L}$ **as** $\boldsymbol{D_1}$, $\boldsymbol{R}$ **as** $\boldsymbol{D_2}$

Encoding TM

- After assigning each state, symbol and direction an integer, we can encode the transition function $\boldsymbol{\delta}$.
- Suppose one transition rule is $\boldsymbol{\delta(q_i, X_j) = (q_k, X_l, D_m)}$, we shall code this rule by the string $\boldsymbol{0^i 1 0^j 1 0^k 1 0^l 1 0^m}$.
  - Notice $\boldsymbol{i, j, k, l, m}$ are at least one, so there're no occurrences of two or more consecutive 1's with in the code for a transition
- So let $\boldsymbol{C_i}$ donate the code for the ith transition rule, we can encode the whole TM as:
  - $C_1 11 C_2 11 C_3 11 \ldots 11 C_n$

Example of Code for TM

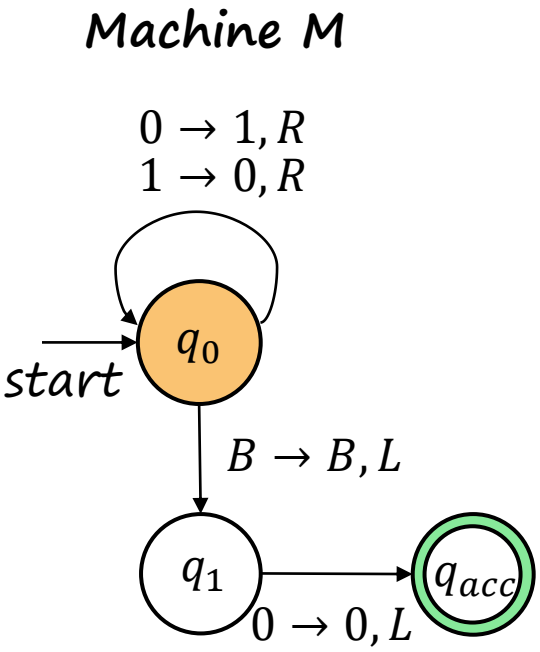
- Let a TM: $M = (\{q_1, q_2, q_3\}, \{0,1\}, \{0,1, B\}, \{\delta\}, q_1, B, q_2)$
  - $X_1 = 0, X_2 = 1, X_3 = B, D_1 = L, D_2 = R$
- Transition Function δ:
  - $\boldsymbol{\delta(q_1, 1) = (q_3, 0, R)}$
  - $\boldsymbol{\delta(q_3, 0) = (q_1, 1, R)}$
  - $\boldsymbol{\delta(q_3, 1) = (q_2, 0, R)}$
  - $\boldsymbol{\delta(q_3, B) = (q_3, 1, L)}$

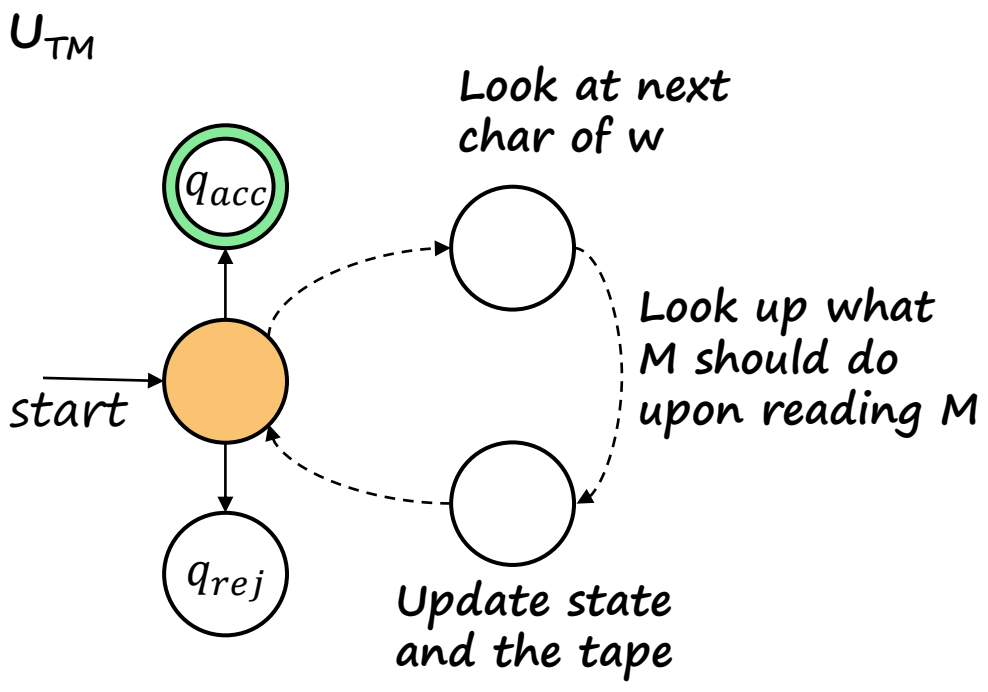
Example of Code for TM

- Let a TM: $M = (\{q_1, q_2, q_3\}, \{0,1\}, \{0,1, B\}, \{\delta\}, q_1, B, q_2)$
  - $X_1 = 0, X_2 = 1, X_3 = B, D_1 = L, D_2 = R$
- Transition Function δ:
  - $\boldsymbol{\delta(q_1, 1) = (q_3, 0, R) \Rightarrow \delta(q_1, X_2) = (q_3, X_1, D_2) \Rightarrow 01001000101 00}$
  - $\boldsymbol{\delta(q_3, 0) = (q_1, 1, R) \Rightarrow \delta(q_3, X_1) = (q_1, X_2, D_2) \Rightarrow 00010101001 00}$
  - $\boldsymbol{\delta(q_3, 1) = (q_2, 0, R) \Rightarrow \delta(q_3, X_2) = (q_2, X_1, D_2) \Rightarrow 00010010101 00}$
  - $\boldsymbol{\delta(q_3, B) = (q_3, 1, L) \Rightarrow \delta(q_3, X_3) = (q_3, X_2, D_1) \Rightarrow 00010001000100 10}$
- **Code for this TM:**

0100100010100110001010100100110001001001010011000100010001001 0

The Universal Turing Machine

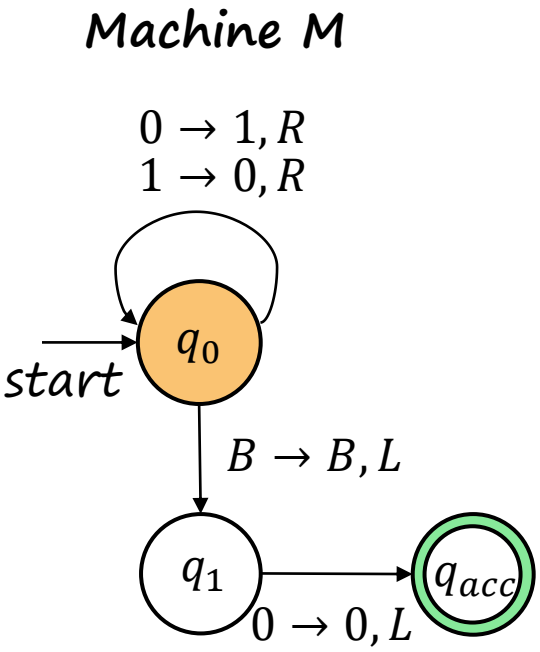


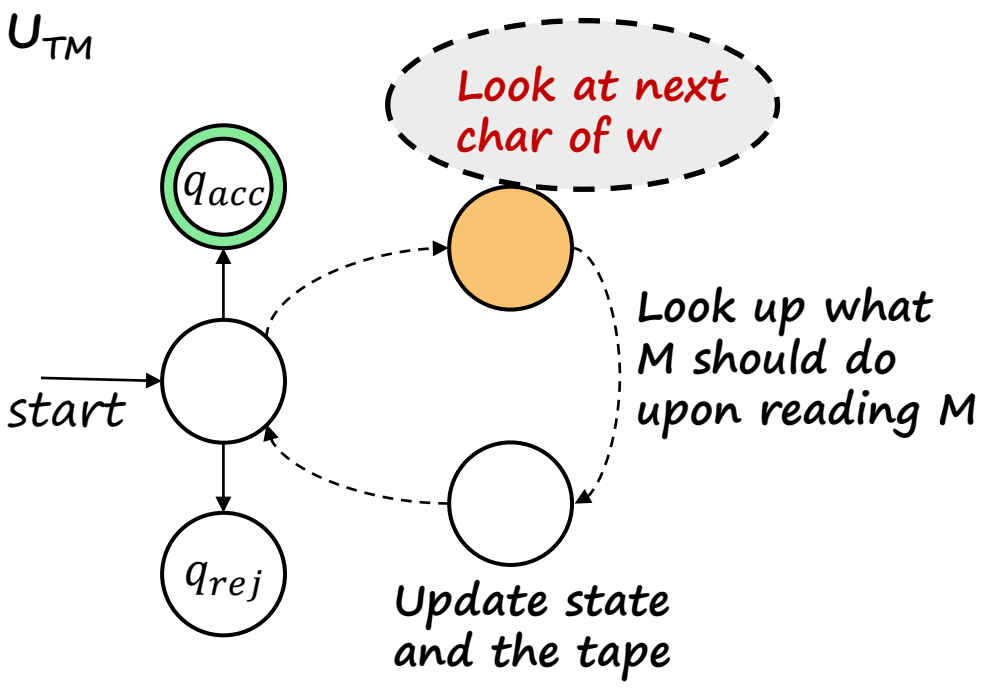


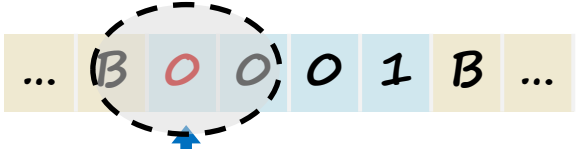
...	B	0	0	0	1	B	...

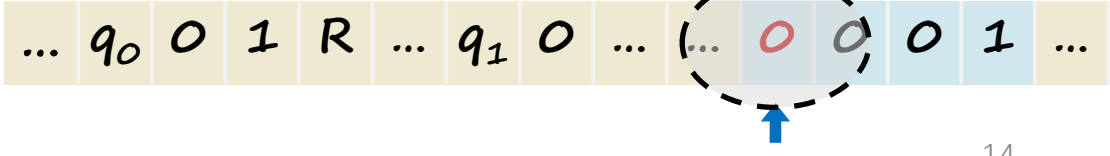
...	q_0	0	1	R	...	q_1	0	...	...	0	0	0	1	...

The Universal Turing Machine

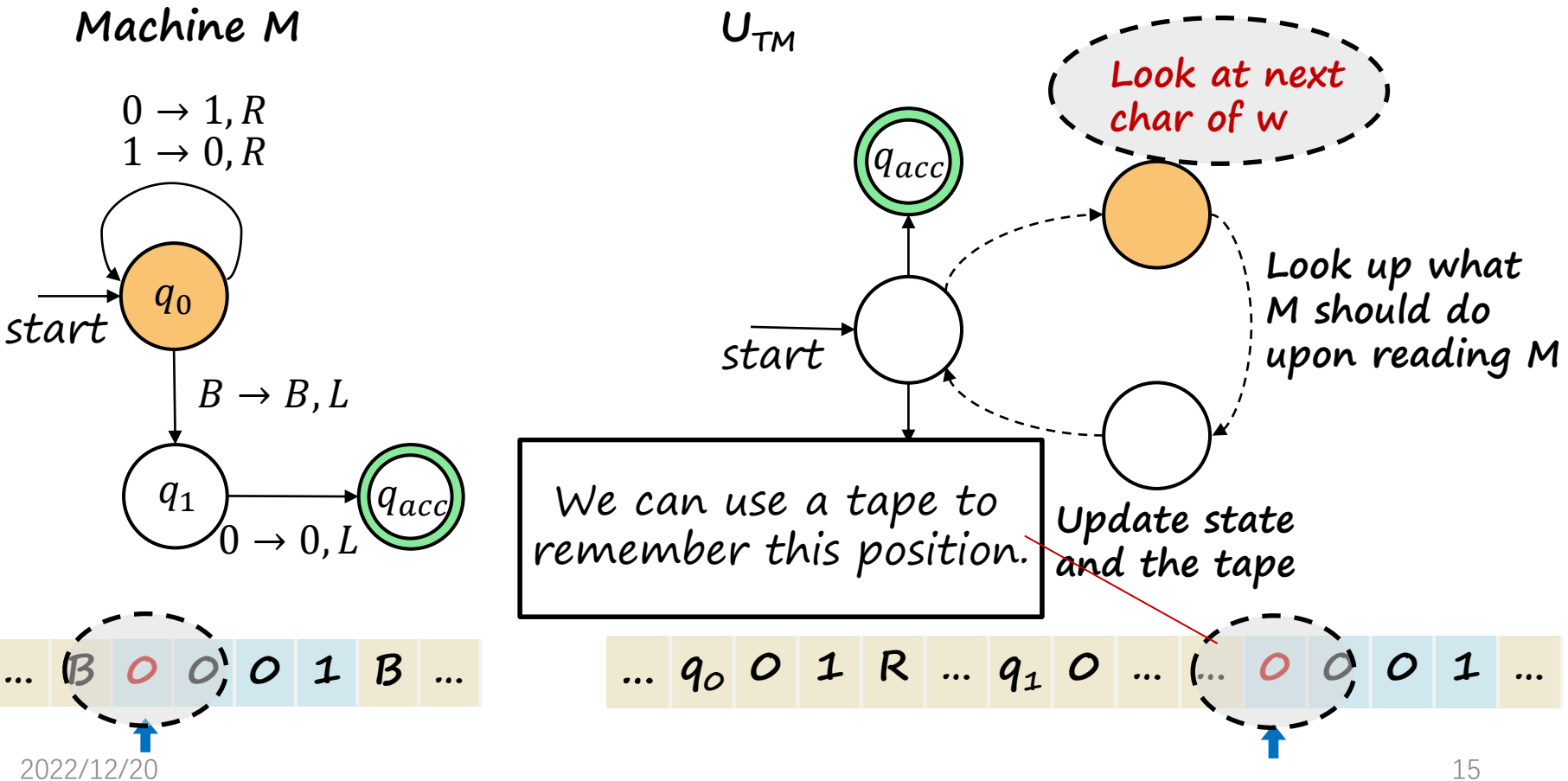




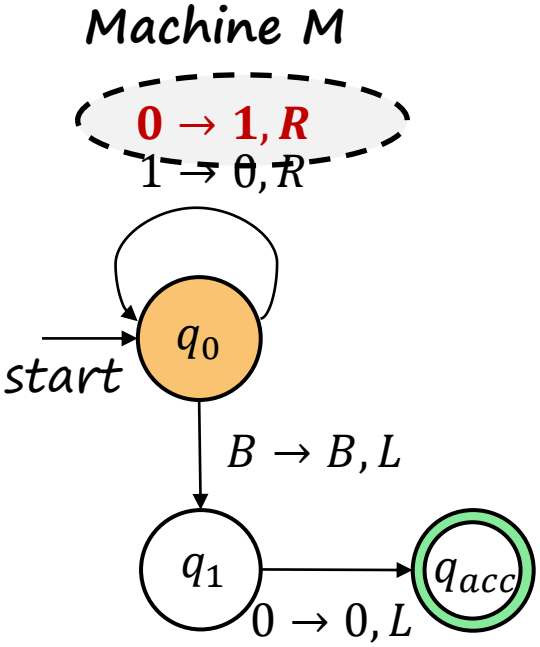


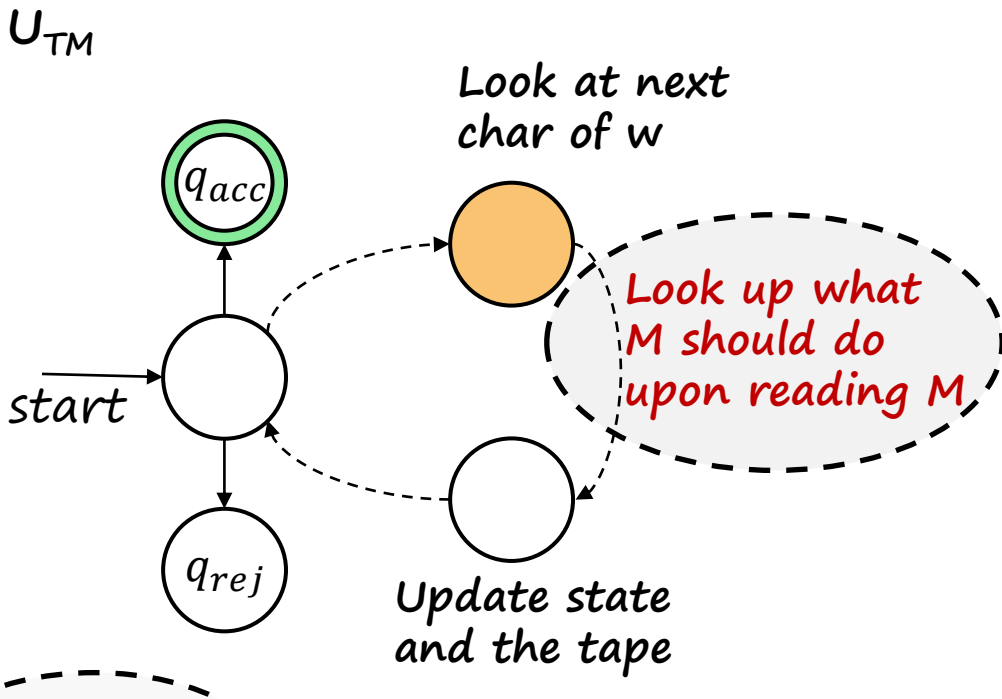


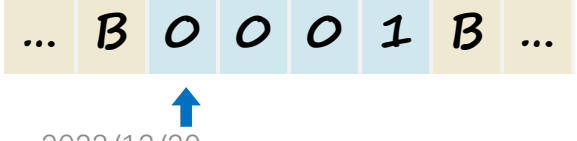
The Universal Turing Machine

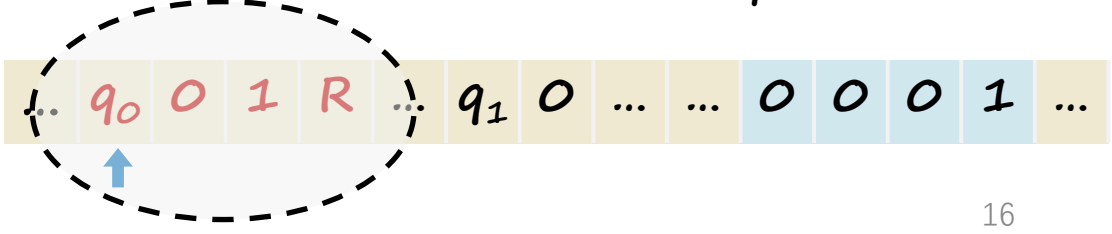


The Universal Turing Machine

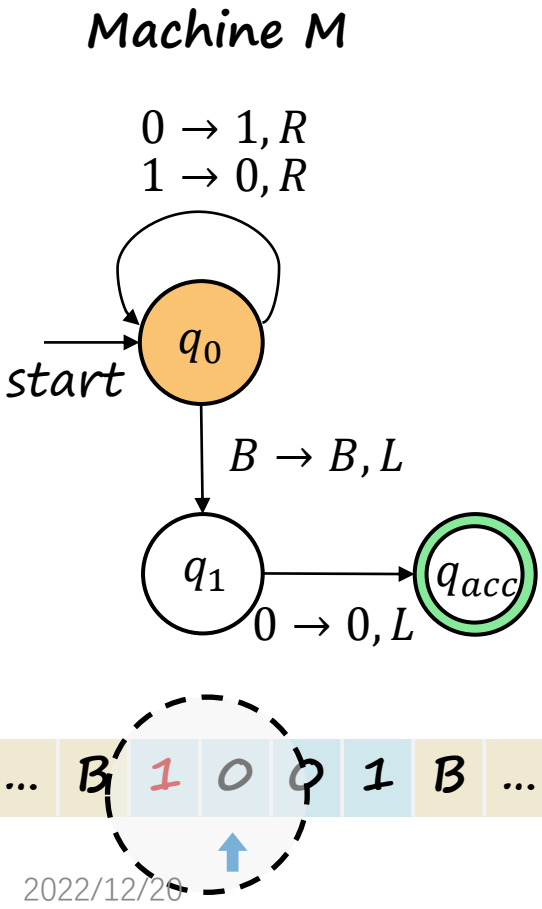


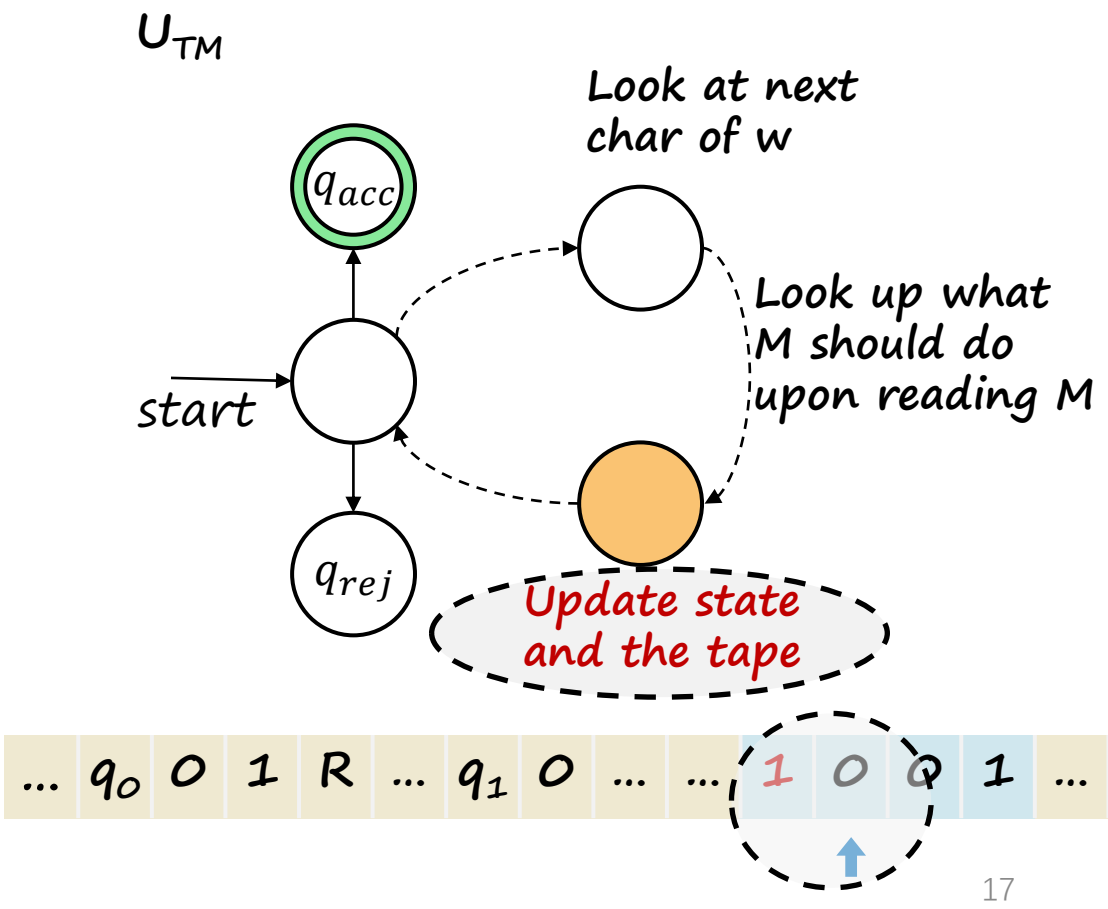




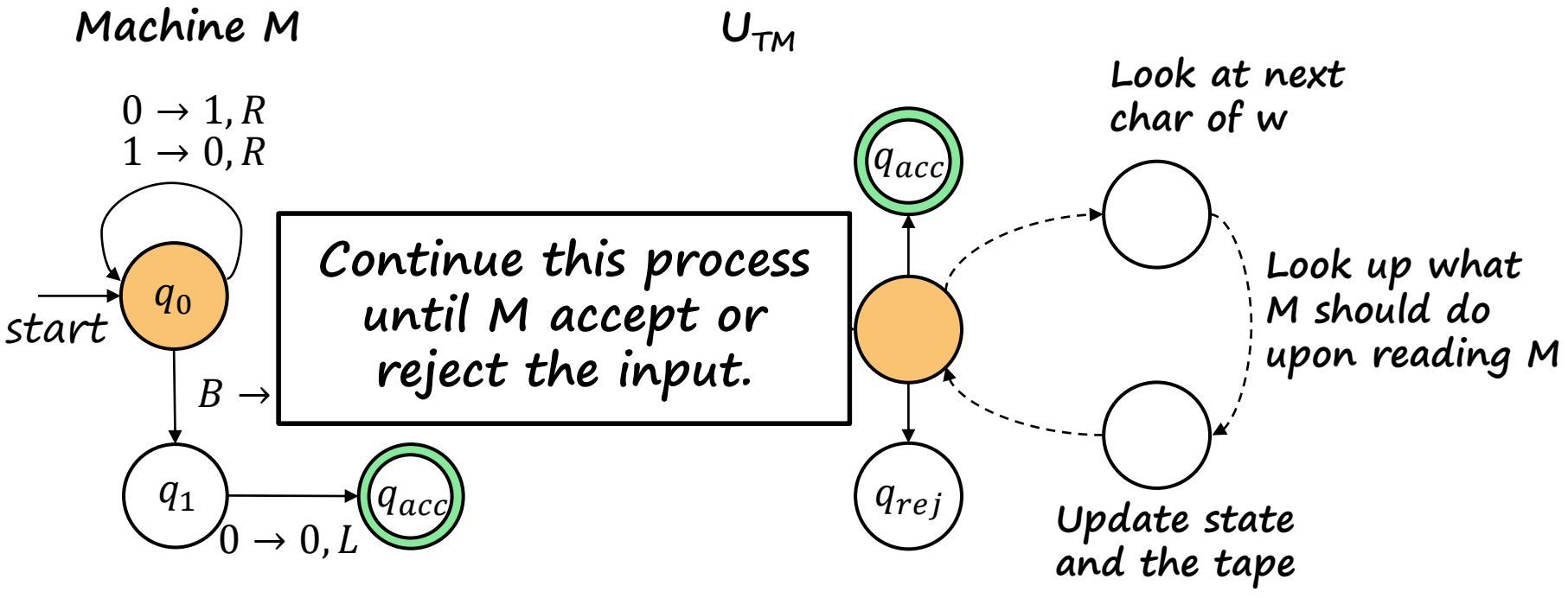


The Universal Turing Machine





The Universal Turing Machine



…	B	1	0	0	1	B	…

…	q_0	0	1	R	…	q_1	0	…	…	1	0	0	1	…

The Language of U_{TM}

- Recall that the language of a TM is the set of all strings that TM accepts.
- $\boldsymbol{U_{TM}}$ when run on a string $\langle \boldsymbol{M}, \boldsymbol{w} \rangle$, where $\boldsymbol{M}$ is a TM and $\boldsymbol{w}$ is a string, will
  - Accept $\langle M, w \rangle$ if M accepts w
  - Reject $\langle M, w \rangle$ if M rejects w
  - Loop on $\langle M, w \rangle$ if M loops on w.

The Language of U_{TM}

- The universal language, donated $\boldsymbol{L_u}$, is the language of the U_{TM}

$$L_u = L(U_{TM}) = \{\langle M, w\rangle | M \text{ is a TM and } M \text{ accepts } w\}$$

$$= \{\langle M, w\rangle | M \text{ is a TM and } w \in L(M)\}$$

- Useful fact:

$$\boldsymbol{\langle M, w\rangle \in L_u \Leftrightarrow M \text{ accepts } w}$$

- Because $L_u = L(U_{TM})$, we know that $L_u \in RE$

The Language of U_{TM}

- If M accepts w, then we have:
  - U_{TM} accepts $\langle M, w \rangle$
  - U_{TM} accepts $\langle U_{TM}, \langle M, w \rangle \rangle$
  - U_{TM} accepts $\left\langle U_{TM}, \langle U_{TM}, \langle M, w \rangle \rangle \right\rangle$
  - ….

How we investigate computability?

2.1 Discussion on Problems

Part1. Intro & Set theory(I) : Basics & Formal Language.

Part2. Set Theory(II): Axiom system & Cardinality.

Part3. Capture Structures: Binary Relation & Function

2.2 Discussion on Computation

Part1. Turing Machine Basics.

Part2. Variants of Turing Machine. Church-Turing Thesis.

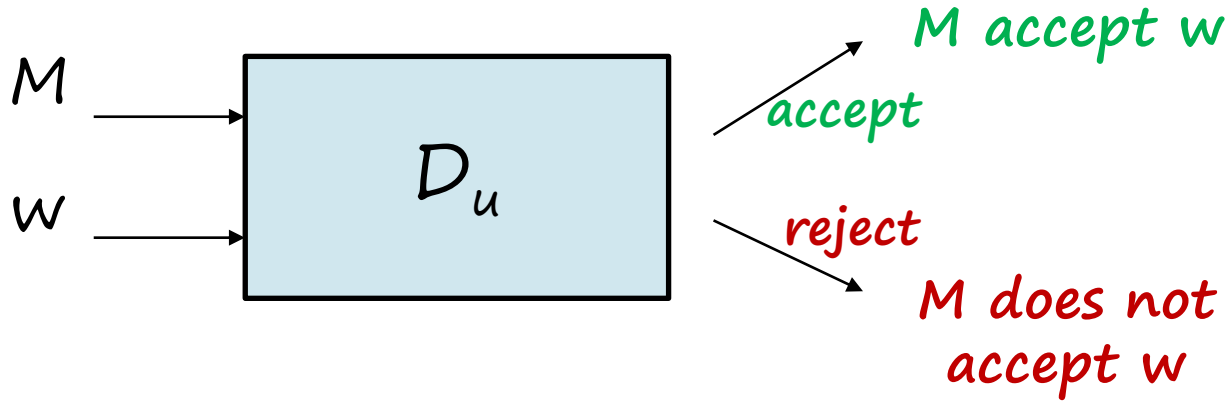
2.3 Discussion on computability

Part1. The Language of Turing Machine. R & RE

Part2. Undecidability.

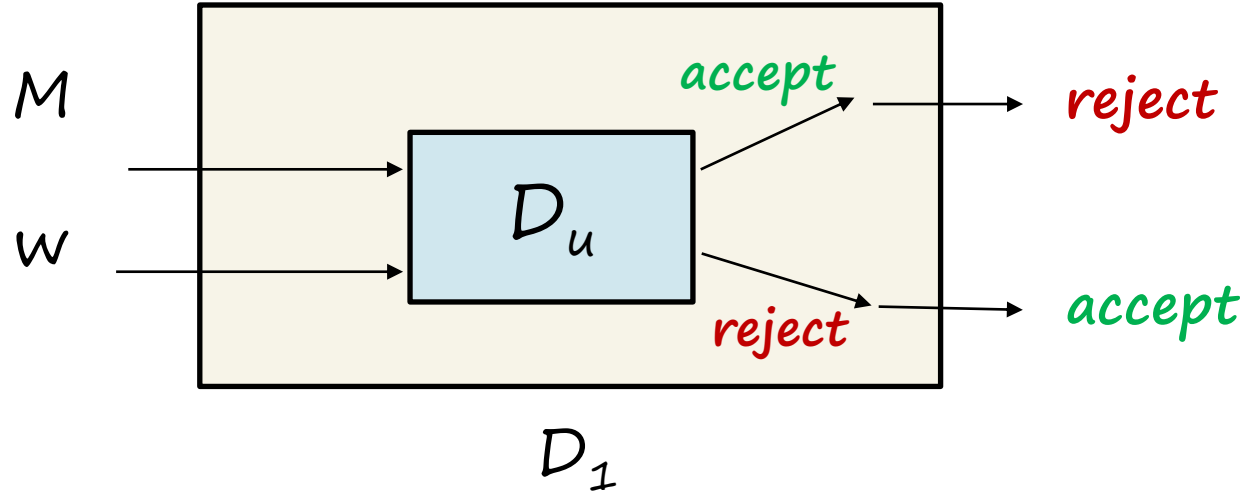
A Decider for L_u

- We know that $L_u \in \mathbf{RE}$, but whether $L_u \in \mathbf{R}$?
- That is, whether there is a decider D_u for L_u?



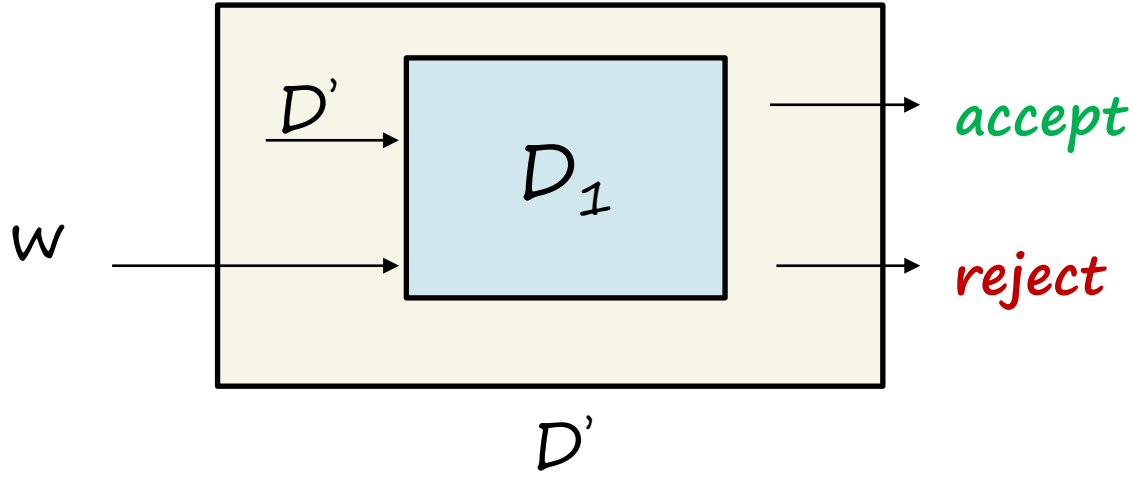
A Decider for L_u

- If D_u exists, we can build a new TM D_1 based on D_u
- D_1 takes a TM M and string w as inputs, and feeds them to D_u, then outputs the **reverse result** of D_u.



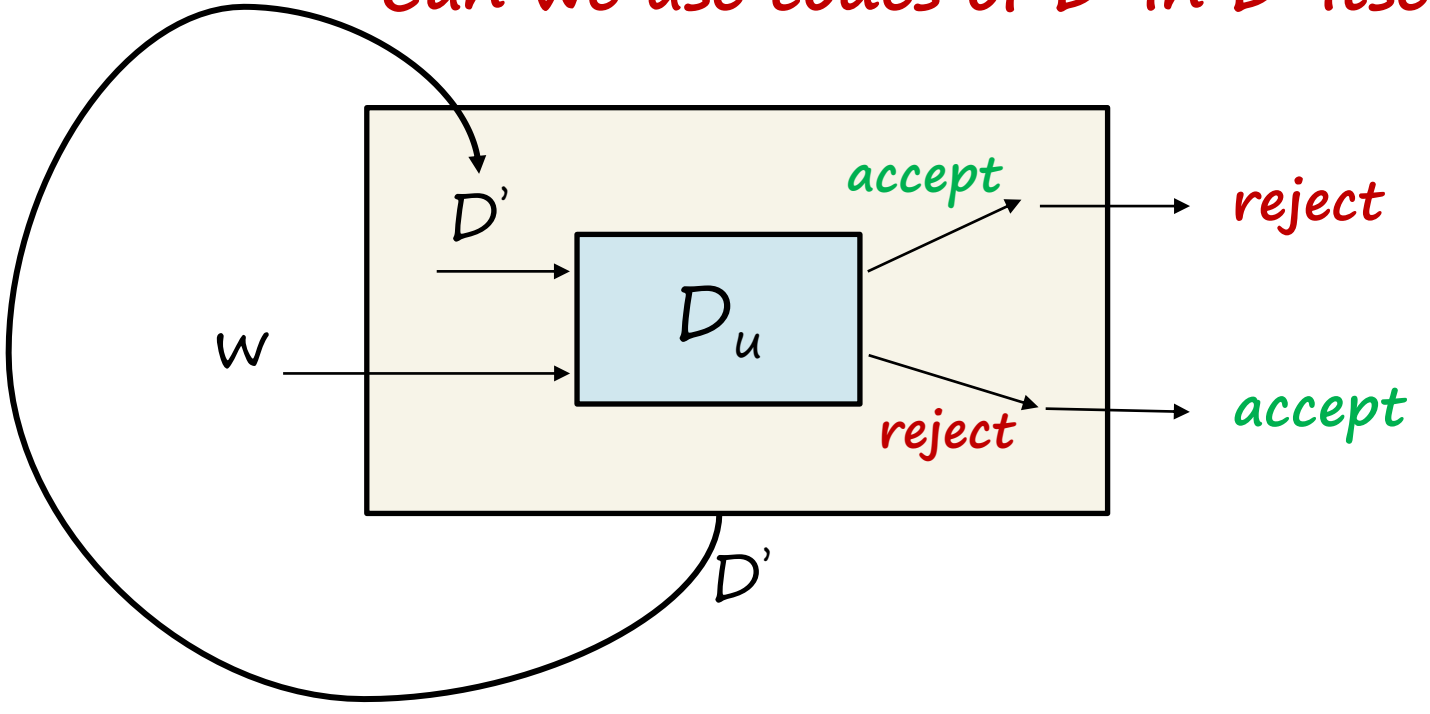
A Decider for L_u

- Then we can build another TM D' based on D_1
- D' only takes a string w as input, and feeds **its own code** and w to D_1, then outputs the result of D_1.

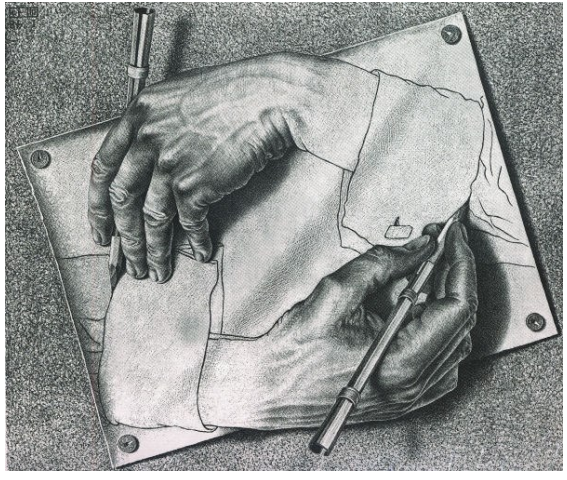


A Decider for L_u

Can we use codes of D' in D' itself?



Self-Reference



Sets that do not contain itself.

"This sentence is wrong."

Self-Referential Software

- A **Quine** is a program that, when running, prints its own source code.
- Quines aren't allowed to just read the file containing their source code and print it out; that's cheating (and technically incorrect if someone changes that file!)
- How would you write such a program?

Example: Quine

```
#include <stdio.h>
char*f="#include <stdio.h>
char*f=%c%s%c;
main(){printf(f,34,f,34,10);}%c";
main(){printf(f,34,f,34,10);}
```

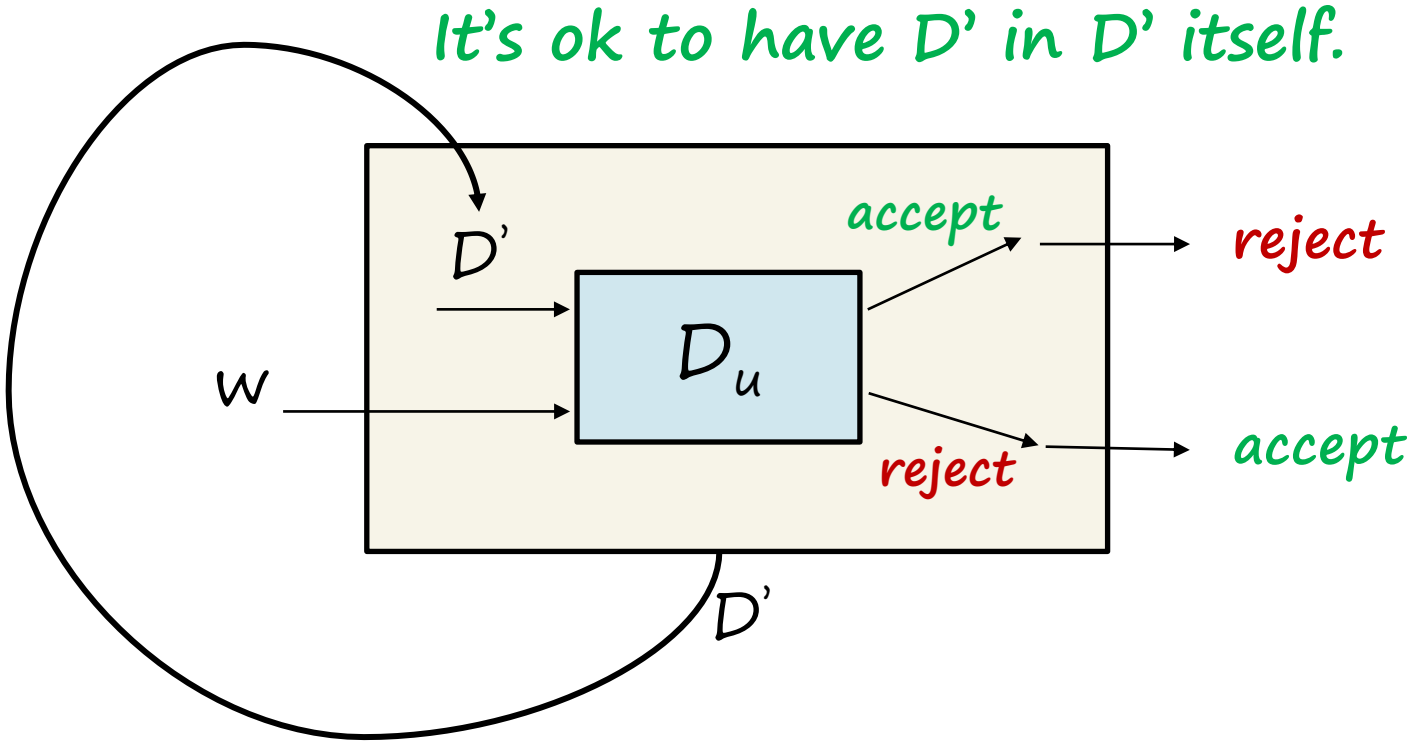
http://www.nyx.net/~gthompso/quine.htm

Self-Referential Software

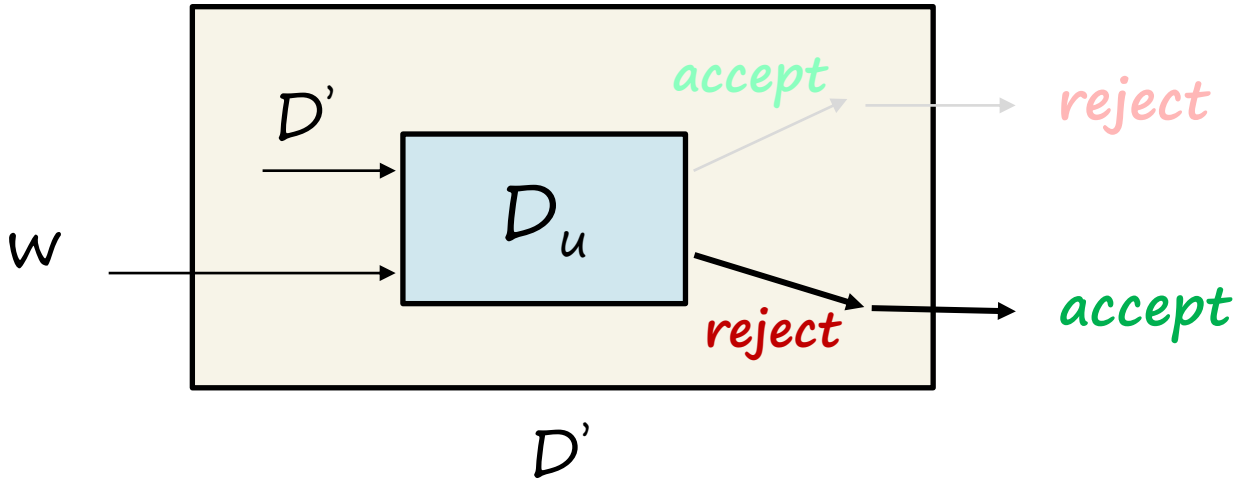
- Going forward, assume that any program can be augmented to include a method called *mySource()* that returns a string representation of its source code.
- Then we can take the whole program codes as an input though we don't know what the whole program codes are, and even what we write now will change the final code itself.

```
char * mySource(){…}
int main(){
  char * s = mySource();
  ......
}
```

A Decider for L_u

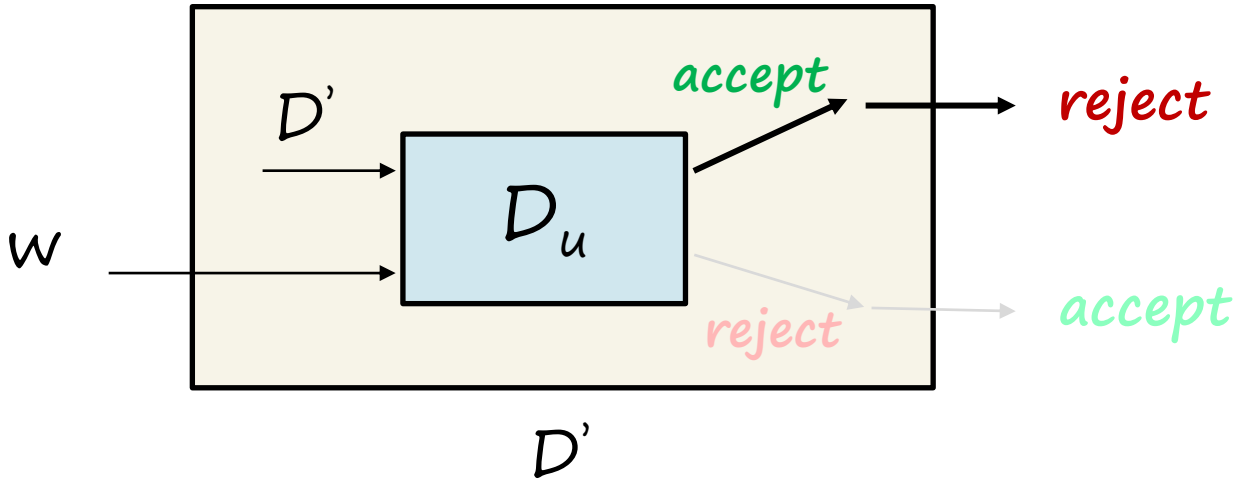


A Decider for L_u



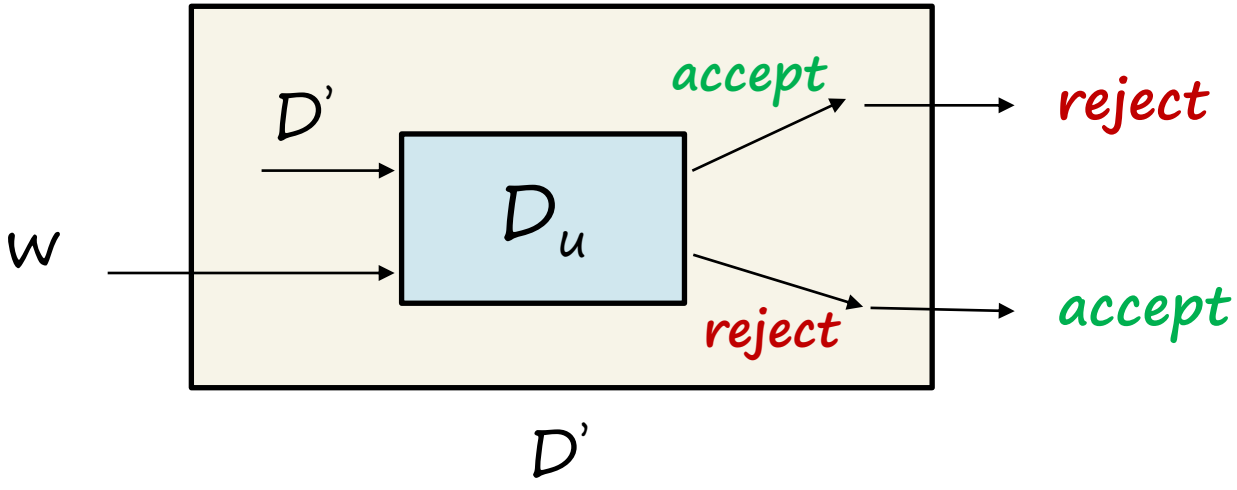
- If D' accepts w.
  - Then D_u rejects $\langle D', w \rangle$, which means that D' rejects w

A Decider for L_u



- If D' accepts w.
  - Then D_u rejects $\langle D', w \rangle$, which means that D' rejects w
- If D' rejects w.
  - Then D_u accepts $\langle D', w \rangle$, which means that D' accepts w

A Decider for L_u



D' accepts $w \Leftrightarrow D'$ rejects w

D_u can not exist!

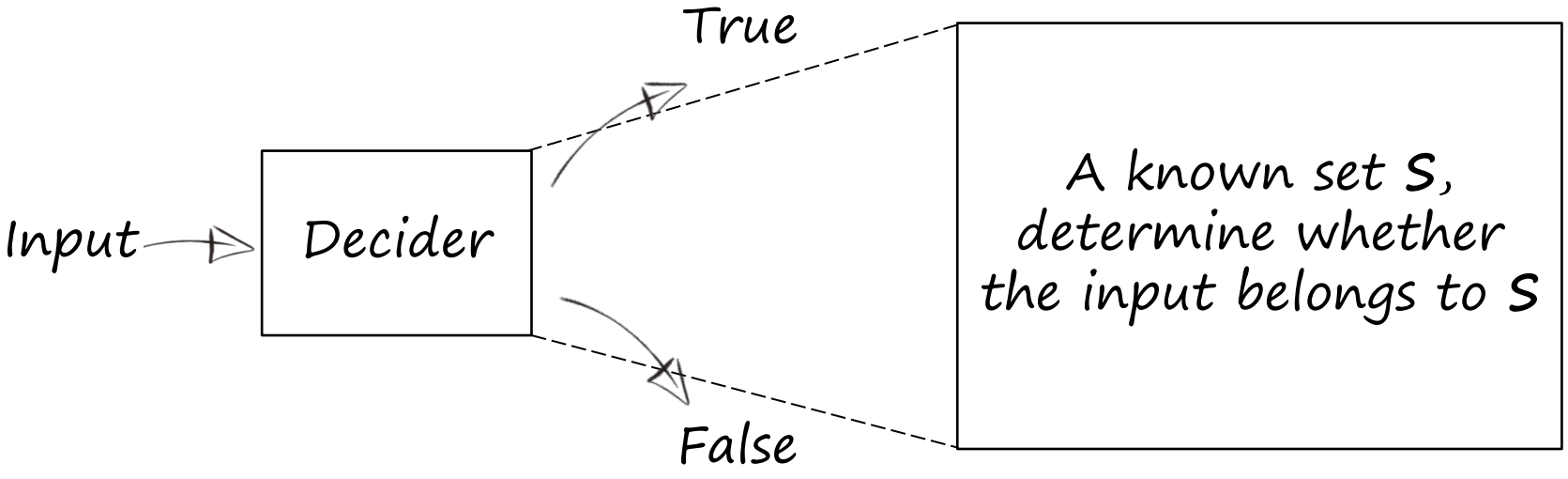
$L_u \notin \mathbf{R}$

- As there are no deciders for L_u, L_u is not in the class of **R.**
- **Undecidable(不可判定):** there is **no algorithm** that can determine whether a program will definitely accept or reject an input.
- Intuition: The only general way to find out what a program will do is to run it.

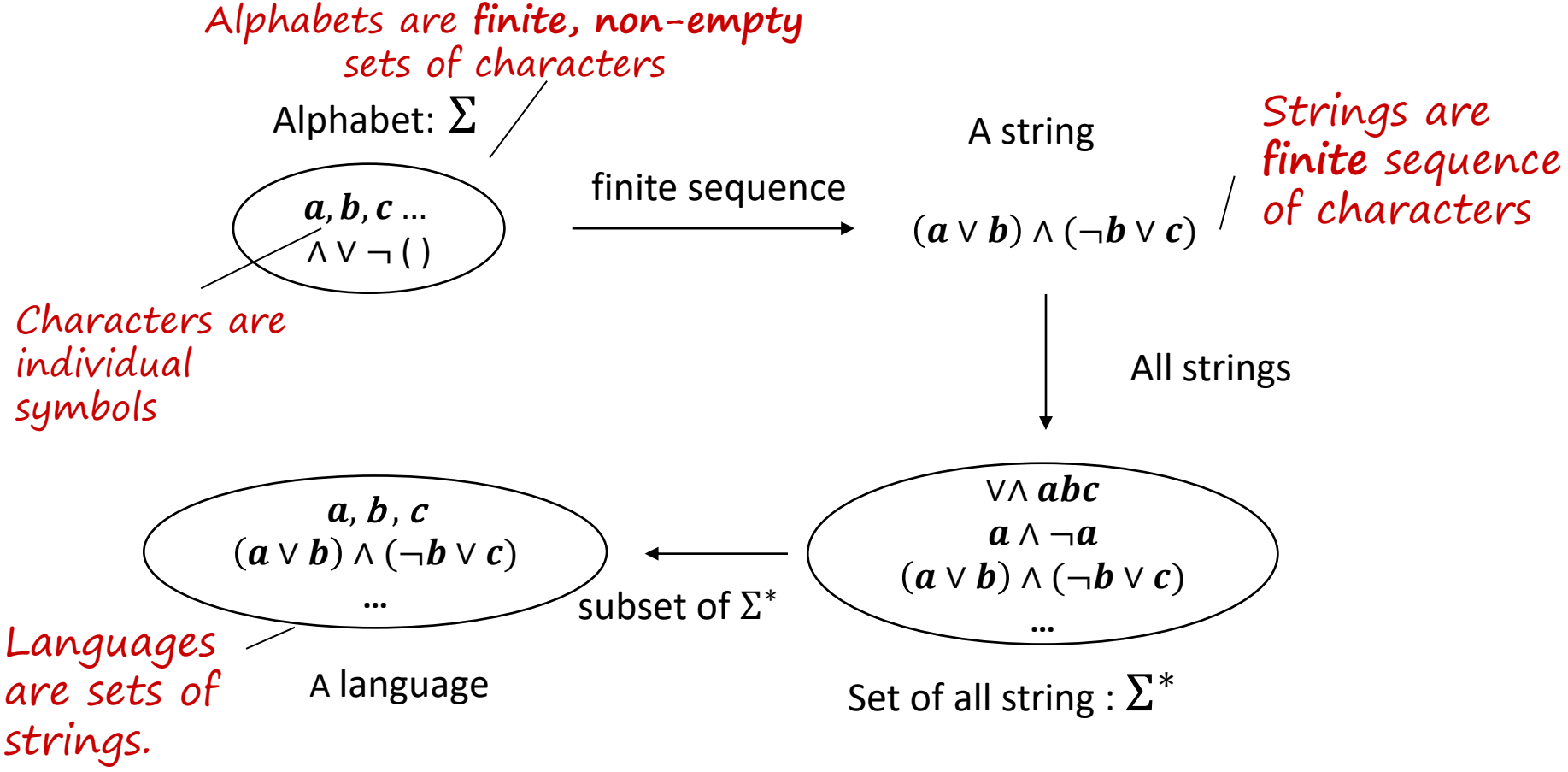
Computable（可计算的）

- **A computational problem is computable if there is a procedure (or algorithm) for it which:**
  - Rigorously follows some finite instructions, requires no ingenuity.
  - Always finishes (terminates) after a finite number of steps.

Solve Decision Problem



Formal Language Theory

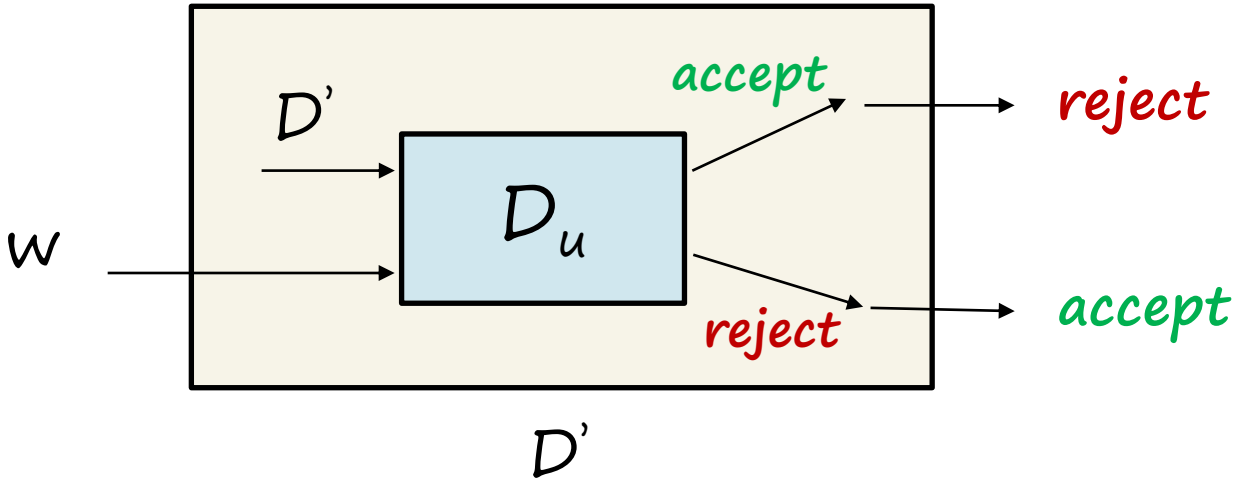


What problems can a computer solve?

What languages can a computer recognize?

What languages can TM recognize?

An Undecidable Problem

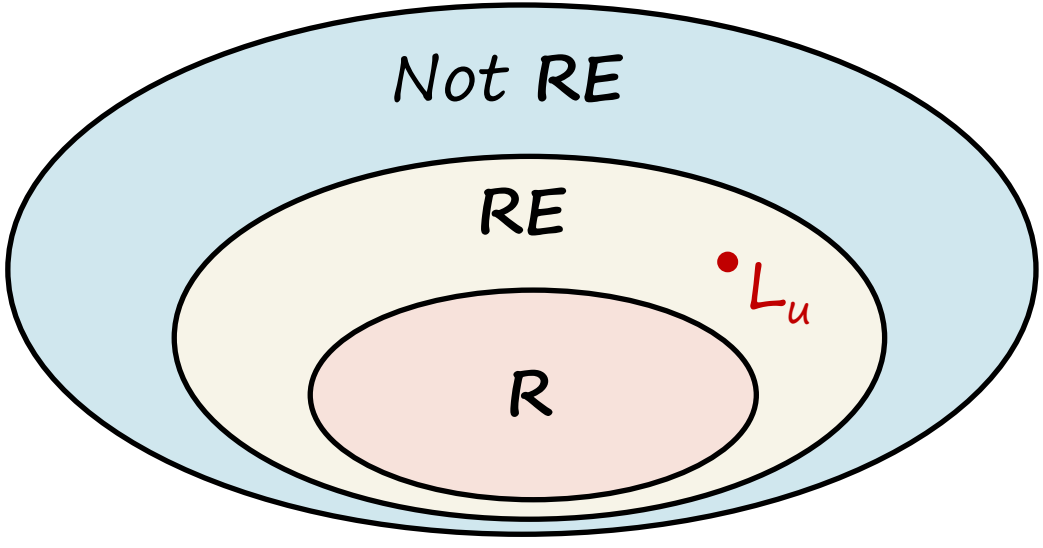


$\boldsymbol{D'}$ accepts $\boldsymbol{w} \Leftrightarrow \boldsymbol{D'}$ rejects $\boldsymbol{w}$

D_u can not exist!

$\mathbf{R} \neq \mathbf{RE}$

- We found a language $L_u \in \mathbf{RE}$, but $L_u \notin \mathbf{R}$, which means $\mathbf{R} \neq \mathbf{RE}$
- Because R ≠ RE, there is a difference between decidability and recognizability:

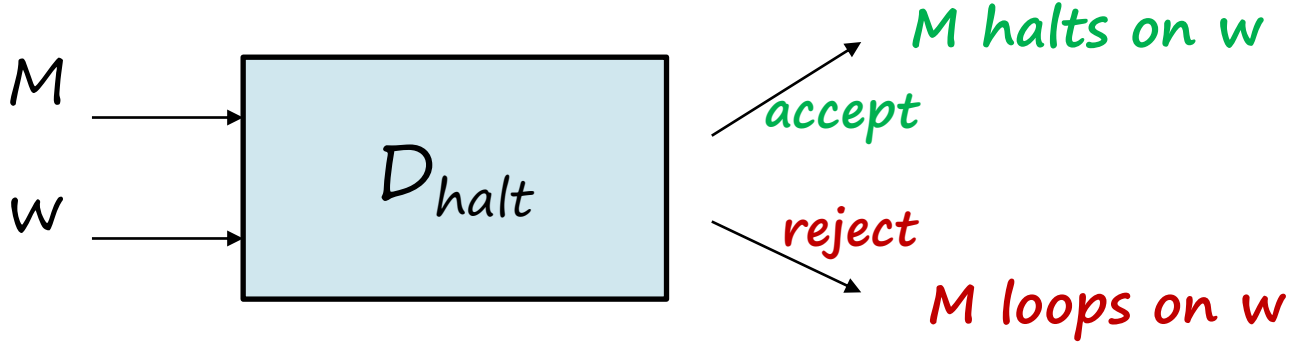


Halting Problem:

Determine whether a program will halt.

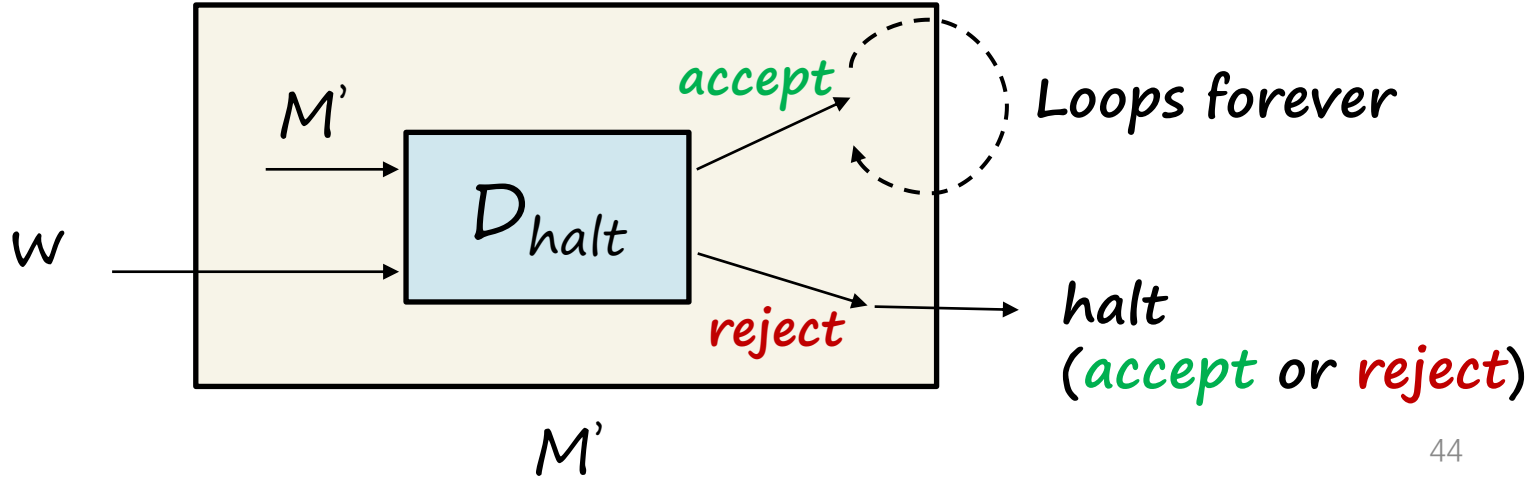
Turing Halting Problem

- Does there exist a TM to **decide** whether a program will finally halt(reject or accept) on a certain input?
- Or whether $L_{halt} = \{\langle M, w\rangle | M \ halts \ on \ w\}$ is in $\mathbf{R}$?

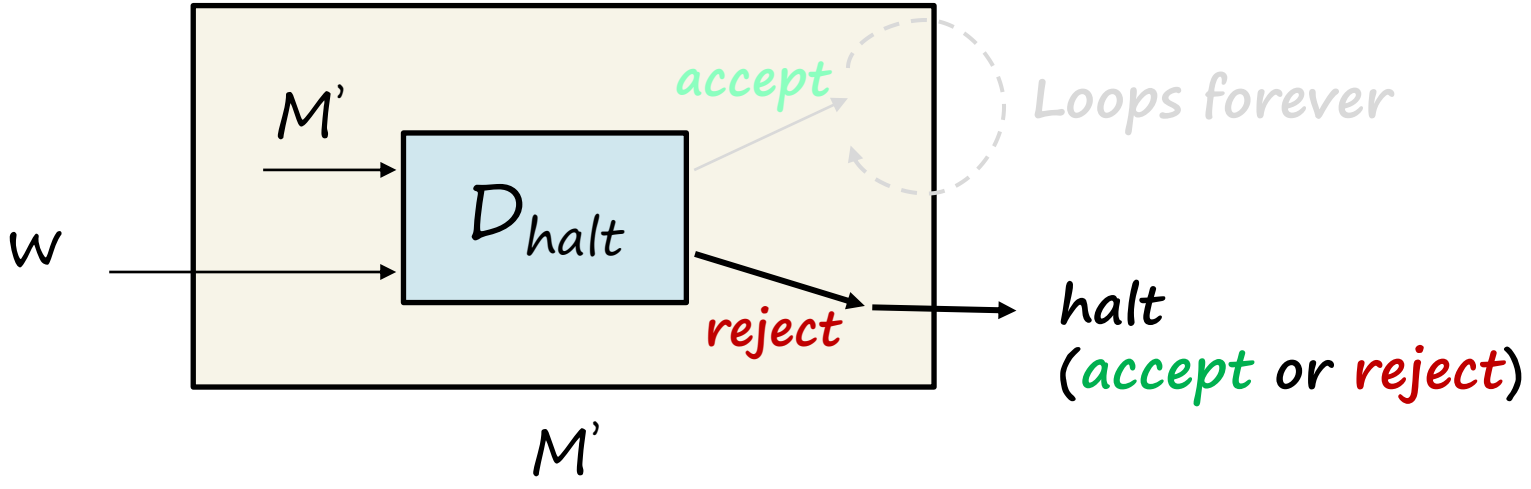


Turing Halting Problem

- Similarly we can build a new TM M' based on D_{halt}.
- M' takes a string w as input, and feed its own code and w to D_{halt}. If D_{halt} rejects the input, then M' halts(accept or reject) on w, otherwise M' loops forever and never stop.

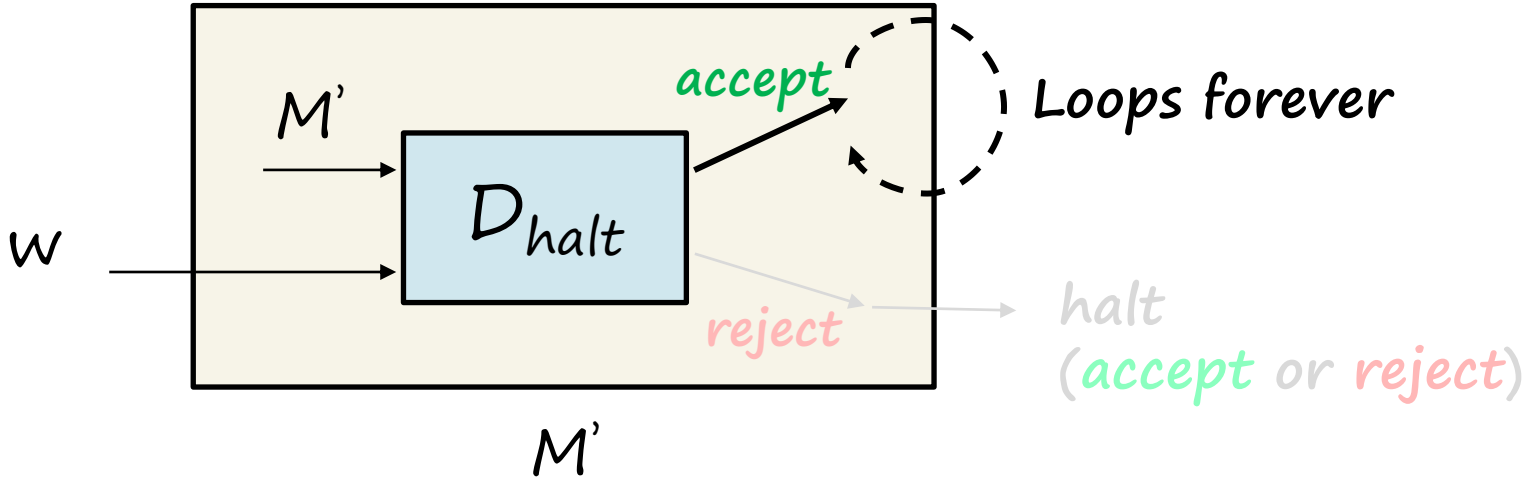


Turing Halting Problem



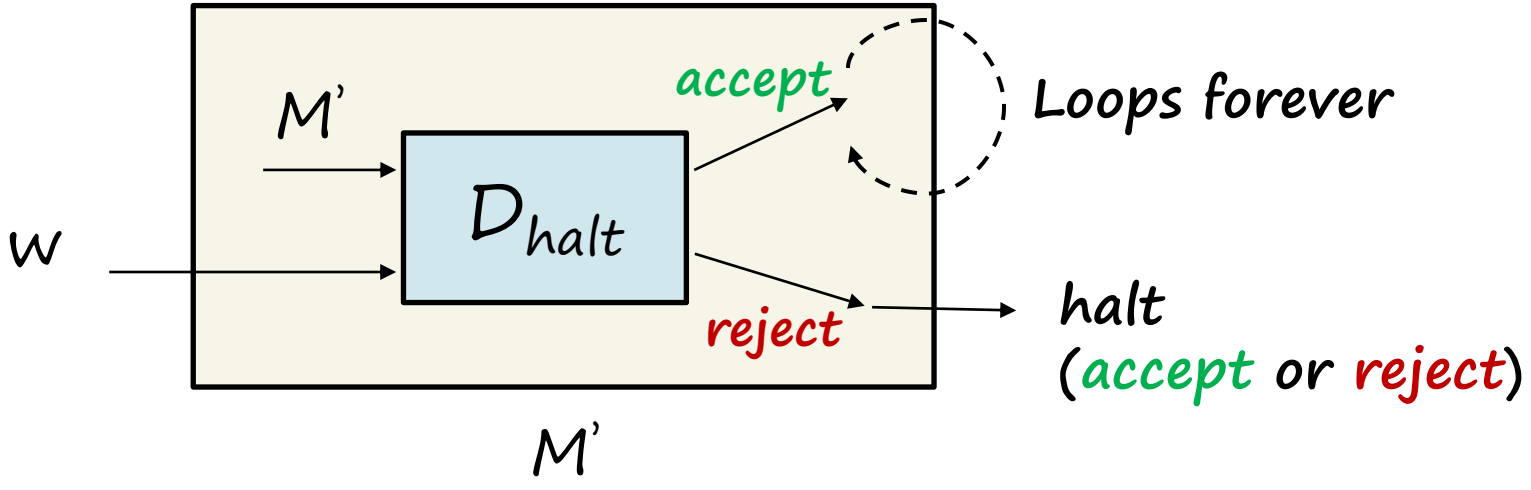
- If M' halts on w
  - Then D_{halt} must reject $\langle M', w \rangle$, which means M' loops on w

Turing Halting Problem



- If M' halts on w
  - Then D_{halt} must reject $\langle M', w \rangle$, which means M' loops on w
- If M' loops on w
  - Then D_{halt} must accept $\langle M', w \rangle$, which means M' halts on w

Turing Halting Problem



M' halts on $w \Leftrightarrow M'$ loops on w

D_{halt} can not exist!

Language that is Non-RE

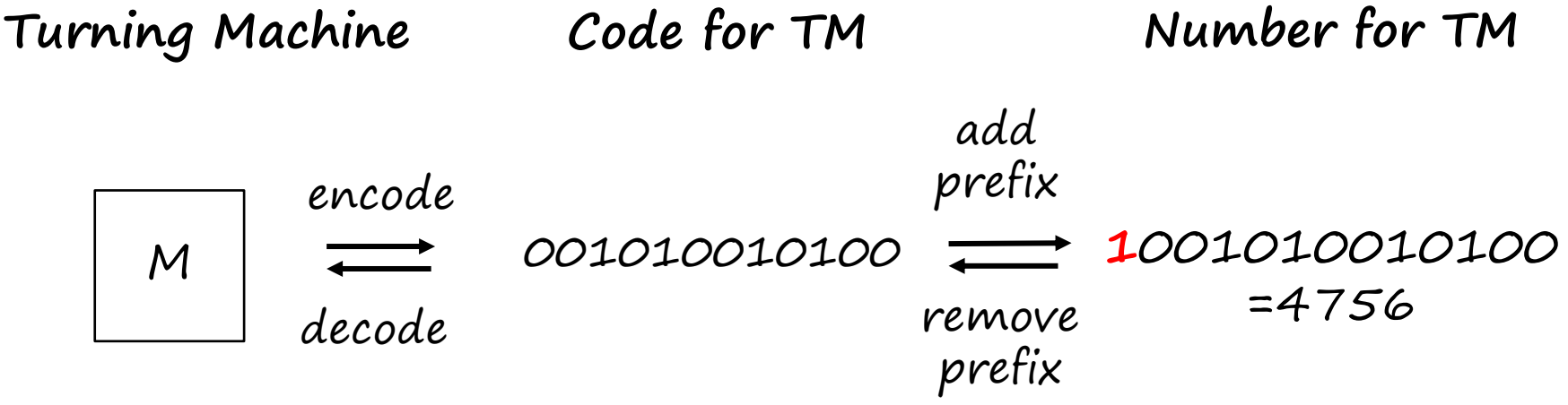
Beyond RE

- The **RE** languages represent languages whose membership can be proven
  - If $w \in L$, we can prove that by a TM
  - So what does a non-RE language look like?
- This language cannot be recognized by any Turing machines

Numbering TMs

- Recall that a TM $\boldsymbol{M}$ can be encoded as a binary. In return, a binary can be interpreted as a TM.
  - Some binary may not be interpreted as a TM (illegal format), we interpret that binary as a TM without any transition function and accepting no string
  - TM code begins with 0's, and we can add a prefix 1 to the code of TM as the number of that TM
    - So the numerical values of different TMs are different.

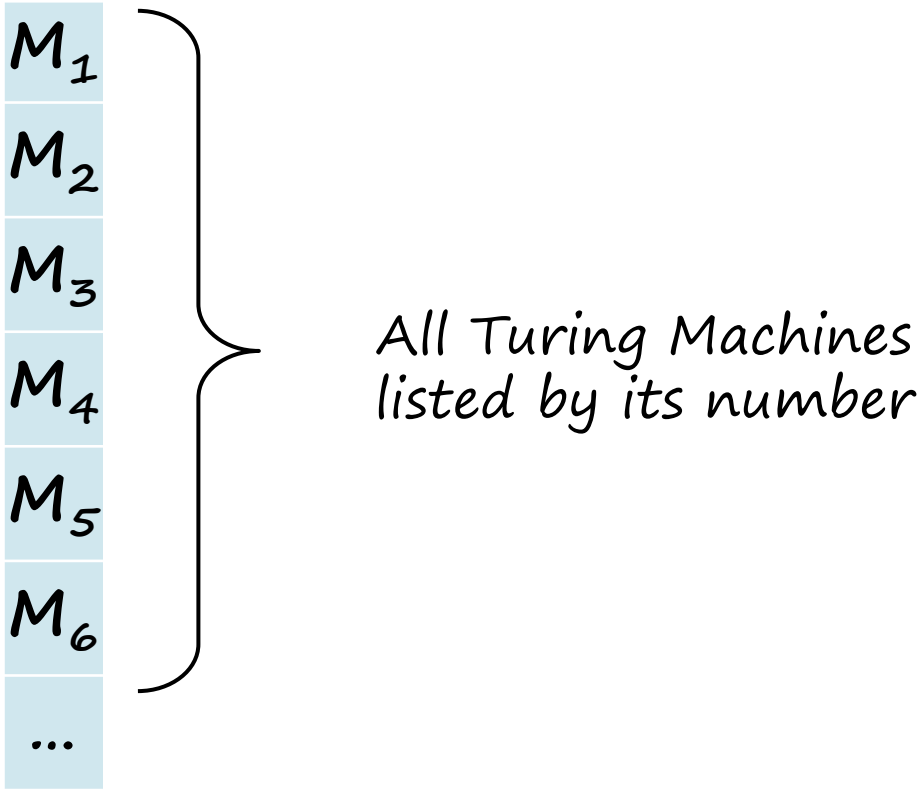
Numbering TMs



M=4756

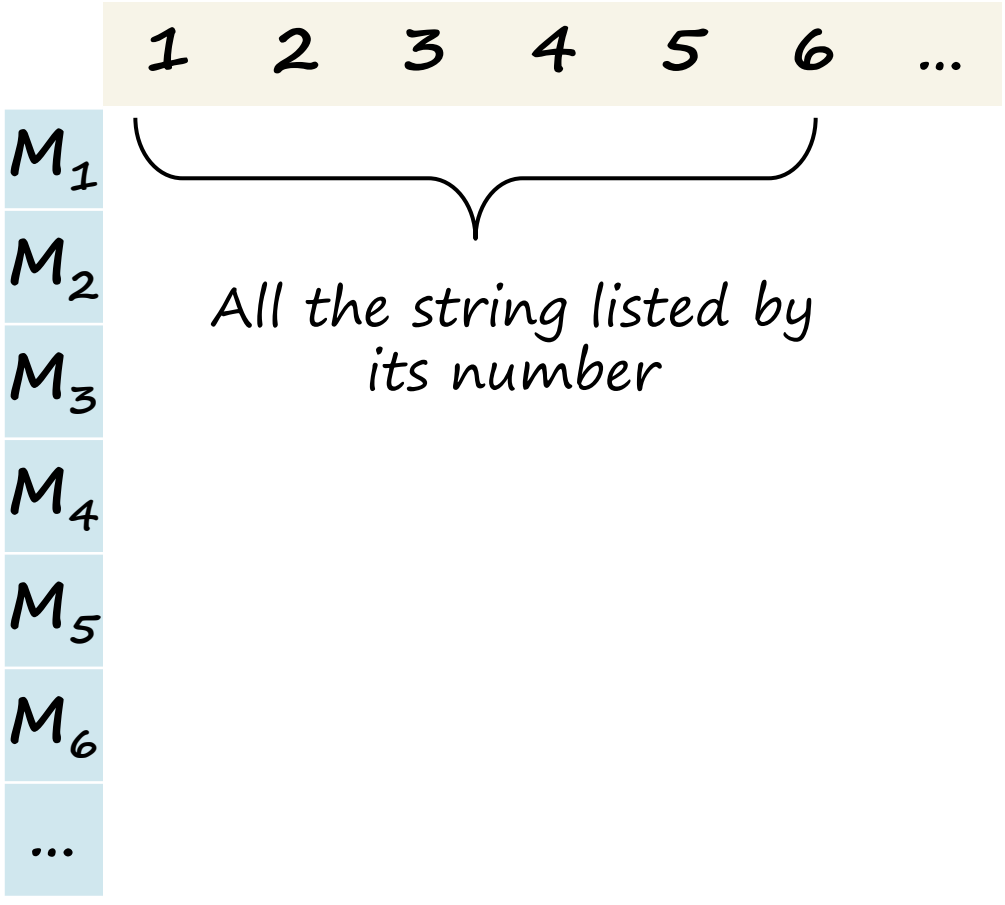
Numbering TMs

- We assign each TM a positive integer, and each positive integer has one TM corresponding to it.
  - We say that the ith TM M_i is the TM whose number is i.
- So we've build a bijection between the set of all the TMs and the set of positive integer. Then we can list all the TMs one by one.


M_1
M_2
M_3
M_4
M_5
M_6
...
All Turing Machines
listed by its number

Numbering All the String

- Each binary string represents a number
- We add a prefix 1 to each binary string to prevent the following condition
  - “01” and “001” are two different binary strings, but they represent the same number (1)
- w_i denotes the string with number i.
  - $w_1 = \epsilon, w_2 = 0, w_3 = 1, w_4 = 00$


1 2 3 4 5 6 ...
M1
M2
M3
M4
M5
M6
...
All the string listed by its number

	1	2	3	4	5	6	…
M_1	0	0	0	1	1	0	…
M_2	0	1	1	0	1	1	…
M_3	0	0	1	1	0	1	…
M_4	1	1	0	1	0	1	…
M_5	1	0	0	0	0	0	…
M_6	0	1	1	0	1	0	…
…	…	…	…	…	…	…	…

1 in row i column j means that M_i accepts w_j.

0 means "not accept".

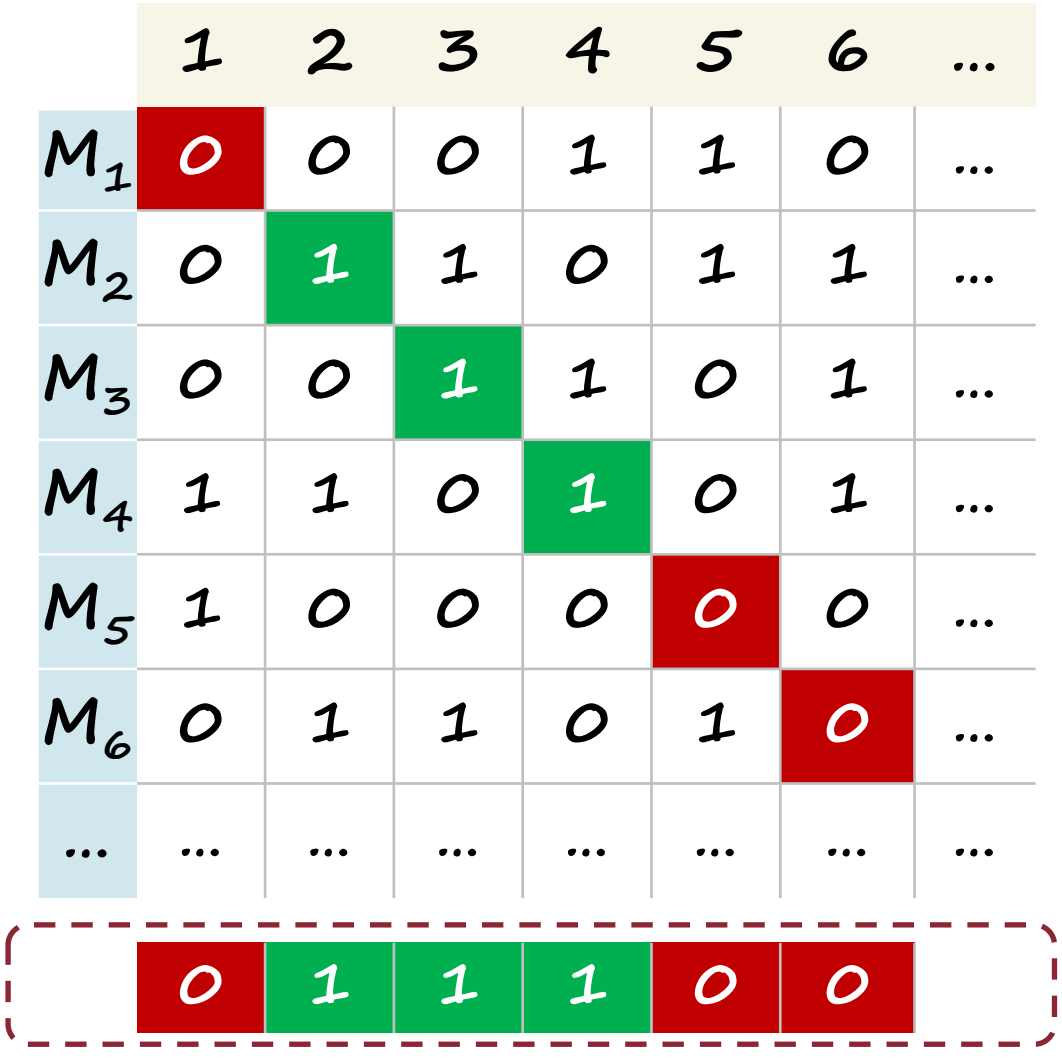
M_3 accepts w_4 (00)

M_6 does not accept w_6(10)

	1	2	3	4	5	6	…
M_1	0	0	0	1	1	0	…
M_2	0	1	1	0	1	1	…
M_3	0	0	1	1	0	1	…
M_4	1	1	0	1	0	1	…
M_5	1	0	0	0	0	0	…
M_6	0	1	1	0	1	0	…
…	…	…	…	…	…	…	…

Row i represents the language of M_i

$L(M_4)=\{w_1,w_2,w_4,w_6\}$


1 2 3 4 5 6 ...
M1 0 0 0 1 1 0 ...
M2 0 1 1 0 1 1 ...
M3 0 0 1 1 0 1 ...
M4 1 1 0 1 0 1 ...
M5 1 0 0 0 0 0 ...
M6 0 1 1 0 1 0 ...
... ...
0 1 1 1 0 0

	1	2	3	4	5	6	…
M_1	0	0	0	1	1	0	…
M_2	0	1	1	0	1	1	…
M_3	0	0	1	1	0	1	…
M_4	1	1	0	1	0	1	…
M_5	1	0	0	0	0	0	…
M_6	0	1	1	0	1	0	…
…	…	…	…	…	…	…	…

1	0	0	0	1	1	

Flip this language.
Swap what's included
and what's excluded

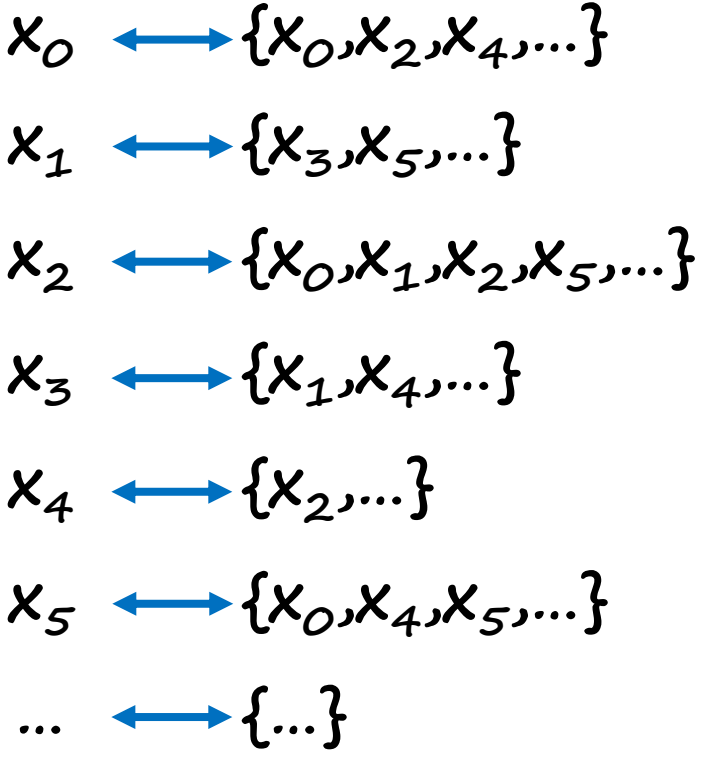
	1	2	3	4	5	6	…
M_1	0	0	0	1	1	0	…
M_2	0	1	1	0	1	1	…
M_3	0	0	1	1	0	1	…
M_4	1	1	0	1	0	1	…
M_5	1	0	0	0	0	0	…
M_6	0	1	1	0	1	0	…
…	…	…	…	…	…	…	…

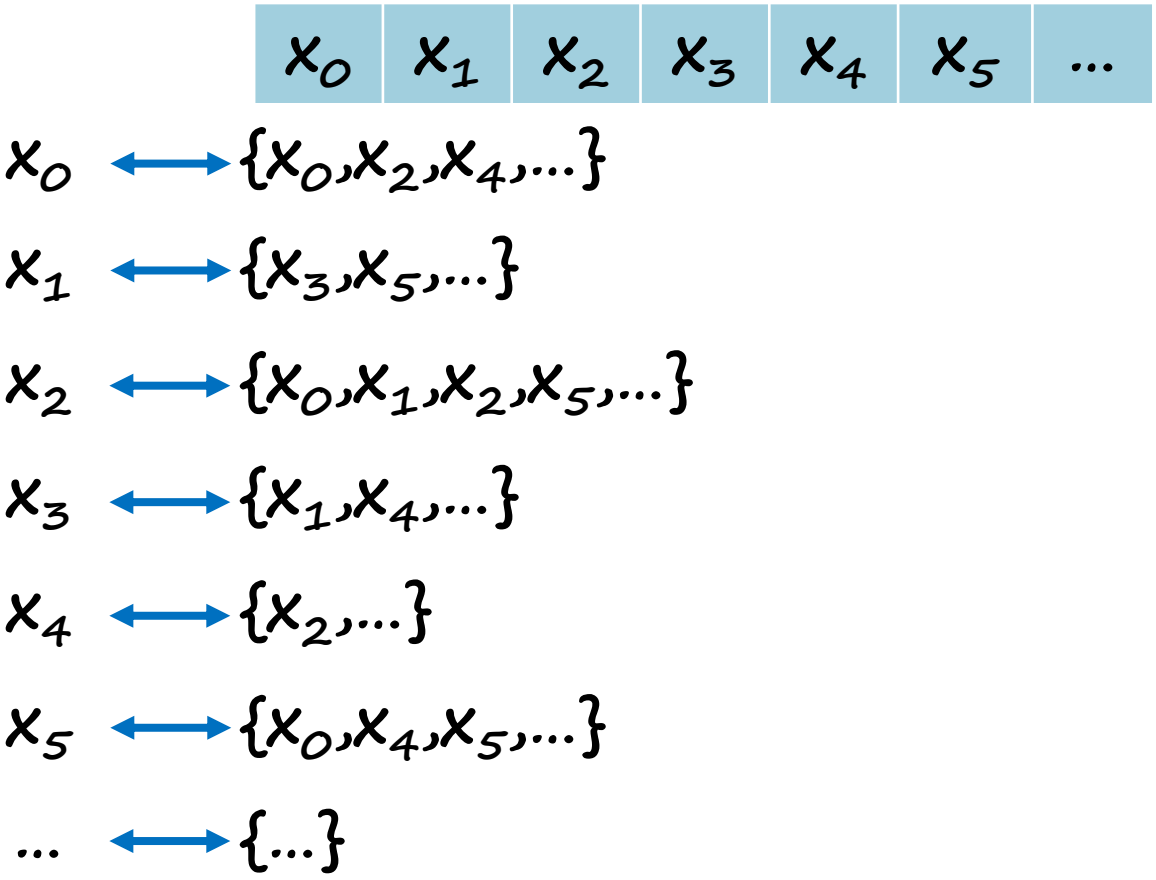
1	0	0	0	1	1

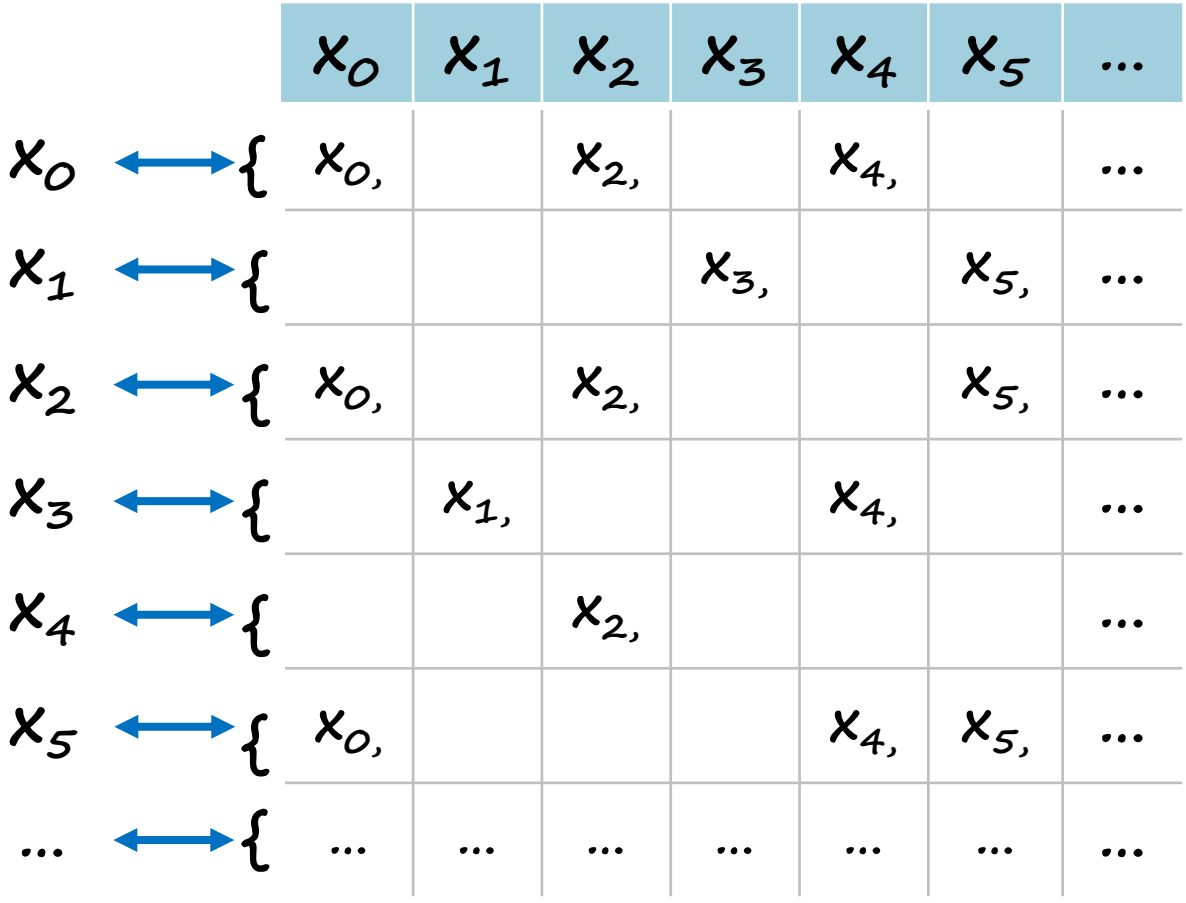
No TM has this behavior!

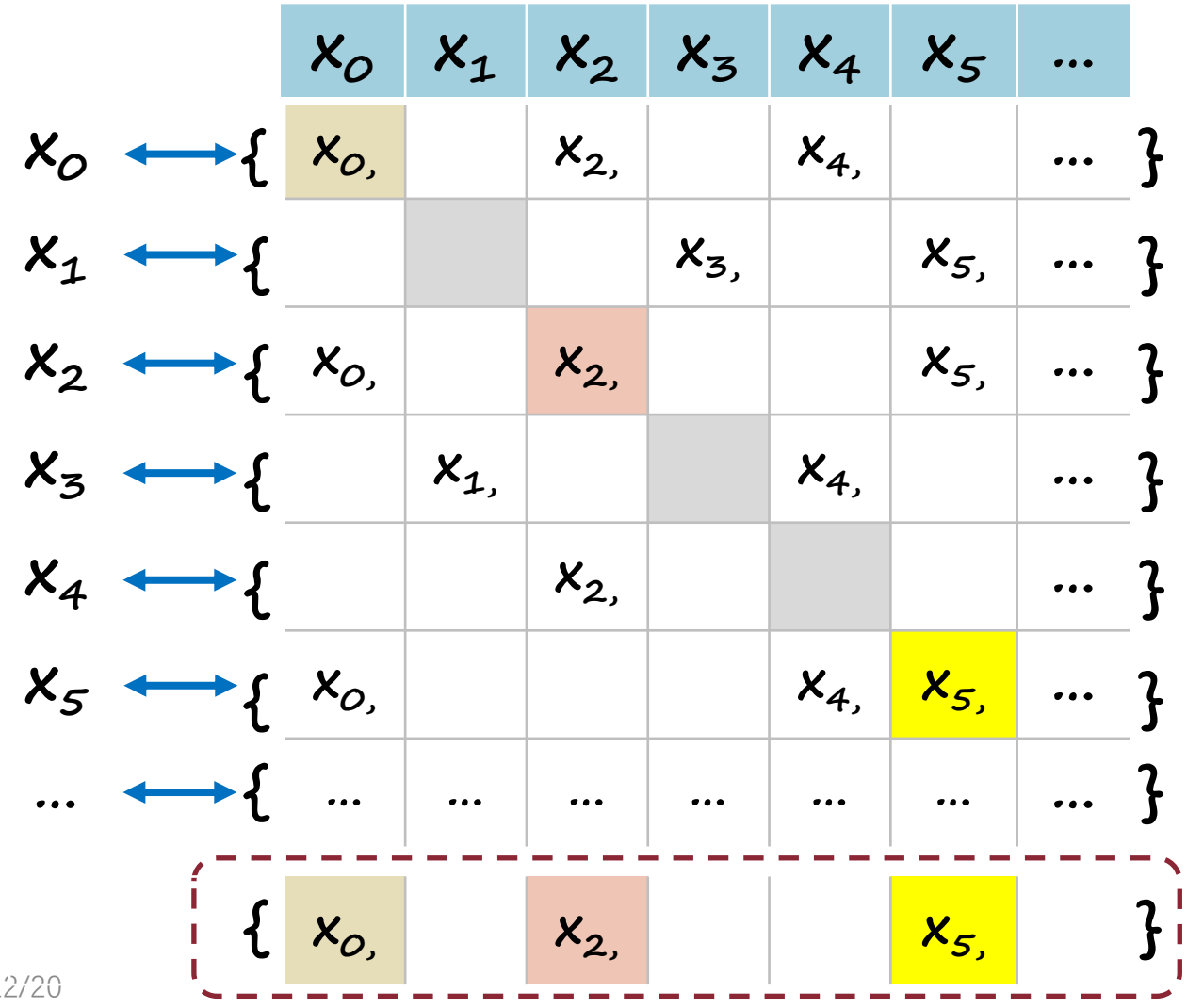
Recall: Cardinality of Power Set

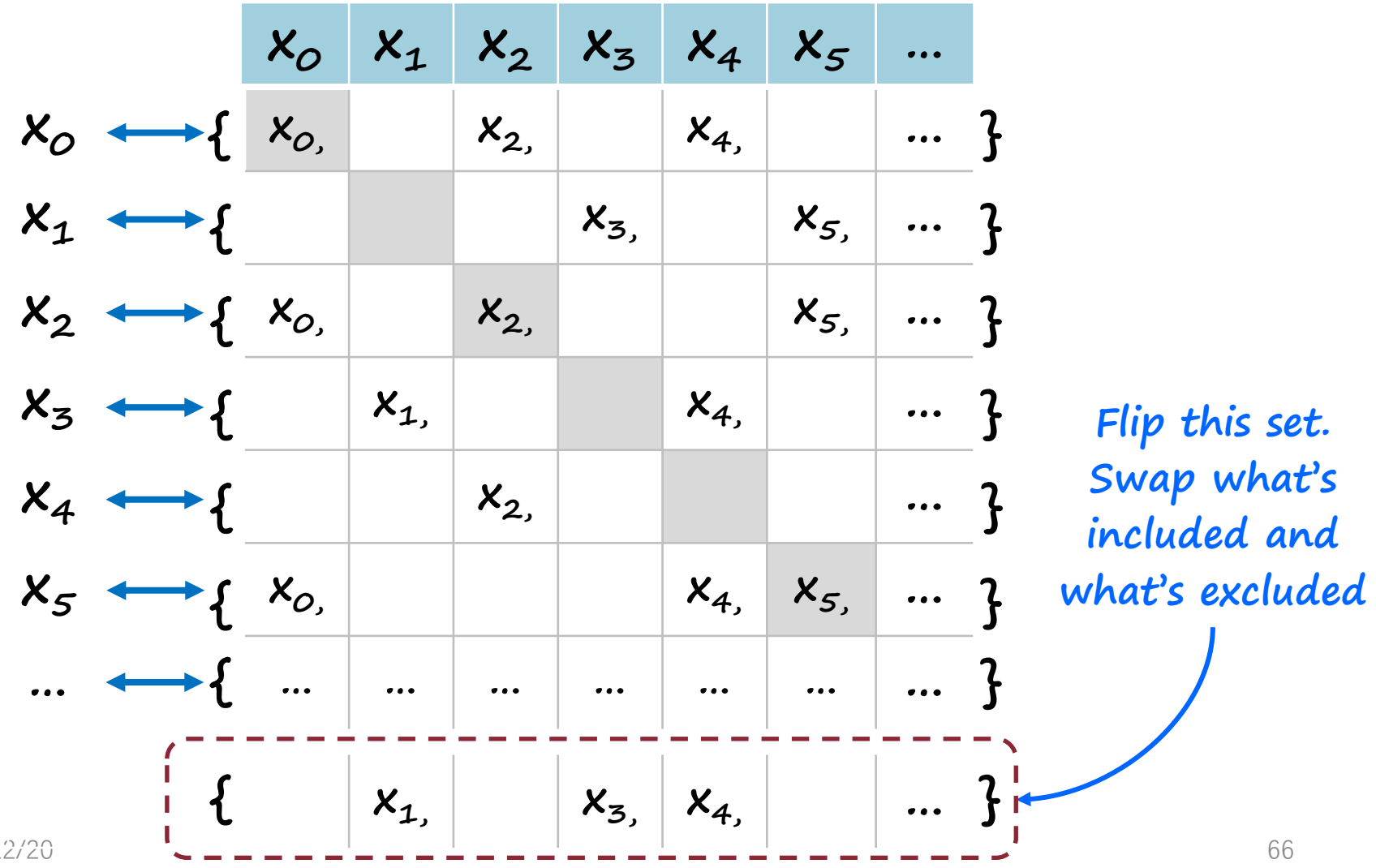
- Does $|\mathcal{P}(S)| > |S|$ for all infinite sets S?
- For an infinite set $S = \{x_0, x_1, x_2, ...\}$, we assume that $\mathcal{P}(S) \approx S$
- It means we can pair up the elements of S and the **subset of S** without leaving anything out.

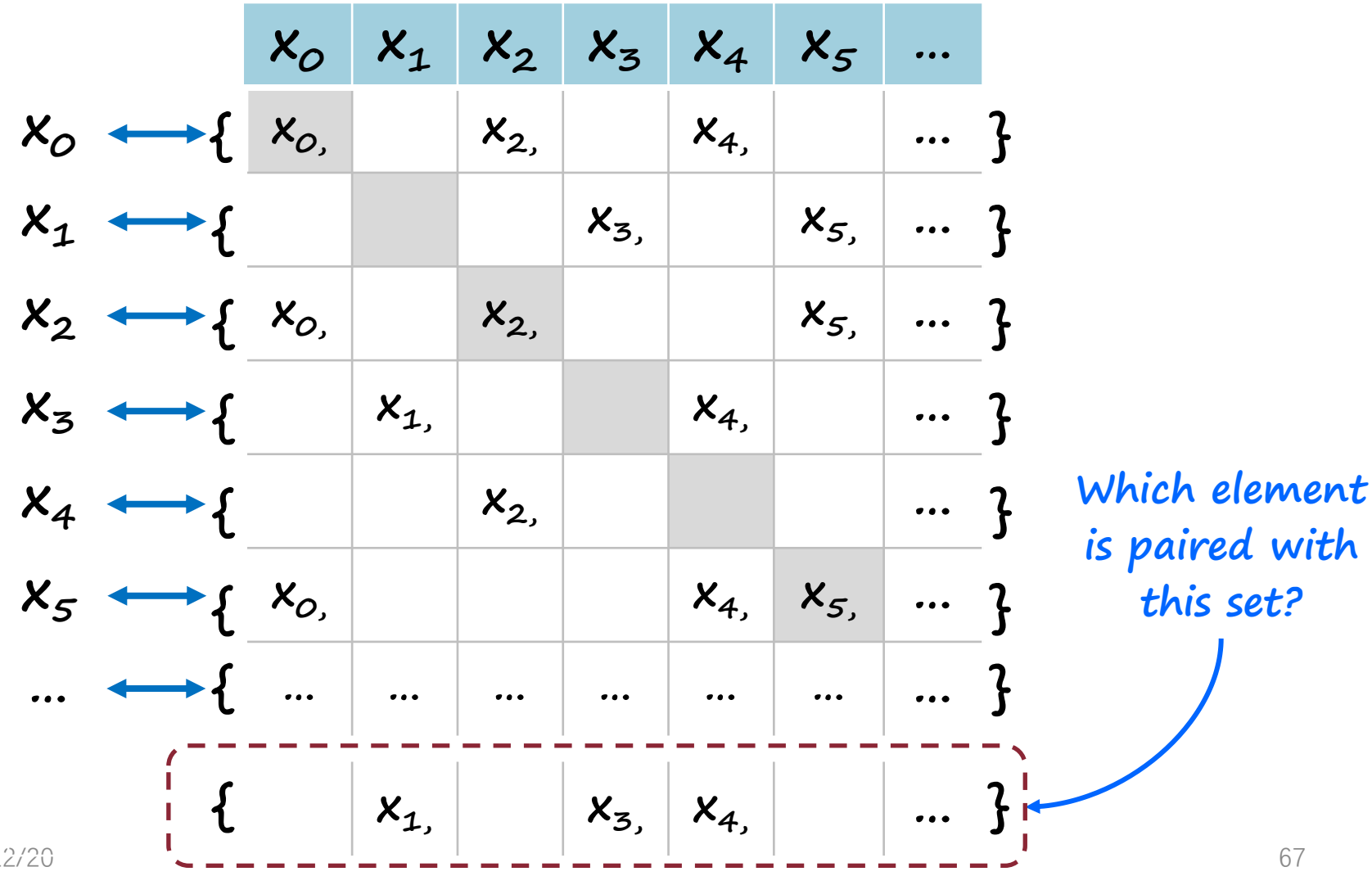

$x_0 \longleftrightarrow \{x_0, x_2, x_4, \ldots\}$
$x_1 \longleftrightarrow \{x_3, x_5, \ldots\}$
$x_2 \longleftrightarrow \{x_0, x_1, x_2, x_5, \ldots\}$
$x_3 \longleftrightarrow \{x_1, x_4, \ldots\}$
$x_4 \longleftrightarrow \{x_2, \ldots\}$
$x_5 \longleftrightarrow \{x_0, x_4, x_5, \ldots\}$
$\ldots \longleftrightarrow \{\ldots\}$

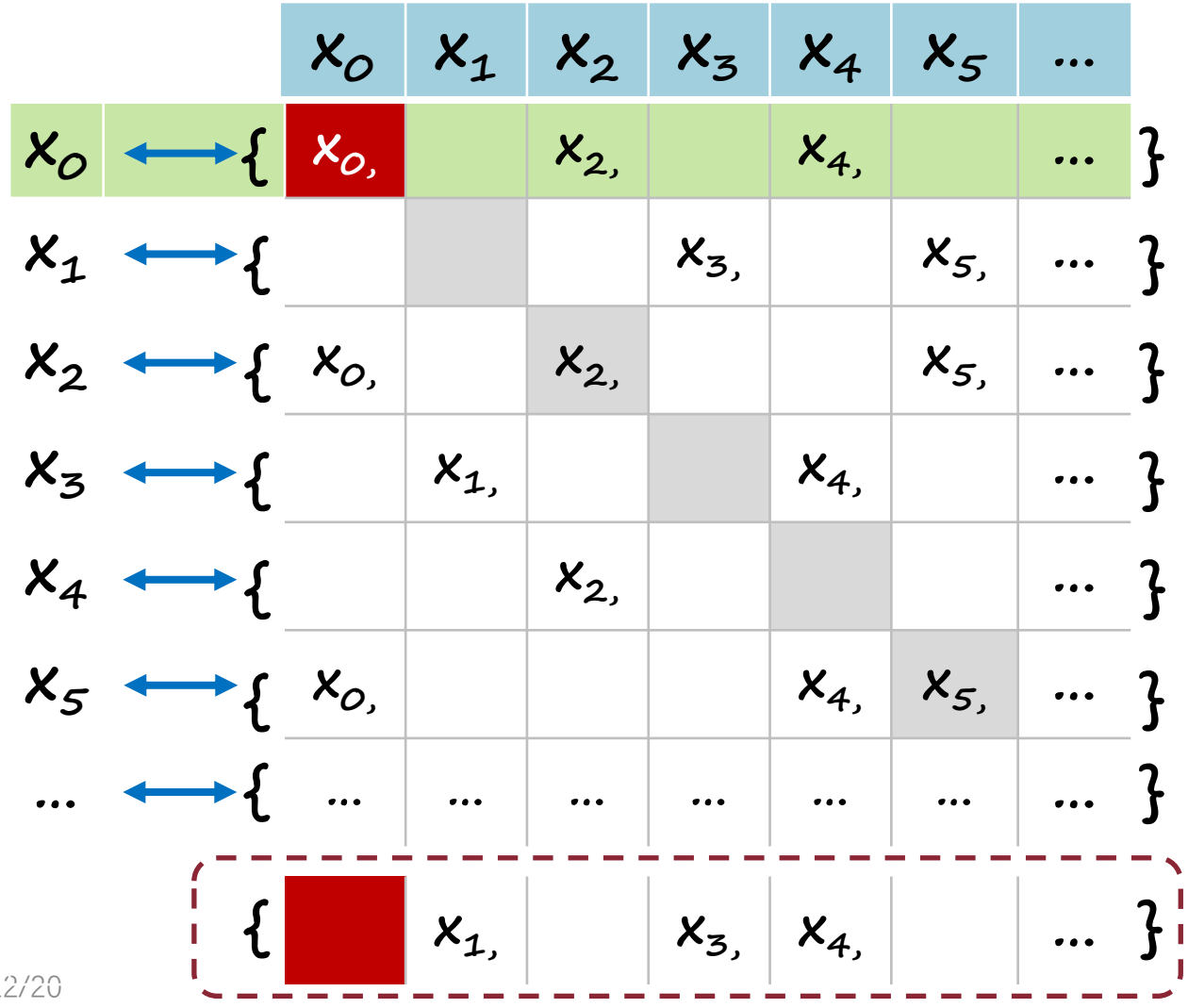

x_0 x_1 x_2 x_3 x_4 x_5 ...
x_0 ⟷ $\{x_0, x_2, x_4, \ldots\}$
x_1 ⟷ $\{x_3, x_5, \ldots\}$
x_2 ⟷ $\{x_0, x_1, x_2, x_5, \ldots\}$
x_3 ⟷ $\{x_1, x_4, \ldots\}$
x_4 ⟷ $\{x_2, \ldots\}$
x_5 ⟷ $\{x_0, x_4, x_5, \ldots\}$
... ⟷ $\{\ldots\}$

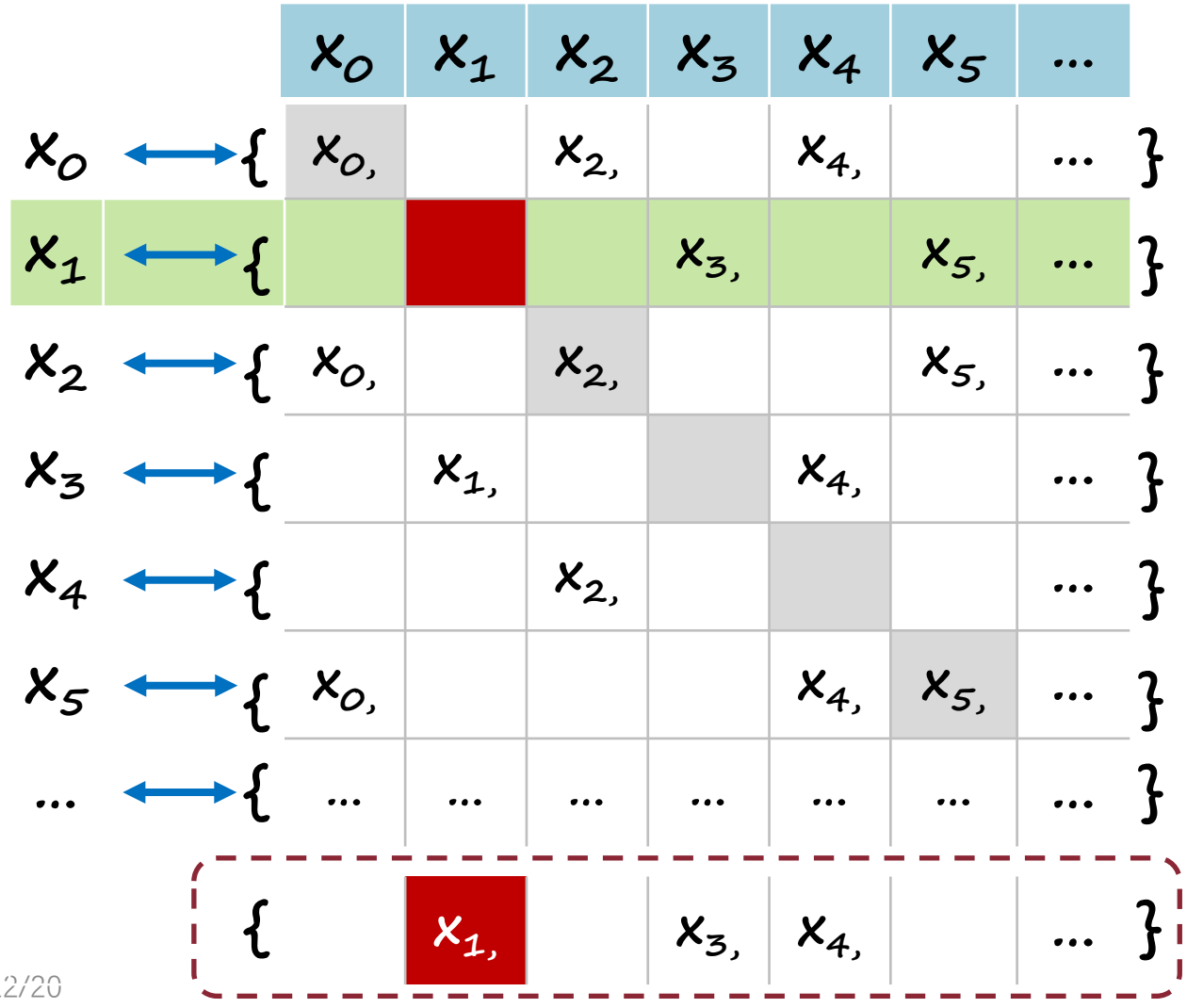

x_0 x_1 x_2 x_3 x_4 x_5 ...
x_0 { x_0, x_2, x_4, ...
x_1 { x_3, x_5, ...
x_2 { x_0, x_2, x_5, ...
x_3 { x_1, x_4, ...
x_4 { x_2, ...
x_5 { x_0, x_4, x_5, ...
... { ... ...

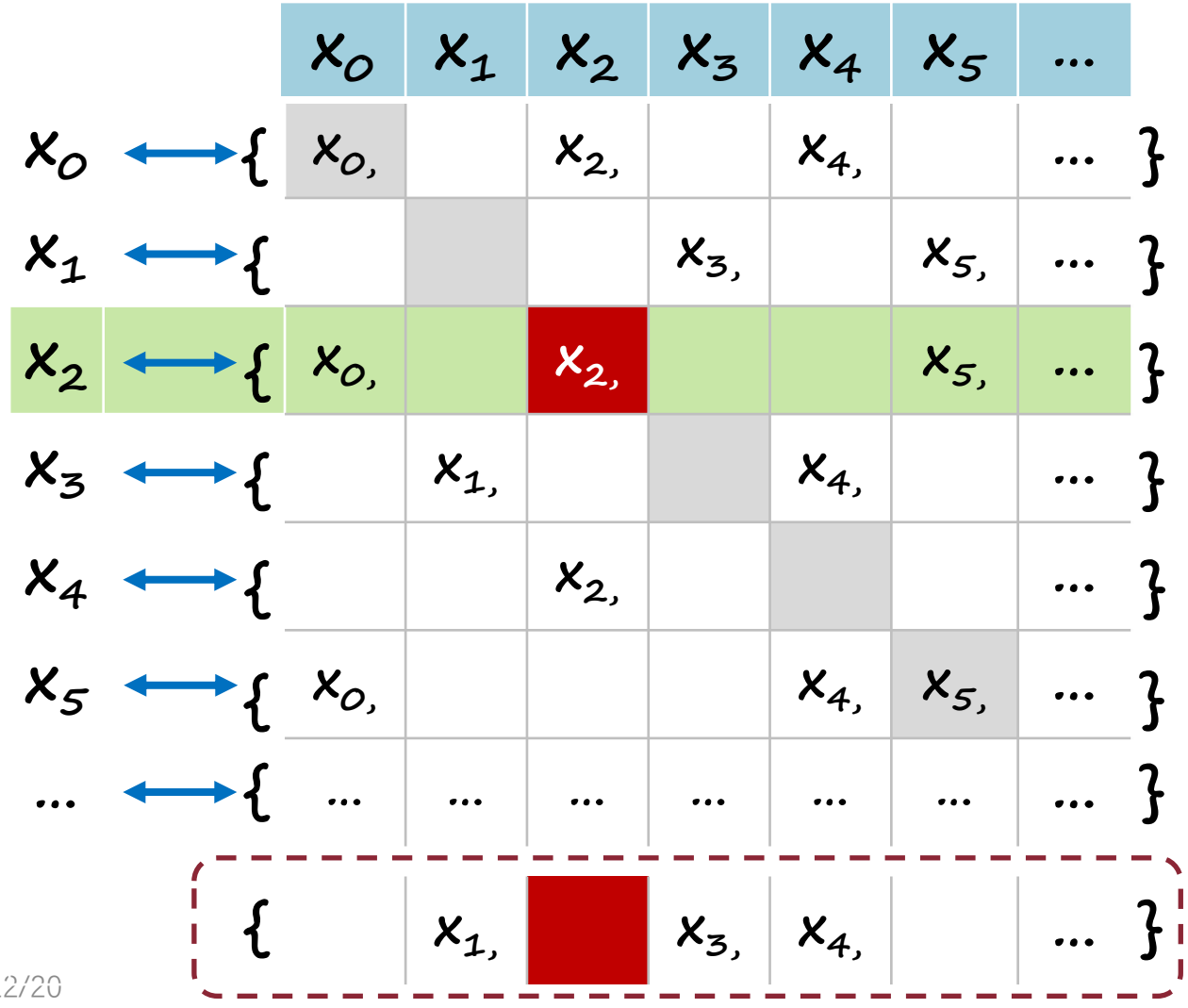

x_0 x_1 x_2 x_3 x_4 x_5 ...
x_0 { x_0, x_2, x_4, ... }
x_1 { x_3, x_5, ... }
x_2 { x_0, x_2, x_5, ... }
x_3 { x_1, x_4, ... }
x_4 { x_2, ... }
x_5 { x_0, x_4, x_5, ... }
... { ... ... }
{ x_0, x_2, x_5, }

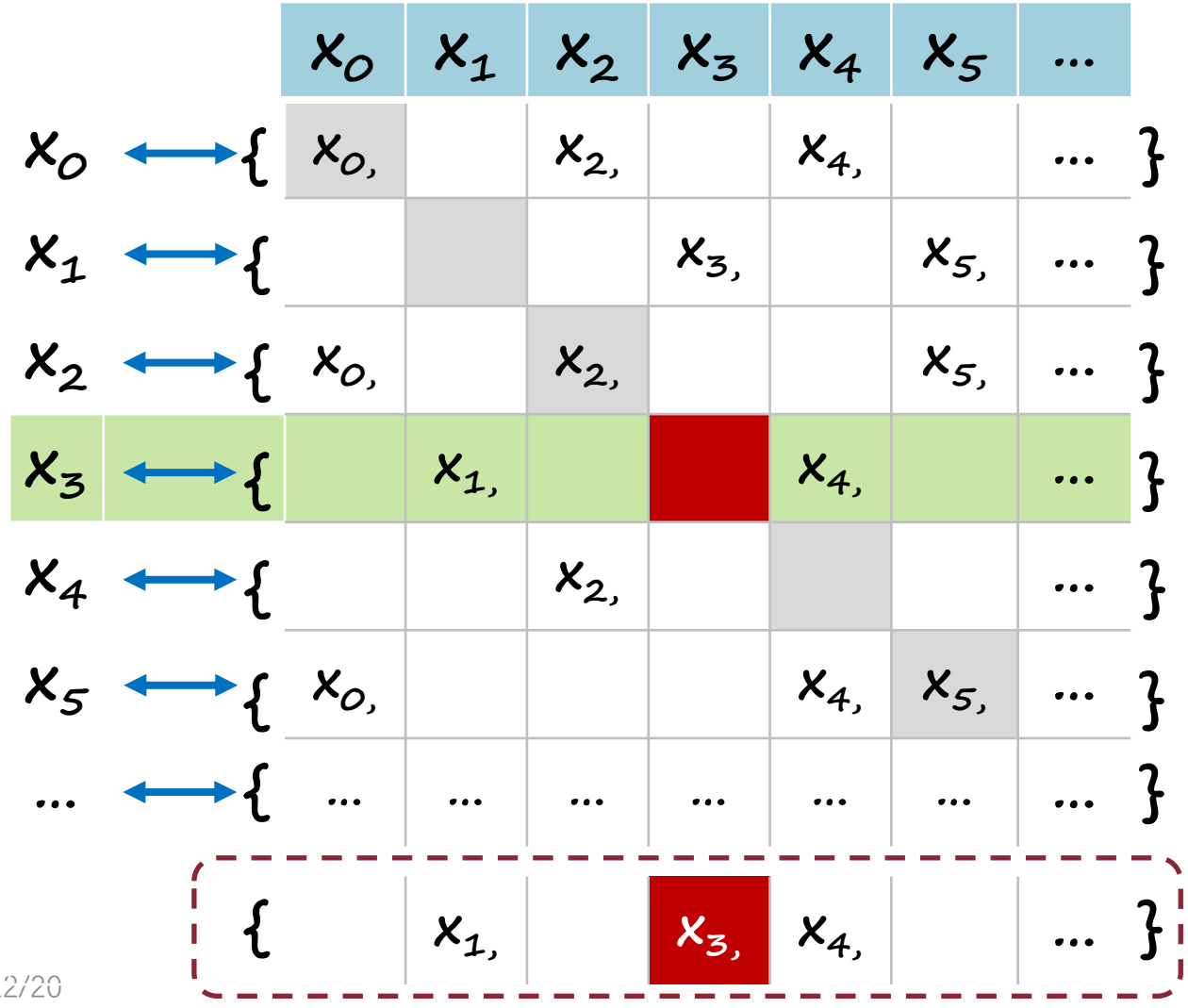

x_0 x_1 x_2 x_3 x_4 x_5 ...
x_0 { x_0, x_2, x_4, ... }
x_1 { x_3, x_5, ... }
x_2 { x_0, x_2, x_5, ... }
x_3 { x_1, x_4, ... }
x_4 { x_2, ... }
x_5 { x_0, x_4, x_5, ... }
... { ... ... }
{ x_1, x_3, x_4, ... }
Flip this set.
Swap what's included and what's excluded

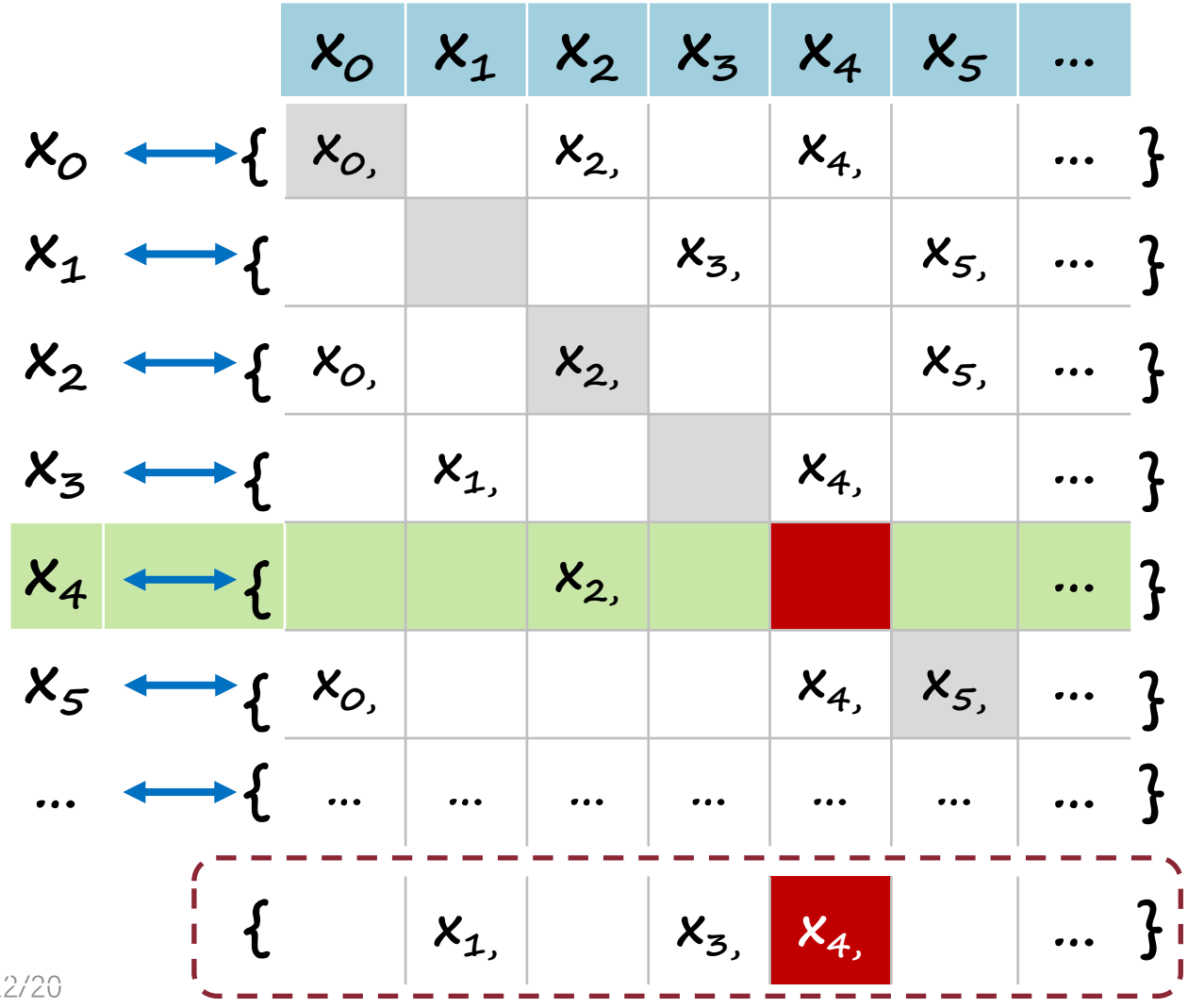

x_0 x_1 x_2 x_3 x_4 x_5 ...
x_0 { x_0, x_2, x_4, ... }
x_1 { x_3, x_5, ... }
x_2 { x_0, x_2, x_5, ... }
x_3 { x_1, x_4, ... }
x_4 { x_2, ... }
x_5 { x_0, x_4, x_5, ... }
... { ... ... }
{ x_1, x_3, x_4, ... }
Which element is paired with this set?

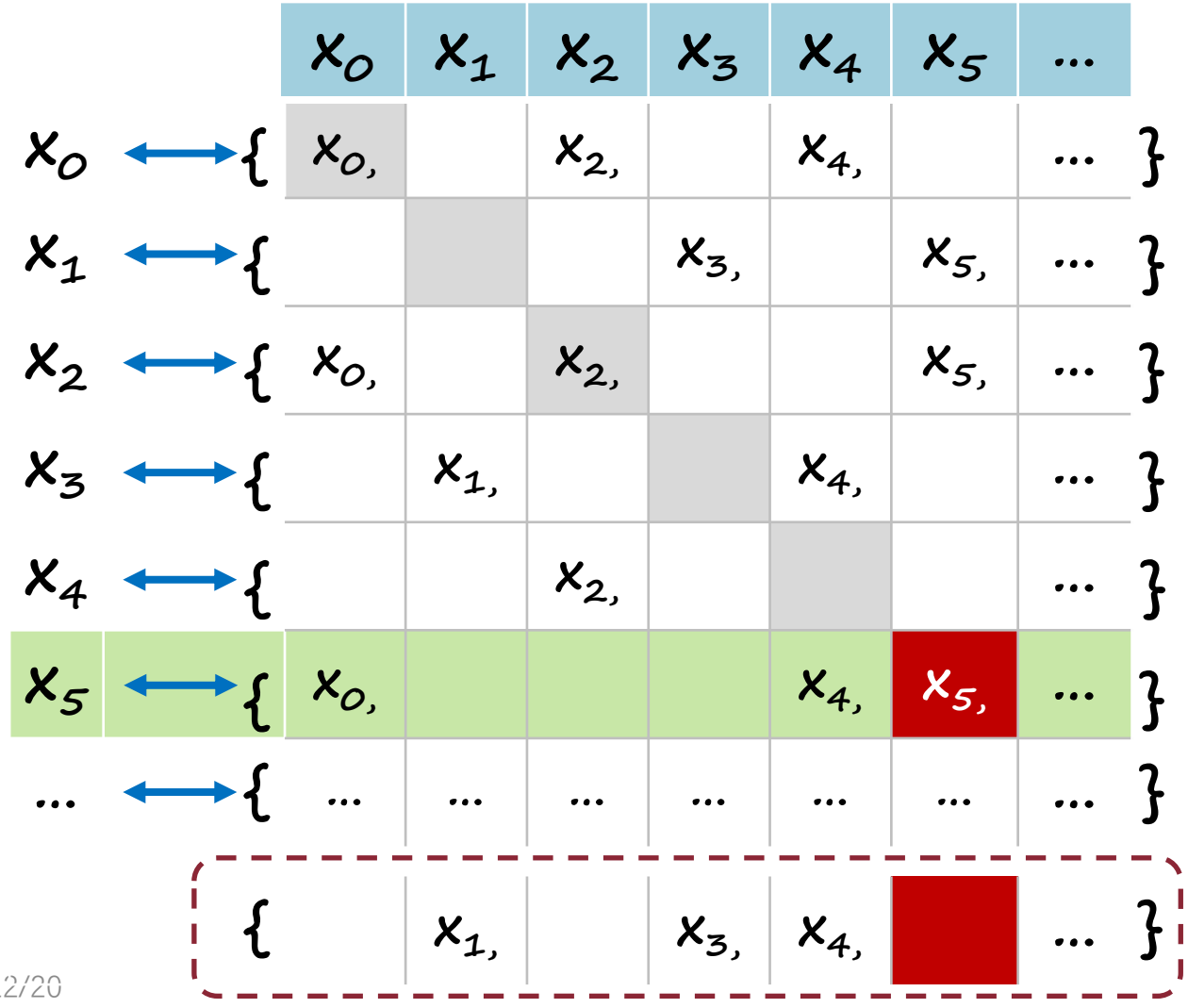

x_0 x_1 x_2 x_3 x_4 x_5 ...
x_0 { x_0, x_2, x_4, ... }
x_1 { x_3, x_5, ... }
x_2 { x_0, x_2, x_5, ... }
x_3 { x_1, x_4, ... }
x_4 { x_2, ... }
x_5 { x_0, x_4, x_5, ... }
... { ... ... }
{ x_1, x_3, x_4, ... }

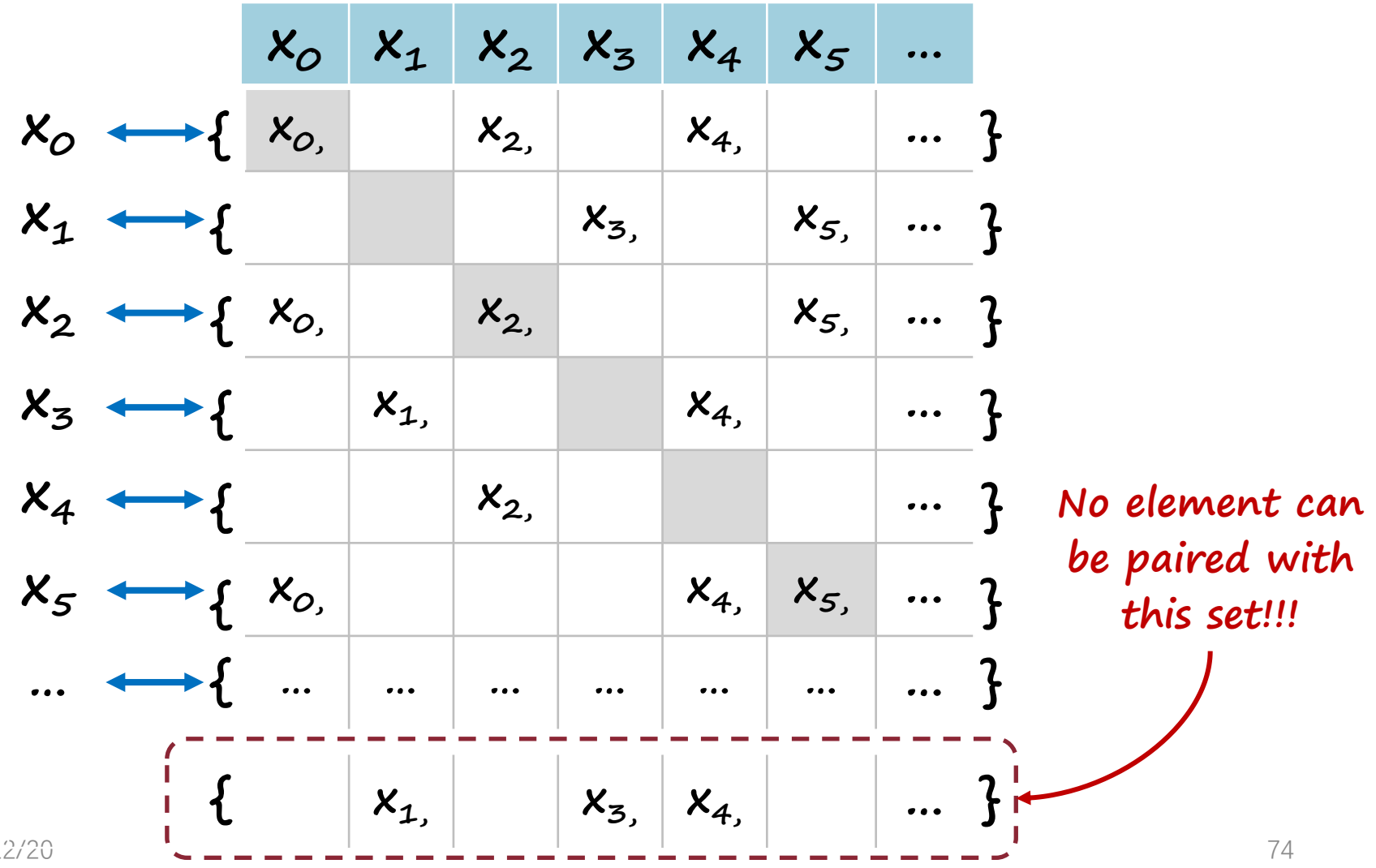

x0 x1 x2 x3 x4 x5 ...
x0 { x0, x2, x4, ... }
x1 { x3, x5, ... }
x2 { x0, x2, x5, ... }
x3 { x1, x4, ... }
x4 { x2, ... }
x5 { x0, x4, x5, ... }
... { ... ... }
{ x1, x3, x4, ... }


x_0 x_1 x_2 x_3 x_4 x_5 ...
x_0 { x_0, x_2, x_4, ... }
x_1 { x_3, x_5, ... }
x_2 { x_0, x_2, x_5, ... }
x_3 { x_1, x_4, ... }
x_4 { x_2, ... }
x_5 { x_0, x_4, x_5, ... }
... { ... ... }
{ x_1, x_3, x_4, ... }

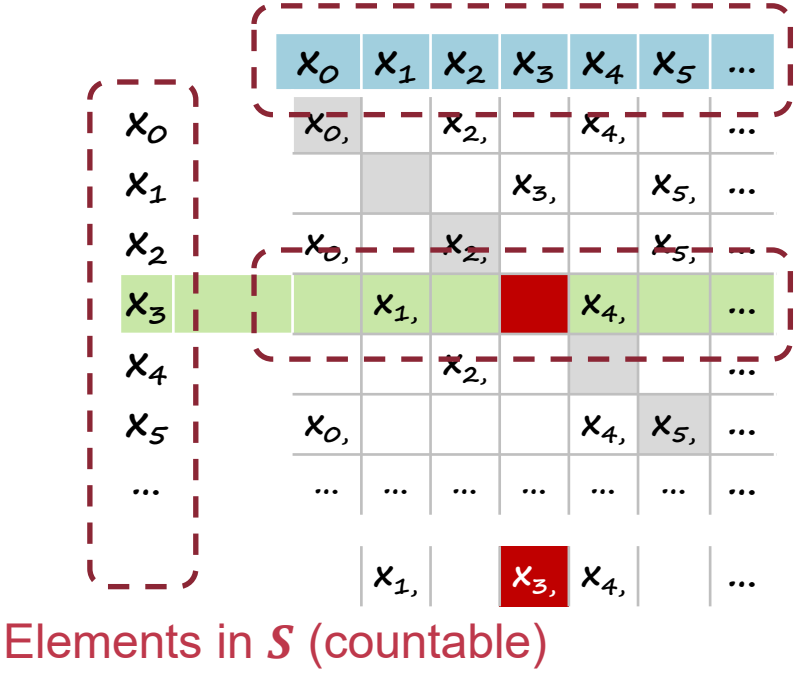

x_0 x_1 x_2 x_3 x_4 x_5 …
x_0 { x_0, x_2, x_4, … }
x_1 { x_3, x_5, … }
x_2 { x_0, x_2, x_5, … }
x_3 { x_1, x_4, … }
x_4 { x_2, … }
x_5 { x_0, x_4, x_5, … }
… { … … … … … … … }
{ x_1, x_3, x_4, … }


x_0 x_1 x_2 x_3 x_4 x_5 ...
x_0 { x_0, x_2, x_4, ... }
x_1 { x_3, x_5, ... }
x_2 { x_0, x_2, x_5, ... }
x_3 { x_1, x_4, ... }
x_4 { x_2, ... }
x_5 { x_0, x_4, x_5, ... }
... { ... ... }
{ x_1, x_3, x_4, ... }


x0 x1 x2 x3 x4 x5 ...
x0 { x0, x2, x4, ... }
x1 { x3, x5, ... }
x2 { x0, x2, x5, ... }
x3 { x1, x4, ... }
x4 { x2, ... }
x5 { x0, x4, x5, ... }
... { ... ... }
{ x1, x3, x4, ... }


x_0 x_1 x_2 x_3 x_4 x_5 ...
x_0 { x_0, x_2, x_4, ... }
x_1 { x_3, x_5, ... }
x_2 { x_0, x_2, x_5, ... }
x_3 { x_1, x_4, ... }
x_4 { x_2, ... }
x_5 { x_0, x_4, x_5, ... }
... { ... ... }
{ x_1, x_3, x_4, ... }
No element can be paired with this set!!!

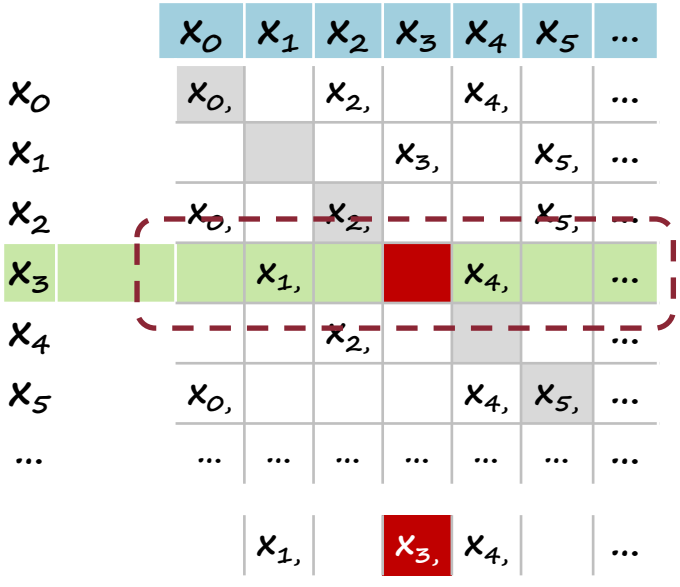
Comparing two diagonal methods



	1	2	3	4	5	6	…
M_1	0	0	0	1	1	0	…
M_2	0	1	1	0	1	1	…
M_3	0	0	1	1	0	1	…
M_4	1	1	0	1	0	1	…
M_5	1	0	0	0	0	0	…
M_6	0	1	1	0	1	0	…
…	…	…	…	…	…	…	…
	1	0	0	0	1	1	

Conclusion: No match between S and $P(S)$

Comparing two diagonal methods



All strings (countable)

All TMs (countable)

	1	2	3	4	5	6	...
M_1	0	0	0	1	1	0	...
M_2	0	1	1	0	1	1	...
M_3	0	0	1	1	0	1	...
M_4	1	1	0	1	0	1	...
M_5	1	0	0	0	0	0	...
M_6	0	1	1	0	1	0	...
...	...	...	...	...	...	...	...
	1	0	0	0	1	1	

A behavior of an automata (corresponding to a language)

Conclusion: Turing machines can not recognize all languages

Diagonalization Language

- The language L_d, the diagonalization language, is the set of string w_i such that w_i is not in $L(M_i)$

$$L_d = \{w_i | w_i \notin L(M_i)\}$$

- Notice that the code for M_i is just w_i, so we can write L_d in a more interesting way:

$$L_d = \{M | M \notin L(M)\}$$

- The diagonalization language is all the codes of Turing machine which does not accept its own code.

Where We're Going

- A string is a sequence of characters.
  - There are at most as many programs as there are strings.
  - There are at least as many problems as there are sets of strings.
- **From Cantor's Theorem, we know that there are more sets of strings than strings**

$$|\text{Programs}| \leq |\text{Strings}| < |\text{Set of strings}| \leq |\text{Problems}|$$

Where We Are

- Now can we answer the question:

  What problems can a computer solve?

- Unfortunately we can only tell there exist some unsolvable(undecidable) problems. But we cannot know exactly what problems can be solved.
  - Actually there's little understanding about the sufficient condition for a problem to be computable.

Where We Are

- Knowing some problem is undecidable means:
  - Hold back our ambition of finishing an impossible task. (The universal program verifier does not exist.)
  - Then try to find another decidable problem to solve. (But we can build a verifier for specific programs.)
  - And it is interesting itself.

How we investigate computability?

2.1 Discussion on Problems

Part1. Intro & Set theory(I) : Basics & Formal Language.

Part2. Set Theory(II): Axiom system & Cardinality.

Part3. Capture Structures: Binary Relation & Function

2.2 Discussion on Computation

Part1. Turing Machine Basics.

Part2. Variants of Turing Machine. Church-Turing Thesis.

2.3 Discussion on computability

Part1. The Language of Turing Machine. R & RE

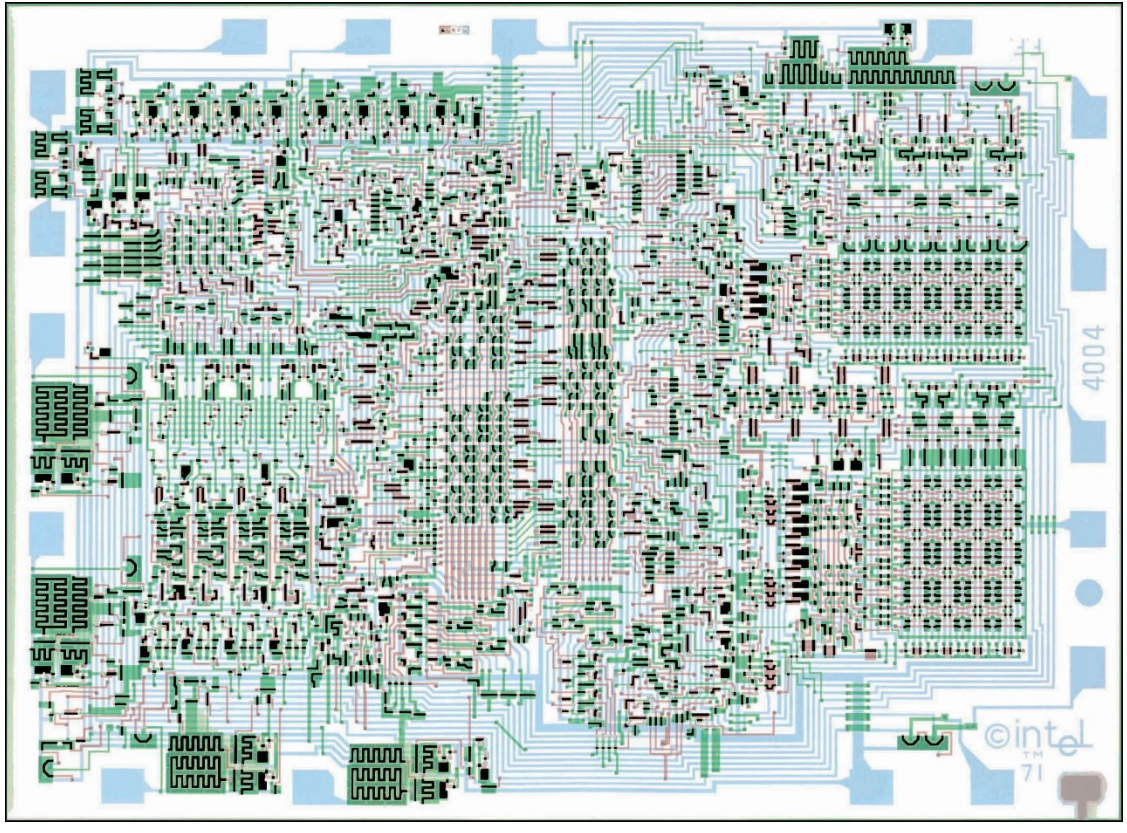
Part2. Undecidability.

Review: DFA & Turing Machine

2022-12-21

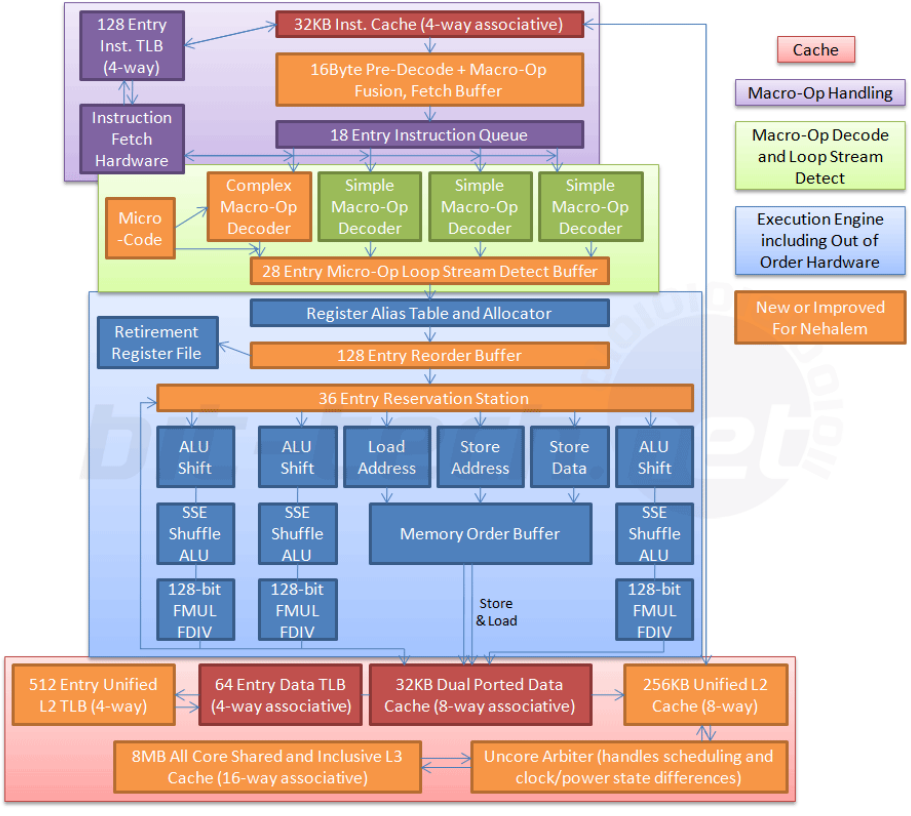
What languages can a **computer** recognize?

Computers are complicated.



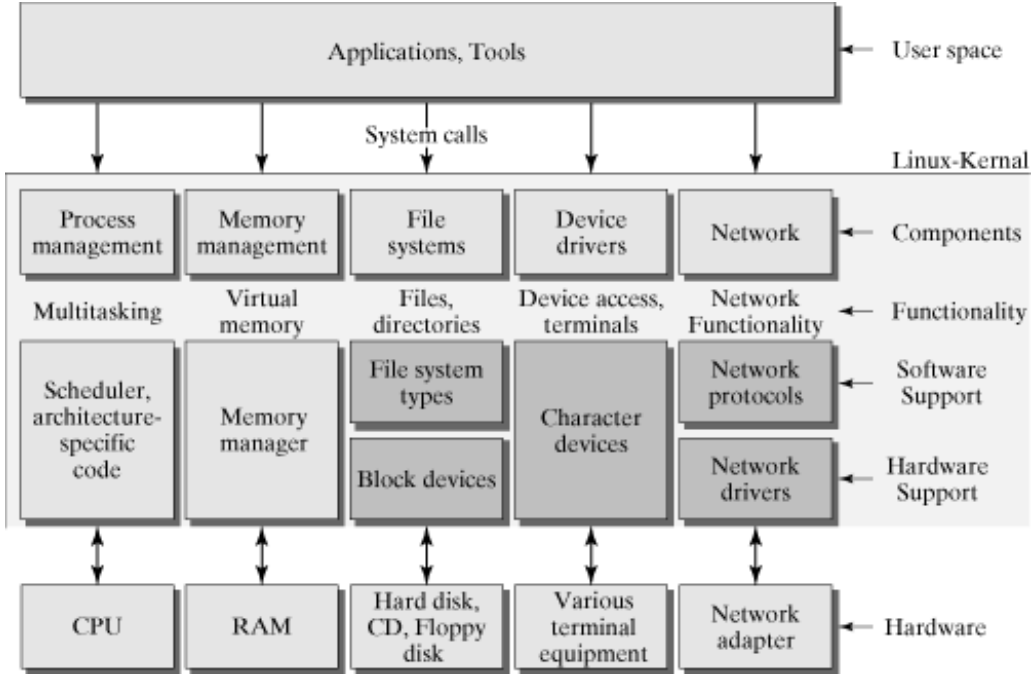
Intel 4004, first commercial CPU

Computers are complicated.



Core i7 Architecture.

Computers are complicated.



Architecture of Operating System

We need to model computer in a simple and clean manner!!

Deterministic Finite Automaton(DFA, 确定有限自动机)

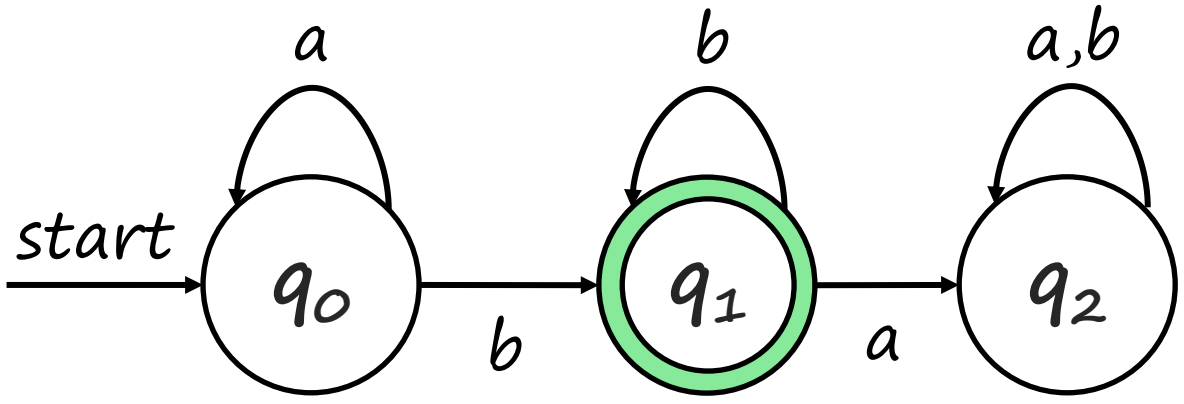
- A **deterministic finite automaton(DFA)** consists of:
  - Q:A finite set of states
  - Σ: A finite set of input symbols
  - δ: A transition function. $Q \times \Sigma \to Q$
  - q_0: The start state
  - F: A set of final or accepting states. $F \subseteq Q$
- We can donate a DFA as a “five tuple”

$$A = (Q, \Sigma, \delta, q_0, F)$$

Deterministic Finite Automaton(DFA)

- A DFA takes a finite string(sequence of input symbols) as input, and process the input symbols of the string one by one, and change its state according to the transition function δ.
- DFA will always stop, if the state is in the set of accepting states(F) when a DFA stops, we say that this DFA **accepts** the input. Otherwise it **rejects** the input.

Simpler Notations for DFA



Transition Diagram

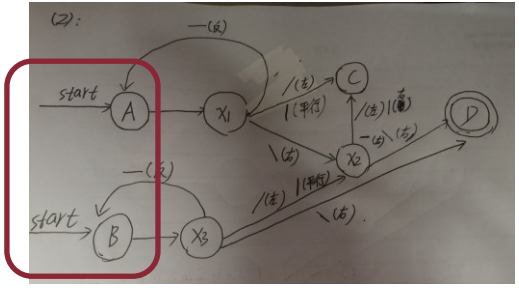
start state

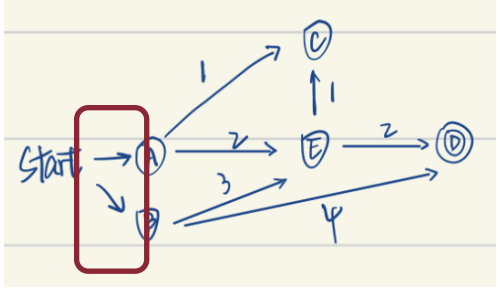
	a	b
→ q_0	q_0	q_1
* q_1	q_2	q_1
q_2	q_2	q_2

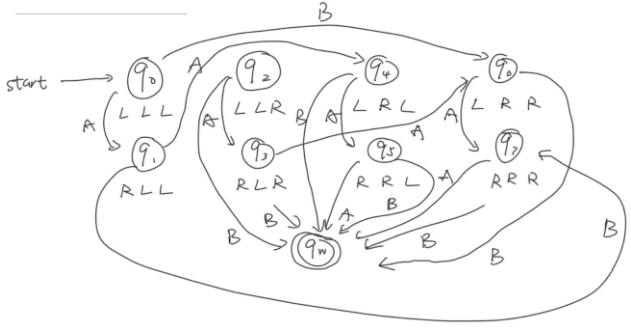
final state

Transition Table

DFA







Unique start state:
There is only one start state for each DFA

No arbitrary transition:
every transition must be triggered by an input symbol

Final state ≠ Stop state:
DFA can still accept symbols at final states

Extending the Transition Function

For a DFA $(Q, \Sigma, \delta, q_0, F)$, we can define an ***extended transition function***

$$\hat{\delta}: Q \times \Sigma^* \rightarrow Q$$

$\hat{\delta}$ takes a state q and a string w as input, and returns the state after processing string w from state q.

- $\hat{\delta}(q, \epsilon) = q$
- If $w = xa$, where $a \in \Sigma$. Then $\hat{\delta}(q, w) = \delta(\hat{\delta}(q, x), a)$

Extending the Transition Function

- $\hat{\delta}(q_0, \epsilon) = q_0$
- $\hat{\delta}(q_0, 1) = \delta\big(\hat{\delta}(q_0, \epsilon), 1\big) = \delta(q_0, 1) = q_1$
- $\hat{\delta}(q_0, 11) = \delta\big(\hat{\delta}(q_0, 1), 1\big) = \delta(q_1, 1) = q_0$
- $\hat{\delta}(q_0, 110) = \delta\big(\hat{\delta}(q_0, 11), 0\big) = \delta(q_0, 0) = q_2$
- $\hat{\delta}(q_0, 1101) = \delta\big(\hat{\delta}(q_0, 110), 1\big) = \delta(q_2, 1) = q_3$
- $\hat{\delta}(q_0, 11010) = \delta\big(\hat{\delta}(q_0, 1101), 0\big) = \delta(q_3, 0) = q_1$
- $\hat{\delta}(q_0, 110101) = \delta\big(\hat{\delta}(q_0, 11010), 1\big) = \delta(q_1, 1) = q_0$

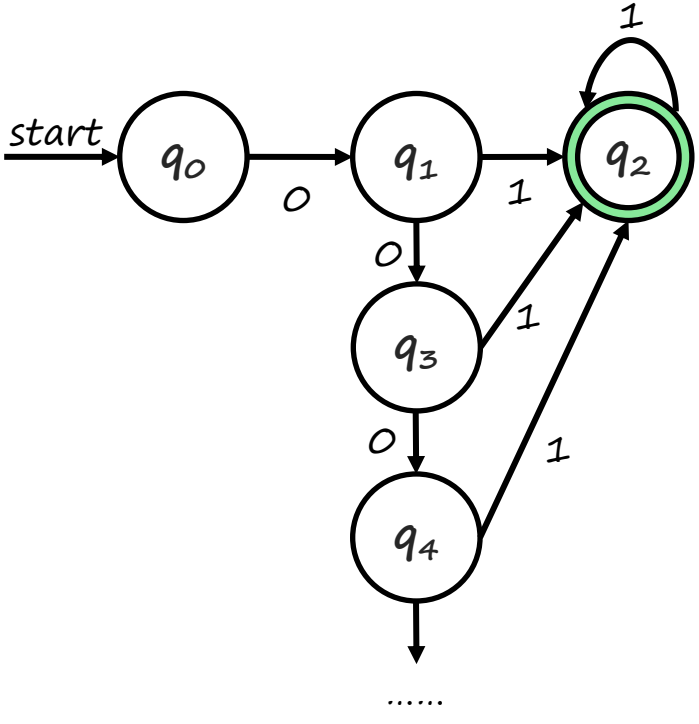
	0	1
$* \rightarrow q_0$	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

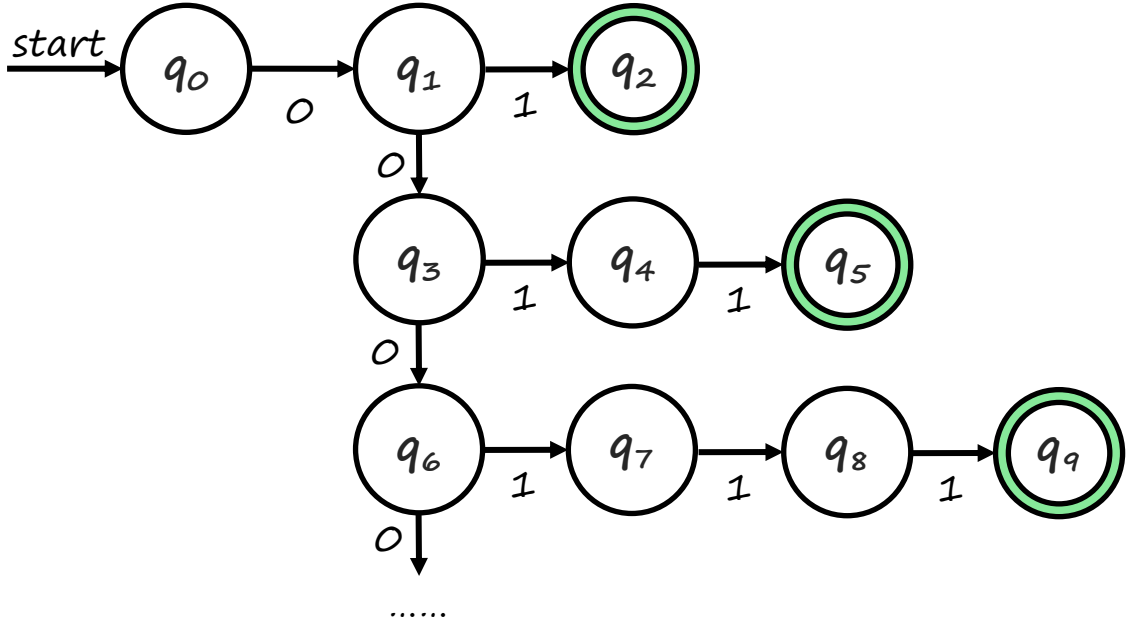
Regular Expression(正则表达式)

- Regular expressions are used to denote regular languages.
- For an alphabet Σ
  - $\emptyset, \epsilon, a$ are regular expressions corresponding to languages $\emptyset, \{\epsilon\}, \{a\}$, where $a \in \Sigma$
  - If $\boldsymbol{r}, \boldsymbol{s}$ are regular expression corresponding to language L_r, L_s.
    - $\boldsymbol{r} + \boldsymbol{s}$ (or $\boldsymbol{r}|\boldsymbol{s}$): $L_r \cup L_s$
    - $\boldsymbol{rs}$: $\{wu | w \in L_r \wedge u \in L_s\}$
    - $\boldsymbol{r^k}$: $\{w_1 w_2 \cdots w_{k-1} w_k | w_1 \in L_r \wedge w_2 \in L_r \wedge \cdots w_k \in L_r\}$
    - $\boldsymbol{r^+}$: $L_{r^1} \cup L_{r^2} \cup L_{r^3} \ldots \cup L_{r^n} \ldots$
    - $\boldsymbol{r^*}$: $L_{r^+} \cup \{\epsilon\}$

The Limitation of DFA

- There is no DFA for languages like $\{0^n 1^n | n \in \mathbb{N}^+\}$.





Turing Machine

- A **Turing Machine** consists of:
  - Q:A finite set of states
  - Σ: A finite set of input symbols
  - Γ: The complete set of tape symbols. $\Sigma \subset \Gamma$
  - δ: A transition function. $Q \times \Sigma \to Q \times \Gamma \times \{L, R\}$
  - q_0: The start state. $q_0 \in Q$
  - B: The blank symbol. $B \in \Gamma \wedge B \notin \Sigma$.
  - F: A set of final or accepting states. $F \subseteq Q$
- We can donate a turning machine as a “seven tuple”

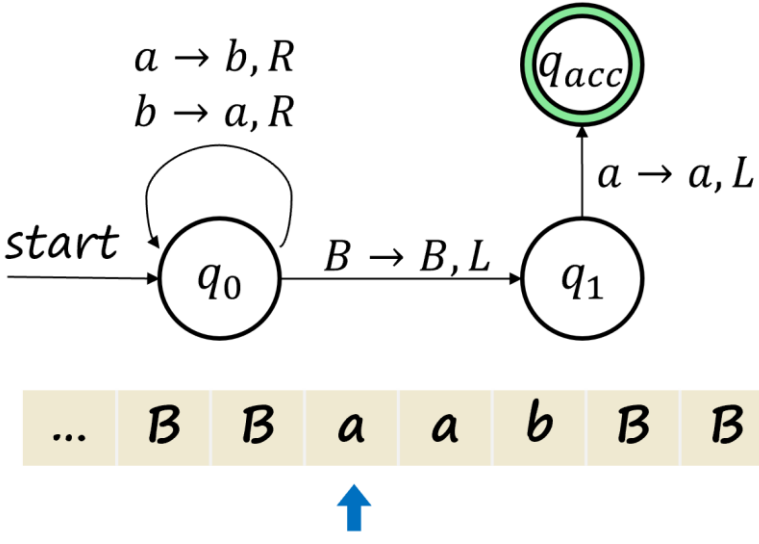
$$A = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

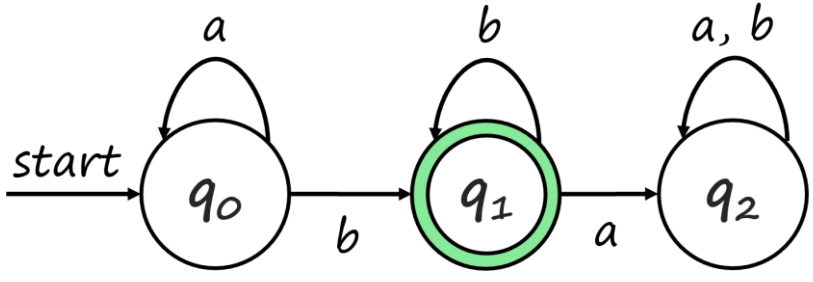
Turing Machine

- Initially the input symbols are put into tape, and the tape heap is pointed to the start of input symbols.
- Turing Machine finishes computation by moving the tape head. In a move, the Turing Machine will:
  - Change state
  - Write a tape symbol in the cell pointed
  - Move the tape head left or right.
- Turing Machine will accept the input once it gets into accept states, or finishes computation with rejection when no move can be made.

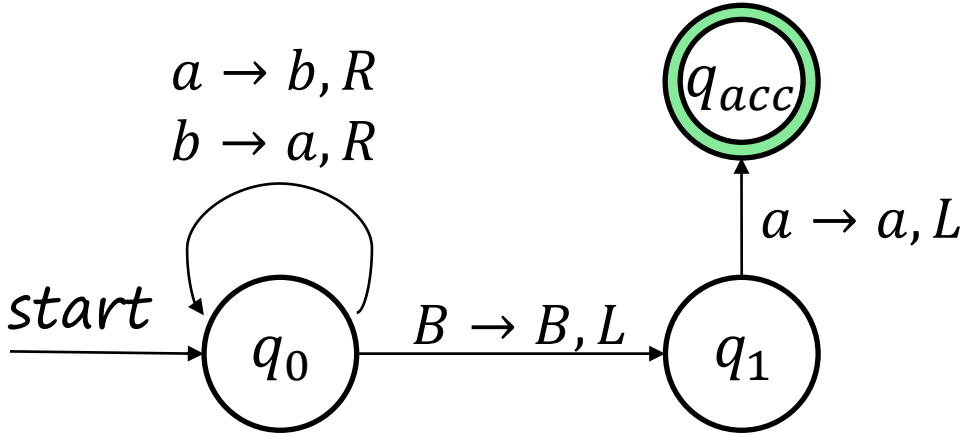
TM and DFA

- DFA has only a bunch of states but TM has an extra "memory-like" tape to store the execution state.
- TM has more capability than DFA
  - A DFA can be converted into a TM





Simpler Notations for TM



Transition Diagram

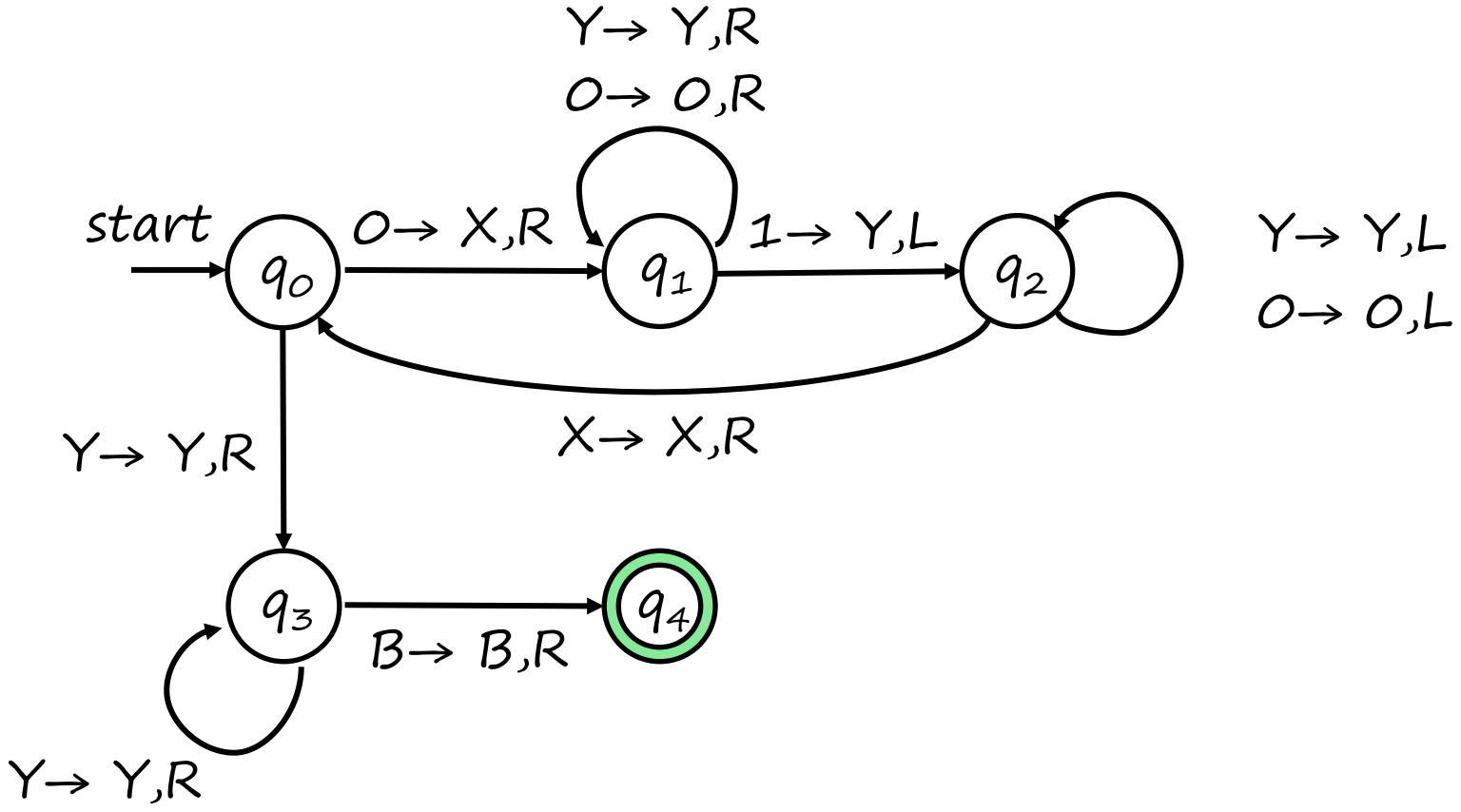
start state

	a	b	B
$\to q_0$	(q_0, b, R)	(q_0, a, R)	(q_1, B, L)
q_1	(q_{acc}, a, L)	–	–
$* q_{acc}$	–	–	–

final state

Transition Table

Example: TM for $\{0^n 1^n\}$



Church-Turing Thesis

> Every effective method of computation is either equivalent to or weaker than a Turing machine.

- This is not a **theorem** but a falsifiable scientific hypo**thesis**. But it has been thoroughly tested! So we have strong faith in its correctness.
- This means Turing Machine can model any “computation”.