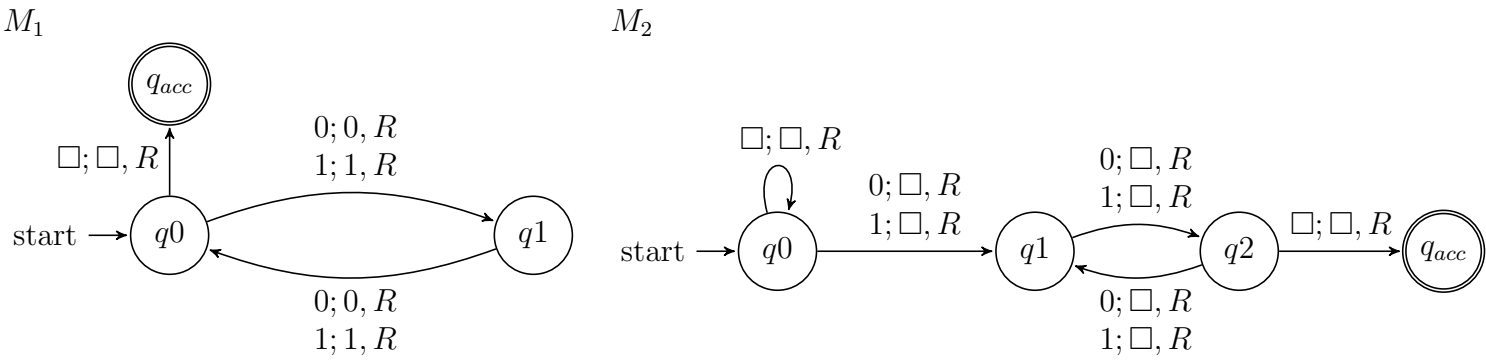# Day20

Acceptance problem		
for Turing machines	A_{TM}	$\{\langle M, w \rangle \mid M$ is a Turing machine that accepts input string $w\}$
Language emptiness testing		
for Turing machines	E_{TM}	$\{\langle M \rangle \mid M$ is a Turing machine and $L(M) = \emptyset\}$
Language equality testing		
for Turing machines	EQ_{TM}	$\{\langle M_1, M_2 \rangle \mid M_1$ and M_2 are Turing machines and $L(M_1) = L(M_2)\}$

M_1

M_2

Example strings in A_{TM}

Example strings in E_{TM}

Example strings in EQ_{TM}

Theorem: A_{TM} is Turing-recognizable.

Strategy: To prove this theorem, we need to define a Turing machine R_{ATM} such that $L(R_{ATM}) = A_{TM}$.

Define $R_{ATM} =$ "

Proof of correctness:

We will show that A_{TM} is undecidable. *First, let's explore what that means.*

To prove that a computational problem is **decidable**, we find/ build a Turing machine that recognizes the language encoding the computational problem, and that is a decider.

How do we prove a specific problem is **not decidable**?

How would we even find such a computational problem?

Counting arguments for the existence of an undecidable language:

- The set of all Turing machines is countably infinite.

- Each recognizable language has at least one Turing machine that recognizes it (by definition), so there can be no more Turing-recognizable languages than there are Turing machines.

- Since there are infinitely many Turing-recognizable languages (think of the singleton sets), there are countably infinitely many Turing-recognizable languages.

- Such the set of Turing-decidable languages is an infinite subset of the set of Turing-recognizable languages, the set of Turing-decidable languages is also countably infinite.

Since there are uncountably many languages (because $\mathcal{P}(\Sigma^*)$ is uncountable), there are uncountably many unrecognizable languages and there are uncountably many undecidable languages.

Thus, there's at least one undecidable language!

What's a specific example of a language that is unrecognizable or undecidable?

To prove that a language is undecidable, we need to prove that there is no Turing machine that decides it.

Key idea: proof by contradiction relying on self-referential disagreement.

Theorem: A_{TM} is not Turing-decidable.

Proof: Suppose **towards a contradiction** that there is a Turing machine that decides A_{TM}. We call this presumed machine M_{ATM}.

By assumption, for every Turing machine M and every string w

- If $w \in L(M)$, then the computation of M_{ATM} on $\langle M, w \rangle$ _____

- If $w \notin L(M)$, then the computation of M_{ATM} on $\langle M, w \rangle$ _____

Define a **new** Turing machine using the high-level description:

$D =$ " On input $\langle M \rangle$, where M is a Turing machine:

1. Run M_{ATM} on $\langle M, \langle M \rangle \rangle$.

2. If M_{ATM} accepts, reject; if M_{ATM} rejects, accept."

Is D a Turing machine?

Is D a decider?

What is the result of the computation of D on $\langle D \rangle$?

Summarizing:

- A_{TM} is recognizable.

- A_{TM} is not decidable.

Recall definition: A language L over an alphabet Σ is called **co-recognizable** if its complement, defined as $\Sigma^* \setminus L = \{x \in \Sigma^* \mid x \notin L\}$, is Turing-recognizable.

and Recall Theorem (Sipser Theorem 4.22): A language is Turing-decidable if and only if both it and its complement are Turing-recognizable.

- A_{TM} is recognizable.
- A_{TM} is not decidable.
- $\overline{A_{TM}}$ is not recognizable.
- $\overline{A_{TM}}$ is not decidable.

Day21

Mapping reduction

Motivation: Proving that A_{TM} is undecidable was hard. How can we leverage that work? Can we relate the decidability / undecidability of one problem to another?

> If problem X is **no harder than** problem Y
>
> ... and if Y is easy,
>
> ... then X must be easy too.

> If problem X is **no harder than** problem Y
>
> ... and if X is hard,
>
> ... then Y must be hard too.

"Problem X is no harder than problem Y" means "Can answer questions about membership in X by converting them to questions about membership in Y".

Definition: For any languages A and B, A is **mapping reducible to** B means there is a computable function $f : \Sigma^* \to \Sigma^*$ such that *for all* strings x in Σ^*,

$$x \in A \qquad \text{if and only if} \qquad f(x) \in B.$$

Notation: when A is mapping reducible to B, we write $A \leq_m B$.

Intuition: $A \leq_m B$ means A is no harder than B, i.e. that the level of difficulty of A is less than or equal the level of difficulty of B.

TODO

1. What is a computable function?

2. How do mapping reductions help establish the computational difficulty of languages?

Computable functions

Definition: A function $f : \Sigma^* \to \Sigma^*$ is a **computable function** means there is some Turing machine such that, for each x, on input x the Turing machine halts with exactly $f(x)$ followed by all blanks on the tape

Examples of computable functions:

The function that maps a string to a string which is one character longer and whose value, when interpreted as a fixed-width binary representation of a nonnegative integer is twice the value of the input string (when interpreted as a fixed-width binary representation of a non-negative integer)

$$f_1 : \Sigma^* \to \Sigma^* \qquad f_1(x) = x0$$

To prove f_1 is computable function, we define a Turing machine computing it.

High-level description

"On input w

1. Append 0 to w.
2. Halt."

Implementation-level description

"On input w

1. Sweep read-write head to the right until find first blank cell.
2. Write 0.
3. Halt."

Formal definition $(\{q0, qacc, qrej\}, \{0, 1\}, \{0, 1, \llcorner\}, \delta, q0, qacc, qrej)$ where δ is specified by the state diagram:

The function that maps a string to the result of repeating the string twice.

$$f_2 : \Sigma^* \to \Sigma^* \qquad f_2(x) = xx$$

The function that maps strings that are not the codes of NFAs to the empty string and that maps strings that code NFAs to the code of a DFA that recognizes the language recognized by the NFA produced by the macro-state construction from Chapter 1.

The function that maps strings that are not the codes of Turing machines to the empty string and that maps strings that code Turing machines to the code of the related Turing machine that acts like the Turing machine coded by the input, except that if this Turing machine coded by the input tries to reject, the new machine will go into a loop.

$$f_4 : \Sigma^* \to \Sigma^* \qquad f_4(x) = \begin{cases} \varepsilon & \text{if } x \text{ is not the code of a TM} \\ \langle (Q \cup \{q_{trap}\}, \Sigma, \Gamma, \delta', q_0, q_{acc}, q_{rej}) \rangle & \text{if } x = \langle (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej}) \rangle \end{cases}$$

where $q_{trap} \notin Q$ and

$$\delta'((q, x)) = \begin{cases} (r, y, d) & \text{if } q \in Q, \ x \in \Gamma, \ \delta((q, x)) = (r, y, d), \text{ and } r \neq q_{rej} \\ (q_{trap}, \llcorner, R) & \text{otherwise} \end{cases}$$

Definition: A is **mapping reducible to** B, $A \leq_m B$m means there is a computable function $f : \Sigma^* \to \Sigma^*$ such that *for all* strings x in Σ^*,

$$x \in A \qquad \text{if and only if} \qquad f(x) \in B.$$

In this case, we say the function f **witnesses** that A is mapping reducible to B.

Making intutition precise ...

Theorem (Sipser 5.22): If $A \leq_m B$ and B is decidable, then A is decidable.

Theorem (Sipser 5.23): If $A \leq_m B$ and A is undecidable, then B is undecidable.

Day22

Recall definition: A is **mapping reducible to** B means there is a computable function $f : \Sigma^* \to \Sigma^*$ such that *for all* strings x in Σ^*,

$$x \in A \qquad \text{if and only if} \qquad f(x) \in B.$$

Notation: when A is mapping reducible to B, we write $A \leq_m B$.

Intuition: $A \leq_m B$ means A is no harder than B, i.e. that the level of difficulty of A is less than or equal the level of difficulty of B.

Example: $A_{TM} \leq_m A_{TM}$ $\qquad\qquad A_{TM} = \{\langle M, w \rangle \mid M$ is a TM and w is a string and $w \in L(M)\}$

To prove, need a witnessing function $f : \Sigma^* \to \Sigma^*$ that is (1) computable and (2) for each $x \in \Sigma^*$, $x \in A_{TM}$ iff $f(x) \in A_{TM}$

Corollary: For any language L, $L \leq_m L$, as witnessed by

Example: $A_{DFA} \leq_m \{ww \mid w \in \{0,1\}^*\}$

To prove, need a witnessing function $f : \Sigma^* \to \Sigma^*$ that is (1) computable and (2) for each $x \in \Sigma^*$, $x \in A_{DFA}$ iff $f(x) \in \{ww \mid w \in \{0,1\}^*\}$

Corollary: For any language decidable language X and any set Y with at least one string string in Y and at least one string not in Y, $X \leq_m Y$, as witnessed by

Next: consider mapping reductions between potentially undecidable languages.

Halting problem

$$HALT_{TM} = \{\langle M, w \rangle \mid M \text{ is a Turing machine}, w \text{ is a string, and } M \text{ halts on } w\}$$

We know A_{TM} is undecidable. If we could prove that $A_{TM} \leq_m HALT_{TM}$ then we could conclude that $HALT_{TM}$ is undecidable too.

Could we adapt our approach from before by tweaking the identity map?

Define $F : \Sigma^* \to \Sigma^*$ by

$$F(x) = \begin{cases} const_{out} & \text{if } x \neq \langle M, w \rangle \text{ for any Turing machine } M \text{ and string } w \text{ over the alphabet of } M \\ \langle M'_x, w \rangle & \text{if } x = \langle M, w \rangle \text{ for some Turing machine } M \text{ and string } w \text{ over the alphabet of } M. \end{cases}$$

$$0; \square, R$$
$$1; \square, R$$
$$\square; \square, R$$

$$\text{start} \longrightarrow \boxed{q0} \qquad \boxed{q_{acc}}$$

where $const_{out} = \langle \qquad \qquad , \varepsilon \rangle$ and M'_x is a Turing machine that computes like M except, if the computation of M ever were to go to a reject state, M'_x loops instead.

$$q_{acc}$$
$$\square; \square, R \qquad 0; 0, R$$
$$1; 1, R$$
$$\text{start} \longrightarrow q0 \qquad\qquad q1$$
$$0; 0, R$$
$$1; 1, R$$

$$F(\langle \qquad\qquad , \varepsilon \rangle) =$$

To use this function to prove that $A_{TM} \leq_m HALT_{TM}$, we need two claims:

Claim (1): F is computable

Claim (2): for every x, $x \in A_{TM}$ iff $F(x) \in HALT_{TM}$.

Day19

The Church-Turing thesis posits that each algorithm can be implemented by some Turing machine.

Describing algorithms (Sipser p. 185) To define a Turing machine, we could give a

- **Formal definition**: the 7-tuple of parameters including set of states, input alphabet, tape alphabet, transition function, start state, accept state, and reject state. This is the low-level programming view that models the logic computation flow in a processor.

- **Implementation-level definition**: English prose that describes the Turing machine head movements relative to contents of tape, and conditions for accepting / rejecting based on those contents. This level describes memory management and implementing data access with data structures.

  - Mention the tape or its contents (e.g. "Scan the tape from left to right until a blank is seen.")
  - Mention the tape head (e.g. "Return the tape head to the left end of the tape.")

- **High-level description** of algorithm executed by Turing machine: description of algorithm (precise sequence of instructions), without implementation details of machine. High-level descriptions of Turing machine algorithms are written as indented text within quotation marks. Stages of the algorithm are typically numbered consecutively. The first line specifies the input to the machine, which must be a string.

  - Use other Turing machines as subroutines (e.g. "Run M on w")
  - Build new machines from existing machines using previously shown results (e.g. "Given NFA A construct an NFA B such that $L(B) = \overline{L(A)}$")
  - Use previously shown conversions and constructions (e.g. "Convert regular expression R to an NFA N")

Formatted inputs to Turing machine algorithms

The input to a Turing machine is always a string. The format of the input to a Turing machine can be checked to interpret this string as representing structured data (like a csv file, the formal definition of a DFA, another Turing machine, etc.)

This string may be the encoding of some object or list of objects.

Notation: $\langle O \rangle$ is the string that encodes the object O. $\langle O_1, \ldots, O_n \rangle$ is the string that encodes the list of objects $O_1, \ldots, O_n$.

Assumption: There are algorithms (Turing machines) that can be called as subroutines to decode the string representations of common objects and interact with these objects as intended (data structures). These algorithms are able to "type-check" and string representations for different data structures are unique.

For example, since there are algorithms to answer each of the following questions, by Church-Turing thesis, there is a Turing machine that accepts exactly those strings for which the answer to the question is "yes"

- Does a string over $\{0, 1\}$ have even length?

- Does a string over $\{0, 1\}$ encode a string of ASCII characters?[1]

- Does a DFA have a specific number of states?

- Do two NFAs have any state names in common?

- Do two CFGs have the same start variable?

A **computational problem** is decidable iff language encoding its positive problem instances is decidable.

The computational problem "Does a specific DFA accept a given string?" is encoded by the language

$$\{\text{representations of DFAs } M \text{ and strings } w \text{ such that } w \in L(M)\}$$
$$=\{\langle M, w \rangle \mid M \text{ is a DFA}, w \text{ is a string}, w \in L(M)\}$$

The computational problem "Is the language generated by a CFG empty?" is encoded by the language

$$\{\text{representations of CFGs } G \text{ such that } L(G) = \emptyset\}$$
$$=\{\langle G \rangle \mid G \text{ is a CFG}, L(G) = \emptyset\}$$

The computational problem "Is the given Turing machine a decider?" is encoded by the language

$$\{\text{representations of TMs } M \text{ such that } M \text{ halts on every input}\}$$
$$=\{\langle M \rangle \mid M \text{ is a TM and for each string } w, M \text{ halts on } w\}$$

Note: writing down the language encoding a computational problem is only the first step in determining if it's recognizable, decidable, or . . .

Deciding a computational problem means building / defining a Turing machine that recognizes the language encoding the computational problem, and that is a decider.

Some classes of computational problems will help us understand the differences between the machine models we've been studying. (Sipser Section 4.1)

---

[1]An introduction to ASCII is available on the w3 tutorial here.

Acceptance problem		
...for DFA	A_{DFA}	$\{\langle B, w \rangle \mid B$ is a DFA that accepts input string $w\}$
...for NFA	A_{NFA}	$\{\langle B, w \rangle \mid B$ is a NFA that accepts input string $w\}$
...for regular expressions	A_{REX}	$\{\langle R, w \rangle \mid R$ is a regular expression that generates input string $w\}$
...for CFG	A_{CFG}	$\{\langle G, w \rangle \mid G$ is a context-free grammar that generates input string $w\}$
...for PDA	A_{PDA}	$\{\langle B, w \rangle \mid B$ is a PDA that accepts input string $w\}$

Language emptiness testing		
...for DFA	E_{DFA}	$\{\langle A \rangle \mid A$ is a DFA and $L(A) = \emptyset\}$
...for NFA	E_{NFA}	$\{\langle A \rangle \mid A$ is a NFA and $L(A) = \emptyset\}$
...for regular expressions	E_{REX}	$\{\langle R \rangle \mid R$ is a regular expression and $L(R) = \emptyset\}$
...for CFG	E_{CFG}	$\{\langle G \rangle \mid G$ is a context-free grammar and $L(G) = \emptyset\}$
...for PDA	E_{PDA}	$\{\langle A \rangle \mid A$ is a PDA and $L(A) = \emptyset\}$

Language equality testing		
...for DFA	EQ_{DFA}	$\{\langle A, B \rangle \mid A$ and B are DFAs and $L(A) = L(B)\}$
...for NFA	EQ_{NFA}	$\{\langle A, B \rangle \mid A$ and B are NFAs and $L(A) = L(B)\}$
...for regular expressions	EQ_{REX}	$\{\langle R, R' \rangle \mid R$ and R' are regular expressions and $L(R) = L(R')\}$
...for CFG	EQ_{CFG}	$\{\langle G, G' \rangle \mid G$ and G' are CFGs and $L(G) = L(G')\}$
...for PDA	EQ_{PDA}	$\{\langle A, B \rangle \mid A$ and B are PDAs and $L(A) = L(B)\}$

Example strings in A_{DFA}

Example strings in E_{DFA}

Example strings in EQ_{DFA}

> $M_1 =$ "On input $\langle M, w \rangle$, where M is a DFA and w is a string:
>
> 0. Type check encoding to check input is correct type. If not, reject.
> 1. Simulate M on input w (by keeping track of states in M, transition function of M, etc.)
> 2. If the simulation ends in an accept state of M, accept. If it ends in a non-accept state of M, reject. "

What is $L(M_1)$?

Is M_1 a decider?

Alternate description: Sometimes omit step 0 from listing and do implicit type check.

Synonyms: "Simulate", "run", "call".

True / False: $A_{REX} = A_{NFA} = A_{DFA}$

True / False: $A_{REX} \cap A_{NFA} = \emptyset$, $A_{REX} \cap A_{DFA} = \emptyset$, $A_{DFA} \cap A_{NFA} = \emptyset$

$E_{DFA} = \{\langle A \rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}$. A Turing machine that decides E_{DFA} is

$M_2 =$ "On input $\langle M \rangle$ where M is a DFA,

1. For integer $i = 1, 2, \ldots$

2.   Let s_i be the ith string over the alphabet of M (ordered in string order).

3.   Run M on input s_i.

4.   If M accepts, reject. If M rejects, increment i and keep going."

$M_3 =$ " On input $\langle M \rangle$ where M is a DFA,

1. Mark the start state of M.

2. Repeat until no new states get marked:

3.   Loop over the states of M.

4.   Mark any unmarked state that has an incoming edge from a marked state.

5. If no accept state of M is marked, _____; otherwise, _____."

To build a Turing machine that decides EQ_{DFA}, notice that

$$L_1 = L_2 \qquad \text{iff} \qquad (\, (L_1 \cap \overline{L_2}) \cup (L_2 \cap \overline{L_1}) \,) = \emptyset$$

There are no elements that are in one set and not the other

$M_{EQDFA} =$

Summary: We can use the decision procedures (Turing machines) of decidable problems as subroutines in other algorithms. For example, we have subroutines for deciding each of A_{DFA}, E_{DFA}, EQ_{DFA}. We can also use algorithms for known constructions as subroutines in other algorithms. For example, we have subroutines for: counting the number of states in a state diagram, counting the number of characters in an alphabet, converting DFA to a DFA recognizing the complement of the original language or a DFA recognizing the Kleene star of the original language, constructing a DFA or NFA from two DFA or NFA so that we have a machine recognizing the language of the union (or intersection, concatenation) of the languages of the original machines; converting regular expressions to equivalent DFA; converting DFA to equivalent regular expressions, etc.