

Week9 monday

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

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

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

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

Last time we proved that $A_{TM} \leq_m HALT_{TM}$ where

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

and since A_{TM} is undecidable, $HALT_{TM}$ is also undecidable. The function witnessing the mapping reduction mapped strings in A_{TM} to strings in $HALT_{TM}$ and strings not in A_{TM} to strings not in $HALT_{TM}$ by changing encoded Turing machines to ones that had identical computations except looped instead of rejecting.

True or False: $\overline{A_{TM}} \leq_m \overline{HALT_{TM}}$

True or False: $HALT_{TM} \leq_m A_{TM}$.

Proof: Need computable function $F : \Sigma^* \rightarrow \Sigma^*$ such that $x \in HALT_{TM}$ iff $F(x) \in A_{TM}$. Define

$F =$ “ On input x ,

1. Type-check whether $x = \langle M, w \rangle$ for some TM M and string w . If so, move to step 2; if not, output $\langle \quad \quad \quad \rangle$
2. Construct the following machine M'_x :

3. Output $\langle M'_x, w \rangle$.”

Verifying correctness: (1) Is function well-defined and computable? (2) Does it have the translation property $x \in HALT_{TM}$ iff its image is in A_{TM} ?

Input string	Output string
$\langle M, w \rangle$ where M halts on w	
$\langle M, w \rangle$ where M does not halt on w	
x not encoding any pair of TM and string	

Theorem (Sipser 5.28): If $A \leq_m B$ and B is recognizable, then A is recognizable.

Proof:

Corollary: If $A \leq_m B$ and A is unrecognizable, then B is unrecognizable.

Strategy:

- (i) To prove that a recognizable language R is undecidable, prove that $A_{TM} \leq_m R$.
- (ii) To prove that a co-recognizable language U is undecidable, prove that $\overline{A_{TM}} \leq_m U$, i.e. that $A_{TM} \leq_m \overline{U}$.

$$E_{TM} = \{\langle M \rangle \mid M \text{ is a Turing machine and } L(M) = \emptyset\}$$

Can we find algorithms to recognize

E_{TM} ?

$\overline{E_{TM}}$?

Claim: $A_{TM} \leq_m \overline{E_{TM}}$. And hence also $\overline{A_{TM}} \leq_m E_{TM}$

Proof: Need computable function $F : \Sigma^* \rightarrow \Sigma^*$ such that $x \in A_{TM}$ iff $F(x) \notin E_{TM}$. Define

$F =$ “ On input x ,

1. Type-check whether $x = \langle M, w \rangle$ for some TM M and string w . If so, move to step 2; if not, output $\langle \quad \quad \quad \rangle$
2. Construct the following machine M'_x :

3. Output $\langle M'_x \rangle$.”

Verifying correctness: (1) Is function well-defined and computable? (2) Does it have the translation property $x \in A_{TM}$ iff its image is **not** in E_{TM} ?

Input string	Output string
$\langle M, w \rangle$ where $w \in L(M)$	
$\langle M, w \rangle$ where $w \notin L(M)$	
x not encoding any pair of TM and string	

Week9 wednesday

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

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

So far:

- A_{TM} is recognizable, undecidable, and not-co-recognizable.
- $\overline{A_{TM}}$ is unrecognizable, undecidable, and co-recognizable.
- $HALT_{TM}$ is recognizable, undecidable, and not-co-recognizable.
- $\overline{HALT_{TM}}$ is unrecognizable, undecidable, and co-recognizable.
- E_{TM} is unrecognizable, undecidable, and co-recognizable.
- $\overline{E_{TM}}$ is recognizable, undecidable, and not-co-recognizable.

$$EQ_{TM} = \{ \langle M_1, M_2 \rangle \mid M_1 \text{ and } M_2 \text{ are both Turing machines and } L(M_1) = L(M_2) \}$$

Can we find algorithms to recognize

EQ_{TM} ?

$\overline{EQ_{TM}}$?

Goal: Show that EQ_{TM} is not recognizable and that $\overline{EQ_{TM}}$ is not recognizable.

Using Corollary to **Theorem 5.28**: If $A \leq_m B$ and A is unrecognizable, then B is unrecognizable, it's enough to prove that

$$\overline{HALT_{TM}} \leq_m EQ_{TM} \quad \text{aka } HALT_{TM} \leq_m \overline{EQ_{TM}}$$

$$\overline{HALT_{TM}} \leq_m \overline{EQ_{TM}} \quad \text{aka } HALT_{TM} \leq_m EQ_{TM}$$

Need computable function $F_1 : \Sigma^* \rightarrow \Sigma^*$ such that $x \in HALT_{TM}$ iff $F_1(x) \notin EQ_{TM}$.

Strategy:

Map strings $\langle M, w \rangle$ to strings $\langle M'_x,$  \rangle . This image string is not in EQ_{TM} when $L(M'_x) \neq \emptyset$.

We will build M'_x so that $L(M'_x) = \Sigma^*$ when M halts on w and $L(M'_x) = \emptyset$ when M loops on w .

Thus: when $\langle M, w \rangle \in HALT_{TM}$ it gets mapped to a string not in EQ_{TM} and when $\langle M, w \rangle \notin HALT_{TM}$ it gets mapped to a string that is in EQ_{TM} .

Define

$F_1 =$ “ On input x ,

1. Type-check whether $x = \langle M, w \rangle$ for some TM M and string w . If so, move to step 2; if not, output $\langle \rangle$
2. Construct the following machine M'_x :

3. Output $\langle M'_x,$  \rangle ”

Verifying correctness: (1) Is function well-defined and computable? (2) Does it have the translation property $x \in HALT_{TM}$ iff its image is **not** in EQ_{TM} ?

Input string	Output string
$\langle M, w \rangle$ where M halts on w	
$\langle M, w \rangle$ where M loops on w	
x not encoding any pair of TM and string	

Conclude: $HALT_{TM} \leq_m \overline{EQ_{TM}}$

Need computable function $F_2 : \Sigma^* \rightarrow \Sigma^*$ such that $x \in HALT_{TM}$ iff $F_2(x) \in EQ_{TM}$.

Strategy:

Map strings $\langle M, w \rangle$ to strings $\langle M'_x, \text{start} \rightarrow \text{q}_0 \rangle$. This image string is in EQ_{TM} when $L(M'_x) = \Sigma^*$.

We will build M'_x so that $L(M'_x) = \Sigma^*$ when M halts on w and $L(M'_x) = \emptyset$ when M loops on w .

Thus: when $\langle M, w \rangle \in HALT_{TM}$ it gets mapped to a string in EQ_{TM} and when $\langle M, w \rangle \notin HALT_{TM}$ it gets mapped to a string that is not in EQ_{TM} .

Define

$F_2 =$ “ On input x ,

1. Type-check whether $x = \langle M, w \rangle$ for some TM M and string w . If so, move to step 2; if not, output $\langle \text{start} \rightarrow \text{q}_0 \rangle$
2. Construct the following machine M'_x :

3. Output $\langle M'_x, \text{start} \rightarrow \text{q}_0 \rangle$ ”

Verifying correctness: (1) Is function well-defined and computable? (2) Does it have the translation property $x \in HALT_{TM}$ iff its image is in EQ_{TM} ?

Input string	Output string
$\langle M, w \rangle$ where M halts on w	
$\langle M, w \rangle$ where M loops on w	
x not encoding any pair of TM and string	

Conclude: $HALT_{TM} \leq_m EQ_{TM}$

Two models of computation are called **equally expressive** when every language recognizable with the first model is recognizable with the second, and vice versa.

Church-Turing Thesis (Sipser p. 183): The informal notion of algorithm is formalized completely and correctly by the formal definition of a Turing machine. In other words: all reasonably expressive models of computation are equally expressive with the standard Turing machine.

*Some examples of models that are **equally expressive** with deterministic Turing machines:*

May-stay machines The May-stay machine model is the same as the usual Turing machine model, except that on each transition, the tape head may move L, move R, or Stay.

Formally: $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ where

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$$

Claim: Turing machines and May-stay machines are equally expressive. *To prove ...*

To translate a standard TM to a may-stay machine: never use the direction S!

To translate one of the may-stay machines to standard TM: any time TM would Stay, move right then left.

Multitape Turing machine A multitape Turing machine with k tapes can be formally represented as $(Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$ where Q is the finite set of states, Σ is the input alphabet with $\sqcup \notin \Sigma$, Γ is the tape alphabet with $\Sigma \subsetneq \Gamma$, $\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R\}^k$ (where k is the number of states)

If M is a standard TM, it is a 1-tape machine.

To translate a k -tape machine to a standard TM: Use a new symbol to separate the contents of each tape and keep track of location of head with special version of each tape symbol. Sipser Theorem 3.13



FIGURE 3.14
Representing three tapes with one

Enumerators Enumerators give a different model of computation where a language is **produced, one string at a time**, rather than recognized by accepting (or not) individual strings.

Each enumerator machine has finite state control, unlimited work tape, and a printer. The computation proceeds according to transition function; at any point machine may “send” a string to the printer.

$$E = (Q, \Sigma, \Gamma, \delta, q_0, q_{print})$$

Q is the finite set of states, Σ is the output alphabet, Γ is the tape alphabet ($\Sigma \subsetneq \Gamma, \sqcup \in \Gamma \setminus \Sigma$),

$$\delta : Q \times \Gamma \times \Gamma \rightarrow Q \times \Gamma \times \Gamma \times \{L, R\} \times \{L, R\}$$

where in state q , when the working tape is scanning character x and the printer tape is scanning character y , $\delta((q, x, y)) = (q', x', y', d_w, d_p)$ means transition to control state q' , write x' on the working tape, write y' on the printer tape, move in direction d_w on the working tape, and move in direction d_p on the printer tape. The computation starts in q_0 and each time the computation enters q_{print} the string from the leftmost edge of the printer tape to the first blank cell is considered to be printed.

The language **enumerated** by E , $L(E)$, is $\{w \in \Sigma^* \mid E \text{ eventually, at finite time, prints } w\}$.

Theorem 3.21 A language is Turing-recognizable iff some enumerator enumerates it.

Proof, part 1: Assume L is enumerated by some enumerator, E , so $L = L(E)$. We'll use E in a subroutine within a high-level description of a new Turing machine that we will build to recognize L .

Goal: build Turing machine M_E with $L(M_E) = L(E)$.

Define M_E as follows: $M_E =$ “On input w ,

1. Run E . For each string x printed by E .
2. Check if $x = w$. If so, accept (and halt); otherwise, continue.”

Proof, part 2: Assume L is Turing-recognizable and there is a Turing machine M with $L = L(M)$. We'll use M in a subroutine within a high-level description of an enumerator that we will build to enumerate L .

Goal: build enumerator E_M with $L(E_M) = L(M)$.

Idea: check each string in turn to see if it is in L .

How? Run computation of M on each string. *But:* need to be careful about computations that don't halt.

Recall String order for $\Sigma = \{0, 1\}$: $s_1 = \varepsilon$, $s_2 = 0$, $s_3 = 1$, $s_4 = 00$, $s_5 = 01$, $s_6 = 10$, $s_7 = 11$, $s_8 = 000$, ...

Define E_M as follows: $E_M =$ “*ignore any input*. Repeat the following for $i = 1, 2, 3, \dots$

1. Run the computations of M on s_1, s_2, \dots, s_i for (at most) i steps each
2. For each of these i computations that accept during the (at most) i steps, print out the accepted string.”

Nondeterministic Turing machine

At any point in the computation, the nondeterministic machine may proceed according to several possibilities: $(Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$ where

$$\delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

The computation of a nondeterministic Turing machine is a tree with branching when the next step of the computation has multiple possibilities. A nondeterministic Turing machine accepts a string exactly when some branch of the computation tree enters the accept state.

Given a nondeterministic machine, we can use a 3-tape Turing machine to simulate it by doing a breadth-first search of computation tree: one tape is “read-only” input tape, one tape simulates the tape of the nondeterministic computation, and one tape tracks nondeterministic branching. Sipser page 178

Summary

Two models of computation are called **equally expressive** when every language recognizable with the first model is recognizable with the second, and vice versa.

To prove the existence of a Turing machine that decides / recognizes some language, it’s enough to construct an example using any of the equally expressive models.

But: some of the **performance** properties of these models are not equivalent.

Week8 monday

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



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.

Week8 wednesday

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

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

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 $f : \Sigma^* \rightarrow \Sigma^*$ 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)

$$f_1 : \Sigma^* \rightarrow \Sigma^* \quad f_1(x) = x0$$

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 ($\{q_0, q_{acc}, q_{rej}\}, \{0, 1\}, \{0, 1, \sqcup\}, \delta, q_0, q_{acc}, q_{rej}$) where δ is specified by the state diagram:

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

$$f_2 : \Sigma^* \rightarrow \Sigma^* \quad f_2(x) = xx$$

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.

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

where $q_{trap} \notin Q$ and

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

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

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

Making intuition 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.

Week8 friday

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

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

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: $A_{TM} \leq_m A_{TM}$

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

Halting problem

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

Define $F : \Sigma^* \rightarrow \Sigma^*$ 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

where $const_{out} = \langle \text{start} \rightarrow q0 \quad q_{acc}, \varepsilon \rangle$ 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.



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

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 $F(x) \in HALT_{TM}$.

Week7 wednesday

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 . $\langle O_1, \dots, O_n \rangle$ is the string that encodes the list of objects O_1, \dots, 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

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

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

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

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

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

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)

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

Acceptance problem		
... for DFA	A_{DFA}	$\{\langle B, w \rangle \mid B \text{ is a DFA that accepts input string } w\}$
... for NFA	A_{NFA}	$\{\langle B, w \rangle \mid B \text{ is a NFA that accepts input string } w\}$
... for regular expressions	A_{REX}	$\{\langle R, w \rangle \mid R \text{ is a regular expression that generates input string } w\}$
... for CFG	A_{CFG}	$\{\langle G, w \rangle \mid G \text{ is a context-free grammar that generates input string } w\}$
... for PDA	A_{PDA}	$\{\langle B, w \rangle \mid B \text{ is a PDA that accepts input string } w\}$
Language emptiness testing		
... for DFA	E_{DFA}	$\{\langle A \rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}$
... for NFA	E_{NFA}	$\{\langle A \rangle \mid A \text{ is a NFA and } L(A) = \emptyset\}$
... for regular expressions	E_{REX}	$\{\langle R \rangle \mid R \text{ is a regular expression and } L(R) = \emptyset\}$
... for CFG	E_{CFG}	$\{\langle G \rangle \mid G \text{ is a context-free grammar and } L(G) = \emptyset\}$
... for PDA	E_{PDA}	$\{\langle A \rangle \mid A \text{ is a PDA and } L(A) = \emptyset\}$
Language equality testing		
... for DFA	EQ_{DFA}	$\{\langle A, B \rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$
... for NFA	EQ_{NFA}	$\{\langle A, B \rangle \mid A \text{ and } B \text{ are NFAs and } L(A) = L(B)\}$
... for regular expressions	EQ_{REX}	$\{\langle R, R' \rangle \mid R \text{ and } R' \text{ are regular expressions and } L(R) = L(R')\}$
... for CFG	EQ_{CFG}	$\{\langle G, G' \rangle \mid G \text{ and } G' \text{ are CFGs and } L(G) = L(G')\}$
... for PDA	EQ_{PDA}	$\{\langle A, B \rangle \mid A \text{ and } B \text{ are PDAs and } L(A) = L(B)\}$

Example strings in A_{DFA}

Example strings in E_{DFA}

Example strings in EQ_{DFA}

Week7 friday

$M_1 =$ “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: $A_{REX} \cap A_{NFA} = \emptyset$, $A_{REX} \cap A_{DFA} = \emptyset$, $A_{DFA} \cap A_{NFA} = \emptyset$

A Turing machine that decides A_{NFA} is:

A Turing machine that decides A_{REX} is:

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

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

1. For integer $i = 1, 2, \dots$
2. Let s_i be the i th string over the alphabet of M (ordered in string order).
3. Run M on input s_i .
4. If M accepts, _____. If M rejects, increment i and keep going.”

Choose the correct option to help fill in the blank so that M_2 recognizes E_{DFA}

- A. accepts
- B. rejects
- C. loop for ever
- D. We can't fill in the blank in any way to make this work

$M_3 =$ “ 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

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

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

$M_{EQ_{DFA}} =$

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.

Week10 monday

In practice, computers (and Turing machines) don't have infinite tape, and we can't afford to wait unboundedly long for an answer. “Decidable” isn't good enough - we want “Efficiently decidable”.

For a given algorithm working on a given input, how long do we need to wait for an answer? How does the running time depend on the input in the worst-case? average-case? We expect to have to spend more time on computations with larger inputs.

A language is **recognizable** if _____

A language is **decidable** if _____

A language is **efficiently decidable** if _____

A function is **computable** if _____

A function is **efficiently computable** if _____

Definition (Sipser 7.1): For M a deterministic decider, its **running time** is the function $f : \mathbb{N} \rightarrow \mathbb{N}$ given by

$$f(n) = \max \text{ number of steps } M \text{ takes before halting, over all inputs of length } n$$

Definition (Sipser 7.7): For each function $t(n)$, the **time complexity class** $TIME(t(n))$, is defined by

$$TIME(t(n)) = \{L \mid L \text{ is decidable by a Turing machine with running time in } O(t(n))\}$$

An example of an element of $TIME(1)$ is

An example of an element of $TIME(n)$ is

Note: $TIME(1) \subseteq TIME(n) \subseteq TIME(n^2)$

Definition (Sipser 7.12) : P is the class of languages that are decidable in polynomial time on a deterministic 1-tape Turing machine

$$P = \bigcup_k TIME(n^k)$$

Theorem (Sipser 7.8): Let $t(n)$ be a function with $t(n) \geq n$. Then every $t(n)$ time deterministic multitape Turing machine has an equivalent $O(t^2(n))$ time deterministic 1-tape Turing machine.

Definitions (Sipser 7.1, 7.7, 7.12): For M a deterministic decider, its **running time** is the function $f : \mathbb{N} \rightarrow \mathbb{N}$ given by

$$f(n) = \max \text{ number of steps } M \text{ takes before halting, over all inputs of length } n$$

For each function $t(n)$, the **time complexity class** $TIME(t(n))$, is defined by

$$TIME(t(n)) = \{L \mid L \text{ is decidable by a Turing machine with running time in } O(t(n))\}$$

P is the class of languages that are decidable in polynomial time on a deterministic 1-tape Turing machine

$$P = \bigcup_k TIME(n^k)$$

Definition (Sipser 7.9): For N a nondeterministic decider. The **running time** of N is the function $f : \mathbb{N} \rightarrow \mathbb{N}$ given by

$$f(n) = \max \text{ number of steps } N \text{ takes on any branch before halting, over all inputs of length } n$$

Definition (Sipser 7.21): For each function $t(n)$, the **nondeterministic time complexity class** $NTIME(t(n))$, is defined by

$$NTIME(t(n)) = \{L \mid L \text{ is decidable by a nondeterministic Turing machine with running time in } O(t(n))\}$$

$$NP = \bigcup_k NTIME(n^k)$$

True or False: $TIME(n^2) \subseteq NTIME(n^2)$

True or False: $NTIME(n^2) \subseteq TIME(n^2)$

Every problem in NP is decidable with an exponential-time algorithm

Nondeterministic approach: guess a possible solution, verify that it works.

Brute-force (worst-case exponential time) approach: iterate over all possible solutions, for each one, check if it works.

Week10 wednesday

Examples in P

Can't use nondeterminism; Can use multiple tapes; Often need to be "more clever" than naïve / brute force approach

$$PATH = \{\langle G, s, t \rangle \mid G \text{ is digraph with } n \text{ nodes there is path from } s \text{ to } t\}$$

Use breadth first search to show in P

$$RELPRIME = \{\langle x, y \rangle \mid x \text{ and } y \text{ are relatively prime integers}\}$$

Use Euclidean Algorithm to show in P

$$L(G) = \{w \mid w \text{ is generated by } G\}$$

(where G is a context-free grammar). Use dynamic programming to show in P .

Examples in NP

"Verifiable" i.e. NP , Can be decided by a nondeterministic TM in polynomial time, best known deterministic solution may be brute-force, solution can be verified by a deterministic TM in polynomial time.

$$HAMPATH = \{\langle G, s, t \rangle \mid G \text{ is digraph with } n \text{ nodes, there is path from } s \text{ to } t \text{ that goes through every node exactly once}\}$$

$$VERTEX - COVER = \{\langle G, k \rangle \mid G \text{ is an undirected graph with } n \text{ nodes that has a } k\text{-node vertex cover}\}$$

$$CLIQUE = \{\langle G, k \rangle \mid G \text{ is an undirected graph with } n \text{ nodes that has a } k\text{-clique}\}$$

$$SAT = \{\langle X \rangle \mid X \text{ is a satisfiable Boolean formula with } n \text{ variables}\}$$

Problems in P	Problems in NP
(Membership in any) regular language	Any problem in P
(Membership in any) context-free language	
A_{DFA}	SAT
E_{DFA}	$CLIQUE$
EQ_{DFA}	$VERTEX - COVER$
$PATH$	$HAMPATH$
$RELPRIME$	\dots
\dots	

Notice: $NP \subseteq \{L \mid L \text{ is decidable}\}$ so $A_{TM} \notin NP$

Million-dollar question: Is $P = NP$?

One approach to trying to answer it is to look for *hardest* problems in NP and then (1) if we can show that there are efficient algorithms for them, then we can get efficient algorithms for all problems in NP so $P = NP$, or (2) these problems might be good candidates for showing that there are problems in NP for which there are no efficient algorithms.

Definition (Sipser 7.29) Language A is **polynomial-time mapping reducible** to language B , written $A \leq_P B$, means there is a polynomial-time computable function $f : \Sigma^* \rightarrow \Sigma^*$ such that for every $x \in \Sigma^*$

$$x \in A \quad \text{iff} \quad f(x) \in B.$$

The function f is called the polynomial time reduction of A to B .

Theorem (Sipser 7.31): If $A \leq_P B$ and $B \in P$ then $A \in P$.

Proof:

Definition (Sipser 7.34; based in Stephen Cook and Leonid Levin's work in the 1970s): A language B is **NP-complete** means (1) B is in NP **and** (2) every language A in NP is polynomial time reducible to B .

Theorem (Sipser 7.35): If B is NP-complete and $B \in P$ then $P = NP$.

Proof:

Week10 friday

NP-Complete Problems

3SAT: A literal is a Boolean variable (e.g. x) or a negated Boolean variable (e.g. \bar{x}). A Boolean formula is a **3cnf-formula** if it is a Boolean formula in conjunctive normal form (a conjunction of disjunctive clauses of literals) and each clause has three literals.

$$3SAT = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable 3cnf-formula}\}$$

Example string in $3SAT$

$$\langle (x \vee \bar{y} \vee \bar{z}) \wedge (\bar{x} \vee y \vee z) \wedge (x \vee y \vee z) \rangle$$

Example string not in $3SAT$

$$\langle (x \vee y \vee z) \wedge (x \vee y \vee \bar{z}) \wedge (x \vee \bar{y} \vee z) \wedge (x \vee \bar{y} \vee \bar{z}) \wedge (\bar{x} \vee y \vee z) \wedge (\bar{x} \vee y \vee \bar{z}) \wedge (\bar{x} \vee \bar{y} \vee z) \wedge (\bar{x} \vee \bar{y} \vee \bar{z}) \rangle$$

Cook-Levin Theorem: $3SAT$ is NP-complete.

Are there other NP-complete problems? To prove that X is NP-complete

- *From scratch*: prove X is in NP and that all NP problems are polynomial-time reducible to X .
- *Using reduction*: prove X is in NP and that a known-to-be NP-complete problem is polynomial-time reducible to X .

CLIQUE: A k -**clique** in an undirected graph is a maximally connected subgraph with k nodes.

$$CLIQUE = \{\langle G, k \rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}$$

Example string in *CLIQUE*

Example string not in *CLIQUE*

Theorem (Sipser 7.32):

$$3SAT \leq_P CLIQUE$$

Given a Boolean formula in conjunctive normal form with k clauses and three literals per clause, we will map it to a graph so that the graph has a clique if the original formula is satisfiable and the graph does not have a clique if the original formula is not satisfiable.

The graph has $3k$ vertices (one for each literal in each clause) and an edge between all vertices except

- vertices for two literals in the same clause
- vertices for literals that are negations of one another

Example: $(x \vee \bar{y} \vee \bar{z}) \wedge (\bar{x} \vee y \vee z) \wedge (x \vee y \vee z)$

Model of Computation	Class of Languages
<p>Deterministic finite automata: formal definition, how to design for a given language, how to describe language of a machine? Nondeterministic finite automata: formal definition, how to design for a given language, how to describe language of a machine? Regular expressions: formal definition, how to design for a given language, how to describe language of expression? <i>Also:</i> converting between different models.</p>	<p>Class of regular languages: what are the closure properties of this class? which languages are not in the class? using pumping lemma to prove nonregularity.</p>
<p>Push-down automata: formal definition, how to design for a given language, how to describe language of a machine? Context-free grammars: formal definition, how to design for a given language, how to describe language of a grammar?</p>	<p>Class of context-free languages: what are the closure properties of this class? which languages are not in the class?</p>
<p>Turing machines that always halt in polynomial time</p> <p>Nondeterministic Turing machines that always halt in polynomial time</p>	<p>P</p> <p>NP</p>
<p>Deciders (Turing machines that always halt): formal definition, how to design for a given language, how to describe language of a machine?</p>	<p>Class of decidable languages: what are the closure properties of this class? which languages are not in the class? using diagonalization and mapping reduction to show undecidability</p>
<p>Turing machines formal definition, how to design for a given language, how to describe language of a machine?</p>	<p>Class of recognizable languages: what are the closure properties of this class? which languages are not in the class? using closure and mapping reduction to show unrecognizability</p>

Given a language, prove it is regular

Strategy 1: construct DFA recognizing the language and prove it works.

Strategy 2: construct NFA recognizing the language and prove it works.

Strategy 3: construct regular expression recognizing the language and prove it works.

“Prove it works” means ...

Example: $L = \{w \in \{0,1\}^* \mid w \text{ has odd number of 1s or starts with } 0\}$

Using NFA

Using regular expressions

Example: Select all and only the options that result in a true statement: “To show a language A is not regular, we can...”

- a. Show A is finite
- b. Show there is a CFG generating A
- c. Show A has no pumping length
- d. Show A is undecidable

Example: What is the language generated by the CFG with rules

$$S \rightarrow aSb \mid bY \mid Ya$$

$$Y \rightarrow bY \mid Ya \mid \varepsilon$$

Example: Prove that the language $T = \{\langle M \rangle \mid M \text{ is a Turing machine and } L(M) \text{ is infinite}\}$ is undecidable.

Example: Prove that the class of decidable languages is closed under concatenation.