

## 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  $\Sigma^*$ ,

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

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

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

**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  $A_{TM} \leq_m \overline{U}$ .

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

Example string in  $E_{TM}$  is \_\_\_\_\_. Example string not in  $E_{TM}$  is \_\_\_\_\_.

$E_{TM}$  is decidable / undecidable and recognizable / unrecognizable.

$\overline{E_{TM}}$  is decidable / undecidable and recognizable / unrecognizable.

**Claim:** \_\_\_\_\_  $\leq_m \overline{E_{TM}}$ .

**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
2. Construct the following machine  $M'_x$ :

3. Output  $\langle M'_x \rangle$ .”

Verifying correctness:

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 friday

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  $\Sigma^*$ ,

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

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

Example string in  $EQ_{TM}$  is \_\_\_\_\_. Example string not in  $EQ_{TM}$  is \_\_\_\_\_.

$EQ_{TM}$  is   decidable / undecidable   and   recognizable / unrecognizable   .

$\overline{EQ_{TM}}$  is   decidable / undecidable   and   recognizable / unrecognizable   .

To prove, show that \_\_\_\_\_  $\leq_m EQ_{TM}$  and that \_\_\_\_\_  $\leq_m \overline{EQ_{TM}}$ .

Verifying correctness:

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	

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

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

*Compare to exponential time: brute-force search.*

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.

## Week8 monday

**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  $\langle D \rangle$ ?

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

**Proof, first direction:** Suppose language  $L$  is Turing-decidable. WTS that both it and its complement are Turing-recognizable.

**Proof, second direction:** Suppose language  $L$  is Turing-recognizable, and so is its complement. WTS that  $L$  is Turing-decidable.

Give an example of a **decidable** set:

Give an example of a **recognizable undecidable** set:

Give an example of an **unrecognizable** set:

**True or False:** The class of Turing-decidable languages is closed under complementation?

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

Notation: The complement of a set  $X$  is denoted with a superscript  $c$ ,  $X^c$ , or an overline,  $\overline{X}$ .

## 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  $\Sigma^*$ ,

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

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  $B$ .

## 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  $\delta$  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 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.

$$f_3 : \Sigma^* \rightarrow \Sigma^* \quad f_3(x) = \begin{cases} \varepsilon & \text{if } x \text{ is not the code of a TM} \\ \langle (Q \cup \{q_{trap}\}, \Sigma, \Gamma, \delta', q_0, q_{acc}, q_{rej}) \rangle & \text{if } x = \langle (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej}) \rangle \end{cases}$$

where  $q_{trap} \notin Q$  and

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

The function that maps strings that are not the codes of CFGs to the empty string and that maps strings that code CFGs to the code of a PDA that recognizes the language generated by the CFG.

*Other examples?*

## 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  $\Sigma^*$ ,

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

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

*Example:*  $\{0^i 1^j \mid i \geq 0, j \geq 0\} \leq_m A_{TM}$

**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.

## 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

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



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



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

# Week5 monday

These definitions are on pages 101-102.

Term	Typical symbol or Notation	Meaning
<b>Context-free grammar</b> (CFG)	$G$	$G = (V, \Sigma, R, S)$
The set of <b>variables</b>	$V$	Finite set of symbols that represent phases in production pattern
The set of <b>terminals</b>	$\Sigma$	Alphabet of symbols of strings generated by CFG $V \cap \Sigma = \emptyset$
The set of <b>rules</b>	$R$	Each rule is $A \rightarrow u$ with $A \in V$ and $u \in (V \cup \Sigma)^*$
The <b>start</b> variable	$S$	Usually on left-hand-side of first/ topmost rule
<b>Derivation</b>	$S \Rightarrow \dots \Rightarrow w$	Sequence of substitutions in a CFG (also written $S \Rightarrow^* w$ ). At each step, we can apply one rule to one occurrence of a variable in the current string by substituting that occurrence of the variable with the right-hand-side of the rule. The derivation must end when the current string has only terminals (no variables) because then there are no instances of variables to apply a rule to.
Language <b>generated</b> by the context-free grammar $G$	$L(G)$	The set of strings for which there is a derivation in $G$ . Symbolically: $\{w \in \Sigma^* \mid S \Rightarrow^* w\}$ i.e.  $\{w \in \Sigma^* \mid \text{there is derivation in } G \text{ that ends in } w\}$
<b>Context-free language</b>		A language that is the language generated by some context-free grammar

Examples of context-free grammars, derivations in those grammars, and the languages generated by those grammars

$G_1 = (\{S\}, \{0\}, R, S)$  with rules

$$S \rightarrow 0S$$

$$S \rightarrow 0$$

In  $L(G_1)$  ...

Not in  $L(G_1)$  ...

$$G_2 = (\{S\}, \{0, 1\}, R, S)$$

$$S \rightarrow 0S \mid 1S \mid \varepsilon$$

In  $L(G_2) \dots$

Not in  $L(G_2) \dots$

$(\{S, T\}, \{0, 1\}, R, S)$  with rules

$$S \rightarrow T1T1T1T$$

$$T \rightarrow 0T \mid 1T \mid \varepsilon$$

In  $L(G_3) \dots$

Not in  $L(G_3) \dots$

$G_4 = (\{A, B\}, \{0, 1\}, R, A)$  with rules

$$A \rightarrow 0A0 \mid 0A1 \mid 1A0 \mid 1A1 \mid 1$$

In  $L(G_4)$  ...

Not in  $L(G_4)$  ...

Design a CFG to generate the language  $\{a^n b^n \mid n \geq 0\}$

*Sample derivation:*



## Week5 wednesday

Warmup: Design a CFG to generate the language  $\{a^i b^j \mid j \geq i \geq 0\}$

*Sample derivation:*

Design a PDA to recognize the language  $\{a^i b^j \mid j \geq i \geq 0\}$

**Theorem 2.20:** A language is generated by some context-free grammar if and only if it is recognized by some push-down automaton.

Definition: a language is called **context-free** if it is the language generated by a context-free grammar. The class of all context-free language over a given alphabet  $\Sigma$  is called **CFL**.

Consequences:

- Quick proof that every regular language is context free
- To prove closure of the class of context-free languages under a given operation, we can choose either of two modes of proof (via CFGs or PDAs) depending on which is easier
- To fully specify a PDA we could give its 6-tuple formal definition or we could give its input alphabet, stack alphabet, and state diagram. An informal description of a PDA is a step-by-step description of how its computations would process input strings; the reader should be able to reconstruct the state diagram or formal definition precisely from such a description. The informal description of a PDA can refer to some common modules or subroutines that are computable by PDAs:
  - PDAs can “test for emptiness of stack” without providing details. *How?* We can always push a special end-of-stack symbol, \$, at the start, before processing any input, and then use this symbol as a flag.
  - PDAs can “test for end of input” without providing details. *How?* We can transform a PDA to one where accepting states are only those reachable when there are no more input symbols.

Suppose  $L_1$  and  $L_2$  are context-free languages over  $\Sigma$ . **Goal:**  $L_1 \cup L_2$  is also context-free.

*Approach 1: with PDAs*

Let  $M_1 = (Q_1, \Sigma, \Gamma_1, \delta_1, q_1, F_1)$  and  $M_2 = (Q_2, \Sigma, \Gamma_2, \delta_2, q_2, F_2)$  be PDAs with  $L(M_1) = L_1$  and  $L(M_2) = L_2$ .

Define  $M =$

*Approach 2: with CFGs*

Let  $G_1 = (V_1, \Sigma, R_1, S_1)$  and  $G_2 = (V_2, \Sigma, R_2, S_2)$  be CFGs with  $L(G_1) = L_1$  and  $L(G_2) = L_2$ .

Define  $G =$

Suppose  $L_1$  and  $L_2$  are context-free languages over  $\Sigma$ . **Goal:**  $L_1 \circ L_2$  is also context-free.

*Approach 1: with PDAs*

Let  $M_1 = (Q_1, \Sigma, \Gamma_1, \delta_1, q_1, F_1)$  and  $M_2 = (Q_2, \Sigma, \Gamma_2, \delta_2, q_2, F_2)$  be PDAs with  $L(M_1) = L_1$  and  $L(M_2) = L_2$ .

Define  $M =$

*Approach 2: with CFGs*

Let  $G_1 = (V_1, \Sigma, R_1, S_1)$  and  $G_2 = (V_2, \Sigma, R_2, S_2)$  be CFGs with  $L(G_1) = L_1$  and  $L(G_2) = L_2$ .

Define  $G =$

## Summary

Over a fixed alphabet  $\Sigma$ , a language  $L$  is **regular**

iff it is described by some regular expression  
iff it is recognized by some DFA  
iff it is recognized by some NFA

Over a fixed alphabet  $\Sigma$ , a language  $L$  is **context-free**

iff it is generated by some CFG  
iff it is recognized by some PDA

**Fact:** Every regular language is a context-free language.

**Fact:** There are context-free languages that are not nonregular.

**Fact:** There are countably many regular languages.

**Fact:** There are countably infinitely many context-free languages.

*Consequence:* Most languages are **not** context-free!

## Examples of non-context-free languages

$$\begin{aligned} &\{a^n b^n c^n \mid 0 \leq n, n \in \mathbb{Z}\} \\ &\{a^i b^j c^k \mid 0 \leq i \leq j \leq k, i \in \mathbb{Z}, j \in \mathbb{Z}, k \in \mathbb{Z}\} \\ &\{ww \mid w \in \{0, 1\}^*\} \end{aligned}$$

(Sipser Ex 2.36, Ex 2.37, 2.38)

There is a Pumping Lemma for CFL that can be used to prove a specific language is non-context-free: If  $A$  is a context-free language, there there is a number  $p$  where, if  $s$  is any string in  $A$  of length at least  $p$ , then  $s$  may be divided into five pieces  $s = uvxyz$  where (1) for each  $i \geq 0$ ,  $uv^i xy^i z \in A$ , (2)  $|uv| > 0$ , (3)  $|vxy| \leq p$ . *We will not go into the details of the proof or application of Pumping Lemma for CFLs this quarter.*

## Week5 friday

## Week4 monday

Recap so far: In DFA, the only memory available is in the states. Automata can only “remember” finitely far in the past and finitely much information, because they can have only finitely many states. If a computation path of a DFA visits the same state more than once, the machine can’t tell the difference between the first time and future times it visits this state. Thus, if a DFA accepts one long string, then it must accept (infinitely) many similar strings.

**Definition** A positive integer  $p$  is a **pumping length** of a language  $L$  over  $\Sigma$  means that, for each string  $s \in \Sigma^*$ , if  $|s| \geq p$  and  $s \in L$ , then there are strings  $x, y, z$  such that

$$s = xyz$$

and

$$|y| > 0, \quad \text{for each } i \geq 0, xy^iz \in L, \quad \text{and} \quad |xy| \leq p.$$

**Negation:** A positive integer  $p$  is **not a pumping length** of a language  $L$  over  $\Sigma$  iff

$$\exists s ( |s| \geq p \wedge s \in L \wedge \forall x \forall y \forall z ( (s = xyz \wedge |y| > 0 \wedge |xy| \leq p) \rightarrow \exists i (i \geq 0 \wedge xy^iz \notin L) ) )$$

*Informally:*

Restating **Pumping Lemma**: If  $L$  is a regular language, then it has a pumping length.

**Contrapositive:** If  $L$  has no pumping length, then it is nonregular.

The Pumping Lemma *cannot* be used to prove that a language *is* regular.

The Pumping Lemma **can** be used to prove that a language *is not* regular.

*Extra practice:* Exercise 1.49 in the book.

**Proof strategy:** To prove that a language  $L$  is **not** regular,

- Consider an arbitrary positive integer  $p$
- Prove that  $p$  is not a pumping length for  $L$
- Conclude that  $L$  does not have *any* pumping length, and therefore it is not regular.

**Example:**  $\Sigma = \{0, 1\}$ ,  $L = \{0^n 1^n \mid n \geq 0\}$ .

Fix  $p$  an arbitrary positive integer. List strings that are in  $L$  and have length greater than or equal to  $p$ :

Pick  $s =$

Suppose  $s = xyz$  with  $|xy| \leq p$  and  $|y| > 0$ .

Then when  $i =$  ,  $xy^iz =$

**Example:**  $\Sigma = \{0, 1\}$ ,  $L = \{ww^{\mathcal{R}} \mid w \in \{0, 1\}^*\}$ . Remember that the reverse of a string  $w$  is denoted  $w^{\mathcal{R}}$  and means to write  $w$  in the opposite order, if  $w = w_1 \cdots w_n$  then  $w^{\mathcal{R}} = w_n \cdots w_1$ . Note:  $\varepsilon^{\mathcal{R}} = \varepsilon$ .

Fix  $p$  an arbitrary positive integer. List strings that are in  $L$  and have length greater than or equal to  $p$ :

Pick  $s =$

Suppose  $s = xyz$  with  $|xy| \leq p$  and  $|y| > 0$ .

Then when  $i =$  ,  $xy^iz =$

**Example:**  $\Sigma = \{0, 1\}$ ,  $L = \{0^j1^k \mid j \geq k \geq 0\}$ .

Fix  $p$  an arbitrary positive integer. List strings that are in  $L$  and have length greater than or equal to  $p$ :

Pick  $s =$

Suppose  $s = xyz$  with  $|xy| \leq p$  and  $|y| > 0$ .

Then when  $i =$  ,  $xy^iz =$

**Example:**  $\Sigma = \{0, 1\}$ ,  $L = \{0^n1^m0^n \mid m, n \geq 0\}$ .

Fix  $p$  an arbitrary positive integer. List strings that are in  $L$  and have length greater than or equal to  $p$ :

Pick  $s =$

Suppose  $s = xyz$  with  $|xy| \leq p$  and  $|y| > 0$ .

Then when  $i =$  ,  $xy^iz =$



Extra practice:

Language	$s \in L$	$s \notin L$	Is the language regular or nonregular?
$\{a^n b^n \mid 0 \leq n \leq 5\}$			
$\{b^n a^n \mid n \geq 2\}$			
$\{a^m b^n \mid 0 \leq m \leq n\}$			
$\{a^m b^n \mid m \geq n + 3, n \geq 0\}$			
$\{b^m a^n \mid m \geq 1, n \geq 3\}$			
$\{w \in \{a, b\}^* \mid w = w^R\}$			
$\{ww^R \mid w \in \{a, b\}^*\}$			

## Week4 wednesday

Regular sets are not the end of the story

- Many nice / simple / important sets are not regular
- Limitation of the finite-state automaton model: Can't "count", Can only remember finitely far into the past, Can't backtrack, Must make decisions in "real-time"
- We know actual computers are more powerful than this model...

The **next** model of computation. Idea: allow some memory of unbounded size. How?

- To generalize regular expressions: **context-free grammars**
- To generalize NFA: **Pushdown automata**, which is like an NFA with access to a stack: Number of states is fixed, number of entries in stack is unbounded. At each step (1) Transition to new state based on current state, letter read, and top letter of stack, then (2) (Possibly) push or pop a letter to (or from) top of stack. Accept a string iff there is some sequence of states and some sequence of stack contents which helps the PDA processes the entire input string and ends in an accepting state.

Is there a PDA that recognizes the nonregular language  $\{0^n 1^n \mid n \geq 0\}$ ?



The PDA with state diagram above can be informally described as:

Read symbols from the input. As each 0 is read, push it onto the stack. As soon as 1s are seen, pop a 0 off the stack for each 1 read. If the stack becomes empty and we are at the end of the input string, accept the input. If the stack becomes empty and there are 1s left to read, or if 1s are finished while the stack still contains 0s, or if any 0s appear in the string following 1s, reject the input.

Trace the computation of this PDA on the input string 01.

Trace the computation of this PDA on the input string 011.

A PDA recognizing the set  $\{ \}$  can be informally described as:

Read symbols from the input. As each 0 is read, push it onto the stack. As soon as 1s are seen, pop a 0 off the stack for each 1 read. If the stack becomes empty and there is exactly one 1 left to read, read that 1 and accept the input. If the stack becomes empty and there are either zero or more than one 1s left to read, or if the 1s are finished while the stack still contains 0s, or if any 0s appear in the input following 1s, reject the input.

Modify the state diagram below to get a PDA that implements this description:



**Definition** A **pushdown automaton** (PDA) is specified by a 6-tuple  $(Q, \Sigma, \Gamma, \delta, q_0, F)$  where  $Q$  is the finite set of states,  $\Sigma$  is the input alphabet,  $\Gamma$  is the stack alphabet,

$$\delta : Q \times \Sigma_{\epsilon} \times \Gamma_{\epsilon} \rightarrow \mathcal{P}(Q \times \Gamma_{\epsilon})$$

is the transition function,  $q_0 \in Q$  is the start state,  $F \subseteq Q$  is the set of accept states.

## Week4 friday

Draw the state diagram and give the formal definition of a PDA with  $\Sigma = \Gamma$ .

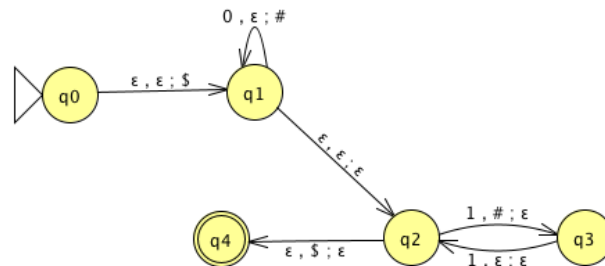
Draw the state diagram and give the formal definition of a PDA with  $\Sigma \cap \Gamma = \emptyset$ .

For the PDA state diagrams below,  $\Sigma = \{0, 1\}$ .

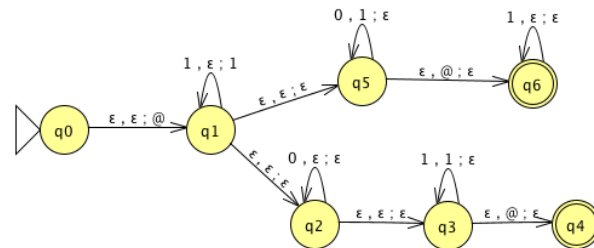
Mathematical description of language

State diagram of PDA recognizing language

$\Gamma = \{\$, \#\}$



$\Gamma = \{ @, 1 \}$



$$\{0^i 1^j 0^k \mid i, j, k \geq 0\}$$

*Note: alternate notation is to replace ; with  $\rightarrow$*

*Big picture:* PDAs were motivated by wanting to add some memory of unbounded size to NFA. How do we accomplish a similar enhancement of regular expressions to get a syntactic model that is more expressive?

DFA, NFA, PDA: Machines process one input string at a time; the computation of a machine on its input string reads the input from left to right.

Regular expressions: Syntactic descriptions of all strings that match a particular pattern; the language described by a regular expression is built up recursively according to the expression's syntax

**Context-free grammars:** Rules to produce one string at a time, adding characters from the middle, beginning, or end of the final string as the derivation proceeds.

## Week6 monday

We are ready to introduce a formal model that will capture a notion of general purpose computation.

- *Similar to DFA, NFA, PDA*: input will be an arbitrary string over a fixed alphabet.
- *Different from NFA, PDA*: machine is deterministic.
- *Different from DFA, NFA, PDA*: read-write head can move both to the left and to the right, and can extend to the right past the original input.
- *Similar to DFA, NFA, PDA*: transition function drives computation one step at a time by moving within a finite set of states, always starting at designated start state.
- *Different from DFA, NFA, PDA*: the special states for rejecting and accepting take effect immediately.

(See more details: Sipser p. 166)

Formally: a Turing machine is  $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$  where  $\delta$  is the **transition function**

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

The **computation** of  $M$  on a string  $w$  over  $\Sigma$  is:

- Read/write head starts at leftmost position on tape.
- Input string is written on  $|w|$ -many leftmost cells of tape, rest of the tape cells have the blank symbol. **Tape alphabet** is  $\Gamma$  with  $\sqcup \in \Gamma$  and  $\Sigma \subseteq \Gamma$ . The blank symbol  $\sqcup \notin \Sigma$ .
- Given current state of machine and current symbol being read at the tape head, the machine transitions to next state, writes a symbol to the current position of the tape head (overwriting existing symbol), and moves the tape head L or R (if possible).
- Computation ends **if and when** machine enters either the accept or the reject state. This is called **halting**. Note:  $q_{accept} \neq q_{reject}$ .

The **language recognized by the Turing machine**  $M$ , is  $L(M) = \{w \in \Sigma^* \mid w \text{ is accepted by } M\}$ , which is defined as

$$\{w \in \Sigma^* \mid \text{computation of } M \text{ on } w \text{ halts after entering the accept state}\}$$



Formal definition:

Sample computation:

$q0 \downarrow$						
0	0	0	□	□	□	□

The language recognized by this machine is ...

**Describing Turing machines** (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; or,
- **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.
- **High-level description:** description of algorithm (precise sequence of instructions), without implementation details of machine. As part of this description, can “call” and run another TM as a subroutine.

Fix  $\Sigma = \{0, 1\}$ ,  $\Gamma = \{0, 1, \sqcup\}$  for the Turing machines with the following state diagrams:



Example of string accepted:  
Example of string rejected:

Implementation-level description

High-level description



Example of string accepted:  
Example of string rejected:

Implementation-level description

High-level description



Example of string accepted:

Example of string rejected:

Implementation-level description

High-level description



Example of string accepted:

Example of string rejected:

Implementation-level description

High-level description



# Week6 wednesday

*Sipser Figure 3.10*

**Conventions in state diagram of TM:**  $b \rightarrow R$  label means  $b \rightarrow b, R$  and all arrows missing from diagram represent transitions with output  $(q_{reject}, \sqcup, R)$



Computation on input string 01#01

[illegible]

Implementation level description of this machine:

Zig-zag across tape to corresponding positions on either side of  $\#$  to check whether the characters in these positions agree. If they do not, or if there is no  $\#$ , reject. If they do, cross them off.

Once all symbols to the left of the # are crossed off, check for any un-crossed-off symbols to the right of #; if there are any, reject; if there aren't, accept.

The language recognized by this machine is

$$\{w\#w \mid w \in \{0,1\}^*\}$$

High-level description of this machine is

*Extra practice*

Computation on input string 01#1

[illegible]

*Recall:* 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.

A language  $L$  is **recognized by** a Turing machine  $M$  means

A Turing machine  $M$  **recognizes** a language  $L$  means

A Turing machine  $M$  is a **decider** means

A language  $L$  is **decided by** a Turing machine  $M$  means

A Turing machine  $M$  **decides** a language  $L$  means

Fix  $\Sigma = \{0, 1\}$ ,  $\Gamma = \{0, 1, \sqcup\}$  for the Turing machines with the following state diagrams:

 <p>Decider? Yes / No</p>	 <p>Decider? Yes / No</p>
 <p>Decider? Yes / No</p>	 <p>Decider? Yes / No</p>

## Week6 friday

A **Turing-recognizable** language is a set of strings that is the language recognized by some Turing machine. We also say that such languages are recognizable.

A **Turing-decidable** language is a set of strings that is the language recognized by some decider. We also say that such languages are decidable.

An **unrecognizable** language is a language that is not Turing-recognizable.

An **undecidable** language is a language that is not Turing-decidable.

**True or False:** Any decidable language is also recognizable.

**True or False:** Any recognizable language is also decidable.

**True or False:** Any undecidable language is also unrecognizable.

**True or False:** Any unrecognizable language is also undecidable.

**True or False:** The class of Turing-decidable languages is closed under complementation.

Using formal definition:

Using high-level description:

**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.

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

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

**Proof, first direction:** Suppose language  $L$  is Turing-decidable. WTS that both it and its complement are Turing-recognizable.

**Proof, second direction:** Suppose language  $L$  is Turing-recognizable, and so is its complement. WTS that  $L$  is Turing-decidable.

**Notation:** The complement of a set  $X$  is denoted with a superscript  $c$ ,  $X^c$ , or an overline,  $\overline{X}$ .

**Claim:** If two languages (over a fixed alphabet  $\Sigma$ ) are Turing-decidable, then their union is as well.

**Proof:**

**Claim:** If two languages (over a fixed alphabet  $\Sigma$ ) are Turing-recognizable, then their union is as well.

**Proof:**

## 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?<sup>1</sup>
- 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.

---

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

## Week7 friday

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

### 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  $EQ_{DFA}$

$M_1 =$  “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 simulations 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  $A$  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*

$M_{EQDFA} =$

**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.

## Week3 friday

**Definition and Theorem:** For an alphabet  $\Sigma$ , a language  $L$  over  $\Sigma$  is called **regular** exactly when  $L$  is recognized by some DFA, which happens exactly when  $L$  is recognized by some NFA, and happens exactly when  $L$  is described by some regular expression

**We saw that:** The class of regular languages is closed under complementation, union, intersection, set-wise concatenation, and Kleene star.

**Prove or Disprove:** There is some alphabet  $\Sigma$  for which there is some language recognized by an NFA but not by any DFA.

**Prove or Disprove:** There is some alphabet  $\Sigma$  for which there is some finite language not described by any regular expression over  $\Sigma$ .

**Prove or Disprove:** If a language is recognized by an NFA then the complement of this language is not recognized by any DFA.

Fix alphabet  $\Sigma$ . Is every language  $L$  over  $\Sigma$  regular?

Set	Cardinality
$\{0, 1\}$	
$\{0, 1\}^*$	
$\mathcal{P}(\{0, 1\})$	
The set of all languages over $\{0, 1\}$	
The set of all regular expressions over $\{0, 1\}$	
The set of all regular languages over $\{0, 1\}$	

Strategy: Find an **invariant** property that is true of all regular languages. When analyzing a given language, if the invariant is not true about it, then the language is not regular.

**Pumping Lemma** (Sipser Theorem 1.70): If  $A$  is a regular language, then there is a number  $p$  (a *pumping length*) where, if  $s$  is any string in  $A$  of length at least  $p$ , then  $s$  may be divided into three pieces,  $s = xyz$  such that

- $|y| > 0$
- for each  $i \geq 0$ ,  $xy^iz \in A$
- $|xy| \leq p$ .

### Proof illustration

**True or False:** A pumping length for  $A = \{0, 1\}^*$  is  $p = 5$ .

## Week10 monday

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

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

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

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

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

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

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:**  $NTIME(n^2) \subseteq DTIME(n^2)$

## 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.*

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

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

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

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



## 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.

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

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.

## Week10 wednesday

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:

**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 strings in  $3SAT$

Example strings not in  $3SAT$

**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 strings in  $CLIQUE$

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

# Week10 friday

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