Introduction to The Theory of Computation

→ Much of computer science is about solving a problem using an efficient algorithm. In contrast, our main question throughout this first chapter is — which problems cannot be solved by an efficient algorithm?

→ The key topic of this course: The limits of using algorithms to solve problems.

What are the 3 central areas of the theory of computation?

→ Automata, computability, and complexity

→ They are all linked by the question: What are the fundamental capabilities & limitations of computers?

→ Each area answers this question differently

What is complexity theory?

→ refers to figuring out the complexity of an algorithm, and quantifying the resources required to solve computational problems.

→ central question: What makes some problems computationally hard, and others easy?

→ the objective: to classify problems as easy ones and hard ones

What is computability theory?

→ deals with what can & cannot be computed on a particular computing model

→ the objective: to classify problems by those that are solvable & those that are not

What is automata theory?

→ deals with the definitions and properties of mathematical models of computation.

→ there are several models of computation that we will learn about within this theory. For example:

- "finite automation" model — used in text processing, compilers, and hardware design.
- "context-free grammar" model — used in programming langs and artificial intelligence.

→ Concepts in the lecture notes but not the text:

- "trap" or "dead" state
- Thm: A reg ⇒ $\bar{A}$ reg ... meaning of $\bar{\ }$
- $A \cap B$ ... what is $\cap$
- * Non regular $\{0^n 1^n \mid n \geq 0\}$ $= \{\varepsilon, 01, 0011, 000111, \dots\}$
- root of computation tree

Part 1: Automata and Languages

Ch 1: Regular Languages

1.1 Finite Automata

What is a "computational model"?

→ An "idealized computer" used to help us understand what a computer is and how to theorize about them.

What is the finite automation?

→ Also called the "finite state machine."

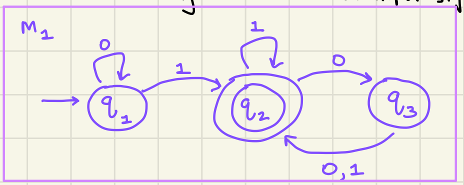
→ the simplest computational model

→ A good model for computers with an extremely limited amount of memory.

→ Ex: an automatic door (like at grocery stores)!

- a computer that has just a single bit of memory, capable of recording which of 2 states the controller is in — OPEN or CLOSED — as well as receiving a limited no. of input signals (aka people arriving)

Example of an (abstract) finite automation?



← a state diagram of M_1

- q_1: represents the **start state** (b/c arrow pointing at it from nowhere)
- q_2: represents the **accept state** (indicated by the 2 circles)
- q_3: a third state
- the arrows are called **transitions**

How does this automation (M_1) work?

1. receives a binary input string (of 0s and 1s)
2. processes the symbols one by one from left to right, moving itself from one state to another along the transition that has that symbol (aka number) as its label
3. After reading the last symbol, M_1 produces its **output**:
   - if the machine ends in an accept state (aka q_2), the output is accept
   - else, the output is reject

Example?

→ Feed the input 1101 to the machine:

1. Start in state q_1
2. Read 1, follow transition from q_1 to q_2
3. Read 1, go from q_2 to q_2
4. Read 0, go from q_2 to q_3
5. Read 1, go from q_3 to q_2
6. ACCEPT b/c M_1 is in an accept state at q_2

What pattern can be revealed regarding the machine?

→ experimenting with a variety of input strings reveals that M_1 accepts any string that either a) ends with a 1, or b) ends with an even number of 0s!

What are some rules about finite automata?

→ They <u>are</u> allowed to have <u>0</u> accept states (don't need to have one)

→ They must have **exactly** one transition exiting every state for each possible input symbol.

What is a "tuple"?

→ A list of elements. for ex, a 5-tuple is a list of 5 elements.

What is the formal definition of a finite automation?

→ A **deterministic finite automaton (DFA)** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a **finite** set, called the states
2. Σ is a **finite** set, called the alphabet
   - represents the "input alphabet"
   - indicates the allowed input symbols
   - for ex, with binary string inputs, $\Sigma = \{0, 1\}$
3. $\delta : Q \times \Sigma \longrightarrow Q$ is the transition function
4. $q_0 \in Q$ is the start state
5. $F \subseteq Q$ is the **set of** accept states.

What is some relevant info on set notation? (RECALL: COMP283)

→ $\in$ = "is an element of"

→ $\subseteq$ = "is a subset of"

→ let set $A = \{a_1, a_2\}$ and set $B = \{b_1, b_2, b_3\}$

→ the notation $A \times B$ describes the <u>cartesian product</u> of set A and set B, resulting in **a set of all ordered pairs** where the first element is a member of A and the second is a member of B

- $A \times B = \{(a_1, b_1), (a_1, b_2), (a_1, b_3), (a_2, b_1), (a_2, b_2), (a_2, b_3)\}$

→ the notation $f : D \rightarrow R$ describes a function f with a domain D that "maps" onto a range R

So what does "$\delta : Q \times \Sigma \rightarrow Q$" mean?

→ describes the **transition function** as a mapping that begins with (aka, has a domain of) the **cartesian** set of all possible combinations of states (Q) and input symbols (Σ), where each combination maps to/results in some state Q

→ **transition functions are used to define the rules of moving.**

How can we denote specific rules of transition functions?

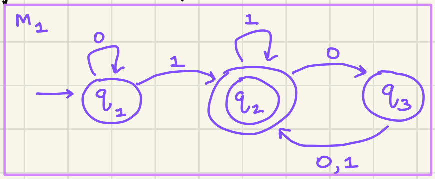
→ ex: $x \xrightarrow{1} y$

- if the automation is in state x when it reads an input symbol of <u>1</u>, it moves to state y
- We can indicate this same sentiment by saying $\delta(x, 1) = y$

→ $\delta(q \in Q, e \in \Sigma) = q \in Q$

How do we use the formal definition to describe individual finite automata?

→ Lets go back to example M_1:



← state diagram

→ formal description of M_1:

$M_1 = (Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0, 1\}$,
3. δ is described as

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

,

4. q_1 is the start state , and
5. $F = \{q_2\}$.

→ The state diagram & formal definition of a given machine contain the same information, just in different forms.

What is a language?

→ a set of strings.

→ if A is the set of all strings that machine M accepts, then we say that A is the language of machine M – denoted $L(M) = A$

- we say this as "M recognizes A"

→ A machine may accept several strings, but it always recognizes only 1 language.

→ If a machine accepts no strings, it still recognizes a language! Namely, the empty language $\emptyset$

How can we discuss the language of an individual finite automation?

→ example M_1: let

$A = \{w \mid w$ contains at least one 1 and an even number of 0s follows the last 1$\}$

(notation: "A is the set of all elements w such that w contains at least... follows the last 1.")

→ Then $L(M_1) = A$ ... also known as "M_1 recognizes A."

CLARIFY: What does "accept" mean in this context?

→ "accept" is a verb used to describe a machine's relationship to a string of input symbols.

- A machine "accepts" a string if the machine ends in one of its accept states (2 circles) after reading every symbol in the string.

CLARIFY: What does "recognize" mean in this context?

→ "recognize" is a verb used to describe a machine's relationship to a language. (a "language" being some specific set of strings, usually defined by some rules)

- A machine "recognizes" a language if it "accepts" every element of the language set (aka every string).

What is the formal definition of a finite automaton's computation?

→ Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automation and let $w = w_1 w_2 \ldots w_n$ be a string where each w_i is a member of the alphabet Σ.

(so "w" represents any possible string of input symbols, like "010110")

Then, M accepts w if a sequence of states $r_0, r_1, \ldots, r_n$ in Q exists with the following 3 conditions:

1. $r_0 = q_0$

(that the machine starts in the start state, q_0)

2. $\delta(r_i, w_{i+1}) = r_{i+1}$, for $i = 0, \ldots, n-1$

(that the machine goes from state to state according to the transition function), and

3. $r_n \in F$

(that the machine accepts its input if it ends up in an "ACCEPT" state)

We say that M recognizes language A if

$A = \{ w \mid M \text{ accepts } w \}$

What is a regular language?

→ A language is called a regular language if there exists a DFA that recognizes it.

— The Regular Operations —

What are the regular operations?

→ The three "operations on languages", used to study properties of the regular languages.

→ ANALOGY: if in arithmetic, the basic objects are numbers and the tools are operations for manipulating them — e.g. +, ×, −, ÷ etc — then in the theory of computation,

- the 'objects' are languages, and
- the 'tools' are the regular operations — operations specifically designed for manipulating them.

→ There are 3 regular operations: union, concatenation, and star.

Let A and B be languages.

- Union: $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$
- Concatenation: $A \circ B = \{xy \mid x \in A \text{ and } y \in B\}$
- Star: $A^* = \{x_1 x_2 \ldots x_k \mid k \geq 0 \text{ and each } x_i \in A\}$.

What is the union operation?
→ takes all of the strings, in both A and B, and lumps them together into one language.

What is the concatenation operation?
→ attaches a string from A in front of a string from B in all possible ways to get the strings in the new language.

What is the star operation?
→ A unary operation (applying to just one language), unlike the other 2 which are binary operations.
→ Works by attaching any number of strings in A together to get a string in the new language.
→ the empty string ε is always a member of A^* no matter what (since "any number of" includes zero).

Example for understanding the regular operations?
→ let Σ = the standard 26 letter alphabet $\{a, b, \ldots z\}$
if $A = \{\text{good}, \text{bad}\}$ and $B = \{\text{boy}, \text{girl}\}$, then

- $A \cup B = \{\text{good}, \text{bad}, \text{boy}, \text{girl}\}$
- $A \circ B = \{\text{goodboy}, \text{goodgirl}, \text{badboy}, \text{badgirl}\}$
- $A^* = \{\varepsilon, \text{good}, \text{bad}, \text{goodgood}, \text{goodbad}, \text{badgood}, \text{badbad}, \text{goodgoodgood}, \text{goodbadbad}, \text{goodbadgood}, \ldots\}$

What is a "class"?
→ basically a set whose elements are themselves sets

What does it mean if a set/class is "closed"?
→ A collection of objects is **closed** under some operation if applying that operation to members of the collection returns an object still in the collection.

What is a fundamental fact about the regular operations?
→ The class/collection of all regular languages is closed under all three of the regular operations.

- Aka, if A and B are regular languages, then so are $A \cup B$, $A \circ B$, and A^* (and B^*)

→ There are proofs in the textbook proving this for all 3 operations.

Part 1: Automata and Languages

Ch 1: Regular Languages

1.2 Nondeterminism

What is deterministic computation?

→ When every step of the computation follows in exactly one, unique way from the previous step

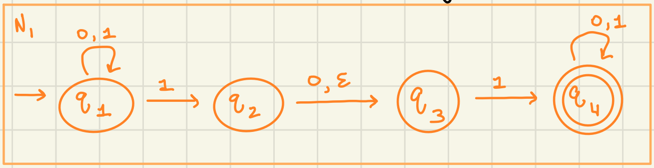
→ When a DFA (deterministic finite automaton) reads the next symbol, we know exactly what the next state will be – it is determined.

→ Everything we looked at in the prev. section were DFAs.

What is different about a nondeterministic machine?

→ in nondeterminstic finite automatons (NFAs), several choices may exist for the next state at any point.

Example of an NFA?



What makes N_1 different from a DFA?

→ Every state of a DFA must have exactly 1 exit transition arrow for each input symbol, but in an NFA a state may have zero, one, or many exiting arrows for each alphabet symbol.

- q_1 has 2 exiting arrows for the input 1
- q_2 has no arrows for 1

→ DFA – labels on arrows are symbols from the alphabet. NFAs can have arrows with the label ε

So How do NFAs compute?

→ When the NFA arrives at a state with multiple ways to proceed (like if we were at q_1 and the next input symbol is a 1, we can either stay in q_1 or move to q_2), the machine splits into multiple copies of itself and then follows all of the possibilities in parallel.

- each copy of the machine takes one of the possible paths & then continues on reading the input
- Every time there are choices, the machine "splits" again

→ NFAs are like a parallel computation where multiple independent "threads" can be running concurrently.

What happens when the NFA arrives on a state that has ε on an exit arrow?

→ Similar: without/before reading any further input, the machine splits into at least two but possibly more copies — one that stays at the current state, and one following each of the exiting ε-labeled arrows.

- For ex, when it arrives at q_2 and the next symbol is a 0, N_1 splits into 2 copies befor reading the next symbol — one that stays at q_2 and one that advances to q_3

What does it mean for a copy (of the NFA) to "die"?

→ If the next input symbol doesn't appear on <u>any</u> of the arrows exiting the state occupied by a copy of the machine, then that copy "dies" along with the branch of computation associated with it.

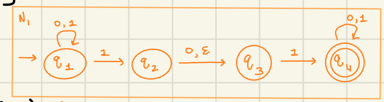
- as in, that copy will no longer read & react to input symbols
- For ex, if a copy of N_1 is in q_3 & the next symbol is a 0, that copy <u>dies</u> permanently.

How do NFAs arrive at an output (aka ACCEPT or REJECT)?

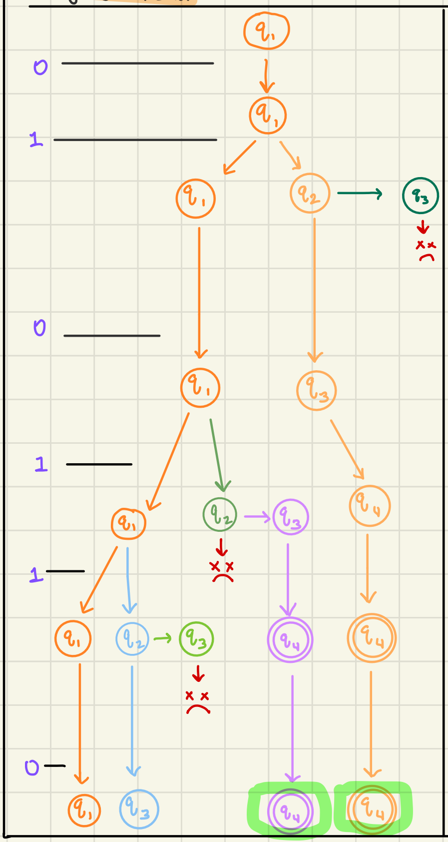
→ Of all of the copies & their paths, if **any one copy of the machine ends in the accept state, then the NFA accepts the input string!**

Example of N_1 as a computation tree?

→ One way to reason about/understand an NFA is to think about a "tree" of possibilities, with branches corresponding to points at which the machine has multiple choices.



→ Consider the computation of N_1 on input 010110 :

Symbol read	Outcome
	
0	• Starting at state q_1 and reading a 0, N_1 has only one possible outcome -- back to q_1
1	• The machine splits to follow each choice: one copy in q_1, and one moving to q_2 • An ε arrow is exiting q_2, meaning that immediately upon entering the state, the machine splits again; → One copy remaining in the current state (q_2) → One copy following the ε arrow (moving to state q_3)
0	• Copy on q_1 stays at q_1 (see the state diagram if confused) • q_2 moves to q_3 • Since there is no transition arrow for 0 exiting q_3, the copy on state q_3 now <u>dies</u>.
1	• Just like before, the copy on q_1 splits: one copy staying at q_1 & one moving to q_2 • Just like before, the q_2 copy splits due to the ε exiting arrow. • The copy on q_3 moves to q_4
1	• Copy on q_2 dies since no exiting arrow for symbol 1 • Copy on q_1 results in copies on q_1, q_2, q_3 • Copy on q_3 moves to q_4 • Copy on q_4 stays at q_4
0	• Copy on q_1 stays at q_1 • Copy on q_2 moves to q_3 • Copy on q_3 dies • Copy on q_4 stays at q_4 (x2)

→ Since at least 1 copy ended on an accept state (q_4), we can say that N_1 accepts the substring 010110

→ By continuing to experiment, we see that $L(N_1) = \{w \mid 101 \in w \mid 11 \in w\}$ (N_1 accepts all strings that contain either 101 or 11 as a substring)

What is the fundamental difference between NFAs and DFAs?

→ The transition function

→ In a DFA, the transition function takes a state & an input symbol as its input, and produces **the next state** ($\delta: Q \times \Sigma \to Q$)

→ In an NFA, the transition function takes a state & an input symbol or the empty string as its input, and produces **the set of possible next states**!

What is some relevant notation needed to define an NFA?

→ For any set Q, we write $P(Q)$ to be the **power set of** Q, meaning the collection of all possible subsets of Q

- for ex, if $A = \{1,2,3\}$,
  $P(A) = \{\{\}, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,2,3\}, \{1,3\}, \{2,3\}\}$

→ For any alphabet Σ, we write Σ_ε to be $\Sigma \cup \{\varepsilon\}$ (the alphabet AND (aka union aka "$\cup$") the empty string ε

What is the formal definition of an NFA?

A nondeterministic finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

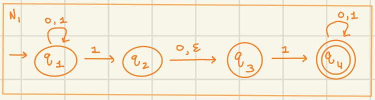
1. Q is a **finite** set of states
2. Σ is a **finite** alphabet
3. $\delta: Q \times \Sigma_\varepsilon \longrightarrow P(Q)$ is the transition function
4. $q_0 \in Q$ is the start state
5. $F \subseteq Q$ is the **set of** accept states.

What is the meaning of the transition function?

→ $\delta: Q \times \Sigma_\varepsilon \longrightarrow P(Q)$, where δ takes an input of the cartesian set of all possible combos of states (Q) and input symbols plus the empty string (Σ_ε), and produces the power set of Q, aka the set of all possible **next states**.

What is the formal description of N_1?

→ $N_1 = (Q, \Sigma, \delta, q_1, F)$, where



1. $Q = \{q_1, q_2, q_3, q_4\}$
2. $\Sigma = \{0, 1\}$
3. δ is given as

	0	1	ε
q_1	$\{q_1\}$	$\{q_1, q_2\}$	$\emptyset$
q_2	$\{q_3\}$	$\emptyset$	$\{q_3\}$
q_3	$\emptyset$	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$	$\emptyset$

4. q_1 is the start state, and
5. $F = \{q_4\}$

What is the formal definition of the computation of an NFA?

→ Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA, and w be a string over the alphabet Σ

(we can write w as $w = y_1 y_2 \cdots y_m$, where each y_i is a member of Σ_ε)

Then, we say that N accepts w if a sequence of states $r_0, r_1, \ldots, r_m$ exists in Q with the following 3 conditions:

1. $r_0 = q_0$

   (that the machine begins in the start state)

2. $r_{i+1} \in \delta(r_i, y_{i+1})$, for $i = 0, \ldots, m-1$, and

3. $r_m \in F$

   (that the last state, r_m, is an accept state)

What does the condition 2 statement mean?

→ That when the NFA N is in state r_i and reads the input symbol y_{i+1} (aka, the next symbol after the one that caused N to be in r_i), the state r_{i+1} is one of the allowable next states

NFAs versus DFAs

What does it mean for two machines to be "equivalent"?

→ If they recognize the same language.

→ Every NFA has an equivalent DFA. – Proven theorem

What is the relationship between NFAs and DFAs?

→ An NFA is like a fancier version of a DFA. However, NFAs are not able to recognize a larger class/scope/set of languages than DFAs (surprisingly).

→ DFAs and NFAs recognize the same class of languages.

→ Every NFA can be converted into an equivalent DFA – and constructing NFAs is usually easier than directly constructing NFAs.

- Describing an NFA for a given language can be easier than describing a DFA for it.

→ The equivalent DFA may have many more states – if an NFA has k states, then it has 2^k subsets of states.

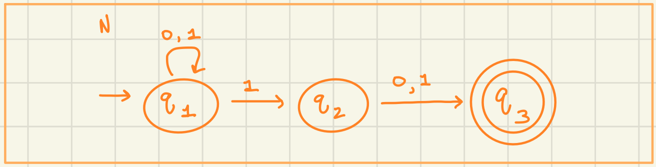
- The DFA simulating the NFA will thus have 2^k states.

What does this theorem mean for regular languages?

→ A language is regular if and only if there exists some NFA that recognizes it.

Converting an NFA into a DFA

→ Example NFA N :



→ the language that N recognizes: $L(N) = \{w \mid w$ has a 1 in the second-to-last position$\}$

→ let $N = (Q, \Sigma, \delta, q_0, F)$ represent the NFA above.

→ let $M = (Q', \Sigma, \delta', q_0', F')$ represent the DFA that we will construct from it.

What will be the states of our DFA?

→ RECALL that the transition function for an NFA returns some subset of Q (aka some subset of states). The set that encapsulates every possible subset of Q is $P(Q)$.

→ The states of our DFA M will be all of the elements of $P(Q)$!

- e.g., each state of M represents one subset of the states of N.

→

→ M works by having each of its states represent the states that "have a copy" of the NFA.

- for ex, if we run M and N on the same input, if at some point M is in state "q_1, q_2", then at the exact same point, N currently has copies in states q_1 and q_2

OK so how do we actually create the equivalent DFA?

1. Make a list of all of the DFA M's states; e.g., all the subsets of the set of states in N.

$$Q' = \{ \emptyset, \{q_1\}, \{q_2\}, \{q_3\}, \{q_1, q_2\} \{q_1, q_3\}, \{q_2, q_3\}, \{q_1, q_2, q_3\} \}$$

2. Make the start state of your DFA the same as the start state of the NFA.

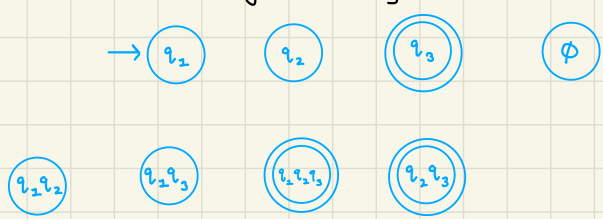
$$q_0' = \{q_1\}$$

3. For the accept state(s) K of NFA N, mark every state of M which contains K in its subset, as an accept state of M.

$$F' = \{q_3\}, \{q_1, q_3\}, \{q_2, q_3\} \{q_1, q_2, q_3\}$$

(Because q_3 is the accept state of N).

4. Draw the outline of M's state diagram, indicating start & accept states.

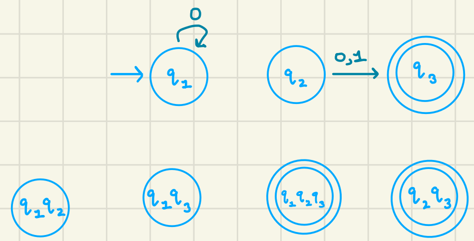


5. For states of M which are equivalent to states in N (aka "subsets" of size 1):

- For each possible input symbol, draw **one** arrow leaving the state, referring to N's state diagram to figure out where they should point.

→ **Nondeterministic Input:** if for input symbol f, the state has exactly one arrow exiting it (in N) – aka nothing NFA-peculiar is happening –, then duplicate the transition arrow onto your M state diagram.

- For ex, in N, q_1 has 1 arrow for symbol "0" and q_2 has 1 arrow for both symbols:



Ch 1: Regular Languages

1.3 Regular Expressions

What is a regular expression?

→ expressions describing languages!

→ ANALOGY: In arithmetic, we use operations (like + and −) to build up expressions whose values equate to numbers;

$(5+3) \times 4 \quad = 32$

- In Models of Comp, we can build expressions – out of the regular operations – whose values equate to languages.

Example of a regular expression?

$(0 \cup 1)\, 0^*$

→ "0" and "1" are shorthand for the sets $\{0\}$ and $\{1\}$

- So the expression "$0 \cup 1$" alone results in the language $\{0, 1\}$
- RECALL $A = \{\text{good}, \text{bad}\}$; $B = \{\text{boy}, \text{girl}\}$; $A \cup B = \{\text{good}, \text{bad}, \text{boy}, \text{girl}\}$

→ "0^*" then means $\{0\}^*$, and its value is the language consiting of all strings containing any number of 0s (including ε)

- RECALL A^* "Works by attaching any number of strings in A together to get a string in the new language."

What is implicitly present in this expression?

→ The concatenation symbol, $\circ$

→ Just like how the multiplication symbol $\times$ is usually implicit in arithmetic expressions, especially with parentheses.

→ So $(0 \cup 1)\, 0^*$ is actually shorthand for $(0 \cup 1) \circ 0^*$

- RECALL $A \circ B = \{\text{goodboy}, \text{goodgirl}, \text{badboy}, \text{badgirl}\}$
- attaches the strings from the 2 parts, $\{0, 1\}$ and $\{0\}^*$ in all possible ways — this forms the value of the entire expression!

→ Thus, we can describe the expression $(0 \cup 1)\, 0^*$ as the language of all strings that

- start with a 0 or a 1
- proceed to contain any number of 0s (or none)
- for ex, $\{0\}$, $\{1\}$, $\{0000\}$, $\{100\}$

What is another ex of a regular expression?

→ $(0 \cup 1)^* = \{0, 1\}^*$ = the language consisting of all possible strings of any number of 0s and 1s

How can we use Σ for regular expression shorthand?

→ if Σ (the alphabet of a given machine) is $\{0, 1\}$, for ex, then we can write Σ as shorthand for the expression $(0 \cup 1)$

→ More generally, for any alphabet Σ, the regular expression Σ describes the language consisting of all strings of length 1 over that alphabet.

- Σ^* describes the language of all strings (of any length) over that alphabet.
- for ex, if $\Sigma = \{0, 1\}$, Σ^* contains $\{0, 1, 01, 1011, 00, 10011, \ldots\}$

What is the order of regular operations?

→ Unless parentheses change the usual order, the order of operation precedence is

1) star operation ($*$)
2) concatenation ($\circ$)
3) union operation ($\cup$)

→ Similar to PEMDAS with arithmetic.

What is the formal definition of a regular expression?

→ Say that R is a regular expression if R is

1. a for some a in the alphabet Σ
   (the regular expression "a" represents the language $\{a\}$)
2. ε
   (the regular expression "ε" represents the language $\{\varepsilon\}$)
3. $\emptyset$
   (the regular expression "$\emptyset$" represents the empty language)
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, OR
6. (R_1^*), where R_1 is a regular expression.

$*$ We write $L(R)$ to be the language described by the regular expression R.

What is the difference between ε and $\emptyset$?

→ The expression ε represents a language containing a single string—namely, **the empty string**

→ The expression $\emptyset$ represents a language that doesn't contain any strings.

What is the relationship between regular expressions and finite automata?

→ The two are equivalent in their descriptive power.

→ Any regular expression can be converted into a finite automaton that recognizes the language it describes, and vice versa!

→ Theorem 1.5: A language is regular if and only if there exists some regular expression that describes it.

More examples of regular expressions?

→ Assuming $\Sigma = \{1, 0\}$,

R	$L(R)$
$0^* 1 \circ 0^*$	$\{w \mid w$ contains exactly one $1\}$
$\Sigma^* 1 \Sigma^*$ (aka $(0 \cup 1)^* 1 (0 \cup 1)^*$)	$\{w \mid w$ contains at least one $1\}$
$1^* (0 1 1^*)^*$	$\{w \mid$ every 0 in w is followed by at least one $1\}$
$(\Sigma \Sigma)^*$	$\{w \mid w$ has an even length$\}$
$(0 \cup \varepsilon) \circ (1 \cup \varepsilon)$	$\{01, 0, 1, \varepsilon\}$ -- "0ε" is equivalent to "0"

Part 1: Automata and Languages

Ch 1: Regular Languages

1.4 Nonregular Languages

What is a nonregular language?

→ A language that cannot be recognized by any finite automaton.

→ For example, the language

$$A = \{0^n 1^n \mid n \geq 0\}$$

is irregular, because the machine seems to need to remember/count how many 0s have been seen so far, as it reads the input.

- Since the no. of 0s isn't limited, the machine will have to keep track of an unlimited no. of possibilities, which it can't do b/c it doesn't have unbounded memory!

How do you prove that a language is nonregular?

→ There are 2 techniques:

a) fooling sets

b) the pumping lemma

What is the pumping lemma?

→ A theorem that states that all regular languages contain a special property:

that all strings in that language can be "pumped" if they are at least as long as a certain special value called the pumping length.

→ If we can show that a language does not have this property, then we can garuantee/prove that it is not regular.

What does it mean that a language can be "pumped"?

→ That each string in the lang contains a section which can be repeated any number of times, with the resulting string remaining a correct component/element of that language.

What is the formal pumping lemma theorem?

→ if A is a regular language, then there is a number p (the pumping length) where, if s = any string in A of length $\geq p$ then s may be divided into 3 pieces,

$s = xyz$, while satisfying the following conditions:

1. for each $i \geq 0$, $xy^iz \in A$

   (• y^i = i copies of y concatenated together. For ex,
   $0^5 = 00000$
   • also remember that $y^0 = \varepsilon$)

2. $|y| > 0$

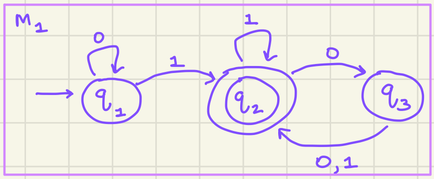
   ($|y| \approx$ the length of the substring y), and

3. $|xy| \leq p$

   (the pieces x and y together have a length, at most, of p)

Wait so what is the pumping lemma actually saying?

→ Lets return to the DFA M_1 as an example:



The regular language $A = \{ w \mid w$ contains at least one 1 and an even number of 0s follows the last 1$\}$ is recognized by M_1.

→ If we assign the "pumping length" p = the no. of states in $M_1 = 3$, then we **know** that the length of the "sequence of states" that M_1 passes through whilst reading a string of length p (3) will be $p+1$, because we also consider the start state (befor the first symbol is read)!

→ And thus, if M_1 only has 3 states but passes through 4 while reading s (where $|s| = 3$), then that must mean that the sequence contains a repeated state.

- For ex, the string 101 (which is accepted by M_1)

  1 0 1

  q_1 q_2 q_3 q_2

  this is the sequence of states, of length 4. State q_2 is repeated.

→ For the splitting of s into substrings, xyz, we let y equal the sequence of input symbols that takes the machine from one state **back to that state**.

- NOTE: as long as the lemma conditions are satisfied (aka that $|xy| \leq p$), it doesn't matter how long each of x, y, & z are — they don't have to be of even length, and either x or z may have a length of 0.

→ for $s = 101$, we let $y = 01$, since that substring takes M_1 from q_2 back to q_2

- then $x = 1$ and $z = \varepsilon$ (nothing)

→ In this example, we can create new strings repeating portion y as many times as we want, and those strings will still be accepted by M_1!

- y effectively doesn't move the machine at all (brings it back to where it started), which is why we can repeat it endlessly. See condition 1: $xy^iz \in A$

  101010101 ✓✓

  10101 ✓✓

  1 ✓✓ (xy^0z, aka xz)

(Notes)

→ p doesn't have to be the # of states (if given a DFA), that was just an ex.

→ $|s| \geq p$; just chose to make it length p in this example.

How do we use the pumping lemma to prove that a language is nonregular?

→ All we have to do is find just one string s (which is accepted by B) for which one of the 3 lemma conditions isn't satisfied.

→ To prove that a language B is nonregular:

1. First assume that B is regular, in order to obtain a contradiction.
2. Use the pumping lemma to "guarantee" the existence of a pumping length p such that all strings in B of length $\geq p$ can be "pumped". (basically state this false guarantee so that you can then contradict it).
3. Find a specific string s in B where $|s| \geq p$, but that cannot be pumped.
4. Demonstrate that s cannot be pumped by considering all ways of dividing s into x, y, and z (taking cond. 3 of the pumping lemma into account if convenient)
5. For each of these "potential s" divisions, find a value i such that $xy^iz \notin B$

What is the significance of the pumping lemma?

→ That it can prove some languages to be nonregular – that is, that we have found a **problem** which cannot be solved by an **algorithm**!

- Recall the main question that this course seeks to answer (pg 6/ first pg of notes)

– A summary of Chapter 1 –

→ 2 different, though ultimately equivalent in their scope, methods of describing languages (more specifically, **regular languages**):

1. **Finite automata** – DFAs and NFAs
2. **Regular Expressions**

→ While many languages can be described with these, some simple ones cannot!

- For ex, $\{0^n1^n \mid n \geq 0\}$

→ For every NFA there exists an equivalent DFA.

Ari Kumar

Due February 2, 2024

COMP 455-001

Homework 1

1.

a) $\Sigma\Sigma\Sigma \cup \Sigma\Sigma \cup \Sigma \cup \varepsilon$

b) $\Sigma^* 000 \Sigma^*$ [aka $\Sigma^* \circ (000) \circ \Sigma^*$]

c) $1^* ((001)(1^*))^*$

2. $\Sigma = \{1\}$ Describe a DFA that recognizes $A = \{1^k \mid k \text{ is a multiple of } 3\}$:

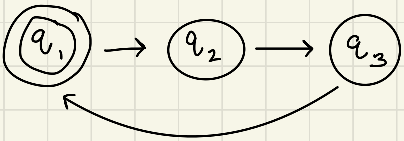
Let $M_1 = \{Q, \Sigma, \delta, q_1, F\}$, where

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{1\}$
3. δ is described as

	1
q_1	q_2
q_2	q_3
q_3	q_1

4. q_1 is the start state, and
5. $F = \{q_1\}$

State Diagram of M_1:



Examples of accepted inputs:

{ } -- aka 1^0, assuming that $0 \in k$

{111}

{111111}

Examples of rejected inputs:

{1}

{11}

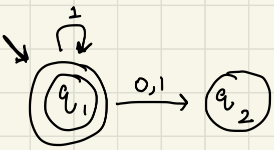
{1111}

3. Let $N_1 = \{Q, \Sigma, \delta, q_1, F\}$, where
1. $Q = \{q_1, q_2\}$
2. $\Sigma = \{0, 1\}$
3. δ is given as

	0	1
q_1	$\{q_2\}$	$\{q_1, q_2\}$
q_2	$\emptyset$	$\emptyset$

4. q_1 is the start state, and
5. $F = \{q_1\}$

State Diagram of N_1:



- N_1 satisfies the rules defining an NFA because it contains a state with several exiting arrows for an input symbol (q_1); as well as a state with no exit arrows for each symbol (q_2).
- We can describe the language A recognized by N_1, $L(N_1) = A$, as

  $A = \{ w \mid w$ doesn't contain any zeroes $\}$, where w is a string of input.

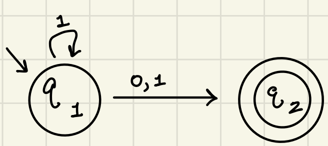
  - N_1 begins in the accept state and remains there until it reads a zero, meaning that it will accept a string containing any no. of 1s, as well as the empty string.
  - As soon as N_1 reads a 0, it permanently leaves the accept state q_1, since there are no transition arrows exiting q_2. Thus, we see that N_1 will accept any string that is either empty, or consists solely of 1s.

- We can prove that the following statement

  "if M is a DFA that recognizes a language A, then swapping the accept states of M with the non-accept states of M results in a DFA M' that recognizes $\bar{A}$."

will not hold true if it were instead talking about NFAs, by imagining the NFA N_2, which swaps the accept and non-accept states of N_1, and then proving that N_2 does <u>not</u> recognize $\bar{A}$.

N_2 state diagram:



$N_2 = \{Q, \Sigma, \delta, q_1, F\}$, where
1. $Q = \{q_1, q_2\}$
2. $\Sigma = \{0, 1\}$
3. δ is given as

	0	1
q_1	$\{q_2\}$	$\{q_1, q_2\}$
q_2	$\emptyset$	$\emptyset$

4. q_1 is the start state, and
5. $F = \{q_2\}$

- the language $\bar{A}$ would then represent all strings which are not in A, which we can describe as
  $\bar{A} = \{ w \mid w \text{ contains at least one } 0 \}$
- According to page 36 (Ch 1.1) of "Introduction to the Theory of Computation", a machine M always recognizes only 1 language A, and that this language A is the set of all strings that M accepts.
- Therefore, we can prove that N_2 does not accept $\bar{A}$ by finding at least one string accepted by N_2, which is not an element of $\bar{A}$.
  - let string $s = 11$. according to the definitions, $s \notin \bar{A}$ (and $s \in A$). Running the string s on N_2, we see that N_2 does accept s. Therefore, the machine N_2, which is a swapped-state iteration of N_1, does not accept the language $\bar{A}$.
- The example NFA N_1 proves that the previous statement regarding DFAs does not hold true for NFAs.

4. Prove that $A = \{0^{2i}1^i \mid i \geq 0\}$ is nonregular. $\Sigma = \{0, 1\}$

Let $A = \{0^{2i}1^i \mid i \geq 0\}$. We use the pumping lemma to prove that A is not regular. This proof is by contradiction.

Assume to the contrary that A is regular. A satisfies the pumping lemma. Let p be the pumping length given by the lemma. Choose s to be the string $0^{2p}1^p$.
If we let $p = 4$, $s = 000000001111$. Therefore, we know that s is a member of A, and that s has a length greater than p. The pumping lemma then guarantees that s can be split into 3 pieces, $s = xyz$, satisfying the 3 conditions of a lemma. We consider 3 cases to show that this result is impossible.

1. The string y consists of only 0s. For example, let $p = 3$:

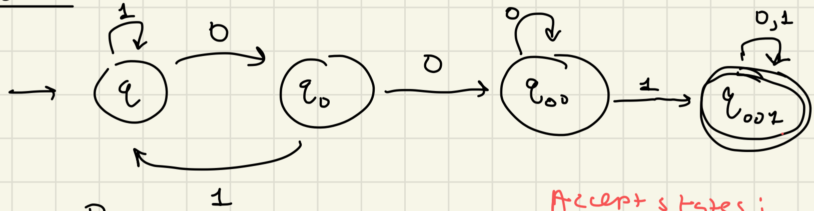
   $s = 0^6 1^3 = 000\underbrace{00}0111$

   No matter what p is, if y is some number of 0s, then the string xy^iz will have more than twice the amount of 0s than 1s and so is not a member of A, violating condition 1 of the pumping lemma. This case is a contradiction.
   - for ex, if $y = 00$, $xyyyz = 00000000111$, and $xyyyz \notin A$
2. The string y consists only of 1s. This case also gives a contradiction because xy^iz will have more 1s than 0s. Additionally, condition 3 (that $|xy| \leq p$) will also be violated because strings will always begin with $\underline{2p}$ 0s. For ex;
   - let $p = 3$ so $s = 000000111$. If we allow $y = $ "1", $z = $ "11", and $x = $ "000000", then $|xy| = 7$, which is greater than 3.
3. The string y consists of both 0s and 1s. This case is immediately impossible as it violates condition 3. The first $\underline{1}$ in $s = 0^{2p}1^p$ doesn't occur until 2p symbols have been read. To include both 0s & 1s in y, the length of xy must be at least $2p + 1$.

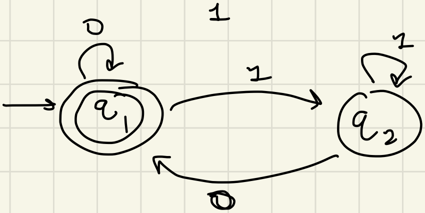
Thus a contradiction is unavoidable if we make the assumption that A is regular, so A is not regular.

Quiz 1

M1:



M2:



Accept states:

(q_{001}, q_1) (q_{001}, q_2)

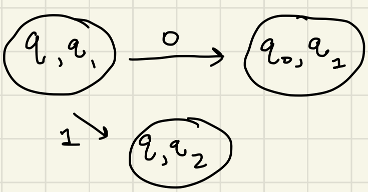
(q, q_1) (q_0, q_1) (q_{00}, q_1)

$\delta((q, q_1), 0) = (\delta_1(q, 0), \delta_2(q_1, 0))$

$\delta_1(q,0) \to q_0$, $\delta_2(q_1,0) \to q_1$

$= (q_0, q_1)$

M3:



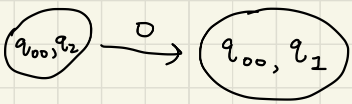
$\delta((q, q_1), 1) = (\delta_1(q, 1), \delta_2(q_1, 1))$

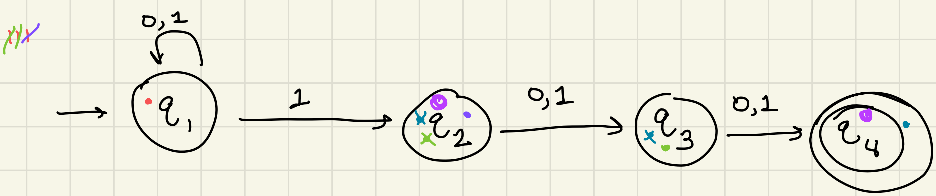
$\delta_1(q,1) \to q$, $\delta_2(q_1,1) \to q_2$

$= (q, q_2)$

$\delta_1(q_{00}, 0) = q_{00}$

$\delta_2(q_2, 0) = q_1$





→ M_3 must "accept" exactly when <u>either</u> M_1 or M_2 would accept

→ simulates both M_1 and M_2

→ upon input, simulate both individually — but can't "rewind the tape"

For each $(r_1, r_2) \in Q$ and each $a \in \Sigma$, let

$$\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a))$$

$$q_0 = (q_1, q_2)$$

Part 1: Automata and Languages

Ch 2 : Context-Free Languages

2.1 Context-Free Grammars

What are context-free grammars?

→ A more powerful method for describing languages (than finite automata or regular expressions)

→ can describe certain features that have a recursive structure.

→ Similar to regular expressions in that they basically denote a set of "rules" that a string must conform to in order to be part of that specific grammar's language.

What are some applications of context-free grammars (CFGs)?

→ In the study of human languages

→ In the specification & compilation of programming languages.

What are "context-free languages"?

→ The collection of languages associated with context-free grammars.

→ Includes all regular languages, as well as many additional languages.

→ A language is context-free if there exists some CFG that generates it.

What are pushdown automata?

→ A class of machines which recognize the CFLs.

What does a grammar consist of?

→ A collection of substitution rules, also known as "productions"

→ each rule appears as a line in the grammar that is comprised of A symbol - a **variable** - on the left side and a string of variables and other symbols - known as **terminals** - on the right side, separated by an arrow.

What are the key terms that describe the components of a grammar?

→ Lets use the example of the grammar G_1:

$$\begin{cases} A \longrightarrow 0A1 \\ A \longrightarrow B \\ B \longrightarrow \# \end{cases}$$

1. the substitution rules: the lines/statements of this lang
2. The variables: often represented by capital letters. Are on the left side of each rule.
   - variables of G_1 : $\{A, B\}$
3. The start variable: one of the variables is designated as the "start variable", and is the first one that you write down when generating a string of the language.
   - by convention, the start var usually occurs on the left side of the topmost rule.
   - start var of G_1 : A
4. The terminals: Analogous to the input alphabet (like in finite automata) — basically the set of symbols out of which the grammar's strings are generated.
   - often represented by lowercase letters, numbers, and/or special symbols.
   - terminals of G_1 : $\{0, 1, \#\}$

How can we use a grammar to describe a language?

→ By generating each string of that language, in the following manner:

1. Write down the start variable (which is the variable on the left side of the first rule, unless stated otherwise);

   A

2. Choose one of the variables that is written down (like A, which we just 'wrote down'), and find a rule that starts with that variable. Replace the written-down variable with the string that its arrow points to;

   rule: $A \to 0A1$

   ~~A~~ 0A1

3. Repeat step 2 until no variables remain (aka your 'written down' string consists only of terminals).

What is a "derivation"?

→ The sequence of substitutions used to generate a string in $L(G)$

Example?

→ For example, a **derivation** of string 000#111 in grammar G_1 is:

$\underbrace{A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111}_{\text{in all 3 of these, we used rule 1 to replace every "A" with "0A1"}} \Rightarrow \underbrace{000B111}_{\text{used rule 2 to replace "A" with "B"}} \Rightarrow \underbrace{000\#111}_{\text{used rule 3 to replace "B" with "\#"}}$

→ We can say that "the grammar G_1 generates the string 000#111"

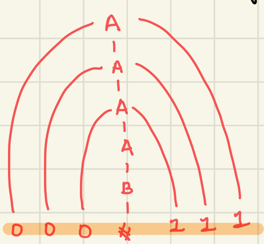
What is a parse tree?

→ A diagram that illustrates how a string gets generated

→ A pictorial way to represent the same info as a **derivation**.

Example?

→ The parse tree for 000#111 in grammar G_1:



What does $L(G)$ denote?

→ "The language of a grammar G"

→ All strings that can be generated using a derivation or parse tree of a grammar G, constitute the **language of the grammar**

- RECALL: a "language" is a set of strings

→ $L(G_1) = \{ \#, 0\#1, 00\#11, 000\#111, \ldots \}$

→ TODO: copy down engl language example bc its cool
copy down "compiler" example from lecture notes 2/05

What abbreviation is used in the substitution rules?

→ The | symbol, which represents an "or"

→ We use it to abbreviate several rules that have the same left hand variable, into one line.

Example of the | symbol?

→ G_1:

$A \to 0A1$

$A \to B$

$B \to \#$

can be equivalently written as

G_1:

$A \to 0A1 \mid B$

$B \to \#$

What is the formal definition of a context-free grammar?

A context-free grammar is a 4-tuple (V, Σ, R, S), where

1. V is a finite set called the **variables**,
2. Σ is a finite set, disjoint from V, called the **terminals**,
   - disjoint = no common elements between V and Σ
3. R is a finite set of **rules**, with each rule being: a variable and a string of variables & terminals, and
4. $S \in V$ is the start variable.

What does "yields" mean in CFGs?

→ Denotes a conversion (following a rule path) where one or more variables is converted into the string that its arrow points to.

→ If u, v, and w are strings (of both variables & terminals), and $A \to w$ is a rule of the grammar, we say that **uAv yields uwv**

- written as $uAv \Rightarrow uwv$
- with "yields", it isn't necessary for the 'yielded' string to consist only of terminals.

What does "derives" mean in CFGs?

→ We can say that u derives v (written as $u \overset{*}{\Rightarrow} v$) if either

a) $u = v$, or

b) There exists some sequence $u_1, u_2, u_3, \ldots, u_k$ for $k \geq 0$ and

$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k \Rightarrow v$

What is the notation to describe the language of a grammar?

→ $\{ w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w \}$

→ basically saying: the set of all strings w such that

- w is made up of any number/combo of the set of terminals, Σ
- There exists some derivation that begins at the **start variable** and returns w

How do context-free grammars relate to computer programming?

→ A **compiler** translates code written in a prog. language into another form that is more suitable for execution (like in Java, where .java files are compiled into .class files of **bytecode**, made up of 0s and 1s)

→ To do this, the compiler uses a process called **parsing** to extract the meaning of the code.

- One tangible representation of this meaning is to view the prog. language as having a context-free grammar, & considering the **parse tree** for the code!

What is an ambiguous grammar?

→ A grammar is ambiguous if it is able to generate the same string in more than one way (more than one pathway of rules to follow)

→ We say that a string is "derived **ambiguously**" from a grammar if it can be derived from the grammar in more than one way

→ to be derived ambiguously, the string must have **2 or more parse trees, specifically** - not necessarily 2 or more derivations.

Why can't derivations indicate ambiguity?

→ Because 2 derivations can differ merely in the order in which they replace variables while being identical still in **structure**.

What are the implications of ambiguous grammars?

→ A grammar that generates strings ambiguously can sometimes be undesirable for programming languages (& other similar applications).

Why? Because in those situations, a program should be able to obtain a <u>unique interpretation</u> of every string in the lang.

Example of an ambiguous grammar?

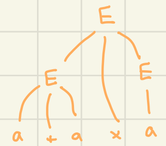
→ Grammar G_5: (let "E" be short for "EXPRESSION")

$E \rightarrow (E + E) \mid (E \times E) \mid E \mid a$

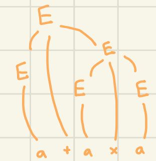
(notice the use of the "or" operator. G_5 is actually composed of 4 rules.)

→ G_5 generates the string "$a + a \times a$" ambigously, by either applying $E \rightarrow E + E$ or $E \rightarrow E \times E$ first:

① $E \Rightarrow E \times E \Rightarrow E + E \times E \Rightarrow a + a \times a$



② $E \Rightarrow E + E \Rightarrow a + E \times E \Rightarrow a + a \times a$



Note: the derivations above don't necessarily indicate the ambiguity, but the parse trees do.

What is a leftmost derivation?

→ A specific type of derivation that replaces variables in a **fixed order**

- More concentrated on structure, and therefore can be used to discern ambiguity of a grammar!

→ A derivation of a string w in a grammar G is a leftmost derivation if at every step, the leftmost remaining variable is the one that gets replaced.

Example?

→ Let grammar G_2 be a CFG where $V = \{S\}$ and $\Sigma = \{0, 1\}$

(1) $S \to 0S1S$

(2) $S \to 1S0S$

(3) $S \to \varepsilon$ — the empty string ε can also be on the right side of a rule. Turning a variable into ε means it just dissapears, basically

→ This derivation for 011100 is leftmost:

- in this step, only the leftmost S is replaced, in this case to ε (rule 3)
- the other S remains

$$S \Rightarrow 0S1S \Rightarrow 0\_1S \Rightarrow 011S0S \Rightarrow 0111S0S0S \Rightarrow 0111\_0S0S$$
$$\Rightarrow 01110\_0S \Rightarrow 011100$$

What is the formal definition of an ambiguous grammar?

→ A grammar is ambiguous if it can generate the same string using 2 or more **leftmost derivations**.

What does "inherently ambiguous" mean?

→ Sometimes there are ambiguous grammars for which we can find an unambiguous grammar that generates the same language.

→ But some context-free langs can only be generated by ambiguous grammars — these are called **inherently ambiguous languages**.

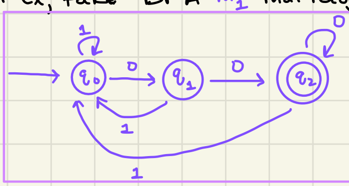
What is the relationship between CFGs and DFAs?

→ For <u>every</u> DFA, there exists an equivalent CFG!

→ We can construct a CFG for a regular language by referring to its DFA.

What are the steps to create a CFG out of a DFA?

→ For ex, take DFA M_1 that recognizes language A ($L(M_1) = A$) and $A = \{w \mid w \text{ ends in } 00\}$



1. make a variable R_i for each state q_i of the DFA:

$\Sigma = R_0, R_1, R_2$

What are the steps to create a CFG out of a DFA? (continued)

2. Add the rule $R_i \rightarrow aR_j$ to the CFG if $\delta(q_i, a) = q_j$ is a transition in the DFA:

δ:

	0	1
q_0	q_1	q_0
q_1	q_2	q_0
q_2	q_2	q_0

$\delta(q_0, 0) = q_1$ $\quad R_0 \rightarrow 0R_1$
$\delta(q_0, 1) = q_0$ $\quad R_0 \rightarrow 1R_0$
$\delta(q_1, 0) = q_2$ $\quad R_1 \rightarrow 0R_2$
$\delta(q_1, 1) = q_0$ $\quad R_1 \rightarrow 1R_0$
$\delta(q_2, 0) = q_2$ $\quad R_2 \rightarrow 0R_2$
$\delta(q_2, 1) = q_0$ $\quad R_2 \rightarrow 1R_0$

3. Add the rule $R_i \rightarrow \varepsilon$ for the DFA's accept state q_i:

$R_2 \rightarrow \varepsilon$

4. Make R_i the start variable of the grammar for the DFA's start state q_i:

$S = R_0$

→ The resulting equivalent CFG to M_1 is grammar G_1, which generates the language A ($L(G) = A$):

$$R_0 \rightarrow 0R_1 \mid 1R_0$$
$$R_1 \rightarrow 0R_2 \mid 1R_0$$
$$R_2 \rightarrow 0R_2 \mid 1R_0 \mid \varepsilon$$

→ We can see for ourself this is true by deriving the string 10100:

$$R_0 \Rightarrow 1R_0 \Rightarrow 10R_1 \Rightarrow 101R_0 \Rightarrow 1010R_1 \Rightarrow 10100R_2 \Rightarrow 10100\ (\varepsilon)$$

What is another way to design a CFG for a context-free language?

→ If the CFL isn't also a regular language (so we can't use a DFA), we can construct a CFG by breaking the CFL into simpler pieces and constructing individual grammars for each piece.

- Many CFLs are the union of simpler CFLs

→ We can then merge the individ. grammars back together

Example of this technique?

→ Lets generate a CFG for the language $\{0^n1^n \mid n \geq 0\} \cup \{1^n0^n \mid n \geq 0\}$

1. Construct a grammar for the 1st part of the language ($\{0^n1^n \mid n \geq 0\}$):

$$S_1 \rightarrow 0S_11 \mid \varepsilon$$

2. Construct a grammar for the 2nd part of the language ($\{1^n0^n \mid n \geq 0\}$), using a different variable:

$$S_2 \rightarrow 1S_20 \mid \varepsilon$$

(ctd next page)

3. Construct a new grammar that contains all the rules from the other grammars, as well as a new rule of the format $J \rightarrow J_1 \mid J_2 \mid \ldots \mid J_k$, where the variables $J_1, J_2, \ldots J_k$ are the **start variables** of the other individ. grammars

→ basically adding a rule with a new variable that "directs" / "refers" the combined grammar to all of the littler ones.

Grammar G_1 which generates language $L = \{0^n 1^n \mid n \geq 0\} \cup \{1^n 0^n \mid n \geq 0\}$:

$$S \rightarrow S_1 \mid S_2$$
$$S_1 \rightarrow 0 S_1 1 \mid \varepsilon$$
$$S_2 \rightarrow 1 S_2 0 \mid \varepsilon$$

What is the Chomsky normal form?

→ A way to put a CFG into a simplified form, which can be useful in giving algorithms for working with CFGs.

→ **DEFN:** A CFG is in Chomsky normal form (CNF) if every rule is of the form

$$A \longrightarrow BC$$
$$A \longrightarrow a$$

- where $A, B, C \in V$, $a \in \Sigma$, and neither B nor C are the start variable. (so basically the start variable isn't allowed to be on the right-hand side of a rule)
- in addition, the rule $S \longrightarrow \varepsilon$ is allowed, where S is the start variable.

Can any CFG be converted into CNF?

→ Yes!

→ **Theorem:** Every context-free grammar can be converted to an equivalent CFG which is in Chomsky normal form.

Part 1: Automata and Languages

Ch 2: Context-Free Languages

2.2 Pushdown Automata

What are pushdown automata?
→ a type of computational model (like DFAs and NFAs)
→ Equivalent in power to CFGs (aka, they also recognize all CFLs)
→ Basically an NFA that also has an extra component called a **stack**.

RECALL: Why can't DFAs or NFAs recognize nonregular languages?
→ because they have a limited ("finite") memory, and most nonregular languages (like $L = \{0^n 1^n \mid n \geq 0\}$) require a recognizing machine to be able to "count" the number of each type of input symbol received in order to calculate whether a given string is a part of that language
- but to count & keep track of so many symbols requires unbounded memory..
- this idea is basically the basis of the pumping lemma proof.

→ As opposed to regular languages, where a series of 'rules' (via transition functions & states) is enough to be able to recognize strings of any length
- with regular langs, no need to "keep track" of the symbols being read.

What is the difference between deterministic & nondeterministic pushdown automata (PDAs)?
→ RECALL: "deterministic" = only one possible outcome/choice while nondeterministic = set of possible outcomes/choices
→ Nondeterministic Pushdown Automata are stronger than deterministic PDAs — they recognize a larger class of languages.
- Unlike with finite automata, where DFAs and NFAs are equivalent in power and recognize the exact same class of languages.

Which type of PDA will we focus on?
→ Only **nondeterministic** PDAs are equivalent to CFGs in their power, so we will focus on those. Unless stated otherwise, assume PDA to mean "nondeterministic pushdown automata"

Recap of Models learned thus far

computational model	language recognized
Deterministic Finite Automata	Regular Languages
Nondeterministic Finite Automata	Regular languages
Context-Free Grammar	Regular languages Context-Free languages
Pushdown Automata	Regular languages Context-free languages

Why are PDAs useful?
→ We now have 2 options for proving a language is context-free. We can either give
- a CFG that generates it or • a PDA that recognizes it

→ Certain languages are easier to describe with CFGs, and vice versa with PDAs.

What is a stack?

→ A component that provides **additional memory** beyond the finite amount available in the control.

→ Valuable because it can hold an unlimited amount of information.

→ A PDA is able to recognize languages that NFAs and DFAs can't (like L from prev. page) because it has a stack that allows it to hold/store numbers of unbounded size!

Diagram of a finite automation:

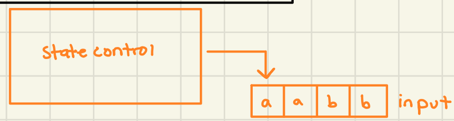
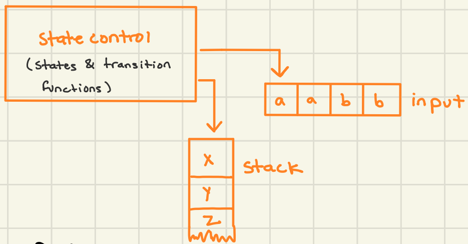


Diagram of a Pushdown automation:



How does the stack work?

→ a PDA can write symbols on the stack & read them back later

- writing a symbol "pushes down" all the other symbols on the stack

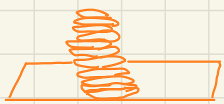
→ At any time, the symbol on top of the stack — aka, the most recently added symbol — can be read and removed.

How can we visualize the "stack"?

→ RECALL COMP 210! Stack is a data structure defined by "last in, first out"

→ Imagine a stack of plates resting on a spring — When a new plate is placed on top of the stack, the rest of the plates below it are pushed down.

- If you want to take out a plate, it must be the one on top of the stack (or else you'll make a mess trying to pull one out from the middle)



What is "pushing" and "popping"?

→ **pushing** : Writing a new symbol on (to) the stack

→ **popping** : Removing a symbol (aka the most recently added one) from the stack

So how does a PDA actually work?

→ Lets take the example nonregular language $L = \{0^n1^n \mid n \geq 0\}$

— note that the CFG for L is $A \to 0A1 \mid \varepsilon$ $(V = A, \Sigma = \{0,1\})$ —

→ The PDA P_1 reads symbols from the input and functions like so:

- everytime P_1 reads a 0, it pushes it onto the stack
- As soon as (and everytime) P_1 sees a 1, it pops a 0 off the stack
- if P_1 finishes reading the input exactly when the stack becomes empty (all 0s popped) ⟶ input accepted
- if the stack becomes empty while 1s remain to be read ⟶ input rejected
- if the input is finished but the stack isn't empty ⟶ input rejected
- if a 0 appears in the input after a 1 ⟶ input rejected

How do we formally define a PDA?

→ similar to formal def of an NFA, except for the stack;

Q, Σ, and q_0 are basically the same

What is the "formal" defn of the stack?

→ Formally, the stack is a device containing symbols drawn from some alphabet.

→ The machine can use **different alphabets** of symbols for its input and its stack, so now we must also specify **a stack alphabet**, Γ

What is the **domain** of a PDA's transition function?

→ aka, what are the components that may determine the next move of a PDA?

→ ANS: The current state, the next input symbol read, and the top symbol of the stack

- since the current state could be any $q \in Q$, the next input could be any $a \in \Sigma$, and the top stack symbol could be any $t \in \Gamma$, we'd define the domain as the **cartesian set** of all possible combos of the 3

→ Additionally, either the input or the stack symbol are allowed to be ε (where the machine moves w/o reading a symbol from the input, or w/o reading a symbol from the stack)

- RECALL that $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$

→ Therefore, we define the domain of the transition function δ as

$$Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon$$

What is the **range** of a PDA's transition function?

→ aka, what are the possible next moves of the PDA when it is in a particular situation and reads an input symbol (from Σ)?

→ ANS: It may enter some new state or stay at its current state. And it may or may not write some new symbol on top of the stack.

- we can write this as the cartesian set of states (Q) and stack symbols (Γ_ε) (since an input will result in the PDA being at one of the Q states and adding one of the Γ_ε symbols to stack): $Q \times \Gamma_\varepsilon$

So the range is $Q \times \Gamma_\varepsilon$?

→ Umm no! Not just yet. Since this is a **nondeterministic** machine, the PDA can have several possible next moves.

→ So we should return a set of members of $Q \times \Gamma_\varepsilon$ ... the **power set**

$P(Q \times \Gamma_\varepsilon)$

- represents the set of all possible next states

RECALL: What is a power set?

→ $P(B)$ = the power set of B = the collection of all possible subsets of B

- for ex, if $B = \{1,2,3\}$, $P(B) = \{\{\}, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,2,3\}, \{1,3\}, \{2,3\}\}$

So what is our final PDA transition function?

→ Putting it all together,

$$\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow P(Q \times \Gamma_\varepsilon)$$

So what is the formal definition of a pushdown automata?

A pushdown automaton is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where Q, Σ, Γ, and F are all finite sets, and

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow P(Q \times \Gamma_\varepsilon)$ is the transition function,
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

When does a PDA accept an input?

→ A PDA $M = (Q, \Sigma, \Gamma, \delta, q, F)$ accepts an input w (if w can be written as $w = w_1 w_2 \dots w_m$) where:

- each $w_i \in \Sigma_\varepsilon$
- a sequence of states $r_0, r_1, \dots, r_m \in Q$ and a sequence of strings $s_0, s_1, \dots, s_m \in \Gamma$

(where s_i represents the sequence of stack contents that M has on the accepting branch of the computation)

exist such that the following 3 conditions are satisfied:

1. $r_0 = q_0$ and $s_0 = \varepsilon$
   - that M begins in a start state & with an empty stack
2. for $i = 0, \dots, m-1$, we have $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\varepsilon$ and $t \in \Gamma^*$
   - this condition basically states that M moves properly according to the state, stack, and next input symbol.
3. $r_m \in F$
   - that an accept state occurs at the end of the input.

How do we use the formal definition to describe individual PDAs?

→ Lets take the example nonregular language $L = \{0^n 1^n \mid n \geq 0\}$

— note that the CFG for L is $A \rightarrow 0A1 \mid \varepsilon$ $(V = A, \Sigma = \{0, 1\})$ —

→ The formal description of the PDA M_1 that recognizes L:

let M_1 be $(Q, \Sigma, \Gamma, \delta, q_1, F)$ where

1. $Q = \{q_1, q_2, q_3, q_4\}$,
2. $\Sigma = \{0, 1\}$,
3. $\Gamma = \{0, \$\}$
4. $F = \{q_1, q_4\}$
5. $q_1 \in Q$ is the start state, and

6. δ is given by the following table, where blank entries signify $\emptyset$ (the empty language):

	Input:	0			1			ε		
	Stack:	0	\$	ε	0	\$	ε	0	\$	ε
state	q_1									$\{(q_2, \$)\}$
	q_2			$\{(q_2, 0)\}$	$\{(q_3, \varepsilon)\}$					
	q_3				$\{(q_3, \varepsilon)\}$				$\{(q_4, \varepsilon)\}$	
	q_4									

RECALL: if a certain input yields $\emptyset$ (in the transition chart), this means that a state has <u>no actions / transition arrows</u> for that input – aka, if a opy of the machine receives that input, it has no further actions and dies

→ These 3 fields of the chart are what comprise the domain $Q \times \Sigma_\varepsilon \times T_\varepsilon$ of a PDA; based on the current state and the symbol currently at the top of the stack, this chart tells us what will happen if each input symbol $0, 1, \varepsilon \in \Sigma_\varepsilon$ were the next to be read.

→ These values represent results of various inputs into $\delta(Q \times \Sigma_\varepsilon \times T_\varepsilon)$ to yield a set of (Q, T_ε) pairs.

- For example, when M_1 is in state q_2, the stack is currently empty and it reads an input of <u>0</u>, it remains in q_2 and pushes a 0 onto the stack -- given by $\{(q_2, 0)\}$
- When M_1 is in q_2, has a 0 at the top of its stack, and reads a <u>1</u>, it moves to q_3 but does not add anything to stack -- given by $\{(q_3, \varepsilon)\}$!

and pushes the <u>0</u> off the stack... right?

How does a PDA check to see if the stack is empty?

→ The formal definition of a PDA doesn't have any explicit mechanism for the PDA to test for an empty stack (which, with example lang L, is something it needs to be able to do).

→ To get the same effect, we use the stack symbol \$ (\$ $\in T$)

How is the \$ symbol used?

→ The PDA initially places the \$ symbol on the stack – but never again after that.

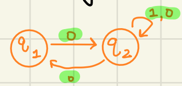
→ Then, if it ever looks to the stack and sees the \$ at the top, it knows that there is nothing underneath, and thus the stack is effectively "empty"

Example?

→ In the δ chart for M_1 notice that when M_1 is in q_1 (aka the start state!) with an empty stack, it yields the move $\{(q_2, \$)\}$ when it reads ε from the input.

- This is essentially describing the first transition that happens before all other transitions. Reading a "ε" from input ≈ action taken simply by default.
- The first move of M_1, as we can see, is to push a \$ onto stack!

How do we create a state diagram for a PDA?	→ Very similar to making state diagrams for NFAs ; all we have to add is a feature to show how the PDA uses its **stack** when going from state to state. → **Theorem**: every NFA can be converted to an equivalent PDA.
How is the stack represented in PDA state diagrams?	→ Rather than just putting an input symbol on each transition arrow, like with NFAs ;



We write a statement of the form $a, b \longrightarrow c$, where

a = the input symbol read

b = a stack symbol that may get popped off the stack & replaced by c

c = a stack symbol that may be added on top of the stack (as part of this transition) — i.f.f. the current stack symbol is b.

→ "When the machine is reading an a from input, it may replace the symbol b on top of the stack with a c."

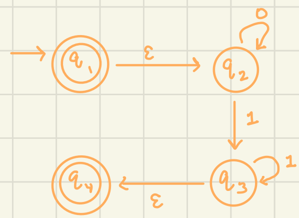
Example of creating a PDA?

→ To make a state diagram of M_1 (same example as formal defn):

1. Draw an NFA of the machine based on its transition function table — ignore the parts about the stack, & draw it exactly as you would any NFA:

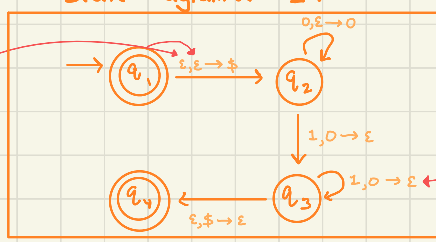
	Input:	0			1			ε		
	Stack:	0	\$	ε	0	\$	ε	0	\$	ε
state	q_1									$\{(q_2, \$)\}$
	q_2			$\{(q_2, 0)\}$	$\{(q_3, \varepsilon)\}$					
	q_3				$\{(q_3, \varepsilon)\}$				$\{(q_4, \varepsilon)\}$	
	q_4									

δ ↗



2. Add in the $a, b \longrightarrow c$ statements, where b is the current symbol and c is the potential replacement.

State Diagram of M_1:



What does it mean when a, b or c are ε?

→ $a = \varepsilon$: signifies a transition that M_1 makes <u>without</u> reading an input symbol first to trigger it. (RECALL — in NFAs, an "input" of ε means that the machine automatically splits into 2 copies)

→ $b = \varepsilon$: signifies a transition that M_1 makes w/o reading & popping any symbol off the stack.

→ $c = \varepsilon$: signifies a transition that M_1 makes by <u>popping</u> a symbol b off the stack, but not replacing it with any symbol (nothing <u>pushed</u> on)

What is another ex of a PDA?

→ a PDA M_2 which recognizes the language
$\{a^i b^j c^k \mid i,j,k \geq 0 \text{ and } i=j \text{ or } i=k\}$

How does M_2 work, informally?

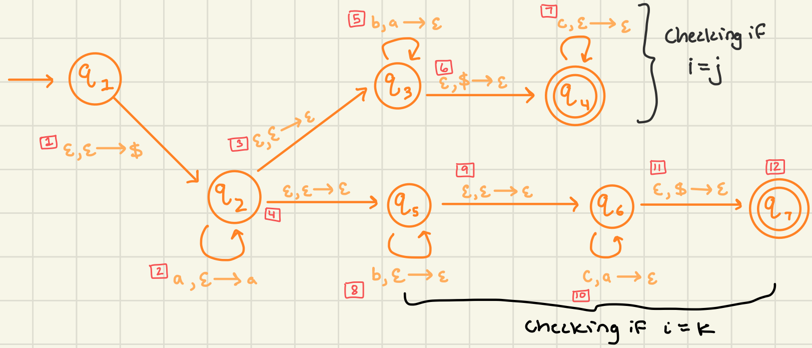
→ M_2 works by <u>first</u> reading and pushing all of the as — so when the as are done, M_2 has them all on the stack so that it can match them with either the bs or the cs (since $i=j$ or k)

→ Next, M_2 uses **nondeterminism (which is essential here)** to have 2 branches — one for each possible input of a b or a c. If either of them matches, it **accepts**.

- w/o nondeterminism, it wouldn't know in advance whether to match (/push?) the as with the bs or the cs.

aaR_2

State diagram of M_2?



What does each transition ($\delta(Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon)$) in M_2 mean?

1. using \$ to test for empty stack! Automatic first transition.

2. as long as M_2 continues to read as, it pushes the a symbol onto stack

3,4. M_2 automatically creates 2 branches (which don't add anything to stack), one to account for each of the input symbols b and c

–this "spawning" of 2 copies is known as a **mode shift**.

5. As long as bs are read from the input and the stack contains as, then an a is "popped" off of the stack (denoted by $\_, a \to \varepsilon$) for each b read

6. Once all of the as have been popped off the stack, the remaining top symbol will be a \$ — this indicates that an equivalent amount of bs have been read & that currently, <u>$i=j$</u>. To act on this, and ONLY once the top of stack is \$, a transition to accept state q_4 is <u>automatically</u> made (since input is "ε")

7. Now that we have attained $\boxed{i=j}$, M_2 no longer cares ab counting the # of cs read so it will stay in the accept state as long as it is reading cs

- However, there is no transition arrows from q_4 for if an a or a b is read — since that would be "illegal" to the language atp.
- Thus, if any as or bs are read, the current copy of M_2 would die

8. In this section, we want to see if the # of cs = # of as, so the bs in the middle don't rlly mean anything to or effect M_2. As long as it reads bs, M_2 doesn't touch the stack and remains in state q_6. Similar to $\boxed{7}$ -- "burning through the bs".

9. Additionally, an automatic ("mode shift") transition is created to act on the input symbols after all bs have been "burned"

10. Similar to $\boxed{5}$ -- popping an a off the stack for each c read, but staying in state q_6 the whole time.

11. Same as $\boxed{6}$ but for cs.

12. Unlike the other accept state q_7 has no self-pointing arrows. Why?

- B/c at this point, the input has given us x # of as, some arbitrary # of bs, and exactly x # of cs
- M_2 does not want to see any more input or else the rules won't be fulfilled & the string won't accept.

  Thus, M_2 remains in accept state q_7 iff it doesn't read any more symbols. If it does, there is no arrow corresponding to them and that copy of the machine would then die!

What is an example of a PDA that recognizes the language $L = \{ww^R \mid w \in \{0,1\}^*\}$

→ w^R means w written backwards.

$\{01011\ 11010,\ 0110,\ \varepsilon,\ 11,\ 00\} \in L$

→ the PDA M_3 will work by first pushing all the symbols that are read onto the stack $\boxed{1}$

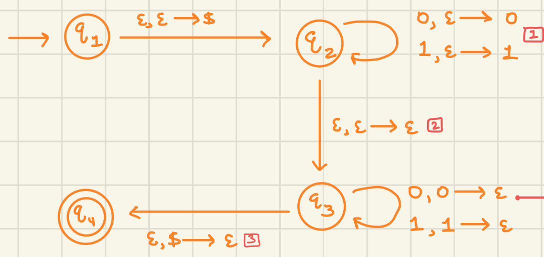
How will M_3 know when to start checking for the w^R portion of a string?

→ At each point (of reading an input), M_3 will use nondeterminism to automatically create another copy $\boxed{2}$ that assumes that the middle of the string has just been reached & its time to start "checking for" w^R

- There, it will switch to popping a symbol off stack for each input read, & checking to see if the two are the same

→ If the input read & stack symbol popped were always the same, & the stack empties at the same time that the input is finished $\boxed{3}$, M_3 accepts. Otherwise, reject.

State Diagram of M_3?



If at any point a 0 is read but the top stack symbol is NOT 0, the copy of M_3 then just dies. This "rule/transition" only applies to the specified symbols of a, b (→ c)

What is the relationship between PDAs and CFGs?

→ PDAs and CFGs are **equivalent** in power — both are capable of describing the class of all context-free languages (CFLs).

→ Any CFG can be converted into an equivalent PDA and vice versa.

→ A language is context-free if & only if there is some PDA OR some CFG that describes it.

How do you convert a CFG into a PDA generating the same language?

→ its sort of complicated but basically, the PDA (P) will work by accepting its input (w) by determining whether there is a **derivation** for w in the grammar (G).

- basically, P will try to derive the string w, and determine whether there is some series of substitutions (using the rules of G) that can lead from the start variable to w.

What is an "intermediate string"?

→ In the derivation for a grammar, each step of it yields an **intermediate string** that is some combo of variables & terminals:

$G_1: A \rightarrow 0A1 \mid \varepsilon$

$A \rightarrow 0A1 \rightarrow 0\underline{0A11} \rightarrow 00\underline{0A1}11$

intermediate strings

How does nondeterminism come into play in this process?

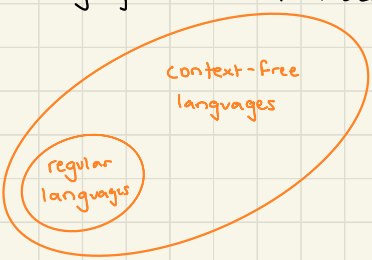
→ Since for every variable on the left-hand side of a rule in G there can be multiple possible substitutions (RECALL: the whole reason for the "|" shorthand), the PDA uses its **nondeterminism** to guess the sequence of correct substitutions for a given input.

- At each step of the derivation, a branch is made for each of the rules for a particular variable, and used to substitute something for it.

– Summary: context-free languages –

What is the relation between regular languages & context-free languages?

→ All regular languages are included in the class of CFLs!



→ CFGs & PDAs describe the same class of languages – CFLs.

- For every CFG that describes a language A, there exists an equivalent PDA that recognizes the same A.

Part 1: Automata and Languages

Ch 2: Context-Free Languages

2.3: Non-Context-Free Languages

How do we prove that a language is not context-free?

→ Similar to proving languages aren't regular, we use an altered version of the **pumping lemma**!

→ For ex, take the non-context-free language

$B = \{a^n b^n c^n \mid n \geq 0\}$

What is the idea behind the pumping lemma for CFLs?

→ RECALL: For regular languages: the pumping lemma says that for any regular language, each string in that lang contains a section which can be repeated any number of times, with the resulting string remaining a correct component/element of that language.

→ For CFLs, the idea is similar – that every context-free language has a special value called the pumping length such that all longer strings in the language can be "pumped" – but the meaning of "pumped" slightly differs.

What does it mean that a CFL can be "pumped"?

→ That the string can be divided into 5 parts so that the 2^{nd} and 4^{th} parts may be repeated any number of times, with the resulting string still a member of the language.

- as opposed to regular languages, where a string is divided into 3 parts and the 2^{nd} part is repeated ('pumped')

What is the formal definition of the pumping lemma for CFLs?

if A is a context-free language, then there is a number p (the pumping length) where, if s is any string in A of length $|s| \geq p$,

then s may be divided into 5 pieces $s = uvxyz$ while satisfying the following conditions:

1. for each $i \geq 0$, $uv^ixy^iz \in A$

   basically saying that the 2^{nd} & 4^{th} parts can be duplicated in s any # of times, and the resulting modified version of s will still be a part of lang A.

2. $|vy| > 0$

   Says that at least one of v or y (but not necessarily both) is not the empty string – otherwise the theorem would be trivially true. (the length of v plus the length of y must be > 0)

3. $|vxy| \leq p$

   Says that the pieces x, y, and v together have a length of at most p.

Example: how do we use the pumping lemma?

→ Lets prove that the language $B = \{a^n b^n c^n \mid n \geq 0\}$ is <u>not</u> context-free.

→ if we let "p" be the pumping length, we know that our chosen string "s" has to be at least p symbols long – $s = a^p b^p c^p$ would satisfy this.

- e.g. abc ($p=1$), aaabbbccc ($p=3$), aabbcc, etc.

What do we need to prove about s?

→ That no matter how we divide s into 5 pieces $uvxyz$, one of the 3 conditions of the pumping lemma will be violated.

→ Keeping condition 2 in mind – that either v or y must be <u>non</u>empty – there are 2 "cases" that encapsulate all possible ways that s can be divided into $uvxyz$:

1. Both v and y contain only one type of alphabet symbol

   for ex, $aaab\underbrace{b}_{v}b\underbrace{c}_{y}cc$ or $a\underbrace{a}_{v}b\underbrace{b}_{y}cc$ or $\underbrace{a}_{v}\underbrace{b}_{y}c$

2. Either v or y contains more than one type of alphabet symbol

   For ex, $a\underbrace{ab}_{v}b\underbrace{c}_{y}c$ or $\underbrace{a}_{v}\underbrace{bc}_{y}$

Why do these cases contradict the p.l.?

1. If v and y both contain only 1 type of symbol, meaning that when we pump v and y, the 3rd symbol will not get duplicated – for example,

   $s = aab\underbrace{b}_{v}\underbrace{c}_{y}c$ so $uv^3xy^3z = aab\,\underline{bbb}\,\underline{ccc}\,c$

   And thus the # of the symbol not included in v or y will become much smaller than the other 2. The string s won't be able to satisfy $a^n b^n c^n$ & therefore cannot be a member of B.

   This violates condition 1 and is thus a contradiction.

2. In this situation, it is possible to have an equal number of as, bs, and cs. However, not in the correct order! For ex:

   $s = a\underbrace{ab}_{v}b\underbrace{c}_{y}c$ so $uv^2xy^2z = a\,\underline{abab}\,b\,\underline{cc}c$

   Hence it cannot be a member of B and a contradiction occurs.

Summary of Ch.1 and 2

1. DFA — the most simple model of what an algorithm can do.

||

2. NFA — introduced NFAs, which are equivalent in power to DFAs

||

regular expressions

3. NPDA — nondeterministic PDAs

$\Downarrow$

CFGs

a way to describe languages

Ari Kumar
Due February 23, 2024

1. CFGs for the following languages, where $\Sigma = \{a, b, c\}$

a) $A = \{a^i b^j \mid i > j \geq 0\}$

let grammar $G_1 = (V, \Sigma, R, S)$ where
$V = \{R_1, R_2\}$,
$\Sigma = \{a, b\}$,
$S = R_1$,
and the set of rules $\underline{R}$ is :

$R_1 \rightarrow \varepsilon \mid aR_1 \mid aR_2$

$R_2 \rightarrow aR_2b \mid \varepsilon$

This grammar accurately generates language A. I created it on the basis of 2 conditions:

1. no $\underline{b}$s should appear in the string until at least one $\underline{a}$ has appeared.
   - The start variable has 2 possible substitutions excluding the empty string ε, and both of them begin with the terminal $\underline{a}$, ensuring no b can appear beforehand.
2. There can be any number of as that appear before a single b, e.g. i can be $j+1$, but it doesn't have to be.
   - One of the start variable's substitution rules, aR_1, allows an unlimited amount of as to appear before a single b.

Additionally, both i, j can be = to 0, so the empty string ε is given as a substitution rule of the start variable R_1.

b) $B = \{a^i b^j c^k \mid i = j \text{ or } i = k \text{ where } i, j, k \geq 0\}$

let grammar $G_2 = (V, \Sigma, R, S)$ where
$V = \{R_1, R_2, R_3, R_4, R_5, R_6\}$,
$\Sigma = \{a, b, c\}$,
$S = R_1$,
and the set of rules $\underline{R}$ is :

$R_1 \rightarrow R_2 \mid R_5$

$R_2 \rightarrow R_3 R_4$

$R_3 \rightarrow aR_3b \mid \varepsilon$

$R_4 \rightarrow cR_4 \mid \varepsilon$

$R_5 \rightarrow aR_5c \mid R_6$

$R_6 \rightarrow bR_6 \mid \varepsilon$

Ari Kumar
Due February 23, 2024

1.

This grammar accurately generates language B. I created it by breaking B into the union of 2 smaller CFLs and then constructing a grammar for each piece.

Language B_1 : $\{a^i b^j c^k \mid i = j \text{ where } i, j, k \geq 0\}$

A grammar to describe B_1 is as follows (informally):

$$R_1 \rightarrow R_2 R_3$$
$$R_2 \rightarrow a R_2 b \mid \varepsilon$$
$$R_3 \rightarrow c R_3 \mid \varepsilon$$

Language B_2 : $\{a^i b^j c^k \mid i = k \text{ where } i, j, k \geq 0\}$

A grammar to describe B_2 is as follows (informally):

$$R_1 \rightarrow a R_1 c \mid R_2$$
$$R_2 \rightarrow b R_2 \mid \varepsilon$$

I then combined these 2 grammars by adding a start variable to G_2 which points to the start variables of the individual grammars.

c) $C = \{a^i b^j c^k \mid i + j = k \text{ where } i, j, k \geq 0\}$

let grammar $G_3 = (V, \Sigma, R, S)$ where

$V = \{R_1, R_2\}$,

$\Sigma = \{a, b, c\}$,

$S = R_1$,

and the set of rules <u>R</u> is:

$$R_1 \rightarrow a R_1 c \mid R_2 \mid \varepsilon$$
$$R_2 \rightarrow b R_2 c \mid \varepsilon$$

This grammar accurately generates language C. Since $i+j=k$, we know that the appearance of <u>each and any</u> as or bs in the input string must also have a corresponding c at the end of the input string. To ensure this, both R_1 and R_2 do not allow any a or b terminals to generate without a c terminal generating as well. R_1 allows a string with any number x of as, as well as x number of cs. Then, there is an option to leave the string as such (by using $R_1 \rightarrow \varepsilon$), yielding a string where $j = 0$ and $i + j = k$ - this string is an element of C. Alternatively, we can add as many bs to the string as we want, and each step of this will also add another c to the end such that $i + j$ always $= k$.

Ari Kumar

Due February 23, 2024

COMP 455-001

Homework 2

Page 3

2. Give a formal description of a PDA which recognizes the language $A = \{ w \mid w = w^R \}$. Give an informal description of why your answer is correct.

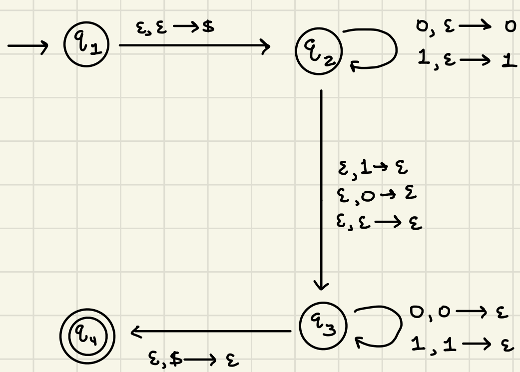
let $M_1 = (Q, \Sigma, \Gamma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3, q_4\}$,
2. $\Sigma = \{0, 1\}$
3. $\Gamma = \{0, 1, \$\}$
4. $F = \{q_4\}$, and

δ is given by the following table, wherein blank entries signify $\emptyset$.

Input:	0				1				ε			
Stack:	0	1	\$	ε	0	1	\$	ε	0	1	\$	ε
state: q_1												$\{(q_2, \$)\}$
q_2				$\{(q_2, 0)\}$				$\{(q_2, 1)\}$	$\{(q_3, \varepsilon)\}$	$\{(q_3, \varepsilon)\}$		$\{(q_3, \varepsilon)\}$
q_3	$\{(q_3, \varepsilon)\}$					$\{(q_3, \varepsilon)\}$					$\{(q_4, \varepsilon)\}$	
q_4												

A state diagram of M_1:



Explanation:

M_1 accurately recognizes language A. M_1 is modeled after the PDA M_3 from example 2.18 (page 116, ch. 2.2) of the textbook, "Intro to the Theory of Computation". M_3 recognizes the language $B = \{ww^R \mid w \in \{0,1\}^*\}$, which is quite similar to language A.

I determined that $B \subseteq A$; that is, all strings in language B are also palindromes and are also elements of language A. Therefore, all strings accepted by M_3 should also be accepted by M_1.

The difference between A and B is that A includes binary palindromes of an odd-numbered length (where the middle of the palindrome is a 0 or a 1, like 1001001), while B includes only even-length palindromes.

Ari Kumar

Due February 23, 2024

2.

M_3 works by using the transition $\delta(q_2, \varepsilon, \varepsilon) \to (q_3, \varepsilon)$ to nondeterministically guess, at each step, that the middle of the input string has been reached. M_3 then switches to popping symbols off the stack & checking if they match the symbols read (because if its a palindrome, the two will be the same).

(see the explanation of M_3 on page 116)

M_1 functions in a similar manner, and contains the same $\delta(q_2, \varepsilon, \varepsilon) \to (q_3, \varepsilon)$ function for the case of an even-length binary palindrome. However, to account for the possibility that the palindrome contains a 1 or a 0 as its midpoint and is odd-numbered in length, I added the following 2 transitions:

$$\delta(q_2, \varepsilon, 1) \to (q_3, \varepsilon)$$

$$\delta(q_2, \varepsilon, 0) \to (q_3, \varepsilon)$$

These transitions are also created nondeterministically at each step (since Σ is ε for both), and they assume that the midpoint symbol has been reached. If it is a 1, the 1 is popped off the stack & M_1 moves to q_3, where it begins popping symbols off the stack and comparing them to the input for symmetry. The same occurs for the case of the top symbol being a 0.

Ari Kumar

Due February 23, 2024

COMP 455 - 001

Homework 2

3. Informally describe a deterministic Turing Machine that recognizes $A = \{0^n 1^n \mid n \geq 0\}$.

We can design a Turing Machine M_1 that recognizes the language $A = \{0^n 1^n \mid n \geq 0\}$. M_1 works by zig-zagging between 0s and 1s on the tape and "crossing off" a 1 for every 0 read – we can indicate this "crossing off" by replacing crossed of 0s or 1s with the symbol x.

For M_1, let $\Sigma = \{0, 1\}$ and $\Gamma = \{0, 1, x\}$. In order to explain M_1 better, I also rewrite B as

$$B = \{ 0^{n_1} 1^{n_2} \mid n_1 = n_2 \text{ where } n_1, n_2 \geq 0 \}$$

M_1's algorithm given an input <u>w</u> is as follows:

M_1 = "on input string w:

1. If the first/leftmost symbol is a 1, reject – because this implies that either
   a) string w contains no 0s and at least one 1, in which case $w \notin B$
   b) string w contains 1s that precede 0s, in which case $w \notin B$
2. Write over the first 0 read (which, initially, should be the first symbol on the tape unless $w = \varepsilon$) with the symbol "x" in order to mark it off. Move right across the tape until a 1 is read.
3. Write over the first 1 read with an "x". Move left until the first 0 is read.
4. Repeat steps 2 and 3. If at any point a 0 is read and crossed off and then no more 1s are found (aka all 1s have been marked off, meaning that $n_2 < n_1$), reject.

   When all 1s have been crossed off, if any symbols (aka any 0s) remain, reject; otherwise, accept.

The following figure contains several non consecutive snapshots of M_1's tape after it has started on input 000111:

0	0	0	1	1	1	␣	␣	....
x	0	0	1	1	1	␣	␣	....
x	0	0	x	1	1	␣	␣	....
x	0	x	x	1	1	␣	␣	....
x	0	x	x	1	1	␣	␣	....
x	0	x	x	x	1	␣	␣	....
x	0	x	x	x	1	␣	␣	....
x	x	x	x	x	1	␣	␣	....
x	x	x	x	x	x	␣	␣	....

accept

Part 2: Computability Theory

Ch 3: The Church-Turing Thesis

3.1 Turing Machines

What is a Turing Machine?

→ A model of computation (just like DFAs, PDAs, etc.)

→ However, one that is much more powerful and can basically do everything that a "real computer" (e.g. Python) can do!

→ A much more accurate model of a general purpose computer because, unlike finite automatons, it has an unlimited & unrestricted memory.

How does a Turing Machine work, broadly?

→ The TM model uses an infinite tape as its unlimited memory

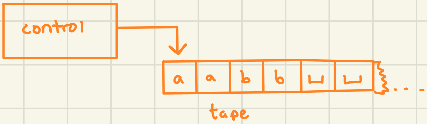
- it has a "tape head" that can read & write symbols onto the tape, as well as move around the tape!

How does a TM use its "tape"?

→ Initially, the tape contains the entire input string and is blank everywhere else.

→ if the machine wants to store info, it can write it onto the tape.

→ if the machine wants to read the info it has written, it can move its head back over it.



How does a TM produce its output?

→ It will continue computing until it enters a designated "accept" or "reject" state, at which point it produces an output.

- if it never enters an accept/reject state it will go on forever, never halting.

How is a TM different from a DFA?

→ Informally, a Turing Machine is just a DFA with an infinite tape!

→ Key differences:

1. a TM can both write on the tape AND read from it.
2. The read-write head can move both to the left and right – i.e., not limited to the top symbol like a PDA and its stack.
3. The special states for accept & reject take effect immediately (unlike FAs, where only the state at the end of an input string matters).

What is "L(M)"?

→ For a Turing Machine M, L(M) = "the language recognized by M"

Example to understand how TMs work?

→ Let's imagine a Turing Machine M_1 for testing membership in the language $B = \{ w\#w \mid w \in \{0,1\}^* \}$ (so like 011#011, 101#101, 0#0, etc.)

→ The input is too long for M_1 to remember all of it, but what it can do is move between the 2 sides of the # and check if the symbols match.

M_1's algorithm (informally)

1. Zig-zag across the tape to corresponding positions on either side of the # symbol to check whether these positions contain the same symbol.
   (w.r.t. the string w, e.g. the 2^{nd} symbol in the entire input should correspond to the 2^{nd} symbol after the #, and so on)
   - if they do not correspond at any point, reject.
   - if no # symbol is found, reject.
   - Cross off symbols as they are checked to keep track of which symbols correspond.
2. When all symbols to the left of the # have been crossed off, check for any remaining symbols to the right of the #. If any symbols remain, reject. Otherwise, accept.

How do we describe a Turing Machine informally?

→ by giving a description of the algorithm & sketching the way that it functions

→ We almost never give formal descriptions of TMs because they tend to be very big.

What is the formal definition of a Turing Machine?

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

1. Q is the set of states
2. Σ is the input alphabet NOT containing the **blank symbol** $\sqcup$
3. Γ is the **tape alphabet**, where
   - $\sqcup \in \Gamma$ (Γ contains the blank symbol)
   - $\Sigma \subseteq \Gamma$ (all elements of Σ are also a part of Γ)
4. $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$ is the transition function,
5. $q_0 \in Q$ is the start state
6. $q_{accept} \in Q$ is the accept state, and
7. $q_{reject} \in Q$ is the reject state, where $q_{accept} \neq q_{reject}$

What is the transition function δ for a TM?

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

L = left
R = right

- if a machine T_1 is in a certain state q and its head is currently over a tape square containing symbol a, we can represent this as $\delta(q, a)$
- and if $\delta(q_1, a) = (q_2, b, L)$, then the machine moves to state q_2, replaces the symbol a on the tape with a b, and the tape head moves to the left (L) after writing.

How does a Turing Machine compute?

→ Initially, a Turing Machine M receives its input $w = w_1 w_2 \ldots w_n \in \Sigma^*$ and writes them onto the leftmost n squares of the tape (b/c it writes from left-to-right)

- The rest of the tape is blank — aka filled with blank symbols, $\sqcup$.

→ The first time that a $\sqcup$ appears on the tape indicates/marks the end of the input, since Σ does not contain $\sqcup$.

→ The head starts on the leftmost square of the tape.

What happens when M starts running?

→ The computation proceeds/moves according to the rules described by the transition function

- For ex, say M starts in q_0 and the leftmost square contains the symbol c
- say that δ of M contains the following rules:

$$\delta(q_0, c) \to (q_1, b, R)$$
$$\delta(q_0, b) \to (q_0, c, L)$$

- Since "$\delta(q_0, c)$" corresponds to M's current situation, M's next action is to enter state q_1, replace the symbol c with a b, and move its head to the right.
- It continues this process for whatever state q_x and tape symbol t_x it currently rests on... $\delta(q_x, t_x) \to \boxed{}$

What happens if M is already on the leftmost square?

→ if the tape head is already on the leftmost square and the transition function indicates L, the head just stays in the same place.

→ The head will never be in the "rightmost square" since it will eventually just be blank symbols.

When does M stop running?

→ For every Turing Machine M on input w, running M on w results in exactly 1 of these possible outcomes:

1. As soon as/if ever M enters q_{accept}, it immediately accepts input w
   - And we say that "M halts on w"
2. As soon as/if ever M enters q_{reject}, it immediately rejects input w
   - And we say that "M halts on w"
3. M never halts (doesn't enter q_{accept} or q_{reject})
   - And we say that "M loops on w"
   - basically, a TM doesn't have to change states or modify its tape — all it has to do is keep moving around (left & right) while everything else stays the same.
   - i.e., $\delta(q_1, a) \to (q_1, a, L)$ is a valid transition function.

Summary (?) on how a TM works?	→ Basically, the TM first writes all input symbols onto its tape, and then proceeds to move around the tape and (potentially) rewrite/modify some of the symbols (according to the δ rules) in order to help it reach an output. • it does all this whilst simultaneously moving from state to state, and goes on forever OR stops if it ever lands on q_{accept} or q_{reject}
What does Turing-recognizable mean?	→ RECALL that a TM M can respond to an input w in 1 of 3 ways: • accepting it by reaching a q_{accept} state & halting operations. • rejecting it by reaching a q_{reject} state & halting operations. • looping indefinetly, moving between states & tape squares but never landing on the accept or reject state. → A language A is **Turing-recognizable** if there exists some TM M that recognizes it. This means that for every string x, $x \in A$ if M accepts x $x \notin A$ if M rejects x OR if M loops on x → We then say that "TM M recognizes language A".
What is a decider?	→ A Turing Machine that is programmed so that it never reverts to looping on <u>any</u> given input – every input string leads to either q_{accept} or q_{reject} – is called a **decider**.
What does Turing-Decidable mean?	→ A language A is **Turing-decidable** if there exists some TM M such that for every string x, $x \in A$ means that M accepts x $x \notin A$ means that M rejects x → aka, if there exists some <u>decider</u> that recognizes it! → We then say that "TM M decides language A." → All languages that are Turing-decidable are also inherently Turing-recognizable.

Summary: where do all types of languages fall in respect to each other?

→ regular languages:

- the easiest to solve
- can be recognized by DFAs
- recognizing machines don't need to keep track of how many symbols they've seen so far ... & they can't anyways because they have a finite number of states (limited memory!).
- $\boxed{\text{EX}}$ $L = \{ w \mid w \text{ ends in a } 1 \}$

→ context-free languages:

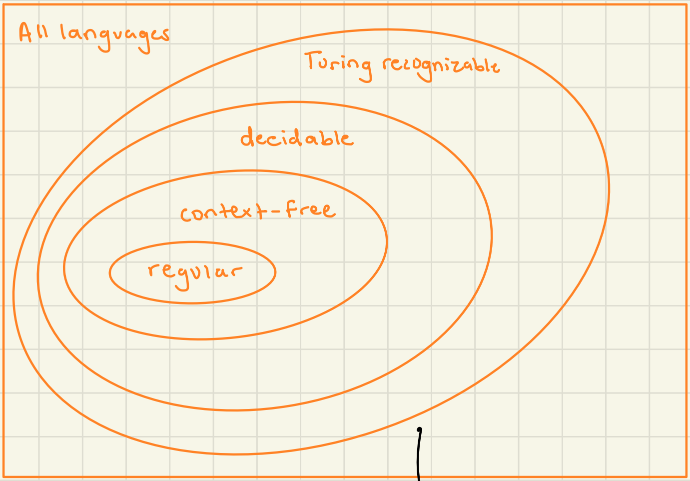
- recognizing machines are now equipped with a stack which gives them an "infinite memory" to keep track of what they've seen.
- $\boxed{\text{EX}}$ $L = \{ 0^n 1^n \mid n \geq 0 \}$, which is nonregular.

→ decidable languages:

- languages that are recognized by Turing Machines which always reject or accept any input, and never loop.
- $\boxed{\text{EX}}$ $L = \{ a^n b^n c^n \mid n \geq 0 \}$, which is non-context-free.

→ recognizable languages:

- languages that are recognized by Turing Machines
- $\boxed{\text{EX}}$ $L = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}$, which is undecidable.



note that there are also languages which do not fall into any of these categories!

Part 2: Computability Theory

Ch 3: The Church-Turing Thesis

3.2: Variants of Turing Machines

What is a "variant" in this context?	→ a variant of a Turing Machine is an alternative definition of a Turing Machine, a type of TM that alters one of the rules in some way. → For ex, a Turing Machine that has multiple tapes, a TM that includes an option to "stay put" (as well as move L or R), or a Turing Machine that employs nondeterminism.
Do TM variants differ in their computing power?	→ No! The original TM model and all of its variants are equivalent in power; that is, they all recognize the same class of languages → Similar to the relationship between coding languages like Python, Java, and C — some languages might provide a more efficient or intuitive way to complete a task, but all of them are ultimately equivalent in terms of what they have the power to do.
How can we prove that a variant isn't more powerful?	→ To show that 2 computing models are equivalent, we simply need to show that one can simulate the other. → For ex, a TM that allows the ability to stay put instead of being forced to move L or R. • its transition function would look like $\delta: Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$ → We know that this feature does not give the variant any more power because we can convert any TM with this "stay put" to a regular TM: • all we have to do is replace every transition $Q \times \Gamma \to Q \times \Gamma \times S$ with 2 transitions - one moving to the right, and one moving back to the left.
What is a multitape Turing Machine?	→ like a normal TM but with several tapes — each tape with its own head for reading & writing → The input is initially written onto tape 1, while the others stay blank
What is the transition function for a multitape TM?	→ To account/allow for reading, writing, and moving the heads on some or all of the tapes simultaneously, we modify $\delta: Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ to a new transition function $\delta: Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, S\}^k$, where k is the number of tapes. The expression $\delta(q_i, a_1, \ldots a_k) = (q_j, b_1, \ldots b_k, L, R \ldots, L)$ says that if the machine is in state q_i and heads 1 through k are reading symbols a_1 through a_k, then the machine moves to state q_j, writes symbols b_1 through b_k on the corresponding tapes, and moves each tape's head (starting with tape 1) left or right or to stay put, as specified.

What is a nondeterministic Turing Machine?

→ **Theorem**: every multitape Turing Machine has an equivalent single-tape (regular) Turing Machine.

→ At any point in a computation, the TM may proceed according to several possibilities (normal defn of nondeterminism that we've seen so far)

→ The transition function is $\delta : Q \times \Gamma \rightarrow P(Q \times \Gamma \times \{L, R\})$

→ **Theorem**: every nondeterministic Turing Machine has an equivalent single-tape (regular) Turing Machine.

Part 2: Computability Theory

Ch 3: The Church-Turing Thesis

3.3: The Definition of Algorithm

What does "algorithm" mean?

→ Informally speaking, an algorithm is just a collection of simple instructions for carrying out some task.

- Recipes, procedures — these are everyday use 'algorithms'

→ The understanding of what an algorithm <u>is</u> was, for a long time, just informal; an intuitive understanding.

Brief review:
what is a polynomial?

→ a sum of terms, where each **term** is a <u>product</u> of certain variables and a coefficient (a constant).

What is a root?

→ A root of a polynomial is an assignment of values to each of its variables such that the equation = 0.

- for ex, a root of $6x^3yz^2 + 3xy^2 - x^3 - 10$ is $x = 5, y = 3, z = 0$

→ An **integral root** is one where all the variable values are <u>integers</u> (like above)

Why do we need a formal definition of an 'algorithm'?

→ As we know, the basis of this class is the "limits of using algorithms to solve problems"

→ So, some problems/tasks do not have any algorithm that solves them.

- For ex, there is no algorithm that can test and determine whether a **polynomial** has an **integral root**.

→ Proving that an algorithm does not exist requires having a clear **definition** of algorithm.

What is the Church-Turing thesis?

→ The first formal definition of an algorithm!

→ 1936: Alonzo Church used a notational system called λ-calculus to define algorithms, and Alan Turing did it with his machines.

- the 2 definitions were shown to be equivalent.

→ $\boxed{\text{Intuitive notion of algorithms}}$ equals $\boxed{\text{Turing Machine Algorithms!!}}$

So what does it mean to solve a problem?

→ The formalization of an algorithm says that solving a yes/no problem P is equivalent to designing a Turing Machine M that **decides** a language A, where A is essentially a set of strings that consists of all the "yes" instances of the problem - all the possibilities that result in the ans being <u>yes</u>

- $A = \{ w \mid w \text{ is a "yes" instance} \}$
- RECALL: "decides" = always an accept or reject output; no looping.

How do we put a 'yes/no problem' into T.M. terms?

→ Lets go back to the example with polynomials – is there an algorithm that can test & determine whether a polynomial has an integral root?

→ Ideally, this algorithm should return the set of all polynomials w such that $w = 0$ has an integer solution. We can write this as language $D = \{ w \mid w \text{ is a polynomial with an integral root} \}$

- our alphabet for D would look like $\Sigma = \{0, 1, \ldots 9, +, -, *, y, \text{etc.}\}$

→ It was proven that D is **not** a Turing-Decidable language, though it is **recognizable**

→ $6x^3yz^2 + 3xy^2 - x^3 - 10$ would be an example of an accepted string by M.

Example of a problem that is decidable?

→ For polynomials with only one variable, like $4x^3 - 2x^2 + x - 7$. We can let language $D_1 = \{ p \mid p \text{ is a polynomial over } x \text{ with an integral root} \}$

→ A TM M_1 that **decides** D_1:

M_1 = "on input <p>: where p is a polynomial over the variable x.

1. Evaluate p with the vals of x set successively to 0, 1, -1, 2, -2, 3, -3, ... If at any point the polynomial evaluates to 0, **accept**.
2. If the roots of p do not lie within the bounds of $\pm k \frac{c_{max}}{c_1}$, where k = # of terms in p; c_{max} is the coefficient w/ largest absolute value; c_1 is the coefficient of the highest order term – **reject**.

What would a TM that recognizes D be?

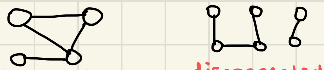
→ Similar to the TM M_1, but only with component 1, which defines the conditions to enter q_{accept}

- However, there is no way to calculate bounds to enter q_{reject}. Thus, M is merely **recognizable**.

What is another example of a problem that is **decidable**?

→ An algorithm (aka a Turing Machine!) that determines whether a given undirected graph is connected.

- **RECALL COMP 210:** an undirected graph is one with no arrows indicating direction
- **a graph is connected** if every node can be reached from every other node by traveling along the edges of the path. Basically that the whole graph is "one piece"



connected   disconnected

What would be the algorithm that solves it?

→ First, lets rephrase the problem as a language A, consisting of all <u>strings</u> representing undirected graphs that are connected:

$$A = \{\langle G \rangle \mid G \text{ is a connected undirected graph}\}$$

How would a <u>graph</u> be encoded as a string?

UNFINISHED

Part 2: Computability Theory

Ch 4: Decidability

4.1 Decidable Languages

What are some **decidable** problems that concern regular languages?

→ Algorithms that test aspects of the automatons that recognize regular languages — namely, DFAs and NFAs.

→ For ex, testing whether a finite automaton **accepts** a string, whether the language of a given finite automaton is empty, or testing whether 2 FAs are equivalent.

→ All of these problems **are decidable** — there is an algorithm that solves them.

How can we represent such 'problems' in TM terminology?

→ Like discussed before: by representing them as **languages**, where the string input / input alphabet basically provide some way to represent/define a specific DFA (or NFA) and its properties using a string of symbols

- the specifics of how the deciding TMs alphabet might look aren't really important b/c we aren't formally defining it anyway

Example of a decidable problem?

→ The **acceptance problem**: testing whether a particular DFA accepts a given string.

→ We can express this problem as a language A_{DFA}

$$A_{DFA} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$$

What does the language set A_{DFA} contain?

→ each element of A_{DFA} is an encoding of a DFA together with a string it accepts.

→ so A_{DFA} is basically a giant language that contains the encodings of **all DFAs (in existence)** together with each of the strings that they accept.

→ By expressing a computational problem as a language, we can more easily prove whether or not it is **decidable** — we just have to determine whether there exists a TM that decides the language.

- testing whether a DFA B accepts a language w = testing whether $\langle B, w\rangle$ is a member of the language A_{DFA}

What is a T.M. that decides A_{DFA}?

→ A T.M. M = "on input $\langle B, w\rangle$, where B is a DFA and w is a string:

1. If the input is not in the form of "DFA, then string"; reject
2. Simulate B running on input w
3. If the simulation ends with B in an accept state; accept. else reject."

→ Theorem: A_{DFA} is decidable.

What about the "acceptance test" problem for NFAs?

→ Since we already have proved that <u>any</u> NFA N can be converted into an equivalent DFA D, it follows that the language A_{NFA} is **also decidable**

- the TM that decides it just contains one extra step where it converts the NFA it receives as input into an equivalent DFA.
- the rest of the steps are identical to those of TM M which decides A_{DFA}.

What about the 'acceptance test' for regular expressions?	→ Same idea, since we have <u>also</u> already proven that any regular expression R can be converted into an equivalent NFA N.
	→ Theorem: A_{NFA} and A_{REX} are decidable languages.
What is the 'emptiness testing' problem?	→ For the language of a finite automation, it is the 'problem' of determining whether or not a finite automaton <u>accepts any strings at all.</u>
	→ We can write this problem as a language E_{DFA} : $E_{DFA} = \{\langle A \rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}$
	→ Theorem: E_{DFA} (and thus E_{NFA}, E_{REX}) are decidable.
What is the TM that decides E_{DFA} ?	→ If we simply do the reverse of the previous TM's strategy (which decided A_{DFA}) by running the DFA on the given input string and "reject if A accepts string, else accept", it **will not work** • when a given DFA that actually is a member of the language (aka a DFA which **does not accept any string**) is run by the TM, we expect the TM to accept it. • However, since A won't accept any string, the TM won't be able to automatically decide to accept it; instead, the TM will **loop on** A — which isn't what we want.
Alternative TM design?	→ Instead, we can design a TM (M_2) that tests A without having to refer to any input string. • A DFA accepts some string iff it is possible to reach an accept state from the start state by traveling along the arrows of the DFA (aka that there exists a pathway – e.g. a sequence of transitions/states – from the start state to an accept state.) • to test this condition, M_2 can begin by "marking" the start state & then considering <u>all possible pathways</u> that can emerge from it. If none of them result in the accept state; **reject**.
How do we write this M_2 idea into an (informal) description?	→ M_2 = "On input $\langle A \rangle$, where A is a DFA: 1. Mark the start state of A. 2. Repeat the following until no new state gets marked: 3. Mark any state that has a transition arrow pointing at it from a state that is already marked. 4. Once every state that could be marked has been: • if no accept state is marked, accept • else, reject

– Decidable problems for context-free languages –

→ RECALL: CFLs are generated by context-free grammars as well as PDAs (which are basically NFAs with a stack).

How do we express the 'acceptance problem' for CFGs as a language?

→ $A_{CFG} = \{ \langle G, w \rangle \mid G \text{ is a CFG that generates string } w \}$

→ Theorem: A_{CFG} is decidable.

What is the TM that decides A_{CFG}?

→ RECALL: When a grammar G is in **Chomsky Normal Form**, any derivation of a given input string w has exactly $2n-1$ steps, where n = the length of w.

→ TM S for A_{CFG}: S = "On input $\langle G, w \rangle$ where G is a CFG and w a string:

1. Convert G to an equivalent grammar in Chomsky Normal Form.
2. List <u>all</u> derivations with $2n-1$ steps
   - <u>except</u> if $n = 0$, then list all derivations with 1 step.
3. If any of these derivations generate w, accept. if not, reject.

How do we prove the statement "Every context-free language is decidable"?

→ This is a theorem that we can prove to be true by designing a Turing Machine that answers (aka "decides") it!

→ What we need: a TM that tests whether a given language A is decidable.

→ Our TM S from above example can be used on <u>any</u> CFG to determine whether or not it accepts a certain string.

- We can use this TM in our new TM to test every input for a given CFL

What is the TM that decides a CFL A?

→ Let G be a CFG for A. To design a TM M_G that decides A, we build a copy of G into M_G like this:

M_G = "On input w:

1. Run TM S on input $\langle G, w \rangle$
2. If this machine accepts, accept. If it rejects, reject.

Part 2: Computability Theory

Ch 4: Decidability

4.2: Undecidability

What does it mean for a problem to be alogrithmically unsolvable?

→ A problem for which we **cannot** devise an algorithm to consistently solve it.

→ A.k.a., a language which is not **Turing-decidable**!

→ There are problems that not even the full power of Turing machines, Python etc. can solve.

- when we take "solve" to mean **decidable** — meaning that looping isn't an option.

→ Even if we took "solve" to mean Turing-recognizable, there are still unsolvable problems.

What is an example of an **unsolvable** problem?

→ The problem of determining whether a Turing Machine accepts a given input string.

- a.k.a. the "acceptance problem" for TMs, described by

$A_{TM} = \{\langle M, w \rangle \mid M \text{ is a T.M. and } M \text{ accepts } w\}$

→ There is no algorithm — aka no Turing Machine — that can take in a T.M. and an input string, and accurately, decidedly tell you whether the string will be accepted or rejected.

→ Recall that we have already determined the acceptance problem for DFAs, CFGs, NFAs, and regular expressions (A_{CFG}, A_{DFA}, A_{NFA}, A_{REX}) to be **solvable**.

Is A_{TM} Turing-recognizable?

→ Yes! Recognizers are more powerful than deciders because the TMs are not required to halt (aka choose "accept" or "reject") on all inputs; they are allowed to loop.

- that requirement restricts the kind of languages that a TM can recognize.

What is a TM that **recognizes** A_{TM}?

→ U = "On input $\langle M, w \rangle$, where M is a TM and w is a string:

1. Simulate M on input w.
2. If M ever enters its accept state, accept. if it ever enters its reject state, reject.

→ the TM U isn't a decider of A_{TM} because it doesn't halt on every input — if M loops on w, then U will loop on input $\langle M, w \rangle$.

How can we prove that the language A_{TM} is undecidable?

→ For contradiction, let's assume that $A_{TM} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$ is decidable and there exists a TM H which decides A_{TM}

→ Definition of H: $H(\langle M, w\rangle) = \begin{cases} \text{halt and accept} & \text{if } M \text{ accepts } w \\ \text{halt and reject} & \text{if } M \text{ does not accept } w \end{cases}$

(For a machine x, $\langle x\rangle$ denotes the string description of x)

→ Now lets define a new Turing Machine D. D's algorithm does the following:

- D takes in a Turing Machine M as an input.
- It then calls the TM for A_{TM}, H in order to determine what M outputs when its input string is its own description, denoted $\langle M\rangle$.
  - → aka, D calls H on input $\langle M, \langle M\rangle\rangle$.
- Finally it outputs the **opposite** of H's output.

→ A more formal description of D:

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

1. Run H on input $\langle M, \langle M\rangle\rangle$
2. Output the opposite of what H outputs from this input. That is, if H accepts, reject. If H rejects, accept."

→ Basically, the **language** that D **recognizes** consists of all Turing Machines (obviously given in their string 'description' form) which **reject themselves.**

- Because D only accepts an input T.M. $\langle M\rangle$ if H rejects input $\langle M, \langle M\rangle\rangle$. And H only rejects $\langle M, \langle M\rangle\rangle$ if M rejects when its run on input $\langle M\rangle$.

→ Now we must ask the **Key question**: What happens when we run D with its **own description**, $\langle D\rangle$, as its input? Does D accept $\langle D\rangle$?

Does D accept $\langle D\rangle$?

→ No! Consider the following cases.

$\boxed{\text{Case 1}}$ D does accept $\langle D\rangle$.

- if D accepts the input $\langle D\rangle$, then working backwards from D's description above, this means that H has rejected its input, which is $\langle D, \langle D\rangle\rangle$. This then implies that when D is run on input $\langle D\rangle$, D rejects $\langle D\rangle$ — which is a direct contradiction of the case itself!

$\boxed{\text{Case 2}}$ D does not accept (aka rejects) $\langle D\rangle$

- D only rejects its input if H accepts its input $\langle D, \langle D\rangle\rangle$. H only accepts its input if D accepts when run on input $\langle D\rangle$ — again, a direct contradiction!

→ We can summarize D's behavior on input $\langle D\rangle$ as follows:

$$D(\langle D\rangle) = \begin{cases} \text{accept} & \text{if } D \text{ rejects/does not accept } \langle D\rangle \\ \text{reject} & \text{if } D \text{ accepts } \langle D\rangle. \end{cases}$$

→ No matter what D does, it is forced to do the opposite, which is obviously a contradiction. Therefore, neither TM D nor TM H can exist.

What is Theorem 4.22?

→ A language A is decidable if and only if
- A is Turing-recognizable, and
- the complement of A, $\bar{A}$, is also Turing-recognizable

What is the PROOF?

→ There are 2 directions to prove -

1. if we assume a language A is decidable, prove that both A and $\bar{A}$ are recognizable.
2. if we assume a language A as well as its complement $\bar{A}$ are both Turing-recognizable, prove that A is **decidable**.

Proof for direction 1?

→ Let M be a TM that decides A ... M also recognizes A, inherently.

→ We can prove that $\bar{A}$ – aka the set of all strings which are not in A – is also Turing-recognizable: all we need is a TM M_2 which incorporates M and outputs the opposite of M on input A!
- This is the same idea behind the proof that the complement of a decidable language is also decidable.

Proof for direction 2?

→ if both A and $\bar{A}$ are Turing-recognizable, let M_1 and M_2 be the recognizing TMs for A and $\bar{A}$, respectively.

→ We can then prove that A is decidable by devising a TM M which decides it:

M = "on input w:

1. Run both M_1 and M_2 on input w in parallel.
   (aka take turns simulating 1 step of each machine in turn)
2. If M_1 accepts, accept. If M_2 accepts, reject.

How do we know that this TM M actually decides A?

→ Every single string w is either in A or $\bar{A}$, which means that one of the 2 machines (M_1 and M_2) will always reach an accept state when given w

→ And because M halts whenever M_1 or M_2 accepts, it follows that M **always halts** – and so it is a decider.

What fact is proven by Theorem 4.22?

→ Corollary: the complement language of A_{TM}, $\overline{A_{TM}}$, is **not** Turing-recognizable.

Class Notes: CFGs, set notation

→ Given CFGs A and B, we can construct a CFG which accepts the languages $L(A) \cup L(B)$

but not necessarily $(L(A) \cap L(B))$, or $\overline{L(A)}$

Why not $(L(A) \cap L(B))$?

→ It isn't garunteed to be a context-free language! For ex;

let $L(A) = \{a^i b^j c^k \mid i = j \text{ and } i,j,k \geq 0\}$

let $L(B) = \{a^i b^j c^k \mid j = k \text{ and } i,j,k \geq 0\}$

→ the union of these languages would be language c,

$c = \{a^i b^i c^i \mid i \geq 0\}$,

which is **not** context free!

Why not $\overline{L(A)}$?

→ Again, not garunteed to be context free.

→ because of the set notation rule $S \cap T = \overline{\bar{S} \cup \bar{T}}$

- since we've proven that "$S \cap T$" isn't possible, "$\overline{\bar{S} \cup \bar{T}}$" is also not possible.

Part 2: Computability Theory

Ch 5: Reducibility

5.1: Undecidable Problems from Language Theory

What is reducibility?

→ The primary method we use to prove that problems are computationally unsolvable.

→ To prove that a problem is **undecidable**: show that some other problem already known to be undecidable, **reduces to it.**

What is a reduction?

→ A way of converting one problem A into another problem B in such a way that the solution to the second problem (B) can be used to solve the first problem (A).

→ NOTATION: "A is reducible to / reduces to B" $\approx$ $A \leq B$

→ Real life $\boxed{\text{EX}}$ of a "reducibility":

- The problem of finding your way around a city can be **reduced** to the problem of obtaining a map of the city
- The problem traveling from NY to LA → the problem of buying a plane ticket from NY to LA → the problem of earning money for the ticket → the problem of finding a job.

Why is reducibility important?

→ plays an important role in both **computability** and **complexity** theory (later).

→ In complexity theory – When A is reducible to B, solving A cannot be harder than solving B.

→ In computability theory – important for classifying problems by their decidability.

How do we use reducibility in computability theory?

→ if A is reducible to B, and B is **decidable** ... then A is also decidable!

→ if A is undecidable and A is reducible to B ... then B is also **undecidable**.

(the key to proving that various problems are undecidable!)

What is the "halting problem"?

→ the problem of determining whether a T.M. **halts** (by accepting or rejecting) on a given input.

→ Can be described by the language

$$HALT_{TM} = \{\langle M, w\rangle \mid M \text{ is a T.M. and } M \text{ halts on input } w.\}$$

Is $HALT_{TM}$ decidable?

→ No! We can use the already proven **undecidability** of A_{TM} (RECALL ch.4.2) to prove the halting problems undecidability by reducing A_{TM} to $HALT_{M}$.

What is the idea behind this proof?

→ Proof by contradiction: lets assume that we have a T.M. R which decides $HALT_{TM}$

- We can use this assumption to show that A_{TM} is **reducible** to $HALT_{TM}$, by using R to solve A_{TM}
- Showing that $A_{TM} \leq HALT_{TM}$ would then imply / assert that A_{TM} is also decidable
- However, we already know that this isn't true (Theorem 4.11) & thus a contradiction.

How do we use R (which decides $HALT_{TM}$) to "solve" A_{TM}?

→ Review: the job of a T.M. S which decides A_{TM} is to

1. take in an input of ⟨M, w⟩
2. output accept if M accepts w.
3. output reject if M rejects OR loops on w.

→ We can use R to test whether M even halts on w in the first place — meaning that the output of S depends solely on the garvanteed output of M on w

- or if M just loops, in which case S can immediately reject because ⟨M, w⟩ would not be in A_{TM}

What is the finalized $\boxed{\text{PROOF}}$ for thm. "$HALT_{TM}$ is undecidable"?

→ Lets assume for the purpose of obtaining a contradiction that T.M. R decides $HALT_{TM}$. We construct TM S to decide A_{TM}, operating as follows:

S = "On input ⟨M,w⟩, an encoding of a TM M and a string w:

1. Run TM R on input ⟨M,w⟩
2. if R rejects, reject.
3. if R accepts, simulate M on w until it halts.
4. if M has accepted, accept. If M has rejected, reject."

→ Clearly, A_{TM} is reducible to $HALT_{TM}$ because if R decides $HALT_{TM}$ then S decides A_{TM}. Because A_{TM} is proven undecidable, $HALT_{TM}$ also must be undecidable.

What is another problem whose undecidability can be proven by reduction?

→ the "emptiness testing" problem for a Turing Machine — the problem of determining whether or not a particular TM accepts any strings at all.

We can denote this problem as language

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

- $L(M) = \emptyset$: the lang that M recognizes is equal to "∅", aka the empty language.

How do we construct the proof?

→ Similar to the $HALT_{TM}$ proof, lets contradict and assume that there <u>does</u> exist a TM R which decides E_{TM}. R works like this (rough description):

- R accepts a TM M if M rejects every single possible input string — meaning its language contains nothing ($L(M) = \emptyset$)
- R rejects M if at any point it accepts some/any string.

→ **Goal:** to use R to construct a TM S which decides A_{TM}.

How can we use R to help solve A_{TM}?

→ When S is given an input ⟨M,w⟩ (M is a TM, w is a string), we first have it construct another TM M_1 using M and w.

How does M_1 work?

→ M_1 = "on input x:

1. If $x \neq w$ (w: the string initially inputted to S), reject.
2. If $x = w$, run the og machine, M, on input w. If M accepts w, accept."

Explanation: How does M_1 work?

→ Basically, say that M is a TM which does accept some language set of strings (M is not empty)

→ We take M, and we modify it such that it rejects every string (including strings that it might usually accept) except for the string w -- this is our TM M_1

→ But when M_1 does read the input w, instead of rejecting, it then runs the o.g. TM M on input w, and then outputs "accept" iff M accepts w.

→ Consider the following cases:

$\boxed{\text{Case 1}}$ M accepts a language A and $w \in A$

- M_1 then becomes a TM that rejects every string x (in A and otherwise) but accepts x when $x = w$.
- $L(M_1) = \{w\}$; M_1 is nonempty
- So, running R on M_1 outputs reject.

$\boxed{\text{Case 2}}$ M accepts a language A and $w \notin A$

- M_1 then becomes a TM that rejects every string x (in A and otherwise), and also rejects x when $x = w$.
- a.k.a., M_1 accepts no strings and its language is empty!
- So, running R on M_1 outputs accept.

What is the final $\boxed{\text{proof}}$ for "E_{TM} is not decidable"?

→ We assume by contradiction that TM R decides E_{TM} and construct TM S which decides $A_{TM} = \{\langle M, w\rangle \mid \text{TM } M \text{ accepts input } w\}$ as follows:

S = "On input $\langle M, w\rangle$:

1. Use the description of M and w to construct a TM M_1 as described (on prev. page)
2. Run R on input $\langle M_1 \rangle$
3. If R accepts, reject. If R rejects, accept.

→ If R were a decider for E_{TM}, S would be a decider for A_{TM}. A decider for A_{TM} cannot exist, so we know that E_{TM} must be undecidable.

How can we prove that EQ_{TM} is undecidable?

→ EQ_{TM} represents the problem of testing whether 2 TMs are equivalent. Let

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

→ So far, we have been proving that languages $HALT_{TM}$, E_{TM} are undecidable by showing that A_{TM} reduces to each of them. To prove EQ_{TM} undecidable, lets instead show that E_{TM} reduces to it... aka $E_{TM} \leq EQ_{TM}$.

→ Proof by contradiction: lets assume that EQ_{TM} is decidable by a TM R, and use this TM to construct a TM S which decides E_{TM}.

How can we prove that EQ_{TM} is undecidable?

(continued)

→ Proof by contradiction: lets assume that EQ_{TM} is decidable by a TM R, and use this TM to construct a TM S which decides E_{TM}.*

S = "On input <M>, where M is a T.M.:

1. Run R on input $\langle M, M_1 \rangle$, where "M_1" = a TM that rejects all inputs (aka, $L(M_1) = \emptyset$)
2. If R accepts, accept. If R rejects, reject."

*If confused by the logic behind TM S, see notes pg. 79

→ if R decides EQ_{TM}, then S decides E_{TM}. But since E_{TM} is undecidable by Theorem 5.2, EQ_{TM} also must be undecidable.

What is Rice's Theorem?

→ States that any language of the form $\{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ satisfies } P\}$, where P = some (nontrivial) property about languages, is undecidable.

→ For ex, E_{TM} (determining whether $L(M) = \emptyset$)

→ Examples: testing whether $L(M)$ is a CFL, a decidable language, or even a finite language.

Ari Kumar

Due March 8, 2024

1. Let M be a TM that loops indefinetly on all inputs. No matter what string w it is run on, M will loop indefinetly.

- From this description, we can conclude that <u>M is not a decider</u>. To be a decider, a Turing Machine must never loop on any given input; every input must result in the TM halting on an accept or reject state.
- The language of M, which loops on all inputs & never accepts, is then the empty language: $L(M) = \emptyset$
- <u>L(M) is a decidable language</u>. We can easily prove this by describing a TM M_1 which decides L(M):

  M_1 = "on input w:

  1. reject."
- M_1 is simply a TM that rejects all inputs. We know that M_1 is a decider because it never loops, and halts on every input. The language of M_1 is also $\emptyset$, aka L(M). Therefore, L(M) is decidable.

2.

a) If A is decidable, then $\bar{A}$ is decidable. — $\boxed{\text{True}}$

- If A is decidable, then there exists a TM M which decides it — that is, on every given input w, M definitively tells us whether or not w is an element of A. We know that the langvage $\bar{A}$ consists of all strings which are not an element of A. To prove that $\bar{A}$ is decidable, we can construct a TM M_1 that incorporates M:

  M_1 = "on input w:

  1. Run M on input w. If M accepts, **reject**. If M rejects, **accept**."
- Since M never loops, M_1 will never loop either & therefore decides $\bar{A}$.

b) if A is Turing-Recognizable, then $\bar{A}$ is Turing-recognizable. — $\boxed{\text{False}}$

- The above statement is <u>only true</u> if A is also decidable. If A is Turing-recognizable but not decidable, then $\bar{A}$ is not guaranteed to be recognizable.
- Theorem 4.22 (Sipser, ch 4.2 pg. 209) states — and proves — that a language is decidable iff both it and its complement are Turing-recognizable.
- We have proven in class (& in the textbook) that A_{TM} is Turing-recognizable. If $\overline{A_{TM}}$ were also Turing-recognizable, then (according to Thm 4.22), it would mean that A_{TM} is decidable — but we have already proven (in class) that A_{TM} is <u>not</u> decidable. Therefore, the complement of the recognizable lang. A_{TM}, $\overline{A_{TM}}$, is not recognizable. So the statement is False.

Ari Kumar

Due March 8, 2024

c) For any language A, $A \le_m \overline{A}$. — $\boxed{\text{False}}$

- This is false because A_{TM} cannot reduce to its complement. If $A_{TM} \le_m \overline{A_{TM}}$, then $\overline{A_{TM}} \le_m A_{TM}$ via the same mapping reduction.
- We know that if a language $A \le_m B$ and B is Turing-recognizable, then A is also Turing recognizable (Theorem 5.28, Sipser ch 5.3)*.
- if $\overline{A_{TM}} \le_m A_{TM}$, then it would imply that $\overline{A_{TM}}$ is Turing-recognizable (since A_{TM} is recognizable). However, we already know/have proven that $\overline{A_{TM}}$ is not recognizable (Theorem 4.22, Sipser ch 4.2). Therefore, A_{TM} is not reducible to its complement.

d) If A is decidable and $B \subseteq A$, then B is decidable. — $\boxed{\text{False}}$

- This statement is false and can be proven false via a simple contradicting example. Let B be any undecidable language — for example the language $HALT_{TM}$ discussed in class.

  then $B = \{\langle M, w\rangle \mid M \text{ is a T.M. and } M \text{ halts on } w\}$.
- Let $A = \Sigma^*$, the set of all strings over Σ. We know that $B \subseteq A$ because the language B consists of the elements $\langle M, w\rangle$. w is a string input, so all possible strings $w \in \Sigma^*$. M is also a string input — namely, a string encoding of a Turing Machine M. So all of the elements in language B are also in A; $B \subseteq A$.
- We know that the set Σ^* is decidable because there exists an algorithm which decides it — aka, one which can determine whether a given input is a member of Σ^*. A TM which takes a string w & "accepts" if $w \in \Sigma^*$ (or "rejects" if $w \notin \Sigma^*$) is a decider because its output will always be "accept", since the language Σ^* contains every string. So $A = \Sigma^*$ is decidable.
- A is decidable, and $B \subseteq A$. B is undecidable. Therefore this statement is false.

* The proof for this theorem was not explicitly described in class, but Dr. Sun explained that it is almost identical to the proof for the claim "if A is undecidable, then B is undecidable", which he did present (lecture from 3/6). For that reason, I didn't think it was necessary to prove the claim myself.

Ari Kumar

Due March 8, 2024

e) If A is decidable, then A^* is decidable. — $\boxed{\text{True}}$

- Given a TM M that decides A, we can construct a TM M_1 that decides A^* which would basically work like this (informal description):

  M_1 = "on input w:

  1. if $w = \varepsilon$, accept.
  2. nondeterministically split the string w into every possible set of substrings — aka, every single way to "partition" w into separate pieces.
  3. For each set of substrings $\{w_1, w_2 \dots w_k\}$: run M on all of the strings in the set. If M accepts every string in a set, accept.
  4. If M never accepts after repeating step 4 on every set of substrings, reject."

- **Resource Used:** "Closure Properties" notes from Univ. of Illinois: https://courses.engr.illinois.edu/cs373/fa2013/Lectures/lec26.pdf

3. Prove that the language $E = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) \cup L(B) \neq \emptyset\}$ is decidable.

To prove that E is decidable, we can construct a TM M which decides it. Specifically, on a given input $\langle A, B\rangle$, aka an encoding of DFAs A and B, M should determine whether at least one of the languages L(A) or L(B) is nonempty. If so, it should accept. If both languages are empty, M should reject.

To construct M, we can use a TM D which decides the language

$$E_{DFA} = \{\langle A\rangle \mid A \text{ is a DFA and } L(A) = \emptyset\},$$

which has already been proven to be decidable (Sipser Ch 4.1, Theorem 4.4). M runs as follows:

M = "On input $\langle A, B\rangle$, where A and B are DFAs:

1. Run TM D on input $\langle A\rangle$.
2. If D rejects, accept.
3. If D accepts, run D on input $\langle B\rangle$.
4. If D rejects, accept. If D accepts, reject."

Explanation:

We know that M is a decider of E because its output is dependent on the output of D, and we know know that D will never loop because E_{DFA} is a decidable language. In other words, M is a reduction of D ($M \leq D$) and we know that the reduction of a decidable language is always decidable.

Ari Kumar
Due March 8, 2024

4. Let $\Sigma = \{0,1\}$ and let A be any decidable language with alphabet Σ. Prove that $A \leq_m B$ where $B = \{00, 11\}$.

We know that for languages C_1 and C_2, if $C_1 \leq C_2$ and C_2 is decidable, then C_1 is also decidable (proven in class). Since we already know that A is decidable, B must also be decidable (if it is true that $A \leq B$). All that we have to prove is that A is <u>mapping-reducible</u> to B by providing a computable function f that takes an input x (x is a string from alphabet Σ) and returns an output string f(x) such that $x \in A$ iff. $f(x) \in B$.

Let R be a TM that decides A. The following machine F computes a reduction f:

F = "On input x where x is a string of alphabet Σ:

1. Run R on input x.
2. If R accepts, output 00
3. If R rejects, output 2."

→ This reduction proves that $A \leq_m B$ because we can use function f to map elements of A to elements of B.

Part 2: Computability Theory

Ch 5: Reducibility

5.3: Mapping Reducibility

What is mapping reducibility?

→ One way to formalize the notion of **reducibility** (reducing one problem to another)... e.g. "A is reducible to B"

→ There are several ways to formalize this notion; mapping reducibility is one 'type' of reducibility.

→ if $A \leq_m B$, it implies inherently that $A \leq B$ ... mapping reducibility is a special/confined case of general reducibility.

What is a computable function?

→ **DEFN:** a function $f: \Sigma^* \longrightarrow \Sigma^*$ ("Σ^*" ≈ the alphabet, so f takes a string from the alphabet as its input, & outputs some string as well) is a **computable function** if there exists a TM M such that for every input w, $M(w)$ ends with/halts with just $f(w)$ on its tape.

- $M(w)$ = "running M on string w"

→ it basically means that there is a TM which can **compute** f -- that, given an input w, its final output on the tape is $f(w)$.

How does a TM compute a function?

→ Unlike when using a TM to solve a language, a TM M for a function **does not halt on an "accept" or "reject" state** (to then return its output)

→ Instead, it computes by starting with the input to the function on the tape, and **halting** with the "answer": the output of the function on the tape.

Example of a computable function?

→ All usual arithmetic operations on integers are computable functions! For example, the operation $m+n$.

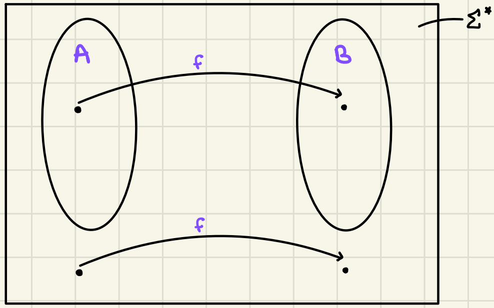
- We can create a TM that takes $\langle m,n \rangle$ as its input and returns $m+n$!

What does it mean for a language to be "mapping-reducible"?

→ **DEFN:**

> Language A is <u>mapping reducible</u> to language B, denoted $A \leq_m B$, if there exists a **computable function** $f: \Sigma^* \rightarrow \Sigma^*$, such that for every string w,
>
> $$w \in A \iff f(w) \in B$$
>
> - aka "for every string w, w is an element of A if and only if $f(w)$ is an element of B."
> - The function f is then called the **reduction** from A to B.

What is a diagram representation of function f reducing A to B?



☆ f should bring elements of A into B, and bring non-elements of A "not into" B.

What is the point of a mapping reduction?

→ A mapping reduction of $\underline{A}$ to $\underline{B}$ provides a way to convert questions about membership testing in A, to membership testing in B.

→ If one problem is mapping reducible to another, **previously solved problem**, we can then obtain a solution to the og problem!

→ a mapping reducibility is sort of a **translation**: a way to translate **any** string s.t. if the og string is in A, then performing the function produces a translation which is in B.

- And if the og string is not in A, the translation should **not** be an element of B.

How do you use mapping reduction to test membership?

→ For $A \leq_m B$: to test whether $w \in A$, we use the **reduction** f to **map** w to $f(w)$, and then test whether $w \in B$.

What is a simple example of proving a mapping reducibility?

→ Let $A = \{ w \mid w \text{ starts with a } 0 \}$ and let $B = \{ w \mid w \text{ starts with a } 1 \}$

→ To show that $A \leq_m B$, we need to define a function whose **input** and **output** is a string, and where inputs from A must output elements of B, and inputs not in A must output elements not in B.

→ $\boxed{\text{Solution}}$ Reduction $f(x)$:

- if $x = \Sigma$ (empty string): return Σ (or anything else not in B)
- $y[1] = 1 - x[1]$ — the first character/digit in string y should be equal to 1 minus the 1st char/digit in string x

  (basically "flips" the 1st digit of x; if its 0 it becomes 1 & vice versa)
- return y

→ $\boxed{\text{Proof}}$: need to prove both directions

1) "if $x \in A$, then $f(x) \in B$"
   - x starts with a 0, so by $f(x)$, y will start with a 1. Thus $f(x) \in B$

2) "if $x \notin A$, then $f(x) \notin B$"
   - if x doesn't start with a 0, y will not start with a 1. Thus $f(x) \notin B$

RECAP: How does mapping reducibility fit into the idea of reducibility as a whole?

→ We have, at this point, 2 definitions of reducibility:

1. General reducibility, e.g. $A \leq B$

→ **DEFN**: if A is reducible to B then, given a decider TM for B, we can design/create a **decider** TM for A.

- a.k.a., if B is decidable and $A \leq B$, A is also decidable.
- if A is undecidable, then B is also undecidable.

→ **EXAMPLES** (recall proofs from ch. 5.1):

- $A_{TM} \leq HALT_{TM}$
- $E_{TM} \leq EQ_{TM}$
- $A_{TM} \leq E_{TM}$
- **RECALL** $A_{TM} \leq HALT_{TM}$: w/o $HALT_{TM}$, there is no way to make a decidable TM for A_{TM} because it would be at risk of looping on any given input $\langle M, w \rangle$
  - with access to a decider for $HALT_{TM}$, we can avoid this issue by plugging $\langle M, w \rangle$ into $HALT_{TM}$'s decider to figure out whether M will halt at all.

→ **SIGNIFICANCE**: The typical strategy for proving some language is undecidable is to show that A_{TM} reduces to it (via a contradiction proof like in the examples in ch. 5.1), since we already know that A_{TM} is undecidable.

2. Mapping reducibility, e.g. $A \leq_m B$

→ **DEFN**: A is mapping reducible to B if there exists a computable function s.t.

$\forall$ string w, $w \in A \iff f(w) \in B$

→ **IMPLICATIONS**: when $A \leq_m B$,

- if B is decidable, then A is decidable (Theorem 5.22)
- if A is undecidable, B is undecidable
- if A is not Turing-recognizable, B is not Turing recognizable

→ **SIGNIFICANCE**: If we want to make an even stronger statement and prove that some language X is not even Turing-recognizable, the strategy is to show that the language $\overline{A_{TM}}$ is mapping reducible to it (e.g. $\overline{A_{TM}} \leq_m X$), because we already know/have proven that $\overline{A_{TM}}$ isn't recognizable.

What is the proof for theorem 5.22?

→ **Thm: if $A \leq_m B$ and B is decidable, then A is decidable.**

→ Let M be the decider for B, and f be the reduction from A to B. We can construct a TM N which decides A as follows:

N = "on input w:

1. Compute $f(w)$
2. Run M on input $f(w)$ and output whatever M outputs.

explanation?

→ $\boxed{\text{Case 1}}$ $w \in A$, meaning we want $N \to$ accept: Step 1 will produce a string $f(w)$ which is $\in B$, so M will accept it and N will output whatever M does – aka N will accept!

→ $\boxed{\text{Case 2}}$ $w \notin A$, meaning we want $N \to$ reject: Step 1 produces a string $\notin B$; M rejects; N rejects!

How do we prove that a language is mapping reducible to another?

→ To show that a language $A \leq_m B$, we start by assuming that we have a decider TM R for B.

→ Then, we need to construct a decider TM S for A. We do this by filling in / "completing" this specific template for S's definition:

Decider S for A on input x :

1. Compute $\boxed{y = f(x)}$ — This is the part that we need to complete!
2. Run TM R on y and return its output.

→ The part of this "template" that we have to specify is the details of the reduction function $(f(x))$ itself — a.k.a., the details of the translation of elements from A to elements in B.

What is another example of a mapping reducibility?

→ RECALL: In ch 5.1, we used a "general" reduction from A_{TM} to prove that $HALT_{TM}$ is undecidable (that $A_{TM} \leq HALT_{TM}$)

→ We can also demonstrate that $A_{TM} \leq_m HALT_{TM}$! To do this, we must present a computable function f that takes input of the form $\langle M, w \rangle$ (a.k.a., the format of elements of A_{TM}), and returns output of the form $\langle M', w' \rangle$, such that:

$\langle M, w \rangle \in A_{TM}$ if and only if $\langle M', w' \rangle \in HALT_{TM}$.

How do we create a function to satisfy this condition?

→ RECALL:

- an input $\langle M, w \rangle$ is $\in A_{TM}$ if the TM M accepts string w.
- $\langle M, w \rangle \notin A_{TM}$ if the TM M either rejects or loops on w.
- an input $\langle M, w \rangle$ is $\in HALT_{TM}$ if the TM M halts upon reading string w. it doesn't matter whether M accepts or rejects w b/c both of those imply that M has halted.
- $\langle M, w \rangle \notin HALT_{TM}$ if the TM M loops on w.

→ From these statements alone, we can start to see how we might map the elements of A_{TM} to $HALT_{TM}$

→ We need to design a new TM M' to map results from A_{TM} to $HALT_{TM}$. M' should work like this:

- if the og machine M accepts w, then M' should halt on w.
    - it doesn't really matter whether we design M' to "accept" or "reject" in this scenario, b/c either of those would imply that M' has halted, and thus would be accepted by $HALT_{TM}$.
- if the og machine M rejects or loops on w, then M' should loop (so that it will be rejected by $HALT_{TM}$.

So what is our reduction for $A_{TM} \leq_m HALT_{TM}$?

→ The following machine F computes a reduction f :

F = "On input $\langle M, w \rangle$:

1. Construct the following machine M';

   M' = "on input x :

   1. Run M on x.
   2. If M accepts, accept.
   3. If M rejects (or loops - this is implied), enter a loop."

2. Output $\langle M', w \rangle$."

What does the language EQ_{TM} represent?

→ The problem of determining whether 2 Turing Machines **recognize the same language**.

$EQ_{TM} = \{ \langle A, B \rangle \mid A \text{ and } B \text{ are TMs and } L(A) = L(B) \}$

How can we prove that $E_{TM} \leq_m EQ_{TM}$?

→ Goal: devise a computable function f s.t. $x \in E_{TM}$ i.f.f. $f(x) \in EQ_{TM}$

→ RECALL: $E_{TM} = \{ \langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset \}$

→ Notice that in this example, our output is of a different form than our input! $f(x)$ must take in an input of a single TM "M" (aka $f(\langle M \rangle)$), but output an encoding in the format of the language EQ_{TM}; that is, it must output 2 turing machines $\langle M_1, M_2 \rangle$

What will be our computable function f ?

$f(x)$, where $x = \langle M \rangle$:

1. If x is not a description/encoding of a TM : return 0.

(we can usually omit this line/statement b/c its obvious & we aren't concerned with that level of specificity)

2. let M_1 be an encoding of a TM, and set $M_1 = M$.
2. let M_2 be an encoding of a TM, and set M_2 = "reject."

(M₂ is a machine which always outputs reject... aka a machine whose language is $\emptyset$!)

3. return $\langle M_1, M_2 \rangle$.

Why does this reduction work?

→ $\boxed{\text{Case 1}}$ $\langle M \rangle \in E_{TM}$, which means that the language of M $L(M) = \emptyset$.

- $f(x)$ will output $\langle M_1, M_2 \rangle$ where $M_1 = M$ (so $L(M_1) = \emptyset$) and M_2 = a TM who always rejects (so $L(M_2) = \emptyset$)
- Therefore $L(M_1) = L(M_2)$ and $\langle M_1, M_2 \rangle \in EQ_{TM}$!

→ $\boxed{\text{Case 2}}$ $\langle M \rangle \notin E_{TM}$, which means that $L(M)$ is not $\emptyset$.

- $M_1 = M$ and $L(M_2) = \emptyset$ so $L(M_1) \neq L(M_2)$, so $\langle M_1, M_2 \rangle \notin EQ_{TM}$!

How can we create a reduction f for $A_{TM} \le_m \overline{E_{TM}}$?

→ Note that: $\overline{E_{TM}} = \{ \langle M \rangle \mid M \text{ is a TM and } L(M) \neq \emptyset \}$ "M is a TM that accepts something"

→ We need to create a reduction function f that maps like so:

$f(\langle M, w \rangle) = \langle M_1 \rangle$ ... s.t. $\langle M, w \rangle \in A_{TM}$ i.f.f. $\langle M_1 \rangle \in \overline{E_{TM}}$

(takes an A_{TM}-format input and returns an $\overline{E_{TM}}$ format output, which as we can see is a single TM encoding).

→ RECALL that we already proved that $A_{TM} \le E_{TM}$ (general reduction) in ch 5.1 (see notes pg. 68-69) by creating a TM M_1 which is a "modified" form of input TM M. M_1 accepted w if M accepts w, rejected w if M rejects w, and rejected any string put it into it that isn't w (regardless of whether it is a part of L(M)).

→ RECALL that our final reduction $A_{TM} \le E_{TM}$ involved outputting the opposite of E_{TM}'s decider's output when it ran on M_1.

- Since we are now trying to map to $\overline{E_{TM}}$, aka the opposite of E_{TM}, all we need to do is return M_1 itself!
- read notes pg 68-69 if confused.

→ Back to $A_{TM} \le_m \overline{E_{TM}}$: For an input $\langle M, w \rangle$, lets map out the possible cases:

1. M accepts w → $\langle M, w \rangle \in A_{TM}$ → need to return a nonempty TM M_1 so that $M_1 \in \overline{E_{TM}}$
2. M rejects w → $\langle M, w \rangle \notin A_{TM}$ → need to return an empty TM M_1 so that $M_1 \notin \overline{E_{TM}}$

What is our final reduction f ?

→ $f(\langle M, w \rangle)$:

- let M_1 be a new TM described as so:

  M_1 = "on input x:
  1. if $x \neq w$: reject.
  2. if $x = w$: run the og TM M on w. if M accepts w, accept."

- return $\langle M_1 \rangle$

How do we prove that f mapping-reduces A_{TM} to $\overline{E_{TM}}$?

→ By considering the possible cases of applying f on a given $\langle M, w \rangle$:

- "forward direction":

  M accepts w ($\langle M, w \rangle \in A_{TM}$) → M_1 accepts w → M_1 is nonempty bc it accepts at least w → $M_1 \in \overline{E_{TM}}$ ✓✓

- "backward direction":

  M does not accept w ($\langle M, w \rangle \notin A_{TM}$) → M_1 accepts nothing → M_1 is empty → $M_1 \notin \overline{E_{TM}}$ ✓✓

Still left:

- proving $A_{TM} \le_m E_{TM}$
- pollev qs at end

Can A_{TM} be mapping reduced to E_{TM}?

→ No.

→ RECALL that in ch 5.1, we proved that $A_{TM} \leq E_{TM}$ — that A_{TM} can be "generally reduced" to E_{TM}, which we proved in order to then prove that E_{TM} is undecidable.

- Did this by using an (assumed) decider TM for E_{TM} to create a decider for A_{TM}

→ Theorem: $A \leq_m B$ i.f.f. $\overline{A} \leq_m \overline{B}$ (aka negating both sides).

→ $\boxed{\text{Proof}}$ Assume by contradiction that $A_{TM} \leq_m E_{TM}$. According to the Thm. above, this means that $\overline{A_{TM}} \leq_m \overline{E_{TM}}$. According to Theorem 5.28, for languages A, B where $A \leq_m B$, if B is Turing-recognizable, then A is Turing-recognizable. We know that $\overline{E_{TM}}$ is Turing-recognizable (proven separately in textbook) — which would imply that $\overline{A_{TM}}$ is also T-recognizable. However, we already know that $\overline{A_{TM}}$ is not recognizable (notes pg. 65); therefore, the statement $\overline{A_{TM}} \leq_m \overline{E_{TM}}$ is a contradiction, proving that A_{TM} is not mapping-reducible to E_{TM}.

Is mapping reducibility an equivalence relation?

→ No. To be an equivalence relation, an operation has to satisfy 3 properties — reflexive, symmetric, and transitive. Mapping reducibility satisfies 2 out of the 3.

What properties does mapping reducibility have?

→ reflexive: yes. For any language A, $A \leq_m A$. $f(x) = x$

→ transitive: yes. For any languages A, B, C, if $A \leq_m B$ and $B \leq_m C$, then $A \leq_m C$

- if $B \leq_m C$ by reduction f and $A \leq_m B$ by reduction g, we can create a reduction h for $A \leq_m C$ by having h apply reductions g and f, consecutively.

→ symmetric: No. if $A \leq_m B$, we cannot assume that $B \leq_m A$.

Midterm 1 Study Guide/key points

Decidable problems

→ $A_{DFA, NFA, CFG, REX, PDA}$, $E_{DFA, NFA, CFG, REX, PDA}$,

$EQ_{DFA, NFA, REX, CFG, PDA}$

Undecidable problems

Problem	Proof
A_{TM}	diagonalization
E_{TM}	$A_{TM} \le E_{TM}$
$HALT_{TM}$	$A_{TM} \le_m HALT_{TM}$
EQ_{TM}	$E_{TM} \le_m EQ_{TM}$
$\overline{E_{TM}}$	$A_{TM} \le_m \overline{E_{TM}}$

Unrecognizable

→ $\overline{A_{TM}}$. Proof: Theorem 4.22

Summary:

- $A_{TM} \le E_{TM}, HALT_{TM}$
- $E_{TM} \le EQ_{TM}$ (and thus $A_{TM} \le \cancel{=} EQ_{TM}$)
- $E_{TM} \le_m EQ_{TM}$
- $A_{TM} \le_m HALT_{TM}$
- $A_{TM} \le_m \overline{E_{TM}}$

Midterm Review

What is COMP 455 about?

→ the limits of using algorithms to solve problems.

→ code, like Python code, can ultimately be translated into a Turing Machine ... TM ≈ Python

→ DFAs/NFAs, PDAs, CFGs are all "formal" models of algorithms (simpler ones)

→ "solving a problem" ≈ accepting/rejecting a string

Ch.1: Regular Languages

→ Regular language: A language A is a regular language i.f.f. there exists some DFA, NFA, or regular expression that describes it.

→ Regular expressions, DFAs, and NFAs are all equivalent in their computing power.

→ **Properties**: for 2 regular languages A and B,

- $C = A \cup B$ is a regular language
- $C = A \circ B$ is a regular language
- $C = \overline{A}$ is a regular language (take a DFA for A and swap all the accept and nonaccept states)
- $C = A^*$ is a regular language
- $C = A \cap B$ is a regular language:
  - if A reg $\Rightarrow \overline{A}$ reg, and if B reg $\Rightarrow \overline{B}$ reg ... so if $A \cup B$ reg, then $\overline{A} \cup \overline{B}$ also reg ... and if $\overline{A} \cup \overline{B}$ is reg, then $\overline{\overline{A} \cup \overline{B}}$ is also reg.
  - $\overline{\overline{A} \cup \overline{B}}$ equivalent to $A \cap B$ ... therefore $A \cap B$ is regular.

Regular Operations

→ $A = \{happy, sad\}$ $B = \{boy, girl\}$

→ Union: $A \cup B = \{x \mid x \in A \text{ or } x \in B\} = \{happy, sad, boy, girl\}$

→ Concatenation: $A \circ B = \{xy \mid x \in A \text{ and } y \in B\} = \{happyboy, happygirl, sadboy, sadgirl\}$

→ Star: $A^* = \{x_1 x_2 x_3 \dots x_k \mid k \geq 0 \text{ and each } x_i \in A\} = \{\varepsilon, happy, sad, happyhappy, happysad, sadsad, sadhappysad \dots\}$

Star op. always includes ε!

→ Intersection: $A \cap B = \{x \mid x \in A \text{ and } x \in B\} = \{\ \}$ ... basically all elements that are common between the 2.

Regular Expressions

→ DEFN: expressions describing languages — which are just sets of strings!

→ $0 \cup 1$ = a reg. expression describing lang $\{0, 1\}$

→ 0^* = language of all strings containing any # of 0s $\{\varepsilon, 0, 000, 00 \dots\}$

→ $(0 \cup 1) 0^* = (0 \cup 1) \circ 0$ ... concat. symbol is implicit ... lang. of all strings that begin w/ either 1 or 0, and proceed to contain any # of 0s.

→ Σ^* = the language of all strings of any length over the alphabet Σ.

Ch.1 : Regular Languages

Deterministic Finite Automata

→ DEFN: A DFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$

→ Example: DFA M for language $A = \{w \mid w$ contains at least one 1, and an even # of 0s follows the last 1$\}$.

Description

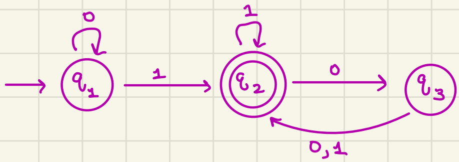
$M = (Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3\}$
2. $\Sigma = \{0, 1\}$
3. δ is described as:

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

4. q_1 is the start state, and
5. $F = \{q_2\}$

State Diagram



→ DFA rules:

- every state must have exactly 1 exit transition arrow for each possible input symbol (it can be 1 arrow sufficing for mult. symbols, but cannot have a state that doesn't have an exit arrow for some symbol).

→ Transition function: $\delta : Q \times \Sigma \longrightarrow Q$

- given a state (aka some element of Q) and an input symbol (aka some element of Σ), the output/result is a state ($\in Q$).

Nondeterministic Finite Automata

→ DEFN: A NFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$

- An NFA is basically a DFA except
  - a) we are allowed to have ε as a symbol, and
  - b) for each symbol s and state q: any # of arrows with symbol s can leave q — aka <u>no arrows</u>, 1 arrow, or up to $|Q|$ arrows, where "$|Q|$" = the number of states in the NFA.

→ Transition Function: $\delta : Q \times \Sigma_\varepsilon \longrightarrow P(Q)$

- Given a state & an input symbol, the result is some element of $P(Q)$. $P(Q)$ is the power set of all possible subsets of the set of states, Q.
  So the result of the function is **one of these subsets** — aka one of the elements of $P(Q)$

→ When 2 possible arrow choices to follow, machine "splits" into 2 copies that follow each 'path'.

→ Converting NFA to DFA !! See notes.

Ch.1 : Regular Languages

Nonregular Languages & the Pumping Lemma

→ Example : $A = \{0^n 1^n \mid n \geq 0\}$

→ Pumping Lemma : if A is a regular language, then there is a number p — aka the pumping length — where for any string s in A, where

$|s| \geq p$ (the length of s is <u>at least</u> p),

Then s can be divided into 3 pieces/substrings,

$s = xyz$

s.t. the following conditions are satisfied :

1. for each $i \geq 0$, $xy^iz \in A$
   - E.g., if $y = 01$ then $xy^1z = x01z$; $xy^3z = x010101z$; $xy^0z = xz$ ($y^0 = \varepsilon$)
2. $|y| > 0$ (the length of the 'y' substring is greater than 0)
3. $|xy| \leq p$. (the substrings 'x' and 'y' together are not longer than the pumping length p)

→ **How to use the pumping lemma to prove a lang. 'B' is nonregular:**

1. Assume that the lang is regular in order to obtain a contradiction
2. Use the p.L. to 'garuantee' a pumping length p s.t. all strings in B which are length $\geq p$ can be "pumped." (basically assert this claim in order to later contradict it)
3. Find a specific string s which is $\in B$ where $|s| \geq p$, but which **cannot be pumped** (aka the 3 conditions above can't be satisfied). Make an assertion about this string being unpumpable (will prove it in next step).
   - s doesn't have to be a specific string, it can be like 0^p1^p, bc we know that $|s|$ would have to be $\geq p$
4. Demonstrate that s can't be pumped by considering **all ways of dividing** s into x, y, z. (taking conditions 2 and 3 into acc't)
5. For each potential 'division' of s, **find a value** i s.t. the string $xy^iz \notin B$

→ To see example of a formal proof, see HW 1 !!

→ (Informal) proof EX: prove that $A = \{0^i 1^j \mid i \leq 3j\}$ (there can only be <u>at most</u> 3x as many 0s as 1s)

- Assume to the contrary that A is regular & thus satisfies the p.L.
- let p be the pumping length & choose s to be the string $0^{3p}1^p$ (for ex, if $p = 2$ then $s = 00000011$). We can show that string $s = 0^{3p}1^p$ cannot be pumped.
- Ways to divide s and how they contradict the p.L. :
  - → y consists only of 0s (for ex, let $p = 2$ then $s = 00000011$; $x = 0$, $y = 0$, and $z = 000011$ since $|xy| \leq 2$.)
    - no matter what p is, if y is some # of 0s then the string xy^iz will have more than 3x the 0s as 1s and therefore won't be a member of A, violating the p.L. condition 1.
    - for ex, if $y = 0$ then $xy^2z = 000000011$ ... $i = 7$ and $j = 2$, $7 \not\leq 3(2)$
  - → y consists only of 1s (for ex, let $p = 2$ and $s = 00000011$; $x = 000000$, $y = 1$, $z = 1$)
    - this splitting of s already violates condition 3... no way to split s s.t. $y = 1$ and $|xy| \leq p$.
  - → y consists of 0s and 1s — also violates condition 3.

Ch.1 : Regular Languages

Nonregular Languages & the Pumping Lemma

→ basically, for each possible "split", show that they violate at least 1 of the 3 conditions.

→ Like in the ex from prev page, we can only even split s into x,y,z s.t. y is all zeroes... the other divisions automatically eliminated b/c of cond. 3. We then consider that "valid" split & show how it violates cond. 1.

→ In a proof, we have to generalize/state that we don't know what p is and are "imagining" it to be some number.

Ch.2: Context-Free Languages

→ Context-Free Language: All languages which can be recognized by a CFG or a PDA

→ CFG & PDAs equivalent in computing power.

→ All regular languages are also context free (but not necessarily vice versa)

→ Ex: $A = \{0^n 1^n \mid n \geq 0\}$ ... CFG G_1 which generates A: $S_1 \to 0 S_1 1 \mid \varepsilon$

Context Free Grammars

→ DEFN: A CFG is a 4-tuple (V, Σ, R, S), where

1. V is the finite set of variables (e.g. R_1, R_2, R_3 etc.)
2. Σ is the finite set (disjoint from V) of terminals (aka input alphabet)
3. R is the set of rules (e.g. $R_1 \to a R_1 b$)
4. $S \in V$ is the start variable.

→ Leftmost Derivation: Deriving a string from a grammar s.t. at every step, you always replace the leftmost variable first (rather than just randomly)

→ Ambiguous Grammar: A grammar that can generate the same string in more than one way — aka, there exist 2 or more leftmost derivations that generate the same string.

- Not every ambiguous grammar can be modified to be/converted into an unambiguous grammar, but some can.

→ **Thm:** every DFA can be converted into an equivalent CFG (see ch.2 notes)

→ Chomsky Normal Form: A CFG is in CNF if every rule is of the form $A \to BC$ or $A \to a$, where $a \in \Sigma$ and $A, B, C \in V$ ... except B or C can't be the start variable. Also, the rule $S \to \varepsilon$ is allowed iff. S is the start variable.

the Rules for a CNF CFG

- the start variable cannot be on the right-hand side of a rule
- no "unit rules" allowed, aka where the r.h.s. is just a single variable (bc then you should just replace it w/ whatever that var points to, to eliminate the "middle man")
  - Ex $A \to B$, $B \to 1 \mid 0$ is NOT CNF but $A \to 1 \mid 0$ is.
- the RHS of a rule can't contain a combo of terminals & symbols (eg $A \to 1C1$) — can only be all terminals or all symbols
- if the RHS is made of terminals, it can be max 1 terminal (?)
- if the RHS is made of symbols, it must be exactly 2 symbols (no more, no less)... (e.g. $A \to B$ and $A \to BCD$ NOT allowed)

→ **Thm:** every CFG can be converted into Chomsky Normal Form

Ch.2: Context-Free Languages

Pushdown Automata

→ Implicitly nondeterministic.

→ a PDA is basically an NFA with a stack.

→ DEFN : a PDA is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where

1. Q = set of states
2. Σ = input alphabet
3. Γ = stack alphabet
4. $\boxed{\text{Transition function :}}$ $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow P(Q \times \Gamma_\varepsilon)$

Domain
- takes in the current state, the next input symbol being read, and the current stack symbol **on top of the stack** (since that's the only one which can be read)
- $\Sigma_\varepsilon \approx \Sigma + \varepsilon$ ... $\delta(Q \times \varepsilon \times \Gamma_\varepsilon)$ indicates a move the PDA makes w/o **reading an input symbol** (aka automatic move due to nondeterminism)
- $\Gamma_\varepsilon \approx \Gamma + \varepsilon$ ... $\delta(Q \times \Sigma_\varepsilon \times \varepsilon)$ indicates a move made w/o **reading (aka popping) a stack symbol**

Range
- outputs the power set of possible next moves (since nondeterministic)
- each possible "next move" : a **state** (b/c PDA will either remain at current state or move to a new one) and an **stack symbol, including** ε (b/c PDA may (or may not) write some new symbol $\in \Gamma_\varepsilon$ onto the stack).

5. $q_0 \in Q$ = start state
6. $F \subseteq Q$ = set of accept states

→ transition functions in state diagrams indicated by $a, b \rightarrow c$ where
- a = next input symbol read ...if $a = \varepsilon$ its an automatic (nondeterministic) move being made
- b = the symbol currently on top of the stack, which may get popped off & replaced by c
  (if $b = \varepsilon$, indicates a transition made **without** popping anything off the stack)
- c = the symbol that may be **pushed on top of the stack** as part of the transition...i.f.f. the current top symbol is b
  (if $c = \varepsilon$, indicates transition made **without pushing** anything onto the stack)

→ $a, \varepsilon \rightarrow c$. upon reading a, the PDA pushes c onto the stack

→ $a, b \rightarrow \varepsilon$: upon reading a, the PDA pops b off the stack but pushes nothing

→ $a, \varepsilon \rightarrow \varepsilon$: upon reading a, the stack **does not change**

→ $a, b \rightarrow c$: upon reading a, **i.f.f.** the current top stack symbol is a b, the PDA then pops b off & pushes c on to the stack.

Ch. 3: Church-Turing Thesis

Turing Machines

→ implicitly deterministic (for now)

→ Features/Key points:

→ a model of computation (like DFAs, PDAs, etc.) — goes from state to state, contains an accept state, etc.

→ Like a PDA except instead of a stack, has an unlimited tape that it can read, write to, & move around w/o restrictions

→ What is diff about TM versus other automations:

- Can both write to & read from any point on the **unlimited** tape
- the read-write head can move both left & right.
- When TM enters an accept or reject state, it takes effect **immediately** — don't need to wait till end of input string.

→ Transition Function: $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$

- $\delta(q, a) = (q_2, b, L)$ -- if the TM is currently in state q and its head is over a square w/ symbol $\underline{a}$, the TM moves to state q_2, replaces the "a" with "b", and moves the tape head <u>Left</u> after writing

→ DEFN: A TM is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ where

1. Q = set of states
2. Σ = input alphabet
3. Γ = tape alphabet
4. $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$
5. $q_0 \in Q$ = start state
6. q_{accept} and q_{reject} are the accept & reject states

→ Computation Process: For every input string, a TM either accepts, rejects, or loops

Recognizable vs Decidable

→ Recognizable Languages: languages for which there exists a TM which **recognizes** it — aka, for every string x

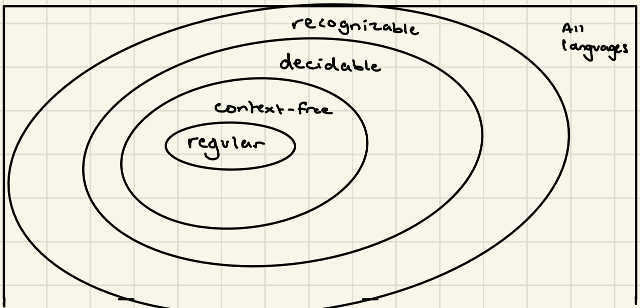
$x \in A$ if the TM accepts x

$x \notin A$ if the TM rejects or loops on x

→ Decidable language: a TM M which decides language A: for every string x,

$x \in A$ if M accepts x

$x \notin A$ if M rejects x



Ch. 4 : Decidability

Decidable Languages

→ $A_{DFA, NFA, CFG, REX, PDA}$, $E_{DFA, NFA, CFG, REX, PDA}$, $EQ_{DFA, NFA, REX, CFG, PDA}$

Undecidable Languages

language	proof
A_{TM}	diagonalization
E_{TM}	$A_{TM} \leq E_{TM}$
$HALT_{TM}$	$A_{TM} \leq_m HALT_{TM}$
EQ_{TM}	$E_{TM} \leq_m EQ_{TM}$
$\overline{E_{TM}}$	$A_{TM} \leq_m \overline{E_{TM}}$

→ Unrecognizable : $\overline{A_{TM}}$

→ Thm : A language A is decidable i.f.f. A and $\bar{A}$ are Turing recognizable.

Reducibility

→ To prove that a language B is **undecidable**: show that $A_{TM} \leq B$ ... assume that a decider TM M_1 exists for B, and use M_1 to design a decider TM for A_{TM}.

→ To prove that a language B is **unrecognizable** : show that $\overline{A_{TM}} \leq_m B$ ... create a computable function $F(x)$ s.t. $\forall$ string x, $x \in \overline{A_{TM}}$ iff $F(x) \in B$

- converting the input of $\overline{A_{TM}}$ into an input for B.

Rules

→ for $A \leq B$ ... if B is decidable, A is also decidable

- if A undecidable, B also undecidable

→ for $A \leq_m B$ ... same as above PLUS

- if A not recognizable, B not recognizable.

→ If A decidable, $\bar{A}$ decidable

→ if A decidable, A^* decidable

→ decidable langs
→ undecidable langs
→ proving that a lang is decidable
→ proving undecidable
→ designing a TM ??

Part 3: Complexity Theory

Ch 7: Time Complexity

7.1 Measuring Complexity

What is complexity theory?

→ So far, we have discussed the concept of whether or not an algorithm exists for a certain problem (or if it is unsolvable)

→ Now, we shift to discussing the resources that it takes to solve a certain (decidable) problem — e.g., how much time, memory, etc.

- more concerned with comparing/analyzing decidable problems, than determining the decidability of a problem (which is what **computability theory** was focused on).

→ Complexity theory: An investigation of the time, memory, and other resources required for solving computational problems.

What is time complexity theory about?

→ Key question (of this chapter): How much time is required to decide a decidable language?

→ In complexity theory, we classify computational problems according to their **time complexity**.

What is a worst-case analysis?

→ Since we are discussing decidable problems, we will be analyzing their decider TMs to evaluate time complexity. Specifically, lets think in the context of single-tape, deterministic TMs.

→ worst-case analysis is a way to evaluate the speed of an algorithm (aka a TM!) where we consider the longest running time of all input strings of a certain length.

What does "time" mean in this context?

→ In the (possibly oversimplified) context of STDTMs, we take "running time" and the 'amount of time' a TM takes to solve a problem to mean **the number of steps** that the TM takes befor reaching accept/reject

- aka, the number of transition function moves — reading an input, moving around on the tape, reading next input, etc.

→ So worst-case analysis = Given an algorithm, among all inputs of length x, what is the max number of steps the TM takes?

What is "running time"?

→ The running time, aka the **time complexity** of an algorithm, is the number of steps that it takes to solve a problem in the worst case, as a function of the input length.

→ Formal **DEFN**: For a deterministic, decider TM M, the **running time** of M is the function $f: \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n.

- We use n to represent the length of an input (customarily)
- If $f(n)$ is the running time of M, we say that "M is an $f(n)$ Turing Machine" and that "M runs in time $f(n)$"

What is asymptotic analysis?	→ A way to estimate the exact running time of an algorithm in order to understand the running time of the algorithm when it is run on large inputs.
	→ For the running time expression of an algorithm, consider only the highest-order term (aka term with largest exponent), and disregard the coefficient of that term as well as any lower order terms (b/c they are insignificant in comparison).
	• For $\boxed{\text{EX}}$, for the function $f(n) = 6n^3 + 2n^2 + 20n + 45$, we say that f is asymptotically at most n^3
What is big-O notation?	→ The formal way to describe this relationship between the running time expression, $f(n)$, of an algorithm, and its asymptotic estimation.
	• $\boxed{\text{EX}}$ the big-O notation for the expression above is $f(n) = O(n^3)$
	→ **DEFN:** Let f and g be functions $f, g : \mathbb{N} \longrightarrow \mathbb{R}^+$. ($\mathbb{R}^+ =$ set of all nonnegative real numbers) Say that $f(n) = O(g(n))$ if positive integers c and n_0 exist such that for every integer $n \geq n_0$, $$f(n) \leq c\,g(n)$$
	• When $f(n) = O(g(n))$, we say that $g(n)$ is an asymptotic upper bound for $f(n)$.
	→ Basically, big-O estimation gives you a run time that is less than or equal to the exact run time if we are are disregarding 'insignificant' differences up to a constant factor.
	→ $\boxed{\text{Ex}}$ the expression $f(n) = 2^{O(n)}$ represents an upper bound of $\underline{2^{cn}}$ for some constant c.
What is an example of a time analysis of an algorithm?	→ Lets analyze the run time of the single-tape deterministic T.M. M_1 which decides the language $A = \{0^k 1^k \mid k \geq 0\}$
	→ M_1 = "on input w: 1. Scan across the tape and reject if a 0 is found to the right of a 1. 2. Repeat if both 0s and 1s remain on the tape: 3. While the tape has at least one 0 and at least one 1, scan across the tape, crossing off a single 0 and a single 1. 4. If 0s still remain after all 1s have been crossed off, or if 1s still remain after all 0s crossed off, reject. Else, if everything is crossed off, accept."
How do we analyze M_1?	→ We can determine the time complexity of M_1 by considering each of its 4 steps separately, and adding the times together.

What is the time complexity of each step?

1. In stage 1, the TM scans across the entire tape to verify that no 0s appear after a 1.
RECALL that the head of a TM by default begins at the leftmost tape symbol. So if a given input string is n symbols long, then it takes M_1 $\underline{n}$ steps to do this scan.

- Additionally, M_1 repositions its head back to the start of the tape once it is done — another n steps taken to do this

→ So the total used in this stage, for any input string w where $|w| = n$, is $2n$ steps.
Using the rules of asymptotic analysis (aka big-O notation), we say that stage 1 uses $O(n)$ steps (since we disregard the coefficient '2').

2. & 3. In these stages, M_1 repeatedly scans across the entire tape, crossing off 2 input symbols (a 0 and a 1) during each scan.

- Each of these scans take n steps (since M_1 has to read every symbol from left to right) — aka "$O(n)$ steps"
- Since after each scan, the total # of symbols to read decreases by $\underline{2}$ (b/c 2 get crossed off), the machine only needs to perform this scan action <u>at most</u> $n/2$ times.
- Therefore, the total time taken by stages 2 and 3 is $n \cdot n/2 = \frac{1}{2}n^2 = O(n^2)$ steps!

4. In this step, the machine makes just a single scan to decide whether to accept or reject. So the max steps taken is $\underline{n}$; i.e. $O(n)$ steps.

How do we use this to determine the overall time complexity of M_1?

→ Add up each big-O and then apply the "asymptotic analysis" rules:
The running time of M_1 – aka the total time of M_1 on an input of length n – is then $O(n) + O(n^2) + O(n)$, which is equal to $\underline{O(n^2)}$ after disregarding lower order terms.

"M_1 is a STDTM that decides A in $O(n^2)$ time."

What is the "time complexity class"?

→ For any <u>function of n $t(n)$</u> – aka like $n, n^2, 2^n, n\log n$, etc – the time complexity class **TIME ($t(n)$)** is the collection/set of all languages that are **decidable** by an $O(t(n))$-time Turing Machine.
(presumably an STDTM?)

$$\text{TIME}(t(n)) = \{ B \mid B \text{ is decidable by an } O(t(n)) \text{ time STDTM} \}$$

→ From the $\boxed{\text{Ex}}$ above, we know that the language $A = \{0^k1^k \mid k \geq 0\}$ is an **element of** TIME (n^2)

- Because M_1 decides A in time $O(n^2)$, and TIME (n^2) is the set of all langs that can be decided in $O(n^2)$ time.

Why is TIME (1) a subset of TIME (n^2)

→ The TIME ($t(n)$) sets naturally form subsets, e.g. $\text{TIME}(1) \subseteq \text{TIME}(n) \subseteq \text{TIME}(n^2) \subseteq \ldots$
because any TM can easily become <u>less</u> efficient... a TM that decides a lang in n steps (aka $O(n)$ time) can also decide that language in n^2 steps if it wants to.

Is there a STD Turing Machine which decides A more quickly?

→ Actually, yes; the following STDTM M_2 shows that $A \in TIME(n \log n)$:

(copy down later)

→ **Thm:** Any language that can be decided in $o(n \log n)$ (small-o) time on a single tape deterministic TM, is a regular language

— Complexity Relationships Among Computation Models —

What is a key distinction between computability theory & complexity theory?

→ The discussion of time complexity highlights an important distinction between complexity & computability theory:

- In computability theory, the Church-Turing thesis implies that all reasonable models of computation for some language are **equivalent** (e.g., a DFA, PDA, TM, and multitape TM which all decide the language B)

→ However, in **complexity theory**, we see that the choice of computational model **does** make a difference — different models (like all the ones we've learned about so far) can have different **time complexities** for the same language!

- For ex, a single-tape TM decides the language A (from example) in $O(n^2)$ or at most $O(n \log n)$ time. But a **two-tape TM** M_3 (see textbook pg 281) decides A in $O(n)$ time.
- The complexity of A depends on the model of computation selected.

So then what model do we use to classify computational problems?

→ The time requirements don't actually differ significantly for typical deterministic models; i.e., our classification system isn't very sensitive to relatively small differences in complexity.

- Therefore, we can continue to use the **single tape deterministic TM** as our "formal model" used to classify a language's time complexity

What is the relationship between single- and multi- tape TMs (in terms of complexity)?

→ **Thm:** Let $t(n)$ be a function (where $t(n) \geq n$) that describes the running time of a **multitape** Turing Machine M_1 (aka, M_1 is a $t(n)$-time T.M.)

Every $t(n)$-time multitape TM M_1 can be converted to an equivalent **single tape** Turing Machine M_2, which will take at most $(t(n))^2$ steps; aka an equivalent $t^2(n)$-time STDTM.

- Proof is basically that for every step the $t(n)$-time MTDTM takes, the STDTM will use **at most** $t(n)$ steps to replicate the actions of that single step of the MTDTM. Since the MTDTM takes at most $t(n)$ steps to compute, and the STDTM uses at most $t(n)$ steps for each of those steps, the STDTM therefore will have a big-O runtime of $t(n) \cdot t(n) = \underline{t^2(n)}$ maximum steps.

What is the relationship between deterministic & nondeterministic TMs (re complexity)?

→ **Thm:** Let $t(n)$ be a function, where $t(n) \geq n$. Then for every single-tape **nondeterministic** Turing Machine of $t(n)$-time, there exists an equivalent $2^{O(t(n))}$-time **deterministic** Turing Machine.

→ Key takeaways about model relationships:

- There is at most a square (or "**polynomial**") difference between the time complexity of a problem measured on a STDTM versus a MTDTM
- There is at most an **exponential** difference between the time complexity of a problem measured on a deterministic versus nondeterministic STTM.

What do we know about the time complexity of nonregular languages?

→ **Thm:** If B is a nonregular language, then $B \notin TIME(f(n))$ for any $f(n) = o(n \log n)$

small-o ≈ "less than"

- This basically means that the running-time for all nonregular languages can never be less than $n \log n$; that is the lowest possible running time.
- NOTE: For any positive integer x, $x < x \log x < x^2 < 2^x$

→ By this theorem, we can then see that since the language $A = \{0^k 1^k \mid k \geq 0\}$ is nonregular, there does not exist an $O(n)$-time decider for it (since $n < n \log n$)

What about regular languages?

→ **Thm:** If B is a regular language, then B can always be solved by some TM in $O(n)$-time! aka, $B \in TIME(n)$

Why is this the case?

→ If a language is regular, that means there exists a **DFA** which recognizes it (RECALL ch1.1)

- A Turing Machine can easily decide a regular lang by simply imitating (calling the DFA which recognizes it — it just does everything the DFA does
- Since the DFA takes exactly 1 step upon each input symbol it reads, it takes the DFA "at most" n steps to compute upon an input of length n ... aka $O(n)$ time!
- And since the TM imitates the DFA, it also computes in $O(n)$ time.

Which model of computation do we use?

CLARIFY: Why doesn't it matter which model we use to classify the complexity of problems?

→ We just discussed how the same language can be decided by an $O(n^2)$-time STDTM, an $O(n)$-time MTDTM, or even an $O(n)$-time Python function

- These seem like big differences in time complexity... why doesn't it matter?

→ ANS: Because for the purposes of a theory course, we can pretty much just view all **polynomial running times** as equivalent. e.g.

$O(n) \approx O(n^2) \approx O(n^{25}) \approx \ldots$

Why do we treat all polynomials as equivalent?

→ Several reasons:

1. Any n – polynomial function is still eventually going to be smaller than a function where n is the exponent, even the largest polynomials:

$$n^{999} > n^2 \quad , \quad \text{BUT} \quad n^{999} < 2^n < 3^n$$

2. Treating polynomials as equivalent allows us to be agnostic to the model: don't have to care too much about the details, can work with whichever model you want b/c not worried about complexity differences.

3. In practice, we rarely see runtime polynomials as big as n^{999} anyway... Problems decidable in polynomial time are almost always solvable in real time (aka by a human being manually) anyway.

Part 3: Complexity Theory

Ch 7: Time Complexity

7.2: The Class P

What is a polynomial?
→ A number where n is the base and the exponent is some constant, e.g. n, n^2, n^{999}, $3n^2$, $5n^3$, etc.

What is an exponential?
→ A number where n is the exponent, e.g. 2^n, $2^{O(n)}$, 3^n, etc.

So?
→ In this class/subject – the field of complexity – small differences in run time don't matter (eg, a runtime of $O(n \log n)$ v.s. $O(n^2)$ v.s. $O(n)$ v.s. $O(n^3)$ etc.)
- The only difference we care about is **polynomial v.s. exponential**

What is the difference between the two?
→ Polynomial runtimes equate to problems which could be deemed "easy", "small", "fast", "tractable", etc.
→ On the other hand, exponential runtimes equate to problems deemed "large", "hard", "slow", "intractable"
→ For more info about why we don't distinguish between diff polynomials, see notes prev. ch (pg 97-98), or textbook pg 284-286

What is the class P?
→ A set of languages (RECALL defn of a "class"). Namely, the class of all languages which are decidable on a single-tape deterministic T.M. in **polynomial time**.

$P = \{ L \mid L \text{ is decidable by a polynomial-time STDTM} \}$

→ Formally: $P = \bigcup_k \text{TIME}(n^k)$
- aka, the union of the sets $\text{TIME}(n^0)$, $\text{TIME}(n)$, $\text{TIME}(n^2)$ ... $\text{TIME}(n^k)$
- $\underline{K}$ is a boundless constant; placeholder for ∞ since technically, "n^∞" isn't a polynomial.

Why is the class P important to complexity theory?
1. The elements (aka problems / "languages") in P are invariant for all models of computation which are **polynomially equivalent to the STDTM**
   - This includes ND TMs, regular TMs, MTTMs, all of the other models we've learned so far, and even the "Python model"! a.k.a. the speed that a problem could be solved by a Python-code function.
2. P roughly corresponds to the class of problems that are realistically solvable on a computer.

How do we show that an algorithm is an element of P?
→ To analyze an algorithm to show that it runs in polynomial time, have to do several things:
1. Describe the algorithm with numbered "stages" (like we've been doing; see Ex of M_1 on pg. 94)
2. Give a polynomial upper bound (usually in big-O notation) on the number of stages that the algorithm uses when run on an input of length $\underline{n}$.
3. Examine the individual stages in the description to ensure that each can be implemented in polynomial time on a reasonable deterministic model.

What is an example of a problem in the class P?

→ The PATH problem: to determine whether a directed path exists between 2 nodes on a directed graph. In other words,

PATH = { <G, s, t> | G is a directed graph that has a directed path from vertice s to vertice t. }

→ **Thm:** PATH ∈ P.

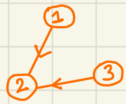
What is a directed graph?

→ A data structure (RECALL COMP 210 lmfao) defined by <u>vertices</u> (aka nodes) and <u>edges</u>, where the edges are directed. Specifically, a directed graph is a pair of sets $G = (V, E)$, where:

- V is the set of vertices, for ex $\{1, 2, 3\}$
- E is the set of edges, where each element of E is a <u>pair of vertices</u> depicting the start & endpoint of the edge.

→ And a **path** is a sequence of vertices that you can make by following the directed edges and without repeating vertices.

→ For ex, graph $G_1 = (V, E)$ $V = \{1, 2, 3\}$ $E = \{(1,2), (3,2)\}$ can be described by this diagram:



How is a directed graph "encoded" (for a computer to interpret it)?

→ As an **adjacency matrix**: an $n \times n$ array where $n = |V|$

- Every entry $[i, j]$ is 1 if an edge from i to j exists, and 0 if not.
- For example, G_1 could be represented as `int G[][] = {{0,1,0}, {0,0,0}, {0,1,0}};`, which would look like this:

$$\begin{array}{c c} & \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \end{array} & \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \end{array}$$

What would be an exponential-time algorithm for PATH?

→ The **brute-force search** of examining every single path in G and determining if any is a directed path from s to t.

- This method would help us prove that PATH is decidable, but doesn't prove that it can be solved in polynomial time.

What is a better, polynomial-time algorithm for PATH?

→ To employ a graph-searching method, such as breadth-first search. Basically, lets devise an algorithm that starts at vertice s and successively "marks" all nodes in G which are reachable from s by a path length of 1, then 2, then 3, and so on until m -- the # of nodes in the graph aka the maximum possible path length. Then, we just check to see if t was marked.

Formal definition of the algorithm?	→ Let M = "On input <G,s,t> where G is a directed graph with nodes s and t : 1. Place a mark on node s. 2. Repeat until no additional nodes are marked: 3. Scan all the edges of G. If an edge (a,b) is found going from a marked node 'a' to an unmarked node 'b', mark node b. 4. If t has been marked, accept. Else, reject."
What is the time analysis for this algorithm?	→ Stage 1: only executed once, $O(1)$ time → Stage 2/3: each time that stage 3 is performed, it marks off at least one additional node in G (except for the last time, at which point the algorithm moves on to stage 4). Since a graph G has m nodes, then stage 3 must run a maximum of m times. $O(m)$ time, where "m" is roughly less than or equal to the size of the input
How do we know that this alg runs in polynomial time?	→ Stage 4: only executed once. → Overall, this algorithm then uses a total number of at most $1+1+m$ stages, which gives a polynomial that is in the size of G. Hence, M is a polynomial time algorithm for PATH
What is another example of an element of P?	→ An algorithm that determines whether an integer is an element of a certain array (Think coding-level, like a Java or Python array). e.g. $A_1 = \{ \langle B,t \rangle \mid t \in B$, where B is an int array and t is an int $\}$ • The running-time of A_1 is $O(n)$ -- just scan the array.

SKIPPED:

Every CFL is in P

Part 3: Complexity Theory

Ch 7: Time Complexity

7.3: The Class NP

Do we always know whether an algorithm is solvable in polynomial time?

→ No! For example, the algorithm to determine whether an `int[]` array contains a subset which can sum to a certain number, aka

A_2 = { <B, t> | There exists some subset of B that sums to t }

- We can easily devise a brute-force algorithm to solve this language — for every possible subset, sum the numbers & check if it equals t — but this would not be polynomial. It would have $O(n \cdot 2^n)$ time.
- **Nobody knows** whether this problem is $\in P$ // solvable in polynomial time.

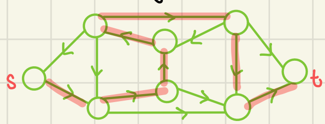
Another example?

→ A notable example of a problem whose polynomial-time algorithm has yet to be discovered is HAMPATH.

HAMPATH = { <G, s, t> | G is a directed graph with a Hamiltonian path from s to t. }

What is a Hamiltonian path?

→ A directed path that passes through <u>each node/vertice</u> exactly once. For example,



The red highlighted Hamiltonian path from s to t.

What do these 2 problems (A_2 and HAMPATH) have in common?

→ They <u>seem</u> impossible to solve in polynomial time ...maybe they are? But it hasn't yet been proven.

→ They are more "difficult"* to devise a non-brute-force algorithm for.

→ Though we can't prove the <u>absolute "hardness"</u> of such problems (aka we can't prove that HAMPATH $\in$ P but we also can't definitively prove that HAMPATH $\notin$ P), we can prove statements about the **relative difficulty** of problems (aka reductions! Like in ch. 2)

- "A ≤ B" ≈ A reduces to B ≈ A is **at most** as hard as B
- "If this problem is hard/easy, then this problem must also be hard/easy"

*

"easy" ≈ solvable in polynomial time
"difficult" ≈ not solvable in polynomial time

→ Also - such problems are difficult to <u>solve</u>, but easy to **verify**.

What does it mean to "verify" a problem?

→ For a given problem, if someone provides a specific element that they claim is in that problem language, 'verifying' is being able to determinedly conclude whether or not that claim is true.

→ Basically: <u>verifying</u> the existence of a hamiltonian path is much easier than <u>determining</u> its existence.

→ This property is very widespread/common among problems not in P (like HAMPATH) and is called **polynomial verifiability**.

What is polynomial verifiability?

→ A feature of a problem where it can **verify** whether a certain object is an element of it in **polynomial time**.

Are all problems polynomially verifiable?

→ **No!** For example, the complement of the HAMPATH problem,

$\overline{\text{HAMPATH}} = \{\langle G, s, t\rangle \mid G$ does not have a Hamiltonian path from s to $t\}$.

- The only way to verify that a graph doesn't contain some specific hamiltonian path is to exhaustively check every possible path, which, as we've already discussed, can't be done by a polynomial-time algorithm.

What is a verifier?

→ **DEFN:** A verifier for a language A is an algorithm (aka a Turing Machine) V where

$$\boxed{A = \{w \mid \text{There exists a string } c \text{ (possibly dependent on } w\text{) s.t. } V \text{ accepts } \langle w, c\rangle.\}}$$

What is "w"?

→ w is a (string) input in the form of the input that the original language A takes

What is "c"?

→ c is the additional information that the verifier uses to **verify whether** string w is a member of A, called the "certificate / proof" (of membership in A)

- Usually, c takes the form of a "proposed solution". For example, if verifying that a path is a ham. path, c would be a string encoding of a <u>specific graph path</u>.

How do we measure the time of a verifier since its input has 2 strings?

→ RECALL that the "n" in $O(n)$ refers to the length of the input string

→ We measure it only in terms of the length of w (not c).

- A **polynomial-time verifier** then runs in polynomial time in the length of w.
- For polynomial verifiers, the certificate c inherently has polynomial length (in the length of w), because that is all that the verifier can access in its time bound.

→ Verifiers usually aren't super clever

How does the machine V work?

→ Let's look at an example of making a verifier for HAMPATH to help understand:

1) <u>specify</u> the certificate c: In this case, the input for the certificate depends on the details of the input graph itself, so we can write it as a function of the graph:

   $c(G, s, t)$ = a specific Hamiltonian path from s to t (i.e. a list of vertices, which is what a "path" is.

2) The input to a verifier is both the a) original input to the problem A, and b) the certificate c: V = "on input $(\langle G, s, t\rangle, c)$:

   1. Check that c is a hamitonian path from s to t.
   2. If so, accept. Else, reject.

→ Similarly, the certificate for a verifier for $A_2 = \{\langle B, t\rangle \mid \exists$ some subset of B which sums to t$\}$ would just be some subset of B. The verifier simply adds up the numbers to see if they sum to t.

What is the class NP?

→ **DEFN:** $NP = \{A \mid$ There exists a polynomial-time verifier for $A\}$

- NP is the class of all languages/problems that have verifiers that run in polynomial time.

What is the relationship between P and NP?

→ **Fact:** $P \subseteq NP$; all problems in P are also in NP

- Why? Because verifying is easier than solving. Any polynomial-time decider of a problem automatically yields a verifier. This verifier doesn't even need a certificate — it can just run the decider machine on its input!

Is P = NP?

→ No one knows! We haven't yet discovered a problem which is in NP but definitively not in P. (Because problems like HAMPATH aren't disproven to be in P; an algorithm just hasn't been found yet.

→ We don't know if using a nondeterministic TM actually gives us more computing power / allows us to solve more.

What is the alternative definition of NP?

→ **Thm:** A language is in NP if & only if it is decided by some polynomial-time nondeterministic Turing Machine.

$$NP = \{A \mid \exists \text{ a polynomial-time NTM that decides } A\}$$

- N.P. = Nondeterministic Polynomial-time

→ Basically, any polynomial-time verifier can be converted to

RECALL: How do nondeterministic TMs operate?

→ A nondeterministic TM operates like a regular TM, except that at any point in its computation, the TM can branch off into several possibilities of what to do next.

- Normal TM transition function: $\delta: Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$, versus NTM transition function: $\delta: Q \times \Gamma \to P(Q \times \Gamma \times \{L, R\})$
- An NTM is a decider iff all of its computation branches halt on all inputs.
- A decider NTM accepts if/as soon as at least 1 branch accepts on a given input.

How is the running time for an NTM calculated?

→ **DEFN:** The running time of an NTM N is the function $f: \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of of length n.

- Basically, the running time = the length of the longest branch.

What is an example of a poly-time NTM?

→ We can use an NTM N_1 that decides HAMPATH & run a time analysis to show that it runs in polynomial time, thus proving that HAMPATH ∈ NP :

→ N_1 = "On input $\langle G, s, t \rangle$, where G is a directed graph with nodes s and t :

1. Write an array P of length m (where m is the # of vertices in the graph) where each $P[0, 1 \ldots m-1]$ is one of the vertices in the graph.
   - When appending elements to the array, nondeterministically select one of $(1, 2, 3, \ldots m)$ to be the next num. in the list
   - So basically, we are nondeterministically going to create m^m array lists, with each branch of the NTM following one list.
2. Check whether the list represents a ham. path from s to t — specifically,
   - check if there are any repetitions in the list
   - check whether $s = P[0]$ and $t = P[m-1]$
   - check whether an edge exists between every node in the list & the nodes to its right & left.
3. If it's a ham. path, accept. Else, reject.

How do we know that N_1 runs in poly-time?

→ The key to checking if a given graph contains a Hamiltonian path is testing every possible path — and there are a **lot**, which is why HAMPATH cannot be decided in polynomial-time by merely a STDTM.

→ With an NTM, we can have each branch test one path. The longest it can take for a branch to compute is

1. Scan list for repetitions & reject if any found: m steps
2. check if the list starts with s & ends with t : m steps
3. check for directed edges: $\sim m$ steps

$m + m + m = \boxed{3m}$ steps

→ Since the running time of an NTM is given by its longest branch, we know that N_1 runs in $O(3m)$, aka $O(n)$ time!

How are these 2 definitions of NP equivalent?

→ The proof that $\{A \mid \exists$ a poly-time verifier for $A\}$ and $\{A \mid A$ is decidable by a poly-time NTM$\}$ are actually the same set (aka NP!) is given by showing that any verifier can be converted to an equivalent poly-time NTM, and vice versa!

- have to prove both directions of this statement

RECAP of what these machines are meant to be doing?

→ RECALL: The goal of a TM that **decides** a language X is to, given an input w, conclude whether or not $w \in X$.

→ RECALL: the goal of a verifier V for a language X is to, given an input of X, w, and a certificate c, to use c to determine whether $w \in X$.

How do you convert a verifier to a (poly-time) NTM?	→ Let $A \in NP$, which indicates by definition that there is a TM V which is the verifier for A : V = "On input $\langle w, c \rangle$: use c to decide if $w \in A$" (generalized) • Assume that V runs in time $O(n^k)$, where n is the length of the input string w. → We can create an NTM N_1 that decides A in polynomial time by basically using the nondeterminism to create every possible certificate string c (aka every possible string of the alphabet (Σ) of c, and then testin each of them through V. • Yes, this seems kind of crazy & unrealistic - how could it be done in poly-time? • ANS: Even though there might be a crazy large amount of computation branches, it doesn't affect the running time of an NTM because its decided solely by the time of the **longest** computation branch. → N = " On input w of length n : 1. Nondeterministically select a string c_1 of length at most n^k. (One nondeterministic branch per possible certificate) 2. Run V on input $\langle w, c_1 \rangle$ 3. Return the output of V (accept if V accepts, reject if V rejects) → This is the general idea behind the NTM decider of any problem in NP : To nondeterministically try every possibility.
How do you convert a poly-time NTM to a verifier?	→ Since the NTM operates by "crafting" every possibility by nondeterministically adding symbols to a string at each step, we can create a verifier by simulating the specific branch of N that corresponds to the symbols in c : → V = " On input $\langle w, c \rangle$, where w and c are strings: 1. Simulate N on input w, treating each symbol of c as a description of the nondeterministic choice to make at each step. 2. If this branch of N's computation accepts, accept. Else, reject."
What is the nondeterministic time complexity class? NTIME(t(n)) =	→ For any function $t(n)$, the n.d. time complexity class NTIME $(t(n))$ is the collection/set of all languages that are decidable by an $O(t(n))$-time **nondeterministic** Turing Machine → $NP = \bigcup_k \text{NTIME}(n^k)$ (just like $\underline{P} = \bigcup_k \text{TIME}(n^k)$), aka $NP = \{ A \mid A$ is decidable by an $O(n^k)$-time NTM $\}$ → By definition $\bigcup_k \text{NTIME}(n^k)$ is a subset of $\underline{\bigcup_k \text{TIME}(2^{n^k})}$ exponential time brute force method

Summary: key points of Ch.7 so far?

→ P = { A | A is decidable in polynomial time by a DTM }

= $\bigcup_k \text{TIME}(n^k)$ (TIME $(n^0, n^1 \ldots)$)

= "easy-to-solve problems"

→ NP = { A | A is decidable in polynomial time by an NTM }

as well as { A | $\exists$ a poly-time verifier for A }

= $\bigcup_k \text{NTIME}(n^k)$

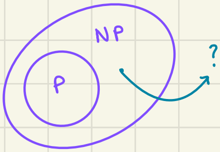
= "easy-to-verify problems"

→ Verifiers - V(w, c): "use c to determine whether w is in A"

→ $P \subseteq NP$; all problems decidable by a poly-time DTM are also decidable by a poly-time NTM

→ No one knows if NP is a "strict superset" of P; e.g., if there are any languages in NP but not P

- aka, problems which are "hard to solve, easy to verify"; think of Sudoku: its very difficult to come up with a Sudoku puzzle, but relatively quite easy to verify, e.g. check a filled-in puzzle to see if it is a correct solution.



Part 3: Complexity Theory

Ch 7: Time Complexity

7.4: NP-Completeness

What is the SAT problem?

→ SAT = { ⟨φ⟩ | φ is a satisfiable boolean function }, where φ is a string comprised of some sequence of:

a) variables $x_1, x_2, \ldots, x_n$ and

b) their negations $\overline{x_1}, \overline{x_2}, \ldots \overline{x_n}$, which are strung together by

c) "or" operators ($\vee$s), "and" operators ($\wedge$s), and parantheses' (( )).

→ For ex, $\varphi_1 = (\overline{x_1} \wedge x_2) \vee (x_1 \wedge \overline{x_3} \wedge x_2)$ could be a sample input to SAT.

What strings does SAT accept?

→ A string φ, which is a sequence of the symbols defined above, is satisfiable (to SAT) i.f.f. there exists some way to assign a value of either 0 (→ False) or 1 (→ True) to each variable $x_1 \ldots x_n$ s.t. the entire expression evaluates to TRUE.

- For ex, φ_1 is satisfiable if we set $x_1 = 0$, $x_2 = 1$, and $x_3 = 0$ or 1.
- Meanwhile, strings like $\varphi_2 = x_1 \wedge \overline{x_1}$ and $\varphi_3 = (x_1 \vee x_2) \wedge (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2})$ are **not satisfiable**.

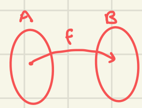
Is SAT in the class NP?

→ Yes! , we can create an NTM that tries all 2^n possible assignments & runs in poly-time

RECALL: What is mapping reducibility?

→ For any 2 languages A, B , $A \leq_m B$ ("A is mapping-reducible to B") if there exists a function f that maps strings to strings ($f: \Sigma^* \to \Sigma^*$) and which is **computable by a DTM** such that:

for every string x, $x \in A$ i.f.f. $f(x) \in B$.



- THEOREM: if A is undecidable and $A \leq_m B$, then B is undecidable

→ Basically, mapping reducibility is used to show that if problem A reduces to problem B, then A is at most as "difficult" of a problem as B, because a solution to B can be used to solve A.

What is polynomial-time (mapping) reducibility?

→ Basically the same idea as mapping-reducibility, except focused on time efficiency, i.e. testing if a language A is **efficiently reducible** to B.

- Instead of just finding a function that maps A-inputs to B-inputs, we specifically want to find a **polynomial time computable function**.

→ **Formal DEFN:**

Language A is <u>polynomial time reducible</u> to language B, written $A \leq_p B$, if there exists some **polynomial time** computable function $f: \Sigma^* \to \Sigma^*$ s.t. for every string w,

$$w \in A \iff f(w) \in B$$

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

But wait – What is a polynomial-time computable function?

→ **DEFN**:

A function $f: \Sigma^* \rightarrow \Sigma^*$ is a polynomial-time computable function if some **polynomial-time T.M.** M exists such that for any input w, when M is started on w, M ends/halts with just $f(w)$ on its tape.

- Same idea as "computable function" defined in 'Mapping Reducibility' notes; that there is a TM which can 'compute' f by outputting $f(w)$ for any input w.
- The only difference is that now, we also take time complexity into account – "f" is only a valid "poly-time computable function" if M runs in polynomial time.

What is the point of this poly-time reducibility concept?

→ To allow us to talk about the **relative hardness** of problems, since we often times don't definitively know whether or not a problem is "hard" (like HAMPATH, SAT, etc)

- With poly-time reducibility, though, we can still make claims like "A is **at most as hard as** B"

→ If A is poly-time reducible to B, it implies that it is not possible that A is "hard" (aka not decidable by a poly-time DTM) and B is "easy"

What do 'easy' and 'hard' mean in this context?

→ IMPORTANT NOTE:

"**easy**" = decidable by a poly-time D.T.M., aka in P

"**hard**" = not* decidable by a poly-time deterministic TM, aka in NP but not in P.

*technically should say "unlikely" instead of "not" because we don't truly know, but yk

Relationship between Ch.5 & Ch.7?

→ Ch.5 was about proving whether problems are decidable or undecidable. Ch 7.4 is about proving whether problems are 'easy' or 'hard'.

How does poly-time reducibility relate to the class P?

→ **Thm:** If $A \leq_p B$ and $B \in P$, then A is also an element of P!

→ **Proof idea:** We are trying to prove that if some T.M. M can decide B in poly-time, then there also exists some T.M. N which decides A in poly-time.

→ The reasoning behind this Theorem is pretty simple. If $A \leq_p B$, then we already know that there exists:

a. A poly-time TM M which decides B, and

b. A poly-time computable function (aka 'reduction') f which maps A to B. aka, if $w \in A$, then $f(w) \in B$

→ We can construct a poly-time TM for A as follows:

N = "On input w:

1. Compute $f(w)$
2. Run M on input $f(w)$ and accept if M accepts. If M rejects, reject.

→ N obviously runs in polynomial time b/c both stages run in poly-time.

What does it mean for a language to be NP-hard?

→ **DEFN:** A language/problem B is **NP-hard** if, for all languages $A \in NP$, A is poly-time reducible to B ($A \leq_p B$).

- a.k.a., a problem B is NP-hard if all languages in NP can be reduced to B by a poly-time function.
- These problems are "unlikely to be in P" and generally very difficult - sometimes not even solvable (decidable)! For ex, A_{TM} is NP-hard.

What does it mean for a language to be NP-complete?

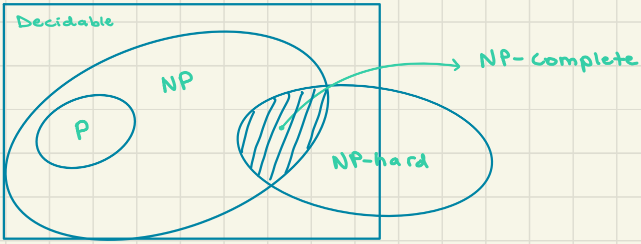
→ **DEFN:** A language B is **NP-complete** if

1. $B \in NP$, and
2. B is NP-hard (every lang in NP reduces to B)

- These are problems which are "easy to verify" but "likely difficult to solve"

→ Basically, if any NP-complete problem can be solved, then every problem in NP can be solved.

How do P, NP, and NP-hardness/completeness relate to one another?



What claim can we make about NP-hardness?

→ **KEY THEOREM:** If $A \leq_p B$ and A is NP-hard, then B is NP-hard.

→ **Proof:** To show that B is NP-hard, we must show that every problem in NP reduces to it

- Let C represent all languages in NP ($C \in NP$). We want to show that $C \leq_p B$

→ We know that A is NP-hard, which implies that there exists a poly-time function f s.t. for every input w, if $w \in C$ then $f(w) \in A$.

→ We also know that $A \leq_p B$, implying that there exists a poly-time function g s.t. for every input x, if $x \in A$ then $g(x) \in B$

→ Then, we can define a poly-time reduction h from C to B which does the following: h = Given input z:

1. let $y = f(z)$ (run f on the input to h)
2. Return $g(y)$ aka $g(f(z))$

How do we know that this is a proper reduction?	→ We know that h is a proper reduction b/c, for any input $\underline{x}$ • if $x \in C$, we know that $f(x) \in A$ since $C \leq_p A$ • if $f(x) \in A$, we know that $g(f(x)) \in B$ since $A \leq_p B$ • Therefore, if $x \in C$, then $\underline{h} = g(f(x))$ must be in B
How do we know that h runs in poly-time?	→ Because both f and g are known to run in poly-time & the composition of 2 polynomials is always a polynomial.
What theorem follows from this one?	→ Thm 7.36: if A is NP-complete and $A \leq_p B$ for language $B \in NP$, then B is NP-complete. •basically same logic as previous theorem. • if A is NP-complete, that means that all problems in NP can be reduced to it. So if we can show that A, in turn, can be reduced to B, it implies that every NP lang can be reduced to B as well. → Thm 7.35: If B is NP-complete and $B \in P$, then P must be equal to NP ($P = NP$)

Poly-time Mapping Reductions V.S. Turing Reductions

What is the "intuition" behind problems in NP-hard and complete?

→ NP-hard: problems which are at least as hard as NP, or harder

- e.g. A_{TM}, which is not $\in$ NP & even harder than NP

→ NP-complete: all of the hardest problems in NP

- All NP-complete problems are also NP-hard

→ NP: problems that are "exactly as hard as NP"

What does poly-time reducibility mean?

→ Saying "$A \leq_p B$" means that, given a poly-time DTM for B, we can design a poly-time DTM for A.

- The DTM for A works by, at some point, calling the DTM for B

What is the invution/motivation to use reducibility?

→ The goal of this chapter is to prove that problems are hard.

→ In order to prove that a problem B is hard, we reduce a problem that we already know (or strongly believe) is hard to it.

→ The goal is to say "B is hard because if B were easy, then A would be easy (aka, if A can be shown to reduce to B). But if/since we believe A is hard, we should also believe that B is hard."

→ The same idea we practiced in ch. 5! E.g., we showed that a decider TM for E_{TM} can be used to create a decider TM for A_{TM}. But since we already know that A_{TM} is unsolvable & no decider exists for it, the proof is a contradiction and we can then conclude that E_{TM} must also be undecidable.

What is a Turing Reduction?

→ Proving that a problem A is reducible to B using a 'Turing reduction' simply means creating a (poly-time) TM ALG_A for A which uses the poly-time TM for B, ALG_B, in its computation.

- We can use ALG_B however we want; call it multiple times, negate its input, etc... no restrictions.
- This means we can get pretty creative/clever with our reduction proofs.

→ Turing reductions feel like a much more intuitive way of proving hardness; if A is (known) hard and a poly-time TM for B can be used to create a poly-time TM for A... then obviously B can't actually be solved in poly-time (and is therefore hard).

How does a poly-time mapping reduction differ from a Turing reduction?	→ To prove a pt mapping reduction, you start w/ the same goal: Trying to design a poly-time TM for A, given one for B. → However, the rules are much stricter. To prove that $A \leq_p B$, the TM that we design for A given ALG_B must follow a very specific format. Namely, the TM ALG_A must look like this: ALG_A = "On input x: 1. Compute $y = f(x)$ 2. Run ALG_B on y and return its output." Where "$f(x)$" is our poly-time computable function! (see notes) → Unlike a Turing reduction, we can't do any other random stuff (like calling ALG_B twice or etc.). We are restricted to this template. → Naturally, a poly-time mapping reduction implies the existence of a Turing reduction... but not vice versa.
So which type of reduction should we use?	→ Using a Turing reduction to prove hardness is NOT correct... we must follow the strict format of poly-time mapping reductions when it comes to proving that problems are hard - for ex, $B \in$ NP-hard i.f.f. $A \in$ NP-hard and $A \leq_p B$.
Why do we have to use poly-time mapping reductions?	• Several reasons: → P.T.M. reductions make a **stronger statement** about the hardness of problems (so its just better practice)... they enable a more fine-grain of complexity classes. → In practice, a lot of reductions made with the "Turing reduction" mindset end up looking like/being poly-time mapping reductions anyway. → KEY REASON: In complexity theory, we believe that the class NP $\neq$ coNP, and mapping reductions distinguish between NP and coNP, while Turing reductions do not (Tangent).
What is coNP?	→ A complexity class which contains the **complements** of all languages in NP; i.e., languages where you can "easily" (in poly-time) verify that a given input is **not** a member of a certain language. • also equivalent to taking the poly-time NTM for a language in NP and swapping the accept & reject states.

What is the mindset for creating/proving a pt mapping reduction?

→ To show $A \leq_p B$, The goal is to find the computable function f, which (obviously) has to be computable in poly-time, <u>such that</u> the following template gives a poly-time decider for A:

ALG_A = "On input x :

1. Let $y = f(x)$
2. Return $ALG_B(y)$."

→ Focused on creating the computable function f, more so than the decider TM ALG_A.

Ari Kumar

Due April 17, 2024

COMP 455-001

Homework 4

Page 1

1.

a) if $A \in P$, then $\bar{A} \in P$ — $\boxed{\text{True}}$

- if $A \in P$, meaning that a TM M_1 decides A in polynomial time, we can show that $\bar{A}$ is also an element of P by constructing a poly-time TM N_1 which uses M_1:

  N_1 = "on input w:

  1. Run M_1 on w. If M_1 rejects, accept.
     Otherwise if M_1 accepts, reject."

- N_1 clearly runs in polynomial time because it only has 1 stage, which runs/imitates M_1. And since M_1 runs in polynomial time, N_1 must do so as well.

b) if $A \in P$ and $B \in P$, then $A \cup B \in P$. — $\boxed{\text{True}}$

- We can prove that $A \cup B \in P$ for 2 languages $A, B \in P$ by constructing a polynomial-time TM N_1. Let M_1 be the TM which decides A. Let M_2 be the TM which decides B.

  N_1 = "on input w:

  1. Run M_1 on w. If it accepts, accept.
  2. Run M_2 on w. If it accepts, accept. Otherwise, reject."

- N_1 clearly runs in polynomial time because it runs for a polynomial number of stages, and each stage can be done in polynomial time (we know that both M_1 & M_2 run in poly-time because A and B are elements of P).

c) if $A \in P$ and $B \in P$, then $A \circ B \in P$ — $\boxed{\text{True}}$

- We can prove that $A \circ B \in P$ for 2 languages $A, B \in P$ by constructing a polynomial-time TM N_1. Let M_1 be the TM which decides A. Let M_2 be the TM which decides B.

N_1 = "on input w:

1. Repeat the following for every possible way to split w into 2 strings $w_1 w_2$, where $0 \le |w_1| \le |w|$ and $0 \le |w_2| \le |w|$:

"stage 2" { 2. Run M_1 on w_1. If it rejects, move on to the next possible division of w into $w_1 w_2$.
3. (Else, if M_1 accepts w_1) run M_2 on w_2. If it accepts, accept. If it rejects, move on to the next possible division of w into $w_1 w_2$. }

4. If w is not accepted after trying every possible split, reject."

- We know that N_1 decides the concatenation of A and B because it accepts a string w i.f.f. w can be written as $w_1 w_2$ such that $w_1 \in A$ and $w_2 \in B$.
- Stage 2 runs in polynomial time since it utilizes M_1 and M_2. Additionally, stage 2 will be repeated at most $n = |w|$ times (because for a string w, there are $|w|$ ways to split it into 2 substrings). Since stage 2 runs in polynomial-time and is repeated at most a polynomial number of times, we know that N_1 runs in polynomial time, and therefore $A \circ B \in P$.

Ari Kumar
Due April 17, 2024

2. Show that P=NP implies that every language B in P, except $\emptyset$ and Σ^*, is NP-Complete.

- If B is a language in P which is not $\emptyset$ or Σ^*, then we know that there are strings which are in B as well as strings which are not. Let b_1 be a string in B ($b_1 \in B$) and let b_2 be a string not in B ($b_2 \notin B$).
- To prove that a language is NP-complete, we most show 2 things:
  1) That the language is in NP.
  2) That the language is NP-hard.
- If P = NP and every language $B \in P$, then naturally all B are also in NP (which is true anyways since we know for a fact that P is at least a subset of NP). Therefore the first part is proven
- To prove that all $B \in P$ are also NP-hard, we must show that all languages C in NP can be poly-time mapping reduced to B (e.g. $C \leq_p B$ for all languages $C \in NP$ and all languages $B \in P$) A reduction to prove this follows.
- Assuming P=NP, then all languages $C \in NP$ are also in P, and therefore, there exists a polynomial-time TM ALG_c which decides every C.
- This is a computable function f which maps C to B:

  f(x):

  1. Run ALG_c on x.
  1. If ALG_c accepts, then output b_1. If it rejects, output b_2.
- This function f clearly reduces C to B in polynomial time because the stages both run in polynomial-time (due to the fact that $C \in P = NP$). Therefore, B is NP-complete.

Ari Kumar

Due April 17, 2024

3. Prove that the given problem is NP-hard by reducing it to the given language.

Let the given problem be denoted as the language B. Let the problem $C = \{\langle G \rangle \mid G$ is a 3-colorable undirected graph$\}$, where the input G consists of a set of vertices/nodes V, and a set of edges E. Each element of E is a pair of vertices in G.

To prove that B is NP-hard, we must show that $C \leq_p B$. We must show that inputs w of C can be mapped to inputs w_1 of B such that $w_1 \in B$ if & only if $w \in C$, by a polynomial-time computable function f. The reduction follows.

f(x):

1. if x is not of the form $\langle G \rangle$: return 0
2. Let T = 3.
3. Let K = The number of vertices in G.
4. Assign each node in G a number 1, 2, ... K. Let the shorthand num(v) denote the number that has been assigned to a vertice v.
5. For each edge (u, v) in G, add a new "student exam list" [num(u), num(v)] to an Array of lists A.
6. Return $\langle A, K, T \rangle$

- f works by taking an undirected graph G and letting each node represent an exam, while each edge represents a student. The 2 nodes that the edge is attached to represent the 2 exams that that student has to take. In a given coloring of a graph, each of the 3 colors would represent the 3 time slots (thus assigning a time slot to each exam (aka node)).

If G were 3-colorable, then every student would be able to take their 2 exams at 2 different times, meaning that a string encoding $\langle A, k, t \rangle$ would be satisfiable & thus an element of B. If G were not 3-colorable, then at least 1 student has 2 exams occurring at the same time slot & thus $\langle A, k, t \rangle$ would not have a "solution" & wouldn't be an element of B.

- Additionally, we know that f is a polynomial-time computable function because it has a polynomial number of stages, each with a polynomial # of steps. The stage with the most steps is stage 5 - it has a maximum of $(n(n-1))/2$ steps, where n = # of nodes. This is clearly a polynomial number of steps.

Ari Kumar
Due April 17, 2024

3.

A DTM ALG_C for C using a TM ALG_B for B could then be constructed as follows:

ALG_C = "On input w:

1. Compute $y = f(w)$
2. Run ALG_B on y and output its output."

Therefore, we have proven that $C \leq_p B$ and thus, B is NP-hard.

Part 3: Complexity Theory

Ch 7: Time Complexity

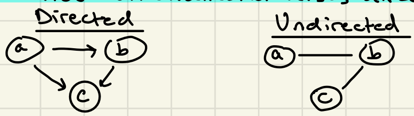
7.5: Additional NP-Complete Problems

What is an example of a polytime mapping reduction?

→ **Thm 7.55:** UHAMPATH is NP-complete.

UHAMPATH = { <G,s,t> | G is an undirected graph with a Hamiltonian path from node s to node t. }

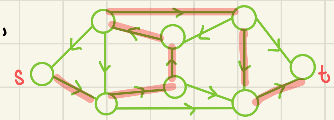
→ RECALL: an undirected versus directed graph:



RECALL: What is HAMPATH?

→ HAMPATH = { <G,s,t> | G is a directed graph with a Hamiltonian path from node s to node t. }

→ For example,



→ **Thm:** we treat HAMPATH ∈ NP-complete as a fact (don't worry about the proof). Meaning

- HAMPATH ∈ NP
- $X \leq_p$ HAMPATH for all problems $X \in$ NP (Defn. of NP-hardness).

How can we prove Theorem 7.55?

→ We can prove that UHAMPATH is NP-complete by proving that

1) UHAMPATH ∈ NP, and
2) that HAMPATH $\leq_p$ UHAMPATH (Thm 7.36)

How do we know that UHAMPATH ∈ NP?

→ This part is relatively easy to prove; we know that UHAMPATH ∈ NP by making a decider NTM that does basically the same thing as the NTM for HAMPATH (see notes pg. 105); testing one graph path per computation branch (of the NTM)

How do we show that HAMPATH $\leq_p$ UHAMPATH?

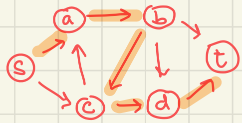
→ We need to design a poly-time computable function f that takes an input in the form of HAMPATH – i.e. an encoding of a graph (G,s,t) – and returns an output in the form of UHAMPATH – an encoding of a graph (G',s',t') such that

G has a ham. path from s → t if & only if
G' has a ham. path from s' → t'.

How do we create our computable function?

→ We have to figure out how to convert a <u>directed graph</u> with a Hamiltonian path to an <u>undirected graph</u> with one.

<u>D.G. G w/ s-t Ham. path</u>



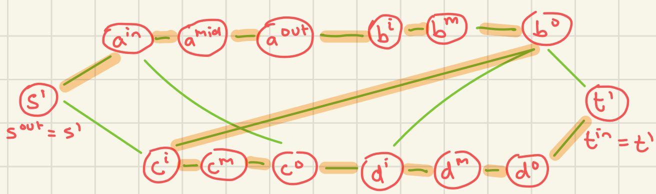
→ $f(x)$:

- If x is not of the form $\langle G, s, t \rangle$: return 0

  (automatic reject; we can usually omit this step)

- Construct an undirected graph G' where, for each node u in G <u>except for</u> s and t: replace u with 3 nodes u^{in}, u^{mid}, and u^{out}.
- replace s with $s' = s^{out}$ and t with $t' = t^{in}$.
- For all nodes u, draw edges connecting u^{in} to u^{mid}
- For all nodes u, draw edges connecting u^{mid} to u^{out}
- for all edges $\{u, v\}$ in G (aka edges from u to v), draw an edge connecting u^{out} with v^{in}.
- return the undirected graph (G', s^{out}, t^{in}).

<u>equivalent U.D.G G'</u>



→ Therefore, we have proven that HAMPATH $\leq_p$ UHAMPATH because we can use f to create a poly-time decider N_1 for HAMPATH:

N_1 = "On input $\langle G, s, t \rangle$:

1. Run $f(x)$ on $y = \langle G, s, t \rangle$.
2. Compute $ALG_{UHAMPATH}$ on y & return its output."

How does this reduction prove that UHAMPATH is NP-complete & a member of NP?

→ If UHAMPATH was $\notin$ NP and <u>was</u> $\in$ P (meaning it can be deterministically decided in polynomial time), then (by proving HAMPATH $\leq_p$ UHAMPATH) we have we have shown that HAMPATH would also be decidable by a poly-time DTM (N_1). But since we know that HAMPATH $\notin$ P, therefore neither is UHAMPATH.

- And since HAMPATH is NP-hard & we reduced it to UHAMPATH, UHAMPATH must also be NP-hard.

Recap: How do we prove that languages are NP-complete?

→ By reducing them to <u>known</u> NP-complete problems (e.g. HAMPATH $\leq_p$ UHAMPATH)

→ But... there had to be an NP-complete problem that we originally began with, and which was proven w/o a reduction, right?

- SIMILAR to how A_{TM} was proven undecidable by diagonalization, and then A_{TM} could be used to prove many other languages undecidable via reduction.

So what was the first NP-complete problem?

→ SAT! RECALL that SAT $= \{ \langle \varphi \rangle \mid \varphi$ is a satisfiable Boolean formula $\}$.

- Σ of strings in SAT: $\{ \wedge, \vee, x_1, \ldots, x_i, \bar{x}_1, \ldots, \bar{x}_i \}$

$\boxed{\text{EX}}$ $\varphi = x_1 \wedge \bar{x}_1 \notin$ SAT

$\varphi = (x_1 \wedge x_2) \vee (x_1 \wedge \bar{x}_2) \in$ SAT

What is the Cook-Levin theorem?

→ Thm: SAT is NP-Complete.

→ One of the reasons that SAT is the o.g. NP-complete problem is bc it basically represents/captures how computer algorithms work (logic statements!)

→ Proof: Must show that SAT $\in$ NP, and that all languages in NP reduce to SAT (aka SAT $\in$ NP-Hard)

How do we know that SAT $\in$ NP?

→ Simple: create an NTM that, for a given input, creates a branch for each possible way to assign each var. $x_1 \ldots x_i$ to 1 (TRUE) or 0 (FALSE) (2^n possibilities/branches)

How do we know that SAT is NP-hard?

→ For a lang $A \in$ NP with a decider NTM N...

see textbook for more details; not super important to know for this class.

What are some definitions pertaining to boolean formulas?

- <u>literal</u>: a variable (or its negation), e.g. $x_1, x_2, \bar{x}_2, \bar{x}_1, x_3, \bar{x}_4$ are all literals.
- <u>clause</u>: an expression consisting of several literals connected with OR ($\vee$) symbols, e.g. $(x_1 \vee \bar{x}_2 \vee \bar{x}_3 \vee x_4)$ is a clause.
- <u>cnf-formula</u>: A boolean formula comprised of several clauses connected with AND ($\wedge$) symbols, e.g.

$(x_1 \vee \bar{x}_2 \vee \bar{x}_3 \vee x_4) \wedge (x_3 \vee \bar{x}_5 \vee x_6) \wedge \ldots$

What is 3SAT?

→ A special case of SAT where the boolean formula is comprised of a cnf-formula where each clause has exactly 3 literals. For ex:

$(x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_3 \vee x_4 \vee \bar{x}_5)$

How do we show that 3SAT is NP-complete?

→ By showing that SAT $\leq_p$ 3SAT!

→ We need a computable function that takes an input to SAT and converts it to an input to 3SAT (s.t. $f(x) \in$ 3SAT i.f.f. $x \in$ SAT)

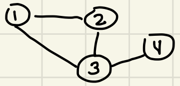
→ Sketch example from class:

$$\varphi = x_1 \vee \bar{x}_2 \vee \bar{x}_3 \vee x_4$$

$$f(\varphi) = (x_1 \vee \bar{x}_2 \vee z) \wedge (\bar{z} \vee \bar{x}_3 \vee x_4)$$

What is the language VERTEX-COVER?

→ For an undirected graph, a "vertex cover" of that graph is a subset of nodes/vertices where every edge in G touches one of those nodes. For example, if G =



, then S = {1, 3, 4} is a vertex cover b/c every edge touches either v=1, v=3, or v=4 (or both)

but S = {1, 2} is not.

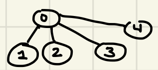
→ VC = { <G, K> | G is an undirected graph that has a (at most) K-node vertex cover (i.e., ∃ a v.c. subset S that has K elements). }

What is the language Independent-Set?

→ For an undirected graph G, an "independent set" is a subset S of nodes s.t. none of the edges in G connect 2 of the nodes in the subset.

∴ i.e., none of the nodes in S are directly connected to each other.

→ For ex, if G =



, then S = {1, 2, 3, 4} is an ind. set b/c for every edge of G {u, v}, ($u \notin S \vee v \notin S$) is true.

i.e., there are no edges connecting v=1,2,3,or 4 to one another.

→ IC = { <G, K> | G is an undirected graph that has an Independent Cover subset with (at least) K elements. }

How do we show that IS is NP-hard?

→ Let's assume (for now) that we know & have proven that VC ∈ NP-HARD. We can show that IS ∈ NP-HARD by proving that VC $\leq_p$ IS!

→ A computable function f that reduces inputs to VC into inputs to IS:

f (G, K):

1. Given an input of an undir. graph G and an int K, return < G, n − K > where n = # of nodes in G.

How does this function work?

→ If a <G, K> ∈ V.C., that means that for every edge E = (u, v), either u or v is an element of the "vertex cover subset" S. And that |S| ≤ K.

→ To produce the set S' of elements which are an independent set of G, we simply choose all the nodes not included in S. Therefore, the length of S' would be (total nodes in G) − (# of nodes in S). And since |S| = K, |S'| = n − K.

But wait .. is VERTEX - COVER even NP-complete?

→ Yep - we can show this by proving that 3SAT $\leq_p$ VERTEX-COVER

→ To map boolean formulas in the form $(a \vee b \vee c) \wedge (c \vee d \vee e) \wedge (c \vee a \vee z) \wedge \ldots$ into undirected graphs with (or without) vertex covers, we must figure out a way to convert the variables & clauses of the formula.

→ See textbook/lecture for the rest idk

Part 3: Complexity Theory

Ch 8: Space Complexity

8.1: Savitch's Theorem

→ So far, we've defined hardness by:

1. **Decidability**: Whether a problem can be solved in the first place
   - problems like A_{TM} and $HALT_{TM}$ can't be solved by a computer
2. **Time complexity**: How long it takes to solve a problem.
   - problems like "P v.s. NP" (aka, does using nondeterminism buy us anything in polynomial time? Does it help us solve more problems? Is P = NP?) are simply unknown! Nobody knows the answer.

Now, we add space complexity to the list.

What is space complexity?

→ Considering/classifying computational problems in terms of the amount of space (aka memory) that they require.

→ Time & Space are 2 of the most important considerations when we seek practical solutions to many computational problems.

→ V. similar (in terms of how we understand & evaluate it) to Time complexity (ch.7)

- We will again use deterministic TMs as our standard model for measuring & comparing the space complexity of problems.

What is the formal defn. of space complexity?

→ **DEFN:**

Let M be a deterministic TM that halts on all inputs (i.e. a decider).
The **space complexity** of M is the function $f: \mathbb{N} \rightarrow \mathbb{N}$, where $f(n)$ is the maximum number of tape cells that M scans on any input of length n.

- If the space complexity of M is $f(n)$, we say that "M runs in space $f(n)$."

What if M were a nondeterministic TM?

→ Then we define its space complexity $f(n)$ to be the # of tape cells used by the longest/largest branch - same idea as with time complexity.

- The maximum space used by any 1 of all the branches.

RECALL: What is big-O?

→ aka **asymptotic notation**; estimating complexity by disregarding smaller coefficients in a given $f(n)$.

- For Ex, if $f(n)$ for TM A $= n^2 + n + 1$, then $O(f(n)) = n^2$.

What is SPACE(f(n))?

→ A **space complexity class**:

$$SPACE(f(n)) = \{ L \mid L \text{ is a language decided by an } O(f(n))\text{-space DTM} \}.$$

- i.e., the space-complexity equivalent of TIME(f(n)) (RECALL!)

What is NSPACE(f(n))?

→ A space complexity class:

$NSPACE(f(n)) = \{ L \mid L$ is a language decided by an $O(f(n))$ space nondeterministic TM. $\}$

- space-comp. equivalent of $NTIME(f(n))$

What is the relationship between time and space complexity?

→ space is more powerful than time – because space can be reused, while time cannot.

→ If a problem is decidable by a $t(n)$-time STDTM N_1, then N_1 also runs in at most $t(n)$-space! It cannot take more space than it does time.

Fact 1: $TIME(t(n)) \subseteq SPACE(t(n))$

Why is space stronger than time?

→ RECALL that if the TM N_1 for a language A runs in $t(n)$-time, this means that it takes at most $O(t(n))$ steps to compute an output.

→ Within $O(t(n))$ steps, N_1 can't possibly use / look at more than $O(t(n))$ cells! Since each step of a TM involves either:

- moving the tape head left by one cell,
- moving the tape head right by one cell, or
- doing smthn else & not moving the tape head at all.

→ Therefore, if $A \in TIME(t(n))$, then $A \in SPACE(t(n))$

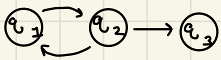
What else is true about the relationship between space & time complexity?

→ If a TM runs in linear (n) space, then the maximum amount of time it can take is 2^n steps. AKA

Fact 2: $SPACE(t(n)) \subseteq TIME(2^{O(t(n))})$

→ **Proof:** Recall that If a TM is a decider, it never loops; it goes through a sequence of states & steps but it never repeats states/sequences (bc then it wouldn't be a decider since its stuck in a loop).

- Unlike DFAs, for ex, which can contain "loops", e.g.



→ For the proof, lets consider a decider TM M which has linear ($f(n)$) space.

→ Since the # of tape cells for M on input length n is bounded, this means that the number of possible **configurations** — i.e. possible combos of [current state, current tape head location, current contents of tape] that M can enter is also bounded: Specifically, for an input of length n, M has (roughly) at most 2^n configurations.

→ And we know that, since M is a decider, none of these "configurations" get repeated (aka no looping). Therefore, M will take at most $2^{O(t(n))}$ steps to compute. AKA, M runs in $2^{O(t(n))}$ time.

What is an example of a space-complexity analysis?	→ Lets show that SAT ∈ SPACE(n); aka, SAT runs in linear space!

- RECALL that SAT ∈ NP and NP-complete. It does not (as far as we know) run in linear time.

not even polynomial! n^2

→ **Proof:** Let M_1 be a TM for SAT.

M_1 = "On input φ, where φ is a boolean formula:

1. For each truth assigment to the variables $x_1, \ldots, x_m$ of φ:
2. Evaluate the status of φ (the formula) on that assignment.
3. If φ ever evalvates to 1 (aka TRUE), accept. If not, reject.

→ M_1 clearly runs in linear space because each iteration of the loop (in steps 1-2) can reuse the same portion of the tape.

→ All that the TM needs to be able to store at one moment is the current assignments (to each variable)

- Since there are m variables (be "$x_1, \ldots x_m$"), M_1 requires at most m tape squares at a given point in its computation, aka $O(m)$ space.

→ And finally, for a given input to φ (e.g. a boolean formula) of length n, the # of variables m which require assignments can't possibly be more than n (it would honestly prob have to be less, since boolean formulas also need operator symbols in them)

- Therefore, M_1 runs in maximum n tape squares, aka $O(n)$ space!

What is the class PSPACE?	→ The class of all problems which can be solved in polynomial space,

written $\text{PSPACE} = \bigcup_k \text{SPACE}(n^k)$, aka the union of the classes $\text{SPACE}(n^1) \cup \text{SPACE}(n^2) \cup \ldots \text{SPACE}(n^{18}) \ldots$

- RECALL: this is the space complexity equivalent of $\boxed{\text{the class } P}$; the class "P" refers to all languages solvable in polynomial time.

What is the class NPSPACE?	→ $\text{NPSPACE} = \bigcup_k \text{NSPACE}(n^k)$

→ The class of all problems which can be solved in polynomial time by a Nondeterministic TM.

- RECALL: Sort of the space complexity equivalent of $\boxed{\text{the class NP}}$

RECALL: What is the relationship between the classes P and NP?

→ KNOWN/proven fact: $P \subseteq NP$ (P is a subset of NP)

- any lang X in P has a DTM that runs in poly-time. Obviously, if a DTM can calculate X in poly-time, then we can make an NTM which does the same thing.

→ UNKNOWN: is $P \subsetneq NP$ (P is a proper/strict subset of NP)

- "proper subset" meaning that there are elements in NP which are not in P.

→ UNKNOWN: is $P = NP$?

- if $P \neq NP$, then $P \subsetneq NP$ (proper subset) This then implies that using nondeterminism does give us "more power" e.g. a larger scope of problems that can be solved in poly-time, than if we were to restrict ourselves to determinism.
- if $P = NP$, this means that P is not a "proper" subset of NP... all problems in NP are also in P and vice versa. This then implies that the power of DTMs and NTMs to solve problems in poly-time is equivalent

What is the relationship between PSPACE and NPSPACE?

→ Unlike with time complexity, we know for a fact that PSPACE = NPSPACE!

- Nondeterminism does not enable us to solve more problems in poly-space than simply using determinism.
- Any lang decidable by a poly-space NTM is also solvable by a poly-space DTM (and vice versa)

What is Savitch's Theorem?

→ THM: For all functions f where $f(n) \geq n$ (aka, the TM has at least n^1 space for an input of length n; basically saying, for any TM that has enough space to put each input symbol $x_1, x_2, \ldots x_n$ onto a tape square $t_1, t_2 \ldots t_n$)

$$NSPACE(f(n)) \subseteq SPACE(f^2(n))$$

What does this theorem mean?

→ Any NTM that solves a language A in $f(n)$-space can be converted into a DTM that solves A in $f^2(n)$-space

- For ex, if NTM B_1 solves A in $f(n) = n^{10}$ space, then there exists a DTM B_2 which solves A in $f^2(n^{10}) = n^{20}$ space
- Basically paying the 'cost' of a polynomial factor of 2.

→ since squaring a polynomial still results in a polynomial (n^{10} and $(n^{10})^2$ are both polynomials), this implies that PSPACE = NPSPACE !!

What is the proof that $NP \subseteq PSPACE$?

→ Thm: $NP \subseteq PSPACE$, aka, all problems which can be solved in polynomial-**time** by a **nondeterministic** TM, can also be solved in polynomial-**space** by a **deterministic** TM. (or, by proxy, a nondeterministic TM as well)

How does Savitch's Theorem already prove this?

→ Savitch's Theorem already proves that $NP \subseteq PSPACE$ because it tells us that PSPACE = NPSPACE, and

→ And "Fact 1" (pg 125) says that a problem cannot take up more space than it does time, which then implies that $NP \subseteq NPSPACE$.

Okay, but what if we want to prove it directly? (without Savitch's)

→ **Proof:** our goal is to show that all problems in NP are also in PSPACE. We can do this by showing how to construct a poly-space DTM for a given language in the class NP.

→ **RECALL:** SAT is our favorite NP-complete problem! If a language is NP-complete, this means that all problems in NP can be reduced to it.

→ If a given language $A \in NP$ (known), this implies that it can be poly-time reduced to SAT (e.g. "$A \leq_p SAT$"); this is a given.

- if $A \leq_p SAT$, this means that there is a computable function f which takes in A-format inputs, and outputs SAT-format strings s.t., string $w \in A$ iff. string $f(w) \in SAT$.

→ Given this, a polynomial-time **deterministic** TM for A computes as follows:

ALG_A = "on input <n>:

1. Run $y = f(x)$ on input $x = $ <n>.
2. Run SAT's TM, ALG_{SAT}, on input y. Accept if ALG_{SAT} accepts. else reject."

How do we know that ALG_A runs in poly-space?

→ Let's analyze it! We know that <u>stage 1</u> runs in poly-time because <u>f</u> is a poly-time computable func. & given "Fact 1", we know that all poly-time TMs are also poly-space TMs. Therefore stage 1 runs in poly-space.

→ We know that <u>stage 2</u> runs in poly-space because it calls ALG_{SAT} and, as we proved earlier (pg. 126), ALG_{SAT} runs in **linear** $O(n)$ time!

→ Therefore, A is an element of PSPACE!

What Theorem is proved by this proof?

→ **Thm:** if $A \leq_p B$ and $B \in PSPACE$, then $A \in PSPACE$!

- Ex: what we just did! $SAT \in PSPACE$ and $A \leq_p SAT$, and we just proved that $A \in PSPACE$

RECALL: How does the use of a multitape TM affect **time** complexity? Why?

→ Recall the theorem that states that

any $t(n)$-time Multitape DTM $\xrightarrow{\text{can be converted to}}$ an $O(t^2(n))$-time Single tape DTM

→ For every step that the MTDTM takes on its c # of tapes, the STDTM just takes (at most) $t(n)$ steps to replicate each single step of the MTDTM.

($t(n)$ steps to compute MTDTM) × ($t(n)$ steps per step that STDTM replicates) = at most $t^2(n)$ steps.

How does the use of a multitape TM affect space complexity?

→ **Thm:**

any $t(n)$-space Mutitape DTM $\xrightarrow{\text{can be converted to}}$ an $O(t(n))$-space single tape DTM

Why?

→ Since space is measured in # of tape cells used, the STDTM uses the same amt. of space as the MTDTM because it copies all the squares of each of the MTDTM's tapes.

SUMMARY: What is the relationship between all complexity classes discussed so far?

→ $P \subseteq NP$ -- just discussed

→ $NP \subseteq NPSPACE$

- Why? "Fact 1" (pg 125), that $TIME(f(n)) \subseteq SPACE(f(n))$

→ PSPACE = NPSPACE (Savitch's Thm.)

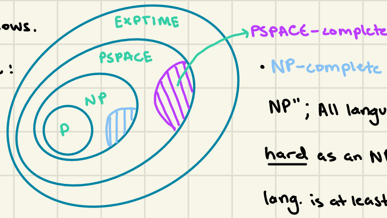
→ (PSPACE = NPSPACE) ⊆ EXPTIME

- Why? "Fact 2" (pg 125), that $SPACE(t(n)) \subseteq TIME(2^{O(t(n))})$
- $EXPTIME = \bigcup_k TIME(2^{O(n^k)})$ for all integers k.

→ In whole: $P \subseteq NP \subseteq (PSPACE = NPSPACE) \subseteq EXPTIME$

- We believe that all of these "⊆" are actually "⊊" (proper subsets)...but no one actually knows.

→ What we believe:



- NP-complete: the "hardest problems in NP"; All languages in NP are at most as hard as an NP-complete lang. An NP-comp. lang. is at least as hard as every lang in NP

→ UNKNOWN: is P ⊊ PSPACE or is P = PSPACE?

→ KNOWN: P ⊊ EXPTIME

- There are problems solvable by exponential-time DTM which are not solvable by poly-time DTM.

What does it mean for a language to be PSPACE-complete?

→ **DEFN:** A language B is PSPACE-complete if it satisfies 2 conditions:

1. B ∈ PSPACE
2. B is PSPACE-hard; all languages in PSPACE are poly-time reducible to B.

→ PSPACE-complete basically represents The hardest problems in PSPACE. They are all also even harder than NP-complete problems.

What is the relationship between PSPACE-hardness and NP-hardness?

→ **Thm:** For every language B, if B ∈ PSPACE-hard, then B is also ∈ NP-hard!

- PSPACE-hard problems are harder than NP-hard problems.

→ For ex, SAT can be p.t reduced to TQBF.

What is an example of a PSPACE-complete problem?

→ TBQF, a similar problem/language to SAT; involves boolean formulas – except now, we include **quantifiers**. T.Q.B.F. = "True quantified boolean formulas"

→ universal quantifier: $\forall$ "for all"

→ existential quantifier: $\exists$ "there exists"

What is a fully quantified boolean formula?

→ A quantified boolean formula is a formula w/ boolean vars that has quantifiers.

- The possible values for each variable is $\{0,1\}$, where $0 = \text{FALSE}$ and $1 = \text{TRUE}$.
- For ex: $\exists y\,[(x \vee y) \wedge (\bar{x} \vee \bar{y})]$ → "There exists a value for y such that the statement "$(x \vee y) \wedge (\bar{x} \vee \bar{y})$" evaluates to true.

→ A **fully quantified** boolean formula is one where each variable in the formula appears in the "scope" of at least one quantifier. e.g., every var gets a quantifier assigned to it

- For ex: The ex above is not fully quantified, but could be made so by adding a quantifier for x: $\boxed{EX1}$ $\forall x\, \exists y\, [(x \vee y) \wedge (\bar{x} \vee \bar{y})]$ → "For all possible vals of x, there exists some value for y s.t. $[\ldots]$ evaluates to true.

Okay, so what is the language TQBF?

→ TQBF $= \{\langle \varphi \rangle \mid \varphi$ is a **true** fully quantified Boolean Formula.$\}$

- basically, given an input like the one above, TBQF accepts it if its true. e.g., for $\forall x$, meaning for both $x=0$ & $x=1$, can we find an assignment for y s.t. the given statement evaluates to true?

→ String $\langle EX1 \rangle \in$ TQBF, because $\forall x = \{x=0,\ x=1\}$

- if $x = 0$, let $y = 1$: $[(\underbrace{\text{false} \vee \text{true}}) \wedge (\underbrace{\overline{\text{false}} \text{ or } \overline{\text{true}}})]$
  $[\ (\text{True}) \ \wedge \ (\text{True})\ ] = \text{TRUE}$

- if $x = 1$, let $y = 0$: $[(\text{true} \vee \text{false}) \wedge (\overline{\text{true}} \text{ or } \overline{\text{false}})] = \text{TRUE}$

Therefore $\langle Ex1 \rangle$ = True

→ **Thm:** TBQF is PSPACE-complete.

- TBQF is to PSPACE as SAT is to NP!
- all problems in PSPACE can be reduced to TQBF.

How do we prove that TQBF is PSPACE-complete?

→ We have to show that <u>1.</u> TQBF is in PSPACE, and <u>2.</u> all problems in PSPACE can be reduced to TQBF by a poly-time algorithm.

→ **Proof:** TQBF is in PSPACE. Let M_1 be a TM for TQBF. M_1 = "On input φ:

1. If φ has no quantifiers, scan φ to see if its true. (Basically doing what SAT does)
2. Else, if $\varphi = \exists x\,[\ldots]$:
   - 2a. Set $x = 0$ and run M_1 on φ
   - 2b. Set $x = 1$ and run M_1 on φ
   - 2c. If either returned true, accept.

3. If $\varphi = \forall x\, [\dots]$:

3a. Set $x = 0$ and run M_1

3b. Set $x = 1$ and run M_1

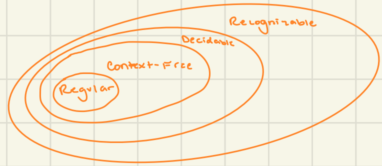
3c. If <u>both</u> return true, accept."

"if you can solve SAT in poly-time, you can solve every problem in NP (in poly-time)

"if ... TQBF, ... in PSPACE, including NP problems!

Final Exam Review

Language Relationships: Computability Theory



→ Regular: DFAs, NFAs, Reg. Expressions

→ Context-Free: CFGs, PDAs

→ if A and B are both CFLs, $A \cap B$ is NOT necessarily a CFL

→ ALL Regular Languages are $\in TIME(n)$ (decidable in linear $O(n)$ time) Why? Just have the TM imitate the lang's DFA.

→ ALL Context-Free Languages (and thus reg. too) are $\in$ the class P.

Language Relationships: Complexity Theory

→ The class P: problems solvable in poly-time by a ST DTM ... $\bigcup_k TIME(n^k)$

→ The class NP: " by a ST NDTM ... $\bigcup_k NTIME(n^k)$

→ The class PSPACE: problems solvable in poly-space by a ST DTM ... $\bigcup_k SPACE(n^k)$

→ The class NPSPACE: " by a ST NDTM ... $\bigcup_k NSPACE(n^k)$

→ P is not necessarily equal to NP... its unknown. Because one day, we might find an alg to solve NP problems like SAT in poly-time, deterministically. But we think that $P \subsetneq NP$

→ PSPACE = NPSPACE! Specifically, a NDTM that takes F(n)-space can be converted into a DTM that takes $F^2(n)$ space (Savitch's Thm.). But since both n and n^2 are polynomial, all NPSPACE problems are also in PSPACE.

→ Time v.s. Space:

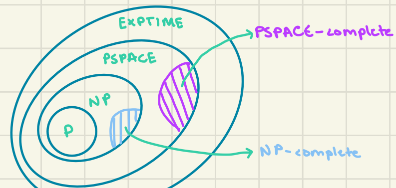
- problems cannot use more space than they do time. If a TM runs in f(n)-time, it runs in at most f(n)-space. Therefore, P, NP both $\subseteq$ PSPACE (and thus also NPSPACE)
- REVERSE: If a TM runs in f(n)-space, it runs in at most $2^{O(f(n))}$ - time.

  e.g., TM M, needs 4 squares to compute $(f(n)=4)$. The maximum time it can take is $2^4 = 16$ steps.

→ P $\subseteq$ NP $\subseteq$ (PSPACE = NPSPACE) $\subseteq$ EXPTIME → $\bigcup_k TIME(2^{n^k})$, aka problems solvable by a DTM in exponential time.

→ = We don't actually know if these are subsets. We don't even know if P = PSPACE; its possible just like its possible that P = NP. And its possible that PSPACE = EXPTIME.

→ Known: P ≠ EXPTIME. Which implies that at least one of the red-highlighted "subset" symbols is true. Bc there has to be a separation between P and EXPTIME at some level.



- NP-complete: the "hardest problems in NP"; All languages in NP are at most as hard as an NP-complete lang. An NP-comp. lang. is at least as hard as every lang in NP
- PSPACE-complete: same as above but replace "NP" with "PSPACE".

→ ALL languages in NP (and thus P) are decidable. If a lang is undecidable, it cannot be in NP.

Part 1: Automata and Languages

→ **Regular Expressions:** $A = \{good, bad\}$ $B = \{boy, girl\}$

- $A \circ B = \{goodboy, goodgirl, badboy, badgirl\}$
- $A^* = \{goodgoodbad, goodbad, badgood, \varepsilon\}$

→ $(0 \cup 1) \circ 0^* = 0000000, 100000, 00, 10, 0, 1$

→ $A = \{w \mid |w| \leq 3\} \longrightarrow (\Sigma \cup \varepsilon) \circ (\Sigma \cup \varepsilon) \circ (\Sigma \cup \varepsilon)$

→ $B = \{w \mid w$ has 000 as a substring$\} \longrightarrow (1 \cup 0)^* \circ 000 \circ (1 \cup 0)^*$

→ $C = \overline{B}$ ... w does not have "000" as a substring $\longrightarrow (1 \cup 01 \cup 001)^* \circ (\varepsilon \cup 0 \cup 00)$

→ **NFA-to-DFA**: make every possible subset of states in NFA, a state in the DFA.

NFA N, $Q = \{q_0, q_1, q_2\}$ ... DFA D, $Q' = \{q_1, q_2, q_3, q_1q_2, q_1q_3, q_2q_3, q_1q_2q_3, \emptyset\}$

→ DFAs: $\delta : Q \times \Sigma \longrightarrow Q$

→ NFAs: $\delta : Q \times \Sigma_\varepsilon \longrightarrow P(Q)$

→ PDAs : "An NFA with a stack"

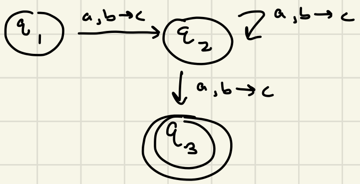
- At each step of computation, you can either push (add symbol to stack), pop (remove top symbol), or replace (aka pop+push)
- $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow P(Q \times \Gamma_\varepsilon)$

Q: curr. state

Σ_ε: curr. input symbol being read when transition occurs

Γ_ε: symbol currently at top of stack

$P(Q \times \Gamma_\varepsilon)$: given the specified curr. state, input, and stack symbol, this is the power set of possible (state to move to, action to execute on stack) combos.



$a, b \to c$: If the next input symbol is $\underline{a}$:

- pop $\underline{b}$ off the stack (if $b = \varepsilon$, pop nothing)
- replace it with c, e.g. push $\underline{c}$ onto the stack (if $c = \varepsilon$, push nothing)
- if $a = \varepsilon$, nondeterministic "automatic" move

"read a, replace b on stack with c"

→ **Pumping Lemma:**

1. $|s| \geq p$
2. $|xy| \leq p$
3. $|y| \geq 1$
4. $xy^iz \in B$

Turing Machines, Decidability, Reducibility

Turing Machines

→ A Turing Machine is a 7-tuple $(Q, \Sigma, T, \delta, q_0, q_{accept}, q_{reject})$ where:

1. Q = set of states
2. Σ = input alphabet
3. T = stack alphabet
4. $\delta : (Q \times \Sigma \times T) = Q \times T \to Q \times T \times \{L, R\}$
5. q_0: start state

→ STNTM: $\delta : Q \times T \to P(Q \times T \times \{L, R\})$

→ Multitape, singletape, nondeterministic, deterministic TMs all recognize the same set of languages.

Decidablity

→ Lang. is Decidable if ∃ a TM which always halts (accepts or rejects); never loops.

→ EXAMPLES:

- $A_{DFA} = \{<B, w> \mid B$ is a DFA that accepts $w\}$ (Algorithm: run $B(w)$ & return it)
- $A_{NFA, CFG, PDA}$; $E_{DFA, NFA, CFG, etc.} = \{<M> \mid M$ is a [m.o.comp.] and $L(M) = \emptyset\}$; $EQ_{NFA, CFG, etc.}$

→ Undecidable Languages:

→ A language A is decidable i.f.f. A and $\bar{A}$ are Turing recognizable

- $\bar{A}_{TM}$ is NOT recognizable

→ The complement of a decidable language is decidable.

Language	Proof
A_{TM}	diagonalization
E_{TM}	$A_{TM} \leq E_{TM}$
$HALT_{TM}$	$A_{TM} \leq_m HALT_{TM}$
EQ_{TM}	$E_{TM} \leq_m EQ_{TM}$
$\overline{E_{TM}}$	$A_{TM} \leq_m \overline{E_{TM}}$

Reducibility

→ General Reducibility $A \leq B$: Constructing a decider TM for A that uses a (given/implied) decider TM for B in its computation.

- THM: For $A \leq B$, if B is decidable, then A is decidable.
- THM: If A is known to be undecidable & you can prove that $A \leq B$, then it proves that B is undecidable.
- USE: To prove that a lang is undecidable, assume for contradiction that it is, & show that $A_{TM} \leq LANG$

→ Mapping Reducibility $A \leq_m B$: Constructing a decider TM for A that runs a computable function f on input $<a>$, and then calls the TM for B at **most once**, to run TM_B on the string outputted by f.

- computable function f: takes in an input "w" that would go in A. Outputs a B-format string "f(w)" s.t. $f(w) \in B$ i.f.f. $w \in A$.
- THM: for $A \leq_m B$, if A is not Turing-recognizable, B is not T-R !!
- USE: To prove that a lang isn't TR, show that $\overline{A_{TM}} \leq_m LANG$

→ If $A \leq_m B$, it does NOT imply that $B \leq_m A$.

→ If A decidable, $\bar{A}$ and A^* decidable

Time Complexity

→ Time defined as $O(f(n))$, the MAXIMUM (big-O notation) # of steps a TM would take to decide a problem with an input string of length n.

→ The Class P : $\{ L \mid L$ is decidable by a polynomial-time STDTM $\}$ ≈ "easy" problems

- Ex: PATH = $\{ \langle G,s,t \rangle \mid G$ is a directed graph w/ a directed path from node s to node t. $\}$
- Includes all context-free languages

→ The Class NP : $\{ L \mid L$ is decidable by a polynomial-time ST<u>N</u>TM $\}$ OR ≈ "hard" problems

$\{ L \mid \exists$ a poly-time verifier for $L \}$

- Verifier : A TM V that takes an input $\langle w,c \rangle$ where (w = a string input to L) and (c = a certificate e.g. "proposed solution" proving that $w \in L$). V basically checks if the given c, which is usually created based on what w is, e.g $c(w)$, is in the lang. L or not. Aka

  $L = \{ w \mid \exists$ a string c s.t. V accepts $\langle w,c \rangle$. $\}$

- Ex: HAMPATH (like PATH except "Hamitonian path"), SAT

→ Relationship between Time & different types of TMs:

- Multitape $f(n)$ - time TM ⟶ Single tape $O(t(n)^2)$ - time TM
- $t(n)$ - time NTM ⟶ $2^{O(t(n))}$ - time DTM.

  Meaning, all languages in NP are solvable by a DTM in exponential time.

Poly-time Reducibility

→ Poly-time Reduction $A \leq_p B$: Same as mapping red., but must be computable in poly-time

- The method of a p.t. reduction $A \leq_p B$ is to a) assume that B has a poly-time DTM, b) create a reduction from A-elements to B-elements, and c) Use the reduction func. & the assumed TM for B to create a poly-time DTM for A.

→ USE: To prove that problems are hard. Given a lang A that we know is NP-complete, we can prove that B is NP-complete by showing $A \leq_p B$: "Assume" that B is easy (Poly-time DTM), reduce A to it. Since we alr know that A isn't easy, our "assumption" is proved false.

→ THM: If $A \leq_p B$ and $B \in P$, A is also $\in P$.

→ NP-hard : A lang. B is NP-hard if $A \leq_p B$ for all languages $A \in NP$.

- Every lang. in NP can be reduced to B in polytime.
- Meaning, if we had a poly-time DTM for B, then we'd be able to construct a poly-time DTM for A.
- NP-hard langs aren't necessarily $\in$ NP. For ex, A_{TM}
- "problems which are at least as hard as NP, or harder.

→ NP-complete : A lang B is NP-complete if it is NP-hard, AND $B \in NP$. All NP-complete are also NP-hard.

- "all of the hardest problems in NP."

→ THM: If a lang. B is in NP and lang A is NP-complete, then if $A \leq_p B$, it proves that B is NP-complete.

- EX: HAMPATH $\leq_p$ UHAMPATH proves that UHAMPATH is NP-comp.

→ Examples of NP-complete problems:

Language	Proof
SAT	The o.g.
HAMPATH	idk
UHAMPATH	HAMPATH $\leq_p$ UHAMPATH
3SAT	SAT $\leq_p$ 3SAT
VERTEX-COVER	3SAT $\leq_p$ V-C
IND.-SET	V-C $\leq_p$ IND-SET

Review Session Notes

NFAs	→ every DFA is automatically an NFA

Pumping Lemma

→ A way to show that a language is <u>not</u> regular.

→ if lang. A is reg, $\exists$ an int p s.t.

- For all strings in A, they are at least length p $(s \geq |p|)$
- $\forall s$, there is some way to split s into $s = xyz$

  s.t. $|y| \geq 1$ and $|xy| \leq p$, s.t.

  $\forall$ int i, $(xy^iz \in A)$

→ To prove: Find a string in the lang for which one of the 3 conditions isn't satisfied.

→ Ex: $A = \{0^n 1^n \mid n \geq 0\}$

assume for contr. that A is regular. let the p.l. be $p = 3$

If it were regular All strings $s \in A$ would sat. the conditions.

let $s = 000111$. possible splits:

1) $\underbrace{0}_{x}\underbrace{0}_{y}\underbrace{0111}_{z}$; only possibility that doesn't violate condition $|xy| \leq p$

2) $\underbrace{0}_{x}\underbrace{00}_{y}111$;

- xy^3z For choice 1 $= 00\,000\,111$ ... not in A!

→ For exam: when we decide a p, we can't just pick any s. We need s to be a string <u>in terms of p</u> so that it works no matter what p is.

- B/c we are "assuming" what p would be.

PRACTICE EXAM

a CFG can generate a reg. & non reg lang... a DFA can only gen. a reg. lang.

all CF are reg, not v.v.

1) a. ~~False~~ True
✓ b. True
✓ c. True
✓ d. False
✓ e. False
✓ f. True
✓ g. Nobody knows
✓ h. True
✓ i. Nobody knows
✓ j. True

3SAT is NP-complete.... if its in P, all languages in NP are in P. so, True?

$P \subseteq NP \subseteq PSPACE \subseteq EXPTIME$

$\downarrow$ a,b — $\downarrow$ a,b,~~c~~ — $\downarrow$ a,b,[redacted] — $\downarrow$ a,b,c,d,e

2) ✓ $0^m 1^m 0^n 1^n \mid m,n \geq 0$ (a above 1^m, b above 1^n)

000111 a b — 0011

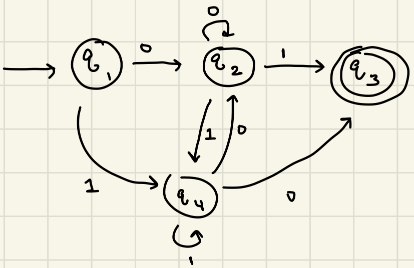
000111 aaaa bbbb — aabb

$$\boxed{\begin{array}{l} R_1 \to \varepsilon \mid R_2 R_3 \\ R_2 \to 0 R_2 1 \mid \varepsilon \\ R_3 \to 0 R_3 1 \mid \varepsilon \end{array}}$$

SOL: $S \to AA$

$A \to 0A1 \mid \varepsilon$

3) ✓ NFA for language $A = \{w \mid w \text{ ends in } 01 \text{ or } 10\}$.



5) $\Sigma^* = 0, 01, 10, 110001, 01101110, \varepsilon$

$B \leq_m E_{DFA}$: given input to B, output input to E_{DFA}

→ DFA B always accepts? as long as input is correct ~~form~~ alphabet. So $\underline{B}$ rejects any DFA encoding which d.n. accept, like "1101" for ex. If ⟨b⟩ rejects "011101", output ⟨A⟩, where A = a DFA that accepts strings ending in 1.

• if ⟨b⟩ ∈ B, then ε accepted by ⟨b⟩, which means that the st. state = accept state!

ANS: A computable function $\underline{f}$ that reduces B to E: f = ~~our~~ on input A:

1. Let A' = a DFA with one state, q_1, no accept states, and transition functions like so: $\to (q_1)$ ⟲ 0,1

2. If ⟨A⟩ were an element of B, it would accept the empty str ε, since $\varepsilon \in \Sigma^*$. If the start state q_0 of B is equal to the accept state, output A'

3. Run ⟨A⟩ on

1. CFGs for the following languages, where $\Sigma = \{a, b, c\}$

a) $A = \{a^i b^j \mid i > j \geq 0\}$

aaabb   aaaaabbbb

$R_1 \to a \mid aR_1 \mid aR_2$

$R_2 \to aR_2b \mid \varepsilon$

b) $B = \{a^i b^j c^k \mid i = j \text{ or } i = k \text{ where } i, j, k \geq 0\}$

i = j :

aaabbbcc

aabbcccc

aaabbbccccc

$R_1 \to R_2 R_3 \mid \varepsilon$

$R_2 \to aR_2b \mid \varepsilon$

$R_3 \to cR_3 \mid \varepsilon$

i = k :

aaabccc

aabbbbcc

$R_1 \to aR_1c \mid \varepsilon \mid R_2$

$R_2 \to bR_2 \mid \varepsilon$

$R_1 \to R_2 \mid R_4 R_5 \mid \varepsilon$

$R_2 \to aR_2c \mid \varepsilon \mid R_3$

$R_3 \to bR_3 \mid \varepsilon$

$R_4 \to aR_4b \mid \varepsilon$

$R_5 \to cR_5 \mid \varepsilon$

c) $C = \{a^i b^j c^k \mid i + j = k \text{ where } i, j, k \geq 0\}$

aabcccc

aabbcccc

abbbbccccc

$R_1 \to aR_1c \mid \varepsilon \mid R_2$

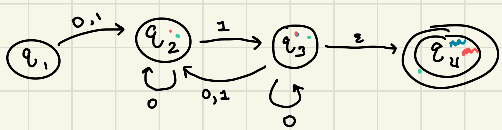
$R_2 \to bR_2c \mid \varepsilon$

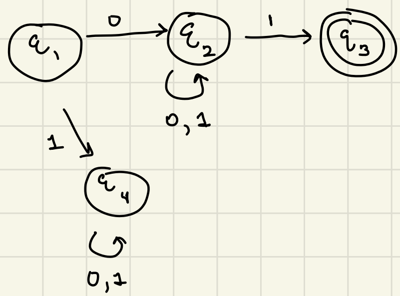
~~0001111~~0101

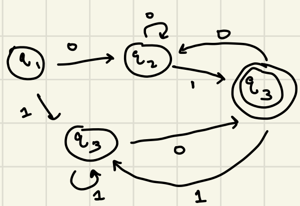
01011000010<u>10</u>

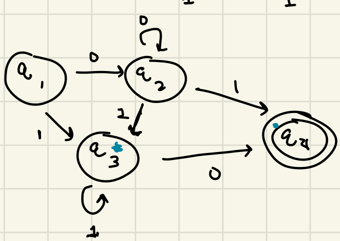
NFA: none arrows for a symbol allowed

· mult. per symbol allowed









7) $A = \{0^i 1^j \mid i,j \geq 0$ and i/j is an integer$\}$

• For ex, 00000000111

→ Assume contrary... A is reg. A satisfies the p.l. Let p be the pumping length given by the lemma. Choose $\underline{s}$ to be the string $0^{2p} 1^p$

(For ex, if p = 3 then s = 000000111)

→ $\underline{s}$ is obvi a member of A b/c $i = 2p$, $j = p$ and $i/j = p$. Its also obvi at least length p.

→ The p.l. then garuantees that s can be split somehow into x, y, z s.t. the 3 cond are satisfied: 1) $|y| \geq 1$ 2) $|xy| \leq p$ 3) $xy^iz \in A$ for int i

We consider all possible ways to split $\underline{s}$ into x, y, z to show that this result is impossible.

1) y consists only of 1s -- not possible, because if $s = 0^{2p} 1^p$, then __at least__ 2p symbols show up bfore first "1", meaning cond 2 viol... $|xy| \leq p$ is __false__

2) y consists only of 0s. For ex, if p = 3, $s = \underbrace{0}_{x}\underbrace{0}_{y}\underbrace{0000000111}_{z}$

Conditions 1 ($|y| \geq 1$) and 2 ($|xy| \leq p$) are satisfied. But cond. 3 is not. let i = 3, then $s_3 = \underbrace{00}_{x}\ \underbrace{000}_{y}\ \underbrace{000\ 111}_{z}$ is __NOT__ $\in$ A. in s_3, i = 8 and j = 3, but 8/3 not an int.

3) y has 0s and 1s — this imm. viol. con 2 b/c first "1" not appear until after 2p symbols so xy would have a minimum length of 2p+1. (2p+1) __not__ $\leq$ p

Thus, __not__ reg. lang.

6.) if $A \leq_M B$ & A not T-R, then B not T-R

By reducing $\overline{A_{TM}} \leq_m B$, proves that B not T-R

$\overline{A_{TM}} = \{\langle M \rangle \mid M$ is a TM and M doesn't accept w$\}$

→ ALG: on input $\langle M, w \rangle$:

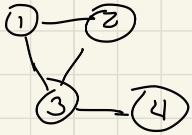
* if $\langle M, w \rangle \in \overline{A_{TM}}$, then output shld be $w: \langle 1(M,w) \rangle$
  - then it means that M doesn't accept w...

$\boxed{\text{OUTPUT } \langle 1(M,w) \rangle}$

1) $\langle M, w \rangle \in \overline{A_{TM}}$ ... output is 1y ✓✓

2) $\langle M, w \rangle \notin \overline{A_{TM}}$ ... output is 1y where $y \notin \overline{A_{TM}}$ ✓✓

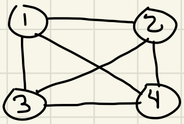
4) VC $\longrightarrow$ IS

G: 

input: ~~G~~ $\langle G, 1 \rangle$

G' = complete graph on 4 vertices

$\rightarrow$ all $4(4-1)/2 = \underline{6}$ undir. edges

G': 

return $G', 1$

This doesn't work?

backwards does work?

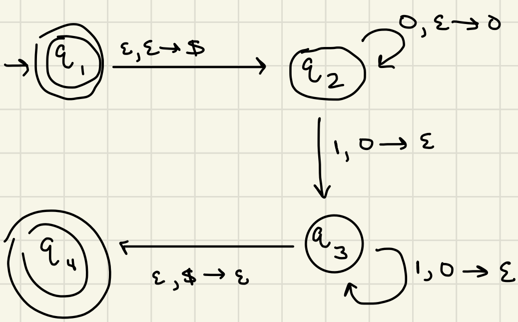
5) • if $\langle w \rangle \in B$, ~~#~~ w accepts everything! output $\langle w' \rangle$ where $w' = \overline{w}$.

• if $\langle w \rangle \notin B$, w could be any other DFA. output some DFA which accepts something (e.g. not empty) . . . . $\overline{w}$ would still not be the empty lang right?

1. output A'

$\{0^n 1^n \mid n \geq 0\}$ 000111

$\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow P(Q \times \Gamma_\varepsilon)$



~~0~~
~~0~~
~~0~~
$

a) 3SAT ∈ NP HP ∈ NP . . . if HP is in P, so is 3SAT

bc HP is NP-comp.

TRUE

b) 3SAT in P, 3SAT NP-comp . . . . but NP-hard doesn't nec mean ∈ NP

so FALSE

c) 3SAT is NP-comp TRUE

3) "from 3SAT to A_{TM}"

$= 3SAT \leq_p A_{TM}$

"from 3-color to STUDENTS"

$= 3COLOR \leq_p STUD$

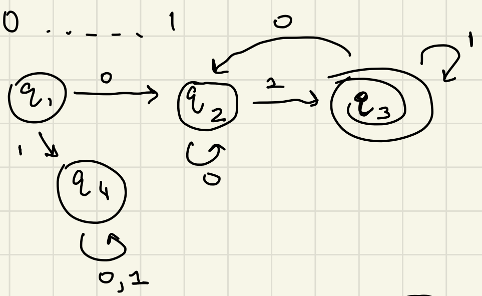
3SAT : $(x_1 \vee x_2 \vee x_3) \wedge (x \vee x \vee x) \wedge \dots$

MIDTERM

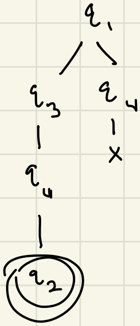
a) True b) T c) T d) F e) F

2) 0 1

001010101101



3) a) Yes



b

a

b

b) no

4) 111111111111 or 00000

(a) $0^* \cup 1^*$ — yes

(b) $\{ww \mid w \in \Sigma^*\}$

p.l.: for p.length p

1) $|s| \geq p$
2) $|y| \geq 1$
3) $|xy| \leq p$
4) $xy^iz \in B$

p.l.: let $s = ww$

e.g., if $\Sigma = \{a, b\}$

and $w = abaabb$ and $p = 3$

$s = \underbrace{aba}_{xy}abb\,abaabb$

let $x = ab$, $y = a$

xy^2z = abaa~~a~~abbabaabb

odd #, so obvi not $\in B$

3. "On input $\langle R, S \rangle$:

1. let the set $X = L(R)$. let set $Y = \overline{L(S)}$.
2. if $(L(R) \cap \overline{B}) = \emptyset$, accept.

else, reject?