

①

جامعة

حلبة اول

## Formal Languages and Automata Theory

نظريات، مسائل، نماذج

### Computer Science

- What kind of things can we compute?
- How fast can we compute?
- How much memory do we need?

Model of computation : theoretical abstract machine

Theory of "computation"

Simple questions , difficult answers - still far from closed

Various models of computation (in order of increasing power) :

- ① Finite memory : finite automata ; regular expressions
- ② Finite memory with stack: pushdown automata
- ③ unrestricted :

Turing machine

Java , Pascal

$\lambda$ -calculus

( you can add any sufficiently powerful PL to the list )

In parallel : Noam Chomsky formalized notion of language & grammar

- ① Right-linear
  - ② Context-free
  - ③ Unrestricted
- \*—————>
- context-sensitive grammars  
linear bounded automata  
(not in this course)

Practical Applications :

\* Regular expressions are used in many systems

grep    global regular expression print    } Search for a text  
 find str (windows)                                 string in a file

: \> more \*.txt

Finite Automata : model protocols ; electronic circuits

DTD (Document Type Definition)

DTD defines the structure and legal elements & attributes of an XML document.

DTD is based on context-free grammars. (CFG)

person (name, addr?, child\*)

CFGs are used to describe the syntax of every PL.

When developing software : limitations of what software can do

\* undecidable : no program whatever can do it

+  
Example : write a compiler that refuses the programs that do not print anything on screen

\* Intractable : there exists programs, but not fast

NP-completeness

First we simplify the discussion on algorithm problems.

Decision Problem: function with one-bit output : "yes" or "no"

- { \* set A of possible inputs
- \* subset  $B \subseteq A$  : "yes"



Example : PRIME , CONNECTED

Abstraction : possible input to decision problem:

finite length strings over some fixed alphabet

Graph, natural numbers, ... can be encoded

alphabet :  $\Sigma$  any finite set  $\{0, \dots, 9\}$   $\{a, b, c, \dots\}$

String over  $\Sigma$  : any finite-length sequence of elements of  $\Sigma$

$x, y, z$  refer to strings

$\Sigma = \{a, b\}$       aabb : string

$|x|$  : number of symbols in  $x$

empty string :  $\epsilon$  (special symbol) null or empty string

$$|\epsilon| = 0$$

$a^n$  : string of  $a$ 's of length  $n$ .

$$a^0 \stackrel{\text{def}}{=} \epsilon$$

$$a^{n+1} \stackrel{\text{def}}{=} a^n a$$

$$\text{Example. } a^4 = aaaa \quad a^0 = \epsilon$$

Set of all strings over  $\Sigma$  :  $\Sigma^*$

Example:  $\Sigma = \{a, b\}$

$$\Sigma^* = \{\epsilon, a, b, ab, ba, \dots\}$$

$$\{a\}^* = \{\epsilon, a, aa, aaa, \dots\} = \{a^n \mid n \geq 0\}$$

$$\emptyset^* = \{\epsilon\}$$

Note:  $\emptyset, \{\epsilon\}, \epsilon$  are different things

Note. Do not confuse sets and strings

$$\{a, b\} = \{b, a\} \text{ but not } ab \neq ba$$

$$\{a, a, b\} = \{a, b\} \text{ but not } aab \neq ab$$


---

Concatenation. takes two strings  $x$  and  $y$  and makes a new string  $xy$  by putting them together

- associative  $(xy)z = x(yz)$
- $xy$  and  $yx$  not equal in general
- null string  $\epsilon$  is identity  $\epsilon x = x\epsilon = x$

$$|xy| = |x| + |y|$$

$x^n$  : concatenating  $n$  copies of  $x$

$$x^0 \stackrel{\text{def}}{=} \epsilon$$

$$x^{n+1} \stackrel{\text{def}}{=} x^n x$$

⑤

$$(aab)^5 = aabaabaabaaabaaab$$


---

Language: set of strings (over some alphabet)

Language describes problems with "yes"/"no" answers

$L$  = All strings containing the substring "eh" ( $\Sigma = \{a, b, \dots, z\}$ )

tehran, ehsan, mehr are in  $L$

$\epsilon$ , sharif, aban are not in  $L$

$$L = \{x \in \Sigma^* \mid x \text{ contains the substring "eh"}\}$$


---

$$\text{Example. } L = \{s \# s \mid s \in \{a, \dots, z\}^*\}$$

Which of the following are in  $L$ ?

ab # ab

yes

ab # ba

no

a # a #

no

①

باعثی

جلد دوم درس نظری

یادآوری

 $\Sigma^*$  : set of all strings on  $\Sigma$ Language : subset of  $\Sigma^*$  (finite or infinite)

Example:

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

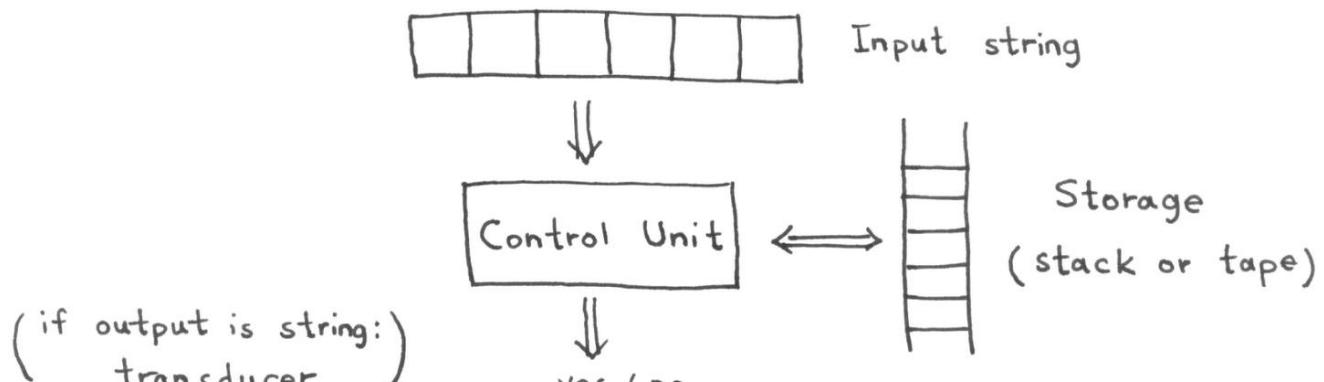
$$L_1 = \{a, ba, abc\}$$

$$L_2 = \{\epsilon, a, aa, aaa, \dots\}$$

$$L_3 = \emptyset$$

Automaton (plural : automata)

Abstract model of a computer



Control Unit:  $\begin{cases} - \text{has some finite memory} \\ - \text{Keeps track of what step to execute next} \end{cases}$

Additional Storage is infinite. We never run out of memory

Finite Automata

Automaton without additional storage

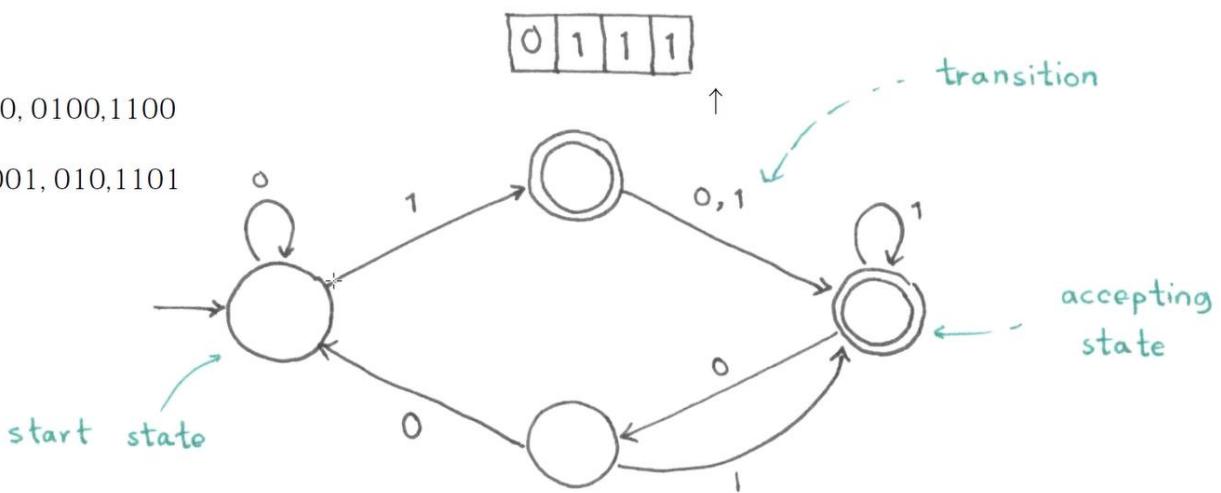
Only the finite memory in the control unit is available

It can remember only a finite number of properties of the past input

②

No: 00, 0100, 1100

Yes: 001, 010, 1101

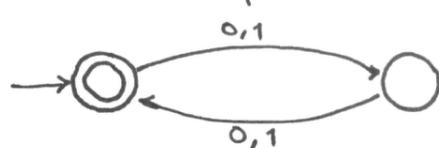
Let  $M$  be a finite automaton over  $\Sigma$ 

$$L(M) = \{x \in \Sigma^* \mid M \text{ accepts } x\}$$

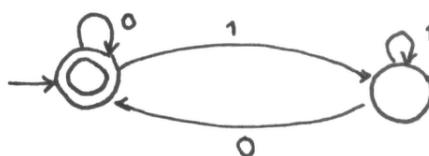
What is the language of the following finite automata?



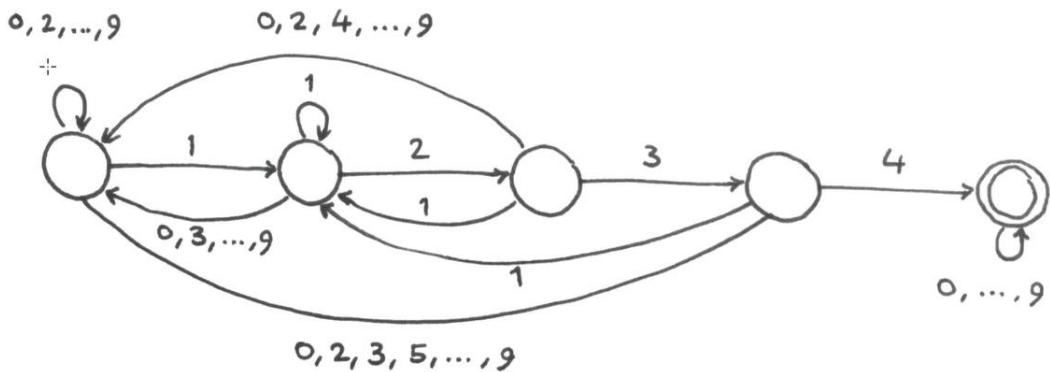
even numbers (1's)



binary string of even length

 $\{x \mid x \text{ ends in } 0\} \cup \{\epsilon\}$ 

|   |   |   |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |
| 0 |   |   |



How can we write a program and code this automaton?

## Formal Definition:

Deterministic Finite Automaton (DFA)  $M = (Q, \Sigma, \delta, q_0, F)$

$Q$ : a non-empty finite set of states

$\Sigma$ : alphabet

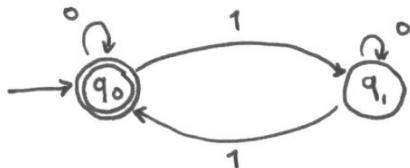
$\delta$ : transition function  $Q \times \Sigma \rightarrow Q$

$q_0 \in Q$  start state

$F \subseteq Q$  set of final states

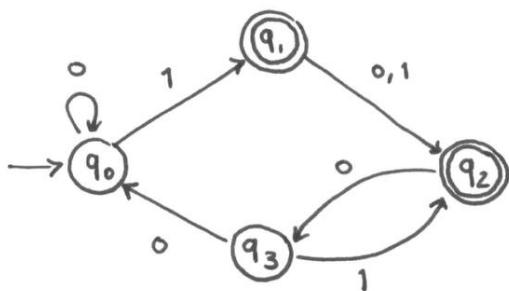


$$\delta(state_0, \text{symbol}) = state_1$$



$$\delta(q_0, 0) = q_0 \quad \delta(q_1, 0) = q_1$$

$$\delta(q_0, 1) = q_1 \quad \delta(q_1, 1) = q_0$$



$$Q = \{q_0, q_1, q_2, q_3\}$$

$$\Sigma = \{0, 1\}$$

$\delta$ :

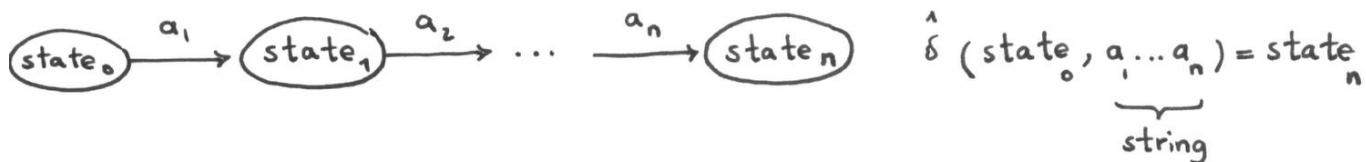
$q_0$ : start

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

| $\delta$ | 0     | 1     |
|----------|-------|-------|
| $q_0$    | $q_0$ | $q_1$ |
| $q_1$    | $q_2$ | $q_2$ |
| $q_2$    | $q_3$ | $q_2$ |
| $q_3$    | $q_0$ | $q_2$ |

## Extended transition function

$\hat{\delta}$ : mapping from  $(state, string)$  to the corresponding state



$$L(M) = \{x \in \Sigma^* \mid \hat{\delta}(q_0, x) \in F\}$$

(4)

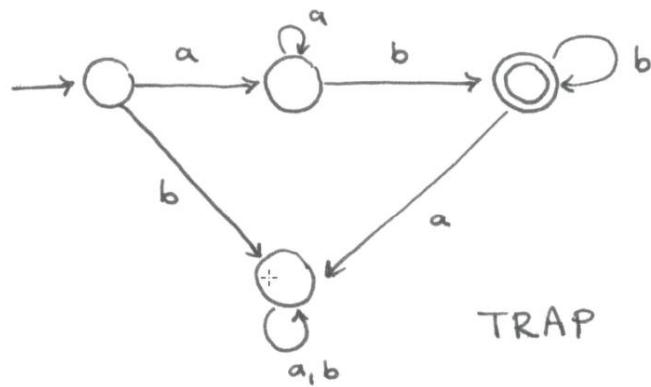
Definition: A language  $L \subseteq \Sigma^*$  is regular if a finite automaton accepts it.

Question: Are all languages regular?

example.  $L_1 = \{0^n 1^n \mid n \in \mathbb{N}\}$

Remember how many 0's are, check if they are equal to 1's

example.  $L_2 = \{a^n b^m \mid n, m \geq 1\}$



example.  $L_3 = \{w \mid w \text{ contains equal number of substrings } 01 \text{ and } 10\}$

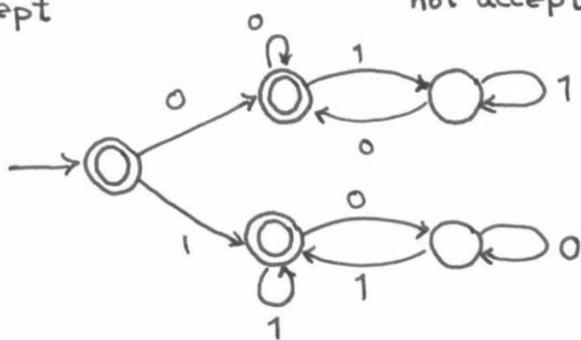
حالات

حالات مرسوب

$L_1 = \{ \omega \mid \omega \text{ contains equal number of substrings } 01 \text{ and } 10 \}$

0 1 ... 1 0  
 ↓  
 not accept      accept

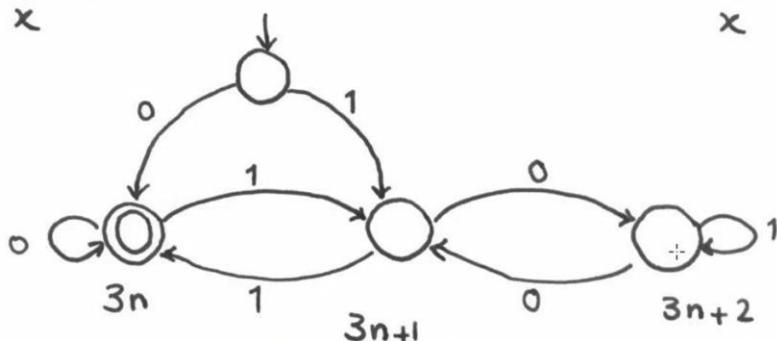
1 0 ... 0 1  
 ↓  
 not accept      accept



$L_2 = \{ \omega \in \{0,1\}^* \mid \omega \text{ is divisible by 3} \}$

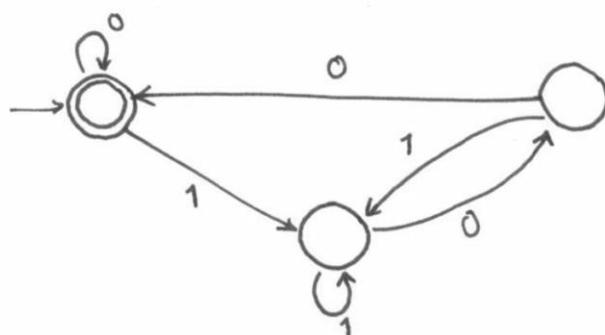
$$\underbrace{\cdot \cdot \cdot \cdot}_{x} 0 = 2x$$

$$\underbrace{\cdot \cdot \cdot \cdot}_{x} 1 = 2x+1$$

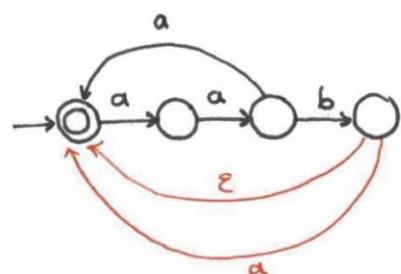
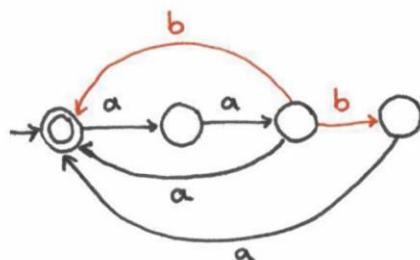
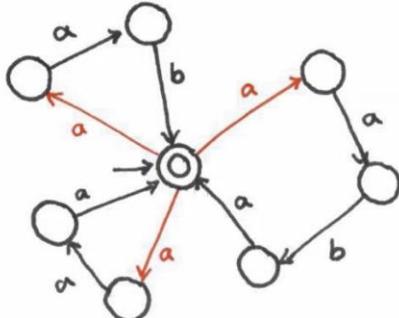


$L_3 = \{ \omega \in \{0,1\}^* \mid \omega \text{ is divisible by 4} \}$

و  $\vdash$   $\omega$  is of the form  $\epsilon, 0, v00 \quad (v \in \Sigma^*)$



$$L = \{aab, aaba, aaa\}^* = \{x_1 x_2 \dots x_k \mid k \geq 0 \text{ and } x_i \in \{aab, aaba, aaa\}\}$$



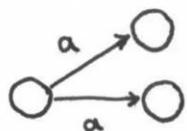
Deterministic finite automaton :  $\text{state}_0 \xrightarrow{a} \text{state}_1$

o

$$\delta(\text{state}_0, a) = \text{state}_1$$

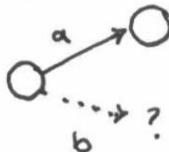
Non-deterministic finite automaton may have:

multiple outgoing edges  
with same symbol

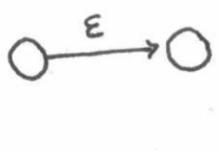


no outgoing edges  
for some symbols

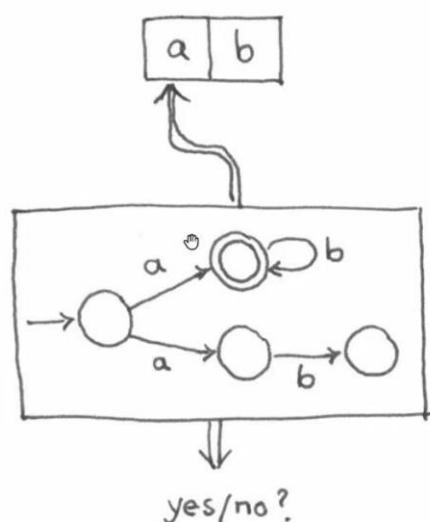
$$\Sigma = \{a, b\}$$



$\epsilon$ -transitions

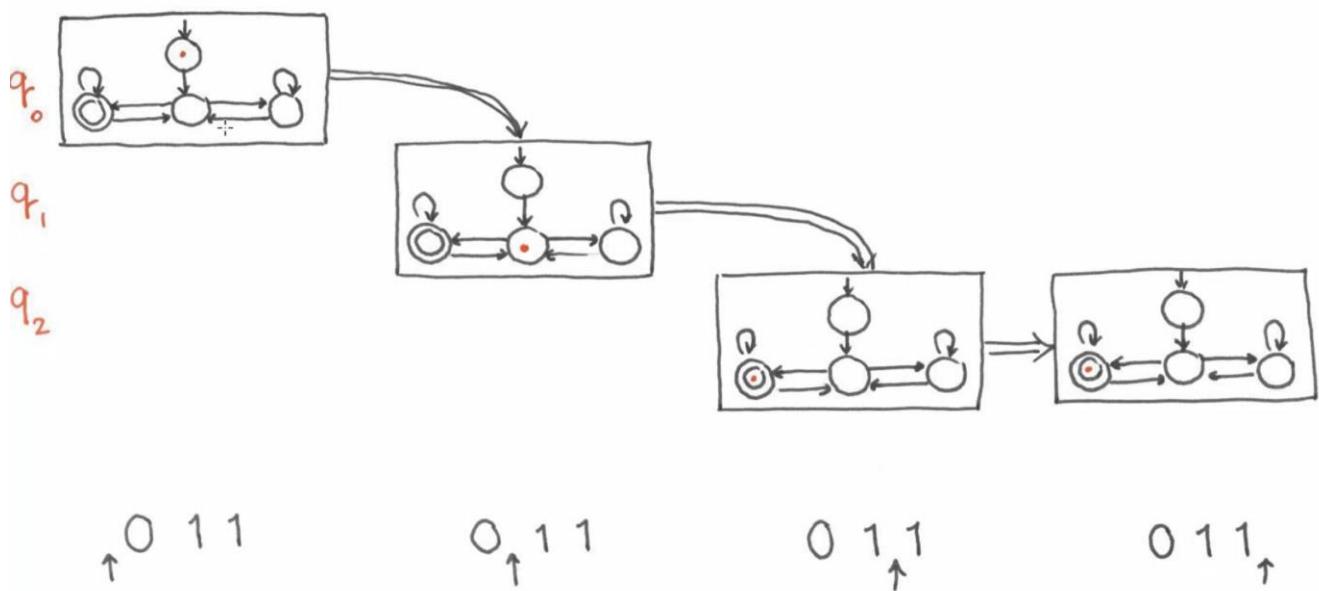


On reading an input symbol, NFA can "choose" to make a transition to one of selected states.



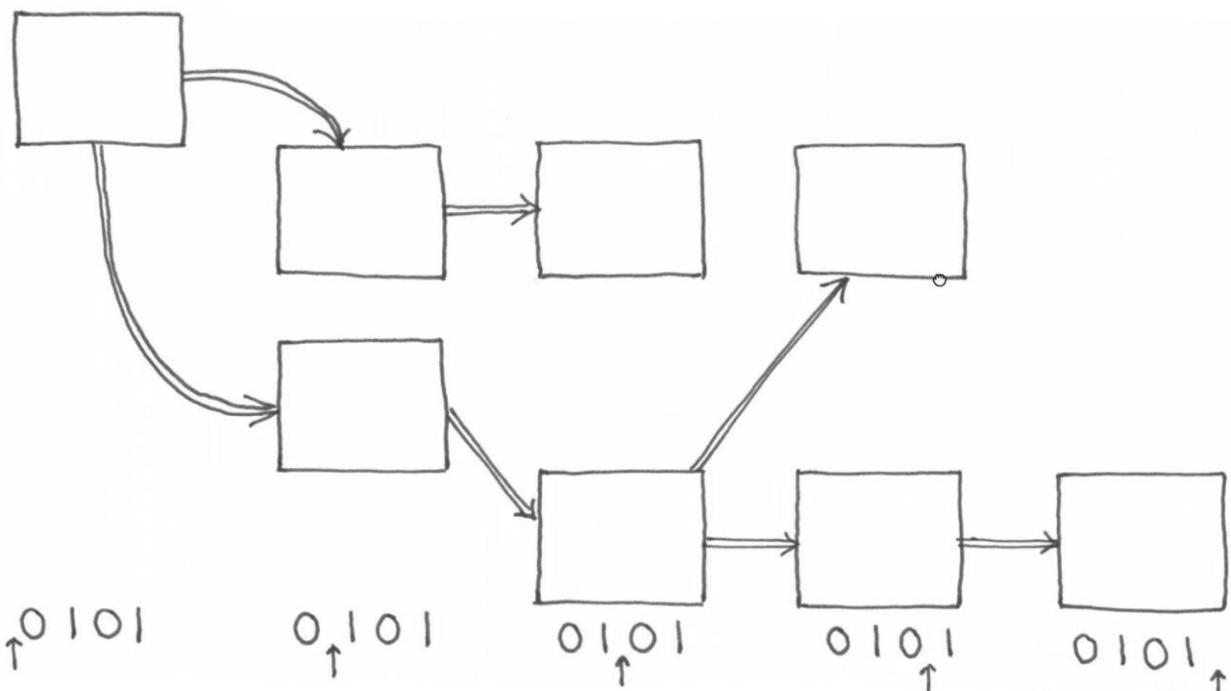
### ③ Tracking Computation

Configuration: current state and remaining input  $(q, w) \in Q \times \Sigma^*$



Deterministic: each step is fully determined by the configuration of the previous step

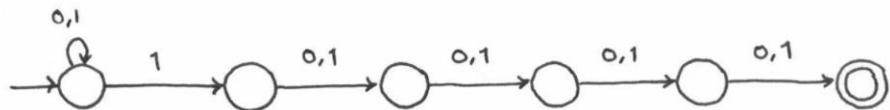
Non-deterministic: at each step the computation is allowed to proceed in multiple ways (zero, one or more)



accept iff at least one configuration accepts  
reject iff no configuration accepts

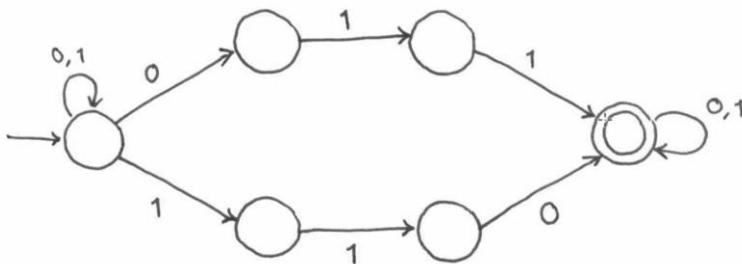
$$L = \{x \in \{0,1\}^* \mid \text{5th symbol of } x \text{ from right is 1}\}$$

(4)

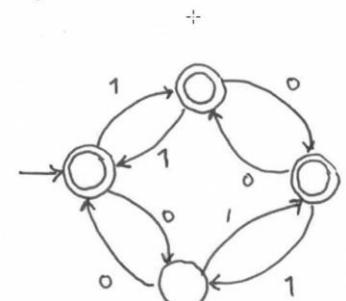
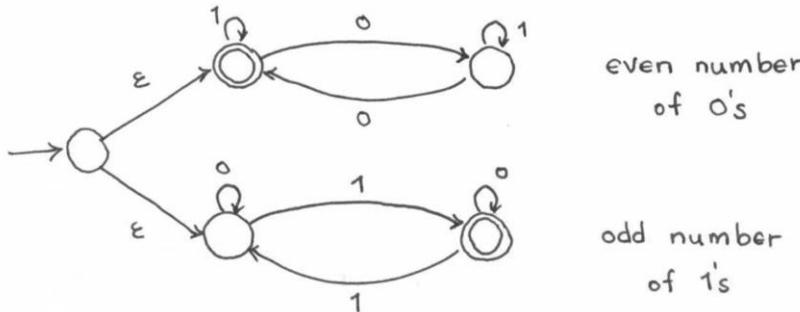


For any input  $x$ , if its 5th symbol from right is 1, then there exists some computation path that ends in accepting state

$$L = \{w \mid w \text{ contains } 011 \text{ or } 110\}$$



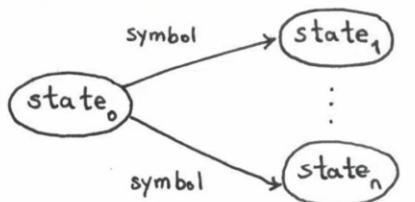
Construct a finite automaton that accepts all strings with an even number of 0's or an odd number of 1's



### Formal Definition

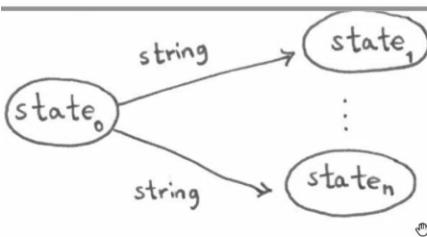
Similar to DFA  $N = (Q, \Sigma, \delta, q_0, F)$

$\delta(q, a)$  : set of states that  $N$  could move from  $q$  on input  $a$



$$\delta(state_0, symbol) = \{state_1, \dots, state_n\}$$

$$\delta: Q \times \Sigma \rightarrow 2^Q$$



$$\hat{\delta}(state_0, string) = \{state_1, \dots, state_n\}$$

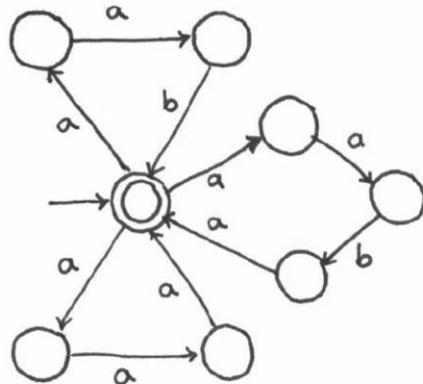
$$L(N) = \{x \in \Sigma^* \mid \hat{\delta}(q_0, x) \in F\}$$

①

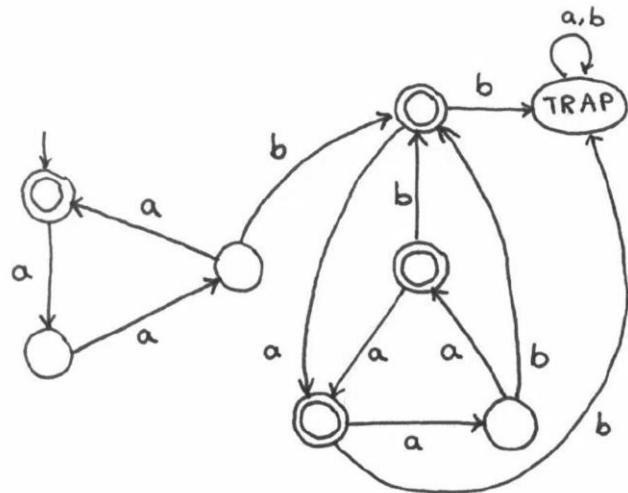
اینهاست

جلد چهارم درس نظری

$$L = \{aab, aaba, aaa\}^*$$



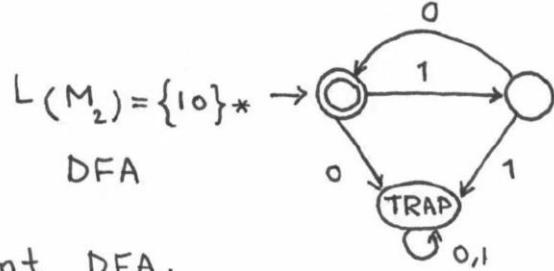
NFA



DFA

Equivalence. Two automata are equivalent if their languages are the same. For  $M_1, M_2$ :  $L(M_1) = L(M_2)$

Example of equivalent automata.



Theorem. Every NFA has an equivalent DFA.

Corollary. A language is regular iff it is recognized by an NFA

From NFA to DFA

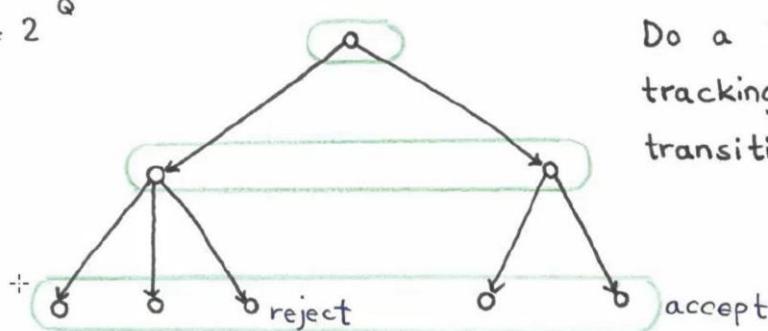
Input:  $N = (Q, \Sigma, \delta, q_0, F)$

Output:  $M = (Q', \Sigma, \delta', q'_0, F')$

$$Q' = ?$$

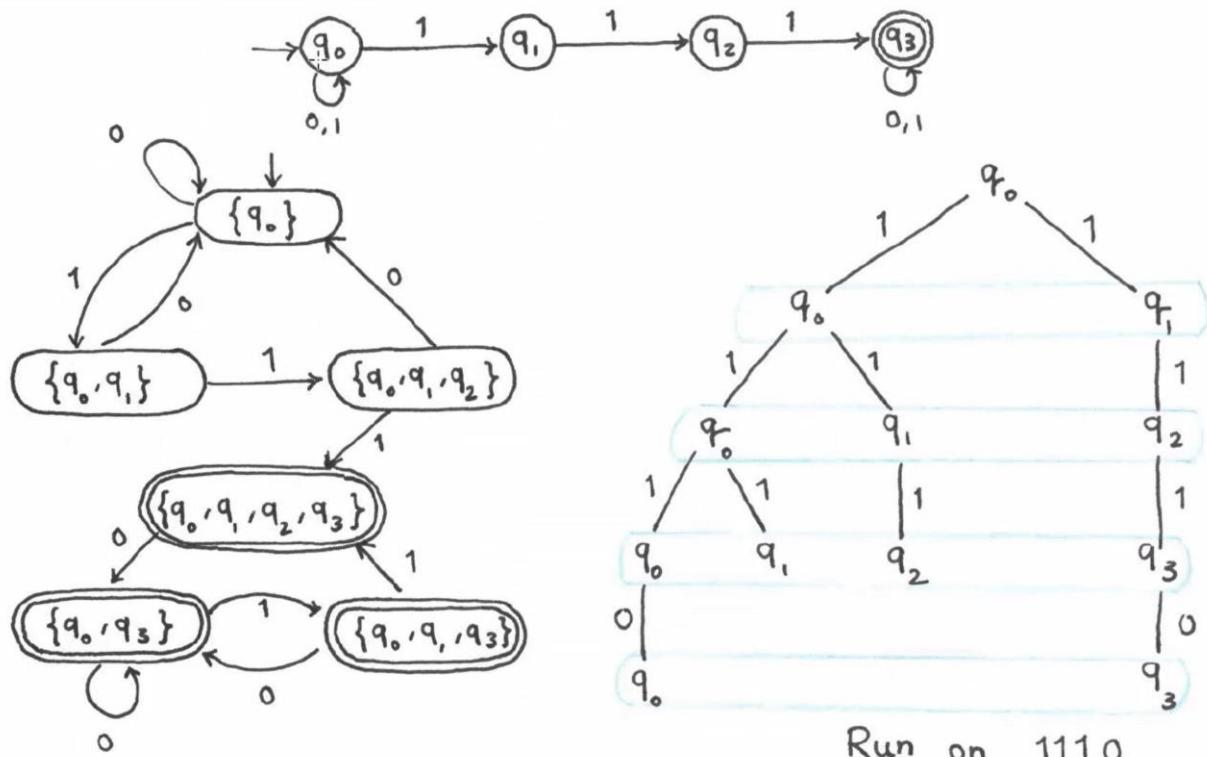
Assume (for now) that there are no  $\epsilon$ -transitions

$$\text{Idea: } Q' = 2^Q$$

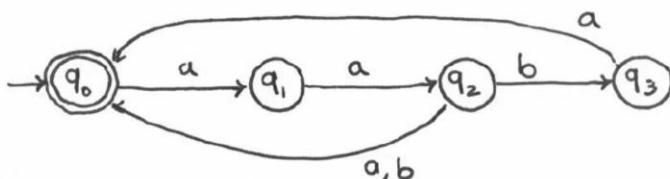


Do a BFS on tree  
tracking set of states  
transitioned to

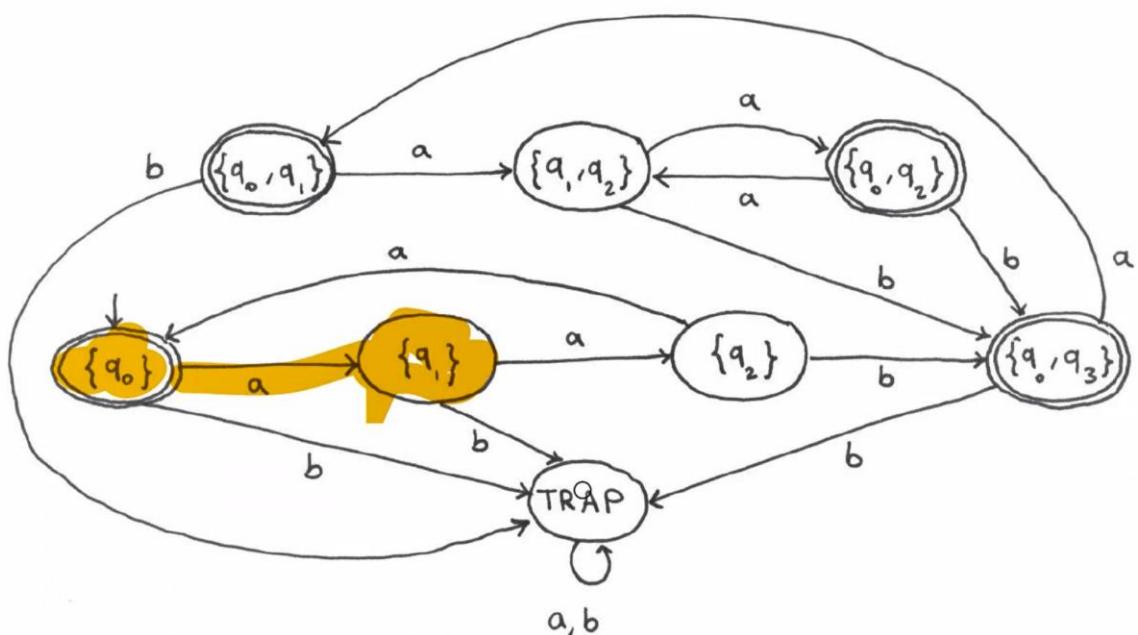
(2)



Example. Non-deterministic Finite Automaton :

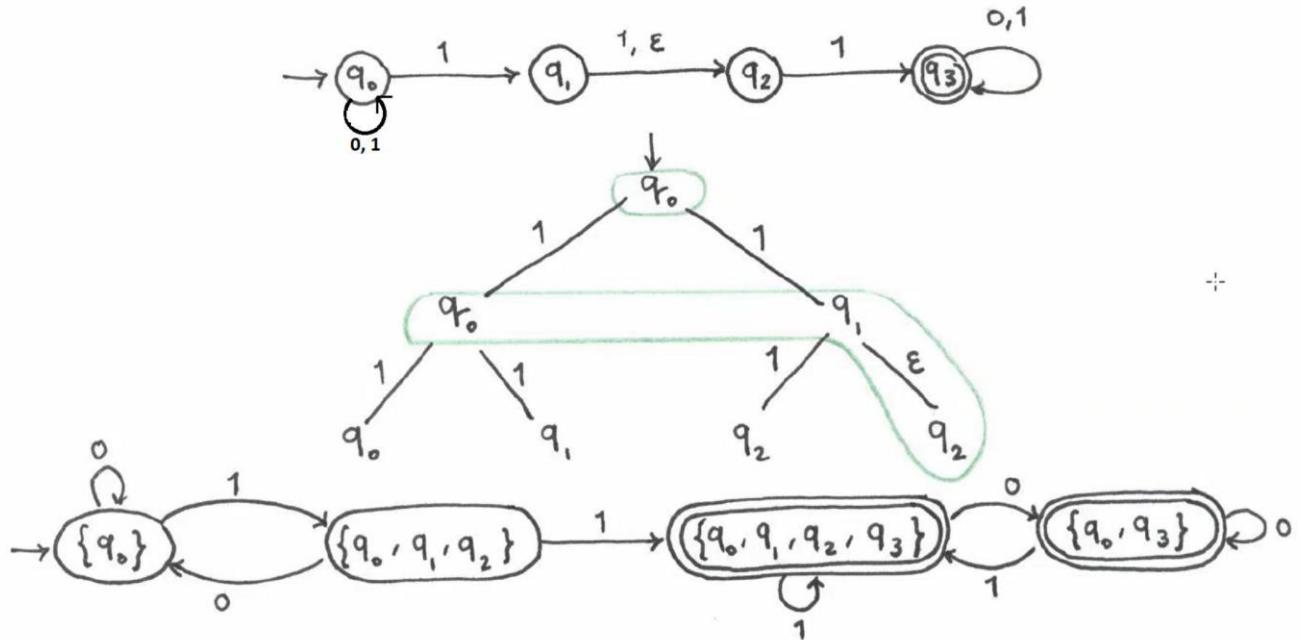


Equivalent deterministic finite automaton :



What if we have  $\epsilon$ -transitions?

(3)



$$E(R) = \{q \mid q \text{ can be reached from } R \text{ by travelling along 0 or more } \epsilon\text{-transitions}\}$$

$(R \subseteq Q)$

$$E(\{q_0\}) = \{q_1, q_2\}$$

After reading an input, follow  $\epsilon$ -transitions until you cannot anymore

$$\text{Input: } N = (Q, \Sigma, \delta, q_0, F)$$

$$\text{Output: } M = (Q', \Sigma, \delta', q'_0, F')$$

$$Q' = 2^Q$$

$$\delta': Q' \times \Sigma \rightarrow Q' \quad \delta'(R, a) = \bigcup_{r \in R} E(\delta(r, a))$$

$$q'_0 = E(\{q'_0\})$$

$$F' = \{R \in Q' \mid f \in R \text{ for some } f \in F\}$$

"Subset Construction"

Worst-case exponentially larger

"Finite Automata and their Decision Problems"

Michael O. Robin & Dana Scott 1959

①

حل نظری درس دیجیتال

حل نظری درس دیجیتال

## Subset Construction

Input:  $N = (Q, \Sigma, \delta, q_0, F)$ Output:  $M = (Q', \Sigma, \delta', q'_0, F')$ 

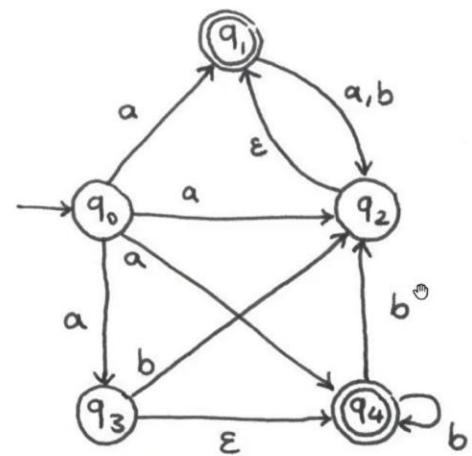
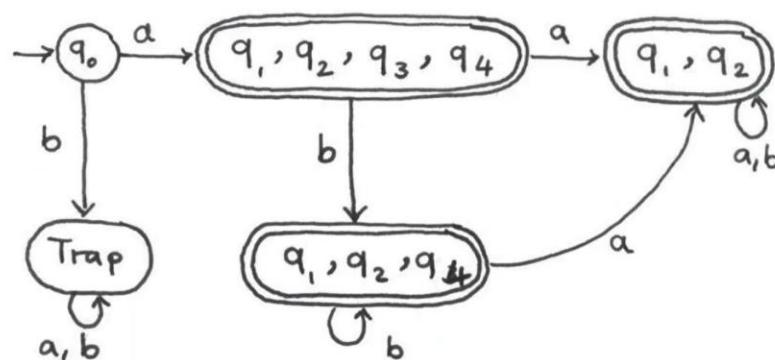
$$Q' = 2^Q$$

$$\delta': Q' \times \Sigma \rightarrow Q'$$

$$\delta'(R, a) = \bigcup_{r \in R} E(\delta(r, a))$$

$$q'_0 = E(\{q_0\})$$

$$F' = \{R \in Q' \mid f \in R \text{ for some } f \in F\}$$



Recall. A language is a set of strings.

We can define the usual set operations on languages.

Given two languages  $L_1$  and  $L_2$ \* Union  $L_1 \cup L_2 = \{w \mid w \in L_1 \vee w \in L_2\}$ \* Intersection  $L_1 \cap L_2 = \{w \mid w \in L_1 \wedge w \in L_2\}$ \* Complementation  $\overline{L}_1 = \{w \mid w \notin L_1\}$  ( $\overline{L}_1 = \Sigma^* - L_1$ )

We define three new operations on languages.

\* Reverse  $L_1^R = \{a_1 a_2 \dots a_k \mid a_k a_{k-1} \dots a_1 \in L_1\}$ \* Concatenation  $L_1 \cdot L_2 = \{vw \mid v \in L_1 \wedge w \in L_2\}$ \* Kleene star  $L_1^* = \{w_1 w_2 \dots w_k \mid k \geq 0 \text{ and each } w_i \in L_1\}$

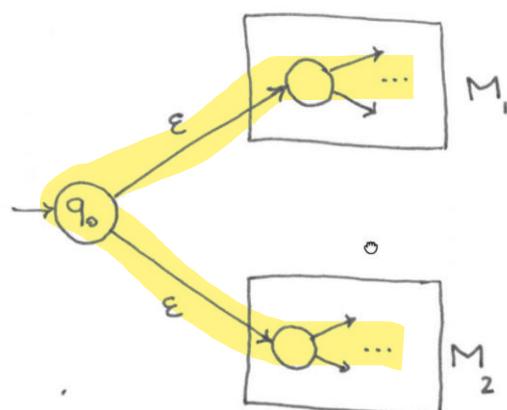
Recall. A language is called regular if it is accepted by a finite state automaton (2)

Definition. A set is closed under an operation if and only if the operation on any two elements of the set produces another element of the set.

example. Set  $\{-1, 1\}$  is closed under multiplication but not addition.

Theorem. Regular languages are closed under union, intersection, complementation, reversal, concatenation and star.

$L_1 \cup L_2$  :

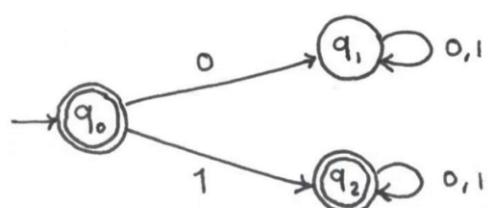
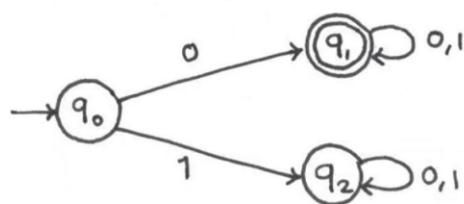


$\bar{L}$  (complementation)

Example:  $L_1 = \{0x \mid x \in \{0,1\}^*\}$  strings start with 0

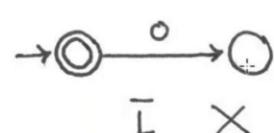
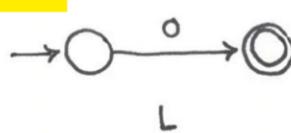
$\bar{L}_1 = \{1x \mid x \in \{0,1\}^*\} \cup \{\epsilon\}$  strings that do not start with 0

Complement of DFA: Swap accepting states and non-accepting states (and vice versa)



Note. Does not work for NFA

$$\Sigma = \{0\} \quad L = \{0\}$$



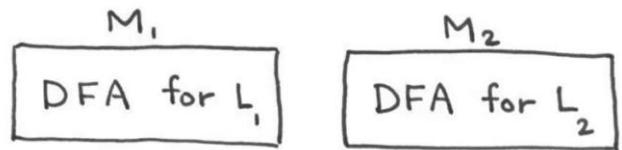
(3)

$$L_1 \cap L_2 :$$

Due to DeMorgan's law we already know regular languages are closed under intersection

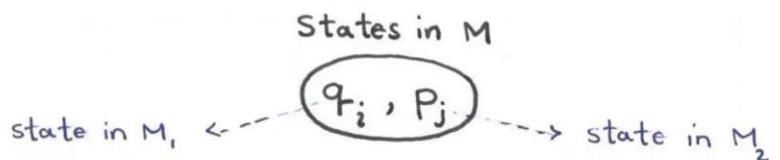
$$L_1 \cap L_2 = \overline{\overline{L}_1 \cup \overline{L}_2}$$

How to construct a new DFA  $M$  that recognizes the intersection of the languages of two DFAs?

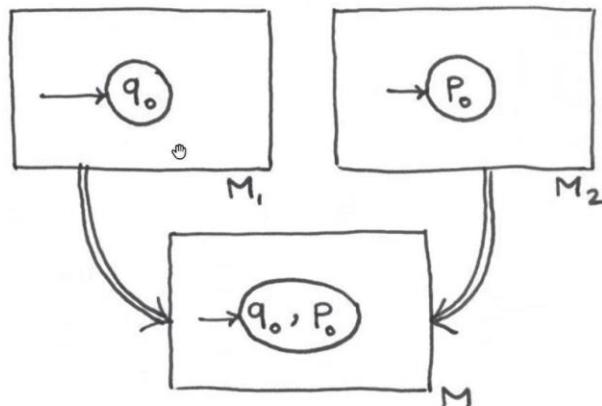


$M$  simulates in parallel

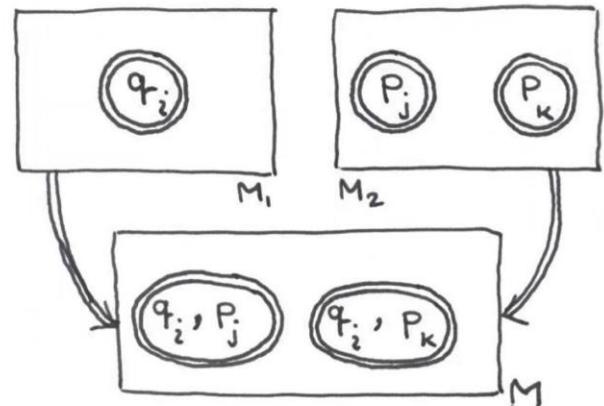
$M_1$  and  $M_2$



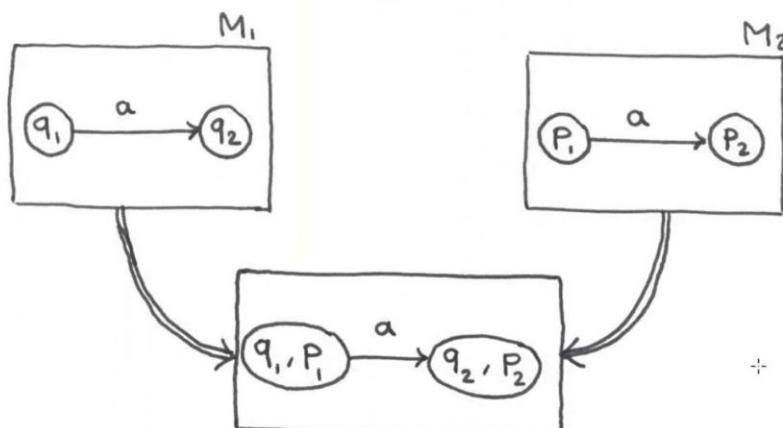
Initial State:



Final State:



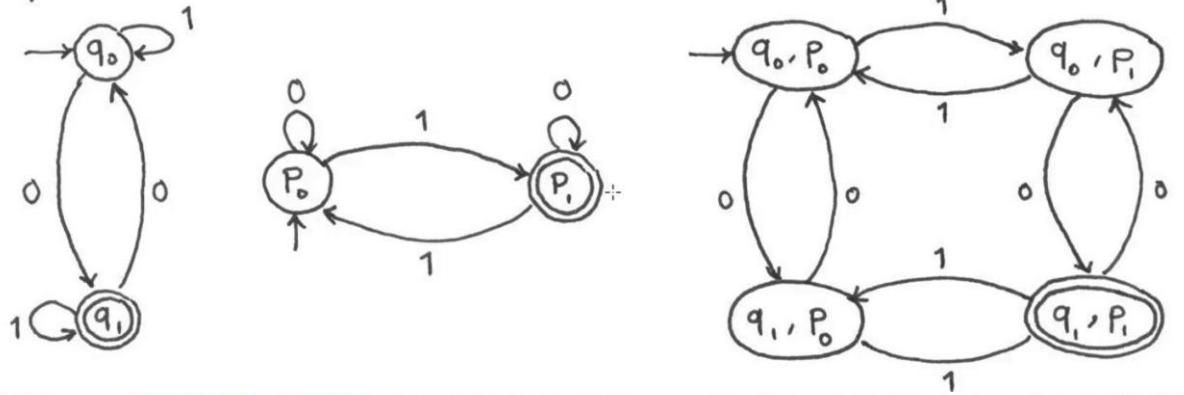
Transitions:



Cross-product DFA : a single DFA which simulates operation of two DFAs in parallel

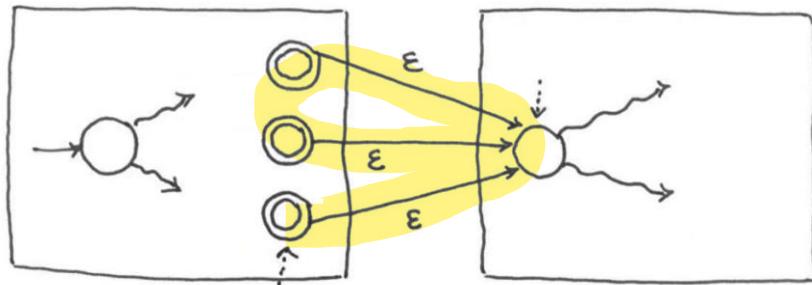
Example:

(4)



$L_1 \cdot L_2$

+



change to non-accepting state

$L^*$  All possible ways of concatenating any number of copies of strings in  $L$  together

$$L^* = \bigcup_{i=0}^{\infty} L^i = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \dots = \{\epsilon\} \cup L \cup LL \cup LLL \dots$$

Question. Can we say  $L^*$  is regular because it is the "union" of several regular languages?

$$L^0 = \{\epsilon\} \quad \text{regular}$$

$$L^1 = L \quad \text{regular}$$

$$L^2 = L \cdot L \quad \text{regular}$$

⋮ ⋮

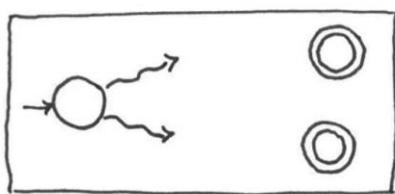
Answer. No. Because it is "infinite" union!

$\{0^n 1^n \mid n \in \mathbb{N}\}$  is not regular

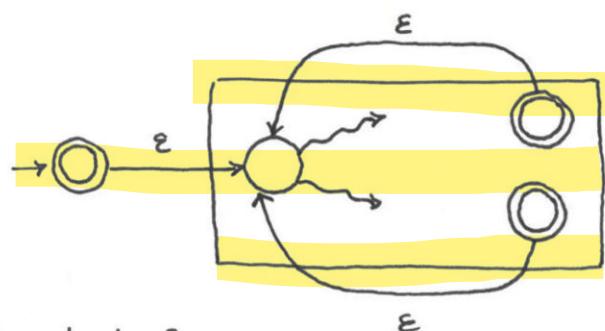
$\epsilon \cup \{01\} \cup \{0011\} \cup \dots$  Infinite union

Suppose  $L$  is regular. Then there exists a DFA that recognizes  $L$ .  
We construct an NFA that accepts  $L$ .

Make a new accepting start state, add  $\epsilon$ -transition  
to the old start state

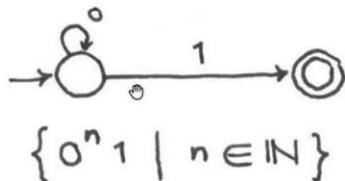


+

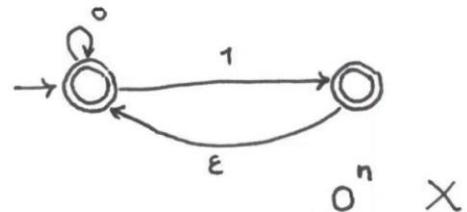


Why do we need the new start state?

Why can't we just change the start state to accepting?



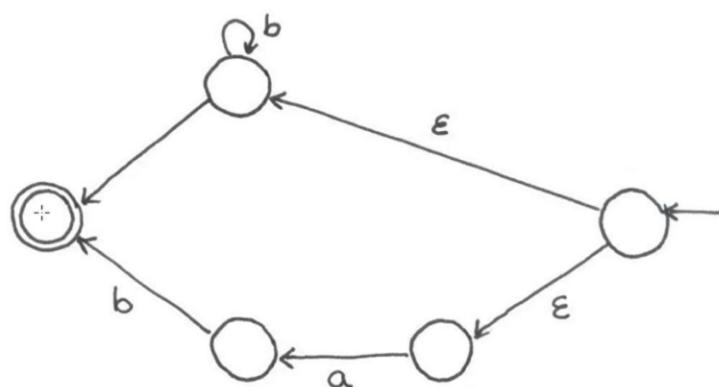
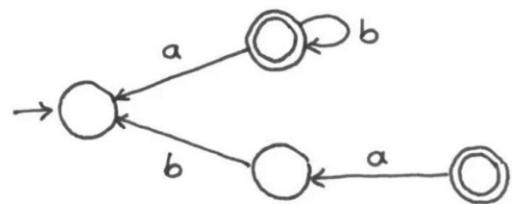
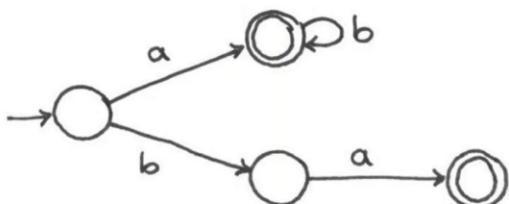
$$\{0^n 1 \mid n \in \mathbb{N}\}$$



$$0^n \times$$

$L^R$

- \* Reverse all the transitions
- \* Add a new start state, make  $\epsilon$ -transitions to every accepting states
- \* Turn the "old" initial state to accepting
- \* Turn the "old" accepting states to accepting



①

باعنوانی

نظری درس در ششم جلسه

DFA : Pictorial representation of a regular language

Regular Expression : Algebraic representation of a regular language

Regular expressions are used by many Unix utilities

rm \* .exe (del)  
\* .bak

Pattern Matching

folder: ( a.exe, b.bak, c.exe, abc.bak )

If  $R_1$  and  $R_2$  are regular expressions representing  $L_1$  and  $L_2$

| Regular Expression   | Regular Language |
|----------------------|------------------|
| $\emptyset$          | $\{\}$           |
| $a \ (a \in \Sigma)$ | $\{a\}$          |
| $\epsilon$           | $\{\epsilon\}$   |
| $R_1 + R_2$          | $L_1 \cup L_2$   |
| $R_1 R_2$            | $L_1 \cdot L_2$  |
| $R_1 *$              | $L_1^*$          |

Example . What does the following represent?

$$\Sigma = \{0, 1\}$$

|                 |                                               |
|-----------------|-----------------------------------------------|
| $(0+1)(0+1)$    | $\{w \mid  w =2\}$                            |
| $0*10*$         | $\{w \mid w \text{ contains a single } 1\}$   |
| $((0+1)(0+1))*$ | $\{w \mid  w  \text{ is even}\}$              |
| $(0+1)*1(0+1)*$ | $\{w \mid w \text{ has at least one } 1\}$    |
| $(0*10*1)*0*$   | $\{w \mid w \text{ has even number of } 1s\}$ |

Operator Precedence

- 1. \*
- 2. .
- 3. +

$$R_1 * R_2 + R_3 = ((R_1 * R_2) \cdot R_3) + R_3$$

$\{w \in \{0,1\}^* \mid w \text{ does not contain the substring } 10\}$  $0 * 1 *$  $\{w \in \{0,1\}^* \mid w \text{ contains } 00 \text{ as a substring}\}$  $(0+1)^* 00 (0+1)^*$  $\{w \in \{0,1\}^* \mid w \text{ does not contain the substring } 00\}$  $(01+1)^* (\epsilon+0)$ 

at most a single 0 between  
two consecutive 1s

ends in at most  
a single 0

 $\{w \in \{0,1\}^* \mid w \text{ has length } \geq 3 \text{ and its 3rd symbol is } 0\}$  $(0+1)(0+1)0(0+1)^*$  $\{w \in \{0,1\}^* \mid w \text{ contains at least three } 0s\}$  $(0+1)^* 0 (0+1)^* 0 (0+1)^* 0 (0+1)^*$  $\{w \in \{0,1\}^* \mid w \text{ contains at most one } 0\}$  $1 * (0+\epsilon) 1 *$  $1 * 0? 1 *$  $R?$ 

shorthand for  $(R+\epsilon)$ : zero or one copies of R

 $R+$ 

shorthand for  $RR^*$ : one or more copies of R

 $\{w \in \{0,1\}^* \mid \text{every run of } 0s \text{ in } w \text{ has length at least } 3\}$  $(1 + 0000^*)^*$  $\{w \in \{0,1\}^* \mid w \text{ does not end in } 00\}$  $\epsilon + 0 + 1 + (0+1)^* (01+10+11)$  $\{w \in \{0,1\}^* \mid |w| \text{ is odd}\}$  $(0+1)((0+1)(0+1))^*$

## Regular Expressions in Java

```
String re = "1*0?1*";  
String input = "11101111";  
System.out.println(input.matches(re));
```

Is input in the set described by re?

---

## Algebraic Laws for Regular Expressions

Union ( $\cup$ ) and concatenation ( $\cdot$ ) behave sort of like addition (+) and multiplication (\*)

- + is commutative and associative
- $\cdot$  is associative (not commutative)

Concatenation distributes over +

## Simplification of Expressions

Two regular expressions  $\alpha$  and  $\beta$  are equivalent  $\alpha \equiv \beta$  if  $L(\alpha) = L(\beta)$

If  $\alpha \equiv \beta$  we can substitute  $\alpha$  for  $\beta$  (or vice versa) in any regular expression and the resulting expression will be<sup>o</sup> equivalent to the original expression.

Note. Relation  $\equiv$  is an equivalence relation.

- \* Reflexive: for all  $\alpha$   $\alpha \equiv \alpha$
- \* Symmetric: for all  $\alpha, \beta$  if  $\alpha \equiv \beta$  then  $\beta \equiv \alpha$
- \* Transitive: for all  $\alpha, \beta, \gamma$  if  $\alpha \equiv \beta$  and  $\beta \equiv \gamma$  then  $\alpha \equiv \gamma$

Some laws that can be used to simplify regular expressions :

- |   |                                                              |                     |
|---|--------------------------------------------------------------|---------------------|
| ① | $\alpha + (\beta + \gamma) \equiv (\alpha + \beta) + \gamma$ | Associativity       |
| ② | $\alpha + \beta \equiv \beta + \alpha$                       | Commutativity       |
| ③ | $\alpha + \emptyset \equiv \alpha$                           | Identity            |
| ④ | $\alpha + \alpha \equiv \alpha$                              | Idempotent          |
| ⑤ | $\alpha(\beta\gamma) \equiv (\alpha\beta)\gamma$             | -                   |
| ⑥ | $\epsilon\alpha \equiv \alpha\epsilon \equiv \alpha$         | -                   |
| ⑦ | $\alpha(\beta + \gamma) \equiv \alpha\beta + \alpha\gamma$   | Left Distributivity |
| ⑧ | $(\alpha + \beta)\gamma \equiv \alpha\gamma + \beta\gamma$   | -                   |
| ⑨ | $\emptyset\alpha \equiv \alpha\emptyset \equiv \emptyset$    | -                   |
| ⑩ | $\epsilon + \alpha\alpha^* \equiv \alpha^*$                  | -                   |
| ⑪ | $\epsilon + \alpha^*\alpha \equiv \alpha^*$                  | -                   |

Let's prove ⑦ law :

$$w \in \alpha(\beta + \gamma)$$

$$\Leftrightarrow \exists u. \exists v. (w = uv) \wedge u \in L(\alpha) \wedge v \in L(\beta + \gamma)$$

$$\Leftrightarrow \exists u. \exists v. (w = uv) \wedge u \in L(\alpha) \wedge (v \in L(\beta) \vee v \in L(\gamma))$$

$$\Leftrightarrow \exists u. \exists v. (w = uv) \wedge ((u \in L(\alpha) \wedge v \in L(\beta)) \vee (u \in L(\beta) \wedge v \in L(\gamma)))$$

$$\Leftrightarrow \exists u. \exists v. (w = uv \wedge u \in L(\alpha) \wedge v \in L(\beta)) \vee (w = uv \wedge u \in L(\beta) \wedge v \in L(\gamma))$$

$$\Leftrightarrow w \in L(\alpha\beta) \quad \vee \quad w \in L(\beta\gamma)$$

$$\Leftrightarrow w \in L(\alpha\beta + \beta\gamma)$$

Prove :

$$(0110 + 01)(10)^* \equiv 01(10)^*$$

$$(0110 + 01)(10)^* = (0110 + 01\epsilon)(10)^* \quad \text{identity}$$

$$\equiv (01)(10 + \epsilon)(10)^* \quad \text{left distributivity}$$

$$\equiv (01)((10 + \epsilon)(10)^*) \quad \text{associativity}$$

$$\equiv (01)(10(10)^* + \epsilon(10)^*) \quad \text{right distributivity}$$

$$\equiv (01)(10(10)^* + (10)^*) \quad \text{identity}$$

$$\equiv (01)(10(10)^* + \epsilon + 10(10)^*) \quad \text{rule } \textcircled{10}$$

$$\equiv (01)(10(10)^* + 10(10)^* + \epsilon) \quad \text{commutativity}$$

$$\equiv (01)(10(10)^* + \epsilon) \quad \text{idempotent}$$

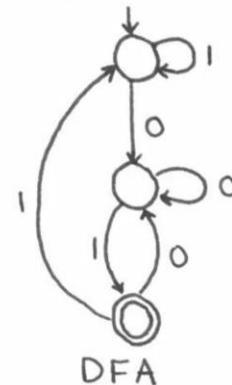
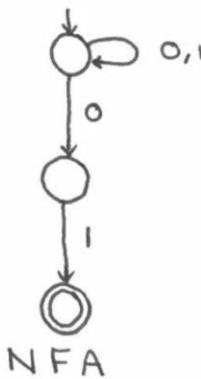
$$\equiv (01)(10)^* \quad \text{rule } \textcircled{10}$$

①

عربية

جامعة مكة المكرمة

$$L = \{x \in \{0,1\}^* \mid x \text{ ends in } 01\}$$



$(0+1)^* 01$   
Regular Expression

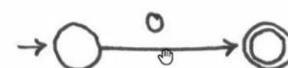
**Theorem.** A language is regular if and only if some regular expression describes it.

Proof has two parts:

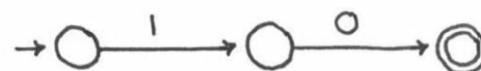
- ① If a language is described by some regular expression, then it is a regular language.
- ② If a language is regular, then it is described by some regular exp.

Example. regular expression  $\Rightarrow$  NFA

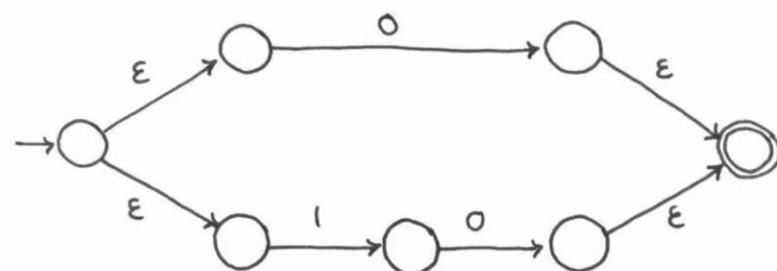
$$R_1 = 0$$



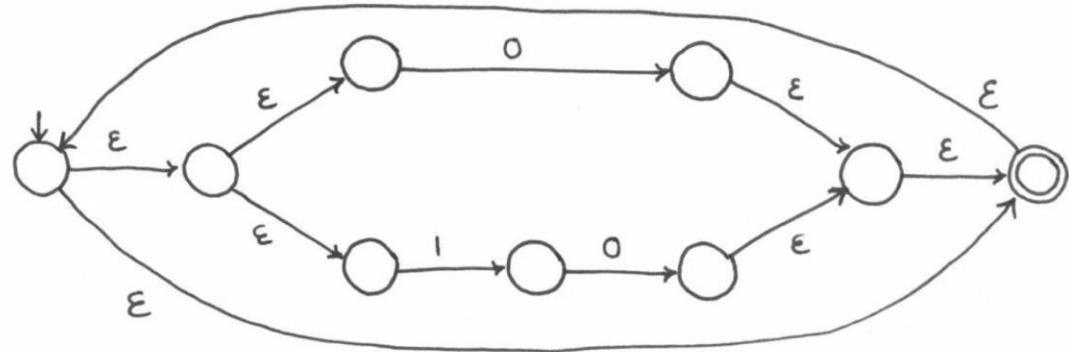
$$R_2 = 10$$



$$R_3 = 0 + 10$$

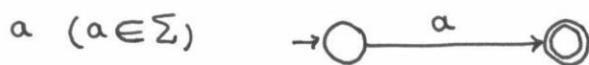


$$R_4 = (0+10)^*$$

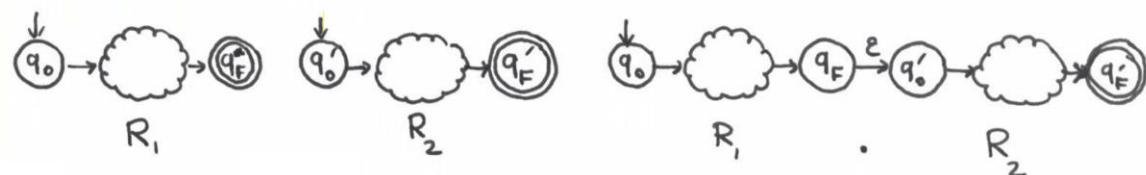


Thompson's Algorithm: an algorithm for converting any regular expression into an NFA. (2)

Base Cases.

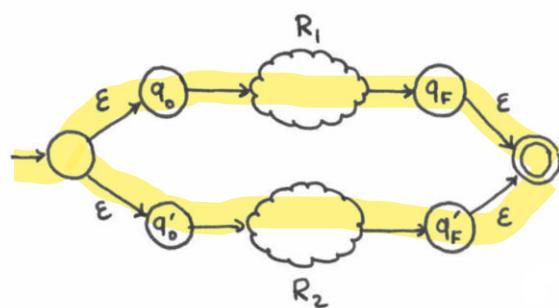
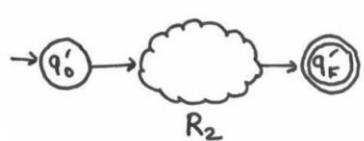
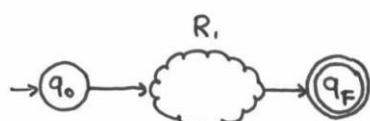


Concatenation  $R_1, R_2$



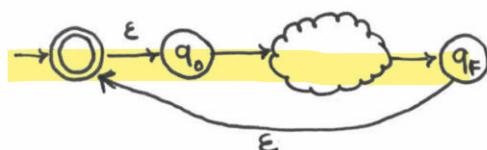
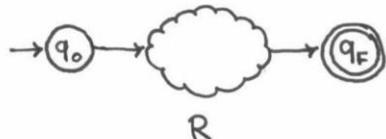
Union

$R_1 + R_2$

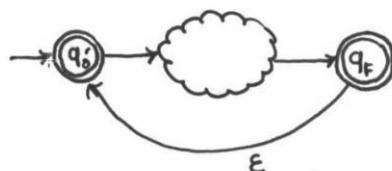


Kleene Star

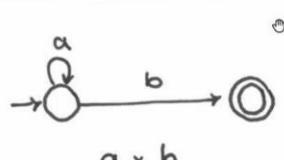
$R^*$



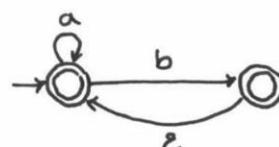
Why doesn't this construction work?



Consider:



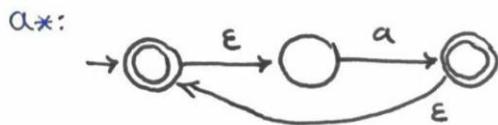
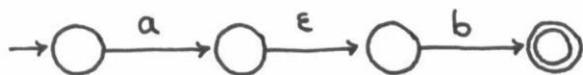
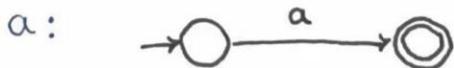
$a * b$



(3)

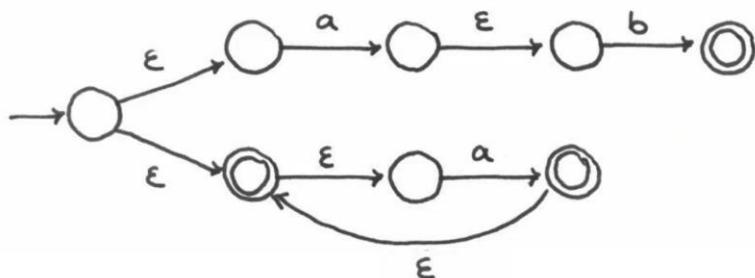
This is not  $(a * b)^*$  because it accepts  $ba$  which is not in  $(a * b)^*$

Example.  $L = ab + a^*$



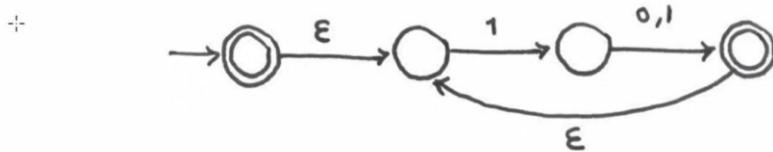
ab:

ab +  $a^*$



---

Example.  $L = (1(0+1))^*$



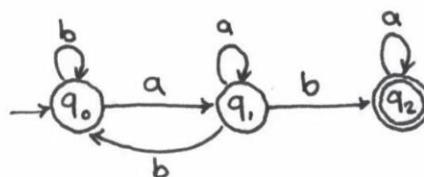
Why does this matter?

Many software tools work by matching regular expression against text

- Convert regular expression to NFA (Tompson's Algorithm)
- Convert NFA to DFA
- Run text through DFA

Theorem. If  $L$  is a regular language, then there is a regular expression  $R$  with  $L = L(R)$ .

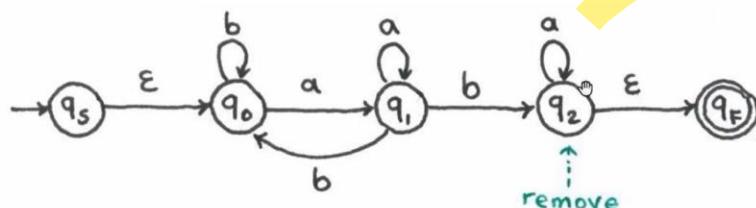
Example.



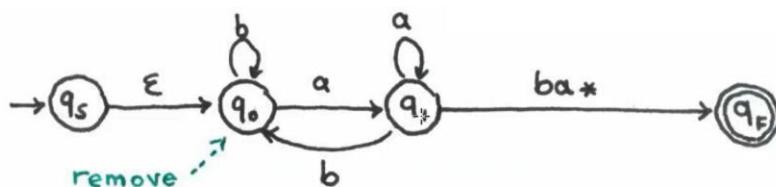
Convert NFA to a special form with : - only one final state

- no incoming edges to start

- no outgoing edges from final

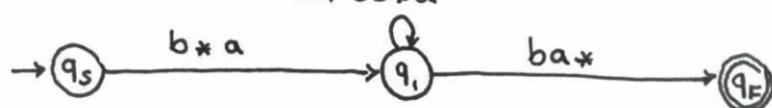


Now we remove states one at a time, replace labels with more complicated regular expression



$ba^*$ : all strings that can move automaton from  $q_1$  to  $q_F$

visiting just  $q_2$  any number of times  
 $a + bb^*a$



$a + bb^*a$ : all strings that can move automaton from  $q_1$  to  $q_F$ , visiting just  $q_0$  any number of times



Final label is the regular expression we are looking for it.

## Generalized NFA (gNFA)

Same as NFA, but :

- only one accept state  $\neq$  start state
- start state has no incoming edges, accept state no outgoing edges
- edges are labeled with regular expressions

First convert the original NFA to gNFA

Successively transform gNFA to equivalent gNFAs (recognizing same language) each time removing one state

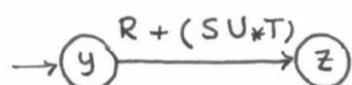
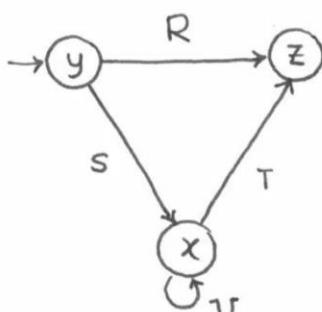


To remove a state  $x$ , consider every pair of states  $y$  and  $z$  (including  $y = z$ )

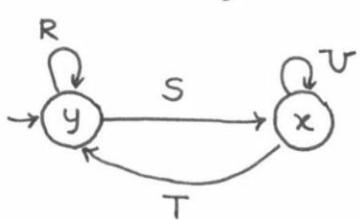
New label for edge  $(y, z)$  is the union of two expressions :

- { - what was there before
- one for paths through just  $x$

if  $y \neq z$



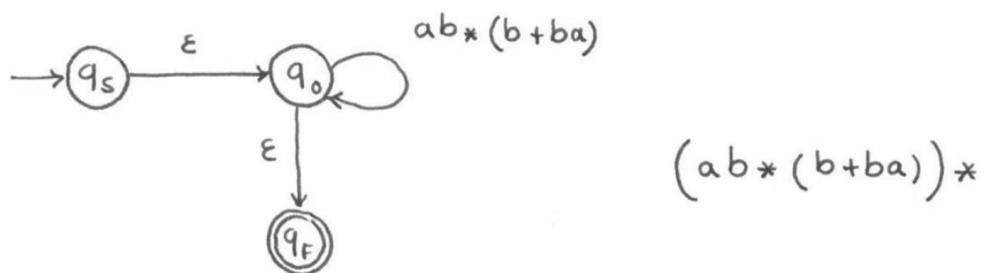
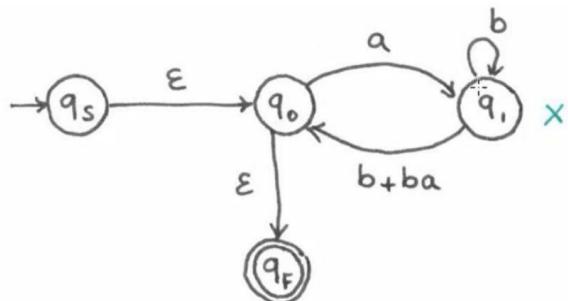
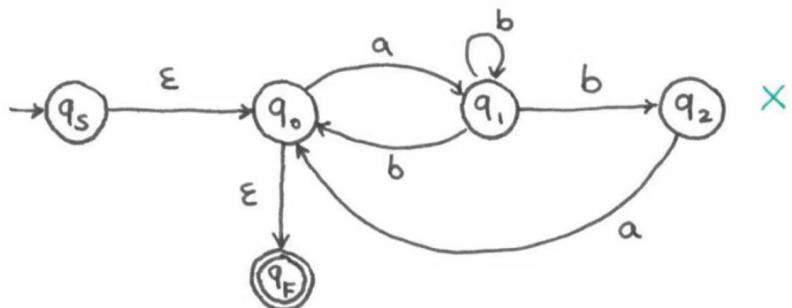
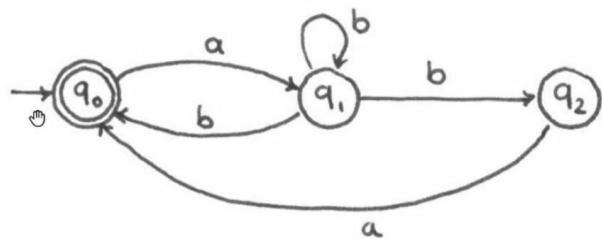
if  $y = z$



$$R + S \cup T_0$$



Example. Convert NFA to regular expression



Are these languages regular?

$$\{0^n 1^n \mid n > 0\}$$

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

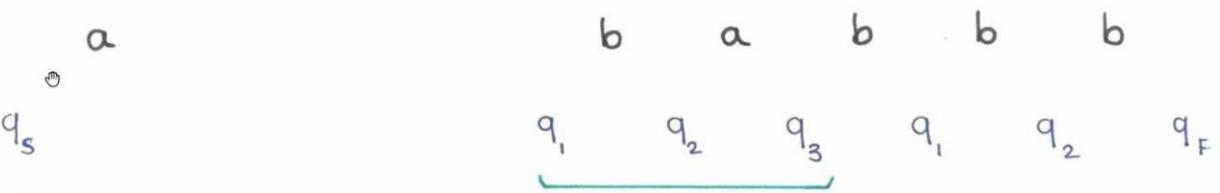
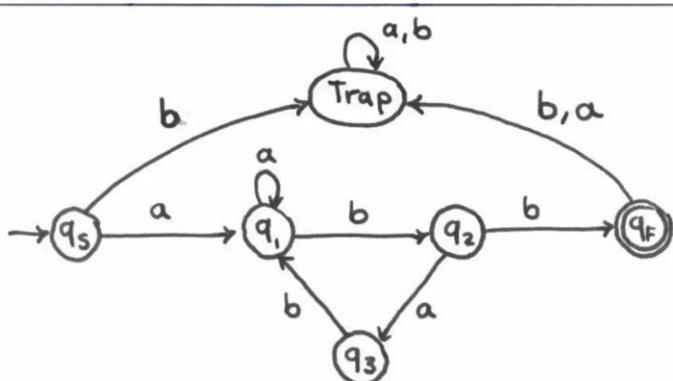
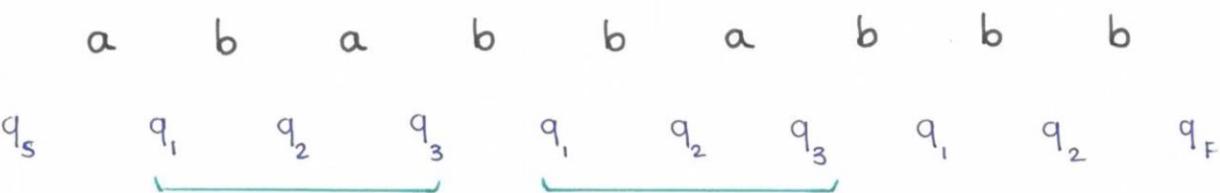
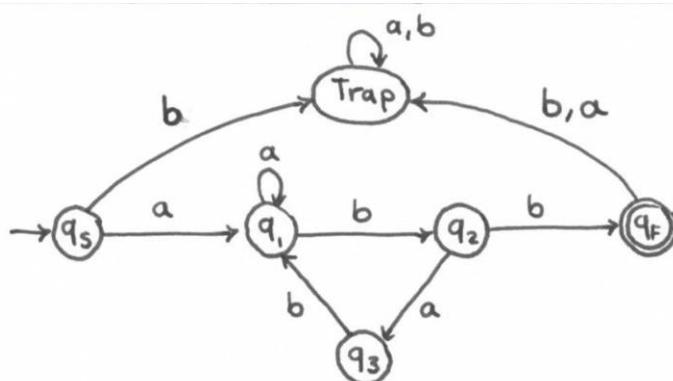
$$\{ww^R \mid w \in \{0,1\}^*\}$$

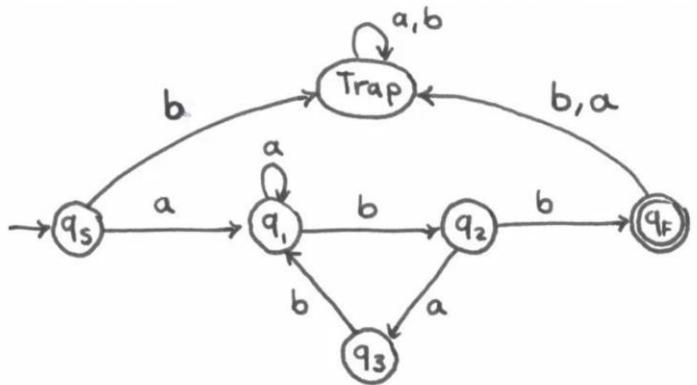
$$\{0^n 1^m \mid n > m\}$$

$$\{0^n 1^m \mid n \leq m\}$$

$$\{w \mid w \text{ contains equal number of } 0's \text{ and } 1's\}$$

$$\{w \mid w \text{ contains equal number of substrings } 01 \text{ and } 10\}$$





a

b b

 $q_S$  $q_1$  $q_2$  $q_F$ 

Let  $L$  be a regular language.

(2)

Consider a <sup>+</sup> string  $w \in L$  that is "sufficiently long"

We can split  $w$  into three pieces and "pump" middle

- Write  $w = xyz$

- Then  $xy^0z, xy^1z, xy^2z, xy^3z, \dots$  are all in  $L$

If  $L$  is regular then

$\exists$  integer  $p \geq 1$

$\forall$  string  $w \in L$  with  $|w| \geq p$

$\exists$  strings  $x, y, z$  satisfying  $w = xyz$  with

-  $|xy| \leq p$

-  $|y| > 0$

$\forall$  integer  $i \geq 0, xy^iz \in L$

$$w = \overbrace{a_1 a_2 a_3 \dots}^x \dots \underbrace{\dots}_{y} \dots a_p$$

DFA:  
 $p$  states

states  $\rightarrow q_0 | q_1 | q_2 | \dots q_i | \dots q_j | \dots q_p$

$$A \stackrel{A}{\Rightarrow} B \equiv \neg B \Rightarrow \neg A$$

( $\neg$  by definition of  $\Rightarrow$ )

$$\neg \forall x \in \mathbb{N}. A(x) \equiv \exists x \in \mathbb{N}. \neg A(x)$$

$$\neg \exists x \in \mathbb{N}. A(x) \equiv \forall x \in \mathbb{N}. \neg A(x)$$

If

$\forall$  integer  $p \geq 1$



Demon, for all  $\forall$  player

$\exists$  string  $w \in L$ , with  $|w| \geq p$



Demon picks  $p$

$\forall$  strings  $x, y, z$  s.t.  $w = xyz$



Demon picks  $x, y, z$

-  $|xy| \leq p$

-  $|y| > 0$  ( $y \neq \epsilon$ )

$\exists$  integer  $i \geq 0$ ,  $xy^i z \notin L$



you pick  $i$

then  $L$  is not regular

you win if  $xy^i z \notin L$

## PUMPING LEMMA.

(3)

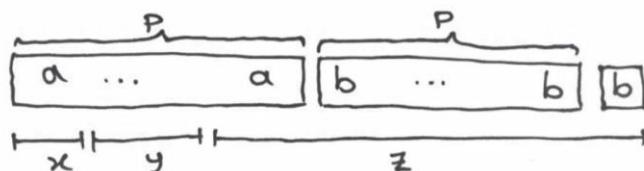
$L$  is not regular if you have a winning strategy.

$L = \{a^n b^n \mid n \geq 0\}$  is not regular

① Demon picks  $p \geq 1$

② You pick  $w = a^p b^p$   $w \in L$  and  $|w| \geq p$

③  $w = xyz$   $|xy| = p' \leq p$  and  $y \neq \epsilon$  +



$$x = a^l \quad y = a^j \quad z = a^{p-p'} b^{p+1}$$

$$(l+j) = p', \quad j \geq 1$$

④ Take  $i = 3$  then  $xy^i z = a^{p+j \times 2} b^{p+1}$

$$L_1 = \{a^n b^m \mid n \leq m\}$$

Proof by contradiction:

$$L_1^R = \{b^m a^n \mid n \leq m\} \quad \checkmark$$

$$L_2 = \{a^n b^m \mid n > m\} \quad \Sigma = \{a, b\} \quad \checkmark$$

$$L_1 \cap L_2 = \{a^n b^n \mid n \in \mathbb{N}\} \quad \times$$

$$L = \{w \in \{a, b\}^* \mid n_a(w) = n_b(w)\}$$

$$L \cap a^* b^* = \{a^n b^n \mid n \geq 0\} \quad (\text{Proof by contradiction})$$

$$L = \{a^n \mid n \text{ is a power of } 2\}$$

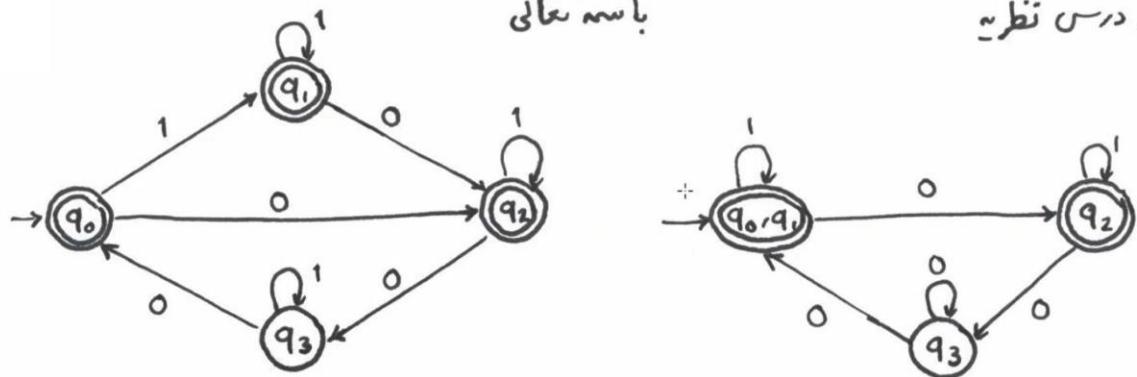
① Demon picks  $p \gg 1$

② You pick  $w = a^{2^p} \quad |w| = 2^p \geq p$

③  $w = xyz \quad x = a^l, \quad y = a^j, \quad z = a^k$

④ Take  $i=2$  then  $xy^2z = a^{2^p+j}$

$$2^p + j \leq 2^p + p < 2^p + 2^p = 2^{p+1}$$



حلقة دوارة ملاحظة  
لكل لغة منطقية L، توجد دالة معرفة مينيمال DFA  
تتحقق L (úniqueness up to relabeling of states)

ما هي المفاهيم؟

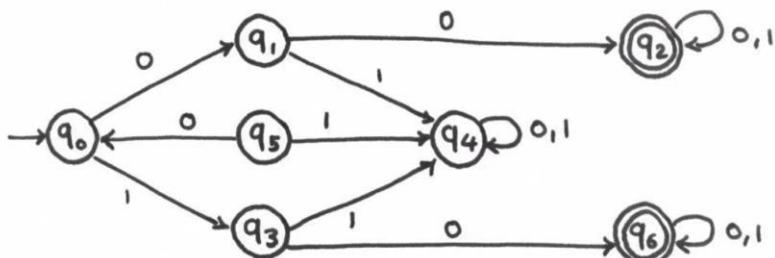
No unique!



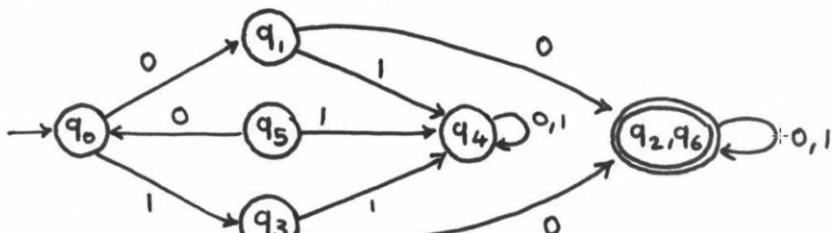
(Kameda-Weiner Algorithm)

لماذا يهم؟ استخدام أقل ذاكرة (hardware)

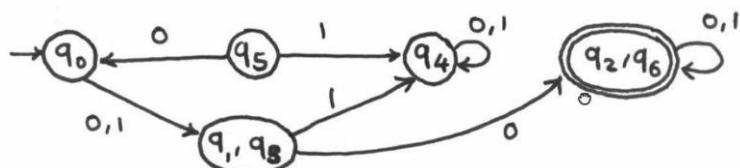
Idea : merge equivalent states



$q_2$  و  $q_6$  : أي سلاسل تصل إلىهما مأكولة في وقت لاحق. أي ملحوظات تليها تنتهي بنتائج متساوية



- إذا كنت في  $q_1$  أو  $q_3$  :
- ① إذا كان النهاية، رفض في كلتا الحالتين
  - ② إذا كان التالي 0، مأكولة في كلتا الحالتين.
  - ③ إذا كان التالي 1، رفض في كلتا الحالتين.

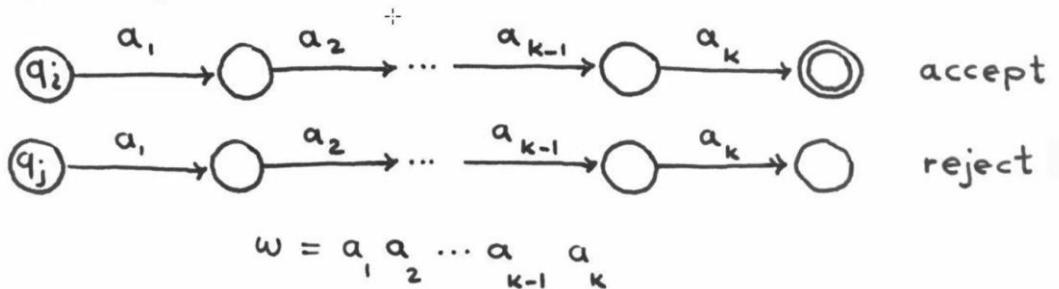


Unreachable states do not affect the accepted language

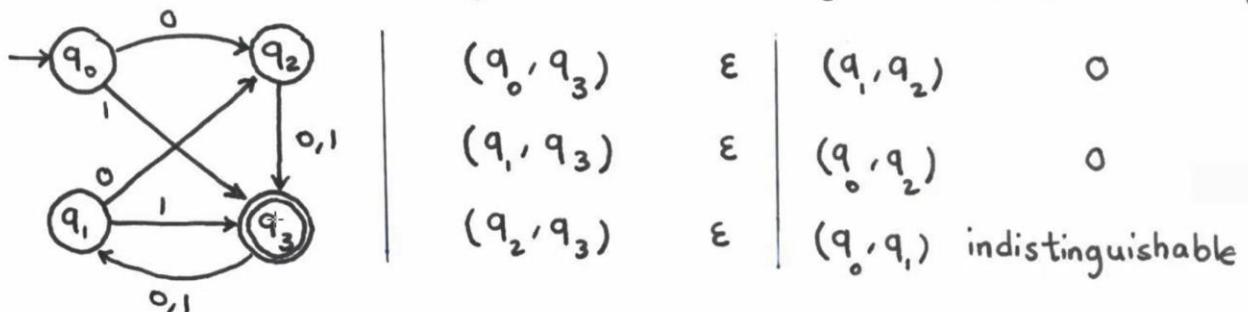


Intuition: two states are equivalent (indistinguishable) if all subsequent behavior from those states is the same.

Two states  $q_i$  and  $q_j$  are "distinguishable" if



Example. Which of the following pairs are distinguishable? by which string?



Two states  $q$  and  $q'$  in a DFA  $M = (Q, \Sigma, \delta, q_0, F)$  are equivalent if for all strings  $w \in \Sigma^*$ , the states on which  $w$  ends on when reading from  $q$  and  $q'$  are both accept or reject

\* Equivalent states can be merged together

DFA: an automaton is irreducible if:

- it contains no unreachable states
- no two distinct states are equivalent

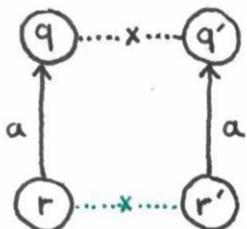
## Finding indistinguishable states

③

Phase 1: 

For each accepting  $q$  and rejecting  $q'$   
Mark  $(q, q')$  as distinguishable  $\times$

## Phase 2:



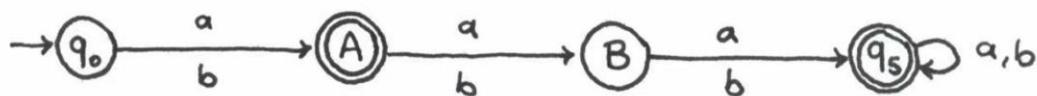
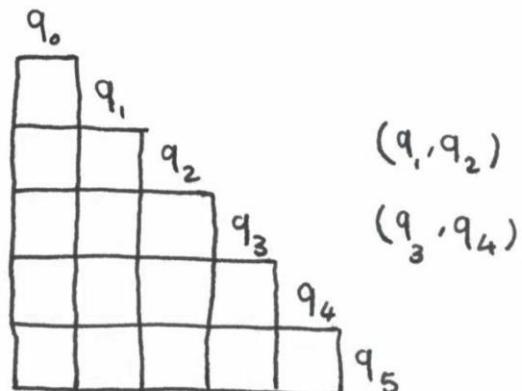
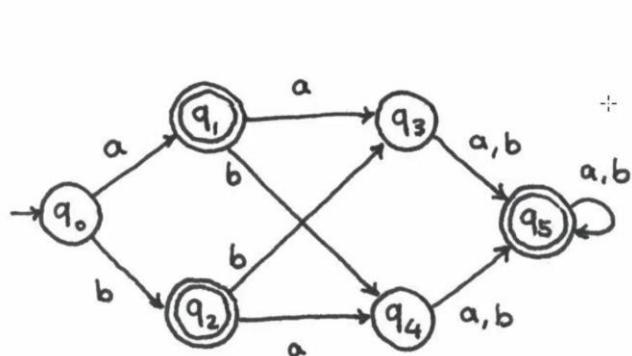
If  $(q, q')$  are marked and

$$r \xrightarrow{a} q \quad r' \xrightarrow{a} q'$$

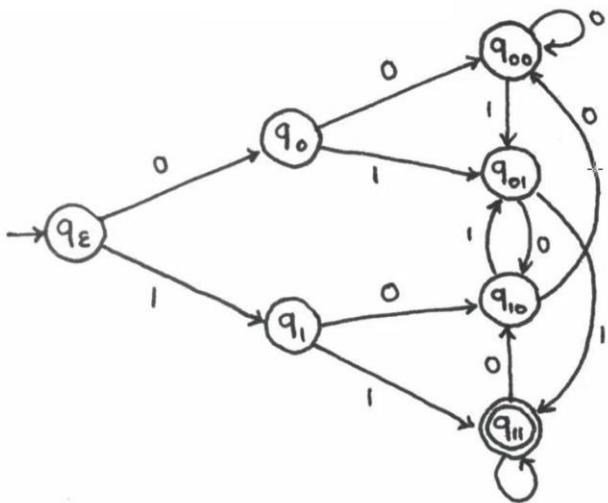
Mark  $(r, r')$  as distinguishable

Phase 3: Unmarked pairs are indistinguishable. Merge them!

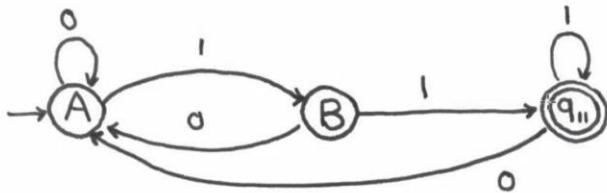
We set up a table in which we can keep track of all pairs of states



|       |   |       |       |       |       |       |
|-------|---|-------|-------|-------|-------|-------|
| $q_0$ |   |       |       |       |       |       |
| X     |   | $q_1$ |       |       |       |       |
| X     | A |       | $q_2$ |       |       |       |
| X     | X | X     |       | $q_3$ |       |       |
| X     | X | X     | B     |       | $q_4$ |       |
| X     | X | X     | X     | X     |       | $q_5$ |



| $q_E$ | $q_0$ | $q_1$ | $q_{01}$ | $q_{10}$ | $q_{11}$ |
|-------|-------|-------|----------|----------|----------|
| A     | $q_0$ |       |          |          |          |
| X     | X     | $q_1$ |          |          |          |
| A     | A     | X     | $q_{01}$ |          |          |
| X     | X     | B     | X        | $q_{10}$ |          |
| A     | A     | X     | A        | X        | $q_{11}$ |
| X     | X     | X     | X        | X        | X        |



## Decision Problems about Regular Languages

Remember: A decision problem is a problem with yes/no answer

- Given a string  $w$  and a language  $L$ , determine whether  $w \in L$

Assume  $L$  is represented by a DFA  $M$ . Simulate the action of  $M$  on the sequence of symbols of  $w$ .

- Given a language  $L$ , determine whether it is empty.

Compute the set of reachable states from the start state of the DFA. Check if any final state is reachable.

- Given a language  $L$ , determine whether it is finite.

Compute all the states that belong to some cycle in the DFA.

Determine if any of them is on a path from an initial to final state.

- Given  $L_1$  and  $L_2$  determine if they are identical.

Minimize the DFAs representing the language  $L_1$  and  $L_2$ .

Check if they are isomorphic.

We used pumping lemma to prove  $L$  is not regular:

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

$$w = a^n b^n = a \boxed{a^{n-1} b^{n-1}} b \quad \text{recursive structure}$$

Basic Idea: use "variables" to stand for sets of strings

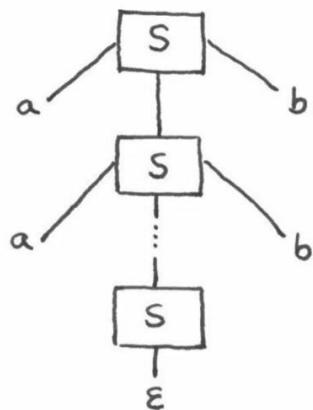
Variables are defined recursively in terms of one another

Recursive rules ("productions") involve only concatenation

Alternative rules for a variable allow union

$$\begin{cases} S \rightarrow aSb \\ S \rightarrow \epsilon \end{cases}$$

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aa\epsilon bb \\ = aabb$$



Terminals = symbols of the alphabet of the language

Variables = non-terminals = a finite set of other symbols,  
each of which represents a language

Start symbol = the variable whose language is the one being defined

### Grammar:

Set of rules capable of generating all the words of a language  
(Chomsky ~ 1950)

Left  $\rightarrow$  Right

+

Anything matches on left-side can be replaced by what is on the right. Left and Right can be any sequence of variables (non-terminals) and symbols (terminals)

(2)

$$Exp \rightarrow Var \mid Num \mid (Exp)$$

$$Exp \rightarrow Exp + Exp$$

$$Exp \rightarrow Exp * Exp$$

$$Var \rightarrow x \mid y \mid z$$

$$Num \rightarrow 0 \mid \dots \mid 9$$

- Start symbol: Exp

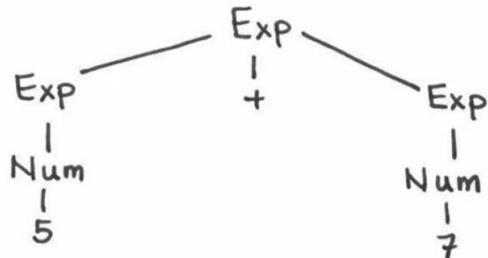
- Finite set of variables:

$$\{Exp, Var, Num\}$$

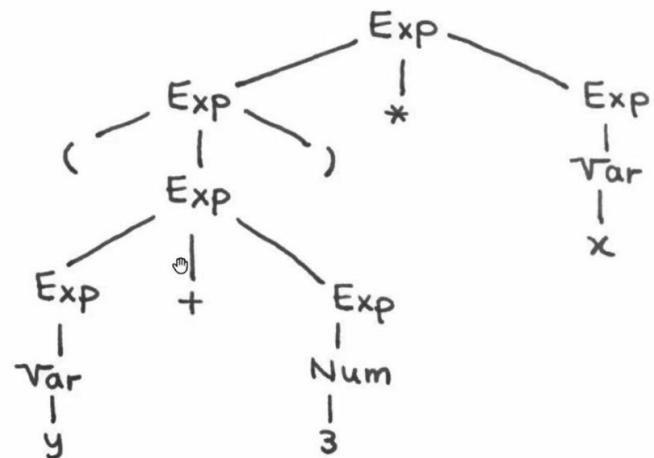
- Finite set of terminals:

$$\{x, y, z, 0, \dots, 9, (, )\}$$

- Finite set of rules



$$5 + 7$$



$$(y + 3) * x$$

- Interior nodes:

variable

- Leaves:

terminals

- Yield:

strings of terminals at the bottom

When designing grammars:

③

- Think recursively: Build up bigger structures from smaller ones

- Have construction plan:

    Know in what order you build up string

- Store information in non-terminals

    Have each non-terminal correspond to some useful information

$$L = \{ w \in \{0,1\}^* \mid w \text{ contains at least three } 1 \}$$

③



$$S \rightarrow X 1 X 1 X 1 X$$

$$X \rightarrow 0 X \mid 1 X \mid \epsilon$$

$$L = \{ w \in \{0,1\}^* \mid \text{length of } w \text{ is odd and middle symbol is } 0 \}$$



First Attempt:

$$\begin{cases} S \rightarrow X 0 X \\ X \rightarrow 0 X \mid 1 X \mid \epsilon \end{cases} \quad \text{wrong!}$$

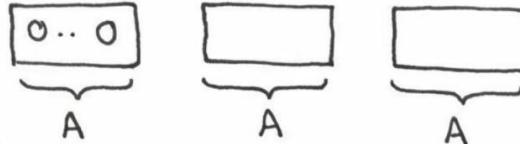
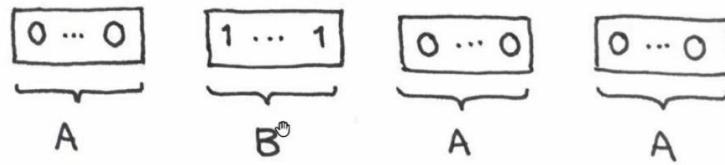
$$S \Rightarrow X 0 X \Rightarrow \underbrace{0 X}_\text{X} \underbrace{0 X}_\text{X} \Rightarrow 0 \epsilon 0 X \Rightarrow 00 X \Rightarrow 00 \epsilon \Rightarrow 00$$

Second Attempt:



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

$$L = \{ 0^n 1^m 0^{2n} \mid n, m \geq 0 \}$$



$$\left\{ \begin{array}{l} S \rightarrow 0 S 00 \\ \mid \epsilon \end{array} \right.$$

$$\begin{cases} S \rightarrow 0 S 00 \mid X \\ X \rightarrow 1 X \mid \epsilon \end{cases}$$

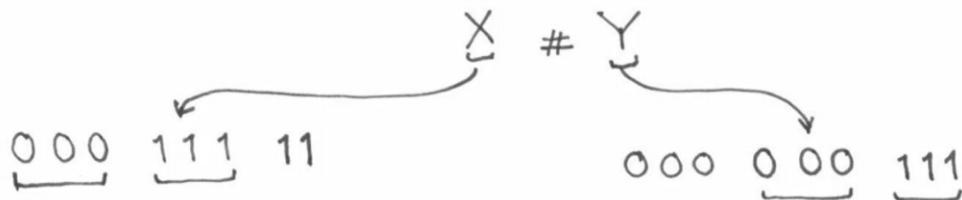
(4)

$$L = \left\{ \omega \in \{0,1\}^* \mid \omega \text{ starts and ends with same symbol} \right\}$$

$$S \rightarrow 0 \times 0 \mid 1 \times 1 \mid \epsilon$$

$$X \rightarrow 0 X \mid 1 X \mid \epsilon$$

$$L = \left\{ 0^i 1^j \# 0^k 1^l \mid i < j \text{ and } k > l \right\}$$



$$S \rightarrow X \# Y$$

$$E \rightarrow a E b \mid \epsilon$$

$$X \rightarrow X 1 \mid E 1$$

$$Y \rightarrow 0 Y \mid 0 E$$

$$L = \left\{ \omega \in \{a,b\}^* \mid \omega = \omega^R \right\}$$

$$S \rightarrow a S a \mid b S b \mid \epsilon \mid a \mid b$$

$$L = \left\{ \omega \in \{a,b\}^* \mid \omega \text{ contains more a's than b's} \right\}$$

$$S \rightarrow T a T$$

$$T \rightarrow a T \mid a T b T \mid b T a T \mid \epsilon$$

## Types of Grammars:

0) Unrestricted Grammars. no restrictions on rewriting rules

1) Context sensitive Grammars.  $\alpha A \beta \rightarrow \alpha \gamma \beta$

A non-terminal,  $\alpha, \beta, \gamma$  strings of terminals and non-terminals  
 $\gamma$  is non-empty

2) Context free Grammars  $A \rightarrow \gamma$ .

$\gamma$  potentially empty string of terminals and non-terminals

3) Regular Grammars

$$A \rightarrow aB \mid \alpha$$

A regular grammar is either right- or left- linear.

Note. If we mix left-linear and right-linear rules the result is not necessarily regular.

$$S \rightarrow aA \mid \epsilon \quad (\text{Right Linear})$$

$$A \rightarrow Sb \quad (\text{Left Linear})$$

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

①

باعثی

خطہ بارڈم مدرس

Context-free Grammar

$$G = (\mathcal{V}, \Sigma, R, S)$$

 $\mathcal{V}$ : finite set (variables) $\Sigma$ : finite set disjoint from  $\mathcal{V}$  (terminals) $R$ : finite non-empty set of rules $S \in \mathcal{V}$  start variableExample  $G = (\mathcal{V}, \Sigma, R, S)$ 

$$\mathcal{V} = \{S\}$$

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

$$R = \{S \rightarrow aSb, S \rightarrow \epsilon\}$$

Start Symbol:  $S$ 

$$L(G) = \{a^n b^n \mid n \geq 0\}$$

Show that a string  $w$  belongs to the language of grammar  $G$ 

- Start with start symbol  $S$
- Repeatedly replace one of the non-terminals by RHS of a rule
- Stop when sentence contains only terminals

Derivation Step     $A \in \mathcal{V}, \alpha, \beta, \gamma \in (\mathcal{V} \cup \Sigma)^*$      $A \rightarrow \gamma \in R$

 $\alpha A \beta \Rightarrow \alpha \gamma \beta$     derivation step

Derivation    Sequence of derivation steps

$$\gamma_0 \Rightarrow \gamma_1 \Rightarrow \dots \Rightarrow \gamma_n \quad (n \geq 0) \quad \gamma_i \in (\mathcal{V} \cup \Sigma)^*$$

If there is a derivation from  $\gamma_0$  to  $\gamma_n$  we write it as  $\gamma_0 \xrightarrow{*} \gamma_n$

Example.     $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aa\epsilon bb \Rightarrow aabb$

$$S \xrightarrow{*} aabb$$

Language of grammar     $L(G) = \{w \in \Sigma^* \mid S \xrightarrow{*} w\}$

Set of strings derivable from start    (Context-Free Languages)

How do we know which rule to apply on every step? ②

- Leftmost derivation: always expand leftmost variable
- Rightmost derivation: always expand rightmost variable

Example.  $G = (\{S\}, \{((), )\}, R, S)$        $R = \{ S \xrightarrow{+} SS \mid (S) \mid () \}$

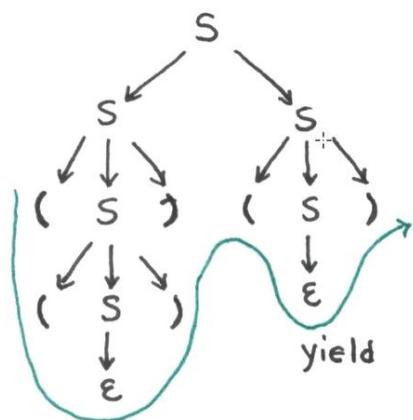
①      ②      ③

Derivation:  $S \xrightarrow{1} SS \xrightarrow{2} (S)S \xrightarrow{3} (S)() \xrightarrow{3} ((())()$

Leftmost Derivation:  $S \xrightarrow{1} SS \xrightarrow{2} (S)S \xrightarrow{3} ((())S \xrightarrow{3} ((())()$

Rightmost Derivation:  $S \xrightarrow{1} SS \xrightarrow{3} S() \xrightarrow{2} (S)() \xrightarrow{3} ((())()$

Leftmost & Rightmost: allow us to describe derivation by listing sequence of rules

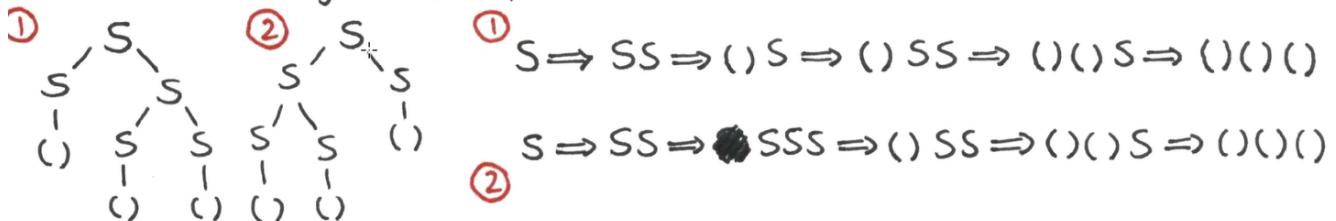


Parse (Derivation) tree shows derivation of a string as tree

- a) root is labelled  $S$
- b) every leaf has a label from  $\Sigma \cup \{\epsilon\}$
- c) every interior has a label from  $V$
- d) if an interior  $A \in V$  has children  $B_1, \dots, B_n$  then  $R$  must have  $A \rightarrow B_1, \dots, B_n$  ( $B_i \in V \cup \Sigma$ )

A given CFG  $G$  is ambiguous if a string  $w \in L(G)$  is the yield of two different parse trees.

Equivalently:  $G$  is ambiguous if a string  $w \in L(G)$  has two different leftmost or rightmost derivations.



Ambiguity is a property of a grammar, not the language it generates

$$S \rightarrow \epsilon \mid (S) S \quad (\text{unambiguous grammar})$$

(3)

There is **no algorithm** to detect if an arbitrary grammar is ambiguous

$$S \rightarrow S + S \mid S * S \mid 0 \mid \dots \mid 9$$

$$1 + 2 * 3$$



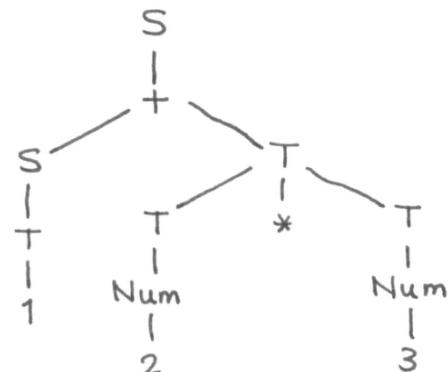
Importance: Compiler does not only say yes/no - has to give semantics

Does this grammar solve the problem?

$$S \rightarrow (S + S) \mid (S * S) \mid 0 \mid \dots \mid 9$$

$$\left\{ \begin{array}{l} S \rightarrow S + T \mid T \\ T \rightarrow T * \text{Num} \mid \text{Num} \\ \text{Num} \rightarrow 0 \mid \dots \mid 9 \end{array} \right.$$

$$\boxed{x * y * z} + \boxed{a * b} + \boxed{d}$$



Is every CFL recognizable by a grammar that has only one variable?

No. Consider the language  $L = \{0\} \cup \{\epsilon, 1, 11, 111, \dots\}$

$$(0 + 1*)$$

Assume  $G$  has only one non-terminal  $S$

Since the grammar has only finite number of rules, there exists some  $n$  such that  $1^n \in L$  and it cannot be generated in one step  $S \xrightarrow{*} 1^n$

$$S \xrightarrow{*} \alpha S \beta \xrightarrow{*} 1^n \quad \alpha \text{ and } \beta \text{ are not both empty}$$

$$S \xrightarrow{*} 1^a S 1^b \quad \text{replace all } S \text{ with } 1 \quad (a, b \text{ are not both } 0)$$

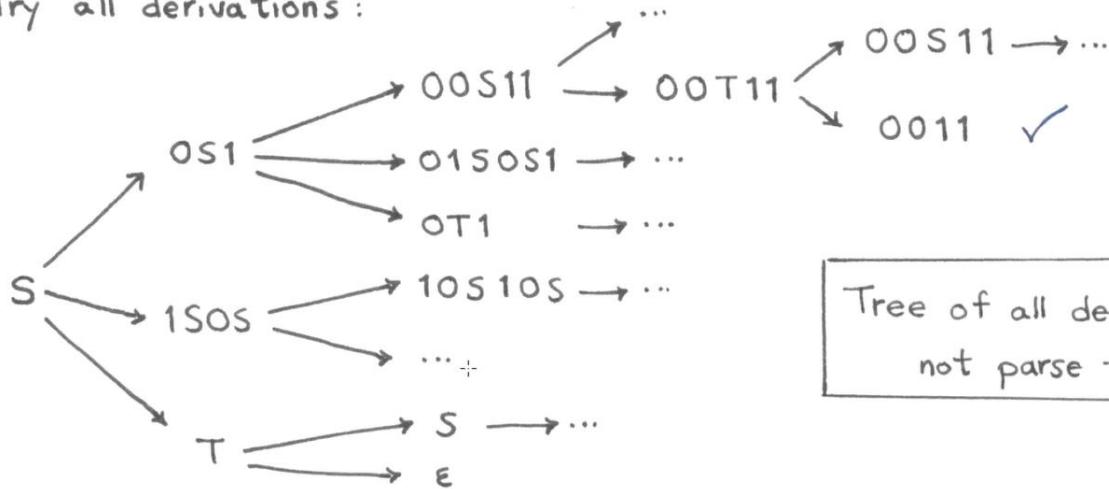
$$S \xrightarrow{*} 1^a 0 1^b \quad X$$

1)

$$\left\{ \begin{array}{l} S \rightarrow OS1 \mid 1SOS \mid T \\ T \rightarrow S \mid \epsilon \end{array} \right.$$

Is  $0011 \in L$ ? If so, how to build a parse tree?

Try all derivations:



Tree of all derivations,  
not parse tree

Problems.

- 1) Trying all derivations may take too long
- 2) If input is not in the language, parsing will never stop

Let's tackle the second problem

Idea: Stop when  $| \text{derived string} | > | \text{input} |$

Problems:

- \* Derived string may shrink because of  $\epsilon$ -productions

$$S \Rightarrow OS1 \Rightarrow OT1 \Rightarrow 01$$

- \* Derivation may loop because of unit-productions

$$S \Rightarrow T \Rightarrow S \Rightarrow T \Rightarrow S \Rightarrow \dots$$

→ We need to remove  $\epsilon$  and unit productions.

- \* Eliminating  $\epsilon$ -productions

Given a grammar  $G$ , produce an equivalent grammar  $G'$  ( $L(G) = L(G')$ ) such that  $G'$  does not have any rule of the form  $A \rightarrow \epsilon$  except possibly  $S \rightarrow \epsilon$ , and  $S$  does not appear on the right hand side of any rule. (Note. If  $S$  can appear on the RHS of a rule, say  $S \rightarrow SS$  then when there is a rule  $S \rightarrow \epsilon$  we can again have long intermediate strings yielding short final strings)

1) Add a new start symbol  $S'$  and the rule  $S' \rightarrow S$  where ②  
 $S$  was the original start symbol - this guarantees the new  
start symbol is not on the RHS of any rule

2) For every rule  $A \rightarrow \epsilon$  where  $A$  is not the (new) start var

a) Remove the rule  $A \rightarrow \epsilon$

b) If you see  $C \rightarrow \alpha A \beta$  add a new rule  $C \rightarrow \alpha \beta$

This must be done for each occurrence of  $A$

$C \rightarrow \alpha A \beta A \gamma$  becomes  $C \rightarrow \alpha \beta A \gamma \mid \alpha A \beta \gamma \mid \alpha \beta \gamma$

$C \rightarrow A$  becomes  $C \rightarrow \epsilon$

If  $C \rightarrow \epsilon$  was removed earlier, don't add it back

$$S' \rightarrow S$$

$$S \rightarrow ABCD \mid E$$

$$\boxed{S \rightarrow E}$$

$$\boxed{B \rightarrow E}$$

$$\boxed{C \rightarrow E}$$

$$\boxed{D \rightarrow E}$$

$$A \rightarrow SaS$$

$$S' \rightarrow \epsilon$$

$$S \rightarrow ACD$$

$$S \rightarrow ABD$$

$$S \rightarrow ABC$$

$$B \rightarrow \epsilon$$

$$C \rightarrow CE$$

$$D \rightarrow C$$

$$C \rightarrow SE$$

$$S \rightarrow AC$$

$$C \rightarrow CSE \mid \epsilon$$

$$A \rightarrow Sa$$

$$E \rightarrow a$$

$$D \rightarrow B$$

$$S \rightarrow AB$$

$$D \rightarrow CB \mid b$$

$$A \rightarrow as$$

$$C \rightarrow E$$

$$S \rightarrow A$$

$$E \rightarrow ab$$

$$A \rightarrow a$$

$$S \rightarrow AD$$

$$D \rightarrow \epsilon$$

A unit production is a production of the form  $A \rightarrow B$

Grammar:

Unit production Graph

$$S \rightarrow OS1 \mid 1SOS \mid T$$

$$T \rightarrow S \mid R \mid \epsilon$$

$$R \rightarrow O SR$$



① If there is a cycle of unit productions  $A \rightarrow B \rightarrow \dots \rightarrow C \rightarrow A$   
delete it and replace everything with  $A$  (any var in cycle)

$$S \rightarrow OS1 \mid 1SOS$$

$$S \rightarrow R \mid \epsilon$$

$$R \rightarrow O SR$$

② replace any chain

$A \rightarrow B \rightarrow \dots \rightarrow C \rightarrow \alpha$

③

by  $A \rightarrow \alpha, B \rightarrow \alpha, \dots, C \rightarrow \alpha$

$$S \rightarrow OS1 \mid 1SOS$$

$$| R | \epsilon$$

$$R \rightarrow OSR$$

S  
↓  
R

$$S \rightarrow OS1 \mid 1SOS$$

$$| OSR | \epsilon$$

$$R \rightarrow OSR$$

Problems. 1) Trying all derivations may take too long

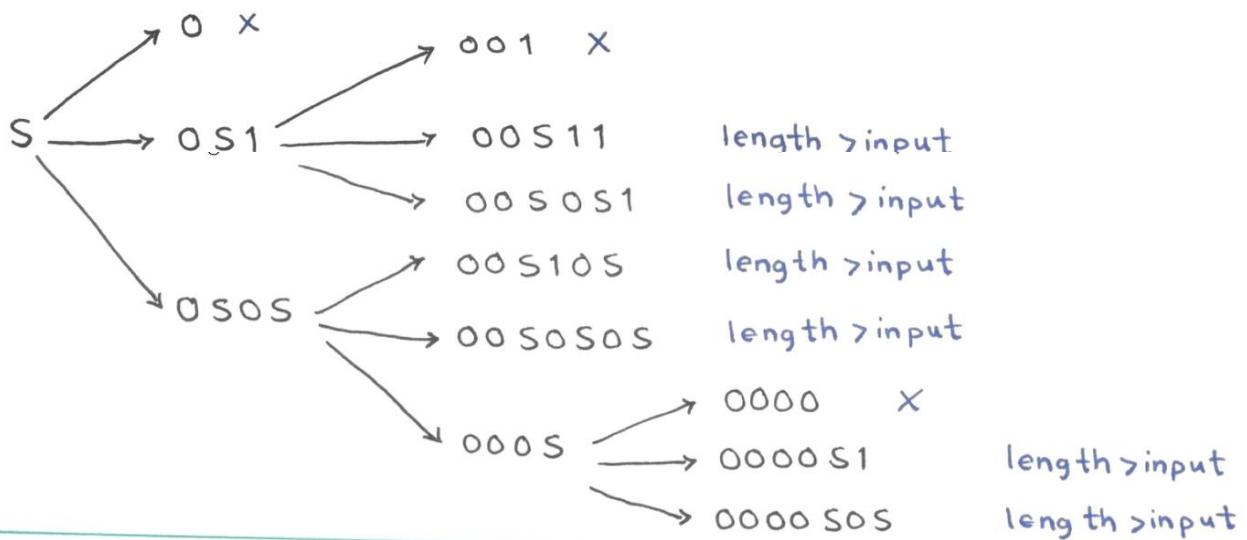
2) If input is not in the language, parsing will never stop ✓

→ Eliminate  $\epsilon$ - and unit- productions

Try all possible derivations but stop searching when  $|\text{derived str}| > |\text{in}|$

$$\begin{cases} S \rightarrow OS1 \mid OSOS \mid T \\ T \rightarrow S \mid 0 \end{cases} \Rightarrow S \rightarrow OS1 \mid OSOS \mid 0$$

input: 0011



Cocke - Younger - Kasami (CYK Algorithm)

Convert grammar to Chomsky Normal Form

A CFG is in CNF if every production is one of the following:

$A \rightarrow BC$  Exactly two non-start variables on right

$A \rightarrow a$  Exactly one terminal on right

$S \rightarrow \epsilon$   $\epsilon$ -production only allowed for start

(4)

For any context-free grammar there exists an equivalent grammar is CNF.

Step 1. Eliminate  $\epsilon$  and unit productions

- Every production RHS is either a single terminal or of length at least 2

Step 2. For each RHS (not single terminal) make RHS all vars

- For each terminal  $a$  create a new variable  $A_a$  and  $A_a \rightarrow a$
- Replace  $a$  by  $A_a$  in RHS of length > 2

Example.  $A \rightarrow BcDe$

$$A_c \rightarrow c, A_e \rightarrow e \implies A \rightarrow BA_c^c DA_e^e$$

Step 3. Break RHS longer than 2 into a chain of productions with RHS of two variables

Example.  $A \rightarrow BCDE$

$$A \rightarrow BF, F \xrightarrow{\text{fresh}} CG, G \rightarrow DE$$

|           |           |           |           |           |
|-----------|-----------|-----------|-----------|-----------|
| $X_{1,5}$ |           |           |           |           |
| $X_{1,4}$ | $X_{2,5}$ |           |           |           |
| $X_{1,3}$ | $X_{2,4}$ | $X_{3,5}$ |           |           |
| $X_{1,2}$ | $X_{2,3}$ | $X_{3,4}$ | $X_{4,5}$ |           |
| $X_{1,1}$ | $X_{2,2}$ | $X_{3,3}$ | $X_{4,4}$ | $X_{5,5}$ |
| $a_1$     | $a_2$     | $a_3$     | $a_4$     | $a_5$     |

$X_{i,j}$  : set of non-terminals deriving  $a_i \dots a_j$

Start symbol derives  $w = a_1 \dots a_n$  if and only if  $S$  is a member of  $X_{1,n}$

if  $B$  belongs to  $X_{1,1}$  and  $C$  to  $X_{2,5}$  and if  $A \rightarrow BC$  is a grammar rule add  $A$  to  $X_{1,5}$

| S    |      |
|------|------|
| S, A |      |
| S    | S    |
| A, B | S    |
| B    | A, C |
| b    | a    |
| b    | b    |
| b    | a    |
| b    | b    |

$$S \rightarrow AB \mid CB$$

$$A \rightarrow BA \mid a$$

$$B \rightarrow BC \mid b$$

$$C \rightarrow AC \mid a$$

①

طبعه سی دهم درس نظری

طبعه سی دهم درس نظری

Two grammars  $G_1$  and  $G_2$  are equivalent if they generate the same language  $L(G_1) = L(G_2)$

Theorem. Given a context-free grammar  $G$  there exists an equivalent Chomsky Normal Form (CNF)  $G'$  such that  $L(G) = L(G')$

CNF. A CFG is in CNF if every production is of the form

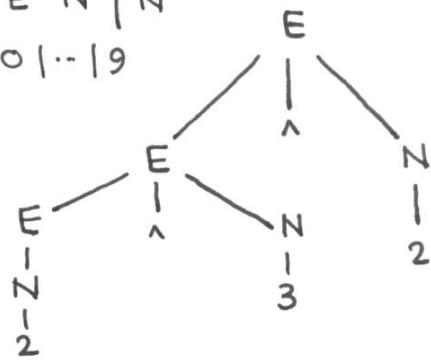
$$A \rightarrow BC$$

$$A \rightarrow a$$

$S \rightarrow \epsilon$  (if  $\epsilon \in L(G)$  and  $S$  does not appear on the RHS of any production)

$$E \rightarrow E^N \mid N$$

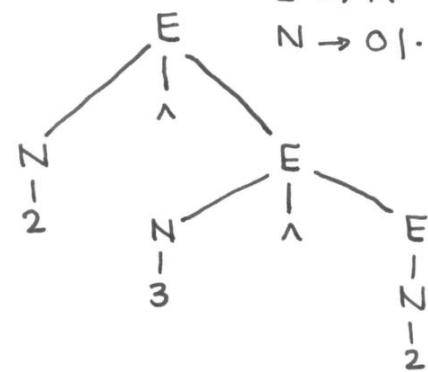
$$N \rightarrow 0 \mid 1 \mid 2$$



$$2^3 \cdot 3^2 = 8^2 = 64$$

$$E \rightarrow N^E \mid N$$

$$N \rightarrow 0 \mid 1 \mid 2$$



$$2^3 \cdot 3^2 = 2^9 = 512$$

Left-recursive Grammar: It has a non-terminal  $A$  such that there is a derivation  $A \xrightarrow{+} A\alpha$  for some string  $\alpha$

Right-recursive Grammar: It has a non-terminal  $A$  such that there is a derivation  $A \xrightarrow{+} \alpha A$  for some string  $\alpha$

Transformation of immediate left recursion (direct)

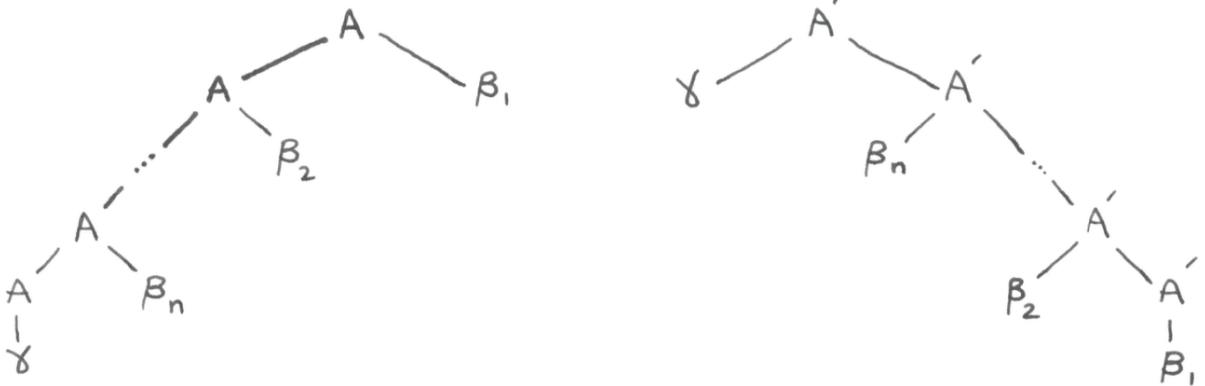
$$\begin{cases} A \rightarrow A\beta_1 \mid A\beta_2 \mid \dots \mid A\beta_h & (h \geq 1) \\ & \text{no } \beta_i \text{ string is empty} \\ A \rightarrow \gamma_1 \mid \gamma_2^+ \mid \dots \mid \gamma_k & (k \geq 1) \end{cases}$$

Create a new fresh non-terminal  $A'$

$$\begin{cases} A \rightarrow \gamma_1 A' \mid \gamma_2 A' \mid \dots \mid \gamma_k A' \\ A' \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_h A' \mid \epsilon \end{cases}$$

Or, after removing the  $\epsilon$ -production rule:

$$\begin{cases} A \rightarrow \gamma_1 A' \mid \gamma_2 A' \mid \dots \mid \gamma_k A' \mid \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_k \\ A' \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_h A' \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_h \end{cases}$$



Every leftmost derivation is replaced by rightmost

$$A \Rightarrow A\beta_2 \Rightarrow A\beta_3\beta_2 \Rightarrow \gamma, \beta_3\beta_2$$

$$A \Rightarrow \gamma, A' \Rightarrow \gamma, \beta_3 A' \Rightarrow \gamma, \beta_3\beta_2$$

Example. Conversion of immediate left recursion

$$\begin{cases} E \rightarrow E + T \mid T \\ T \rightarrow T \times F \mid F \\ F \rightarrow (E) \mid n \end{cases}$$

$$\begin{cases} E \rightarrow TE' \mid T \\ E' \rightarrow +TE' \mid +T \end{cases}$$

$$\begin{cases} T \rightarrow FT' \mid F \\ T' \rightarrow \times FT' \mid \times F \end{cases}$$

Simpler solution in this case :

$$\begin{cases} E \rightarrow T + E \mid T \\ T \rightarrow F \times T \mid F \\ F \rightarrow (E) \mid n \end{cases}$$

Indirect Left Recursion

Example.

$$\begin{cases} A \rightarrow Ba \mid \epsilon \\ B \rightarrow Cb \\ C \rightarrow Ac \end{cases}$$

$$A \Rightarrow Ba \Rightarrow Cba \Rightarrow \underline{Acba} \Rightarrow \dots$$

Idea. First convert indirect left recursion to direct:

$$A \rightarrow Acba \mid \epsilon$$

Then remove direct left recursion:

$$A \rightarrow cbaA \mid \epsilon$$

(3)

$$B \rightarrow A_1 \alpha_1$$

$$A_1 \rightarrow A_2 \alpha_2$$

⋮

$$A_{n-1} \rightarrow B \alpha_n$$

$$B \Rightarrow A_1 \alpha_1 \Rightarrow A_2 \alpha_2 \alpha_1 \Rightarrow \dots \Rightarrow B \alpha_n \alpha_{n-1} \dots \alpha_2 \alpha_1$$

Algorithm for grammar with no  $\epsilon$ -production

$$\text{cycle } A \xrightarrow{*} \alpha \xrightarrow{*} A$$

Pick an arbitrary order for all non-terminals  $\{A_1, A_2, \dots, A_m\}$

Eliminate all left recursions that do not increase the index of  $A_i$

Eliminate all rules of the form  $A_i \rightarrow A_j \alpha$  with  $j \leq i$

If we do this for each  $i$  from 1 to  $m$ , no loop can happen

Why? Every step in such a loop would only increase the index thus the original index cannot be reached again.

for  $i := 1$  to  $m$  do

for  $j := 1$  to  $i-1$  do

replace each rule of type  $A_i \rightarrow A_j \alpha$  ( $i > j$ ) by the rules:

$$A_i \rightarrow \gamma_1 \alpha \mid \gamma_2 \alpha \mid \dots \mid \gamma_k \alpha$$

where  $A_j \rightarrow \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_k$  are alternatives of  $A_j$

// note this may create immediate left recursion

eliminate all the immediate l-recursions that may have appeared as alternatives of  $A_i$

(4)

$$A_1 \rightarrow A_2 a \mid b$$

$$A_2 \rightarrow A_2 c \mid A_1 d \mid e$$

| $i$ | $j$        | Algorithm Action                                                  | Transformation                                                                                                                                 |
|-----|------------|-------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| 1   | skipped    | Eliminate immediate l-recursion of $A_1$                          | Unchanged                                                                                                                                      |
| 2   | 1          | Replace rule $A_2 \rightarrow A_1 d$<br>Expand non-terminal $A_1$ | $A_1 \rightarrow A_2 a \mid b$<br>$A_2 \rightarrow A_2 c \mid A_2 ad \mid bd \mid e$                                                           |
| 2   | terminated | Eliminate two immediate left recursions of $A_2$                  | $A_1 \rightarrow A_2 a \mid b$<br>$A_2 \rightarrow bd A'_2 \mid e A'_2 \mid bd \mid e$<br>$A_2 \rightarrow c A'_2 \mid ad A'_2 \mid c \mid ad$ |

## Greibach Normal Form

$$A \rightarrow a\alpha, \quad a \in \Sigma, \alpha \in V^* \quad S \rightarrow \epsilon \text{ (if } \epsilon \in L(G))$$

Every rule starts with a terminal, followed by zero or more vars

Kind of rules that violate Greibach Normal Form

- Epsilon Transitions  $A \rightarrow \epsilon$
- Unit productions  $A \rightarrow B$
- Productions such as  $X \rightarrow AaB$  (terminal other than leftmost)
- Productions such as  $X \rightarrow AB$  whose RHS starts with var

Transformation:

- Remove all left recursions
- Expand any variable that occurs in the first position of RHS of a rule

$$\left\{ \begin{array}{l} X \rightarrow Y\alpha \\ Y \rightarrow \gamma_1 | \gamma_2 | \dots | \gamma_n \end{array} \right. \quad X \rightarrow \gamma_1\alpha | \gamma_2\alpha | \dots | \gamma_n\alpha$$

$$\left\{ \begin{array}{l} A_1 \rightarrow A_2 a \\ A_2 \rightarrow A_1 c | bA_1 | d \end{array} \right.$$

(5)

- ① Eliminate left recursion

$$\left\{ \begin{array}{l} A_1 \rightarrow A_2 a \\ A_2 \rightarrow A_2 ac | bA_1 | d \end{array} \right.$$

$$\left\{ \begin{array}{l} A_1 \rightarrow A_2 a \\ A_2 \rightarrow bA_1 A'_2 | dA'_2 | d | bA_1 \\ A'_2 \rightarrow acA'_2 | ac \end{array} \right.$$

- ② Expand non-terminals in the first position until a terminal prefix appears

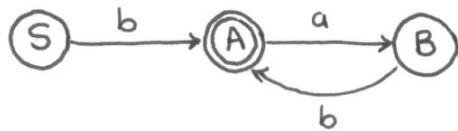
$$\left\{ \begin{array}{l} A_1 \rightarrow bA_1 A'_2 a | dA'_2 a | da | ba, a \\ A_2 \rightarrow bA_1 A'_2 | dA'_2 | d | bA_1 \\ A'_2 \rightarrow acA'_2 | ac \end{array} \right.$$

③ Substitute non-terminals for any terminal not in first place

$$\begin{cases} A_1 \rightarrow bA, A'_2 X_a | dA'_2 X_a | da | bA, X_a \\ A'_2 \rightarrow bA, A'_2 | dA'_2 | d | bA, \\ A'_2 \rightarrow aX_c A'_2 | aX_c \\ X_a \rightarrow a \quad X_c \rightarrow c \end{cases}$$

Some Interesting Features of GNF:

Every derivation of string  $w$  contains  $|w|$  rule applications



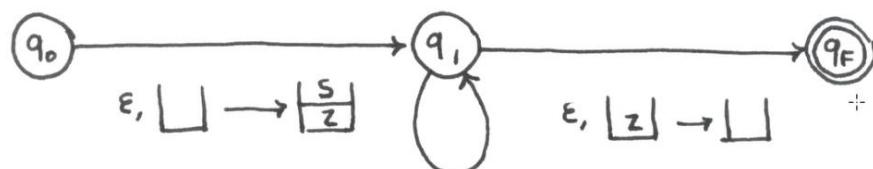
$$\begin{aligned} S &\rightarrow bA \\ S &\rightarrow aS \\ A &\rightarrow \epsilon \\ B &\rightarrow bA \\ A &\rightarrow aB \end{aligned}$$

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

⑥

Grammar:  $S \rightarrow aSb \mid \epsilon$

$$\text{Equivalent GNF : } \begin{aligned} S &\rightarrow \epsilon \mid aXY \mid aY \\ X &\rightarrow aXY \mid aY \\ Y &\rightarrow b \end{aligned}$$



$$a, [ ] \xrightarrow{\quad} [S] [ ]$$

$$a, [ ] \xrightarrow{\quad} [ ]$$

$$a, [S] [ ] \xrightarrow{\quad} [Y] [ ]$$

$$b, [Y] [ ] \xrightarrow{\quad} [ ]$$

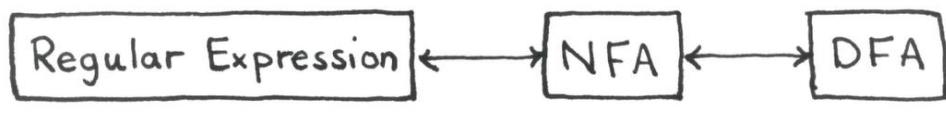
$$a, [X] [ ] \xrightarrow{\quad} [X] [Y]$$

$$\epsilon, [S] [Z] \xrightarrow{\quad} [Z]$$

①

جاءت حفظ

جاءت حفظ

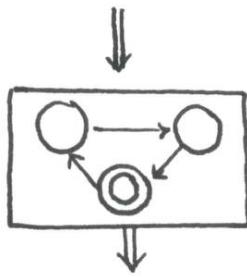
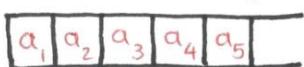


Declarative

Computational

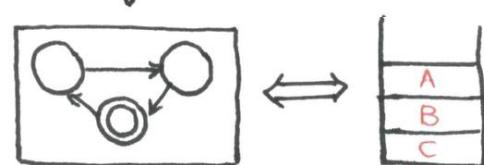
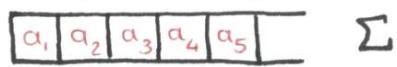


input string



yes/no

input string



Γ

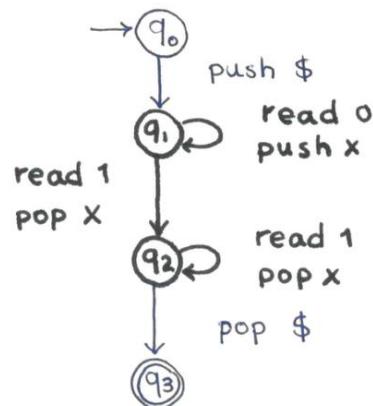
yes/no

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

Remember each 0 by pushing X onto stack

Upon reading a 1, pop X from stack

Accept? Hit stack bottom



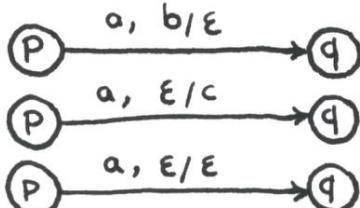
Read a, Pop b / Push c  $\xrightarrow{P} \xrightarrow{a, b/c} q$

- if next symbol is 'a' and top of stack is 'b'

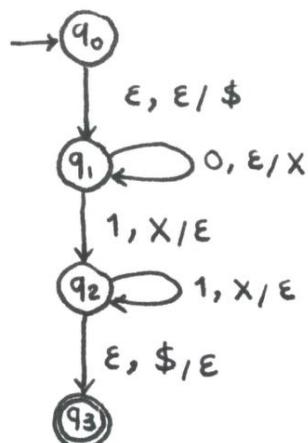
then read 'a', pop 'b' and push 'c'

- if  $a = \epsilon$  don't read next symbol

- if  $b = \epsilon$  don't pop next symbol



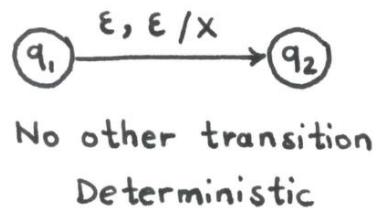
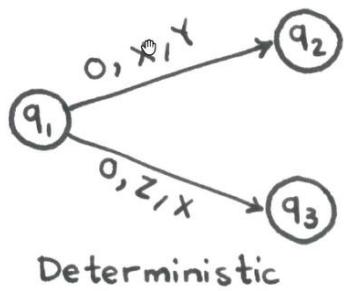
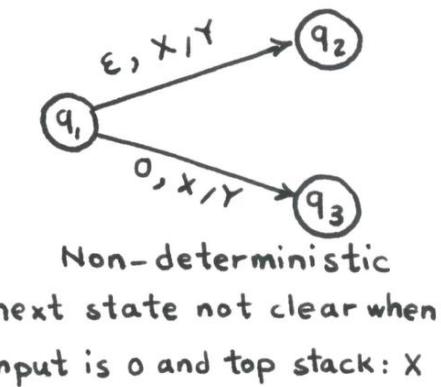
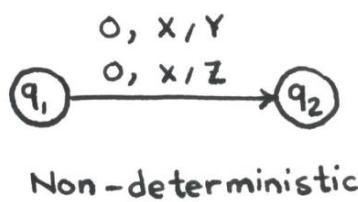
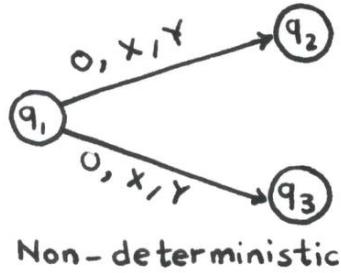
- only pop
- only push
- ignore stack



By default, PDA means non-deterministic pushdown automata. (2)

Non-determinism is essential: there are languages that cannot be recognized by DPDA but can be recognized by NPDA (lecture 17)

Deterministic pushdown automaton: in each state, there is at most one transition for every combination  $(a, b) \in \Sigma \times \Gamma$



A pushdown automaton is  $(Q, \Sigma, \Gamma, \delta, q_0, F)$

$Q$  finite set of states

$\Sigma$  finite set of input alphabet

$\Gamma$  finite set of stack alphabet

$\delta$  transition function

$q_0 \in Q$  initial state

$F \subseteq Q$  set of accepting states

$\delta: Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \rightarrow \text{subsets of } \{Q \times (\Gamma \cup \{\epsilon\})\}$

The transition function  $\delta$  maps a state  $q$ , an input symbol  $a$ , and a stack symbol  $x$  to a subset of possible transitions. The subset contains pairs of states and stack symbols, such as  $(q_1, \$)$  or  $(q_1, X)$ .

A diagram showing a state  $q_0$  transitioning to  $q_1$  via the rule  $\epsilon, \epsilon/\$$ . From state  $q_1$ , there is a transition labeled  $0, \epsilon/X$  back to  $q_0$ .

$$\delta(q_0, \epsilon, \epsilon) = \{(q_1, \$)\}$$

$$\delta(q_0, \epsilon, X) = \emptyset$$

$$\delta(q_0, \epsilon, \$) = \emptyset$$

Note. Finite automaton has a standard definition.

There are different and equivalent definitions of PDA. We use the one described in Sipser.

A PDA accepts input  $x$  if, for some computational path, PDA finishes reading all input symbols and stop at an accepting state.

When accepting an input string, stack needs not be empty.

Language of a PDA is the set of all strings in  $\Sigma^*$  it accepts

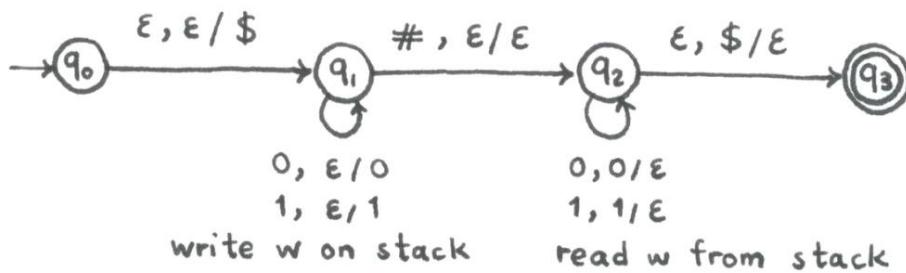
$$L = \{ w \# w^R \mid w \in \{0,1\}^* \}$$

$\#$ ,  $0 \# 0$ ,  $01 \# 10$  in  $L$

$$\Sigma = \{0, 1, \#\}$$

$$\Gamma = \{0, 1, \$\}$$

$\epsilon$ ,  $01 \# 1$ ,  $0 \# \# 0$  not in  $L$

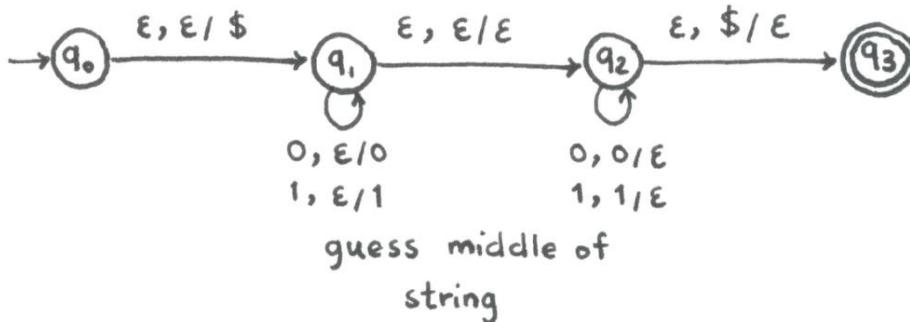


$$L = \{ww^R \mid w \in \Sigma^*\}$$

$$\Sigma = \{0, 1\}$$

$\epsilon$ ,  $00$ ,  $0110$  in  $L$

$011$ ,  $010$  not in  $L$



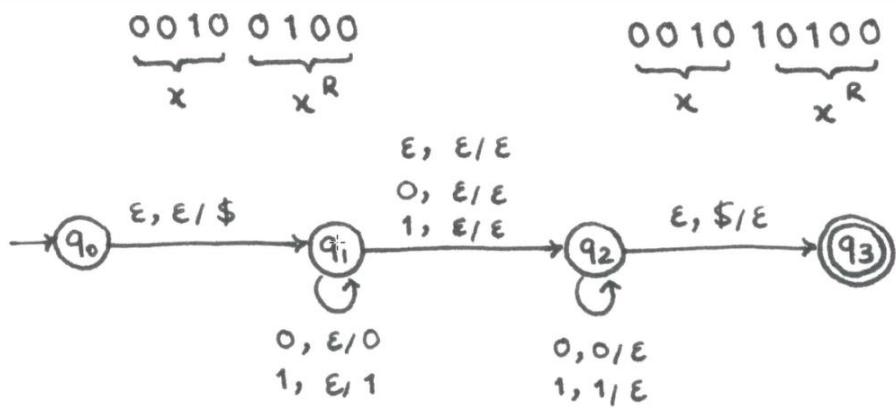
$$L = \{w \in \Sigma^* \mid w = w^R\}$$

$$\Sigma = \{0, 1\}$$

$\epsilon$ ,  $00$ ,  $010$ ,  $0110$  in  $L$

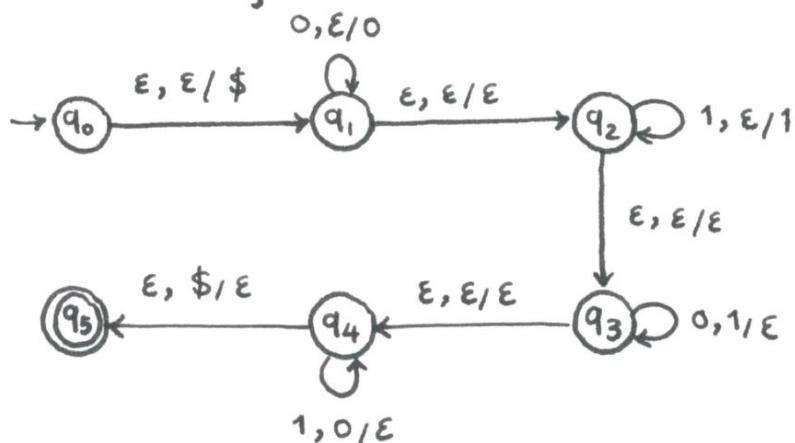
$011$  not in  $L$

(4)



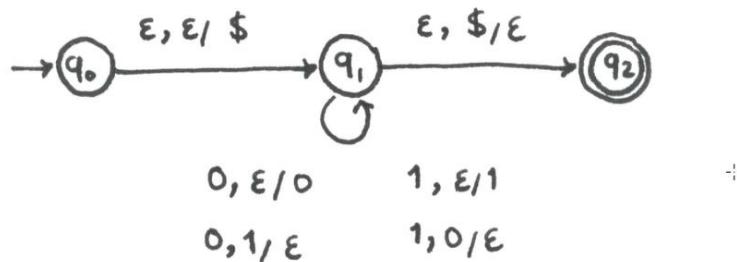
$$L = \{0^n 1^m 0^n 1^m \mid n \geq 0, m \geq 0\}$$

$$\Sigma = \{0, 1\}$$



$$L = \text{same number of 0's and 1's}$$

$$\Sigma = \{0, 1\}$$

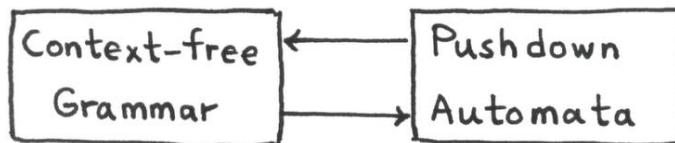


1)

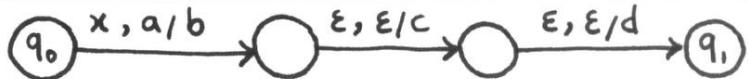
جنبه پارزدهم در نظر

جنبه پارزدهم در نظر

L has a context-free grammar if and only if it is accepted by some PDA



Sequence of transitions like



will be abbreviated as



Given a CFG G, convert it into PDA M with  $L(G) = L(M)$

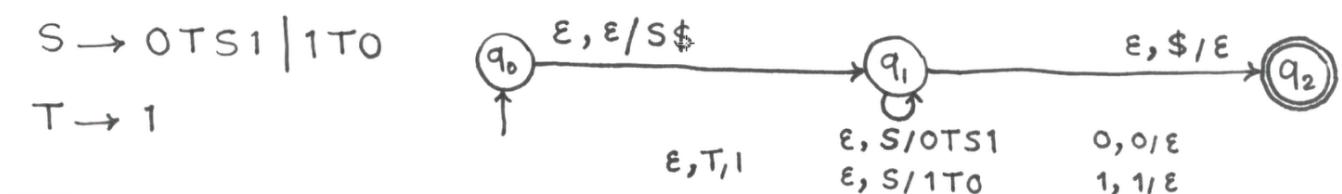
Idea. Build PDA that simulates a leftmost derivation

$$\begin{cases} S \rightarrow OTS1 \mid 1T0 \\ T \rightarrow 1 \end{cases} \quad S \xrightarrow{+}OTS1 \Rightarrow 01S1 \Rightarrow 011T01 \Rightarrow 011101$$

PDA works as follows :

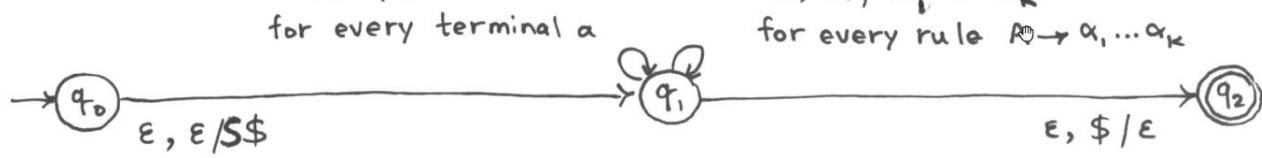
- ① Write start symbol onto stack
- ② Rewrite variable on top of stack according to production rule
- ③ Pop terminal if it matches input

| PDA Control                | Stack     | Input  |
|----------------------------|-----------|--------|
| write start symbol         | \$ S      | 011101 |
| replace by production rule | \$ 1S T O | 011101 |
| pop terminal and match     | \$ 1S T   | 11101  |
| replace by production rule | \$ 1S 1   | 11101  |
| :                          | :         | :      |

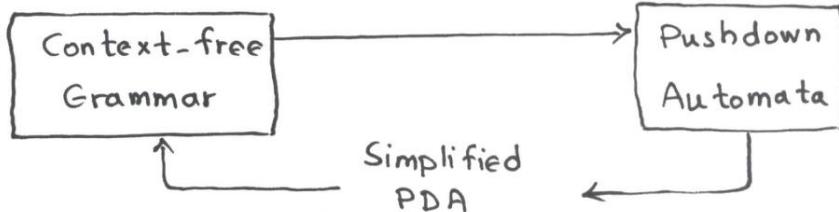


input: ↑011101      ↑011101      01↑11101      0↑11101      01↑1101  
 stack: \$ S       $\Rightarrow$  \$ 1S T O       $\Rightarrow$  \$ 1S T       $\Rightarrow$  \$ 1S 1       $\Rightarrow$  \$ 1S  
 $\Rightarrow$  \$ 1T O 1       $\Rightarrow$  \$ 1O T       $\Rightarrow$  \$ 101       $\Rightarrow$  \$ 10       $\Rightarrow$  \$ 1       $\Rightarrow$  \$

## General CFG to PDA Conversion

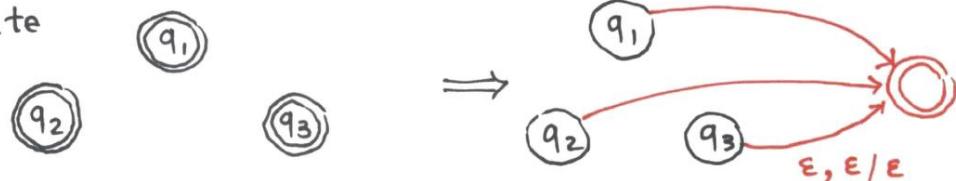


From PDAs to CFGs



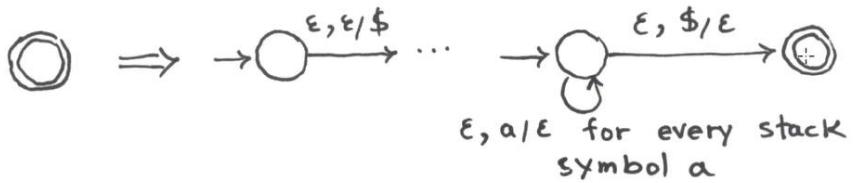
Simplified pushdown Automata:

- Has single accepting state
- Empties its stack before accepting
- Each transition is either a push, or a pop, but not both
- Single accepting state

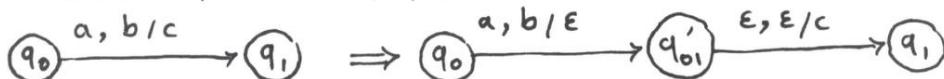


Empties its stack

- before accepting



- Each transition either push, or pop, but not both



Simplified PDA to CFG

For every pair  $(q, r)$  of states in PDA, introduce variable  $A_{qr}$  in the context-free grammar

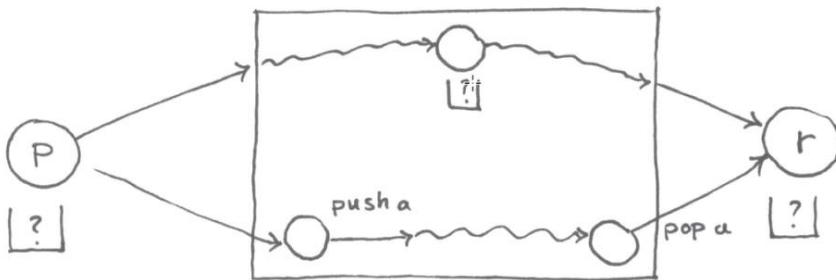
Intention:  $A_{qr}$  generates all strings that allow PDA go from  $q$  to  $r$  with empty stack both at  $q$  and  $r$

If  $q_{in}$  is start and  $q_{acc}$  is accepting state,  $A_{in \text{ acc}}$  generates all strings

Simplified PDA reading word from p to r

Two cases :

- Stack is empty in the middle
- Stack never empty in the middle



PDA

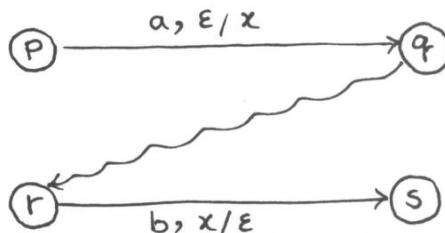
$$\forall p \in Q$$



$$\forall p, q, r \in Q$$



$$\forall p, q, r, s \in Q$$



Start Variable:  $A_{pq}$

CFG

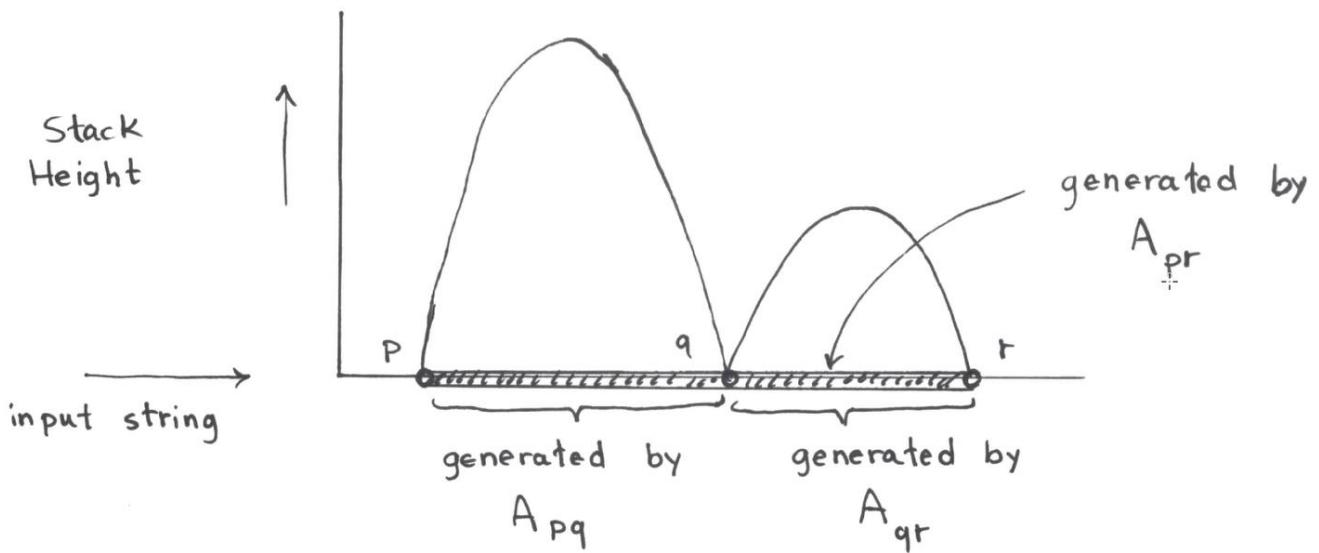
$$A_{pp} \rightarrow \epsilon$$

$$A_{pr} \rightarrow A_{pq} A_{qr}$$

$$A_{ps} \rightarrow a A_{qr} b$$

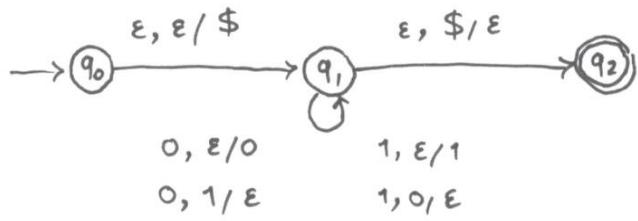
$a = \epsilon$  or  $b = \epsilon$  allowed

p : initial , q : accepting



Example: Simplified PDA to CFG

(4)



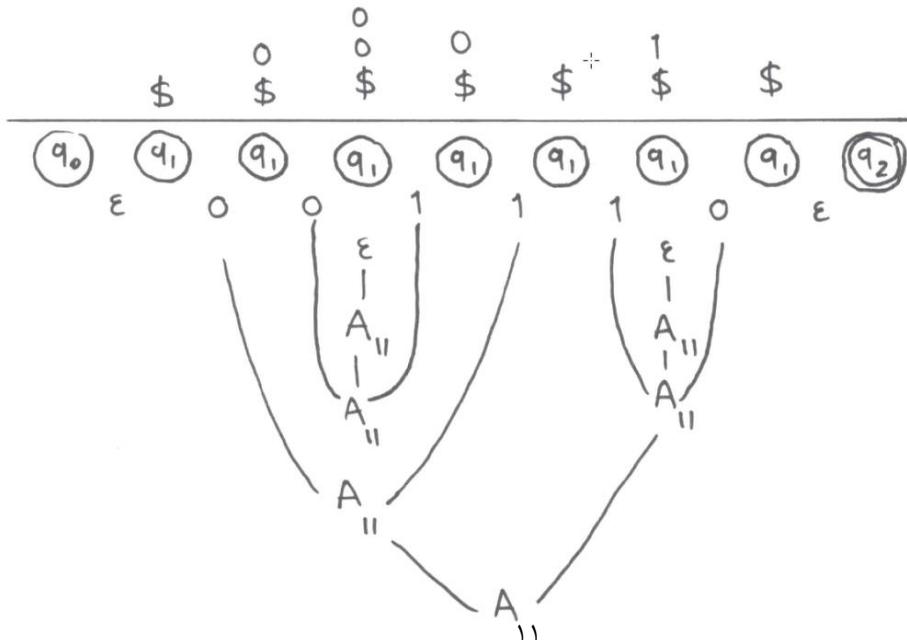
variables:  $A_{00}, A_{11}, A_{22}$   
 $A_{01}, A_{02}, A_{12}, \dots$

start var:  $A_{02}$

$$\begin{array}{l} A_{02} \rightarrow A_{01} A_{12} \\ A_{01} \rightarrow A_{01} A_{11} \\ A_{12} \rightarrow A_{11} A_{12} \\ A_{11} \rightarrow A_{11} A_{11} \end{array}$$

$$\begin{array}{l} A_{11} \rightarrow 0 A_{11} 1 \\ A_{11} \rightarrow 1 A_{11} 0 \\ A_{02} \rightarrow A_{11} \end{array}$$

$$\begin{array}{l} A_{00} \rightarrow \epsilon \\ A_{11} \rightarrow \epsilon \\ A_{22} \rightarrow \epsilon \end{array}$$



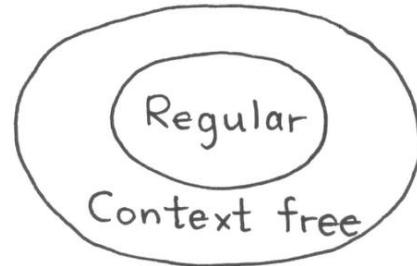
These languages are not regular, are they context-free?

$$L_1 = \{a^n b^n \mid n \geq 0\}$$

$$L_2 = \{a^n b^n c^n \mid n \geq 0\}$$

$$L_3 = \{ww^R \mid w \in \{a,b\}^*\}$$

$$L_4 = \{ww \mid w \in \{a,b\}^*\}$$



Let's try to design a CFG or PDA for  $L_3 = \{a^n b^n c^n \mid n \geq 0\}$

$$S \rightarrow aBc \mid \epsilon$$

read a / push x

$$B \rightarrow ??$$

read b / pop x

??

Intuition: If L has nested matchings then "probably" context-free

If not, "probably" not context-free

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

context-free

000111

nested

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

not context-free

11011101<sup>+</sup>

not nested

$$\{0^n 1^n 2^n \mid n \geq 0\}$$

not context-free

001122

not nested

We use a similar idea to the pumping lemma for regular languages to prove a language is not context-free

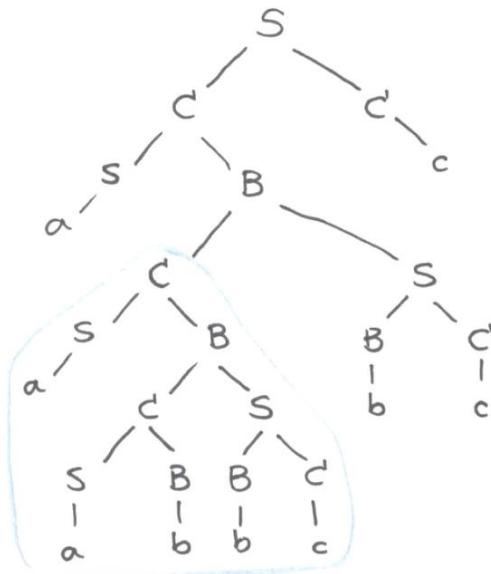
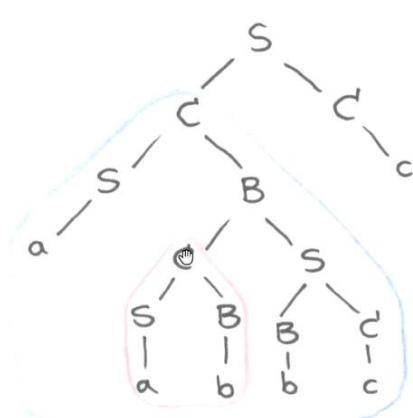
- Regular Languages : if a string is long enough, there must be some state that is repeated in the computation
- Context-free Languages: if a string is long enough, there must be a non-terminal that is repeated in a derivation

(2)

$$\begin{cases} S \rightarrow CC \mid BC \mid a \\ B \rightarrow CS \mid b \\ C \rightarrow SB \mid c \end{cases}$$

If a derivation is long enough, some variable must have appear twice on some root-to-leaf path in a derivation tree

$$\begin{aligned} S &\Rightarrow CC \Rightarrow SBC \Rightarrow SCSC \Rightarrow SSBSC \Rightarrow SSBBCC \\ &\Rightarrow aSBBCC \Rightarrow aaBBCD \Rightarrow aabBCDC \Rightarrow aabbCC \\ &\Rightarrow aabbCc \Rightarrow aabbcc \end{aligned}$$



We can "cut" and "paste" part of derivation tree

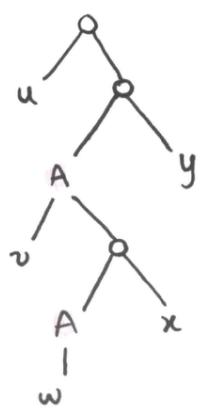
a **ab** bcc

a **aabb** **c** bcc

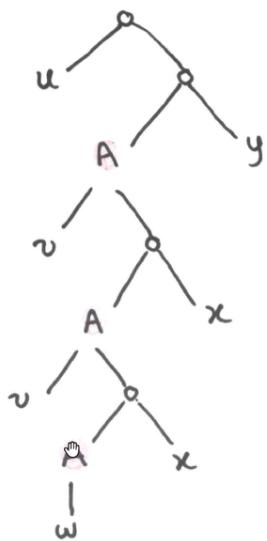
We can repeat this many times

$$\begin{aligned} a\ a\ b\ \mathbf{bc}\ c &\Rightarrow a\ a\ ab\ \mathbf{bc}\ bc\ c \Rightarrow a\ a\ a\ ab\ \mathbf{bc}\ bc\ bc\ c \Rightarrow \dots \\ &\Rightarrow (a)^i ab (\mathbf{bc})^i c \end{aligned}$$

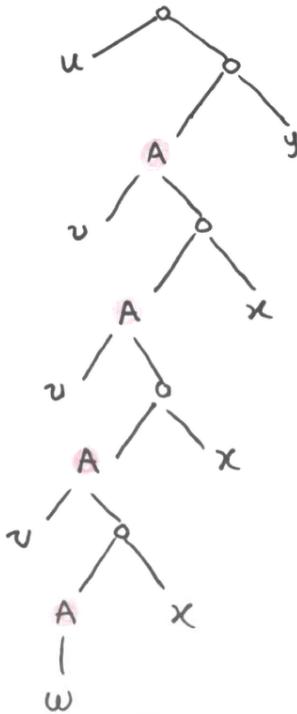
Each sufficiently large derivation will have a middle part that can be repeated indefinitely.



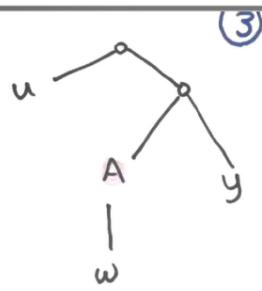
$uvwxy$



$uv^2wx^2y$



$uv^3wx^3y$



$uwxy$

(3)

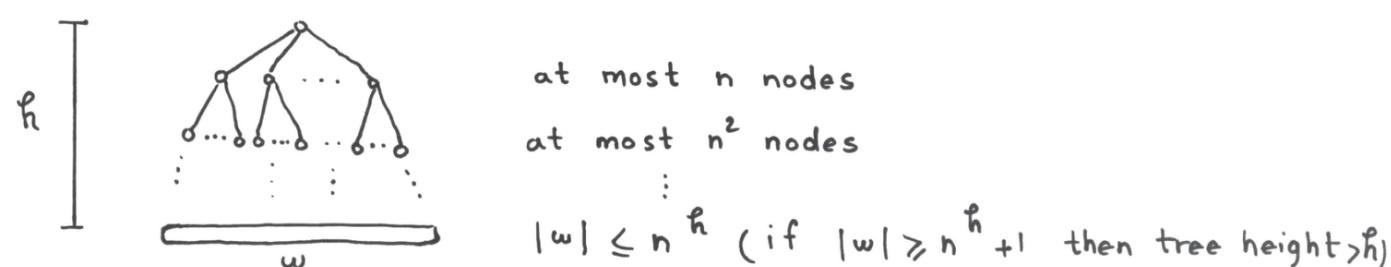
If string  $s \in L$  is "long enough" we can pump it

$G = (V, \Sigma, R, S)$  context-free grammar for  $L$

Assume longest rule of  $G$  has right hand side length  $n \geq 2$

$$A \rightarrow B_1 \dots B_n$$

Each node in derivation tree has  $\leq n$  children



If  $|w| \geq p = n^{|T|} + 1$  then tree height >  $|T|$

Some variable on longest path in tree are repeated

We can pump the derivation tree

(4)

$L$  is a context-free language  $\Rightarrow$

|                                                         |  |
|---------------------------------------------------------|--|
| $\exists p \geq 1$                                      |  |
| $\forall s \in L \cdot  s  \geq p$                      |  |
| $\exists u, v, w, x, y \cdot s = uvwxy$                 |  |
| ① $ vx  > 0$ second and forth are not both null         |  |
| ② $ uvw  \leq p$ middle three segments are not too long |  |
| $\forall i \geq 0 \cdot uv^i w x^i y \in L$             |  |

If

Demon: for all  $\forall$  player

$$\forall p \geq 1$$

Demon picks  $p$ 

$$\exists s \in L \cdot |s| \geq p$$

You pick  $s \in L$ 

$$\forall u, v, w, x, y \cdot s = uvwxy$$

Demon picks  $u, v, w, x, y$ 

$$- |vx| > 0$$

$$- |uvw| \leq p$$

$$\exists i \geq 0 \cdot uv^i w x^i y \notin L$$

You pick  $i$ then  $L$  is not context-freeYou win if  $uv^i w x^i y \notin L$ 

If you want to show that  $L$  is not context-free, it suffices to show you have a winning strategy:

no matter what demon does in 1 and 3, you have moves in 2 & 4

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

(5)

- Demon picks  $p \geq 1$
- You pick  $s = a^p b^p c^p \quad s \in L \quad |s| = 3p \geq p$
- Demon picks  $u, v, w, x, y \quad s = uvwxy \quad ux \neq \epsilon \quad |uvx| \leq p$
- You pick  $i=0 \quad uv^0 w x^0 y = uw y$

Since  $|uvx| \leq p$ ,  $uvx$  cannot contain all three symbols.

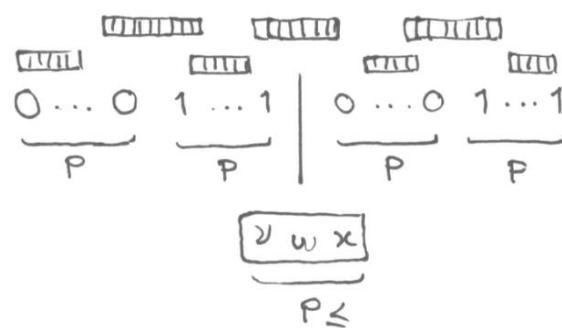
So  $uvx$  either does not have a's, b's or c's.

By  $i=0$  we are changing the balance of a's, b's and c's.

---

$$L = \{ww \mid w \in \{0,1\}^*\}$$

- Demon picks  $p \geq 1$
- You pick  $s = \overset{+}{0^p 1^p 0^p 1^p} \quad s \in L \quad |s| = 4p \geq p$
- Demon picks  $u, v, w, x, y \quad s = uvwxy \quad ux \neq \epsilon \quad |uvx| \leq p$



Suppose  $uvx$  is entirely in the first or second half.

$\rightarrow u v^2 w x^2 y$  is not of the form  $ww$

Suppose  $uvx$  is in the middle

$\rightarrow u v^0 w x^0 y$  is of the form  $0^p 1^i 0^j 1^p$  where either  $i$  or  $j$  is not  $p$ .

①

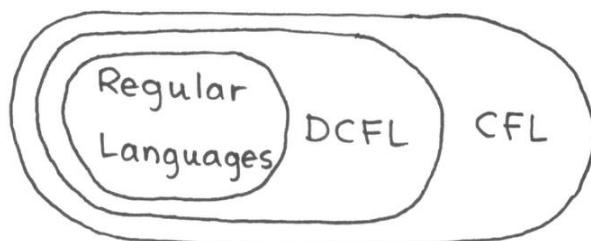
باعثی

جلسہ نظریہ درس مدرسے

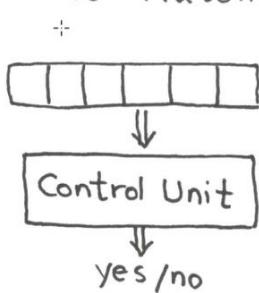
## Deterministic PDA

Parsers are DPDA : class of languages in DPDA gives insight what constructs are usable in programming languages

A context-free language L is called a Deterministic Context-Free Language (DCFL) if there is some DPDA that recognizes L

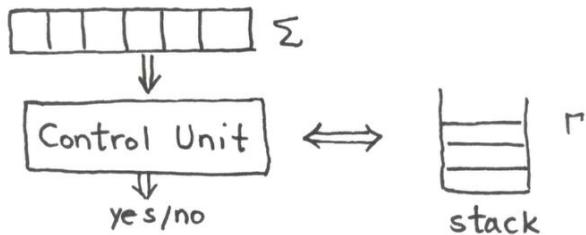


Finite Automata



Deterministic: in each state there is exactly one successor state for  $\forall a \in \Sigma$

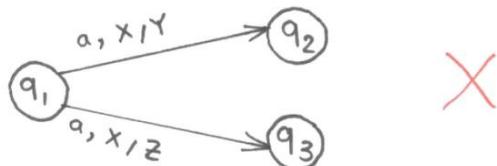
Pushdown Automata



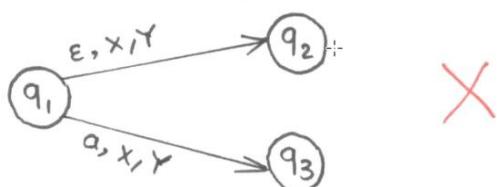
Deterministic: in each state there is at most one successor state for every combination  $(a,b) \in \Sigma \times \Gamma$

In a DPDA at any state  $q$ :

- At most one choice of move for any input symbol 'a' and stack symbol 'x'



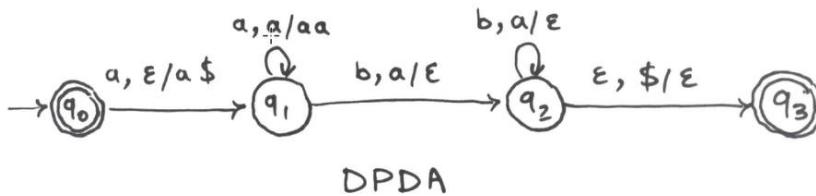
- No choice between using  $\epsilon$  or real input



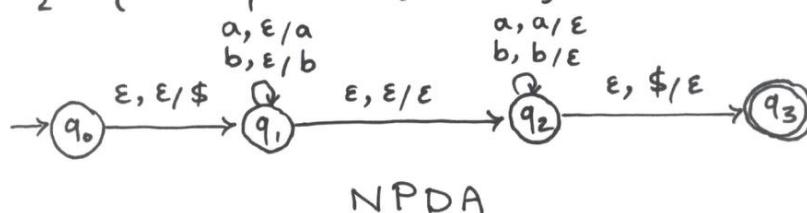
$M = (Q, \Sigma, \Gamma, \delta, q_0, F)$  is deterministic if it satisfies : (2)

- For any  $q \in Q$ ,  $a \in \Sigma \cup \{\epsilon\}$ ,  $x \in \Gamma$  the set  $\delta(q, a, x)$  has at most one element
- For any  $q \in Q$ ,  $x \in \Gamma$ , if  $\delta(q, \epsilon, x) \neq \emptyset$  then  $\delta(q, a, x) = \emptyset$  (for any  $a \in \Sigma$ )

Example :  $L_1 = \{a^n b^n \mid n \geq 0\}$



Example :  $L_2 = \{ww^R \mid w \in \{a, b\}^*\}$



$\left\{ \begin{array}{l} \text{Deterministic} \\ \text{Context-free} \\ \text{Language} \\ (\text{DCFL}) \end{array} \right\} \subset \left\{ \begin{array}{l} \text{Context-free} \\ \text{Languages} \end{array} \right\}$ 

- Every DPDA is also NPDA
- There exists a context-free language  $L$  which is not accepted by any DPDA

$$L = \{a^n b^n \mid n \geq 0\} \cup \{a^n b^{2n}\} \quad n \geq 0$$

Language  $L$  is context-free

$$\begin{array}{ll}
 S \rightarrow S_1 \mid S_2 & \{a^n b^n\} \cup \{a^n b^{2n}\} \\
 S_1 \rightarrow a S_1 b \mid \epsilon & \{a^n b^n\} \\
 S_2 \rightarrow a S_2 b b \mid \epsilon & \{a^n b^{2n}\}
 \end{array}$$

There is no DPDA that accepts  $L$

Assume for contradiction that  $L = \{a^n b^n\} \cup \{a^n b^{2n}\}$  is deterministic context-free. ③

There is a DPDA  $M$  with  $L(M) = \{a^n b^n\} \cup \{a^n b^{2n}\}$

Since  $M$  is deterministic the following path should exist in  $M$



$\{a^n b^n c^n \mid n \geq 0\}$  is not context-free (pumping lemma)

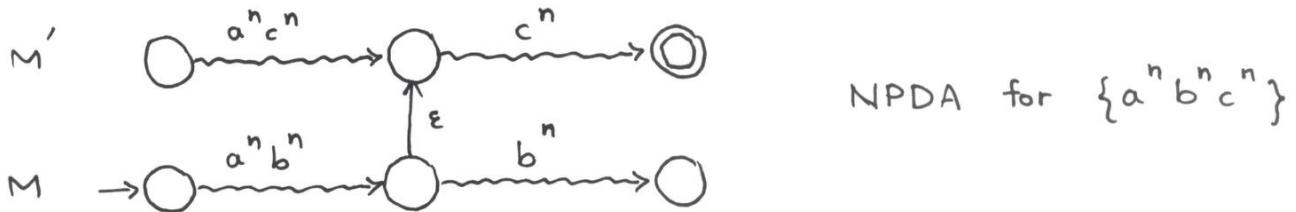
Using  $M$  we construct NPDA that accepts  $a^n b^n c^n$



Modify  $M \downarrow$  replace  $b$  with  $c$



NPDA that accepts  $\{a^n b^n c^n \mid n \geq 0\}$



Closure Properties:

Union, Concatenation, Kleene closure, Reversal

Non-closure Properties:

Difference, Complement, Intersection

Union

$$L_1 = L(G_1) \quad L_2 = L(G_2)$$

+

Rename variables in  $G_1$  so they don't conflict with those of  $G_2$

$S_1$ : start symbol of  $G_1$

$S_2$ : start symbol of  $G_2$

Add production rule  $S \rightarrow S_1 \mid S_2$

## Concatenation

Let  $S_1$  and  $S_2$  be the start symbols of  $G_1$  and  $G_2$

Add a new start symbol  $S$  and production  $S \rightarrow S_1 S_2$

## Star

Let  $L$  have grammar  $G$  with start symbol  $S$ ,

Add production  $S \rightarrow S, S \mid \epsilon$

Rightmost derivation of  $S$  generates a sequence of zero or more  $S$ 's

## Reversal

If  $L$  is a CFL with grammar  $G$ , form a grammar for  $L^R$  by reversing the right side of every production

Example: Production rules of  $G$ :  $S \rightarrow 0S1 \mid 01$

Reversal of  $G$ :  $S \rightarrow 1S0 \mid 10$

Non-closure under intersection (Does not mean it is never)

$L_1 = \{a^n b^n c^n \mid n \geq 1\}$  not CFL

$L_2 = \{a^n b^n c^m \mid n \geq 1, m \geq 1\}$  is CFL

$$S \rightarrow AB$$

$$A \rightarrow 0A1 \mid 01$$

$$B \rightarrow 2B \mid 2$$

$L_3 = \{a^m b^n c^n \mid n \geq 1, m \geq 1\}$  is CFL

$$L_1 = L_2 \cap L_3$$

Note. Intersection of a context-free language and a regular language is context-free.

## Non-closure under difference

Any class of languages that is closed under difference is closed under intersection

$$L \cap M = L \setminus (L \setminus M)$$

## Non-closure under complement

(5)

1- Assume complement of every CFL is a CFL

2- Let  $L_1$  and  $L_2$  be two CFLs

3- CFL are closed under union

$$\overline{L_1 \cup L_2} \quad X$$



Alan Turing (1912 - 1954)

Film: "The Imitation Game" بازی تعلیم

Turing Test: Classic test of Artificial Intelligence that asks a computer to trick a human into thinking they are talking to a person

Cryptography: Alan Turing broke German encryption machines during WWII

Turing Award: The "Nobel Prize" of computer science

Hilbert's Entscheidungsproblem (Decision Problem)

"Write a program" to solve the following task:

Input: mathematical statement (in first-order logic)

Output: whether the statement is true

(In fact, he didn't ask to "write a program", but to design a decision procedure)

Examples of statements expressible in first-order logic

Fermat's last theorem:

$$x^n + y^n = z^n$$

has no integer solution  
for integers  $n \geq 3$

Twin prime conjecture:

There are infinitely many pairs of primes of the form  $p$  and  $p+2$

Church (1935 - 1936) and Turing (1936 - 1937) independently showed the procedure that Entscheidungsproblem asks for cannot exist

Church:  $\lambda$ -calculus

Turing: Turing machine

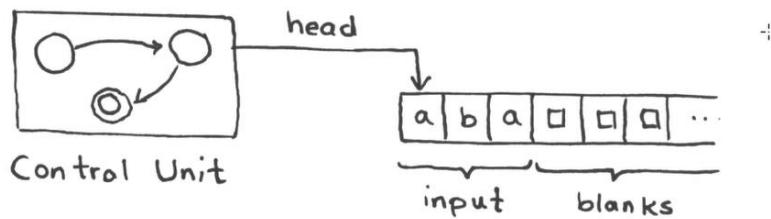
Goal: Model that can compute anything that we can compute

Church-Turing Thesis: Any "algorithm" can be carried out by Turing machine

Intuitive notion of algorithms coincides with those implementable on Turing machines

Turing machine : Finite - state control unit equipped with an infinite tape as its memory (2)

Tape begins with the input to the machine written on it, surrounded with infinitely many blank cells



Machine has a tape head that can read and write on a single memory cell at a time

At each step:

- Write a symbol to the tape cell under the tape head
- Change state
- Move the tape head to the left or to the right

A Turing machine has two alphabets :

① Input alphabet  $\Sigma$

② Tape alphabet  $\Gamma$  ( $\Sigma \subset \Gamma$ )

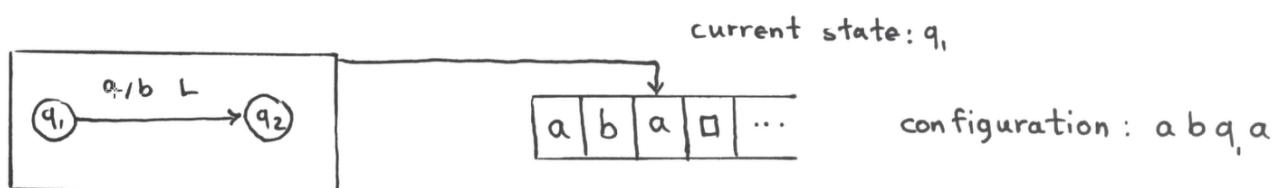
$\Gamma$  can contain any number of symbols, but always contain at least

Guarantee:  $\square \notin \Sigma$  blank  $\square$

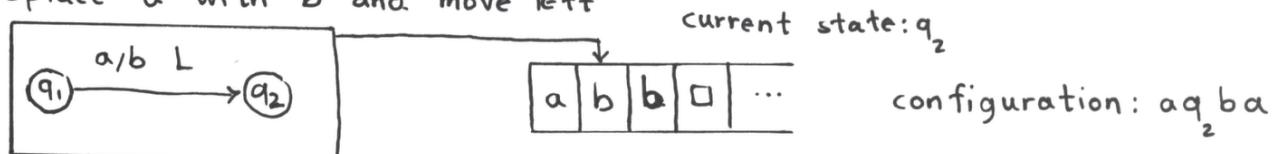
At startup:

Tape contains a finite input string followed by infinite sequence of blanks, and the head is positioned over the first cell on the tape.

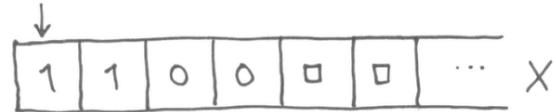
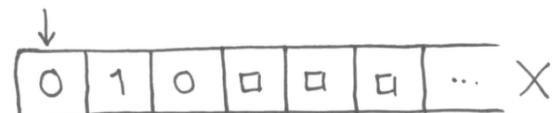
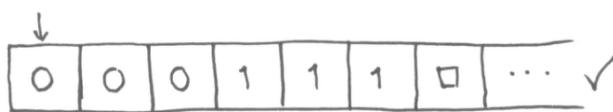
How Turing Machine operate



Replace 'a' with 'b' and move left

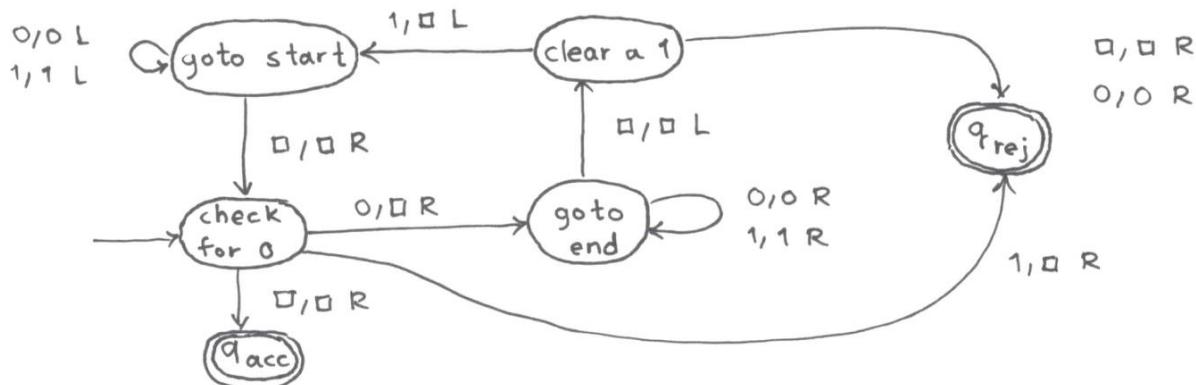


$$L = \{0^n 1^n \mid n \in \mathbb{N}\}$$



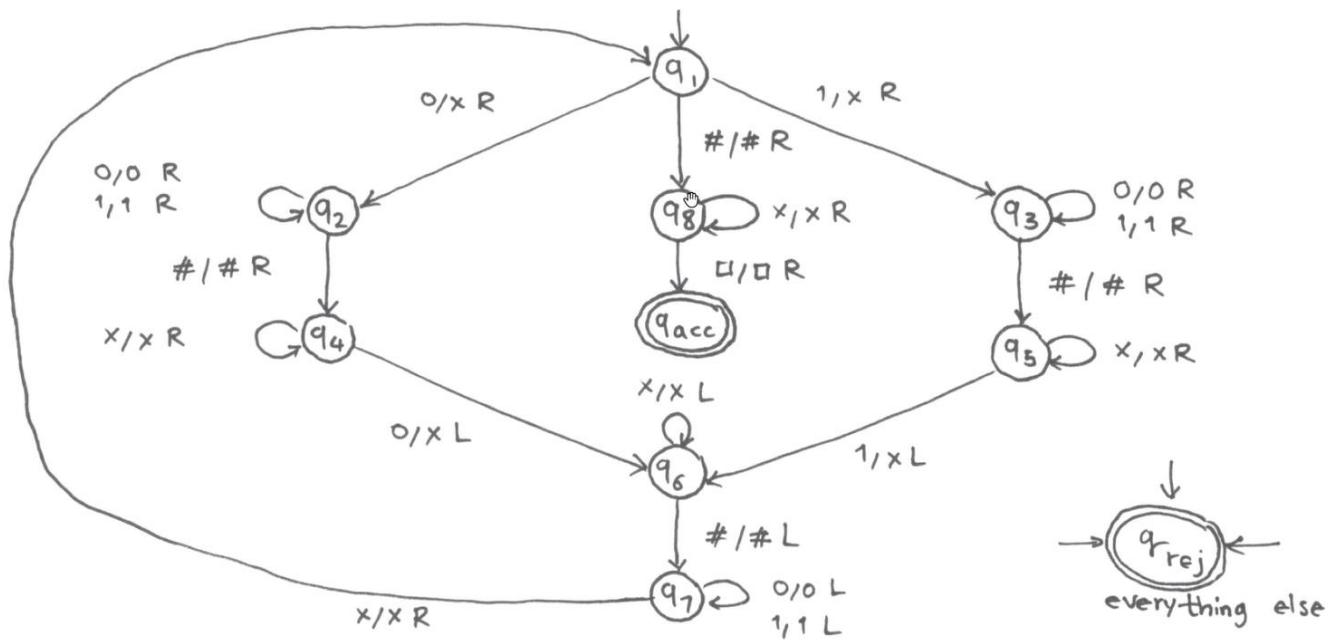
A Recursive Approach:

- The string  $\epsilon$  is in  $L$
- The string  $0w1$  is in  $L$  iff  $w$  is in  $L$
- Any string starting with 1 is not in  $L$
- Any string ending with 0 is not in  $L$



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

- ① Until you reach #
- ② Read and remember entry  $x \underline{1} 100 \# x 1100$
- ③ Write x  $x \underline{x} 100 \# x 1100$
- ④ Move right past # and past all x's  $x x 100 \# x \underline{1} 100$
- ⑤ If this entry is different, reject  $\circlearrowleft$
- ⑥ Write x  $x x 100 \# x \underline{x} 100$
- ⑦ Move left past # and to right of first x  $x x \underline{1} 00 \# x x 100$
- ⑧ If you see only x's followed by □, accept



Turing Machine  $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$

$Q$  finite set of states

$\Sigma$  finite set of input alphabets not containing  $\square$

$\Gamma$  finite set of tape alphabet including  $\square$        $\Sigma \subseteq \Gamma$

$q_0 \in Q$  initial state

$q_{\text{acc}}, q_{\text{rej}}$  accepting and rejecting states      ( $q_{\text{acc}} \neq q_{\text{rej}}$ )

$\delta$  transition function

$$\delta : (Q \setminus \{q_{\text{acc}}, q_{\text{rej}}\}) \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$$

Turing machines are deterministic. They do not get stuck.

Starting configuration       $q_0 w$

Accepting configuration contains  $q_{\text{acc}}$

Rejecting configuration contains  $q_{\text{rej}}$

Turing machine accepts  $x$  if there is a sequence of configurations

$C_0, C_1, \dots, C_k$  where

$C_0$  is starting       $C_i$  yields  $C_{i+1}$        $C_k$  accepting

Language recognized by  $M$ : set of all strings that  $M$  accepts

جامعة عاليٰ

حلبة نزدِهم درس نظریٰ

Church-Turing

Thesis

Intuitive notion

of Algorithm

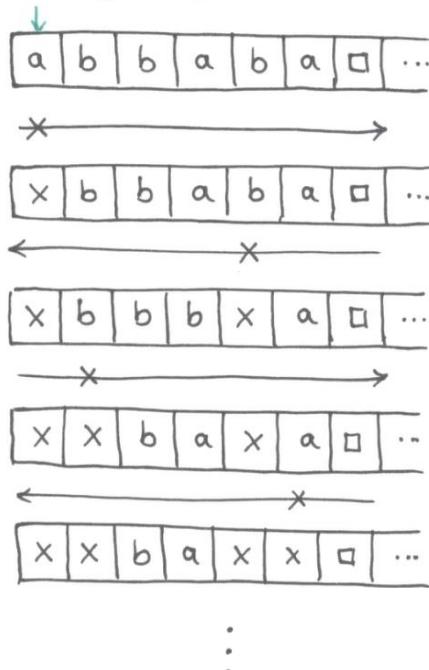
equals

Turing machine

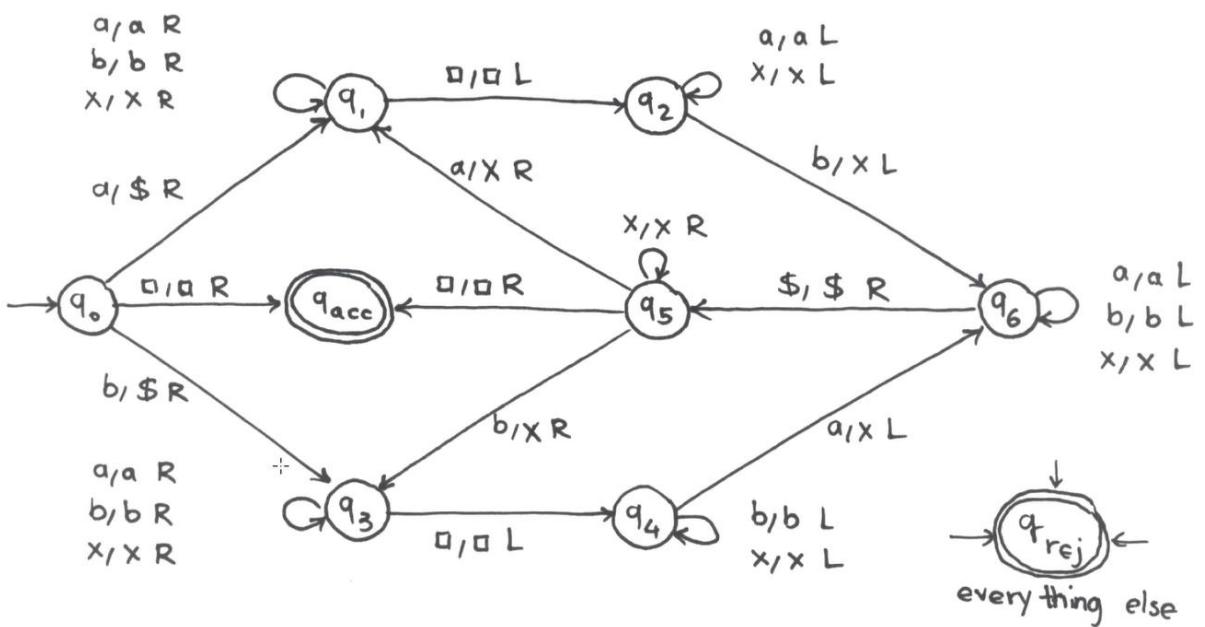
algorithms

(Sipser, p. 183)

$$L = \{w \in \{a, b\}^* \mid n_a(w) = n_b(w)\}$$



:



(2)

$$L = \{ w \in a^* \mid |w| = 3^n, n \geq 0 \}$$

High-level idea: repeatedly divide by 3

Separate a's into groups of three a's

Three cases:

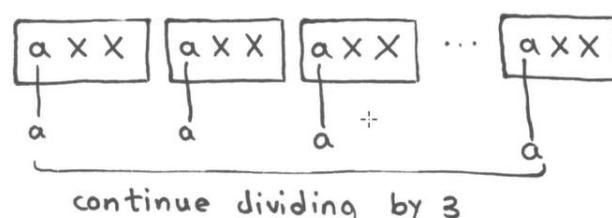




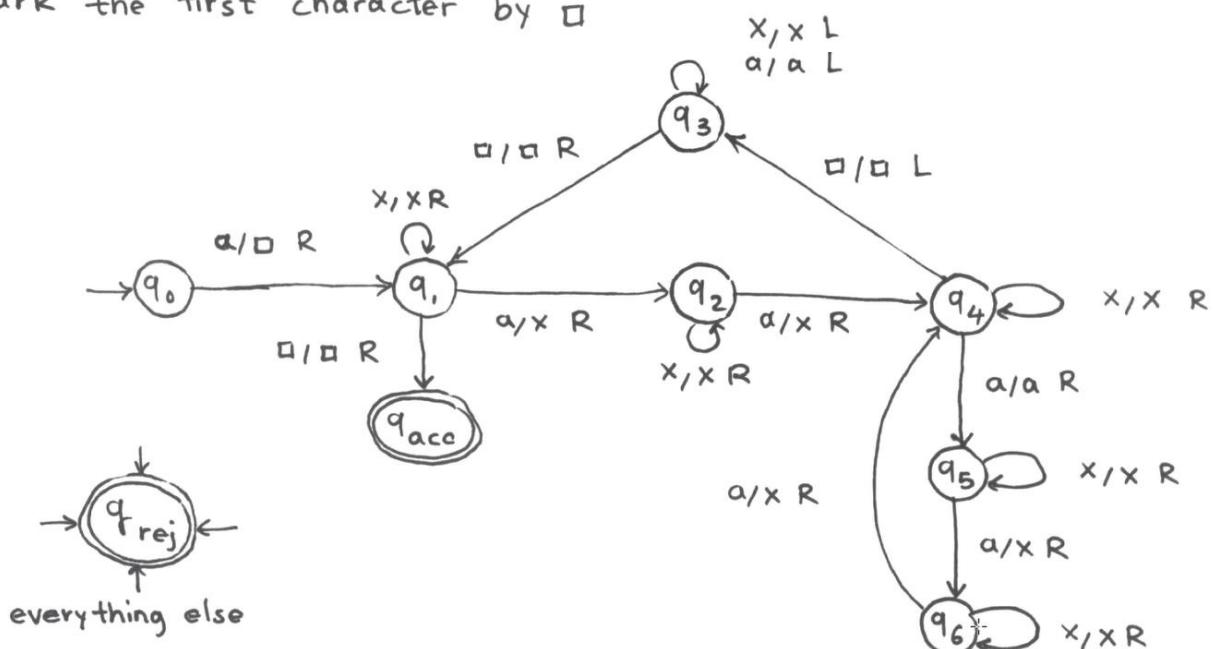


Cross off the last two a's in each block

Continue only if the last block ends as 



Mark the first character by  $\square$



$$L = \{a^i b^j c^k \mid i+j+k \text{ and } i, j, k > 0\}$$

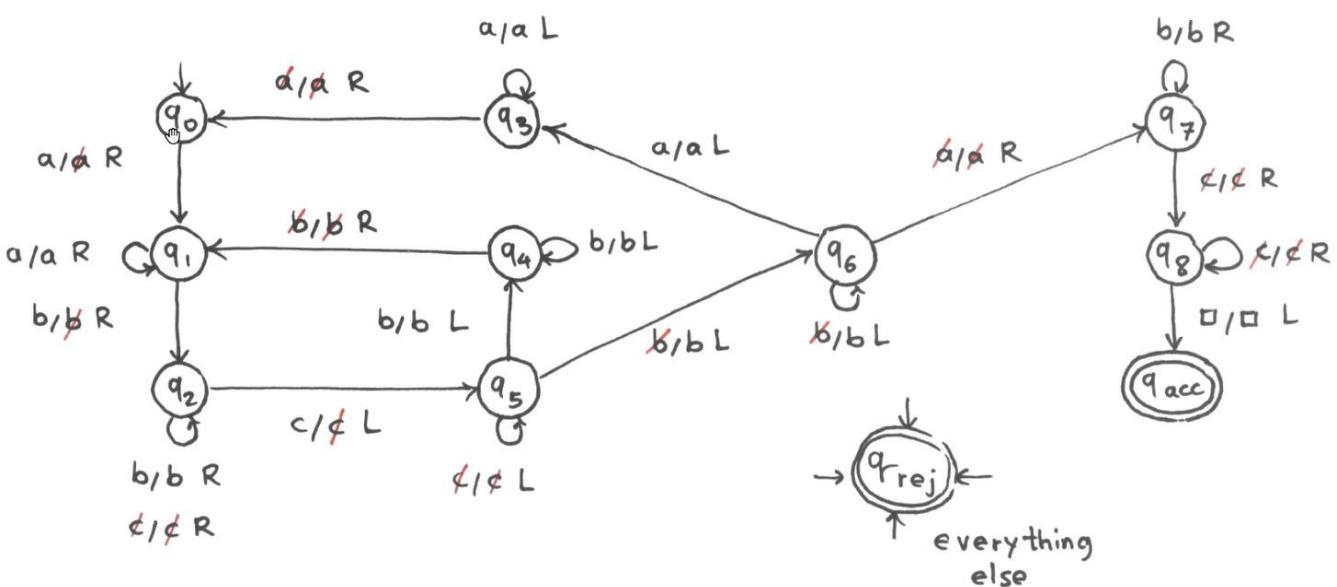
High-level description of TM:

- ① For every a:  
Cross off a
- ② Cross off same number of b's and c's
- ③ Uncross the crossed b's (but not c's)
- ④ If all a's and c's are crossed off, accept

$$\Sigma = \{a, b, c\} \quad \Gamma = \{a, b, c, \alpha, \beta, \gamma, \square\}$$

Example:

- ① a a b b c c c
- ② ~~a~~ a b b c c c
- ③ ~~a~~ a ~~b~~ ~~b~~ ~~c~~ c c
- ④ ~~a~~ a b b ~~c~~ c c
- ⑤ ~~a~~ ~~a~~ b b ~~c~~ c c
- ⑥ ~~a~~ ~~a~~ ~~b~~ b ~~c~~ c c
- ⑦ ~~a~~ ~~a~~ ~~b~~ b ~~c~~ c c
- ⑧ ~~a~~ ~~a~~ b b ~~c~~ c c



Unlike for DFAs, NFAs, PDAs we rarely give complete state diagrams of Turing machines

We usually give a high-level description

We are interested in algorithms behind Turing machines

### Element Distinctness Problem

$$L = \{ \# x_1 \# x_2 \dots \# x_m \mid x_i \in \{0,1\}^* \text{ and } x_i \neq x_j \text{ for every } i \neq j \}$$

Set a left marker (red) at the first  $\neq$

$\# x_1 \# x_2 \# x_3 \dots \# x_m$

Scan to right until you reach a  $\#$  and set a right marker (blue)

$\# x_1 \# x_2 \# x_3 \dots \# x_m$

Zig zag compare  $x_1$  and  $x_2$ :

If every character of  $x_1$  matches  $x_2$  with no extra character  $\rightarrow$  reject

If not, move right marker to next  $\#$  and compare  $x_1$  and  $x_3$

$\# x_1 \# x_2 \# x_3 \dots \# x_m$

If  $x_1$  is different from  $x_2, \dots, x_m$  go back and advance and

Reset the blue marker and repeat

$\# x_1 \# x_2 \# x_3 \dots \# x_m$

---

$$L = \{ a^p \mid p \text{ is a prime number} \}$$

Sieve of Eratosthenes      عَرْبَلِيَّةُ اِرَاثَتِينِ

Suppose we want to check if  $p$  is prime

We write all numbers from 2 to  $p$  in order

Repeat:

- Find the smallest number in the list: declare it prime
- Cross all multiples of that number
- Until all numbers are declared prime or crossed

|    |    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|----|
| 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |

...

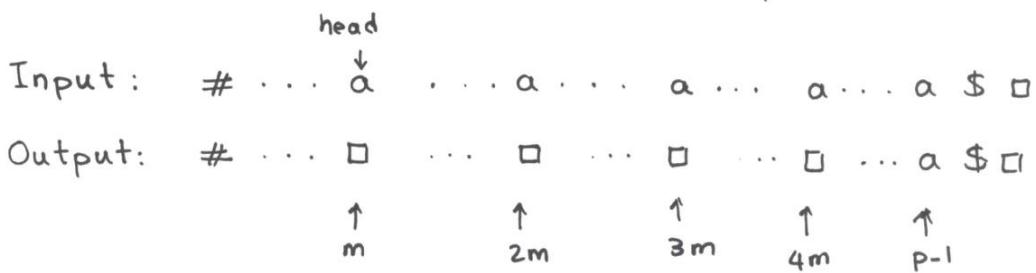
Tape: a a a a  $\square$   $\oplus$

Put  $\#$  at 1 and  $\$$  at  $p$  to know the start and end of the tape

$\# \uparrow a \ a \dots a \ a \$ \square \oplus$   
1

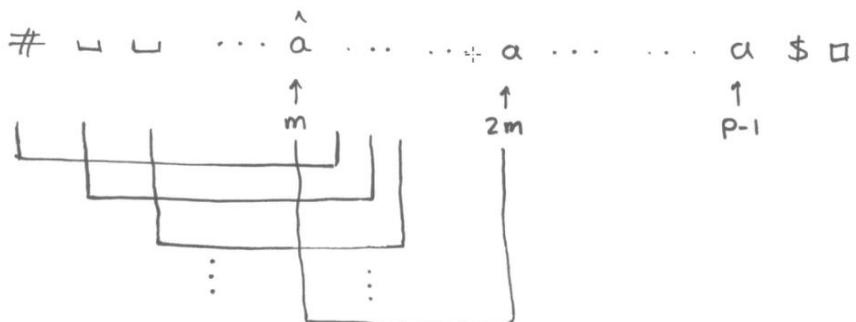
We first solve the following subproblem:

**SUBPROBLEM:** Assume the head at position  $m$  of tape. Convert all the multiples of  $m$  to  $\square$ . If  $\$$  is a multiple of  $m$  convert it to  $\$^*$ .



First put a special symbol  $\hat{a}$  at position  $m$ .

Zig-zag and match the characters between # and  $\hat{a}$ .



This will get us to position  $2m$ . Mark it with  $\hat{a}^*$ .

Zig-zag and match characters between  $\hat{a}$  and  $\hat{a}^*$  to reach position  $3m$ .

Now replace  $\hat{a}$  with  $\square$ ,  $\hat{a}^*$  with  $\hat{a}$  and position  $3m$  with  $\hat{a}$ .

Continue zig-zag matching using  $\hat{a}$  and  $\hat{a}^*$ . This way we can find all the multiples of  $m$  and remove them.

During the zig-zag matching:

- If the head passes over  $\$$ , stop. Convert  $\hat{a}$  and  $\hat{a}^*$  to  $\square$ .
- If the head wants to replace  $\$$  ( $\$$  is on a position which is multiple of  $m$ ) replace it with  $\$^*$ .

**PROBLEM:** Start at position 2.

Do the following until the head reaches  $\$^*$  or  $\$$ :

{ Convert all the multiples of the position of head to blank (subproblem)  
 { Continue to next position which is not  $\square$  above

If head reaches  $\$$  accept. If head reaches  $\$^*$  reject.

①

جاء نظریہ

جاء بیتم درس نظریہ

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

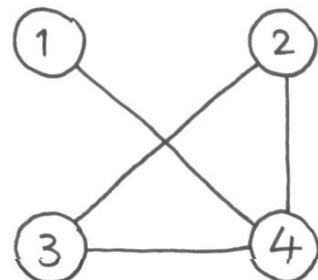
- How do we give a graph to a Turing machine as input?
- How to encode a graph  $G$  as a string  $\langle G \rangle$ ?

One way to encode a graph:

(nodes) (edges)

$(1, 2, 3, 4) ((1, 4), (2, 3), (3, 4), (4, 2))$

- no node appears twice
- edges are pairs (first node, second node)



On input  $\langle G \rangle$ :

0. Verify that  $\langle G \rangle$  is a description of a graph

No node/edge repeats : similar to element distinctness

Edge endpoints are nodes: also similar to element distinctness

1. Mark the first node of  $G$

Mark the leftmost digit with a dot, e.g. 12 becomes  $\dot{1}2$

2. Repeat until no new nodes are marked

2.1 For each node, mark it if it is attached to an already marked node

For every dotted  $u$  and every undotted node  $v$ :

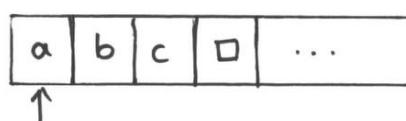
Underline both  $u$  and  $v$  from the list

Try to match them with an edge from the edge list

If not found, remove underline from  $u$  and/or  $v$  and try another

3. If all nodes are marked, accept, otherwise reject

Standard Turing machine:



Extends infinitely to right  
(semi-infinite tape)

## Turing Machine Variants

We prove each new class has the same power with standard TM

For every machine  $M_1$  of first class

there is a machine  $M_2$  of second class

such that  $L(M_1) = L(M_2)$  <sup>⊕</sup> and vice versa

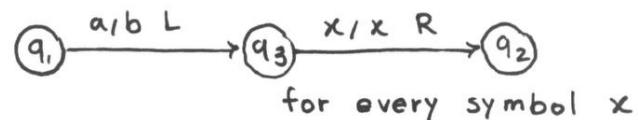
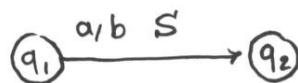
### Turing Machine with Stay Option

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

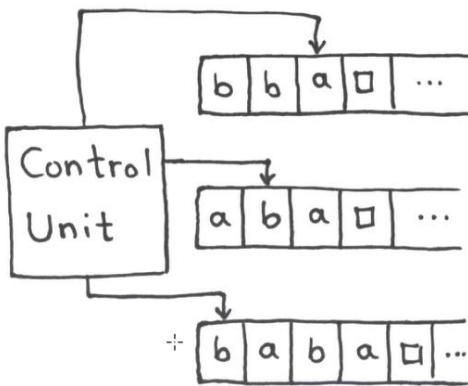
\* This TM can easily simulate an ordinary TM (just do not use S)

\* Ordinary TM can simulate a TM with stay option

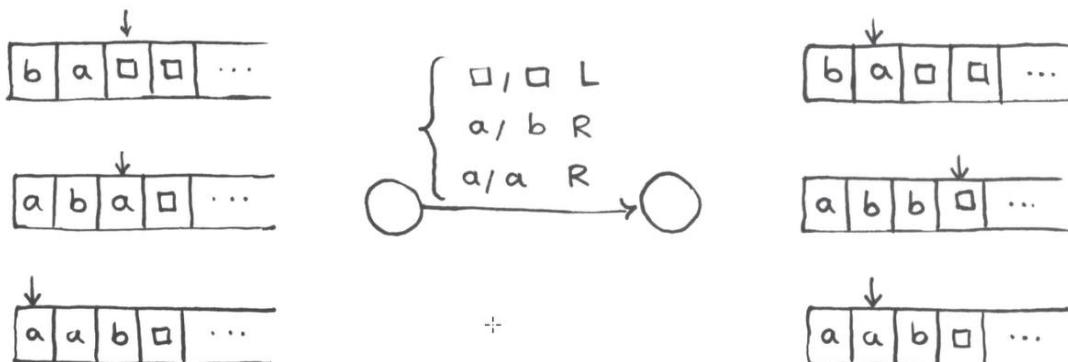
For each S transition introduce a new state and two transitions



### Multi-tape Turing Machine



- \* Transitions depend on the contents of all cells under heads
- \* Different heads can move independent
- \* Initially: Input is on the first tape and all other tapes are blank

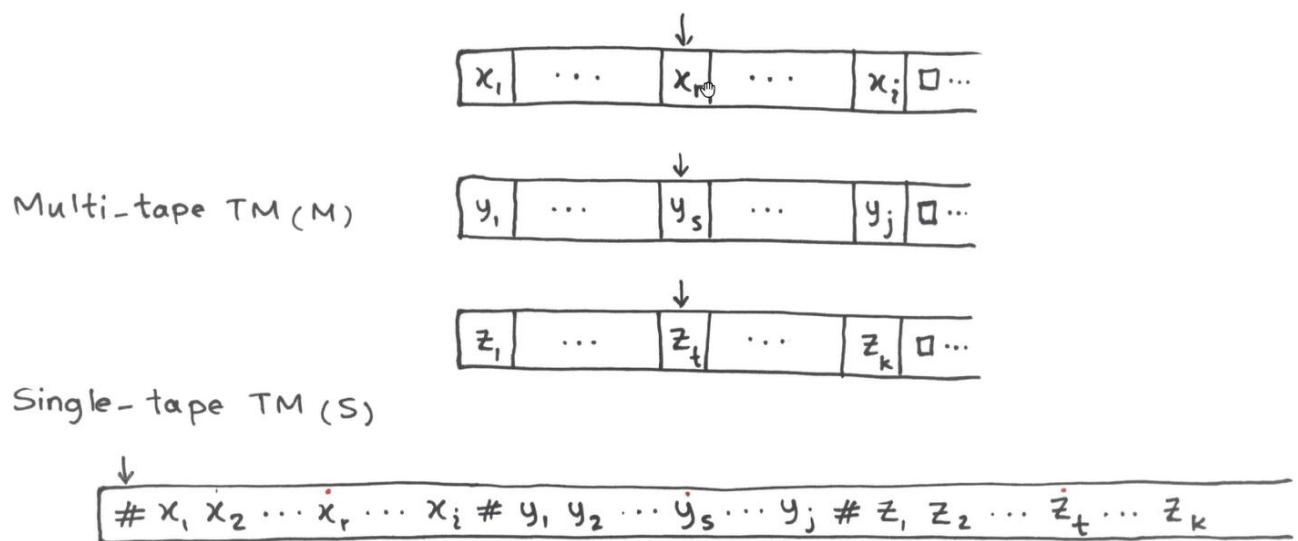


$$\delta : Q \times \Gamma^K \rightarrow Q \times \Gamma^K \times \{L, R\}^K$$

Multiple tapes are convenient

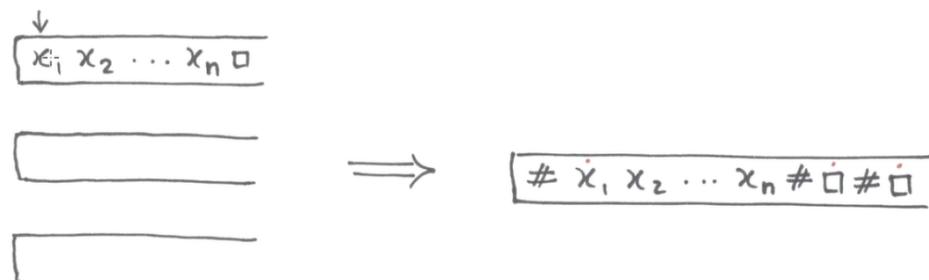
One tape can serve as temporary storage

How to simulate a multi-tape TM on a single-tape TM?

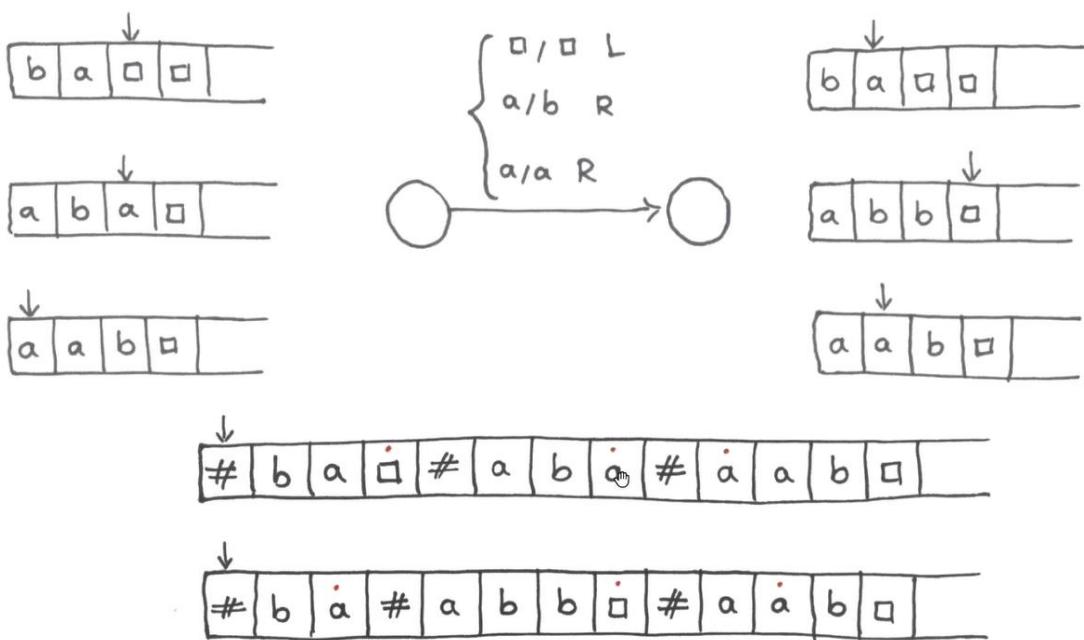


$$\Gamma' = \Gamma \cup \{t \mid t \in \Gamma\} \cup \{\#\}$$

Initialization:



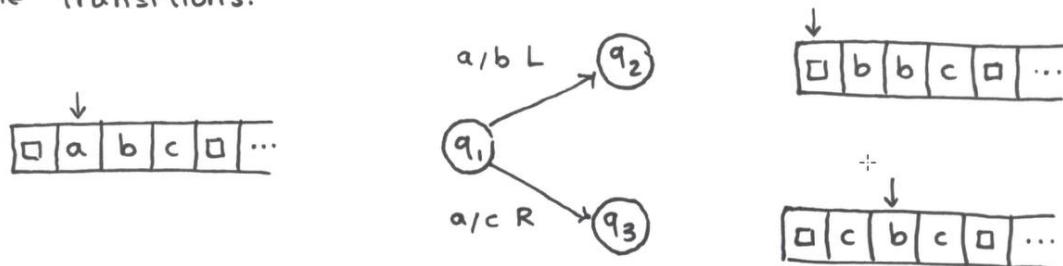
We simulate a move of M on single-tape TM S like this:



Make a pass over tape to find dotted elements. Update state/tape accordingly. If M reaches accept/reject state, S accepts/rejects.

## Non-deterministic Turing Machine

At each step of execution the machine can choose from a set of possible transitions.



$M$  accepts  $w$  iff at least one of its computations accepts

$M$  rejects  $w$  iff all of its computations reject

$$L = \{ w \in \{0,1\}^* \mid w \text{ is a binary encoding of a composite number} \}$$

- ① Non-deterministically choose two binary numbers  $p, q > 1$  such that  $2 \leq |p|$  and  $|q| \leq |w|$

Write them on tape after  $w$

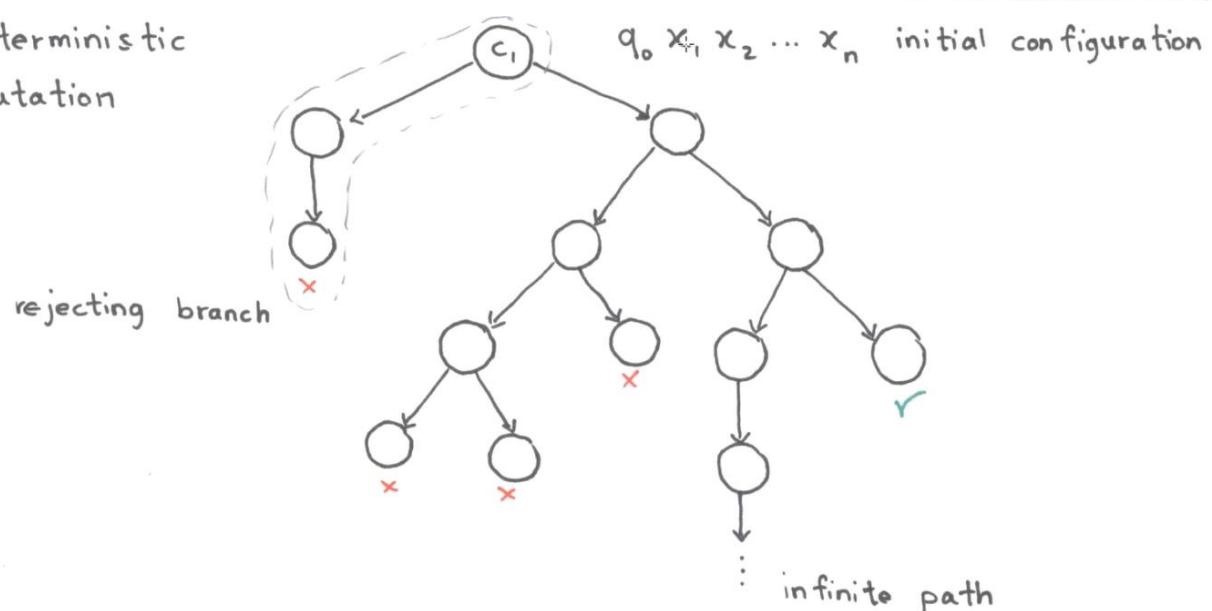
$$\underbrace{\begin{array}{cccccc} 1 & 1 & 0 & 0 & 1 & 1 \end{array}}_w \# \underbrace{\begin{array}{ccc} 1 & 1 & 1 \end{array}}_p \# \underbrace{\begin{array}{cc} 1 & 1 & 1 & 1 \end{array}}_q \square$$

- ② Multiply  $p$  and  $q$  and put the answer on tape in place of  $p$  and  $q$

$$\underbrace{\begin{array}{cccccc} 1 & 1 & 0 & 0 & 1 & 1 \end{array}}_w \# \underbrace{\begin{array}{ccccc} 1 & 0 & 1 & 1 & 1 \end{array}}_{p \times q} \square$$

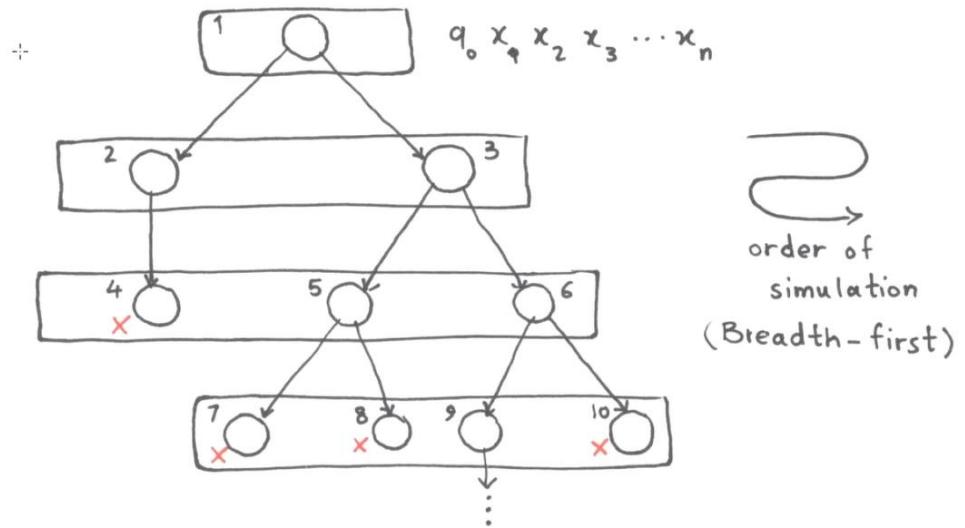
- ③ Compare  $w$  and  $p \times q$ . If equal, accept. Else, reject.

Non-deterministic  
Computation

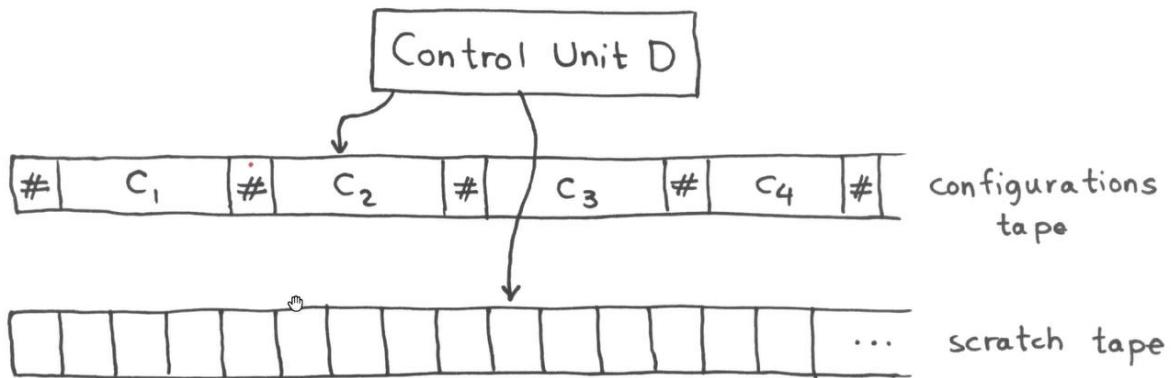


Deterministic TM D simulates the non-deterministic TM N

(5)



2-tape DTM D simulates N



Build into finite control of D is the knowledge of what moves N has for each state and input

- D examines state and input symbol of current configuration (right after #)
- If the state of the current config is the accept state of N, D accepts and stops
- D copies K copies of the current configuration to the scratch tape
- D applies one non-deterministic move of N to each copy
- D copies the new configurations from scratch tape back to end of tape 1 and clears the scratch tape
- D returns to the marked current configuration, erases the mark, and marks the next configuration
- D returns to first step. If there is no next configuration, reject.

Simulation can take exponentially more time than non-deterministic TM.

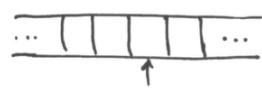
It is not known whether or not this exponential slowdown is necessary.

1-way tape Turing machine (standard)



(6)

2-way tape Turing machine

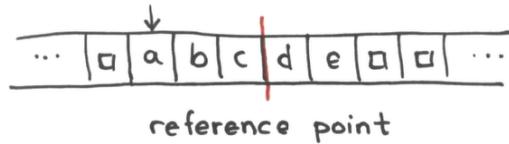


+

2-way tape Turing machine simulates 1-way tape Turing machine (trivial)

1-way tape Turing machine simulates 2-way tape Turing machine

2-way tape Turing machine



1-way tape Turing machine with two tracks

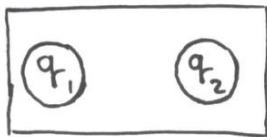
Right Part

Left Part

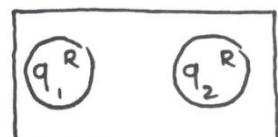
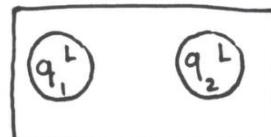
|   |   |   |   |   |     |
|---|---|---|---|---|-----|
| # | d | e | q | o | ... |
| # | c | b | a | o | ... |



2-way tape Turing Machine



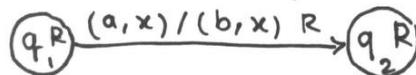
1-way tape Turing Machine



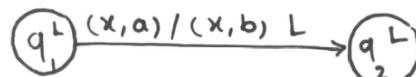
2-way tape Turing Machine



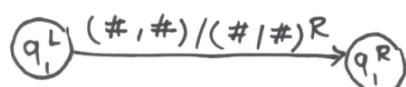
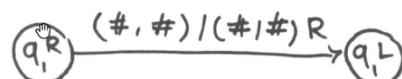
1-way tape Turing Machine

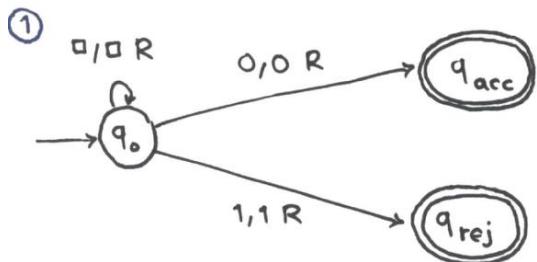


for every symbol x



At the border:





حل بیت و کم

$$\Sigma = \{0,1\}$$

input: ε

Turing machine may not halt

Inputs to Turing machine can be divided into 3 types:



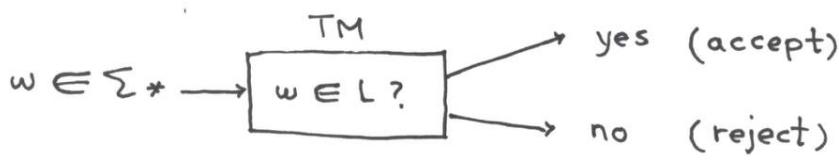
Accept



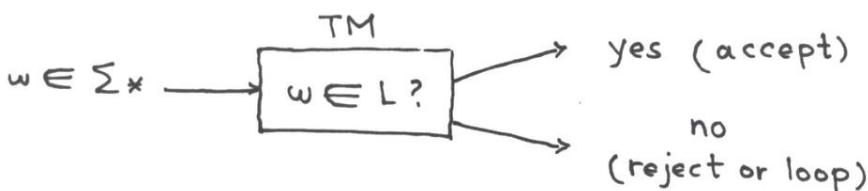
Reject



Infinite Loop

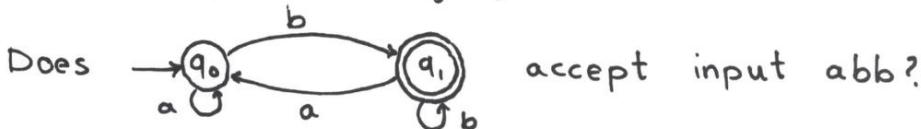


L is decidable  
(recursive)



L is recognizable  
(recursively enumerable)

Some examples of languages that are decidable by algorithms



We can formulate this question as a language

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

Is  $A_{DFA}$  decidable?

One possible way to encode a DFA  $D = (Q, \Sigma, \delta, q_0, F)$  and input  $w$

$$\underbrace{(q_0, q_1)}_{Q} \underbrace{(a, b)}_{\Sigma} \underbrace{((q_0, a, q_0)(q_0, b, q_1)(q_1, a, q_0)(q_1, b, q_1))}_{\delta} \underbrace{(q_0)(q_1)}_{F} \underbrace{(abb)}_w$$

Pseudo code

TM Description

On input  $\langle D, w \rangle$  where  $D = (Q, \Sigma, \delta, q_0, F)$

Set  $q \leftarrow q_0$ .

For  $i \leftarrow 1$  to  $\text{length}(w)$

$$q \leftarrow \delta(q, w_i)$$

If  $q \in F$  accept, else reject

On input  $\langle D, w \rangle$  where  $D$  is a DFA,  
 $w$  is a string

Simulate  $D$  on input  $w$

If simulation ends in an accept state, accept; else reject

Turing Machine Details :

Check input is in correct format (transition function complete, no duplicate)

Perform simulation (very high-level)

Put markers on start state of D and first symbol of w

Until marker for w reaches last symbol :

Update both markers

If state marker is on accepting state, accept; else reject

Conclusion:  $A_{\text{DFA}}$  is decidable

$((q_0, q_1)(a, b)((q_0, a, q_0), (q_0, b, q_1), (q_1, a, q_0), (q_1, b, q_1)) (q_0)(q_1)) (abb)$

$((q_0, q_1)(a, b)((q_0, a, q_0), (q_0, b, q_1), (q_1, a, q_0), (q_1, b, q_1)) (q_0)(q_1)) (abb)$

$((q_0, q_1)(a, b)((q_0, a, q_0), (q_0, b, q_1), (q_1, a, q_0), (q_1, b, q_1)) (q_0)(q_1)) (abb)$

$((q_0, q_1)(a, b)((q_0, a, q_0), (q_0, b, q_1), (q_1, a, q_0), (q_1, b, q_1)) (q_0)(q_1)) (abb)$

$A_{\text{NFA}} = \{ \langle N, w \rangle \mid N \text{ is an NFA that accepts input } w \}$

Convert N to a DFA  $D$ , using the conversion procedure

Run TM M for  $A_{\text{DFA}}$  on input  $\langle D, w \rangle$

If M accepts, accept; else reject

$A_{\text{NFA}}$  is decidable

$A_{\text{REX}} = \{ \langle R, w \rangle \mid R \text{ is a regular expression that generates } w \}$

Convert R to an NFA N using the conversion procedure

Run TM for  $A_{\text{NFA}}$  on input  $\langle N, w \rangle$

$A_{\text{REX}}$  is decidable

$\text{MIN}_{\text{DFA}} = \{ \langle D \rangle \mid D \text{ is a minimal DFA} \}$

Run DFA minimization algorithm from Lecture 9 on D

If every pair of states is distinguishable, accept; else reject

$\text{MIN}_{\text{DFA}}$  is decidable

(3)

$$EQ_{DFA} = \{ \langle D_1, D_2 \rangle \mid D_1 \text{ and } D_2 \text{ are DFAs and } L(D_1) = L(D_2) \}$$

Run DFA minimization algorithm on  $D_1$  to obtain a minimal DFA  $D'_1$

Run DFA minimization algorithm on  $D_2$  to obtain a minimal DFA  $D'_2$

If  $D'_1 = D'_2$  accept; else reject

$EQ_{DFA}$  is decidable

$$E_{DFA} = \{ \langle D \rangle \mid D \text{ is a DFA and } L(D) \text{ is empty} \}$$

Run the TM  $S$  for  $EQ_{DFA}$  on input  $\langle D, D' \rangle$  where  $D'$  is any DFA that accepts no input



If  $S$  accepts, accept, else reject

$E_{DFA}$  is decidable

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

Convert  $G$  into Chomsky Normal Form  $G'$

Run Cocke-Younger-Kasami algorithm on  $\langle G', w \rangle$

If CYK algorithm finds a parse tree, accept; else reject

$A_{CFG}$  is decidable

$$EQ_{CFG} = \{ \langle G_1, G_2 \rangle \mid G_1, G_2 \text{ are CFGs and } L(G_1) = L(G_2) \}$$

What is the difference between  $EQ_{DFA}$  and  $EQ_{CFG}$ ?

To decide  $EQ_{DFA}$  we minimize both DFAs

But there is no method that, given a CFG or PDA, produce a unique equivalent minimal CFG or PDA

Note. Since CFGs are not closed under intersection or complement, we cannot use symmetric difference



if  $L_1 = L_2 \Rightarrow$  symmetric difference  
is empty

$EQ_{CFG}$  is undecidable.

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

(4)

- Convert  $G$  into Chomsky Normal Form
- Mark all non-terminals  $A$  which have some rule  $A \rightarrow a$
- Repeat until no new non-terminal are marked:
  - Mark the non-terminal  $A$  if there is a rule  $A^0 \rightarrow BC$  such that  $B$  and  $C$  are already marked
- If  $S$  is marked ( $L(G) \neq \emptyset$ ) then reject, else accept

## Encodings:

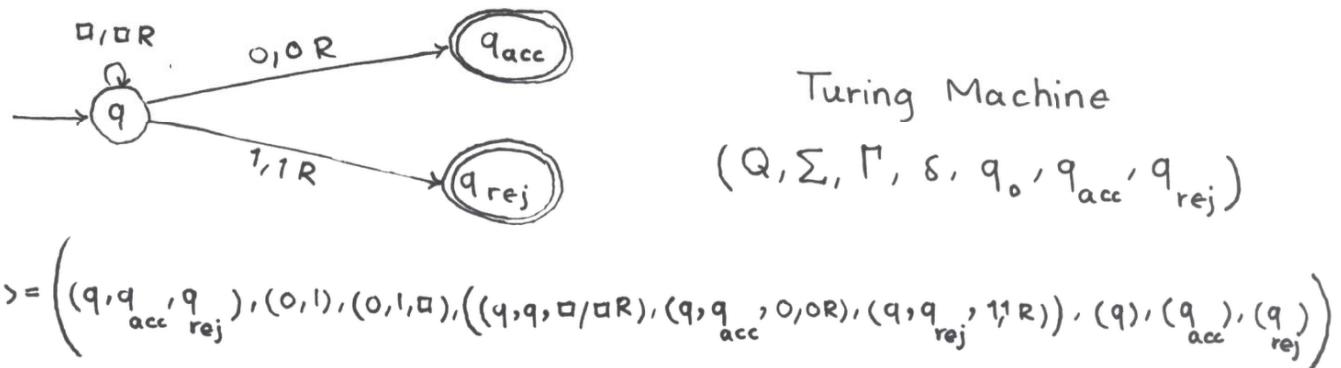
We represent the string encoding of 0 with angle brackets:  $\langle 0 \rangle$

Example.  $\text{MIN}_{\text{DFA}} = \{ \langle D \rangle \mid D \text{ is a minimal DFA} \}$

We represent the string encoding of  $0_1, \dots, 0_n$  with  $\langle 0_1, \dots, 0_n \rangle$

Example.  $A_{\text{CFG}} = \{ \langle G, w \rangle \mid G \text{ is a context-free grammar that generates } w \}$

How can we encode a Turing machine?



We can encode any Turing machine  $M$  in binary alphabet  $\{0, 1\}$

Analogy: Program source code

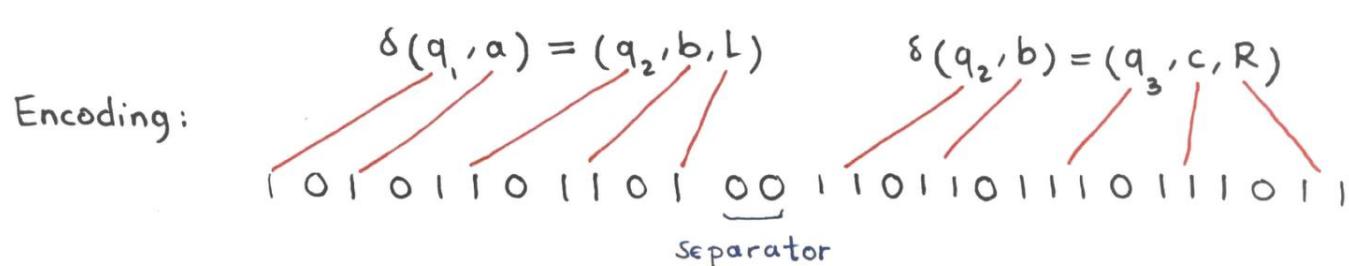
- Data given to a program is encoded using  $\{0, 1\}$
- Program itself is also represented in  $\{0, 1\}$  on disk

One possible encoding:

| Alphabet encoding:               | State encoding:             | Transition Encoding:                                           |
|----------------------------------|-----------------------------|----------------------------------------------------------------|
| Symbols: $a b c d \dots$         | States: $q_1 q_2 q_3 \dots$ | $\delta(q_1, a) = (q_2, b, L)$                                 |
| Encodings: $1 11 111 1111 \dots$ | Encodings: $1 11 111 \dots$ | $10101101101$<br>Head move:<br>$L: 1 \quad R: 11$<br>separator |

Machine Encoding:

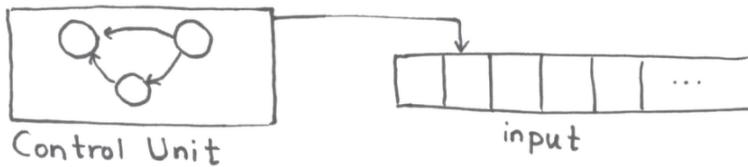
Transitions:



The set of Turing machines forms a language, each string of the language is the binary encoding of a Turing machine ②

$$L = \{ 010100101, 0010010010111, 1110100111100101, \dots \}$$

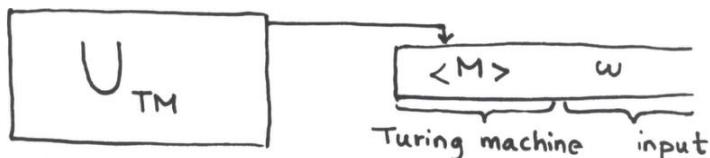
↑                       ↑                       ↑  
Turing machine 1     Turing machine 2     Turing machine 3



So far we only built "dedicated machine"

- Only run one program
- Specified by transitions on states

Can TM be a general-purpose computer? Can we have a TM that simulates an arbitrary TM on arbitrary input?



Universal Turing machine  $U_{TM}$  takes  $\langle M, w \rangle$  as input (where  $M$  is a Turing machine and  $w$  is a string) and simulates running  $M$  on  $w$

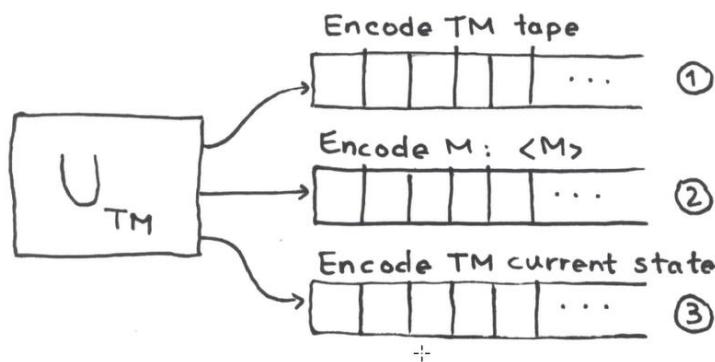
$U_{TM}$  = "On input  $\langle M, w \rangle$ , where  $M$  is a TM and  $w \in \Sigma^*$

Run  $M$  on  $w$

If  $M$  accepts  $w$ ,  $U_{TM}$  accepts  $\langle M, w \rangle$ .

If  $M$  rejects  $w$ ,  $U_{TM}$  rejects  $\langle M, w \rangle$ ."

Note. If  $M$  loops on  $w$ ,  $U_{TM}$  loops as well.



- $U_{TM}$  simulates the TM
- Read tape 1
  - Read tape 3
  - Consult tape 2 what to do
  - Write tape 1
  - Move head tape 1
  - Move head tape 3

Since  $U_{TM}$  is a Turing machine, it has a language.

(3)

What is the language of the universal Turing machine?

The language of the universal Turing machine :  $A_{TM}$

$$A_{TM} = L(U_{TM}) \quad \text{Acceptance language for Turing machine}$$
$$= \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}$$

Useful facts:  $\langle M, w \rangle \in A_{TM} \Leftrightarrow M \text{ accepts } w$

Recall : Definition

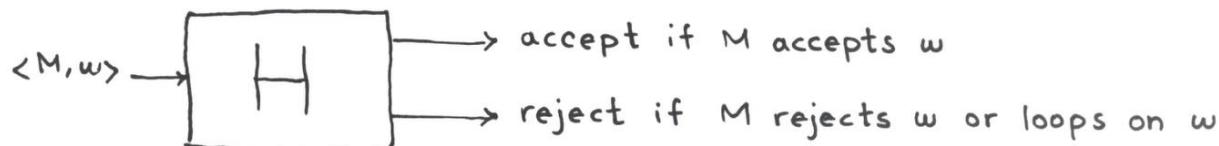
A Turing machine  $M$  is said to **recognize** a language  $L$  if  $L = L(M)$

A Turing machine  $M$  is said to **decide** a language  $L$  if  $L = L(M)$  and  $M$  halts on every input.

Because  $A_{TM} = L(U_{TM})$  we know  $A_{TM}$  is Turing-recognizable.

Turing's Theorem: The language  $A_{TM}$  is undecidable.

Proof by contradiction: Suppose  $A_{TM}$  is decidable, then some TM  $H$  decides it

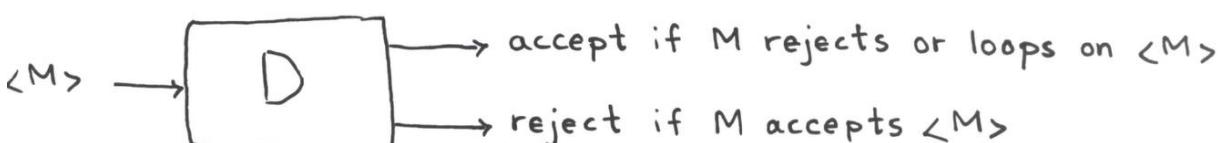


Construct a new TM  $D$  (that uses  $H$  as a subroutine)

On input  $\langle M \rangle$  (i.e. description of a Turing machine  $M$ )

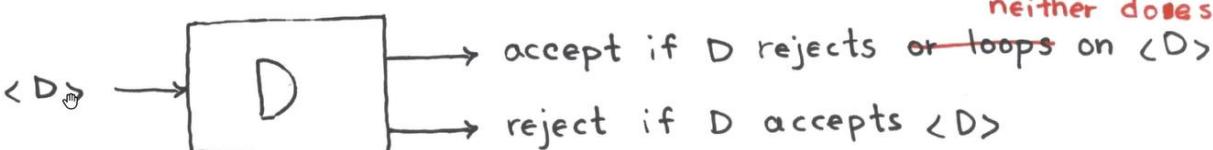
1. Run  $H$  on input  $\langle M, \langle M \rangle \rangle$

2. Output the opposite of  $H$ : if  $H$  accepts,  $D$  rejects, if  $H$  reject,  $D$  accepts



What happens if  $M = D$ ?

$H$  never loops indefinitely,  
neither does  $D$



If D rejects  $\langle D \rangle$ , then D accepts  $\langle D \rangle$

If D accepts  $\langle D \rangle$ , then D rejects  $\langle D \rangle$

Contradiction! D cannot exist, H cannot exist!

Assume  $A_{TM}$  is decidable  $\Rightarrow$  Then there are TM H and D  
 $\Rightarrow$  But D cannot exist

Proof using Diagnolization

Write an infinite table for pairs  $(M, w)$

|                | $\epsilon$ | 0   | 1    | 00   | ... | all possible inputs |
|----------------|------------|-----|------|------|-----|---------------------|
| M <sub>1</sub> | acc        | rej | loop | rej  | ... |                     |
| M <sub>2</sub> | loop       | acc | rej  | loop | ... |                     |
| M <sub>3</sub> | :          | :   | :    | :    | :   |                     |
| M <sub>4</sub> | :          |     |      |      |     |                     |
| ⋮              |            |     |      |      |     |                     |

Only look at those w that describe Turing machine:

|                | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | ... |
|----------------|-----------------------|-----------------------|-----------------------|-----|
| M <sub>1</sub> | acc                   | loop                  | rej                   | ... |
| M <sub>2</sub> | rej                   | acc                   | loop                  | ... |
| M <sub>3</sub> | acc                   | rej                   | rej                   | ... |
| M <sub>4</sub> | rej                   | acc                   | loop                  | ... |
| ⋮              | ⋮                     | ⋮                     | ⋮                     | ⋮   |

If H exists, table of outputs of H:

|                | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\dots \langle D \rangle \dots$ |
|----------------|-----------------------|-----------------------|-----------------------|---------------------------------|
| M <sub>1</sub> | acc                   | rej                   | rej                   | ... acc ...                     |
| M <sub>2</sub> | rej                   | rej                   | acc                   | ... rej ...                     |
| M <sub>3</sub> | acc                   | rej                   | acc                   | ... rej ...                     |
| M <sub>4</sub> | rej                   | acc                   | rej                   | ... acc ...                     |
| ⋮              | ⋮                     | ⋮                     | ⋮                     | ⋮                               |
| D              | rej                   | acc                   | rej                   | ?                               |

D on  $\langle M_i \rangle$  accepts iff  $M_i$  on  $\langle M_i \rangle$  rejects

D on  $\langle M_i \rangle$ : opposite of  $M_i$  on  $M_i$

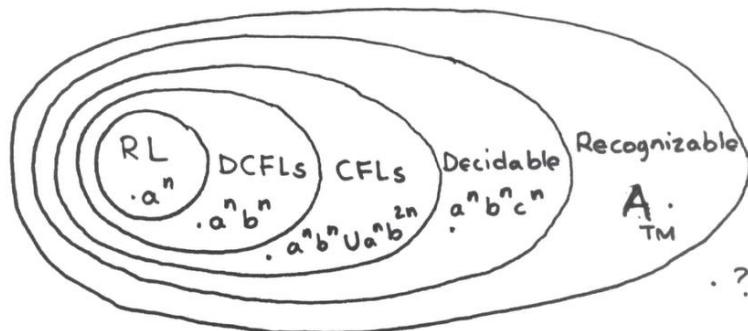
So, D on  $\langle D \rangle$  will accept iff D on  $\langle D \rangle$  rejects  $\Rightarrow$  contradiction!

①

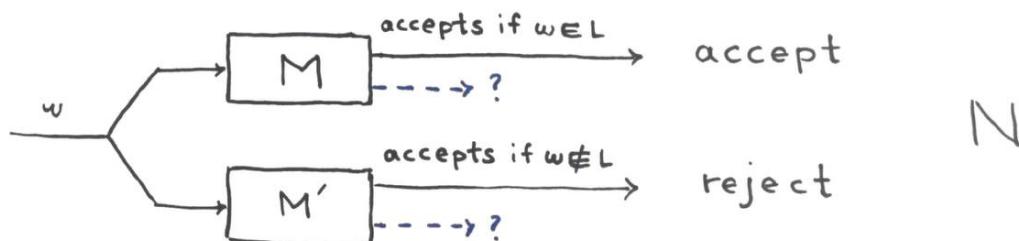
لے بنیت دسم

لے بنیت دسم

Language hierarchy:

Language  $A_{TM}$  is recognizable but not decidable

How about languages that are not recognizable?

Theorem. If  $L$  and  $\bar{L}$  are both recognizable, then  $L$  is decidableSuppose  $M$  accepts  $L$  and  $M'$  accepts  $\bar{L}$ Turing machine  $N$  decides  $L$ :

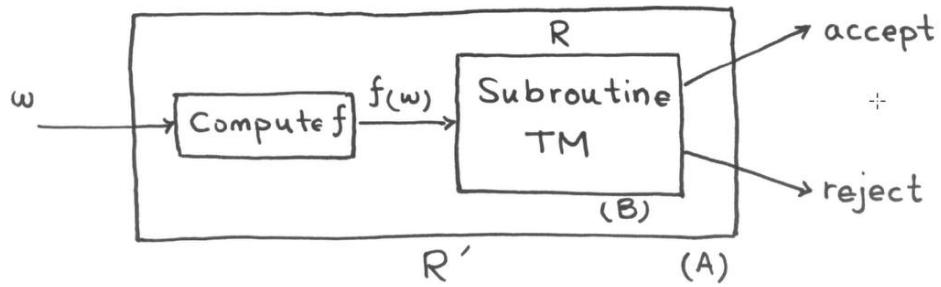
- 1) Simulate  $M$  on input  $w$ . If  $M$  accepts,  $N$  accepts
- 2) Simulate  $M'$  on input  $w$ . If  $M'$  accepts,  $N$  rejects

Important detail:  $M$  and  $M'$  must run in parallel, not sequentiallyRun in parallel: Perform one step of  $M$ , followed by one step of  $M'$   
alternating between two machinesClaim. Language  $\overline{A_{TM}}$  is not recognizable.Proof. We know  $A_{TM}$  is recognizable.If  $\overline{A_{TM}}$  was also, then  $A_{TM}$  would be decidableBut Turing's theorem says  $A_{TM}$  is not decidable.

Suppose you have a program  $R$  that solves problem  $B$

(2)

Now we want to solve a problem  $A$ . If you can reduce  $A$  to  $B$   
then you can solve problem  $A$ , using  $R$  as a subroutine

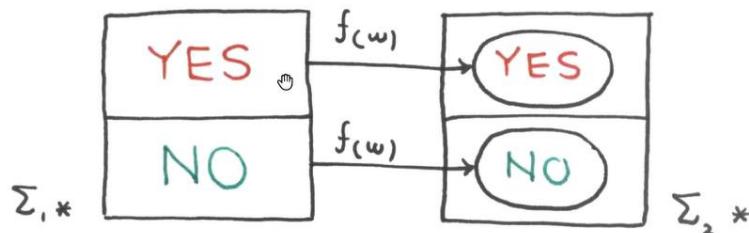


$R'$ : on input  $w$ :

- Transform the input  $w$  to  $f(w)$
- Run machine  $R$  on  $f(w)$
- If  $R$  accepts  $f(w)$ , then  $R'$  accepts  $w$
- If  $R$  rejects  $f(w)$ , then  $R'$  rejects  $w$

### Defining Reductions

A reduction from  $A$  to  $B$  is a function  $f: \Sigma_1^* \rightarrow \Sigma_2^*$  such that  
for any  $w \in \Sigma_1^*$ ,  $w \in A$  iff  $f(w) \in B$



Every  $w \in A$  maps to some  $f(w) \in B$

Every  $w \notin A$  maps to some  $f(w) \notin B$

$f$  does not need to be injective or surjective

Example.

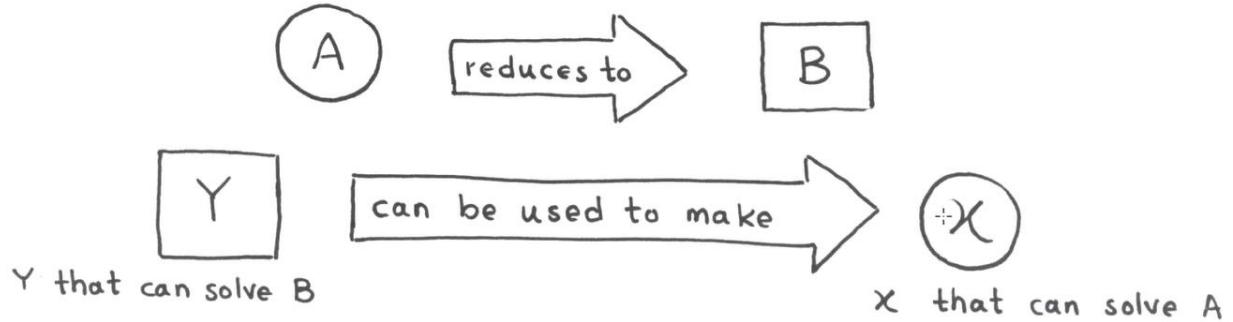
$$A_{DFA} = \{ \langle D, w \rangle \mid D \text{ is a DFA that accepts input } w \}$$

$$A_{NFA} = \{ \langle N, w \rangle \mid N \text{ is an NFA that accepts input } w \}$$

$A_{NFA}$  reduces to  $A_{DFA}$  (by converting NFA to DFA)

(3)

Problem A reduces to problem B iff a solver for B can be used to solve problem A



$$A \leq B$$

then this one is "easy" to

If this one is "hard"

If this one is "easy"  
then this one is "hard" too

$$A \leq B \text{ and } B \text{ decidable} \implies A \text{ decidable}$$

$$A \leq B \text{ and } A \text{ undecidable} \implies B \text{ undecidable}$$

### Halting Problem

Input: Program P in some programming language

Output: true if P terminates, false if P runs forever

halts ("2+2") True

halts ("def f(n):

```
    if n==0 : return 1
    else: return n*f(n-1)
f(10)")
```

True

halts ("def f(n):

```
    if n==0 : return 1
    else: return n*f(n-1)
```

False

f(10.5)")

halts ("x = randint()

while (x>1):

```
    if (x % 2 ==0) : x = x/2
    else x = 3*x + 1")
```

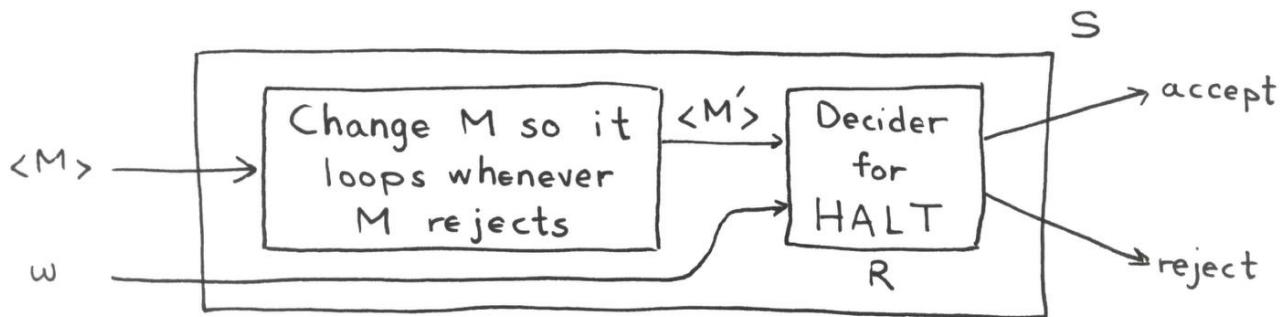
Unknown
(Collatz Conjecture)

$$\text{HALT}_{\text{TM}} = \{\langle M, w \rangle \mid M \text{ is a TM that halts on input } w\} \quad (4)$$

We reduce  $A_{\text{TM}}$  to the language  $\text{HALT}_{\text{TM}}$ :  $A_{\text{TM}} \leq \text{HALT}_{\text{TM}}$

Suppose for a proof by contradiction that  $\text{HALT}_{\text{TM}}$  is decidable.

Then it has a TM R that decides it.



If  $M$  never rejects, then  $M$  accepts iff  $M$  halts on  $w$

$$\langle M, w \rangle \in A_{\text{TM}} \text{ iff } \langle M, w \rangle \in \text{HALT}_{\text{TM}}$$

How can we transform  $M$ ?

Assume  $\Sigma = \{a, b, c\}$ . We convert  $\xrightarrow{q_{\text{rej}}}$  to  $\xrightarrow{\text{a/a R, b/b R, c/c R}}$

Machine S is a decider for  $\text{HALT}$  (contradiction)!

$a/a \text{ R}$   
 $b/b \text{ R}$   
 $c/c \text{ R}$

$S = \text{"On input } \langle M, w \rangle$

Transform  $M$  into  $M'$  by making  $M$  loop instead of rejecting

Run R on  $\langle M', w \rangle$

If R accepts  $\langle M', w \rangle$ , then S accepts  $\langle M, w \rangle$

If R rejects  $\langle M', w \rangle$ , then S rejects  $\langle M, w \rangle$ .

①

Both  $A_{TM}$  and  $\text{HALT}_{TM}$  are undecidable

There is no way to decide whether a TM will accept or eventually terminate.

Both  $A_{TM}$  and  $\text{HALT}_{TM}$  are recognizable

Reduce A to B:

$$A \leq B$$

then A is "easy"

if B is "easy"

If A is "hard"

then B is "hard"

"easy": decidable, recognizable

"hard": undecidable, unrecognizable

$$A_{TM}^{\epsilon} = \{ \langle M \rangle \mid M \text{ is a TM that accepts input } \epsilon \}$$

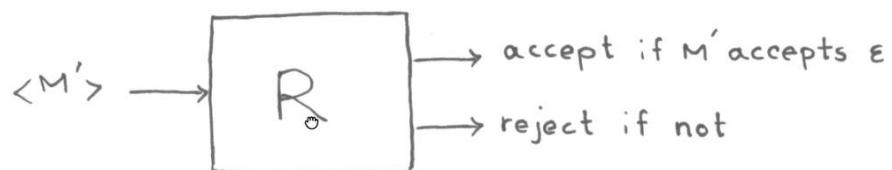
Undecidable.

Intuitive reason: to know whether M accepts  $\epsilon$  require simulating M

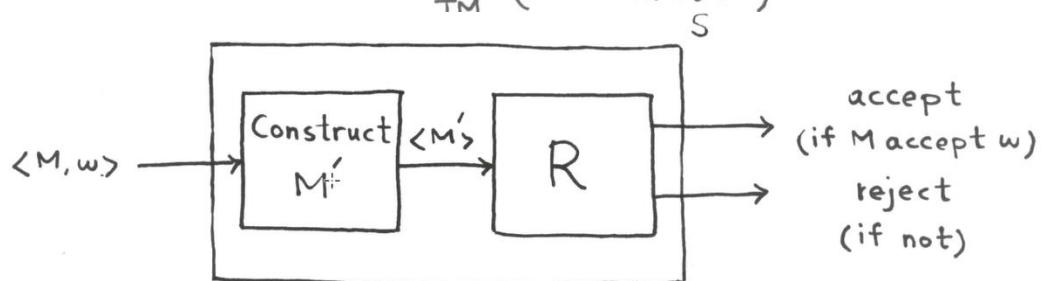
But we need to know whether M halts

$$A_{TM} \leq A_{TM}^{\epsilon}$$

Suppose R decides  $A_{TM}^{\epsilon}$  (proof by contradiction)

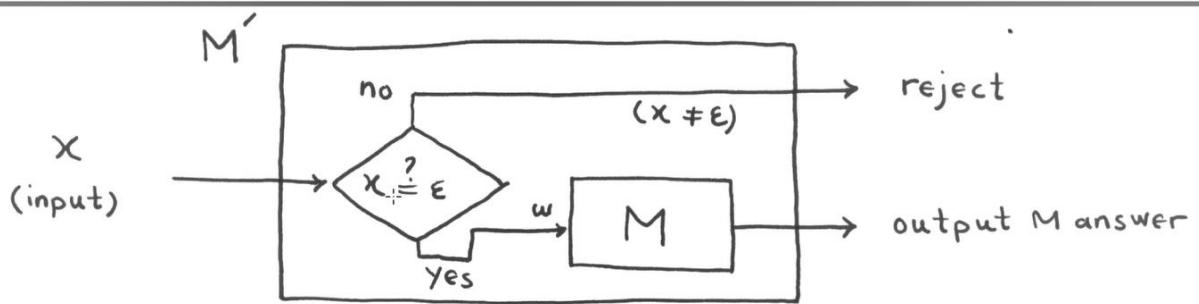


We show we can use R to decide  $A_{TM}$  (contradiction)

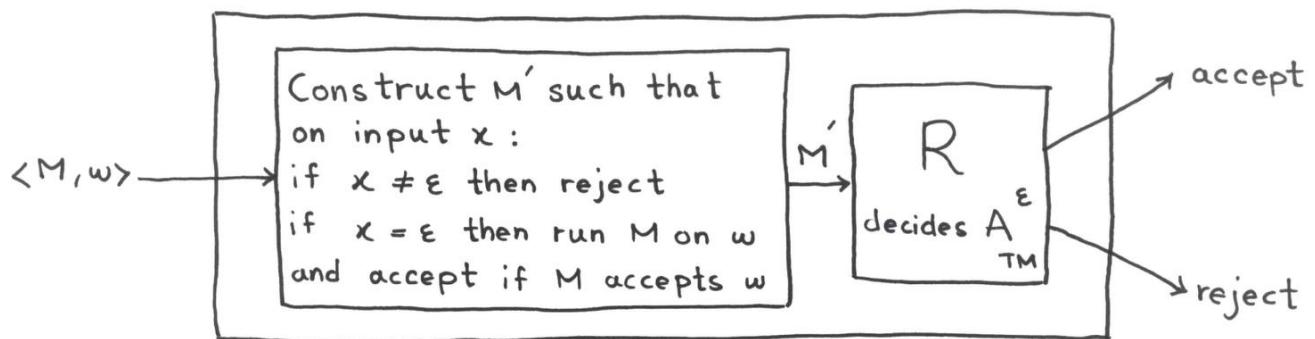


$M'$  accepts  $\epsilon$  if and only if  $M$  accept  $w$

②



- If  $M$  accepts  $w$   
input is  $\epsilon$  and  $M'$  accepts it
- If  $M$  rejects  $w$ 
  - case  $x = \epsilon$  :  $M'$  rejects
  - case  $x \neq \epsilon$  :  $M'$  rejects
- If  $M$  loops on  $w$ 
  - case  $x = \epsilon$  :  $M'$  loops
  - case  $x \neq \epsilon$  :  $M'$  rejects



$$\text{EMPTY}_{\text{TM}} = \{ \langle M \rangle \mid M \text{ is a TM such that } L(M) = \emptyset \}$$

Unrecognizable

$$\overline{A_{TM}} \leq \text{EMPTY}_{\text{TM}}$$

Suppose  $R$  recognizes  $\text{EMPTY}_{\text{TM}}$  (Proof by contradiction)



Either  $L(M') = \emptyset$  or  $L(M') = \{w\}$  (when  $M$  accepts  $w$ ) . . . . . (3)

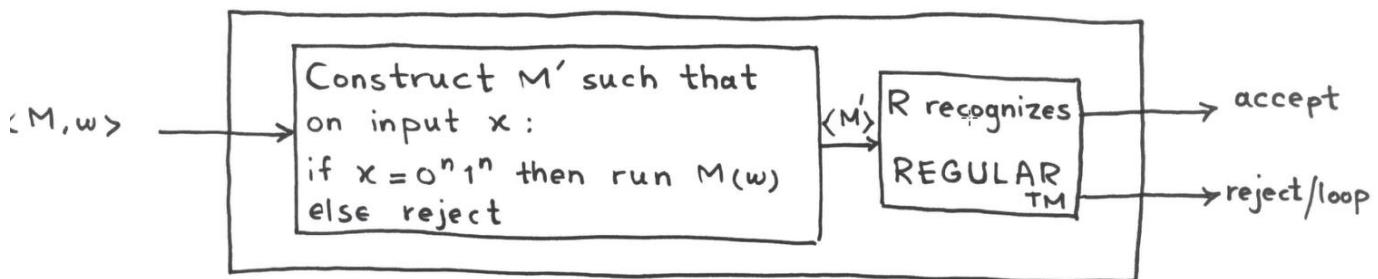
$$\begin{aligned} \langle M, w \rangle \in \overline{A_{TM}} &\iff \langle M, w \rangle \notin A_{TM} \\ &\iff M \text{ does not accept } w \\ &\iff L(M') = \emptyset \\ &\iff M' \in \text{EMPTY}_{TM} \end{aligned}$$

$\text{REGULAR}_{TM} = \{\langle M \rangle \mid M \text{ is a TM such that } L(M) \text{ is regular}\}$

Given a program, is it equivalent to some DFA?

Unrecognizable

$$\overline{A_{TM}} \leq \text{REGULAR}_{TM}$$



$$\langle M, w \rangle \in \overline{A_{TM}} \iff M' \text{ accepts } \{0^n 1^n\}$$

$$\langle M, w \rangle \notin \overline{A_{TM}} \iff M' \text{ accepts nothing}$$

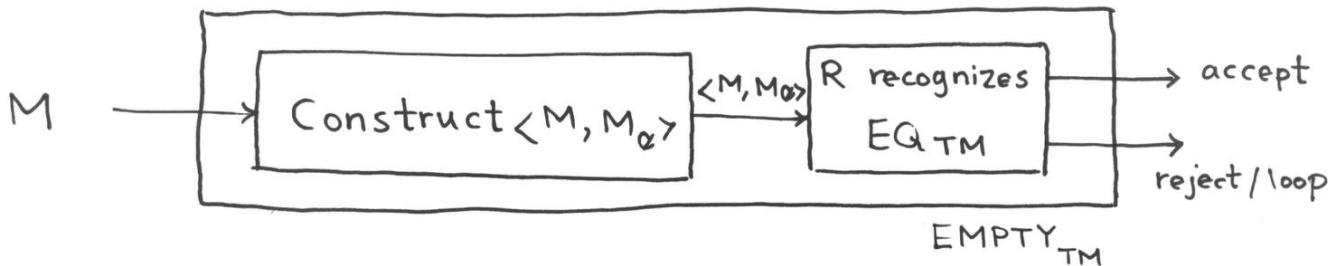
$$\text{So we conclude: } \langle M, w \rangle \notin \overline{A_{TM}} \iff M' \in \text{REGULAR}_{TM}$$

$\text{EQ}_{TM} = \{\langle M_1, M_2 \rangle \mid M_1 \text{ and } M_2 \text{ are TMs such that } L(M_1) = L(M_2)\}$

Unrecognizable

$$\text{EMPTY}_{TM} \leq \text{EQ}_{TM}$$

Let  $M_\alpha$  be some TM with no path from start to accept state



Software is the most complex artifact that human beings build routinely. Analyzing programs is extremely hard!

$L_1 = \{ \langle M, w \rangle \mid M \text{ is a TM that on input } w, \text{ tries to move its head past the left end of the tape at some point} \}$  Undecidable

$L_2 = \{ \langle M, w \rangle \mid M \text{ is a TM that on input } w, \text{ moves its head left at some point} \}$  Decidable

$L_1$  is undecidable.  $A_{TM} \leq L_1$

Idea. Construct a TM  $M'$  which simulates  $M$  but never goes to the left of the starting symbol.

If  $M$  halts, then proceed to go all the way left, until it goes beyond the starting symbol.

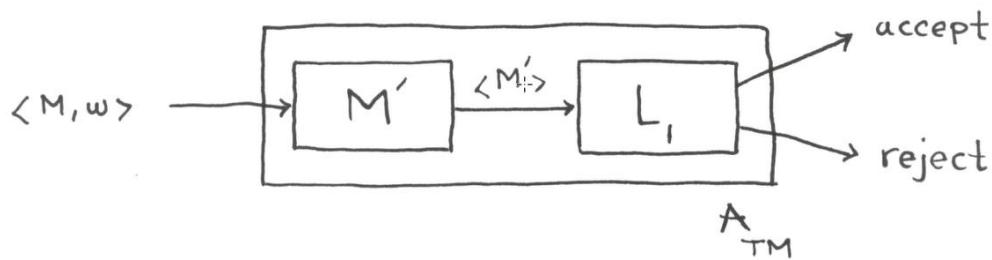
Construct  $M'$ :

Copy  $w$  one position right and put special symbol # before it



Simulate  $M$  on  $w$  with two changes:

- ① If head moves to # but has not yet accepted move head back to right and stay in the same state
- ② If  $M$  accepts, enter a special state where the head just keeps moving left



## $L_2$ is decidable

Does a TM  $M$  on input  $w$  ever try to move its head left?

On input  $\langle M, w \rangle$ :

Simulate  $M$  on  $w$  for  $|w|$  steps.

If  $M$  hasn't made any left moves, then head must be pointing to first  $\square$  after  $w$ .

Search in the state diagram of  $M$  by following all transitions that have  $\square$  as input. Search is similar to the one we did for  $\epsilon$ -closure.

We are only interested in these transitions, because if the machine never moves left, the only available input symbol will be  $\square$ .

If none of the transitions move left, accept, otherwise reject.

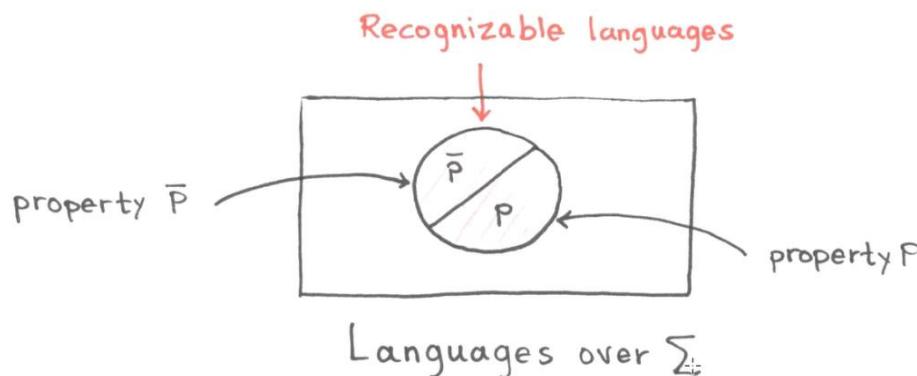
Every time we wish to prove  $L$  is undecidable

Do we need to reduce  $A_{TM}$  to it? Isn't there any shortcut?

Rice theorem states that Turing machines cannot test whether another Turing machine satisfies a non-trivial property.

Rice Theorem: Let  $P$  any subset of the Turing recognizable class such that  $P$  and its complement are both non-empty.

Then the language  $L_p = \{\langle M \rangle \mid L(M) \in P\}$  is undecidable



P<sub>1</sub> : L is empty

- ✓ L =  $\emptyset$
- ✗ L = { University }
- ✗ L = { Alan, Turing }

P<sub>2</sub> : L is regular

- ✓ L =  $\emptyset$
- ✓ L = {  $a^n \mid n \geq 0$  }
- ✗ L = {  $a^n b^n \mid n \geq 0$  }

P<sub>3</sub> : L has size 3

- ✗ L =  $\emptyset$
- ✗ L = { University }
- ✓ L = { Alan, Turing, Machine }

P<sub>4</sub> : L has size at least 0

True for all languages

P<sub>5</sub> : L is accepted by some Turing machine

True for all Turing-recognizable languages

①

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ

جَلَّ بِسْمُهُ وَنَعِمْ

PCP: Post Correspondence Problem (پوسٹ کورسپونڈنس پریبل)

Classic undecidable problem that was introduced by Emil Post in 1946

An undecidable problem that does not directly involve Turing machines

Dominoes:



We get as many of each type as we need.

Goal: Find a finite sequence of dominoes such that the top and bottom strings are the same.

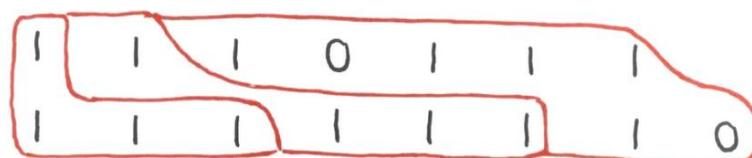


Does a solution exist? The problem is undecidable.

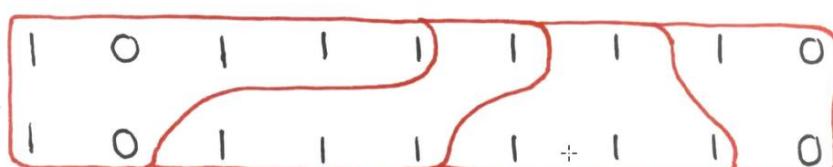
Another instance of PCP:

|   | A         | B       |
|---|-----------|---------|
| ① | 1         | 1 1 1 0 |
| ② | 1 0 1 1 1 | 1 0     |
| ③ | 1 0       | 0       |

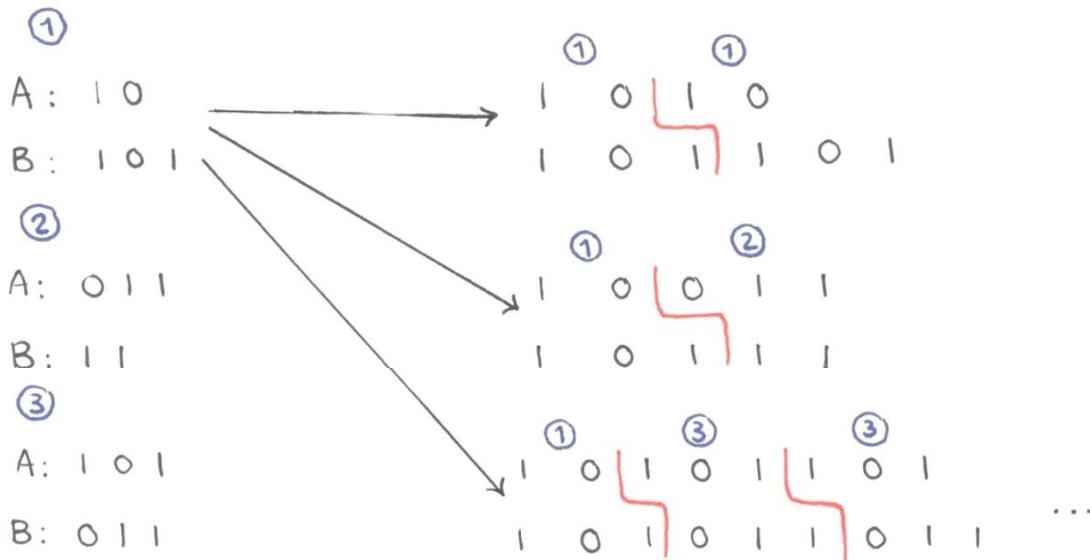
A non-solution: ① ① ②



A solution: ② ① ① ③



|   | A     | B     |
|---|-------|-------|
| ① | 1 0   | 1 0 1 |
| ② | 0 1 1 | 1 1   |
| ③ | 1 0 1 | 0 1 1 |



How can we prove that PCP is undecidable?

Configuration :

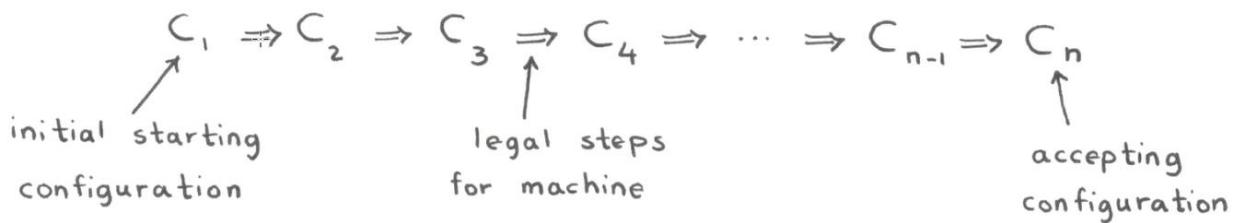


A sequence of configurations:

$$\dots \Rightarrow C_7 \Rightarrow C_8 \Rightarrow C_9 \Rightarrow \dots$$

$\underbrace{\quad\quad\quad}_{\text{step in the computation}}$

An "accepting computation history"



A "rejecting computation history"

$$C_1 \Rightarrow C_2 \Rightarrow \dots \Rightarrow C_n$$

Same, but the last configuration has a rejecting state

A computation history is a finite sequence.

If the machine does not halt  $\Rightarrow$  NO HISTORY

On a given input:

- Deterministic machine: at most one history

+  
- Non-deterministic machine: may have many histories

In this lecture we only consider deterministic TMs.

There is either  $\left\{ \begin{array}{l} \text{an accepting history} \\ \text{a rejecting history} \\ \text{no history / loop} \end{array} \right\}$  on a particular input.

We can represent a computation history with a string.

#  $q_{r_0} | 0 | \# | q_{r_4} 0 + 1 \# \dots \# 0 | 1 | q_{\text{acc}} | \#$

↑  
initial state at  
the left end of tape

↑  
this is an accepting history

**Theorem.** The Post Correspondence Problem is undecidable

Proof Sketch:

Reduce  $A_{\text{TM}}$  to PCP

$A_{\text{TM}} \leq \text{PCP}$

If  $A_{\text{TM}}$  accepts then

- $M$  computes on  $w$  and accepts
- there is a finite computation history which describes the computation of  $M$  on  $w$

We encode  $\langle M, w \rangle$  into a PCP instance.

There is a solution to PCP iff there is an accepting computation

If we could decide this instance of PCP,

then we could decide  $A_{\text{TM}}$  +

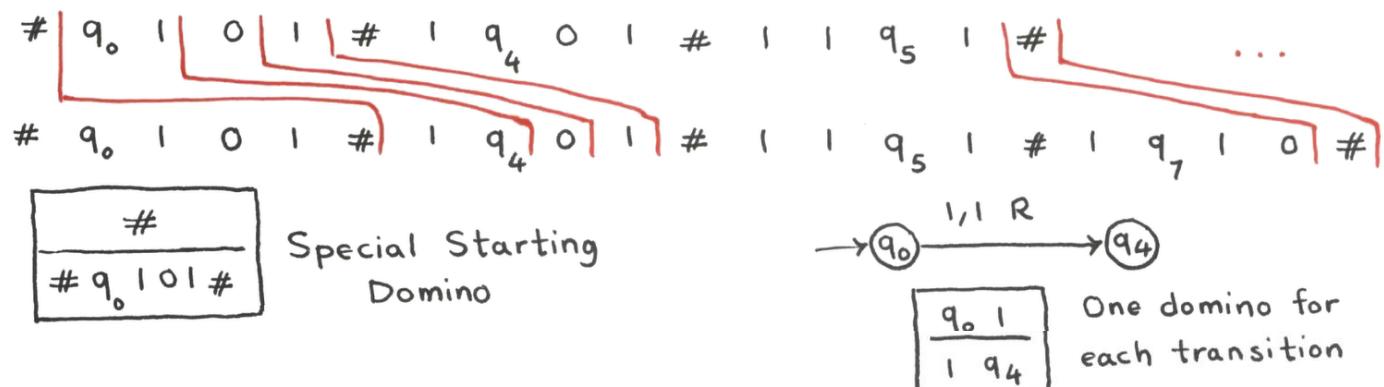
But we cannot decide  $A_{\text{TM}}$ !

$$\langle M, w \rangle \Rightarrow \boxed{-} \quad \boxed{-} \quad \dots \quad \boxed{-}$$

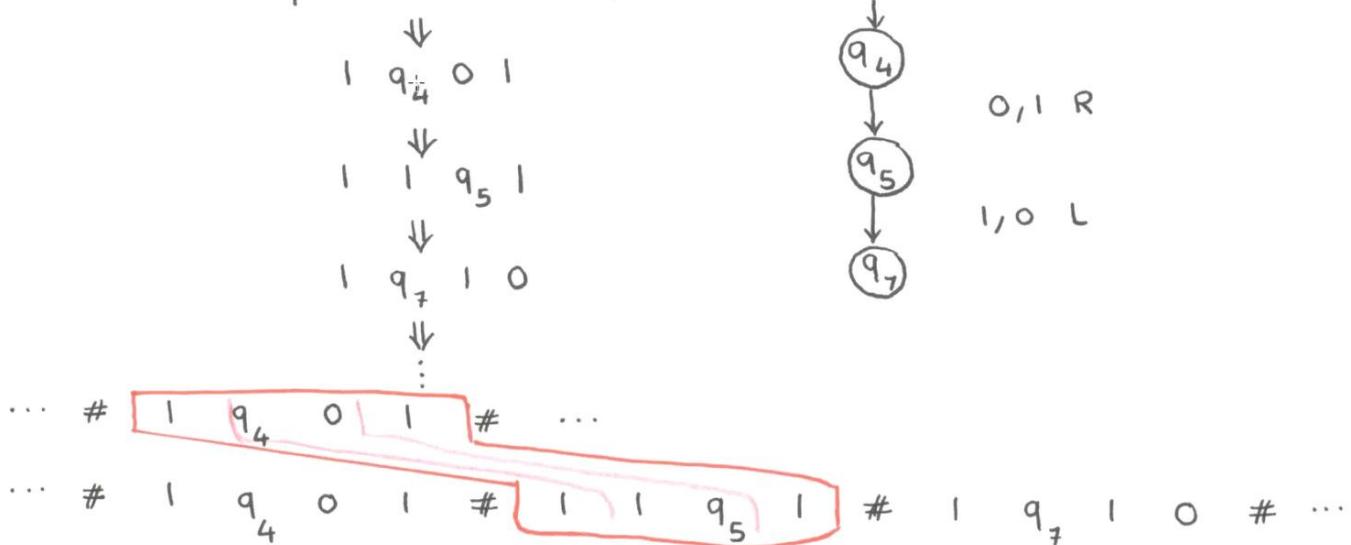
construct an instance of dominos

Solution to this instance of PCP  $\Rightarrow$  you have an accepting history

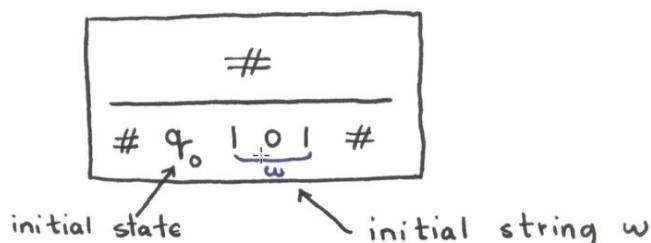
A solution will look like this :



Computation History



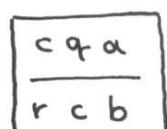
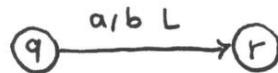
Special Starting Domino :



RIGHT MOVE

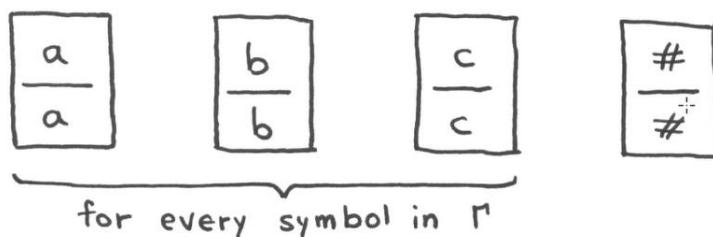


LEFT MOVE

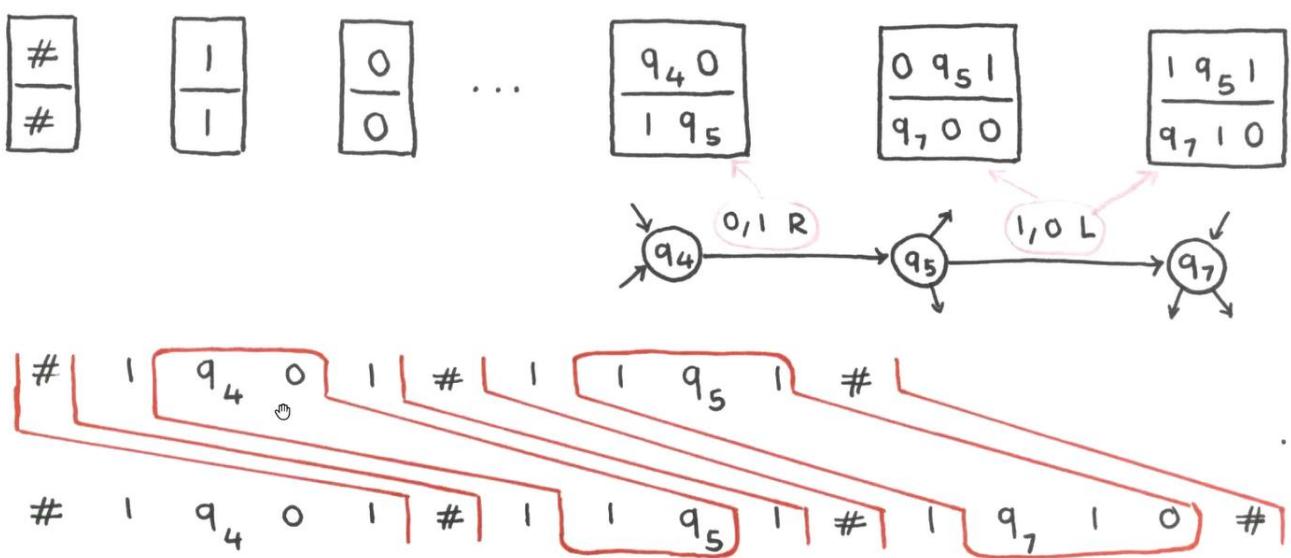


For every  $c \in \Gamma$ , Add one domino like this

Dominoes to "copy" the tape:



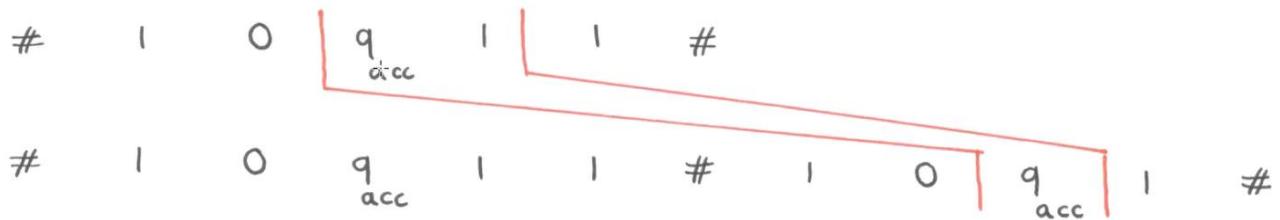
Used to copy the other parts of the tape that are not next to head



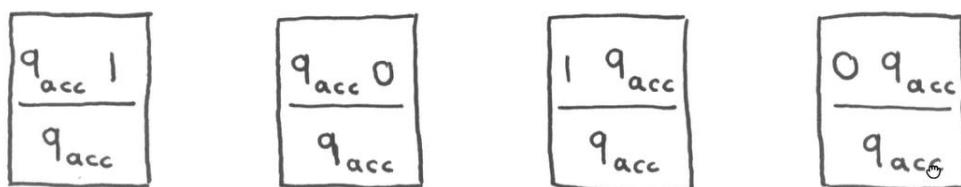
Question: How do we accept?

Answer: Complete the match!

Add special dominoes to allow  $q_{acc}$  to "eat" the symbols on tape

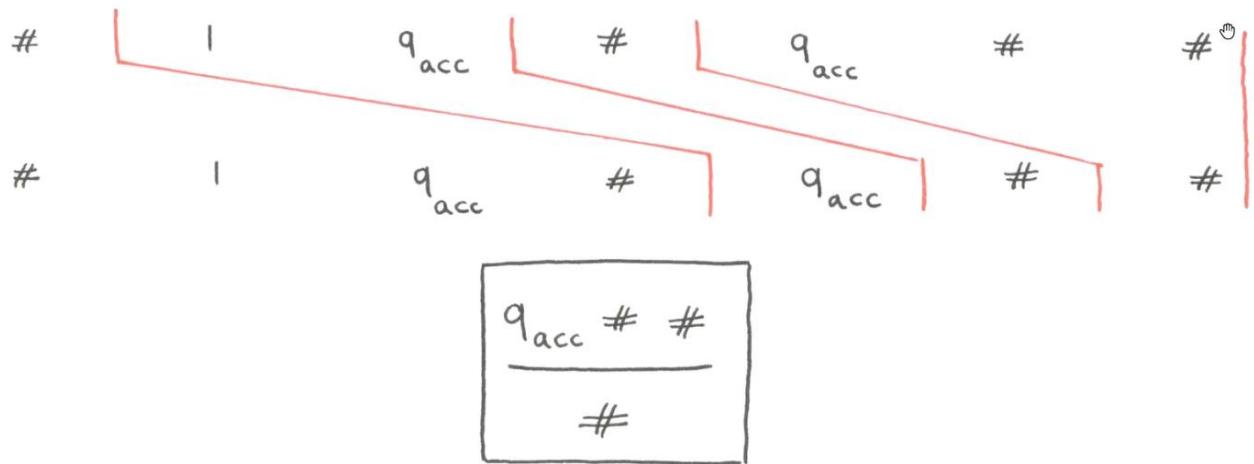


For every symbol in  $\Gamma$ :



Eventually, nothing remains except  $q_{acc}$ . (6)

Add a domino to finish the match.



---

Review of the proof:

If you can find a solution to this instance of PCP,  
then you have found a legal accepting computation history,  
in which machine M accepts string w.

- \* Does a solution exist?
- \* If you can decide the answer, then you can decide  
whether M accepts w

We know  $A_{TM}$  is undecidable!

---

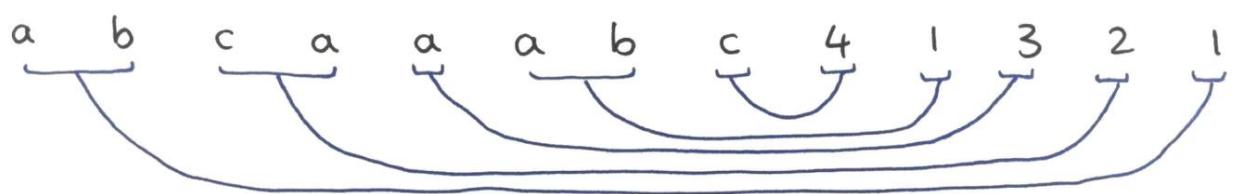
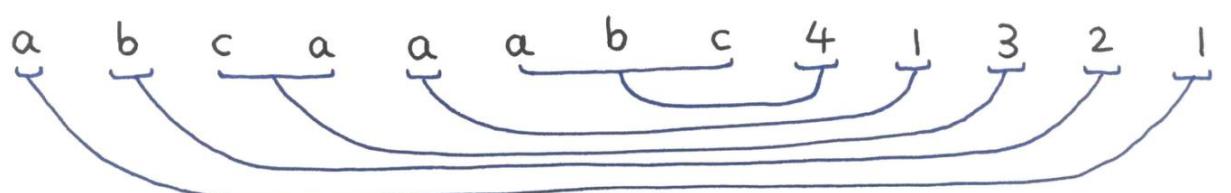
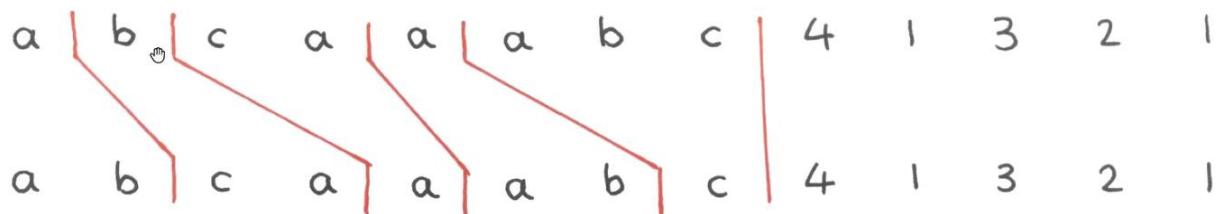
Undecidability of ambiguity problem for CFG

We illustrate the reduction using the following example:

| A     | B  | S → A   B                    |
|-------|----|------------------------------|
| ① a   | ab | A → aA1   bA2   caA3   abcA4 |
| ② b   | ca | a1   b2   ca3   abc4         |
| ③ ca  | a  | B → abB1   caB2   aB3   cB4  |
| ④ abc | c  | ab1   ca2   a3   c4          |

1, 2, 3, 4 : record which domino was used

There are two distinct leftmost derivations for the string abc a a a b c 4 1 3 2 1 because this instance of PCP has solutions. ⑦

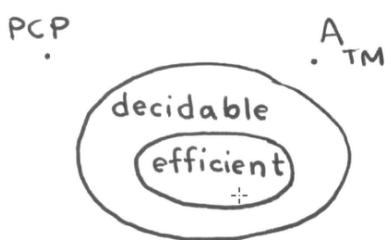


Undecidable Problems :

We cannot find solutions in any finite amount of time.

Decidable Problems :

We can solve them, but it may take a very long time.



We typically look for efficient algorithms.

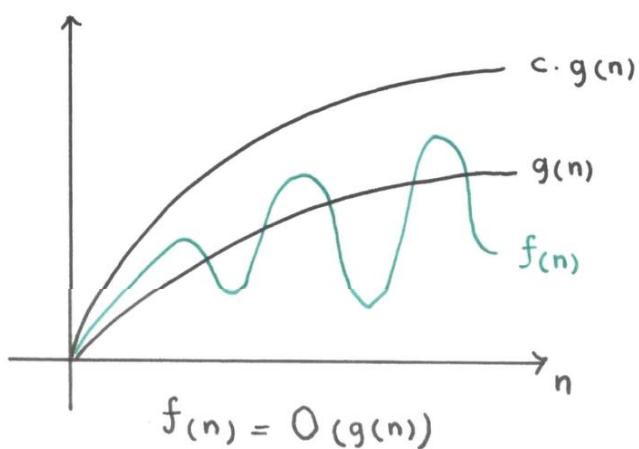
Running time depends on the input.

For longer inputs, we should allow more time.

Efficiency is measured as a function of input size.

## Asymptotic Analysis

The asymptotic behavior of a function  $f(n)$  refers to the growth of  $f(n)$  as  $n$  gets large. We typically ignore small values of  $n$ , since we are usually interested in estimating how slow the program is on large input.



$$\text{Eventually } |f(n)| \leq c \cdot |g(n)|$$

We measure run-time in terms of input size "n"

E.g. Run-time grows on the order of input size  $O(n)$

$$5n^3 + 23n^2 + 6n \lg n + 186 = O(n^3)$$

Running time of Turing machine  $M$  is the function  $t_M(n)$ : ②

$t_M(n)$ : maximum number of steps that  $M$  takes on any input of length  $n$

$$L = \{ w \# w \mid w \in \{a, b\}^* \}$$

abaab  $\neq$  abaab

~~a~~baab  $\neq$  abaab

~~a~~baab  $\neq$  ~~a~~baab

~~a~~baab  $\neq$  ~~a~~baab

$M$ : On input  $x$ , until you reach  $\#$   $O(n)$  times

Read and cross off first a or b before  $\#$

" " " after  $\#$

If mismatch, reject

If all symbols except  $\#$  are crossed off, accept  $O(n)$  steps

running time:  $O(n^2)$

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

$M$ : On input  $x$ ,

Check input is of the form  $0^* 1^*$   $O(n)$  steps

Until everything is crossed off:  $O(n)$  times

Cross off the leftmost 0 +

Cross off the following 1

If everything is crossed off, accept

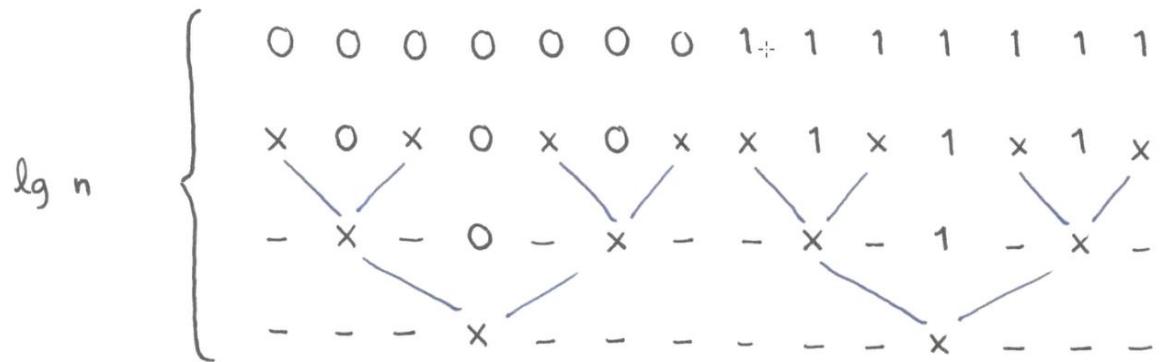
$O(n)$  steps

running time:  $O(n^2)$

Can we find a faster algorithm to decide  $L$ ?

Cross off every other 0, cross off every other 1

(3)



If the parities of 0's and 1's don't match, reject

(odd) 0 0 0 0 0 0 0 0 1 1 1 1 1 (odd)

(odd) X 0 X 0 X 0 X X 1 X 1 X (even)

M : On input  $x$ ,

Check input is of the form  $0^*1^*$

$O(n)$  steps

Until everything is crossed off:

$O(\lg n)$  times

Find parity of the number of 0s

{

Find parity of the number of 1s

$O(n)$  steps

If the parities don't match, reject

Cross off every other 0 and every other 1

}

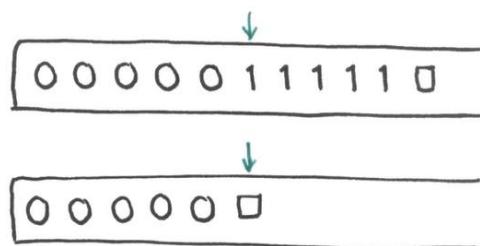
If everything is crossed off, accept

$O(n)$  steps

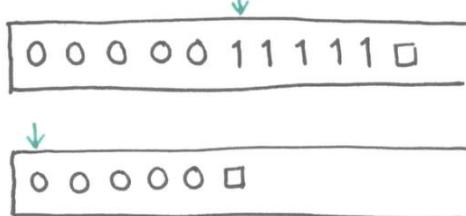
running time:  $O(n \lg n)$

What if we have a two-tape TM?

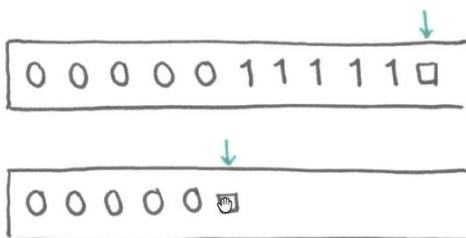
- Copy all 0's to tape 2.



- Reposition tape 2 to beginning



- Scan both tape simultaneously, make sure both heads hit  $\square$  at the same time



M: On input  $x$ ,

Check input is of the form  $0^* 1^*$   $O(n)$  steps

Copy  $0^*$  part of the input to second tape  $O(n)$  steps

Until  $\square$  is reached:

Cross off next 1 from first tape

Cross off next 0 from second tape

If both tapes reach  $\square$  simultaneously, accept

$\left. \begin{array}{l} \\ \\ \end{array} \right\} O(n) \text{ steps}$

$O(n)$  steps

running time:  $O(n)$

Running time can change depending on the computation model

1-Tape Turing machine :  $O(n \lg n)$

2-Tape Turing machine :  $O(n)$

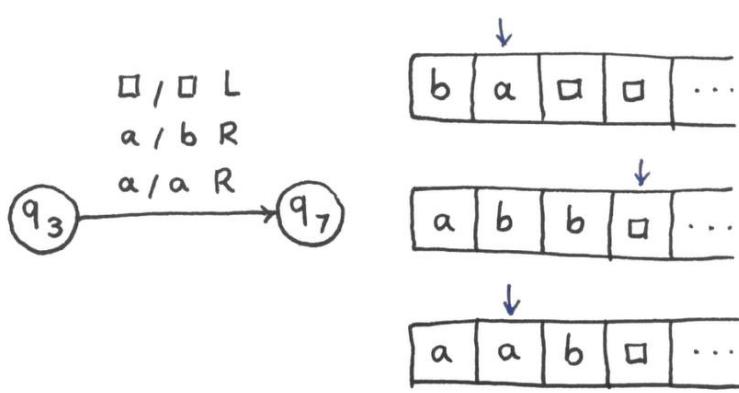
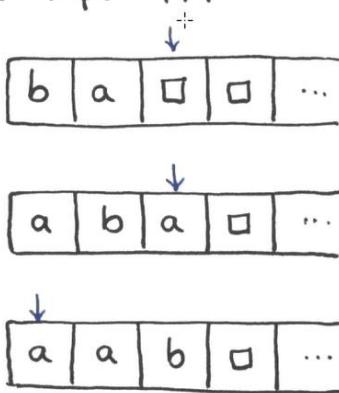
Church-Turing thesis states that any feasible model of computation is no more powerful than a TM.

However, some models of computation are more efficient in terms of computation time.

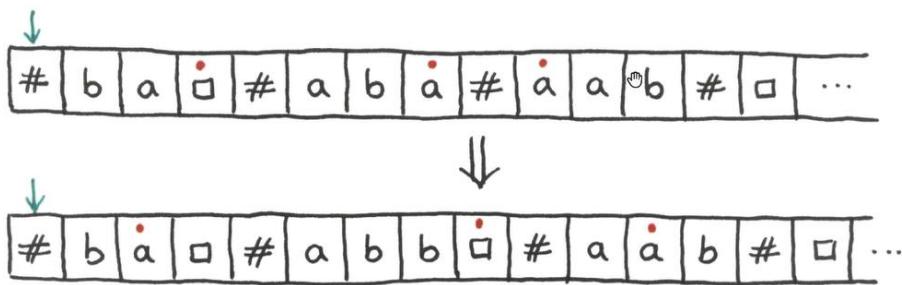
Model of computation matters for complexity!

Theorem: Let  $t(n)$  be a function such that  $t(n) \geq n$ . (5)

Then every  $t(n)$ -time multi-tape TM has an equivalent  $O(t(n)^2)$  single tape TM



1-tape TM simulates multi-tape TM: simulate each step using 2 scans over non-blank portion of tapes



How big can the non-blank portion of the multi-tape TM's tapes become?

- Initially,  $n$  for the input
- In  $t(n)$  steps, no bigger than  $t(n)$ , because that's how far the heads can travel (starts at left)

So the number of steps by the 1-tape TM is at most

$$t(n) \times c t(n)$$

+ +

Number of steps of multi-tape machine      steps taken by the scans, to emulate one step of multi-tape TM

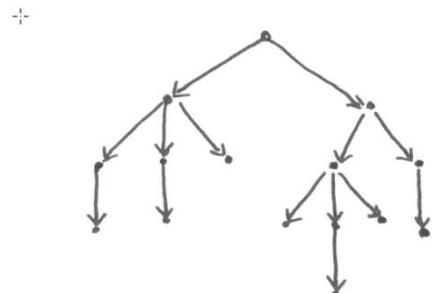
hence  $O(t^2(n))$

Theorem: Let  $t(n)$  be a function where  $t(n) \geq n$ . (6)

Then every  $t(n)$ -time non-deterministic single-tape TM has an equivalent  $2^{O(t(n))}$ -time deterministic single-tape TM.



Deterministic Computation



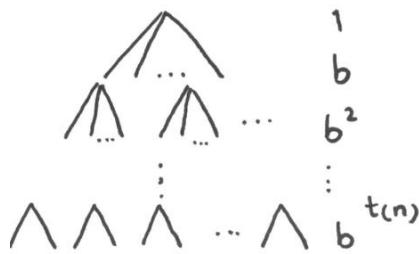
Non-deterministic Computation

We construct a deterministic TM by simulating an NTM using breadth first search. On an input of length  $n$ , every branch of the NTM's non-deterministic computation tree has a length of at most  $t(n)$ .

Every node in the tree can have at most  $b$  children, where  $b$  is the max number of legal choices given by NTM's transition function. Total number of leaves in the tree is at most  $b^{t(n)}$

---

Lemma: Total number of nodes in the tree is less than twice the maximum number of leaves.



$$\begin{aligned} &\frac{\text{Total number of nodes}}{\text{Number of leaves}} \\ &= \frac{1 + b + b^2 + \dots + b^{t(n)}}{b^{t(n)+1}} \\ &= \frac{\frac{b^{t(n)+1} - 1}{b - 1}}{b^{t(n)}} < \frac{\frac{b^{t(n)+1}}{b - 1}}{b^{t(n)}} = \frac{b}{b - 1} \leq 2 \end{aligned}$$

---

From Lemma we conclude the total number of nodes is bounded by the number of leaves  $O(b^{t(n)})$ . The time it takes to start from the root and travel down to a node is  $O(t(n))$ .

The total running time is  $O(t(n) b^{t(n)})$

$$t(n) b^{t(n)} = 2 \quad \lg(t(n)) + \lg(b^{t(n)}) = 2 \quad \lg(t(n)) + t(n) \lg b = 2 \quad O(t(n)) \quad (7)$$

Deterministic  $k$ -tape TM  $\xrightleftharpoons[\text{polynomial difference}]{}$  Single tape TM

Non-deterministic TM  $\xrightleftharpoons[\text{exponential difference}]{}$  Deterministic TM

Polynomial time algorithms are considered tractable

Exponential time algorithms are rarely usable for large input size

$$\lim_{n \rightarrow \infty} \frac{2^n}{p(n)} \quad p(n) \text{ polynomial}$$

All reasonable deterministic computational models are polynomially equivalent.

Any of them can simulate another with only a polynomial increase in running time.

①

الآن!

前途，现在！

- Polynomial functions "scale well".
  - Small changes to the size of the input do not typically induce enormous changes to the overall time.
- Exponential functions scale terribly
  - Small changes to the size of the input induce huge changes in the overall runtime.

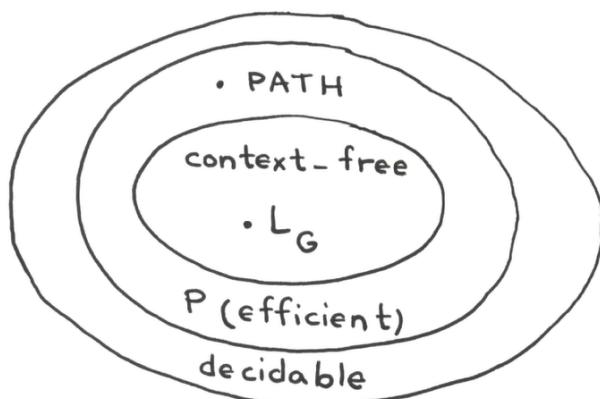
A language  $L$  can be decided efficiently iff there is a TM that decides it in polynomial time.

The time complexity class  $\text{TIME}(f(n))$  is the set of languages decidable by a single-tape TM with run-time  $O(f(n))$

$P$  is the class of languages that are decidable in polynomial time on a deterministic single-tape TM

$$P = \bigcup_k \text{TIME}(n^k)$$

- ①  $P$  is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape TM
- ②  $P$  roughly corresponds to the class of problems that are realistically solvable on a computer.



$\text{PATH} = \{ \langle G, s, t \rangle \mid \text{Graph } G$   
has a path from node  $s$  to node  $t \}$

$$L_G = \{ \omega \mid \text{CFG } G \text{ generates } \omega \}$$

$$\text{PATH} = \{ \langle G, s, t \rangle \mid \text{Graph } G \text{ has a path from node } s \text{ to node } t \}$$

M = On input  $\langle G, s, t \rangle$

where G is a graph with nodes s and t

Place a mark on node s

Repeat until no additional nodes are marked:

$O(n)$

Scan the edges of G

$O(m)$

If some edges has both marked and unmarked endpoints

Mark the unmarked endpoint

If t is marked, accept

+

(G has n vertices, m edges)

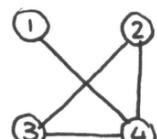
running time:  $O(mn)$

A Hamiltonian path in G is a path that visits every node exactly once

HAMPATH =  $\left\{ \langle G, s, t \rangle \mid \text{Graph } G \text{ has a Hamiltonian path from node } s \text{ to node } t \right\}$



We do not know if HAMPATH is in P, we believe it is not!



\* Clique: subset of vertices that are pairwise adjacent  
(induced subgraph is complete)

$\{1, 4\}$     $\{2, 3, 4\}$     $\{1\}$    are cliques

\* Independent set: subset of vertices that are pairwise non-adjacent

$\{1, 2\}$     $\{1, 3\}$     $\{4\}$    are independent sets

\* Vertex cover: subset of vertices that touches (covers) all edges

$\{2, 4\}$     $\{3, 4\}$     $\{1, 2, 3\}$    are vertex covers

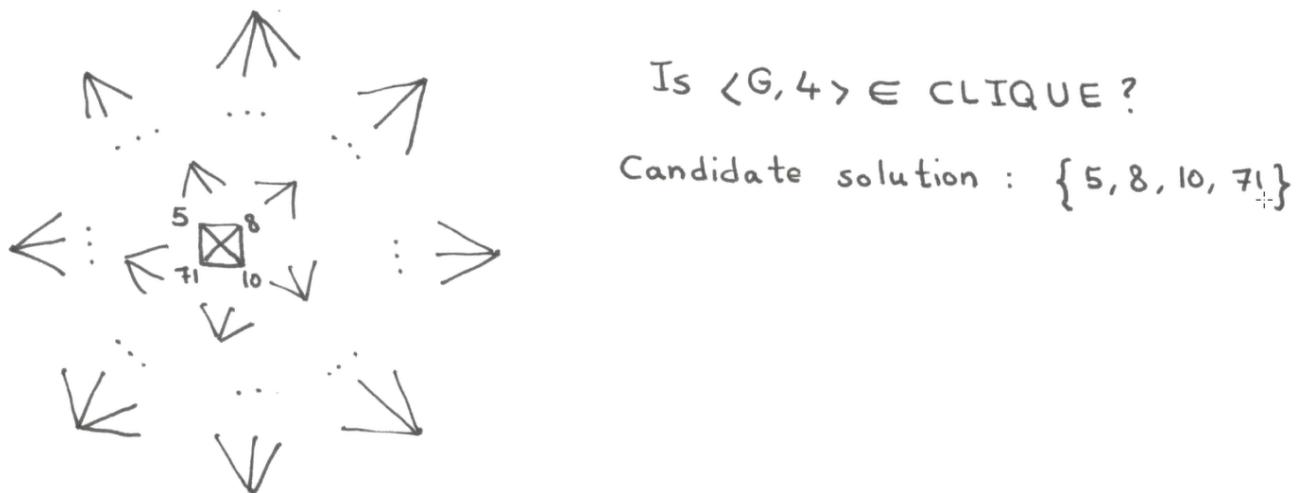
$\text{CLIQUE} = \left\{ \langle G, k \rangle \mid G \text{ is a graph having a clique of } k \text{ vertices} \right\}$  ③

$\text{INDEPENDENT-SET} = \left\{ \langle G, k \rangle \mid G \text{ is a graph having an independent set of } k \text{ vertices} \right\}$

$\text{VERTEX-COVER} = \left\{ \langle G, k \rangle \mid G \text{ is a graph having a vertex cover of } k \text{ vertices} \right\}$

What do these problems have in common?

- Given a candidate solution, we can quickly check if it is valid
- We don't know how to solve these problems quickly



Question: Does graph  $G$  have a clique of size  $k$ ?

$M$  = "On input  $\langle G, k \rangle$ :

- Non-deterministically guess a  $k$ -subset of nodes of  $G$
- Deterministically check whether it is complete
- If so, accept; if not, reject."

Recall. NTM has the following pattern:

$M$  = "On input  $w$ :

- Nondeterministically guess some string  $x$ ,
- Deterministically check whether  $x$  solves  $w$ .
- If so, accept; otherwise, reject."

The complexity class NP (non-deterministic polynomial time) contains all problems that can be solved in polynomial time by a single tape NTM

Alternative way to characterize the NP class:

A polynomial-time verifier is a deterministic TM of the form:

$V$  = "On input  $\langle w, x \rangle$ :

- Deterministically check whether  $x$  solves  $w$ .
- If so, accept; otherwise, reject."

such that  $V$  runs in polynomial time in the length of  $w$  (not  $x$ )

(The string  $x$  is called a certificate or a witness for  $w$ )

NP: class of languages that have polynomial-time verifiers

CLIQUE is in NP:

$V$  = "On input  $\langle G, k \rangle$ , a set of nodes  $C$ ,

If  $C$  has size  $k$  and all edges between nodes  $C$  are present in  $G$   
accept

Otherwise reject"

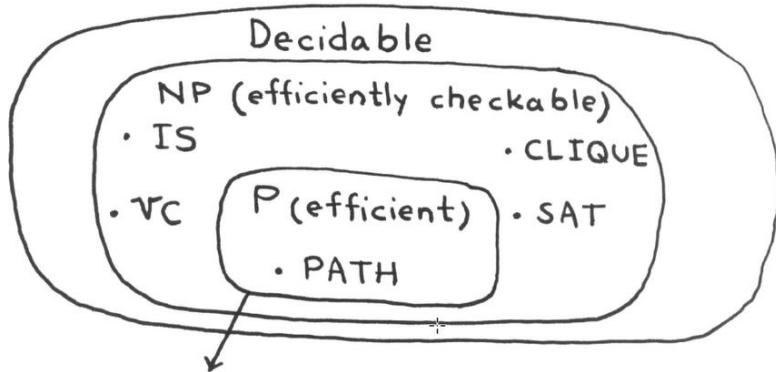
Running time:  $O(k^2)$

P is contained in NP: because verifier can ignore candidate solution

IS: INDEPENDENT SET

VC: VERTEX COVER

(will see SAT later)



can we remove this line? Is  $P = NP$ ?

We do not know (yet!). One reason to believe  $P \neq NP$  is that intuitively searching for a solution is harder than verifying its correctness.

For example, solving homework problems (searching for solutions) is harder than grading (verifying the candidate solution is correct) لَا جُوْهَارْ سِنْسَانْ بُوْدَ

CLIQUE = { $\langle G, k \rangle \mid G$  is a graph having a clique of  $k$  vertices} ⑤

M = On input  $\langle G, k \rangle$ :

For all subsets S of vertices of size  $k$

If every pair  $u, v \in S$  are adjacent

accept

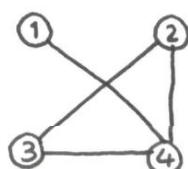
else reject

Example.

input:  $\langle G, 3 \rangle$

subsets:  $\{1, 2, 3\}$      $\{1, 2, 4\}$      $\{1, 3, 4\}$      $\{2, 3, 4\}$   
              X            X            X            ✓

Graph G:



Running time analysis

M = On input  $\langle G, k \rangle$ :

For all subsets S of vertices of size  $k$

$\binom{n}{k}$  subsets

If every pair  $u, v \in S$  are adjacent

$k^2$  pairs

accept

else reject

---

running time:  $k^2 \binom{n}{k}$

According to Stirling's formula  $k^2 \binom{n}{k} \geq 2^n$  when  $k = \frac{n}{2}$

We strongly suspect that problems like CLIQUE, IS, VC require roughly  $2^n$  time to solve

We do not know how to prove this, but we can prove:

If any one of them can be solved efficiently,

then all of them can be solved efficiently

Next lecture: All problems such as CLIQUE, IS, VC are as hard as one another.

Moreover, they are at least as hard as any problem in NP!

①

جذب ممكنا

پولیت، بیت جذب

$P = \{ L \mid \text{There is a polynomial time decider for } L \}$

$NP = \{ L \mid \text{There is a non-deterministic polynomial-time decider for } L \}$

$P \stackrel{?}{=} NP$  Most important question in theoretical computer science

If a solution to a problem can be verified efficiently,  
can that problem be solved efficiently?

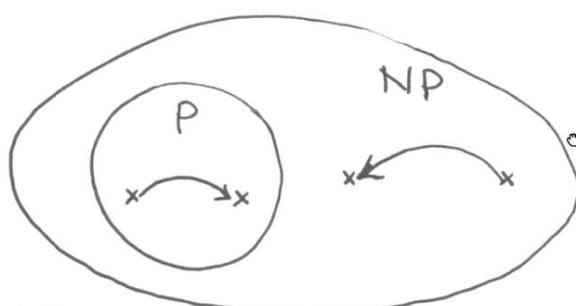
A polynomial-time reduction from  $L_1$  to  $L_2$  is a function  $f: \Sigma^* \rightarrow \Sigma^*$  such that

- for any  $w \in \Sigma^*$ ,  $w \in L_1$  iff  $f(w) \in L_2$ .
- the function  $f$  can be computed in polynomial time.

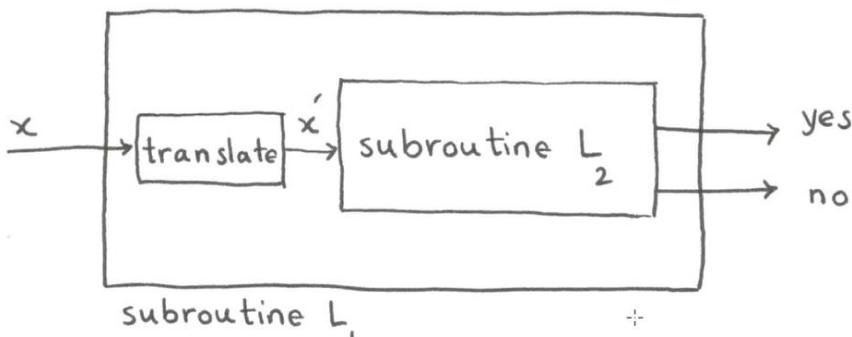
If  $L_1 \leq_p L_2$  and  $L_2 \in P$ , then  $L_1 \in P$

Contrapositive : If  $L_1 \leq_p L_2$  and  $L_1 \notin P$ , then  $L_2 \notin P$

If  $L_1 \leq_p L_2$  and  $L_2 \in NP$ , then  $L_1 \in NP$



Reduce  $L_1$  to  $L_2$  : Allows us to solve  $L_1$  in poly-time if we have a poly-time solver for  $L_2$

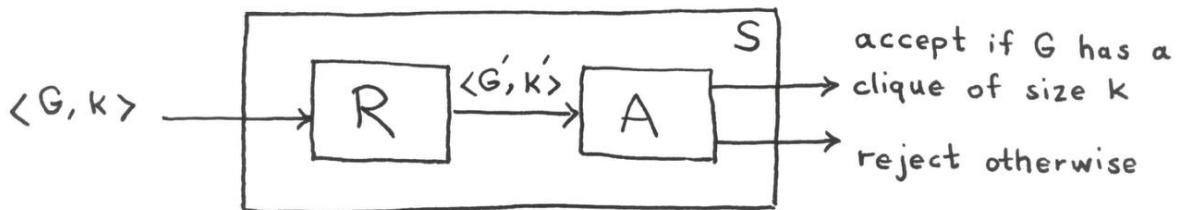


Theorem . CLIQUE  $\leq_p$  INDEPENDENT-SET

(2)

Suppose IS is decided by a poly-time TM A.

We want to build a TM S that uses A to solve CLIQUE.

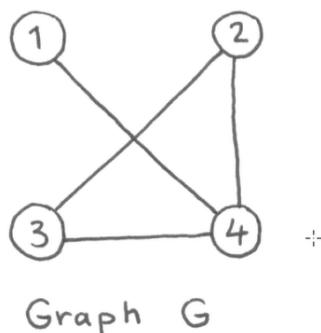


We look for a polynomial-time Turing machine R that turns question

"Does G have a clique of size k?"

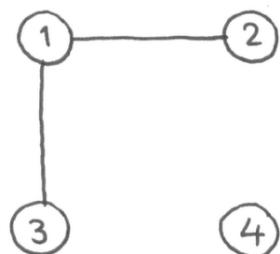
into

"Does  $G'$  have an independent set (IS) of size  $k'$ ?"



Graph G

Flip edges



Graph  $G'$

clique of size k

$\longleftrightarrow$

IS of size  $k'$

Cliques in G

$\longleftrightarrow$

Independent sets in  $G'$

\* If G has a clique of size k,

then  $G'$  has an independent set of size k

\* If G does not have a clique of size k,

then  $G'$  does not have an independent set of size k

On input  $<G, k>$ :

Construct  $G'$  by flipping all edges of G

Set  $k' = k$

Output  $<G', k'>$

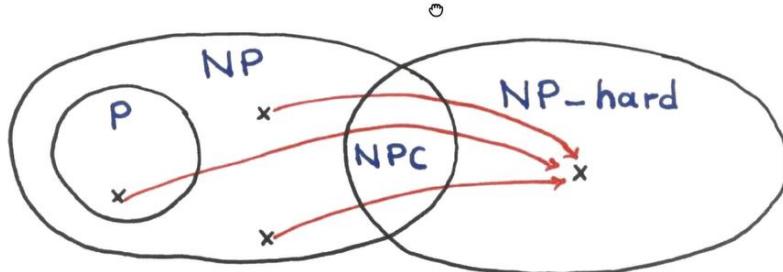


(3)

A language  $L$  is called **NP-hard** iff for every  $L' \in NP$ :  $L' \leq_p L$

A language  $L$  is called **NP-complete** iff  $L$  is NP-hard and  $L \in NP$

The class **NPC** is the set of NP-complete problems



**NPC**: subclass of NP which contains the hardest problems of NP  
If you can efficiently solve one NP-complete problem, then you can efficiently solve all NP problems

If you find a polynomial time solution to an NP-complete problem:

- You have a bug
- Get ready to receive 1000,000 \$ !

The first NP-complete problem found was the satisfiability problem.

A Boolean formula is an expression made of variables, ANDs, ORs, and negation, like:

$$\varphi = (x_1 \vee \bar{x}_3) \wedge (x_2 \vee x_1 \vee x_3) \wedge (\bar{x}_1)$$

Task: Assign True/False values to variables so that the formula evaluates to True

$$x_1 = \text{False} \quad x_2 = \text{True} \quad x_3 = \text{False}$$

$$\text{SAT} = \left\{ \langle \varphi \rangle \mid \varphi \text{ is a satisfiable Boolean formula} \right\}$$

Theorem (Cook-Levin): SAT is NP-complete.

Finding a solution:

Try all possible assignments

FFF    FFT    FTF    FTT

TFF    TFT    TTF    TTT

For  $n$  variables, there are  $2^n$  possible assignment (Takes exp time)

Verifying a solution :

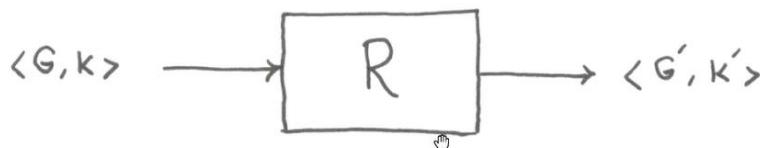
substitute  $x_1 = F$ ,  $x_2 = T$ ,  $x_3 = F$

evaluate the formula:

$$\varphi = (x_1 \vee \bar{x}_3) \wedge (x_2 \vee x_1 \vee x_3) \wedge (\bar{x}_1)$$

This can be done in linear time.

Theorem : INDEPENDENT-SET  $\leq_p$  VERTEX-COVER

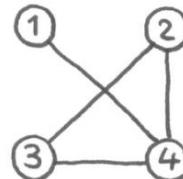


$G$  has an IS of size  $k$   $\iff G'$  has a VC of size  $k'$

Example :

Independent sets:

$$\emptyset, \{1\}, \{2\}, \{3\}, \{4\}, \\ \{1, 2\}, \{1, 3\}$$



vertex covers:

$$\{2, 4\}, \{3, 4\}, \\ \{1, 2, 3\}, \{1, 2, 4\}, \\ \{1, 3, 4\}, \{2, 3, 4\}, \\ \{1, 2, 3, 4\}$$

Claim:  $S$  is an independent set if and only if  $\bar{S}$  is a vertex cover

Proof :

$S$  is an independent set



no edge has both endpoints in  $S$



every edge has an endpoint in  $\bar{S}$



$\bar{S}$  is a vertex cover

| IS          | VC               |
|-------------|------------------|
| $\emptyset$ | $\{1, 2, 3, 4\}$ |
| $\{1\}$     | $\{2, 3, 4\}$    |
| $\{2\}$     | $\{1, 3, 4\}$    |
| $\{3\}$     | $\{1, 2, 4\}$    |
| $\{4\}$     | $\{1, 2, 3\}$    |
| $\{1, 2\}$  | $\{3, 4\}$       |
| $\{1, 3\}$  | $\{2, 4\}$       |



R : On input  $\langle G, k \rangle$

Output  $\langle G', n-k \rangle$

$G$  has an IS of size  $k$   $\longleftrightarrow$   $G$  has a VC of size  $n-k$

Overall sequence of reductions :

SAT  $\rightarrow$  3SAT  $\rightarrow$  CLIQUE  $\xrightarrow{\checkmark}$  IS  $\xrightarrow{\checkmark}$  VC

What is 3SAT?

First some definitions :

- A literal in a Boolean formula is a variable or its negation

$x, \bar{x}$  are literals

$x \wedge y$  is not

- A clause is disjunction (OR) of literals.

$(\bar{x} \vee y \vee z), \bar{x}$  are clauses

$x \vee (\bar{y} \vee z)$  is not

- A Boolean formula is in conjunctive normal form (CNF) if it is the AND of clauses.

$(x \vee y \vee z) \wedge (\bar{x} \vee y) \wedge (x)$  is in CNF

$(x \vee (y \wedge z)) \wedge (\bar{x} \vee \bar{y})$  is not in CNF

- A Boolean formula is in 3-CNF if

\* It is in CNF

\* Every clause has exactly three literals

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

But not:  $(x \vee y \vee z \vee w) \wedge (x \vee y \vee z)$

The language 3SAT is defined as:

(6)

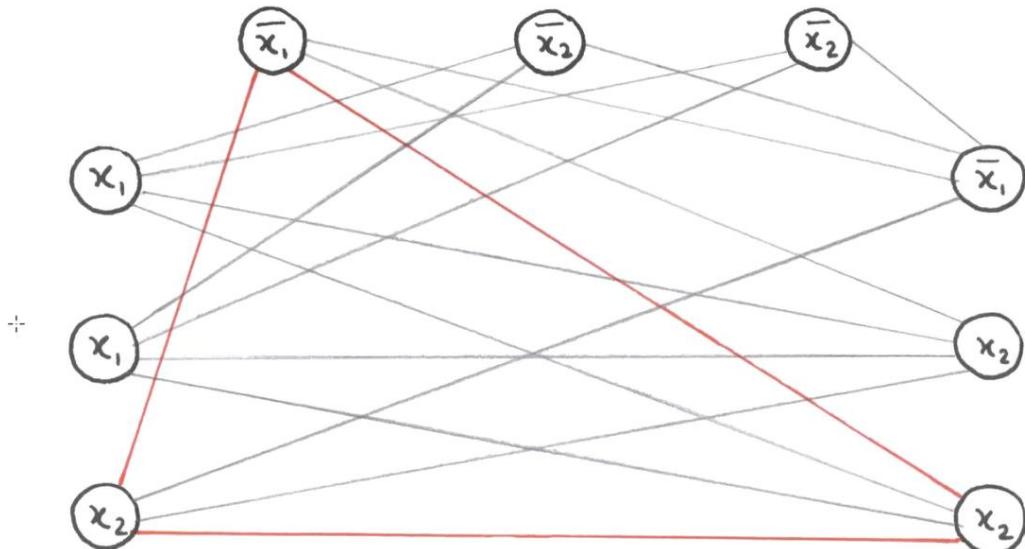
$$3SAT = \{ \langle \varphi \rangle \mid \varphi \text{ is a satisfiable 3-CNF formula} \}$$

Theorem:  $3SAT \leq_p CLIQUE$

Transform a 3-CNF formula  $\varphi$  into  $\langle G, k \rangle$  such that

$$\varphi \in 3SAT \Leftrightarrow \langle G, k \rangle \in CLIQUE$$

Example:  $(x_1 \vee x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_2 \vee x_2)$



Put edges between all pairs of nodes in different clusters,  
except pairs of the form  $\{x, \bar{x}\}$  (inconsistent pairs)

Do not connect any nodes in the same cluster



R : On input  $\varphi$ , where  $\varphi$  is a 3CNF formula with  $k$  clauses

Construct the following graph  $G$ :

$G$  has  $3k$  vertices, divided into  $k$  groups

One for each literal occurrence in  $\varphi$

If vertices  $u$  and  $v$  are in different groups and consistent

Add an edge  $\langle u, v \rangle$

Output  $\langle G, k \rangle$



Every satisfying assignment of  $\varphi$  gives a clique of size  $k$  in  $G$

Conversely, every clique of size  $k$  in  $G$  gives a satisfying assignment of  $\varphi$

Overall sequence of reductions:

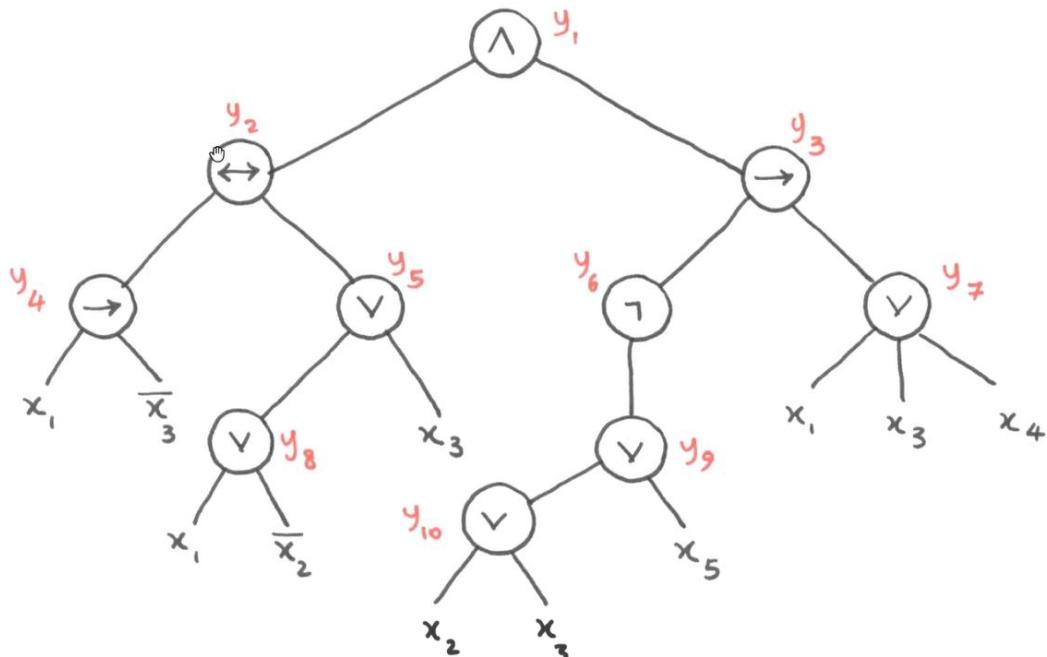
$$\text{SAT} \xrightarrow{\quad} \text{3SAT} \xrightarrow{\quad} \text{CLIQUE} \xrightarrow{\quad} \text{IS} \xrightarrow{\quad} \text{VC}$$

Convert arbitrary Boolean formula  $\varphi$  into formula  $\varphi^*$  in 3-CNF

- \* Straightforward approach may give exponential size  $\varphi^*$

First. Convert formula to tree representation

$$((x_1 \rightarrow \overline{x}_3) \leftrightarrow (x_1 \vee \overline{x}_2 \vee x_3)) \wedge ((\overline{x}_2 \vee x_3 \vee x_5) \rightarrow (x_1 \vee x_3 \vee x_4))$$



Second.

- \* Introduce extra variables  $y_i$  for output of every internal node

- \* Write formulas for relations between variables

$$y_1 \leftrightarrow (y_2 \wedge y_3)$$

$$y_2 \leftrightarrow (y_4 \leftrightarrow y_5)$$

$$y_3 \leftrightarrow (y_6 \rightarrow y_7)$$

⋮

Third. Write formula to express satisfiability of the whole thing (8)

$$\begin{array}{c} y_1 \\ \uparrow \\ \text{final output} \end{array} \wedge \underbrace{(y_1 \leftrightarrow (y_2 \wedge y_3))}_{\substack{\uparrow \\ \text{output of nodes consistent with children}}} \wedge \underbrace{(y_2 \leftrightarrow (y_4 \leftrightarrow y_5))}_{\substack{\uparrow \\ \text{output of nodes consistent with children}}} \wedge \dots$$

Fourth. Rewrite each clause into CNF-form

| $y_1$ | $y_2$ | $y_3$ | $y_1 \leftrightarrow (y_2 \wedge y_3)$ | clause is equivalent to                                                                       |
|-------|-------|-------|----------------------------------------|-----------------------------------------------------------------------------------------------|
| 1     | 1     | 1     | 1                                      | $(y_1 \wedge y_2 \wedge \bar{y}_3) \vee (y_1 \wedge \bar{y}_2 \wedge y_3) \vee \dots$         |
| 1     | 1     | 0     | 0                                      | $(y_1 \wedge y_2 \wedge \bar{y}_3) \vee (y_1 \wedge \bar{y}_2 \wedge y_3) \vee \dots$         |
| 1     | 0     | 1     | 0                                      | use De Morgan $\bar{a \wedge b} = \bar{a} \vee \bar{b}$                                       |
| ...   |       |       |                                        | $(\bar{y}_1 \vee \bar{y}_2 \vee y_3) \wedge (\bar{y}_1 \vee y_2 \vee \bar{y}_3) \wedge \dots$ |

After the fourth step we have formula in CNF-form

$$y_1 \wedge (\bar{y}_1 \vee \bar{y}_2 \vee y_3) \wedge (\bar{y}_1 \vee y_2 \vee \bar{y}_3) \wedge \dots$$

But some clauses have only one or two literals

Fifth. Add extra variables and use them to fill up these clauses

Example. Use extra variables  $p, q$  to replace  $y_i$  by

$$(y_i \vee p \vee q) \wedge (y_i \vee \bar{p} \vee q) \wedge (y_i \vee p \vee \bar{q}) \wedge (y_i \vee \bar{p} \vee \bar{q})$$

After fifth step we have

- formula  $\varphi^*$  in 3-CNF form
- that is satisfiable if and only if the original formula  $\varphi$  is satisfiable
- and conversion can be done in polynomial-time

Overall Sequence of reductions.

$$\text{SAT} \xrightarrow{\text{?}} 3\text{SAT} \xrightarrow{\checkmark} \text{CLIQUE} \xrightarrow{\checkmark} \text{IS} \xrightarrow{\checkmark} \text{VC}$$