

Theory of Computation

INTRODUCTION



王轩
Wang Xuan

► Instructor:

- 王轩 (L1421)
- Tel. 13760199977
- Mail: wangxuan@cs.hitsz.edu.cn

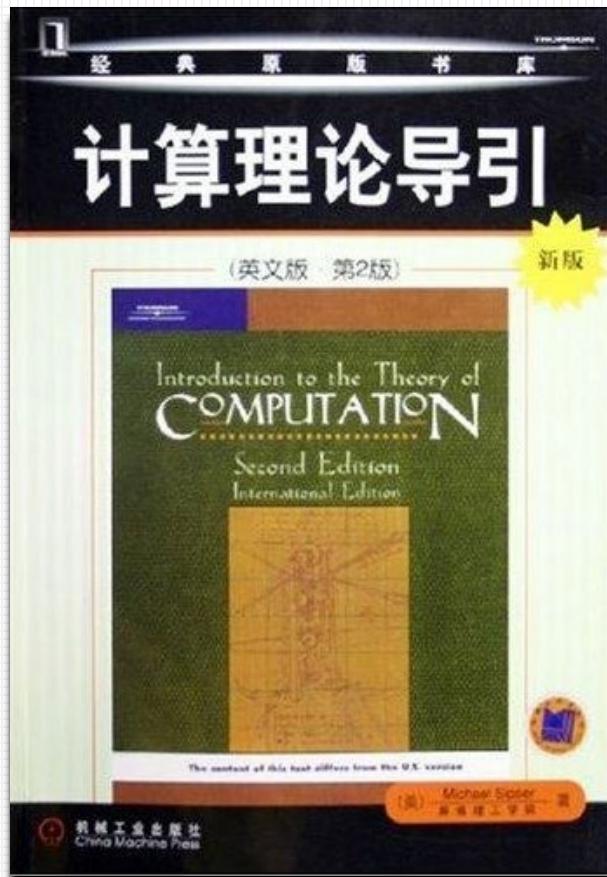
► TA:

- 侯晓涵(L1709)
- Tel. 13713512806
- Mail: toc2020@163.com
- QQ Group: 1151335383

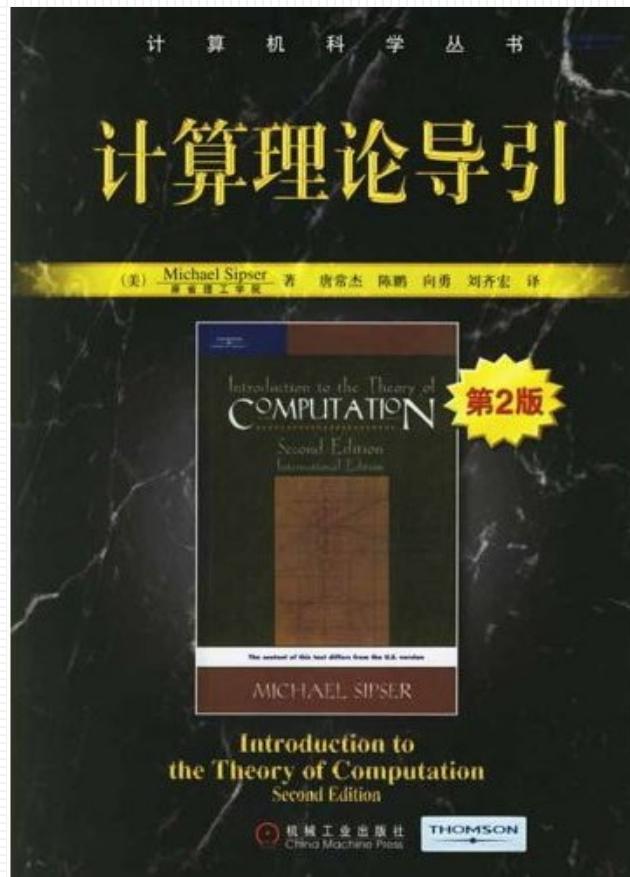


- ◆ 2020.08.31 —— 2020.10.28 (week 1-5, 7-9)
- ◆ 13:45-15:30 on Monday (week 1-5, 7-9)
- ◆ 13:45-15:30 on Wednesday (week 1-5, 7-9)
- ◆ Room A304

Textbook



Introduction to the Theory of Computation.
Michael Sipser, 2nd edition



计算理论导引(第二版)
唐常杰,陈鹏,向勇,刘齐宏 译,
机械工业出版社

- Introduction to Automata Theory, Languages, and Computation. John E. Hopcroft, Jeffrey D. Ullman
- Elements of the Theory of Computation, 2nd ed. Lewis, Harry R., and Papadimitriou, Christos H.. Prentice-Hall, 1997.
- 形式语言与自动机理论 蒋宗礼,姜守旭 编著,清华大学出版社
- 计算复杂性导论 堵丁柱,葛可一,王洁 著,高等教育出版社

Attendance	8%
Homework	22%
Projects	30%
Final Exam	40%

Course description

- Theory of computation, as the name implies, is a theory system that uses mathematical tools to analyze and describe the abstract concept 'computation'.
- This textbook falls into three parts: automata, computability, complexity. They represent three different aspects of computation: model, limits and cost. Automata describes the problems, solutions and the form of computation mechanism theoretically, and the forms of computation mechanism. Computability focus on the feasibility, Complexity focuses on the cost, that is, the difficulty of solving the problem.

Course description

What can we learn?

- What is a computer?
- What can be computed?
- What kind of problems is difficult, and what kind of problems are easy?
- Those are the fundamental questions of computer science, the subject of computation theory and the subject of this book.

Why shall we learn this knowledge?

1. All the knowledge is the primordial reality of computer science. As computer science majors, we should have enough understanding about this.
2. They are wonderful, worthy to experience in our life.
3. They are useful in our daily life.

Alan Turing

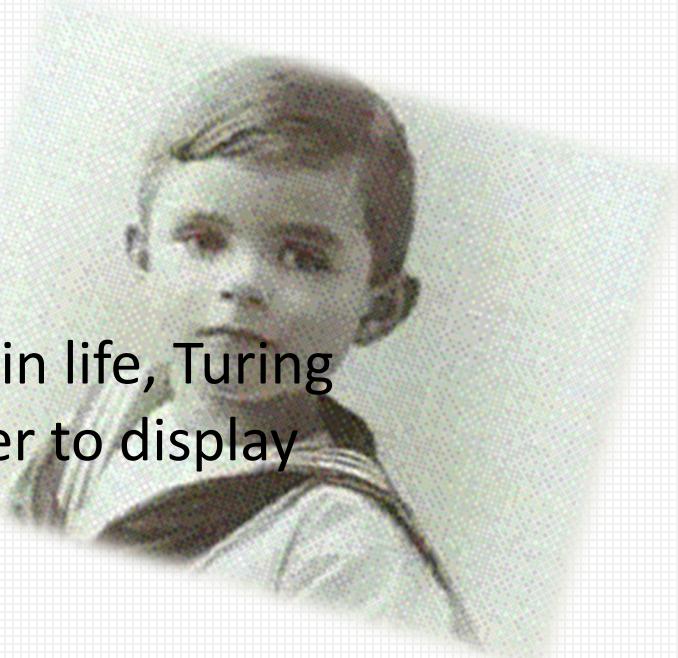


Alan Mathison Turing
(23 June 1912 – 7 June 1954)

- Turing was highly influential in the development of computer science, giving a formalization of the concepts of "algorithm" and "computation" with the Turing machine, which can be considered a model of a general purpose computer.

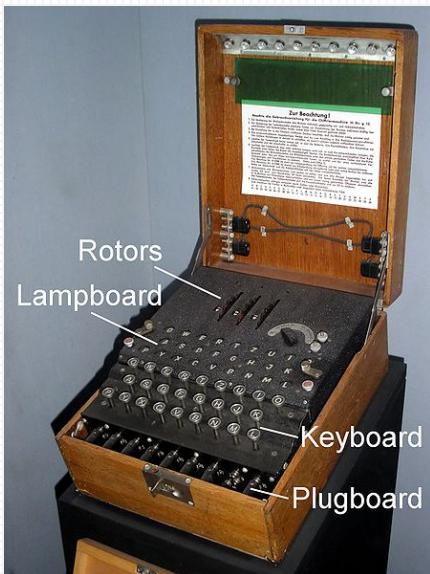
- Turing is widely considered to be the father of computer science and artificial intelligence.

- Turing was born in London. Very early in life, Turing showed signs of the genius he was later to display prominently.
- In 1928, aged 16, Turing encountered Albert Einstein's work; not only did he grasp it, but he extrapolated Einstein's questioning of Newton's laws of motion from a text in which this was never made explicit.



Turing in WW2

- During World War II, Turing worked for the Government Code and Cypher School at Bletchley Park, Britain's code breaking center.
- He devised a number of techniques for breaking German ciphers, including the method of the bombe, an electromechanical machine that could find settings for the Enigma machine.



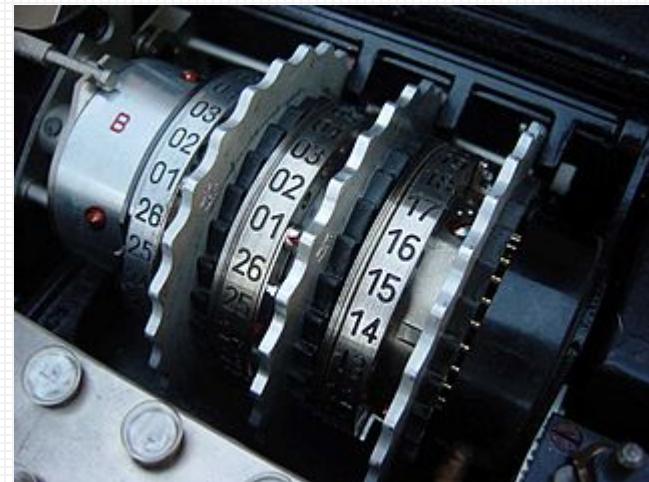
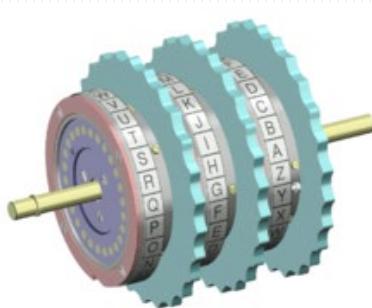
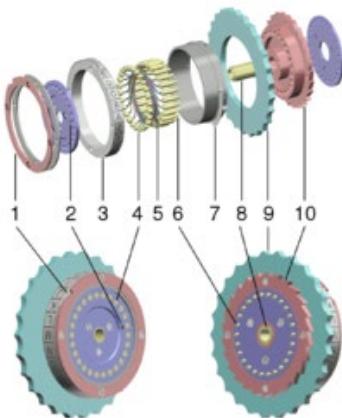
Enigma machine



Turing at Bletchley Park

Enigma machine

- The early models of enigma machine were used commercially from the early 1920s, and adopted by military and government services of Nazi Germany before and during World War II.
- In 1941, British Navy capture an U-110 submarine and get the codebook. Then it was first broken by Bletchley Park, when Alan Turing worked at there.

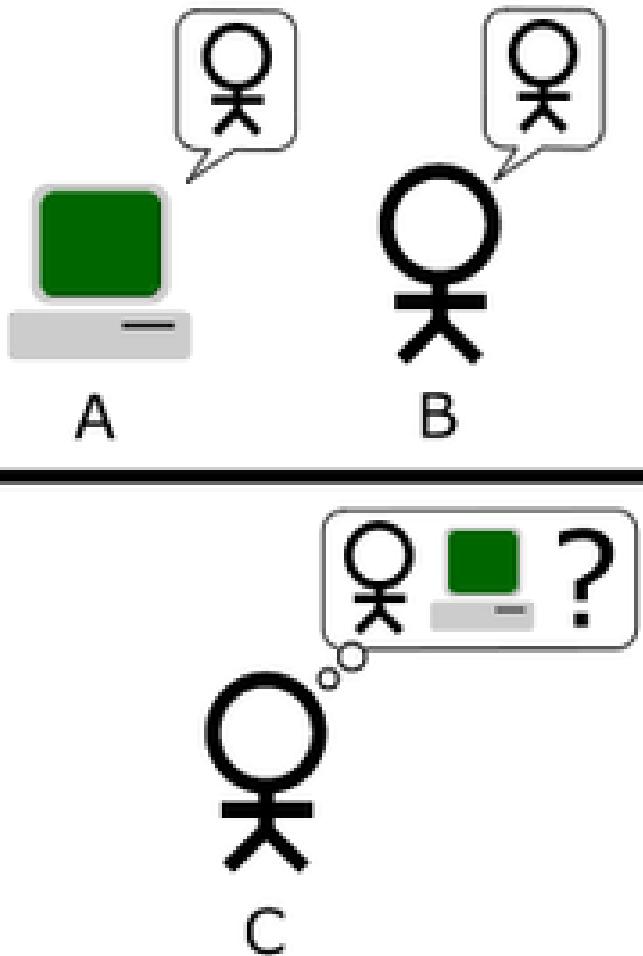


The Film U-571

- This film showed the story about getting the codebook of Enigma machine, and then helps Allies win in world war II.

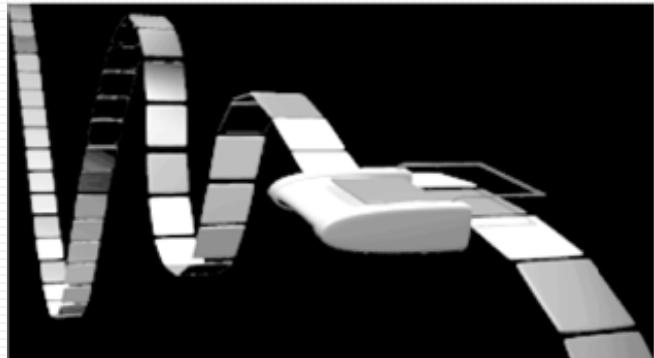


Turing test

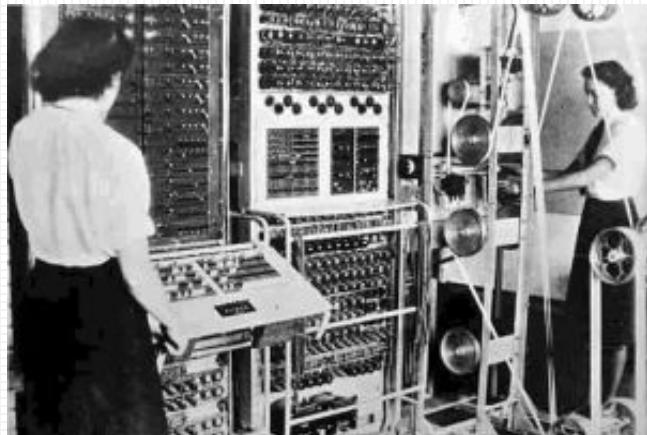


- Turing wrote an article entitled “Can Machines Think?”, which proposed an experiment which became known as the Turing test, an attempt to define a standard for a machine to be called "intelligent".

Turing machine



- A Turing machine is a hypothetical device that manipulates symbols on a strip of tape according to a table of rules.
- It was invented in 1936 by Alan Turing who called it an "a-machine" .



Alan Turing



- Turing was a talented long-distance runner. His best result in marathon was 2 hours 46 minutes 3 seconds.
- While working at Bletchley, Turing occasionally ran the 40 miles to London when he was needed for high-level meetings, and he was capable of world-class marathon standards.

Alan Turing



- Turing's homosexuality resulted in a criminal prosecution in 1952, when homosexual acts were still illegal in the United Kingdom. He died in 1954, just over two weeks before his 42nd birthday, from cyanide poisoning.
- On 10 September 2009, following an Internet campaign, British Prime Minister Gordon Brown made an official public apology on behalf of the British government for "the appalling way he was treated."

- λ -calculus
- Register machine
- Markov algorithm
- Computable function

- Lambda calculus is a formal system in mathematical logic and computer science for expressing computation by way of variable binding and substitution. It was first formulated by Alonzo Church.



Alonzo Church

Lambda calculus has played an important role in the development of the theory of programming languages, including Lisp, ML, Haskell, etc.

Alonzo Church was an American mathematician and logician who made major contributions to mathematical logic and the foundations of theoretical computer science.

- Historical development of the register machine model
 - 1 (1954, 1957) Wang's model: Post-Turing machine
 - 2 Minsky, Melzak-Lambek and Shepherdson-Sturgis models "cut the tape" into many
 - 3 (1961) Melzak's model is different: clumps of pebbles go into and out of holes
 - 4 Lambek (1961) atomizes Melzak's model into the Minsky (1961) model: INC and DEC-with-test
 - 5 Elgot-Robinson (1964) and the problem of the RASP without indirect addressing
 - 6 Hartmanis (1971)
 - 7 Cook and Reckhow (1973) describe the RAM

Markov algorithm

- Markov algorithms are named after the mathematician Andrey Markov, Jr.
- Andrey Andreyevich Markov Jr. was a Soviet mathematician, the son of the Russian mathematician Andrey Andreyevich Markov Sr, and one of the key founders of the Russian school of constructive mathematics and logic.
- Refal is a programming language based on Markov algorithms.

Church-Turing Thesis

In 1936, Turing used his model Turing machine to prove his theory. But Church had proposed a similar thought several months earlier, within λ calculus and Computable function. Soon, under Church's help, Turing found that all these models are computationally equivalent to the Turing machine. Such models are said to be Turing complete.



The Church–Turing thesis states that a function is algorithmically computable if and only if it is computable by a Turing machine.

Universal Turing Machine

- In computer science, a universal Turing machine (UTM) is a Turing machine that can simulate an arbitrary Turing machine on arbitrary input. Alan Turing introduced this machine in 1936–1937.
- A universal Turing machine can calculate any recursive function, decide any recursive language, and accept any recursively enumerable language.
- According to the Church-Turing thesis, the problems solvable by a universal Turing machine are exactly those problems solvable by an algorithm or an effective method of computation, for any reasonable definition of those terms.

Turing Award

1996 年	阿米尔·伯纳利	Amir Pnueli	时序逻辑，程序与系统验证
1997 年	道格拉斯·恩格尔巴特	Douglas Engelbart	互动计算
1998 年	詹姆斯·尼古拉·格雷	James Gray	数据库与事务处理
1999 年	弗雷德里克·布鲁克斯	Frederick P. Brooks, Jr.	计算机体系结构，操作系统，软件工程
2000 年	姚期智	Andrew Chi-Chih Yao	计算理论，包括伪随机数生成，密码学与通信复杂度
2001 年	奥利·约翰·达尔·克利斯登·尼加特	Ole-Johan Dahl Kristen Nygaard	面向对象编程
2002 年	罗纳德·李维斯特·阿迪·萨莫尔·伦纳德·阿德曼	Ronald L. Rivest Adi Shamir Leonard M. Adleman	公钥密码学（RSA）
2003 年	艾伦·凯	Alan Kay	面向对象编程
2004 年	文特·瑟夫·罗伯特·卡恩	Vinton G. Cerf Robert E. Kahn	TCP/IP协议
2005 年	彼得·诺尔	Peter Naur	Algol 60语言
2006 年	法兰西斯·艾伦	Frances E. Allen	优化编译器
	爱德蒙·克拉克	Edmund M. Clarke	

2000 年	姚期智	Andrew Chi-Chih Yao	计算理论，包括伪随机数生成，密码学与通信复杂度
-----------	-----	------------------------	-------------------------

2009 年	查尔斯·萨克尔	Charles Thacker	帮助设计、制造第一款现代PC
2010 年	莱斯利·瓦伦特	Leslie Valiant	对众多计算理论所做的变革性的贡献
2011 年	尤大·伯尔	Judea Pearl	人工智能

The ACM A.M. Turing Award is an annual prize given by the Association for Computing Machinery (ACM) to "an individual selected for contributions of a technical nature made to the computing community".

Andrew Chi-Chih Yao

reading role in
ities do to turn out
ts do to become
s in other countries
on success stories?



- ▶ Andrew Yao received the Turing Award, the most prestigious award in computer science, in 2000, "in recognition of his fundamental contributions to the theory of computation, including the complexity-based theory of pseudorandom number generation, cryptography, and communication complexity".

Turing Award in 2000

John von Neumann

- John von Neumann was a Hungarian-born American pure and applied mathematician and polymath. He made major contributions to a number of fields, including mathematics, physics, economics, computer science, and statistics.

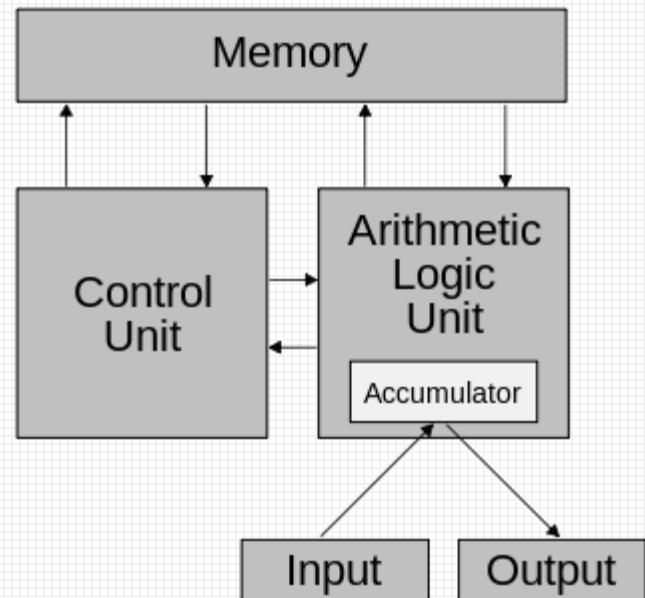
John von Neumann consulted for the ENIAC project, when ENIAC was being modified to contain a stored program. Since the modified ENIAC was fully functional by 1948 and the EDVAC wasn't delivered to Ballistics Research Laboratory until 1949, one could argue that ENIAC was the first computer to use a stored program.



Von Neumann in the 1940s

von Neumann architecture

- The term Von Neumann architecture derives from a 1945 computer architecture description by the mathematician and early computer scientist John von Neumann and others, First Draft of a Report on the EDVAC.
- This architecture is a **Physical device** of Universal Turing Machine.



- Introduction
- Automata and Languages
- Context-Free Languages
- The Church-Turing Thesis
- Decidability
- Reducibility
- Time Complexity

- ▶ Avram Noam Chomsky is an American linguist, philosopher, cognitive scientist, logician, and political commentator and activist. Working for most of his life at the Massachusetts Institute of Technology (MIT), where he is currently Professor Emeritus, he has authored over 100 books on various subjects.
- ▶ Chomskyan linguistics challenges structural linguistics and introduces transformational grammar. This approach takes sequences of words to have a syntax characterized by a formal grammar; in particular, a context-free grammar extended with transformational rules.



Noam Chomsky



Chomsky is also well known as a political activist, and a leading critic of U.S. foreign policy, state capitalism, and the mainstream news media. Ideologically, he aligns himself with anarcho-syndicalism and libertarian socialism.

Chomsky hierarchy

- The Chomsky Hierarchy allows the possibility of a computer science model to accomplish meaningful linguistic goals systematically. Different kind of languages can be recognized by different kind of automatas.
- The following table summarizes each of Chomsky's four types of grammars, the class of language it generates, the type of automaton that recognizes it, and the form its rules must have.

查 · 论 · 编		自动机理论：形式语言和形式文法			隐藏▲
乔姆斯基层级		文法	语言	极小自动机	
类型 0	—	无限制 (无公用名)	递归可枚举	图灵机	
类型 1	—	上下文有关	递归	判定器	
	—	附标	上下文有关	线性有界	
	—	Linear context-free rewriting systems etc.	Mildly context-sensitive	嵌套堆栈	
	—	树-邻接	适度上下文有关	Thread automata	
类型 2	—	上下文无关	上下文无关	嵌入下推	
	—	确定上下文无关	确定上下文无关	非确定下推	
	—	Visibly pushdown	Visibly pushdown	确定下推	
类型 3	—	正则	正则	Visibly pushdown	
	—	—	Star-free	有限	
	—	—	Counter-free (with aperiodic finite monoid)	Counter-free (with aperiodic finite monoid)	

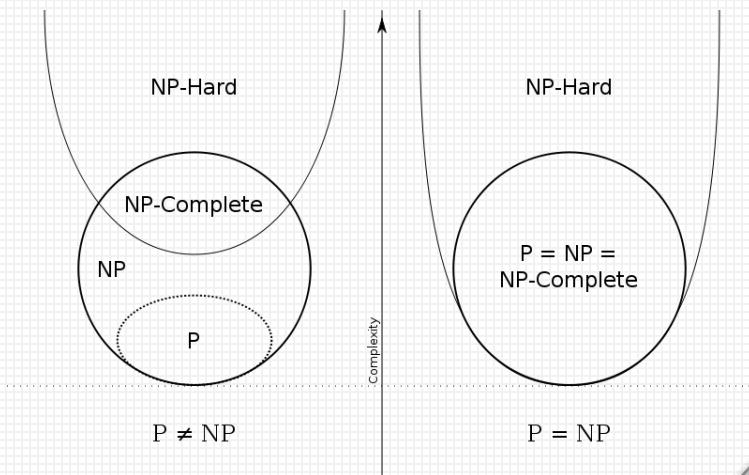
每个语言或文法范畴都是其直接上面的范畴的真子集 Any automaton and any grammar in each category has an equivalent automaton or grammar in the category directly above it.

- A quick way of solving the new problem is to transform each instance of the new problem into instances of the old problem
- Another, more subtle use is this: if we can show that every instance of the old problem can be solved easily by transforming it into instances of the new problem and solving those, we have a contradiction. This establishes that the new problem is also hard.
- In Cryptography reducibility is used a lot to prove the cipher is secure.

- ▶ The Millennium Prize Problems are seven problems in mathematics that were stated by the Clay Mathematics Institute in 2000. As of July 2013, six of the problems remain unsolved. A correct solution to any of the problems results in a US\$1,000,000 prize being awarded by the institute.

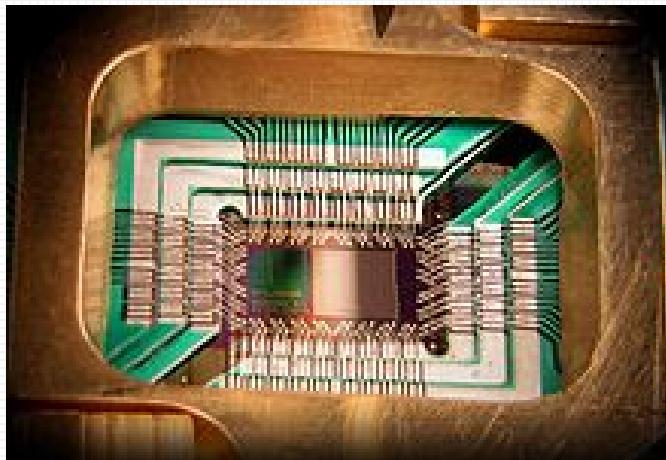
The Millennium Prize Problems:

1. P versus NP
2. The Hodge conjecture
3. The Poincaré conjecture (proven)
4. The Riemann hypothesis
5. Yang–Mills existence and mass gap
6. Navier–Stokes existence and smoothness
7. The Birch and Swinnerton-Dyer conjecture



Quantum computers

- A quantum computer is a computation device that makes direct use of quantum-mechanical phenomena, such as superposition and entanglement, to perform operations on data.



Photograph of a chip in D-Wave One

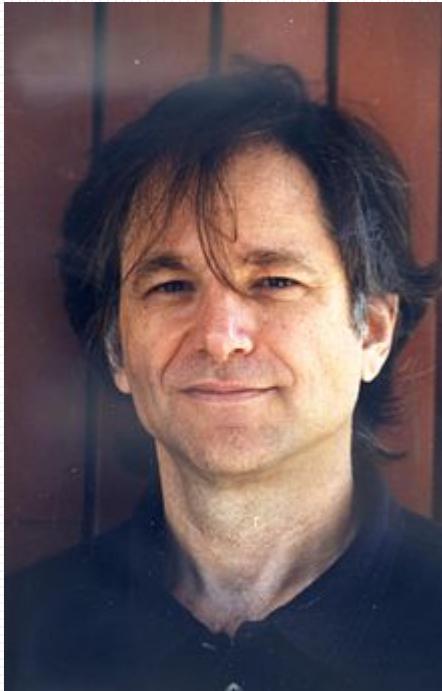
Large-scale quantum computers will be able to solve certain problems much more quickly.
But it does not violate the Church–Turing thesis.

In 2011, D-Wave Systems announced the first commercial quantum annealer on the market by the name D-Wave One.

DNA computing

DNA computing was initially developed by Leonard Adleman of the University of Southern California, in 1994.

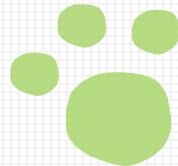
Adleman demonstrated a proof-of-concept use of DNA as a form of computation which solved the seven-point Hamiltonian path problem.



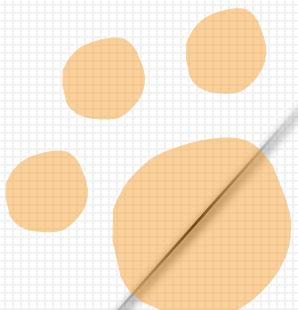
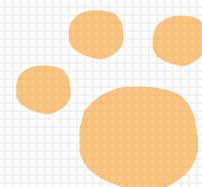
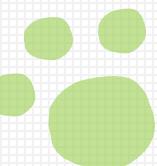
Since the initial Adleman experiments, advances have been made and various Turing machines have been proven to be constructible.

Leonard Max Adleman

THE END

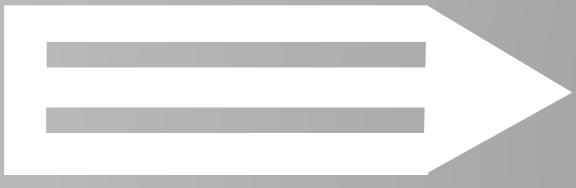


Thank you!!



Theory of Computation

2.1 Mathematical Preliminaries



王轩
Wang
Xuan

1. Set

1.1. Set Definition

1.2. Set Operations

1.3. Properties of set operations

1.4. Cardinality and Power set

2. Relation

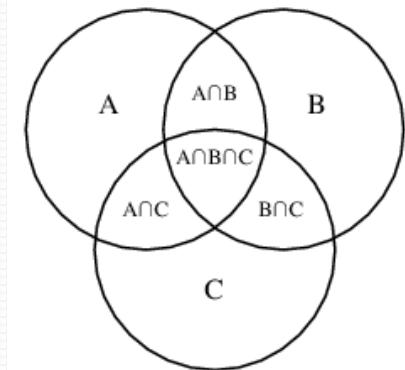
2.1. Equivalence Relation

2.2. Partial Ordering Relations

3. Functions

1.1 Set Definition

- ▶ A set is a group of objects represented as a unit.
 - Repetition don't have meaning
 - Don't have sequence
- ▶ Set is specified in three ways:
 - list of elements in curly braces: $A = \{6, 12, 28\}$
 - characteristic property: $B = \{x \mid x \text{ is a positive, even integer}\}$
 - Venn diagram
- ▶ Set membership: (is [not] element of)
 - a is element of A : $12 \in A$
 - b is not element of A: $9 \notin A$
- ▶ Set Relations(Set inclusion):
 - $A \subseteq B$ (A is a subset of B) if every elements of A is also an element of B. E.x. $\{a,c\} \subseteq \{a,b,c\}$, $\{a,b,c\} \subseteq \{a,b,c\}$
 - $A \subset B$ (A is a proper subset of B) if $A \subseteq B$ but B contains an element not in A. E.x. $\{a,c\} \subset \{a,b,c\}$, $\{a,b,c\} \not\subset \{a,b,c\}$



Venn Diagram

► Size

- $|\{a,b,c\}| = 3$

► Infinite Sets :

- Infinite sets may be countable or uncountable.

► Disjoint Sets :

- iff. $A \cap B = \emptyset$, A and B are disjoint sets.

► Cross Products:

- $A \times B = \{(a,b) : \text{for all } a \in A \text{ and } b \in B\}$ Note—"ordered" pairs
- Is it possible for $A \times B$ to equal $B \times A$?
- E.x. $A=\{1,2\}$, $B=\{a,b,c\}$

$$A \times B = \{(1,a), (1,b), (1,c), (2,a), (2,b), (2,c)\}$$

- Special Sets
 - Empty Set (Null Set) Φ , $\emptyset = \{\}$
 - Universal Set U
- union: $A \cup B = \{x : x \in A \text{ or } x \in B\}$
- intersection: $A \cap B = \{x : x \in A \text{ and } x \in B\}$
- difference: $A - B = \{x : x \in A \text{ and } x \notin B\}$
- Complement : $\bar{A} = \{x : x \in U \text{ and } x \notin A\}$ (always with respect to Universe)

- Example: $A = \{6, 12, 28\}$, $U = \{4, 6, 12, 28, 30\}$
- $A \cup \{9, 12\} = \{6, 9, 12, 28\}$
- $A \cap \{9, 12\} = \{12\}$
- $A - \{9, 12\} = \{6, 28\}$
- $A = \{4, 30\}$

1.3. Properties of set operations

- Idempotency :

$$A \cup A = A$$

$$A \cap A = A$$

- Commutativity :

$$A \cup B = B \cup A$$

$$A \cap B = B \cap A$$

- Associativity :

$$(A \cup B) \cup C = A \cup (B \cup C)$$

$$(A \cap B) \cap C = A \cap (B \cap C)$$

- Distributivity :

$$(A \cup B) \cap C = (A \cap C) \cup (B \cap C)$$

$$(A \cap B) \cup C = (A \cup C) \cap (B \cup C)$$

- Absorption :

$$(A \cup B) \cap A = A$$

$$(A \cap B) \cup A = A$$

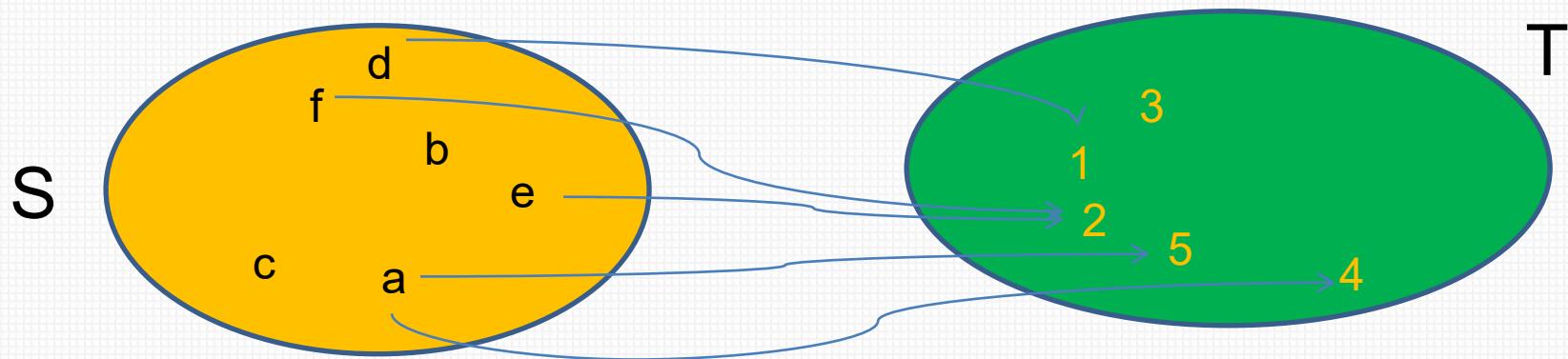
- DeMorgan's Laws :

$$A - (B \cup C) = (A - B) \cap (A - C)$$

$$A - (B \cap C) = (A - B) \cup (A - C)$$

1.4. Cardinality and Power set

- The cardinality of a set, represented by $|S|$, is the number of elements in a set.
 - Let $S = \{2, 4, 6\}$
 - Then $|S| = 3$ or $\text{card}(S) = 3$
- The powerset of S , represented by 2^S , is the set of all subsets of S .
 - $2^S = \{\{\}, \{2\}, \{4\}, \{6\}, \{2, 4\}, \{2, 6\}, \{4, 6\}, \{2, 4, 6\}\}$
 - The number of elements in a powerset is $|2^S| = 2^{|S|}$



- $R = \{ (f, 2), (d, 1), (a, 4), (a, 5), (e, 2) \}$
- A relation on sets S and T is a set of ordered pairs (s, t) , where
 - a) $s \in S$
 - b) $t \in T$
 - c) S and T need not be different

- A subset R of $A \times A$ is called an equivalence relation on A if R satisfies the following conditions:
 - a) Reflexivity(自反性): $(a, a) \in R$ for all $a \in A$ (we show $a \equiv a$)
 - b) Symmetry(对称性): If $(a, b) \in R$, then $(b, a) \in R$, then $(a, b) \in R$
 - c) Transitivity(传递性): If $(a, b) \in R$ and $(b, c) \in R$, then $(a, c) \in R$

- ▶ Give examples of relations R on $A = \{1, 2, 3\}$ with
 - ▶ R being both symmetric and antisymmetric
 - $R = \{(1,1), (2,2)\}$
 - ▶ R being neither symmetric nor antisymmetric
 - $R = \{(1,2), (2,1), (2,3)\}$
- ▶ Antisymmetric (反对称性)
 - a binary relation R on a set X is antisymmetric if, for all a and b in X if $R(a,b)$ and $R(b,a)$, then $a = b$
 - $\forall a, b \in X \ aRb \wedge bRa \Rightarrow a = b$

- Given the relation R in A as
 - $R = \{(1,1), (2,2), (2,3), (3,2), (4,2), (4,4)\}$
- Is R (i) reflexive (ii) symmetric (iii) transitive?
 - i. R is not reflexive because $3 \in A$ but $(3,3) \notin R$
 - ii. R is not symmetric because $(4,2) \in R$ but $(2,4) \notin R$
 - iii. R is not transitive because $(4,2) \in R, (2,3) \in R$ but $(4,3) \notin R$
- Is R antisymmetric?
 - R is not antisymmetric because $(2,3) \in R$ and $(3,2) \in R$ but $2 \neq 3$

- Given a relation R is ‘circular’ if $(a, b) \in R$ and $(b, c) \in R \Rightarrow (c, a) \in R$. Show that a relation is reflexive and circular if and only if it is reflexive, symmetric, and transitive.
- Left to right implication:
 - $(c, a) \in R, (a, a) \in R \Rightarrow (c, a) \in R$, since R is circular.
 - $(a, a) \in R$, since R is reflexive.
 - $(c, a) \in R, (a, a) \in R$ and $(c, a) \in R$, Hence R is transitive.
 - $(c, a) \in R$ and $(a, c) \in R$, Hence R is symmetric .

- Right to left implication:
 - $(a, b) \in R, (b, c) \in R \Rightarrow (a, c) \in R$, since R is transitive
 - $\Rightarrow (c, a) \in R$, since R is symmetric $\Rightarrow R$ is circular
 - $(a, c) \in R, (c, a) \in R \Rightarrow (a, a) \in R$, since R is transitive
 - $\Rightarrow R$ is reflexive

- ▶ For $x, y \in N$, $x \equiv y \pmod{m}$ if and only if $x - y$ is divisible by m , i.e. $x - y = km$, for $k \in z$.
- ▶ Show that the relation “congruence(同余) modulo m ” over the set of positive integers is an equivalence relation.
- ▶ Let $x, y, z \in N$. Then
 - $x - x = 0 \ pmod{m} \Rightarrow x \equiv x \pmod{m}$, for all $x \in N$. Therefore this relation is reflexive
 - $x \equiv y \pmod{m} \Rightarrow x - y = km$, for integer $k \Rightarrow y - x = (-k)m \Rightarrow y \equiv x \pmod{m}$
Therefore the relation is symmetric
 - $x \equiv y \pmod{m}$ and $y \equiv z \pmod{m} \Rightarrow x - y = km$ and $y - z = lm$ for integers k, l .
 $\Rightarrow (x - y) + (y - z) = (k + l)m \Rightarrow (x - z) = (k + l)m \Rightarrow x \equiv z \pmod{m}$ since $k + l$ is also an integer. Therefore the relation is transitive.

- A relation R on a set S is called a “Partial ordering” or a “Partial order”, if R is
 - Reflexive
 - Antisymmetric
 - Transitive
- A set S together with a partial ordering R is called a “Partially ordered set” or “Poset”.
 - Example: The relation \leq on the set \mathbb{R} of real numbers is reflexive, antisymmetric and transitive. Therefore \leq is a “Partial ordering”.

- Thus a partition P of S is a subdivision of S into disjoint nonempty sets
 - $S_1 \cup S_2 \cup \dots \cup S_n = S$
 - $\forall S_i, S_i \neq \emptyset$
 - $\forall S_i, S_j, S_i \neq S_j \Rightarrow S_i \cap S_j = \emptyset$
- Then S_1, S_2, \dots, S_n is called a partition of S.
- If R is an equivalence relation on a set S, for each 'a' in S, let $[a]$ denote the set of elements of S to which 'a' is related under R, i.e.
$$[a] = \{x : (a, x) \in R\}$$
- Here $[a]$ is the "Equivalence class" of 'a' in S.

- $f : S_1 \rightarrow S_2$
 - Domain (f) $\subseteq S_1$
 - Range (f) $\subseteq S_2$
 - S_1 may be same as S_2
- f is Total Function if $\text{Domain}(f) = S_1$ else f is Partial Function

- One-to-One Function (Injection): A function $f : A \rightarrow B$ is said to be one-to-one if different elements in the domain A have distinct images in the range.
 - A function f is one-to-one if $f(a) = f(b)$ implies $a = b$.
- Onto function (Surjection): A function $f : A \rightarrow B$ is said to be an onto function if each element of B is the image of some element of A .

- ▶ One-to-one onto Function (Bijection): A function that is both one-to-one and onto is called a “Bijection”
- ▶ Invertible function: A function $f : A \rightarrow B$ is invertible if its inverse relation f^{-1} is a function from B to A.
A function $f : A \rightarrow B$ is invertible **if and only if** it is both one-to-one and onto.

- Is $f(x) = x^2$ from the set of integers to the set of integers is one-to-one? Why?
 - No e.x. $x = +1, x = -1$
- Given f is a function $f : A \rightarrow B$ where $A = \{a, b, c, d\}$ and $B = \{1, 2, 3\}$ with $f(a) = 3, f(b) = 2, f(c) = 1$, and $f(d) = 3$. Is the function f an onto function?
 - Yes

- ✓ Set Definition
- ✓ Set Operations
- ✓ Properties of set operations
- ✓ Cardinality and Power set
- ✓ Equivalence Relation
- ✓ Partial Ordering Relations
- ✓ Functions

Question?

Theory of Computation

2.2Deterministic Finite Automaton



王轩
Wang
Xuan

- Finite Automata
- Formal Definition of DFA
- Transition Diagrams
- Recognizing Languages
- Regular Languages
- Union of Two Languages and proof
- Concatenation of L_1 and L_2

- A Finite Automata (FA) is the most restricted model of computing that we will work with.
- In a Finite Automaton (FA):
 - Input is read once from left to right.
 - The input is often viewed as being stored/written on a tape.
 - There is no memory, except for one register (also called the finite control) that contains the state.
 - A FA has a fixed and finite number of states.

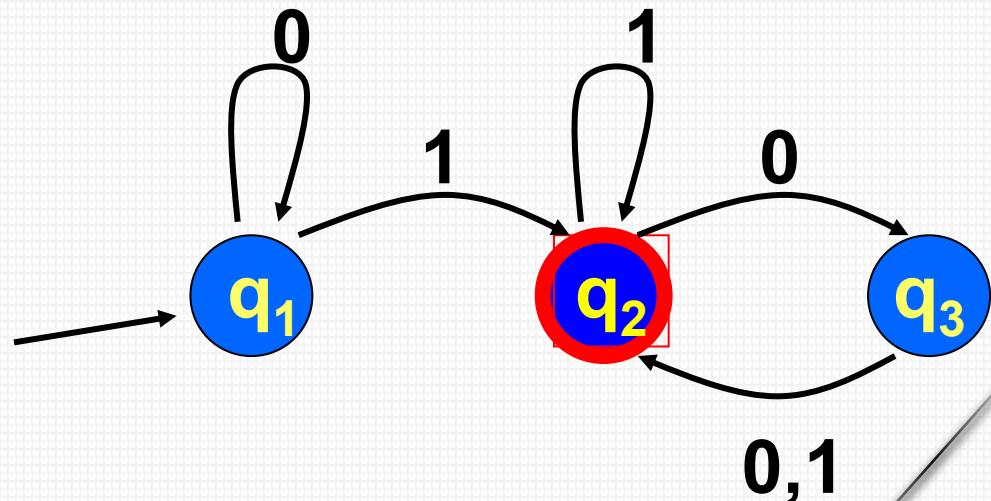
Formal Definition of DFA

- A deterministic finite automaton is a 5-tuple.
- $M = (Q, \Sigma, \delta, q_0, F)$
 - Q : finite set of states
 - Σ : finite alphabet
 - δ : transition function $\delta: Q \times \Sigma \rightarrow Q$
 - $q_0 \in Q$: start state
 - $F \subseteq Q$: set of accepting states

Transition Diagrams

- Transition diagrams can be used to represent a DFA.
 - The vertices are the states.
 - The short arrow points to the initial state.
 - The states with a double circle are the accepting states.
- Matrix Representation

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2



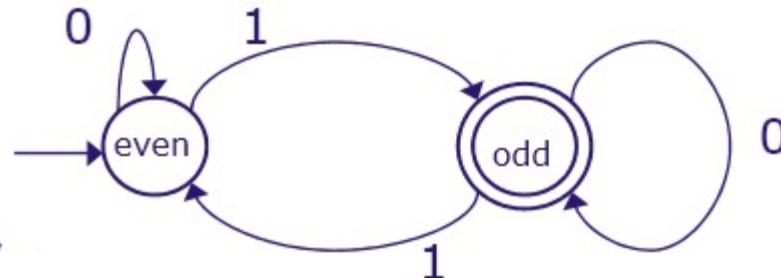
- Consider the language that consists of all strings over $\{1, 0\}$ that contain an odd number of 1's.
- Can we build an algorithm for accepting this language on a FA ????

Specifying the Transitions

- Functions $\delta(\text{even}, 0) = \text{even}; \quad \delta(\text{even}, 1) = \text{odd}; \quad \delta(\text{odd}, 0) = \text{odd}; \quad \delta(\text{odd}, 1) = \text{even};$
- Tables:

State\input	0	1
even	even	odd
odd	odd	even

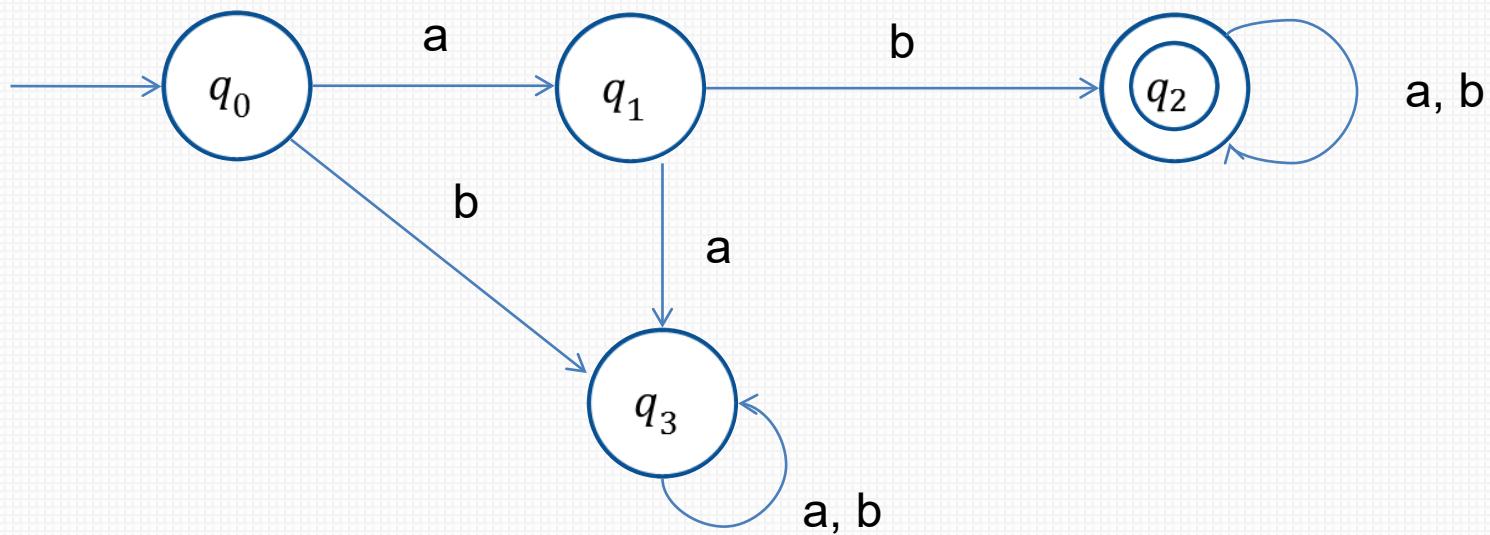
- Graphs
 - Arrow from “nowhere” into initial state
 - Double circle for “final” state



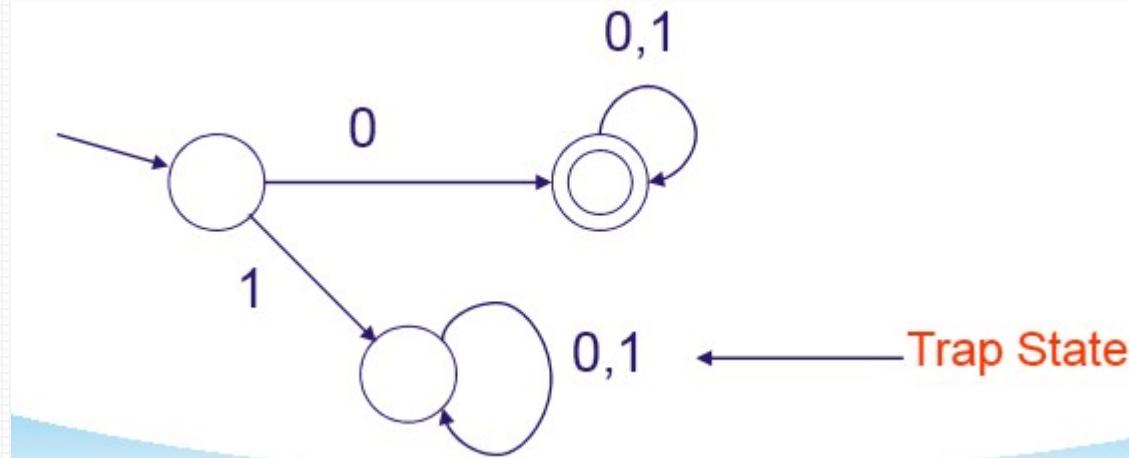
- A Finite Automaton $M = (Q, \Sigma, \delta, q_0, F)$ accepts a string/ word $w = w_1 \dots w_n$ if and only if there is a sequence $r_0 \dots r_n$ of states in Q such that:
 - 1) $r_0 = q_0$
 - 2) $\delta(r_i, w_{i+1}) = r_{i+1}$ for all $i = 0, \dots, n-1$
 - 3) $r_n \in F$
- If L is the set of strings that the automaton accepts, we write $L(M)=L$.
- We say that **M** recognizes L .

- The language accepted by a DFA $M : (Q, \Sigma, \delta, q_0, F)$ is the set of all strings on Σ accepted by M .
- In formal notation $L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\}$.
- **No acceptance** means that the DFA stops in a non-final state. $L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \notin F\}$

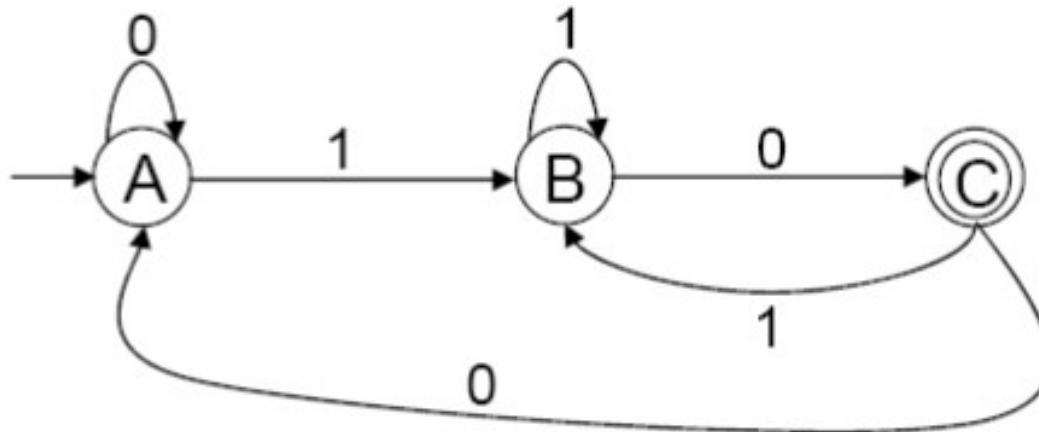
- Find a deterministic finite accepter that recognizes the set of all strings on $\Sigma = \{a, b\}$ starting with the prefix ab.



- Any words with prefixes that take you to a trap state will not be accepted
- Example: Machine accepting words over $\{0,1\}$ that start with 0:



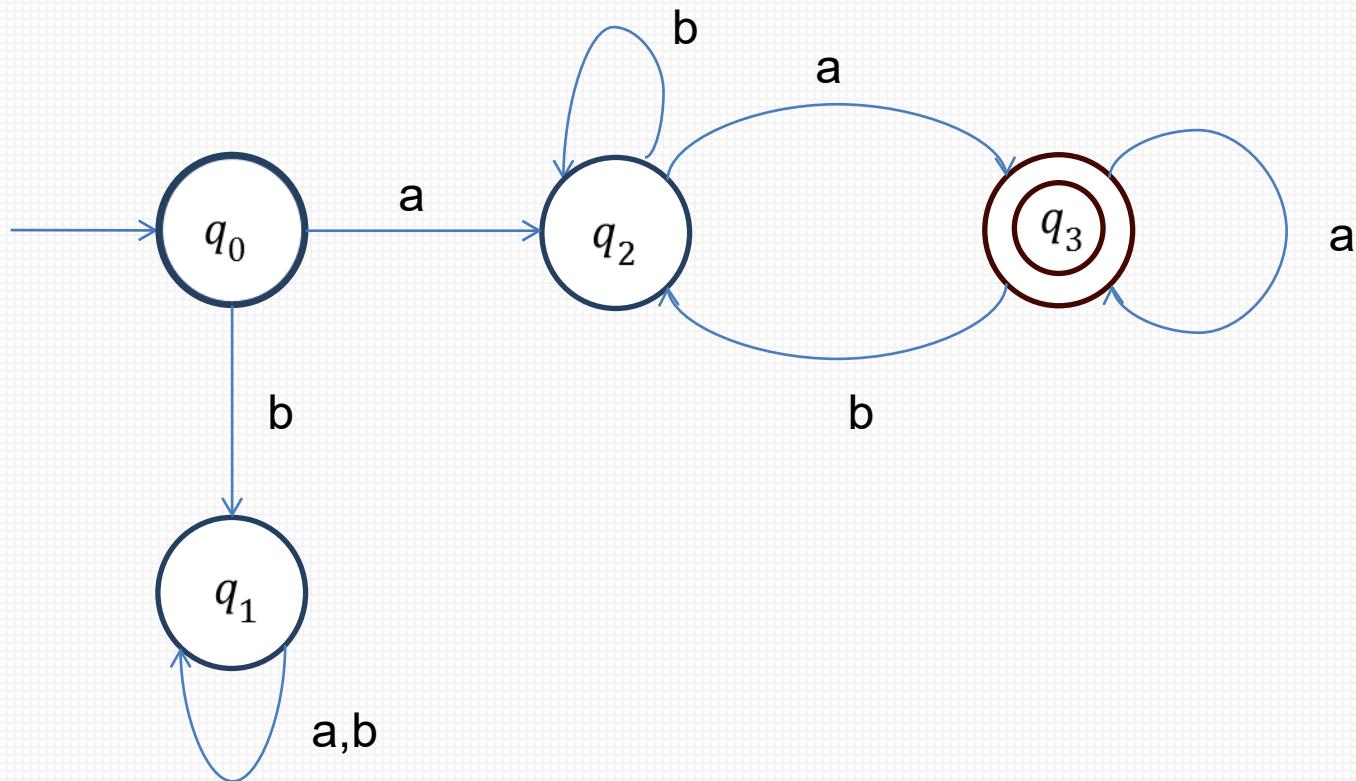
- What language does the FA below accept?



- $0^* 1^+ 0 (1^+ 0)^* (0^+ 1^+ 0 (1^+ 0)^*)^*$

- Every finite automaton accepts some language.
- If we consider all possible finite automata, we get a set of languages associated with them.
- We will call such a set of languages **a family**.
- A language L is called **regular** if and only if there exists some deterministic finite accepter M such that $L = L(M)$
- To prove regular, give DFA accepting it

- Show that the language $L = \{awa : w \in \{a,b\}^*\}$ is regular.



Union of Two Languages

- If A_1 and A_2 are regular languages, then so is $A_1 \cup A_2$. (The regular languages are ‘closed’ under the union operation. Theorem 1.45 in English Edition; theorem 1.12 in Chinese Edition)
- Proof idea: A_1 and A_2 are regular, hence there are two DFA M_1 and M_2 , with $A_1 = L(M_1)$ and $A_2 = L(M_2)$. Out of these two DFA, we will make a third automaton M_3 such that $L(M_3) = A_1 \cup A_2$.

Proof Union-Theorem (1)

- Let $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$, recognize A_1 and A_2
- We construct $M_3 = (Q_3, \Sigma, \delta_3, q_3, F_3)$ to recognize $A_1 \cup A_2$
- $Q_3 = Q_1 \times Q_2 = \{(r_1, r_2) \mid r_1 \in Q_1 \text{ and } r_2 \in Q_2\}$
- $\delta_3((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a))$
- $q_3 = (q_1, q_2)$
- $F_3 = \{(r_1, r_2) \mid r_1 \in F_1 \text{ or } r_2 \in F_2\}$

Proof Union-Theorem (2)

- The automaton $M_3 = (Q_3, \Sigma, \delta_3, q_3, F_3)$ runs M_1 and M_2 in ‘parallel’ on a string w .
- In the end, the final state (r_1, r_2) ‘knows’ if $w \in L_1$ (via $r_1 \in F_1$) and if $w \in L_2$ (via $r_2 \in F_2$)
- The accepting states F_3 of M_3 are such that $w \in L(M_3)$ if and only if $w \in L_1$ or $w \in L_2$, for: $F_3 = \{(r_1, r_2) \mid r_1 \in F_1 \text{ or } r_2 \in F_2\}$.

Concatenation of L_1 and L_2

- Definition: $L_1 \bullet L_2 = \{ xy \mid x \in L_1 \text{ and } y \in L_2 \}$
- Example: $\{a,b\} \bullet \{0,11\} = \{a0, a11, b0, b11\}$
- Theorem 1.13: If L_1 and L_2 are regular languages, then so is $L_1 \bullet L_2$. (The regular languages are ‘closed’ under concatenation.)

Can DFA prove this theorem?

To prove this theorem, let's try something along the lines of the proof of the union case. As before, we can start with finite automata M1 and M2 recognizing the regular languages A1 and A2.

But now, instead of constructing automaton M to accept its input if either M1 or M2 accept, it must accept if its input can be broken into two pieces, where M1 accepts the first piece and M2 accepts the second piece.

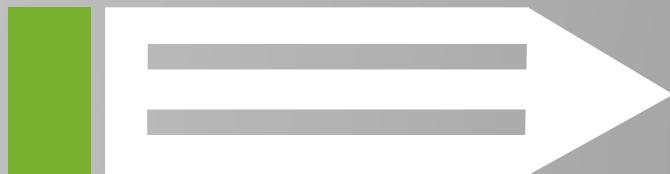
The problem is that M doesn't know where to break its input. To solve this problem we introduce a new technique called nondeterminism.

Review

- ✓ Finite Automata
- ✓ Formal Definition of DFA
- ✓ Transition Diagrams
- ✓ Recognizing Languages
- ✓ Regular Languages
- ✓ Union of Two Languages and proof
- ✓ Concatenation of L_1 and L_2

Theory of Computation

2.3 Nondeterministic Finite Automaton



王轩
Wang
Xuan

非确定性有限状态自动机理论的开创者

1976 年图灵奖获得者：
米凯尔·拉宾和达纳·斯科特

——非确定性有限状态自动机理论的开创者



米凯尔·拉宾



达纳·斯科特

1976 年度的图灵奖由当时在以色列希伯莱大学任教的米凯尔·拉宾 (Michael O. Rabin) 和在英国牛津大学任数理逻辑教授的达纳·斯科特 (Dana Steward Scott) 共同获得。拉宾和斯科特是师兄弟，两人在 20 世纪 50 年代中期先后师从著名的逻辑学家和计算机专家阿隆索·邱奇 (Alonzo Church)，他因与 Curry 一起发明了 λ -演算以及提出了“任何计算，如果存在一有效过程，它就能被图灵机所实现”这一被称为“邱奇论题”的命题而闻名于世)，并在有限自动机及其判定问题的研究中进行合作，奠定了非确定性有限状态自动机的理论基础。之后，他们的研究方向不尽相同，拉宾侧重于计算理论，而斯科特侧重于逻辑学在计算机科学中的应用，在各自的领域中又分别获得重大成果，作出了创造性贡献。

非确定性有限状态自动机理论的开创者

拉宾1931年9月1日生于德国的布雷斯劳。他父亲是一名犹太教教士，也是一位博士；拉宾的母亲也是知识分子，有文学博士头衔。纳粹布特勒上台以后，他们全家于1935年迁回巴勒斯坦，躲过了一劫。1948年以色列建国以后，他们成为以色列公民。

拉宾在濒临地中海的港口城市海法度过了他的童年和少年时代。一次他和比他高好几班的学生比试解欧几里德几何题，他赢了他们，这使他对数学产生了兴趣，因此，从莱利学院（Reali College）毕业以后，他进入希伯莱大学学习数学，在那里，他通过数学家克林（S. C. Kleene, 因提出不动点定理而闻名于世）所著的《元数学》一书首次接触到图灵关于可计算的概念和图灵机这一理论计算模型，立即被深深吸引。但为了打好自己的数学基础，他的硕士论文没有以此为课题，而选择了当时由德国女数学家埃米·诺特（Emmy Noether, 1882-1932）创立不久的抽象代数中关于可交换环理论中的一个问题。

获得数学硕士学位以后，拉宾去了美国，因为20世纪50年代初，以色列建国伊始，经济与科技都还不够发达，很少有人研究计算这类问题，甚至连许计算机都没有。拉宾到美国后，先在宾夕法尼亚大学攻读博士学位。拉宾的博士论文课题将他所熟悉的抽象代数和他感兴趣的可计算性问题联系在一起：群（GROUP）的可计算性问题。拉宾的论文中证明了与群众有关的许多问题，如群是否符合交换律等，都是不能由计算机解答的。

非确定性有限状态自动机理论的开创者

但是使拉宾成名的并非其博士论文而是源于IBM研究中心于1957年向他和他的师弟斯科特提供的一份暑期工作。公司允许他们作他们感兴趣的任何工作，于是拉宾和斯科特就联手研究图灵提出的计算模型，也就是图灵机。图灵机是一种禁止往磁带上写的计算机，叫有限状态自动机(finite state automata，缩写FSA)。图灵在研究这种机器时的基本信条是：机器在输入相同时，其“心智状态”也相同，即对于具有给定指令集的机器而言，一定输入的机器总是按同一方式运行的。

拉宾和斯科特认为，这种具有“确定性”行为的机器带来了局限性。因此，他们定义了一种新的、“非确定性”的有限状态自动机(nondeterministic finite state automata，缩写为NDFSA)，这种机器在读取到一定的输入后，有一个可以进入的新状态的“菜单”可供选择，这样对给定的输入计算便不单一了，每个选择代表一种可能的计算。

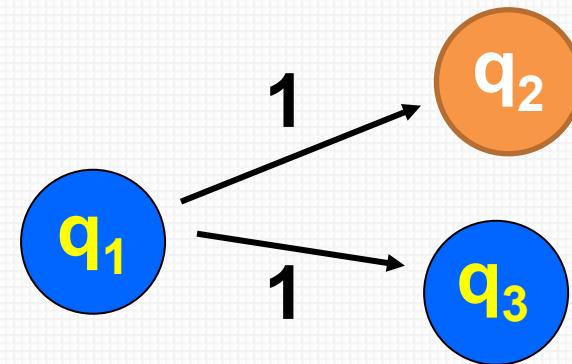
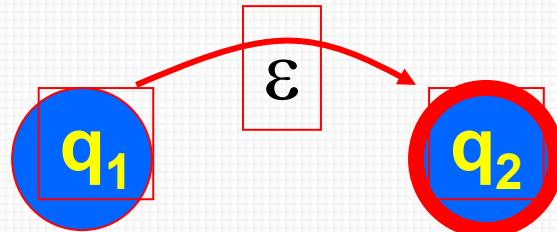
非确定性有限状态自动机理论的开创者

拉宾和斯科特将图灵的有限状态自动机从确定性一种形态扩展到非确定性的另一种形态，极大地推动了有限状态自动机理论的发展。虽然非确定性有限状态自动机的能力并不比确定性的有任何增加(拉宾和斯科特自己已经证明，任何可以用非确定性机器解决的问题都可以在确定性机器上解决，而且提出了将非确定性机器转换为确定性机器的方法问题)，但是它可以简化机器描述和加快解题速度。后来的实践证明，非确定性有限状态自动机在机器翻译、文献检索和字处理程序等应用中都起了重要的作用。

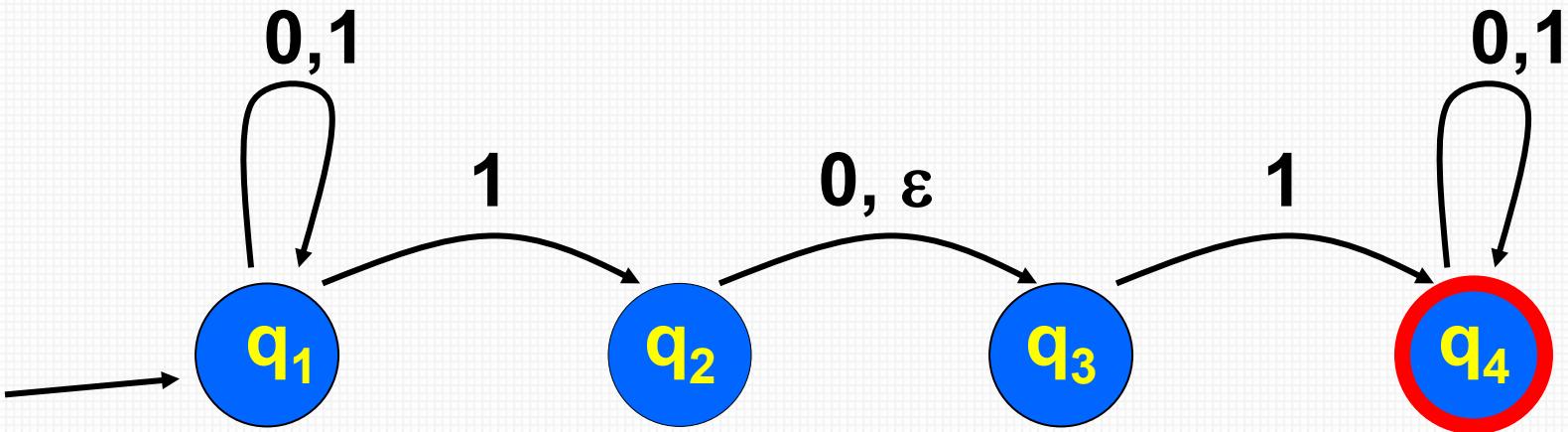
- Formal Definition of NFA
- Recognizing Languages
- Equivalence of DFA and NFA
- Procedure: NFA-to-DFA

Why Non-Determinism?

- An NFA can be converted to a DFA, so we can still build it
- Easier to construct DFA to accept a given language (i.e. pattern matching, grep)
- Multiple choices for same input/state pair, and possibly transitions with no input being “used up”
- A nondeterministic finite automaton has transition rules/possibilities like

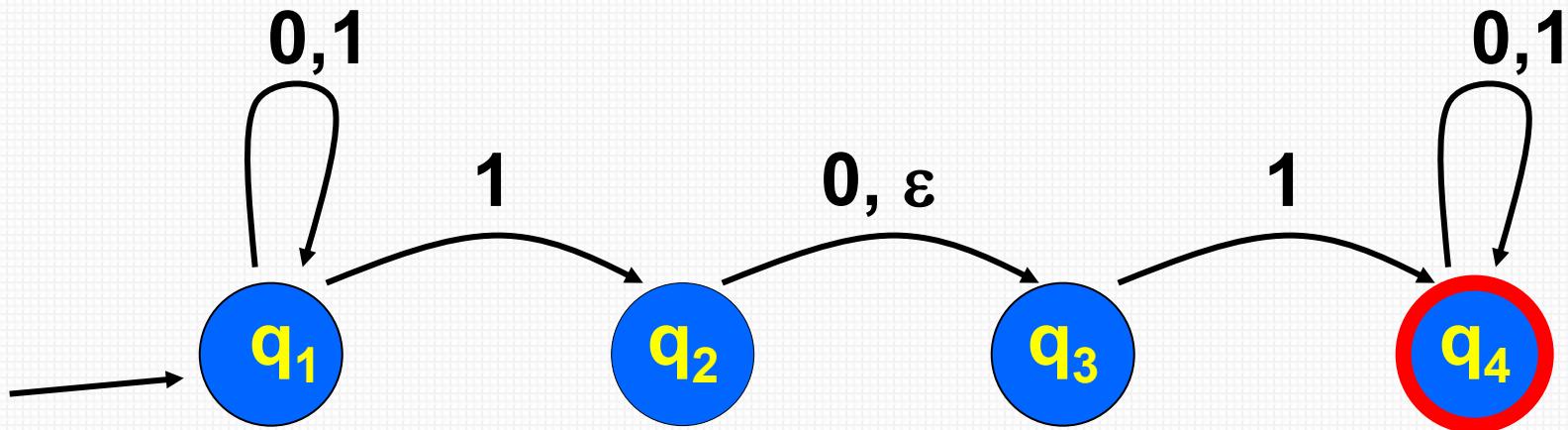


A Nondeterministic Automaton NFA



This automaton accepts “0110”, because there is a possible path that leads to an accepting state, namely:

$$q_1 \rightarrow q_1 \rightarrow q_2 \rightarrow q_3 \rightarrow q_4 \rightarrow q_4$$

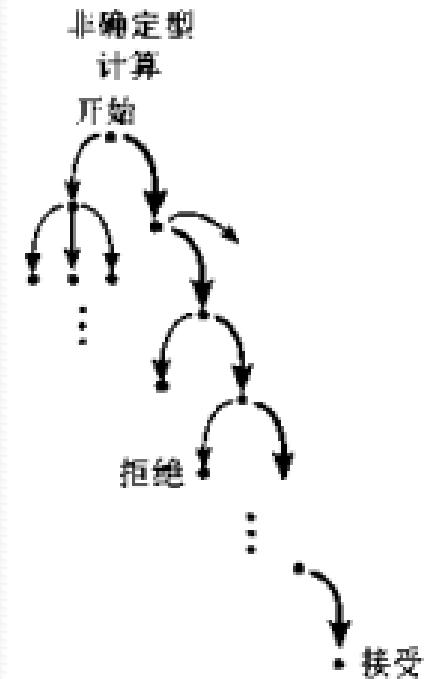


The string “1” gets rejected: on “1” the automaton can only reach: $\{q_1, q_2, q_3\}$.

The set reached doesn’t contain accept states.

Nondeterminism ~ Parallelism

- For any (sub)string w , the nondeterministic automaton can be in a set of possible states.
 - One input with many possible outputs
- If the final set contains an accepting state, then the automaton accepts the string.
- “The automaton processes the input in a parallel fashion. Its computational path is no longer a line, but a tree.”



Formal Definition of NFA

- A nondeterministic finite automaton (NFA) M is defined by a **5-tuple** $M=(Q,\Sigma,\delta,q_0,F)$,
 - Q : finite set of states
 - Σ : finite alphabet
 - δ : transition function $\delta:Q\times\Sigma\rightarrow\mathcal{P}(Q)$
The image of δ is an element of power set
 - $q_0\in Q$: start state, $F\subseteq Q$: set of accepting states

Nondeterministic $\delta: Q \times \Sigma_{\varepsilon} \rightarrow \mathcal{P}(Q)$

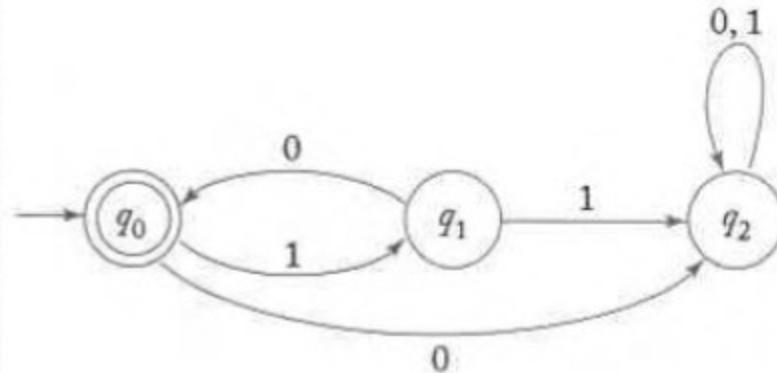
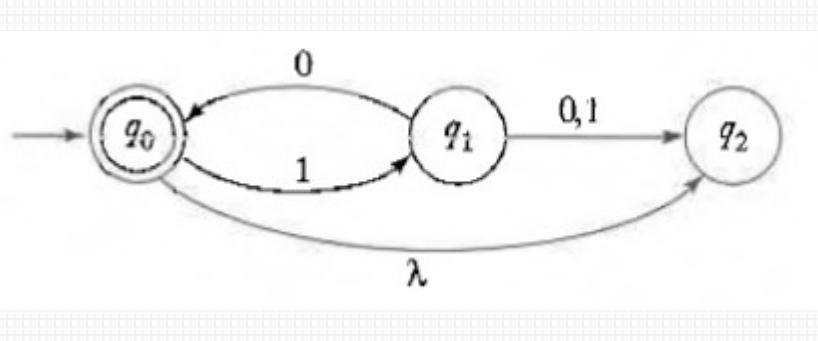
- The function $\delta: Q \times \Sigma_{\varepsilon} \rightarrow \mathcal{P}(Q)$ is the **crucial difference**. It means: “When reading symbol “a” while in state q, one can go to one of the states in $\delta(q, a) \subseteq Q$.
 - When it reads a in state q, it turns to a state in $\delta(q, a)$
- The ε in $\Sigma_{\varepsilon} = \Sigma \cup \{\varepsilon\}$ takes care of the empty string transitions.
 - ε has no input, in another word some state can turn to other states causelessly.

Recognizing Languages

- A nondeterministic FA $M = (Q, \Sigma, \delta, q_0, F)$ accepts a string $w = w_1 \dots w_n$ if and only if we can rewrite w as $y_1 \dots y_m$ with $y_i \in \Sigma_\epsilon$ and there is a sequence $r_0 \dots r_m$ of states in Q such that:
 - 1) $r_0 = q_0$ start state
 - 2) $r_{i+1} \in \delta(r_i, y_{i+1})$ for all $i=0, \dots, m-1$ one path in the tree figure
 - 3) $r_m \in F$ the final state is an element of accepting states

Equivalence of DFA and NFA

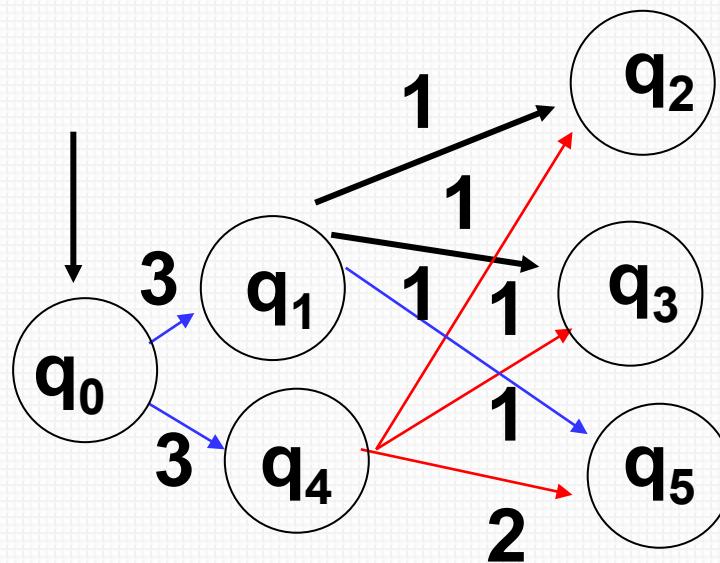
- **Theorem:** Every language accepted by a nondeterministic finite acceptor can be represented with deterministic finite acceptor
- Are these two FAs equivalent?



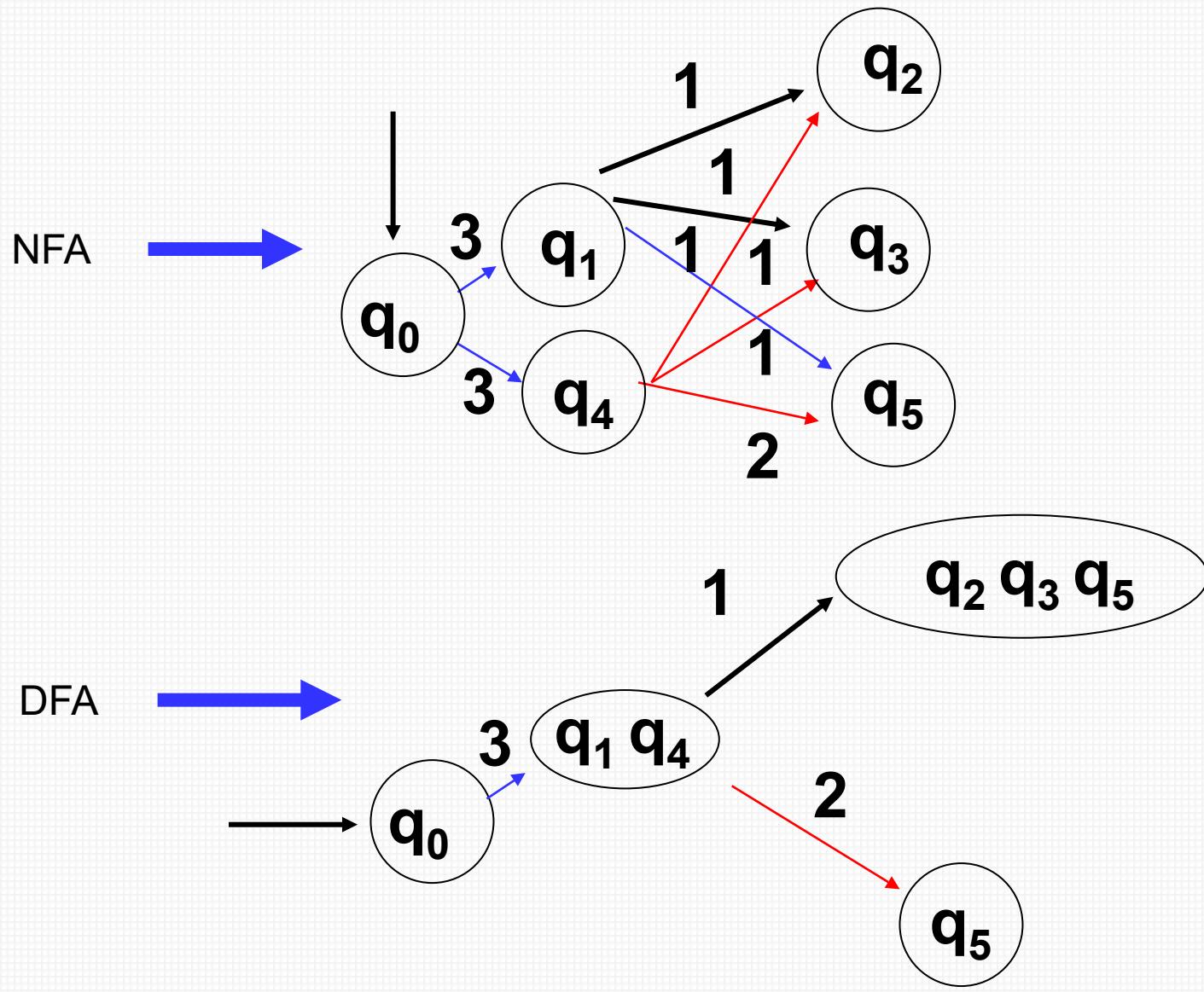
- $\{(10)^n : n \in \mathbb{N}\}$

Example

Giving a NFA like this, turn it into a DFA



NFA to DFA



- ✓ Formal Definition of NFA
- ✓ Recognizing Languages
- ✓ Equivalence of DFA and NFA
- ✓ Procedure: NFA-to-DFA

Theory of Computation

Equivalence of DFA and NFA



王 轩
Wang
Xuan

- ▶ Prove the equivalence of DFA and NFA
- ▶ Closure under the regular language
 - Union
 - Concatenation
 - Star Operation

- If a language is recognized by an NFA, then we must show the existence of a DFA that also recognizes it.
- How to construct the DFA?

- If k is the number of states of the NFA, it has 2^k subsets of states. Each subset corresponds to one of the possibilities that the DFA must remember, so the DFA simulating the NFA will have 2^k states.

Let $N = (Q, \Sigma, \delta, q_0, F)$ be the NFA recognizing language A, we construct a DFA $M = (Q', \Sigma, \delta', q'_0, F')$ recognizing A

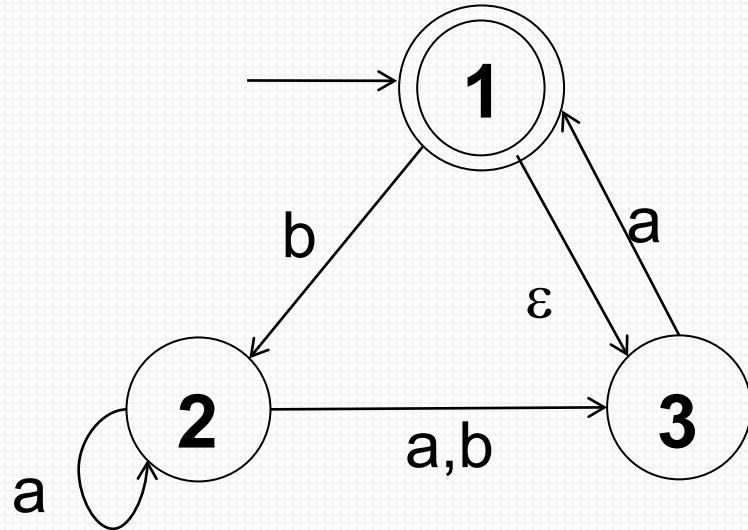
1. First consider no ϵ arrows:

- 1) $\Sigma_N = \Sigma_M$
- 2) $Q' = P(Q)$, that $P(Q)$ is the set of subsets of Q.
 - Each state of M is a set of states of N
- 3) For $R = \{r_1, r_2, r_3, \dots\}, R \in Q'$ and $a \in \Sigma$, let $\delta'(R, a) = \{q \in Q \mid q \in \delta(r_i, a)\}$ for some $r_i \in R\}$
 - $\delta'(R, a) = \bigcup_{r_i \in R} \delta(r_i, a)$
- 4) $q'_0 = \{q_0\}$
- 5) $F' = \{R \in Q' \mid R \text{ contains an accept state of } N\}$.

2. Consider ϵ arrows:

- Setting: $E(R)$ as the collection of states reached from R along ϵ arrows in M
- Modify transition function of M to add the states that can be reached by going along ϵ arrows after every step.
 - $\delta'(R, a) = \{q \in Q | q \in E(\delta(r, a)) \text{ for some } r \in R\}$
 - Start state of M including all possible state reached along ϵ arrows ,
 $q_0' = E(\{q_0\})$

- An NFA



- Construct a DFA M containing state set:
 $\{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$

- State sets in M

\emptyset

$\{1\}$

$\{2\}$

$\{1,2\}$

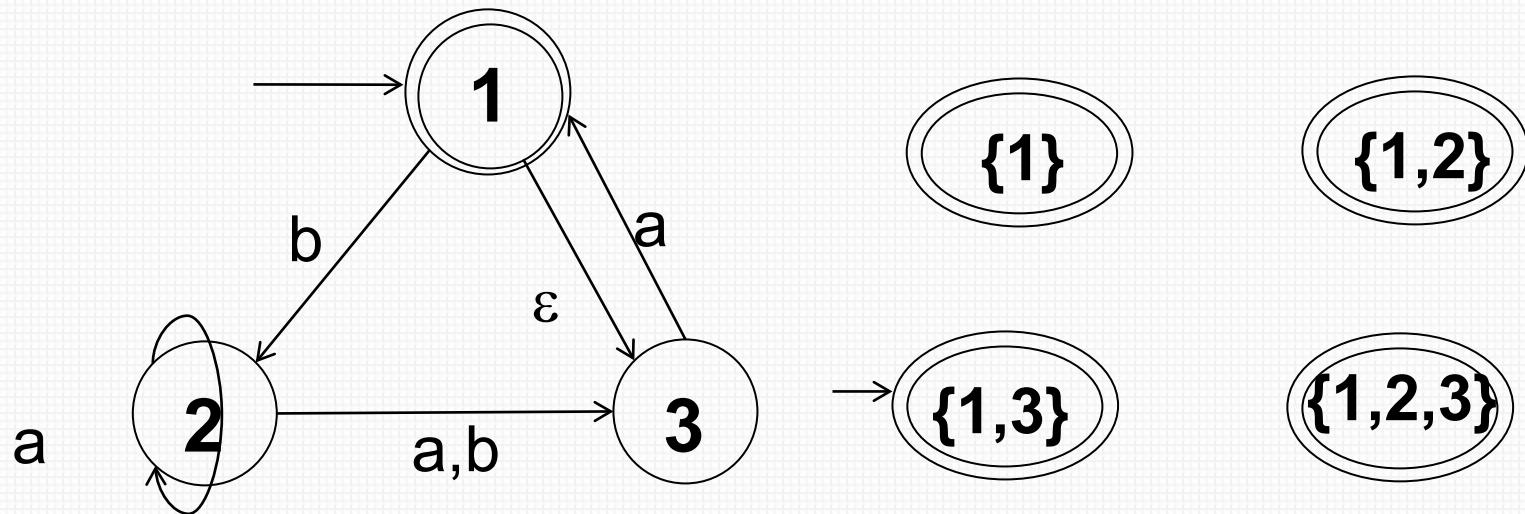
$\{3\}$

$\{1,3\}$

$\{2,3\}$

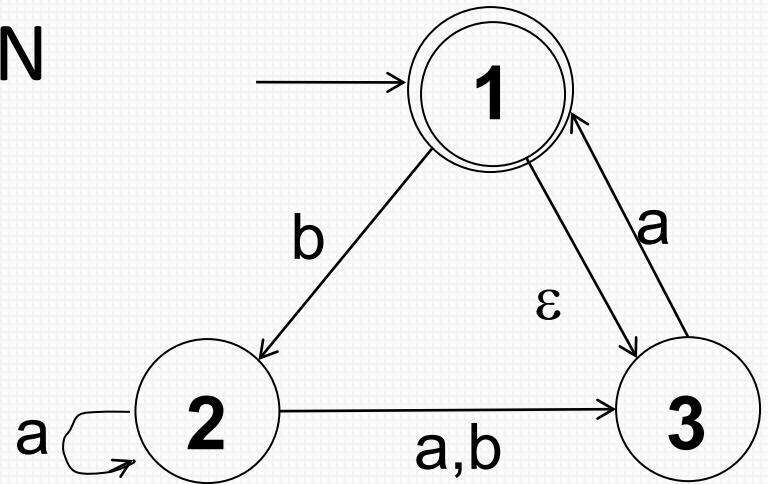
$\{1,2,3\}$

- First, determine the start and accept states of M.
 - Start state is $E(\{1\})$, including states reached from 1 along ϵ arrows and 1 itself. Thus, start state = $\{1,3\}$
 - New accept states contains N's accept state, $\{\{1\}, \{1,2\}, \{1,3\}, \{1,2,3\}\}$



- Determine each state in N

- $\delta(1,a)=\emptyset, \delta(1,b)=\{2\}$
- $\delta(2,a)=\{2,3\}, \delta(2,b)=\{3\}$
- $\delta(3,a)=\{1,3\}, \delta(3,b)=\emptyset$



► In DFA M:

- $\delta(1,a)=\emptyset, \delta(2,a)=\{2,3\}$
- $\delta(1,b)=\{2\}, \delta(2,b)=\{3\}$

so in M $\delta'(\{1,2\},a)=\{2,3\}$
 so in M $\delta'(\{1,2\},b)=\{2,3\}$

We can get the transition function $\delta'(\{1,2\},a)=\{2,3\}$ and $\delta'(\{1,2\},b)=\{2,3\}$ in M, the other states in M following same procedure:

$$\delta'(\emptyset,a)=\emptyset, \quad \delta'(\emptyset,b)=\emptyset, \quad \delta'(\{1\},a)=\emptyset, \quad \delta'(\{1\},b)=\{2\},$$

$$\delta'(\{2\},a)=\{2,3\}, \quad \delta'(\{2\},b)=\{3\}, \quad \delta'(\{3\},a)=\{1,3\},$$

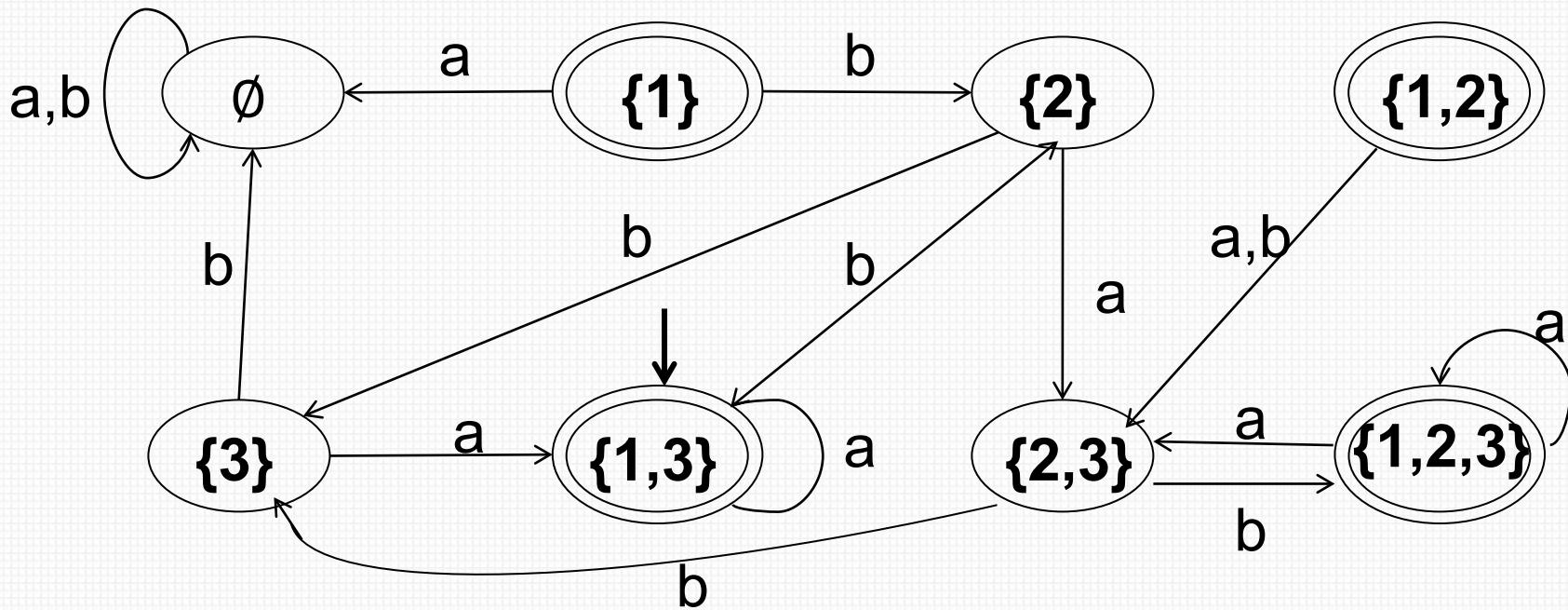
$$\delta'(\{3\},b)=\emptyset,$$

$$\delta'(\{1,2\},a)=\{2,3\}, \quad \delta'(\{1,2\},b)=\{2,3\}, \quad \delta'(\{1,3\},a)=\{1,3\},$$

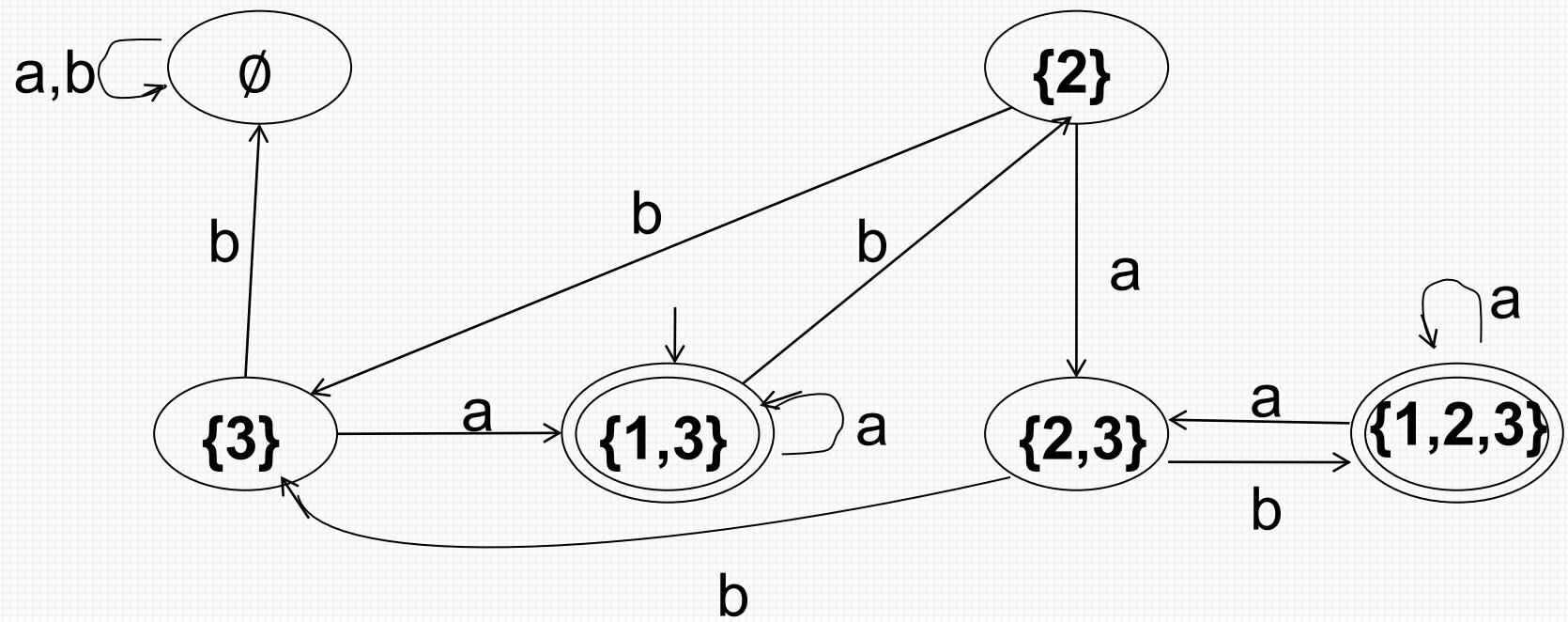
$$\delta'(\{1,3\},b)=\{2\}, \quad \delta'(\{2,3\},a)=\{1,2,3\}, \quad \delta'(\{2,3\},b)=\{3\},$$

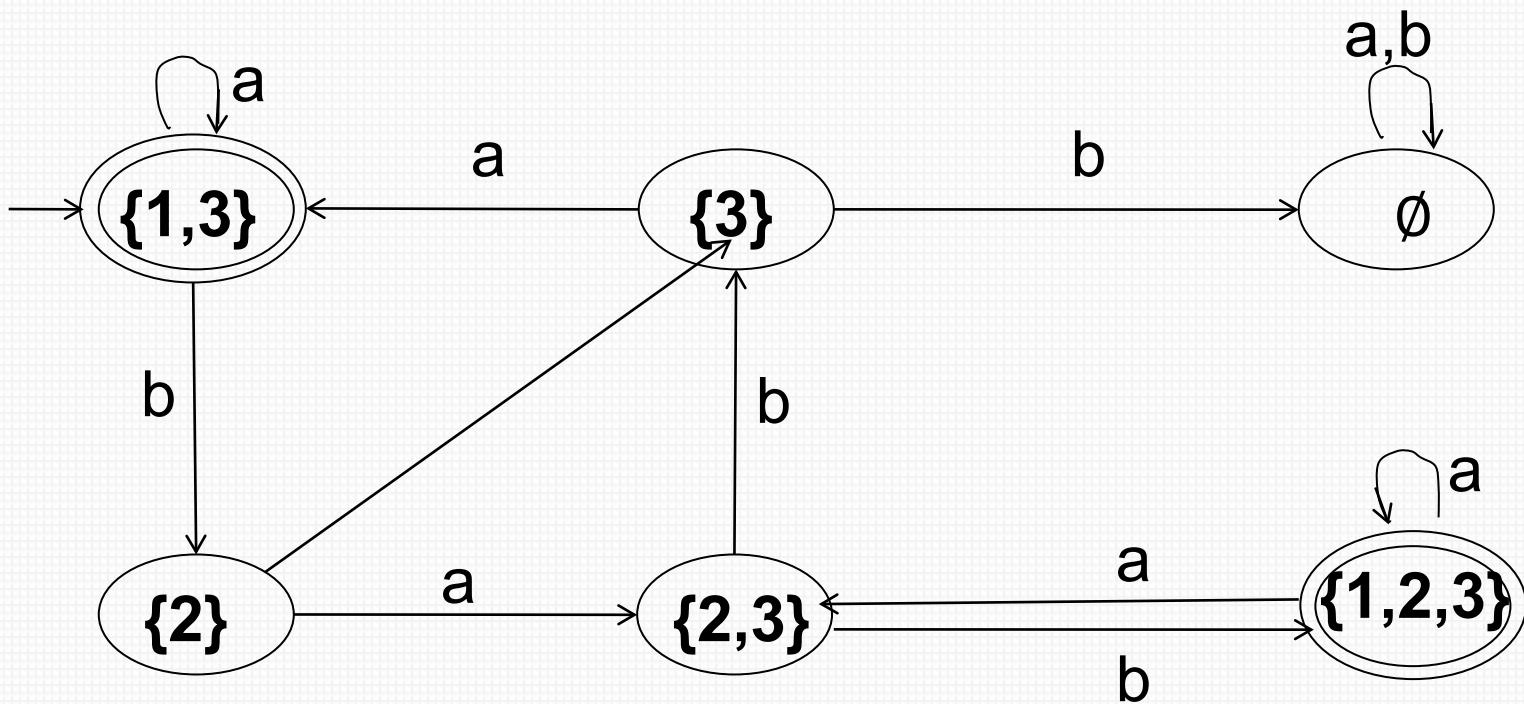
$$\delta'(\{1,2,3\},a)=\{1,2,3\}, \quad \delta'(\{1,2,3\},b)=\{2,3\}$$

- Obtaining



- Simplify DFA by removing the unnecessary states $\{1\}$ and $\{1,2\}$.





2. Closure under the regular language

- We have regular language L_1 and L_2 , we will prove that

Union:

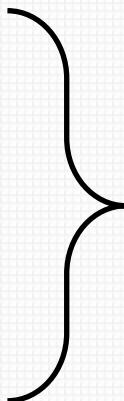
$$L_1 \cup L_2$$

Concatenation:

$$L_1 L_2$$

Star:

$$L_1^*$$



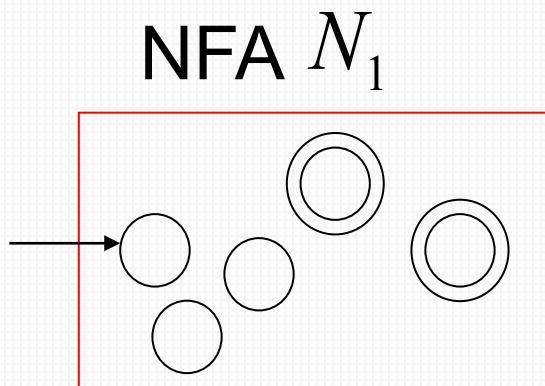
Are regular
Languages

In this lecture, we will use a new way to prove the theorems. Because the use of nondeterminism makes the proofs **much easier**.

Take two languages

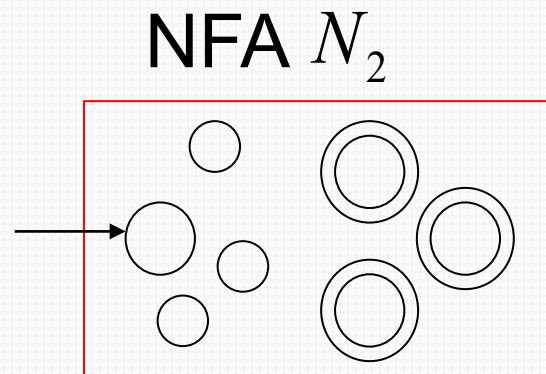
Regular language L_1

$$L(N_1) = L_1$$



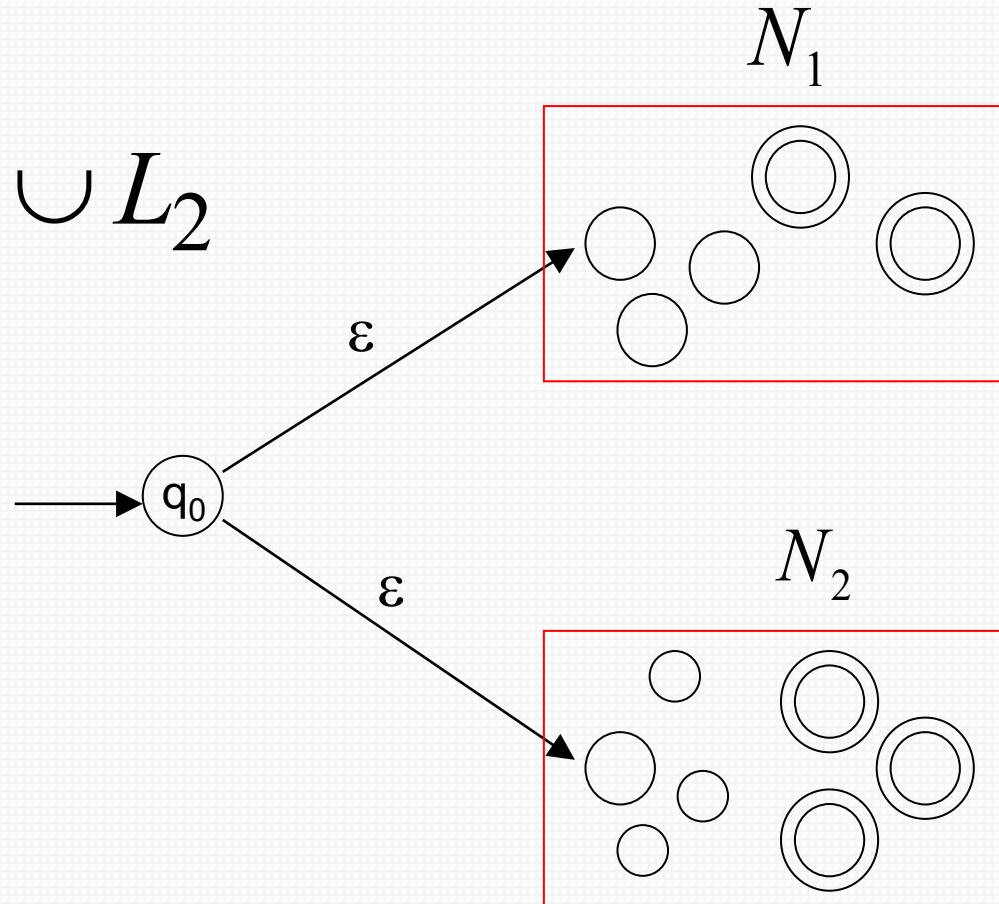
Regular language L_2

$$L(N_2) = L_2$$



NFA for

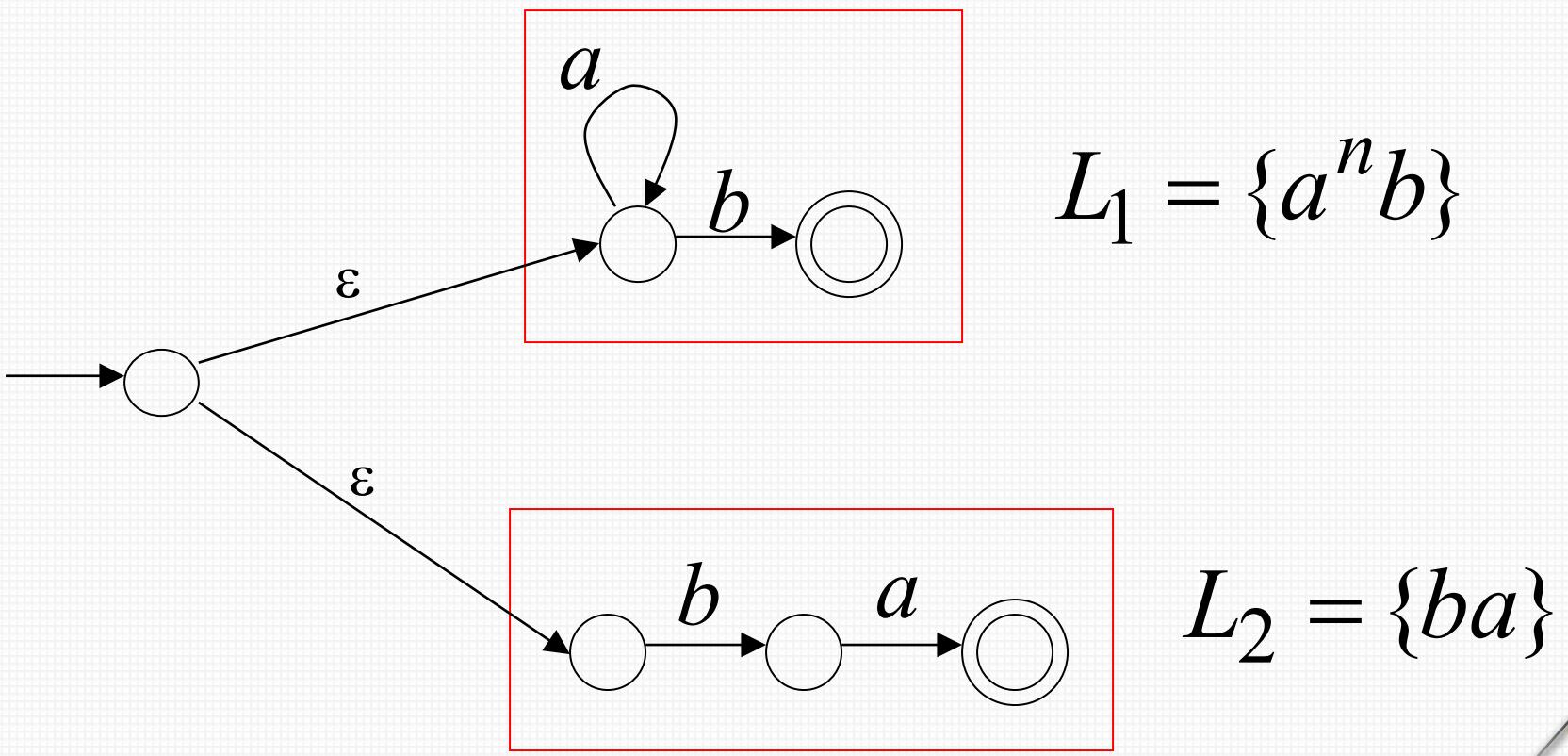
$$L_1 \cup L_2$$



- ▶ $N_1 = \{Q_1, \Sigma, \delta_1, q_1, F_1\}$ recognizes A_1 ,
- ▶ $N_2 = \{Q_2, \Sigma, \delta_2, q_2, F_2\}$ recognizes A_2
- ▶ Construct $N = \{Q, \Sigma, \delta, q_0, F\}$ to recognize $A_1 \cup A_2$
 - $Q = \{q_0\} \cup Q_1 \cup Q_2$
 - The state q_0 is the start state of N
 - The accept states $F = F_1 \cup F_2$
 - Define δ as
- $\delta(q, a) = \begin{cases} \delta_1(q, a) & q \in Q_1 \\ \delta_2(q, a) & q \in Q_2 \\ \{q_1, q_2\} & q = q_0 \text{ and } a = \varepsilon \\ \emptyset & q = q_0 \text{ and } a \neq \varepsilon \end{cases}$

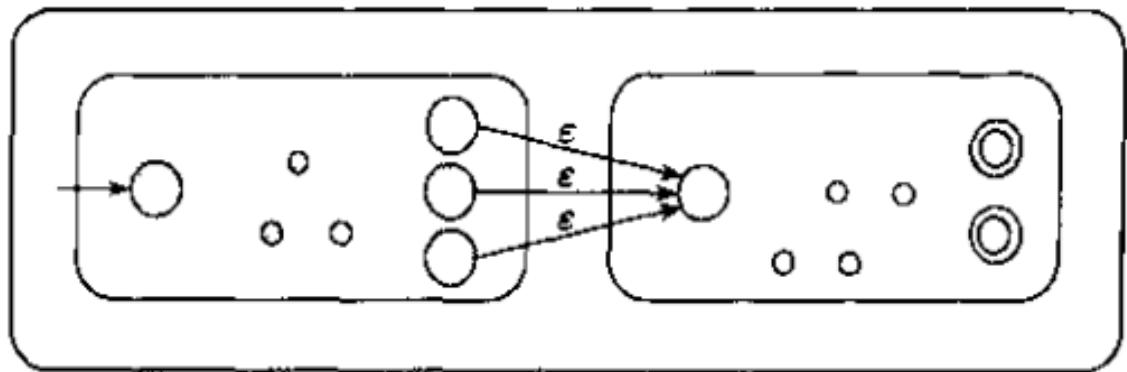
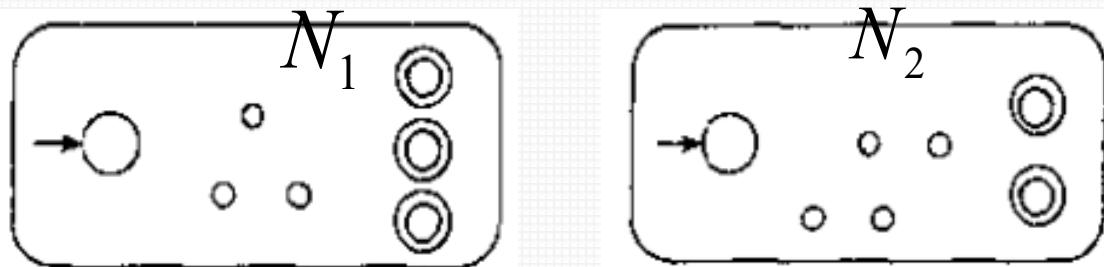
Example

NFA for $L_1 \cup L_2 = \{a^n b\} \cup \{ba\}$



2.2 Concatenation

NFA for $L_1 L_2$

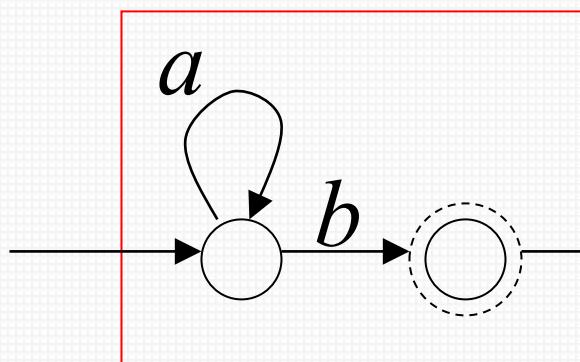


- ▶ $N_1 = \{Q_1, \Sigma, \delta_1, q_1, F_1\}$ recognizes A_1 , $N_2 = \{Q_2, \Sigma, \delta_2, q_2, F_2\}$ recognizes A_2
- ▶ Construct $N = \{Q, \Sigma, \delta, q_1, F_2\}$ to recognize $A_1 \cdot A_2$
 - $Q = Q_1 \cup Q_2$
 - The state q_1 is the same as the start state of N_1
 - The accept states F_2 are the same as the accept states of N_2
 - Define δ as
 - $\delta(q, a) = \begin{cases} \delta_1(q, a) & q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q, a) \\ \delta_1(q, a) \cup \{q_2\} & q \in F_1 \text{ and } a \neq \varepsilon \\ \delta_2(q, a) & q \in F_1 \text{ and } a = \varepsilon \\ & q \in Q_2 \end{cases}$

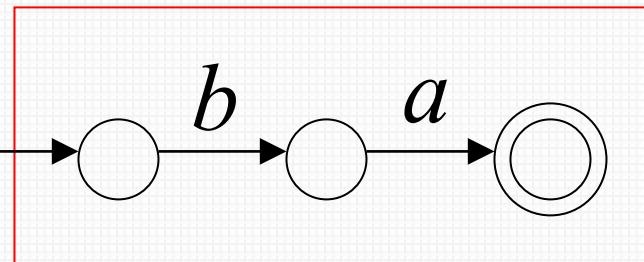
Example

NFA for $L_1 L_2 = \{a^n b\} \{ba\} = \{a^n bba\}$

$$L_1 = \{a^n b\}$$

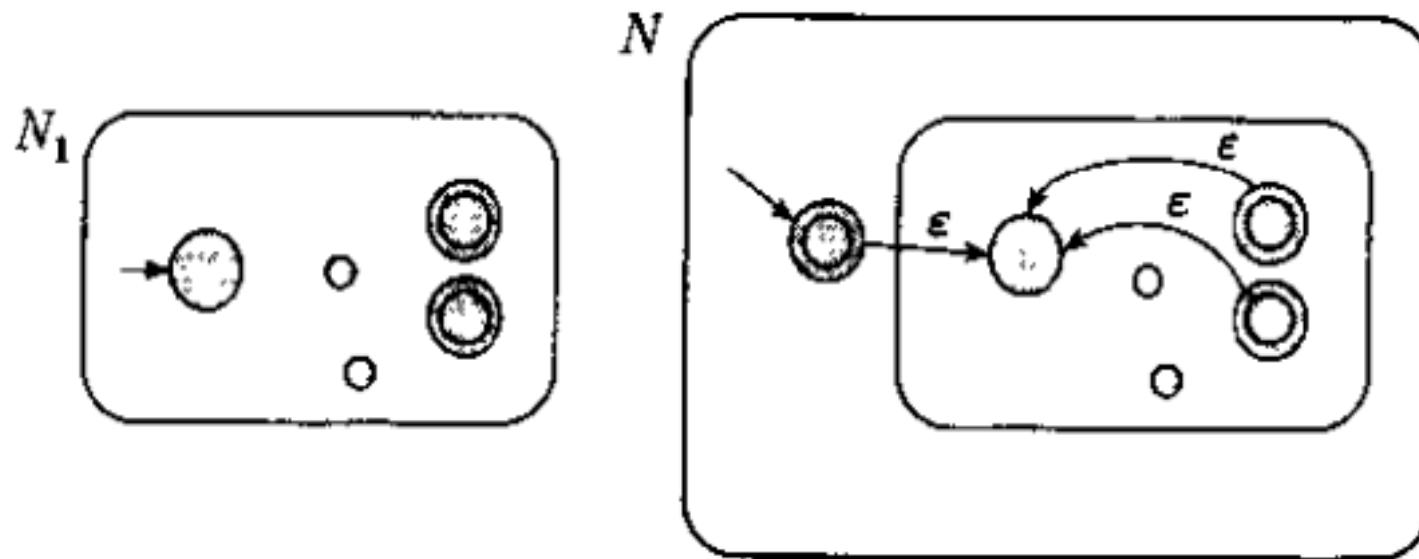


$$L_2 = \{ba\}$$



2.3 Star Operation

NFA for A^*

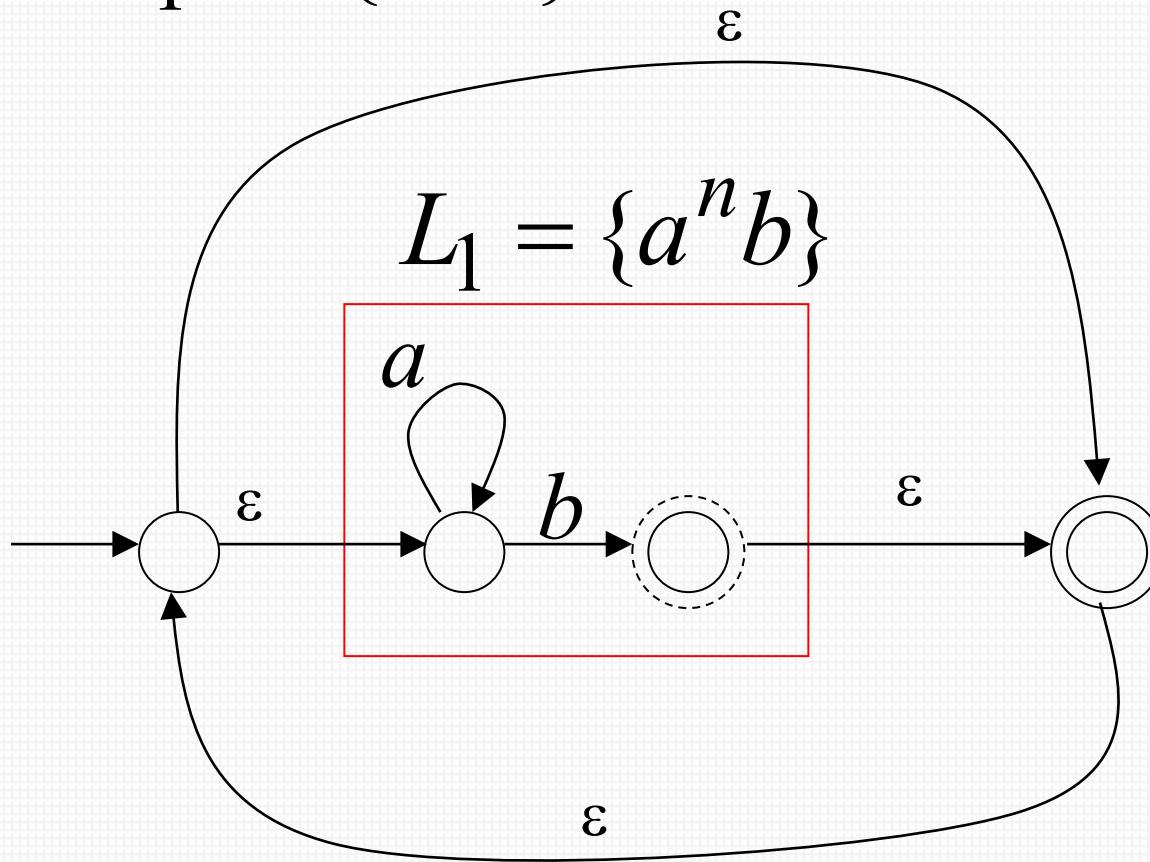


- ▶ $N_1 = \{Q_1, \Sigma, \delta_1, q_1, F_1\}$ recognizes A_1
- ▶ Construct $N = \{Q, \Sigma, \delta, q_0, F\}$ to recognize A_1^*
 - $Q = \{q_0\} \cup Q_1$
 - The state q_0 is the new start state
 - $F = \{q_0\} \cup F_1$
 - Define δ as

$$\circ \quad \delta(q, a) = \begin{cases} \delta_1(q, a) & q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q, a) & q \in F_1 \text{ and } a \neq \varepsilon \\ \delta_1(q, a) \cup \{q_1\} & q \in F_1 \text{ and } a = \varepsilon \\ \{q_1\} & q = q_0 \text{ and } a = \varepsilon \\ \emptyset & q = q_0 \text{ and } a \neq \varepsilon \end{cases}$$

Example

NFA for $L_1^* = \{a^n b\}^*$



Reversal:

$$\frac{L_1}{L_1}$$

Complement:

$$L_1 \cap L_2$$

Intersection:

- Are also closed under regular language.

- ✓ Prove the equivalence of DFA and NFA
- ✓ Closure under the regular language

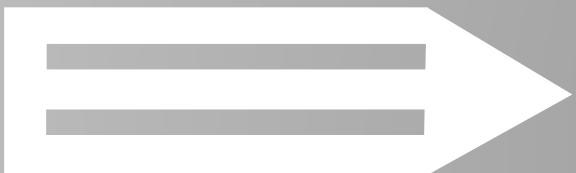
Union

Concatenation

Star Operation

Theory of Computation

Regular Expressions



王轩
Wang
Xuan

- Regular Expressions
- Prove Regular Language = Regular Expression
- Generalized NFA

- ▶ Regular expressions describe regular languages
- ▶ Example: $(0 \cup 1)0^*$
- ▶ Example: $(a + b \cdot c)^*$ describes the language $\{a, bc\}^* = \{\lambda, a, bc, aa, abc, bca, \dots\}$

- Given an alphabet Σ , R is a **regular expression** if R is
 - a, for $a \in \Sigma$
 - ϵ
 - \emptyset
 - $(R_1 \bullet R_2)$, with R_1 and R_2 regular expressions
 - $(R_1 \cup R_2)$, with R_1 and R_2 regular expressions
 - $R = (R_1^*)$, with R_1 a regular expression
 - Notice: do not confuse regular expressions ϵ and \emptyset

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

- 1. $0^*10^* = \{w \mid w \text{ has exactly a single } 1\}.$
- 2. $\Sigma^*1\Sigma^* = \{w \mid w \text{ has at least one } 1\}.$
- 3. $(\Sigma \Sigma \Sigma)^* = \{w \mid \text{the length of } w \text{ is a multiple of three}\}.$
- 4. $0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1 = \{w \mid w \text{ starts and ends with the same symbol}\}.$
- 5. $(0 \cup \varepsilon)1^* = 01^* \cup 1^*.$

The expression $0 \cup \varepsilon$ describes the language $\{0, \varepsilon\}$, so the concatenation operation adds either 0 or ε before every string in 1^* .

Regular Language = Regular Expression

Language={ ...|....}

Expression: $(a+b)^*C^*(d.g)^*$

$$\left\{ \begin{array}{l} \text{Languages} \\ \text{Generated by} \\ \text{Regular Expressions} \end{array} \right\} = \left\{ \begin{array}{l} \text{Regular} \\ \text{Languages} \end{array} \right\}$$

Proof: We need to prove both ways

$$\left\{ \begin{array}{l} \text{Languages} \\ \text{Generated by} \\ \text{Regular Expressions} \end{array} \right\} \subseteq \left\{ \begin{array}{l} \text{Regular} \\ \text{Languages} \end{array} \right\}$$

$$\left\{ \begin{array}{l} \text{Languages} \\ \text{Generated by} \\ \text{Regular Expressions} \end{array} \right\} \supseteq \left\{ \begin{array}{l} \text{Regular} \\ \text{Languages} \end{array} \right\}$$

LEMMA 1:

If a language is described by a regular expression,
then it is regular.

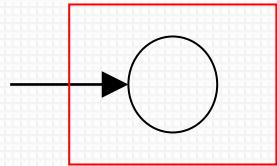
- Idea:

Regular expression R describing some language A . We show how to convert R into an **NFA** recognizing A .

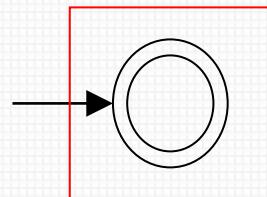
Primitive Regular Expressions:

\emptyset , ϵ , α

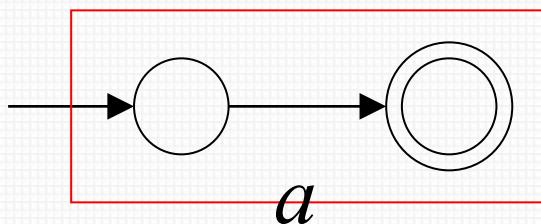
Corresponding NFAs



$$L(M_1) = \emptyset = L(\emptyset)$$



$$L(M_2) = \{\epsilon\} = L(\epsilon)$$



$$L(M_3) = \{a\} = L(a)$$

regular
languages

Inductive Hypothesis

Suppose that for regular expressions r_1 and r_2 , $L(r_1)$ and $L(r_2)$ are regular languages

We will prove:

$$\begin{aligned} L(r_1 \cup r_2) \\ L(r_1 \cdot r_2) \\ L(r_1 *) \end{aligned}$$



Are regular
Languages

By inductive hypothesis we know:

$L(r_1)$ and $L(r_2)$ are regular languages

We also know:

Regular languages are closed under:

Union

$L(r_1) \cup L(r_2)$

Concatenation

$L(r_1) L(r_2)$

Star

$(L(r_1))^*$

Therefore:

$$L(r_1 + r_2) = L(r_1) \cup L(r_2)$$

$$L(r_1 \cdot r_2) = L(r_1) L(r_2)$$

$$L(r_1^*) = (L(r_1))^*$$



Are regular
languages

End of Proof-Part 1

- Regular expression $(ab \cup a)^*$

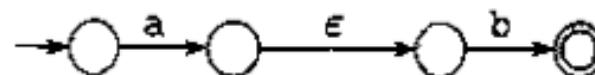
a



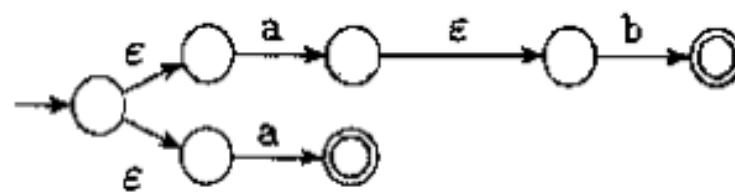
b



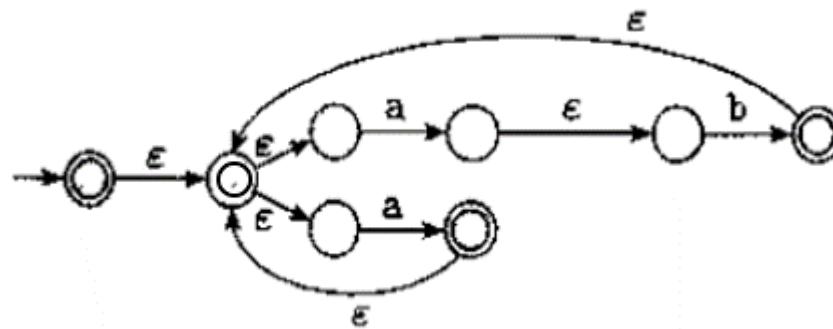
ab



$ab \cup a$



$(ab \cup a)^*$



LEMMA 2:

If a language is regular, then it is described by a regular expression.

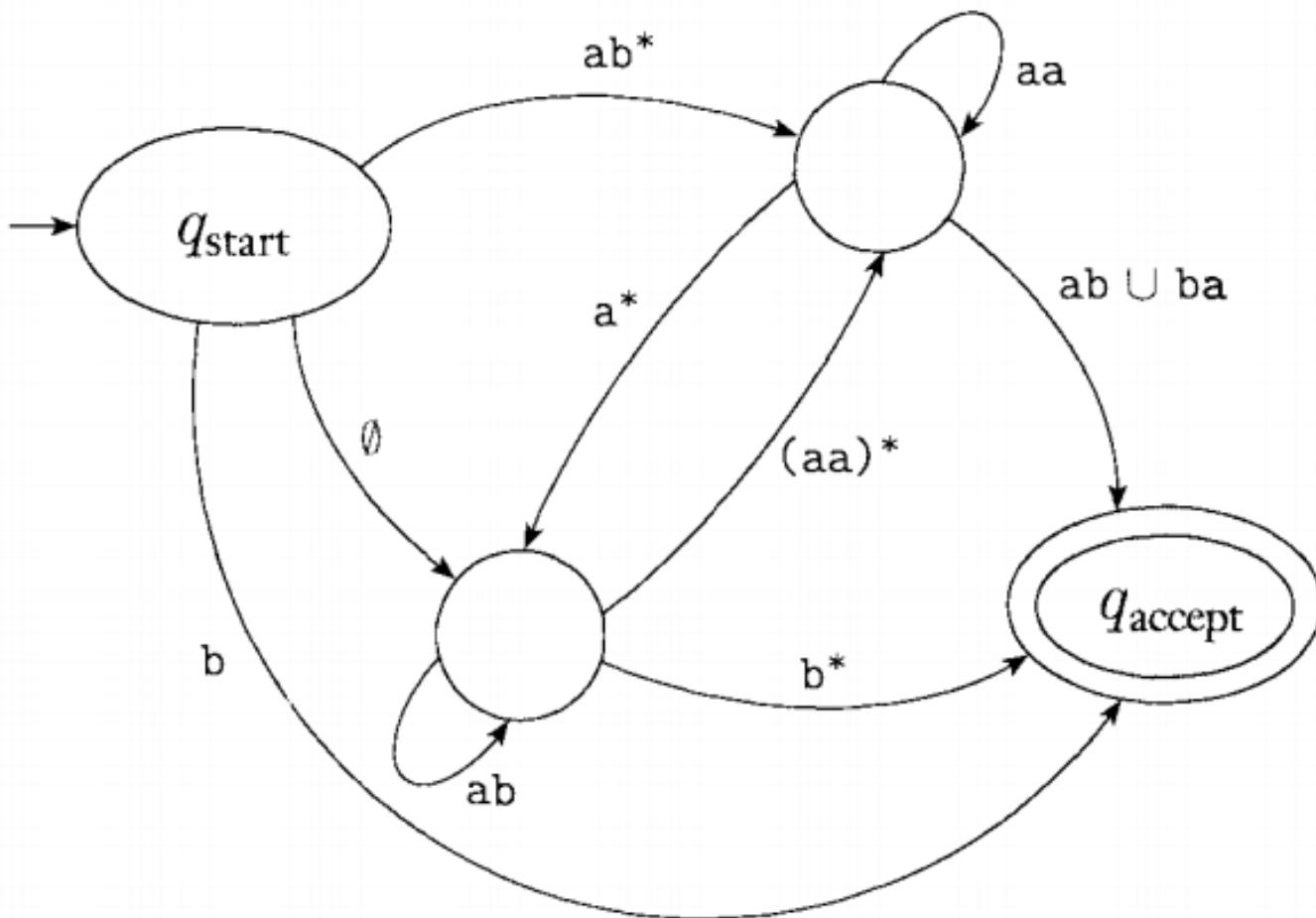
✓ Idea:

A procedure for converting DFAs into equivalent regular expressions.

- Two steps
 - 1 Regular Language is recognized by DFA . We can transform DFA to GNFA
 - 2 Transform GNFA to Regular Expression

- Generalized non-deterministic finite automaton
 $M=(Q, \Sigma, \delta, q_{start}, q_{accept})$ with
 - Q finite set of states
 - Σ the input alphabet
 - q_{start} the start state
 - q_{accept} the accept state
 - $\delta:(Q-\{q_{accept}\})\times(Q-\{q_{start}\}) \rightarrow R$ the transition function
 - (R is the set of regular expressions over Σ)
 - The only difference: the edge is Regular Expression in GNFA

Example GNFA

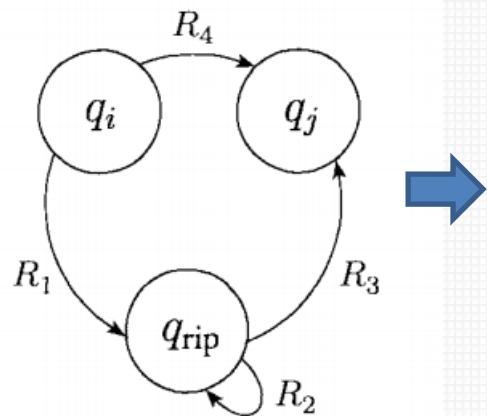


Characteristics of GNFA's δ

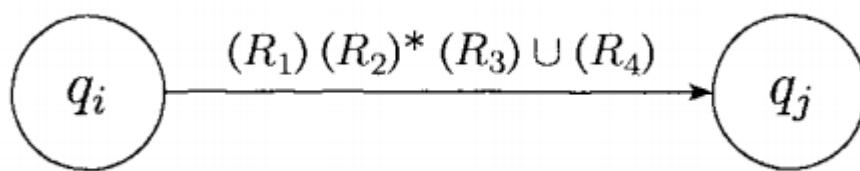
- The start state has transition arrows going to every other state but no arrows coming in from any other state.
- There is only a single accept state, and it has arrows coming in from every other state but no arrows going to any other state. Furthermore, the accept is not the same as the start state.
- Except for the start and accept state, one arrow goes from every state to every other and also from each state to itself.

Converting Example

before

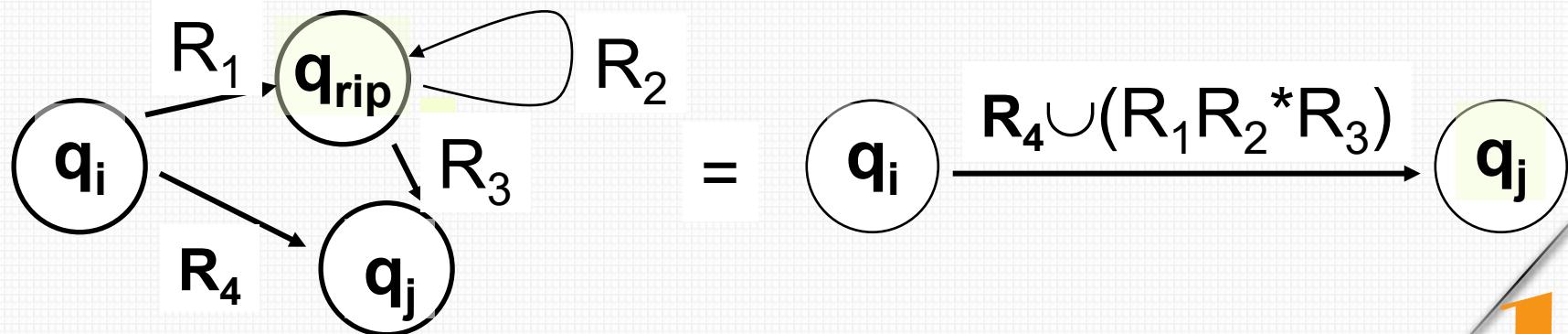


after



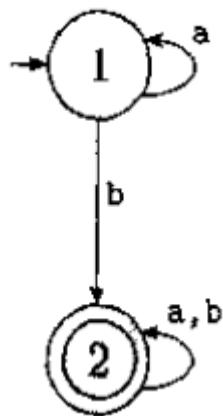
- 1. Let k be the number of states of G
- 2. If $k=2$, then G contains of a start state, an accept state and a single arrow connecting them and labeled with a regular expression R
- 3. If $k>2$, select $q_{rip} \in Q$ different from q_{start} and q_{accept} and let G' be $\text{GNFA}(Q', \Sigma, \delta', q_{start}, q_{accept})$, where
 - $Q' = Q - \{q_{rip}\}$

- And for any $q_i \in Q' - \{q_{\text{accept}}\}$ and any $q_j \in Q' - \{q_{\text{start}}\}$ let
 - $\delta'(q_i, q_j) = (R1)(R2)^*(R3) \cup (R4)$
- $R1 = \delta(q_i, q_{\text{rip}})$, $R2 = \delta(q_{\text{rip}}, q_{\text{rip}})$, $R3 = \delta(q_{\text{rip}}, q_j)$,
 $R4 = \delta(q_i, q_j)$
- 4. Computer convert(G') and return the value.

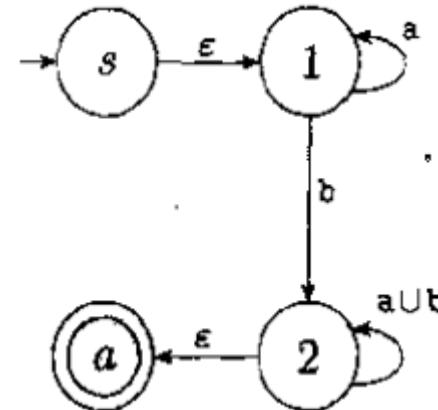


Example

- 1. First make a four state GNFA by adding a new start state and a new accept state
- Replace the label a, b on the self loop at state 2 with label $a \cup b$ (figure b)



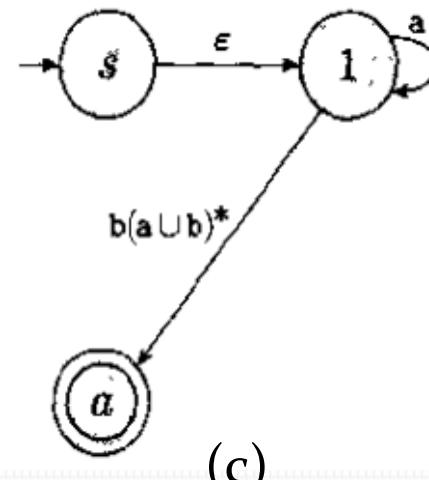
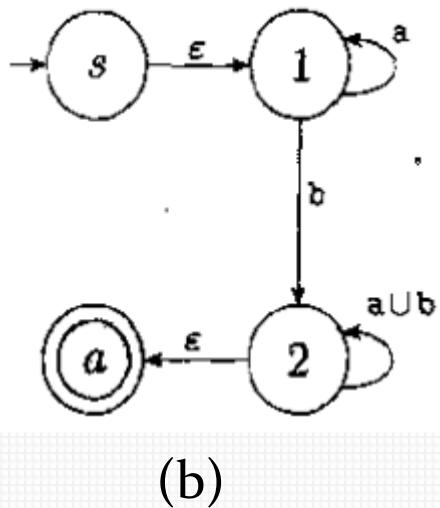
(a)



(b)

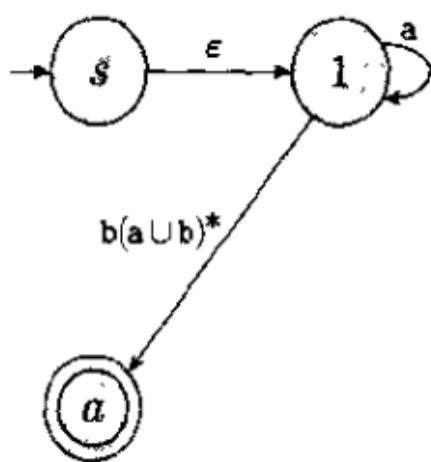
Example

- 2. Remove state 2, and update the remaining arrow labels(figure c)
- State q_i is state 1, state q_j is a, and q_{rip} is 2, so $R1=b$, $R2=a \cup b$, $R3=\epsilon$, $R4=\emptyset$. Therefore the new label on the arrow from 1 to a is $(b)(a \cup b)^*(\epsilon) \cup \emptyset$, simplify as $(b)(a \cup b)^*$.

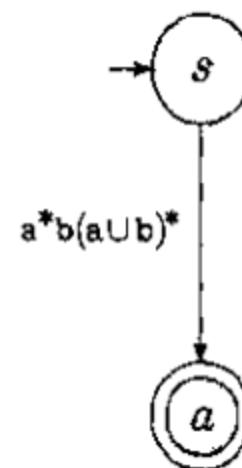


Example

- 3. Follow same procedure and remove state 1 (figure 4).

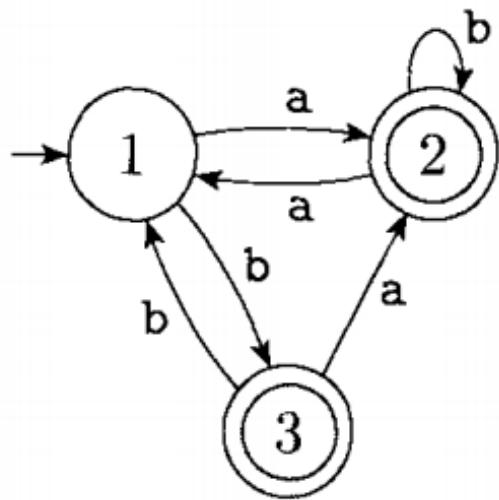


(c)

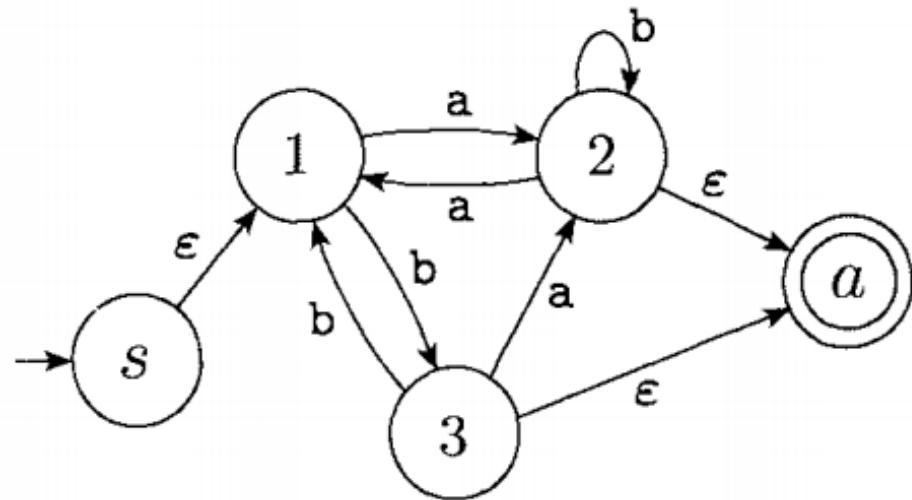


(d)

- Add new start state and new end state

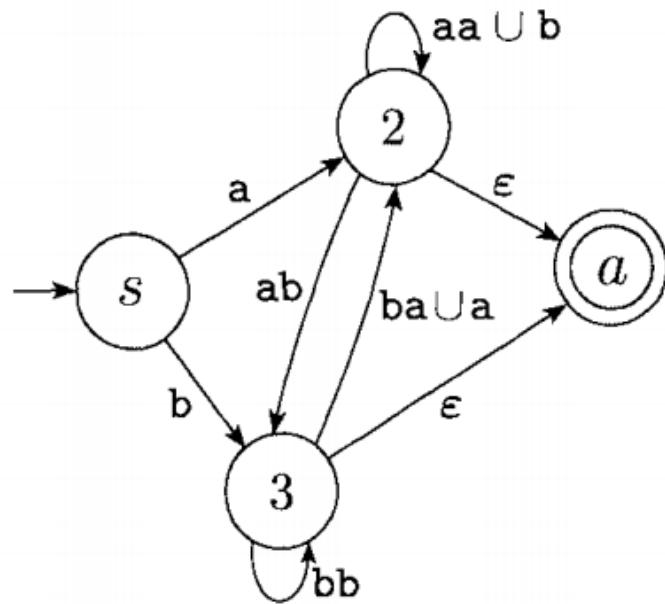


(a)

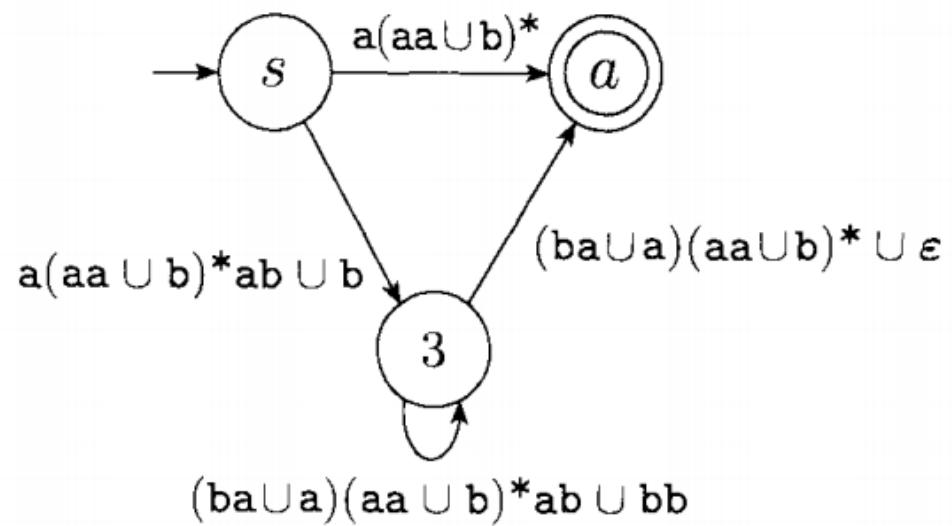


(b)

- Remove state 1 and state 2



(c)



(d)

- Remove state 3 and obtain the regular expression


$$(a(aa \cup b)^*ab \cup b)((ba \cup a)(aa \cup b)^*ab \cup bb)^*((ba \cup a)(aa \cup b)^* \cup \epsilon) \cup a(aa \cup b)^*$$

(e)

Language={ ..|.... }

Experssion: $(a+b)^*C + (d.g)^*$

$$\left\{ \begin{array}{l} \text{Languages} \\ \text{Generated by} \\ \text{Regular Expressions} \end{array} \right\} = \left\{ \begin{array}{l} \text{Regular} \\ \text{Languages} \end{array} \right\}$$

- ✓ Regular Expressions
- ✓ Prove Regular Language = Regular Expression
- ✓ Generalized NFA

Theory of Computation

Nonregular Languages



王轩
Wang
Xuan

- Non-regular languages
- The Pigeonhole Principle
- Pumping lemma
- Applications of the Pumping Lemma

Regular or Nonregular?

- Consider two languages over the alphabet $\Sigma = \{0,1\}$

Non-regular languages

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

Regular languages

$\{w \mid w \text{ has equal number of occurrences of } 01 \text{ and } 10 \text{ as substrings}\}$

How can we prove that a language L is not regular?

Prove that there is no DFA or NFA or RE that accepts L

Difficulty: this is not easy to prove

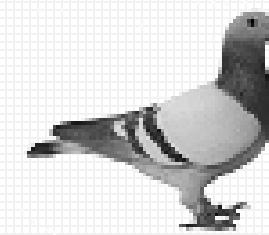
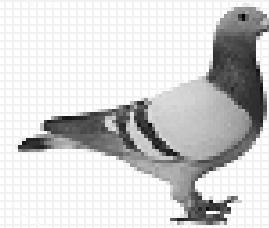
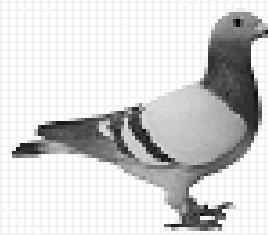
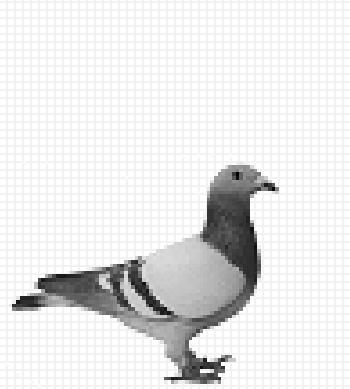
(since there is an infinite number of them)

Solution: Use the Pumping Lemma.

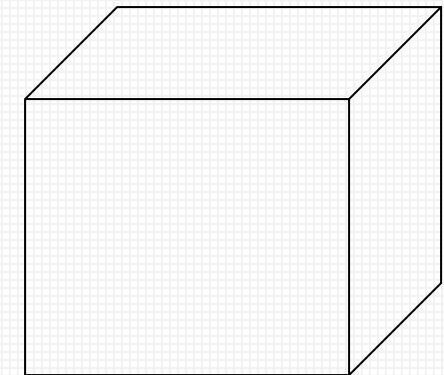
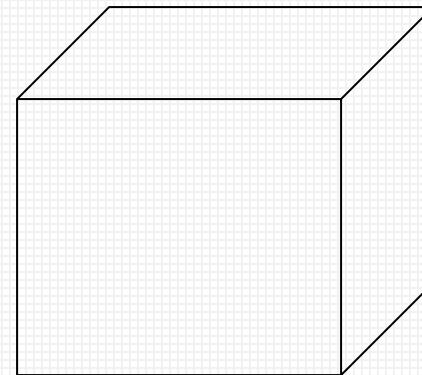
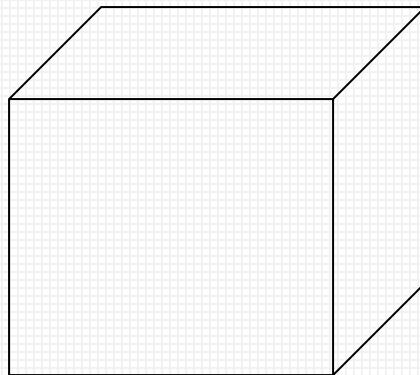
The Pigeonhole Principle

Before we learn the Pumping lemma, we should learn the Pigeonhole Principle at first.

The Pigeonhole Principle



4 pigeons

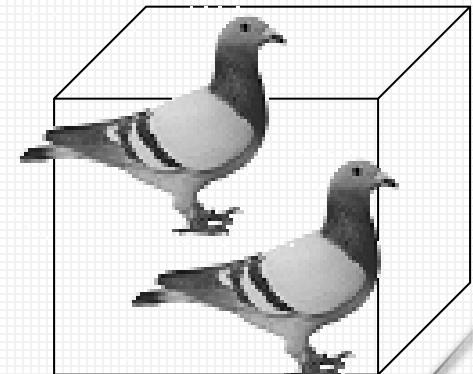
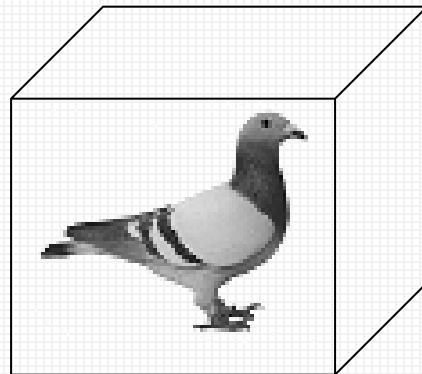
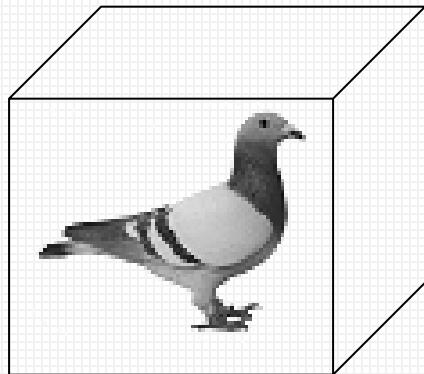


3 pigeonholes

The Pigeonhole Principle

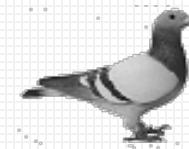
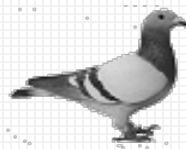
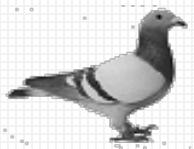
If we want to put every pigeon in a pigeonhole.

A pigeonhole must contain at least two pigeons.

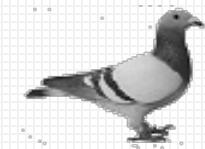


The Pigeonhole Principle

n pigeons

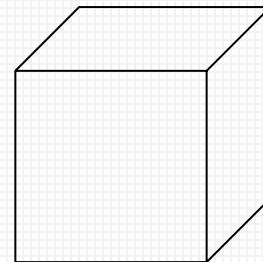
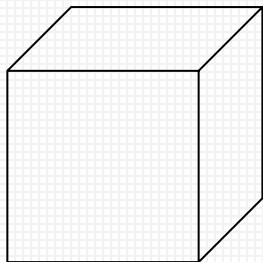


.....

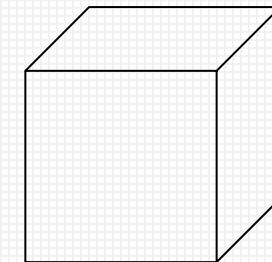


m pigeonholes

$n > m$



.....

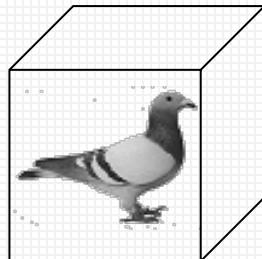
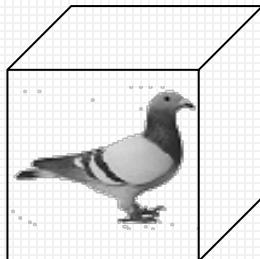


The Pigeonhole Principle

n pigeons

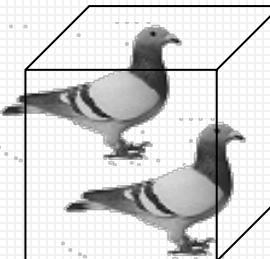
m pigeonholes

$$n > m$$

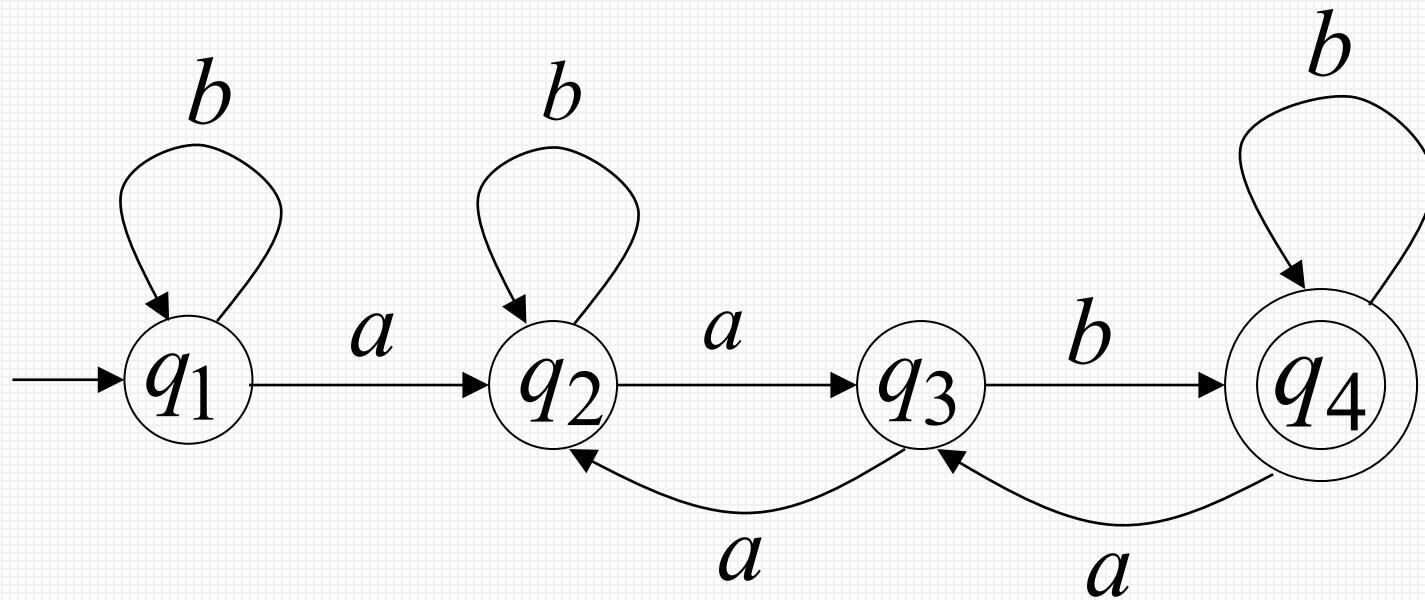


.....

There is a pigeonhole
with at least 2 pigeons

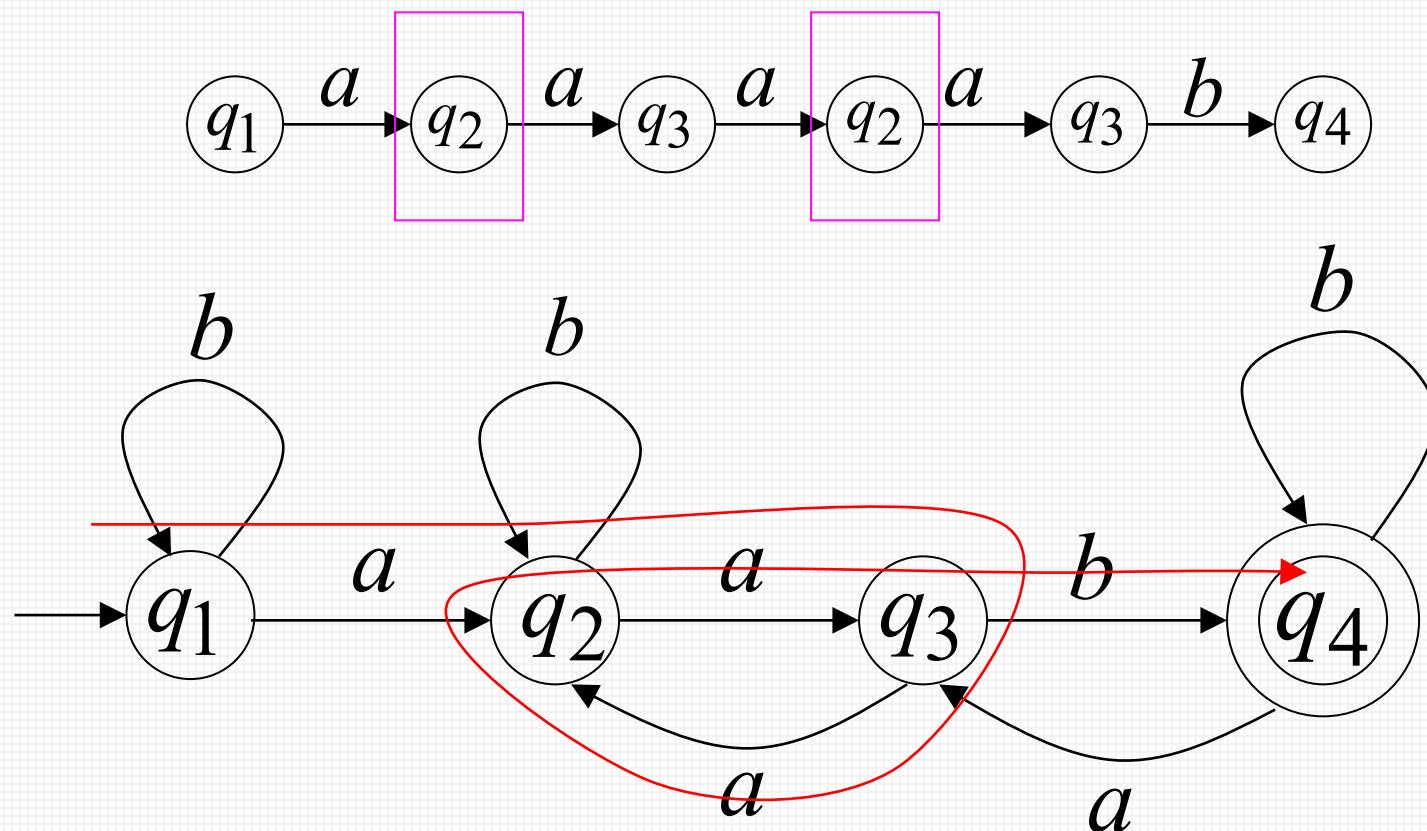


Consider a DFA with 4 states



Consider the walk of a “long” string: **aaaab**
(length at least 4)

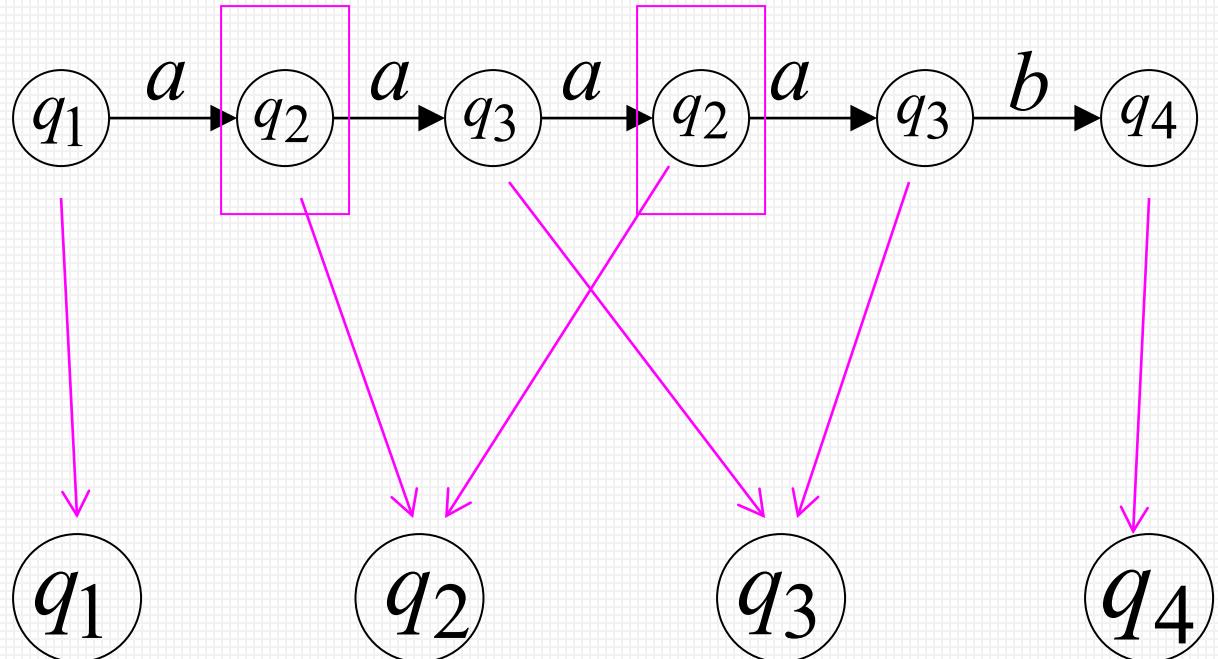
A state is repeated in the walk of **aaaab**



The state is repeated as a result of
the pigeonhole principle

Walk of **aaaab**

Pigeons:
(walk states)



Pigeonholes:
(Automaton states)

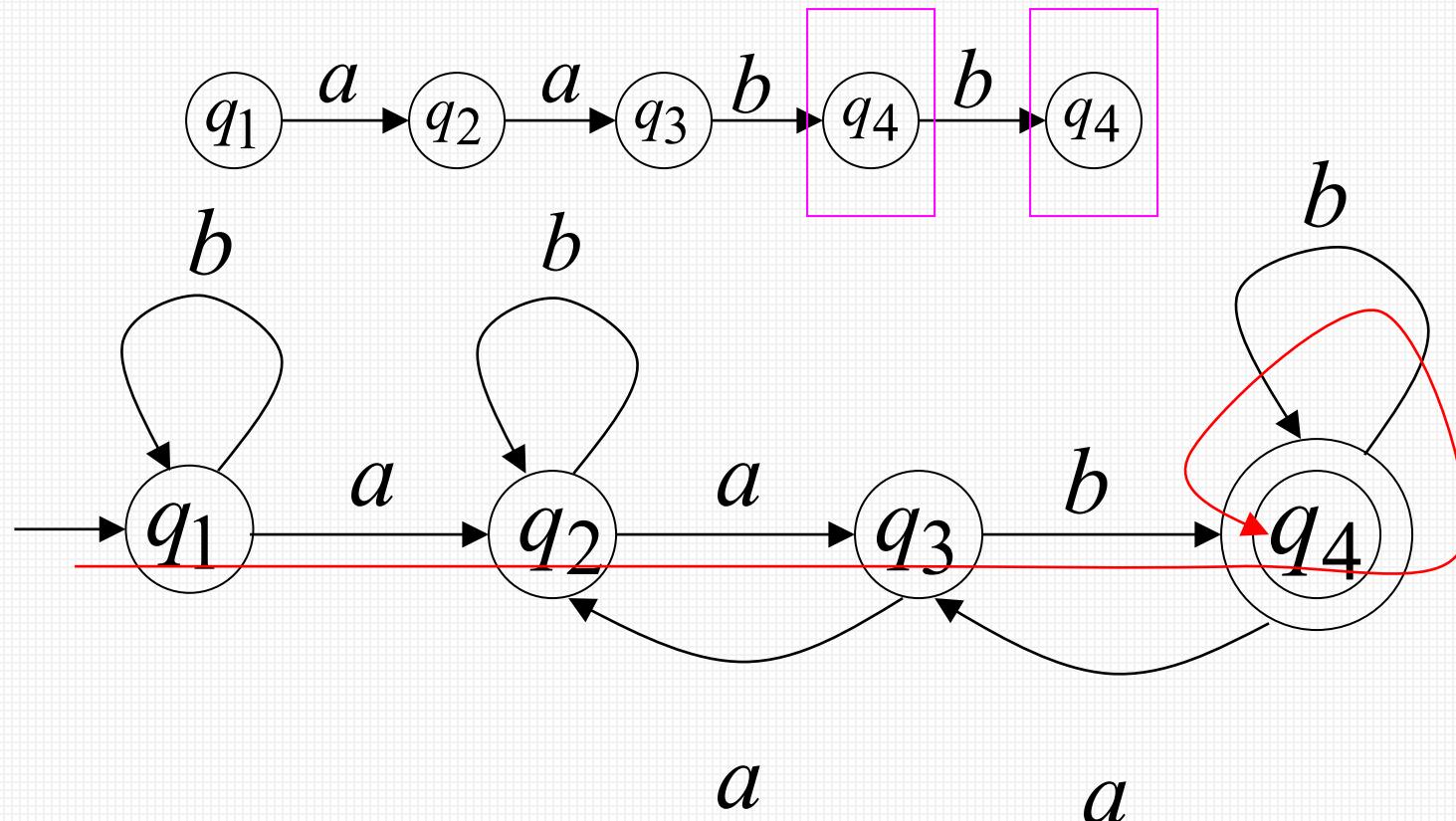
Repeated
state

Repeated
state

Consider the walk of a "long" string: aabb
(length at least 4)

Due to the pigeonhole principle:

A state is repeated in the walk of aabb



The state is repeated as a result of the pigeonhole principle:

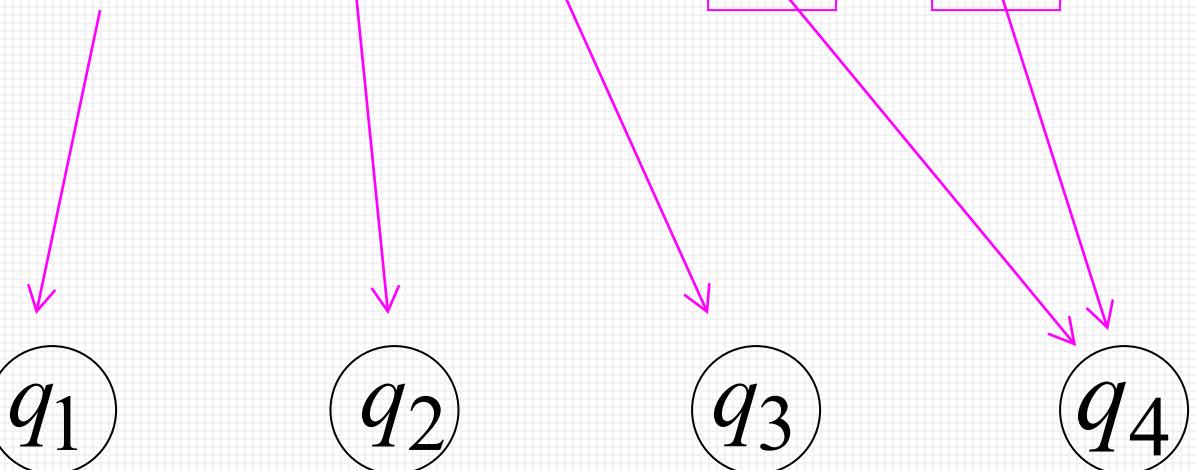
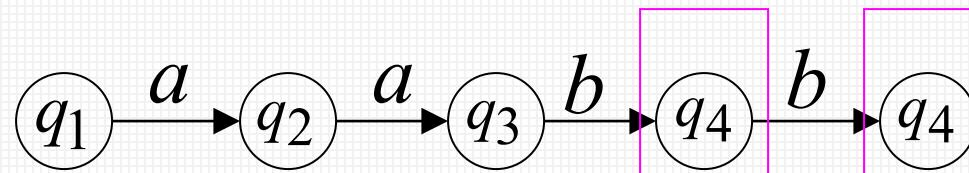
Pigeons:
(walk states)

Pigeonholes :
(Automaton states)

Automaton States

Repeated
state

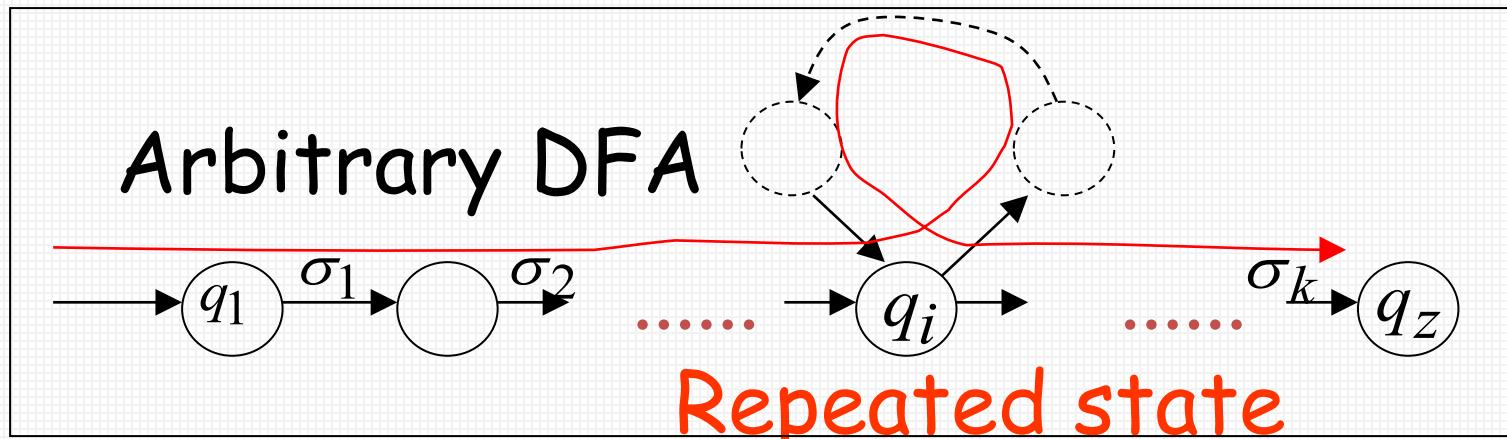
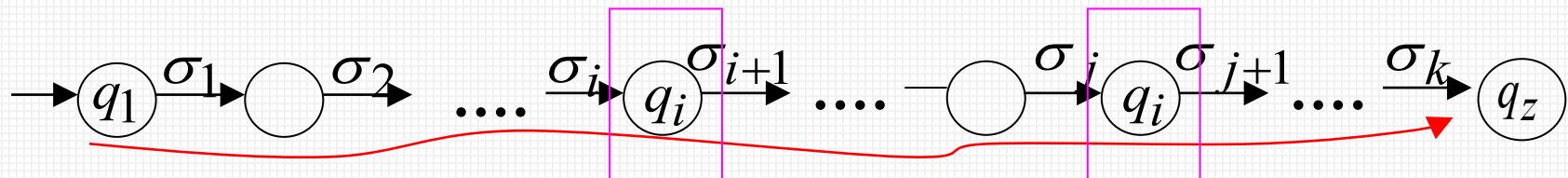
Walk of **aabb**



In General:

If $|w| \geq$ the number of states of DFA, by the pigeonhole principle, a state is repeated in the walk w

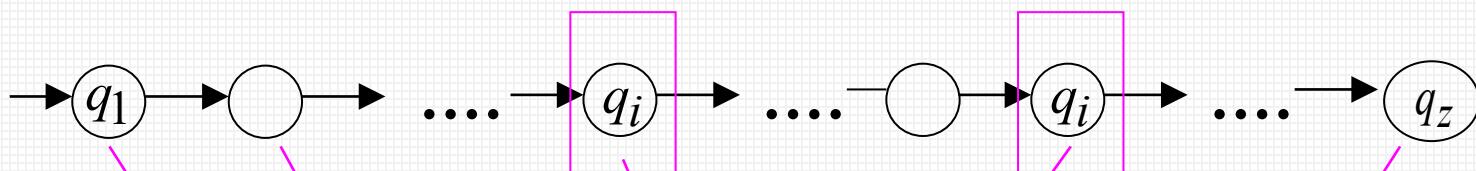
Walk of $w = \sigma_1 \sigma_2 \cdots \sigma_k$



Let m be the number of states of DFA, $|w| \geq m$

Number of states in walk is at least $m+1$

Pigeons: (walk states)



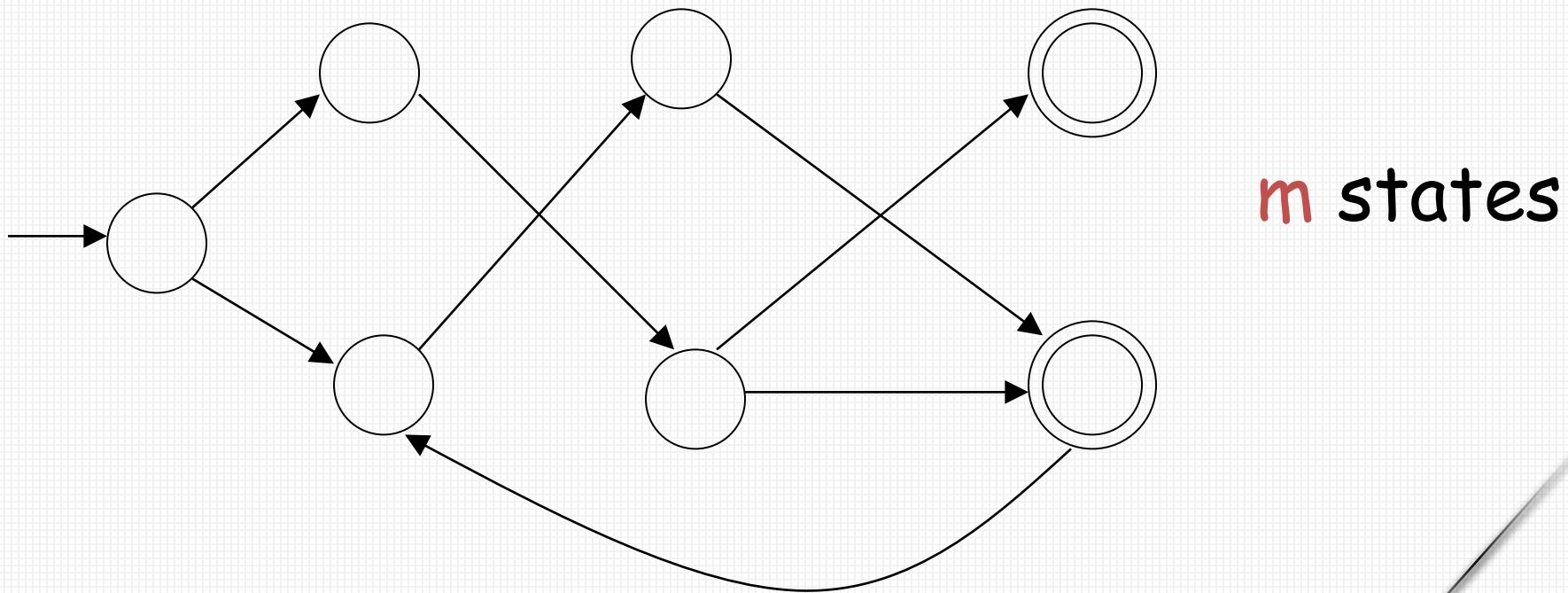
Walk of w

Pigeonholes: $q_1, q_2, \dots, q_{m-1}, q_m$
(Automaton states)

A state is repeated

Pumping Lemma

Take an infinite regular language L
(contains an infinite number of strings)
There exists a DFA that accepts L

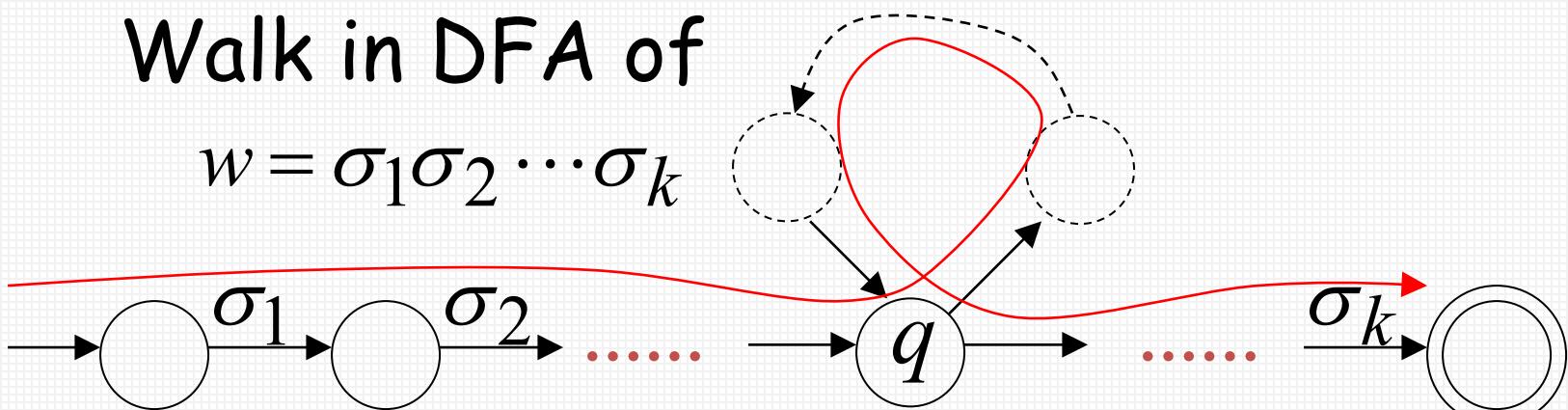


Take string $w \in L$ with $|w| \geq m$ (m is the number of states of DFA)

then, at least one state is repeated in the walk of w

Walk in DFA of

$$w = \sigma_1 \sigma_2 \cdots \sigma_k$$

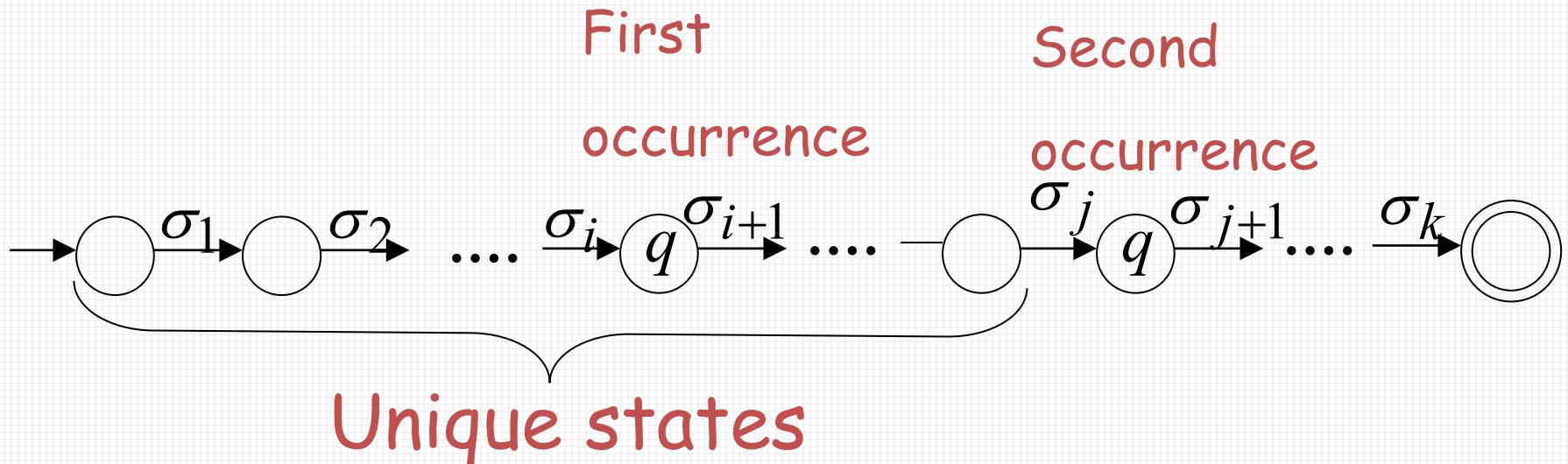


Repeated state in DFA

There could be many states repeated

Take q to be the first state repeated

One dimensional projection of walk w :



We can write $w = xyz$

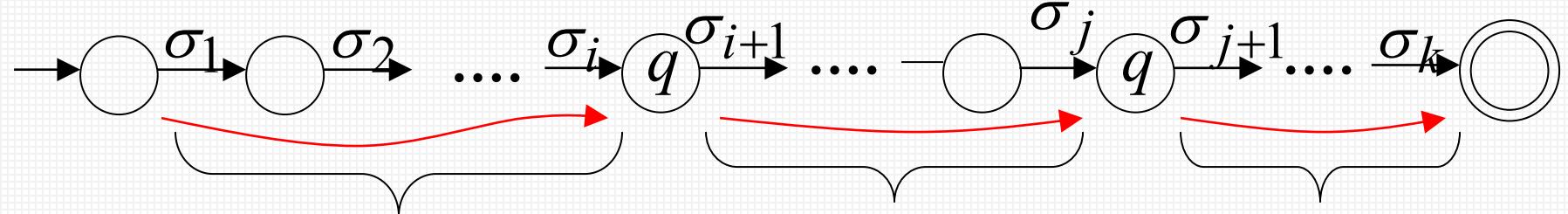
One dimensional projection of walk w :

First

Second

occurrence

occurrence



$$x = \sigma_1 \cdots \sigma_i$$

$$y = \sigma_{i+1} \cdots \sigma_j$$

$$z = \sigma_{j+1} \cdots \sigma_k$$

x is the substring before the two q.

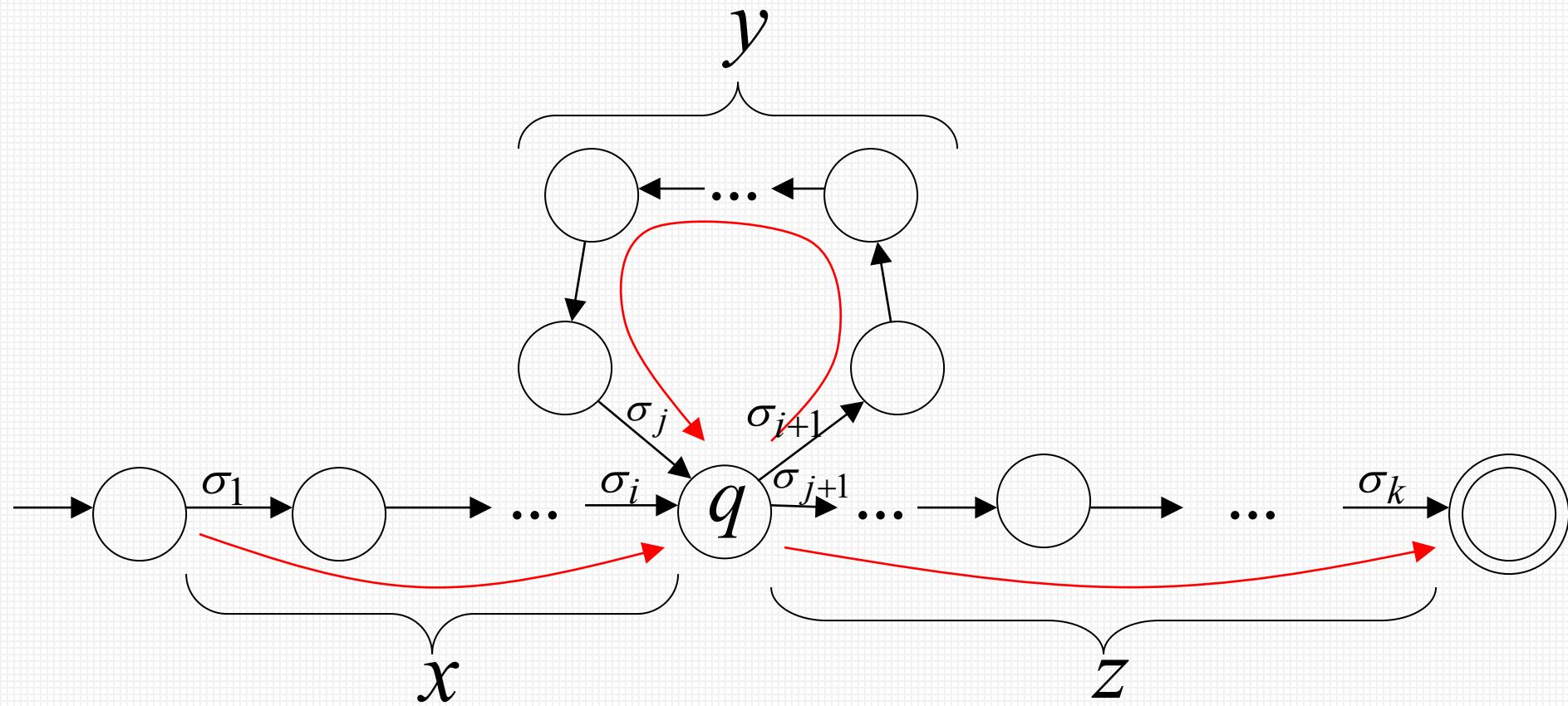
z is the substring of the other parts.

y is the substring between the two q.

In DFA:

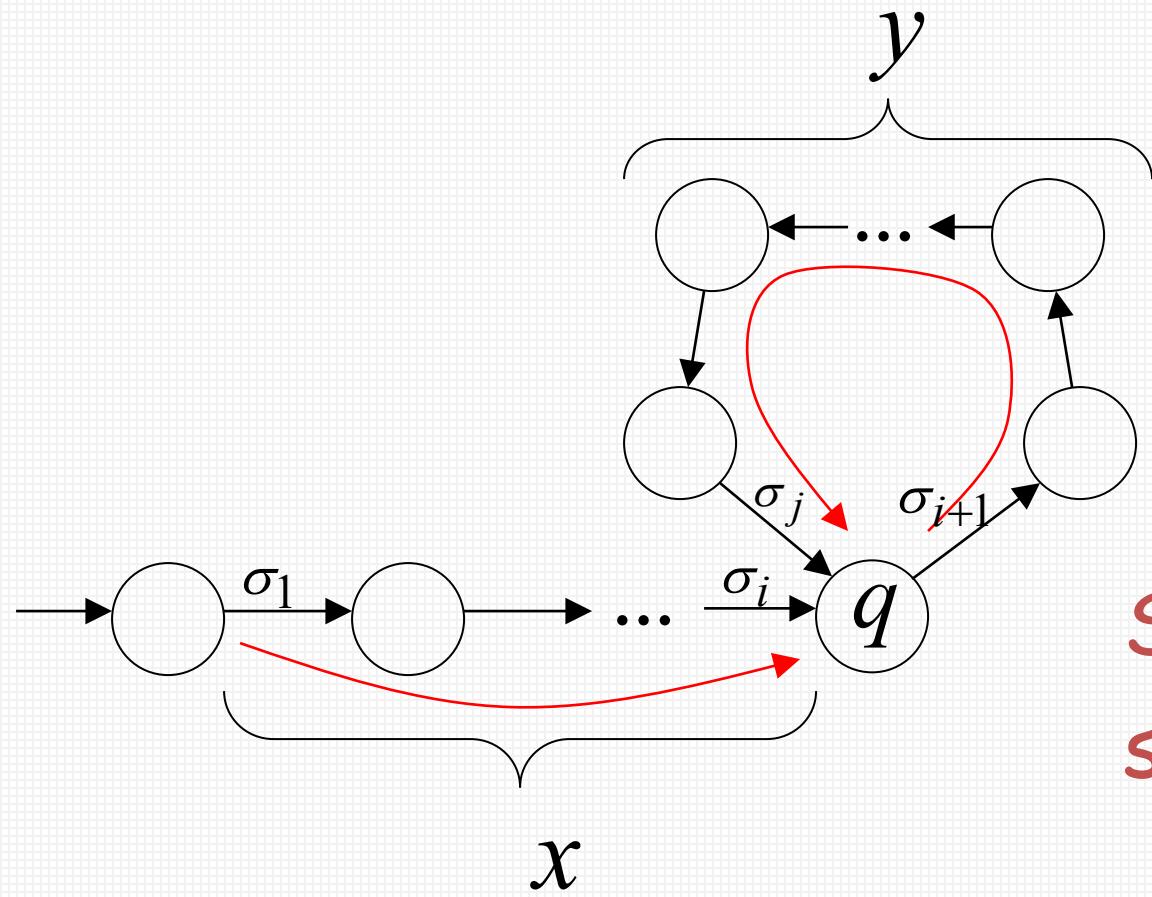
$$w = x \ y \ z$$

Where y corresponds to substring
between first and second occurrence of q



Observation: Length $|xy| \leq m$

m is the number of states of DFA



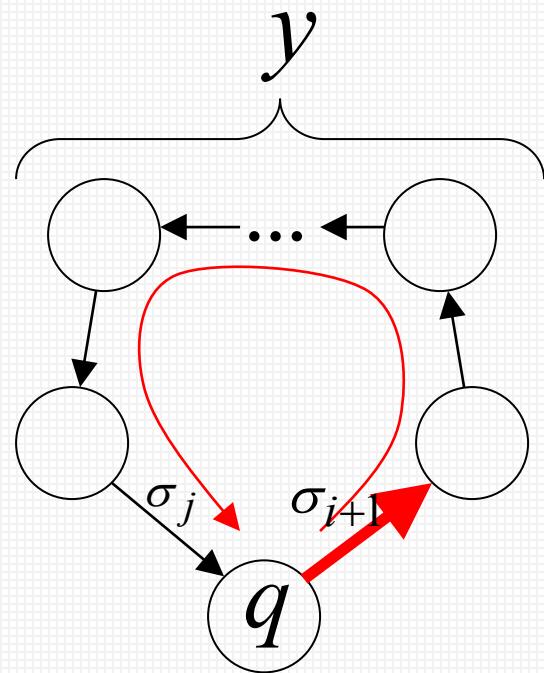
Because of
unique states in xy

Since, in xy no
state is repeated

(except q)

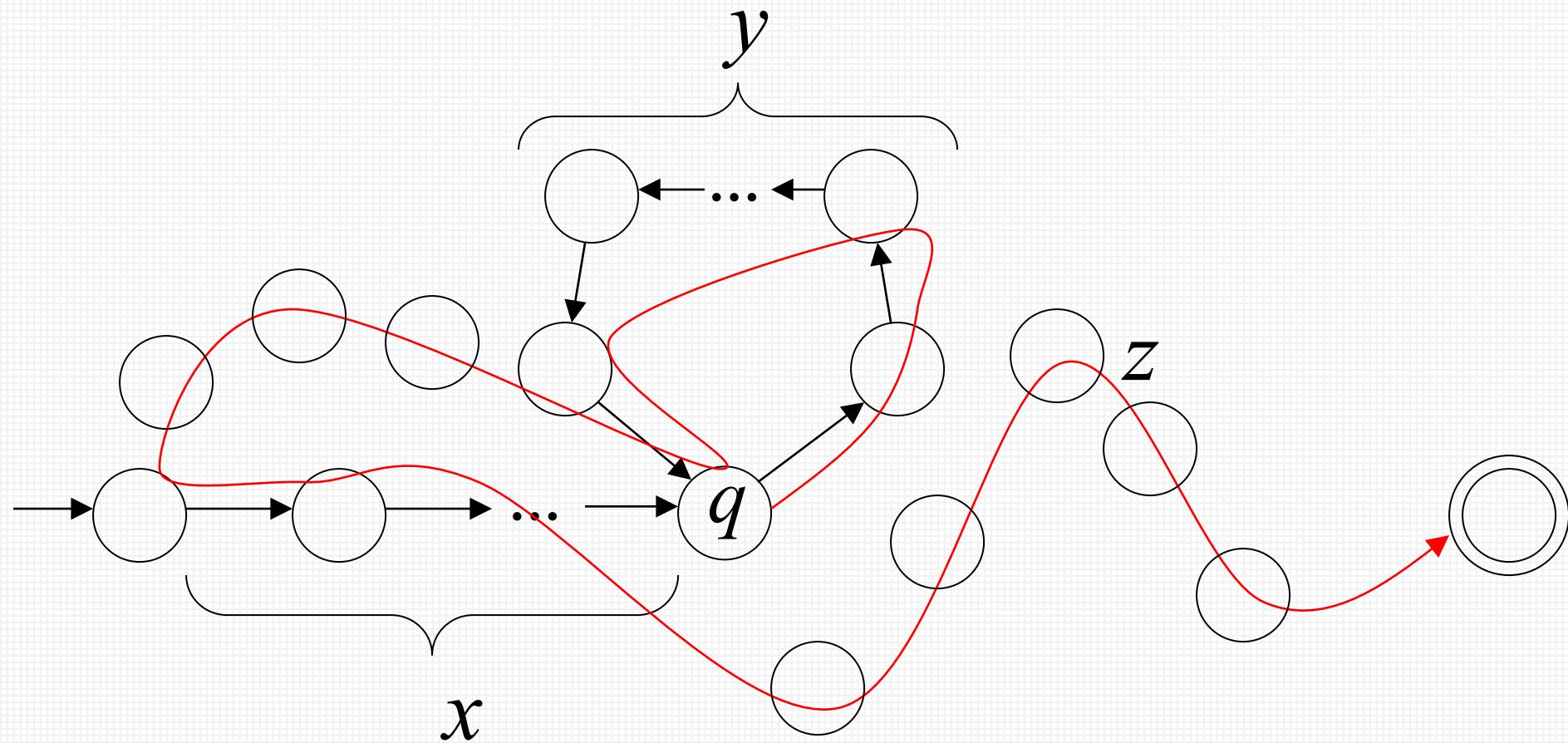
Observation: $\text{length } |y| \geq 1$

Since there is at least one transition in loop



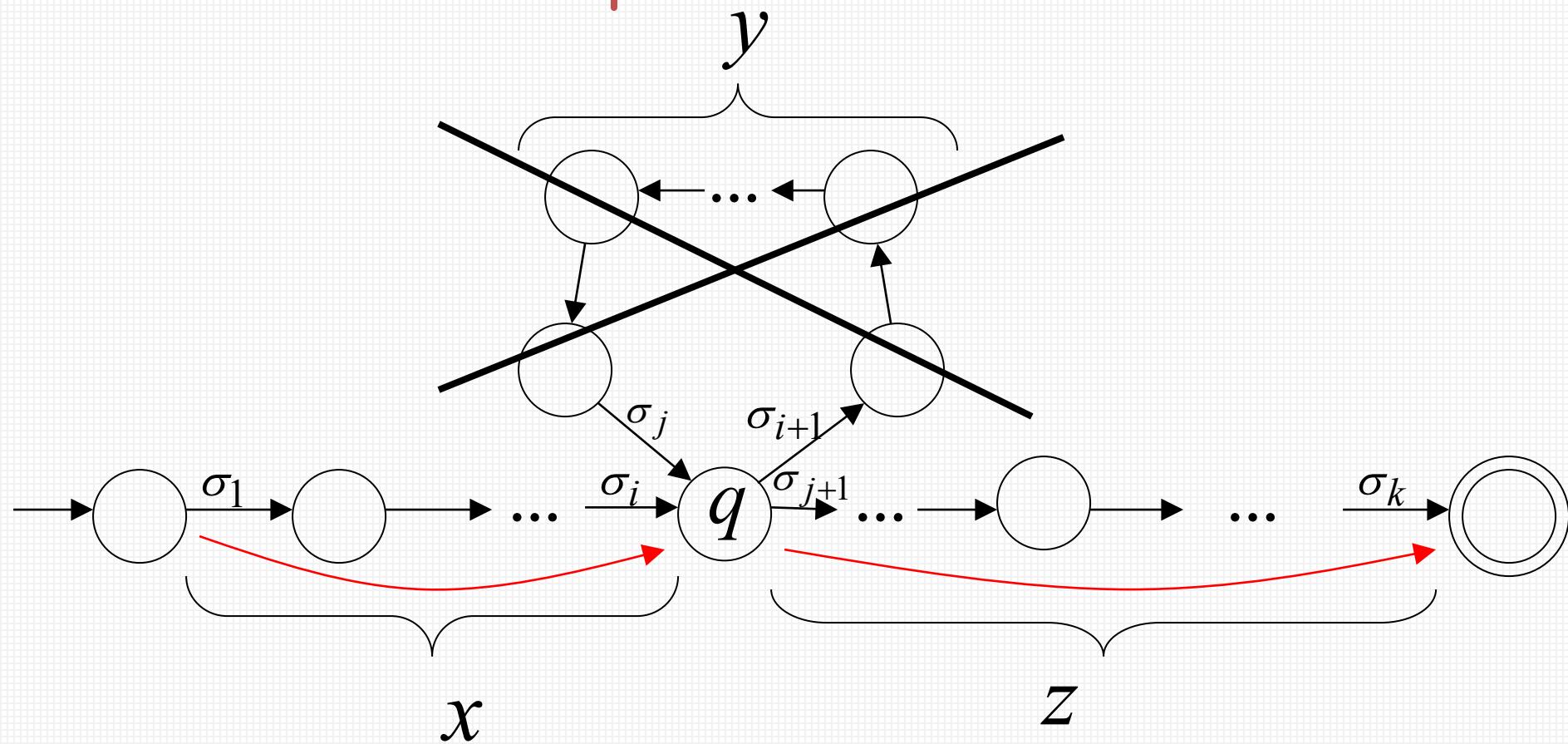
We do not care about the form of string z

z may actually overlap with the paths of x and y



Additional string: The string $x z$
is accepted

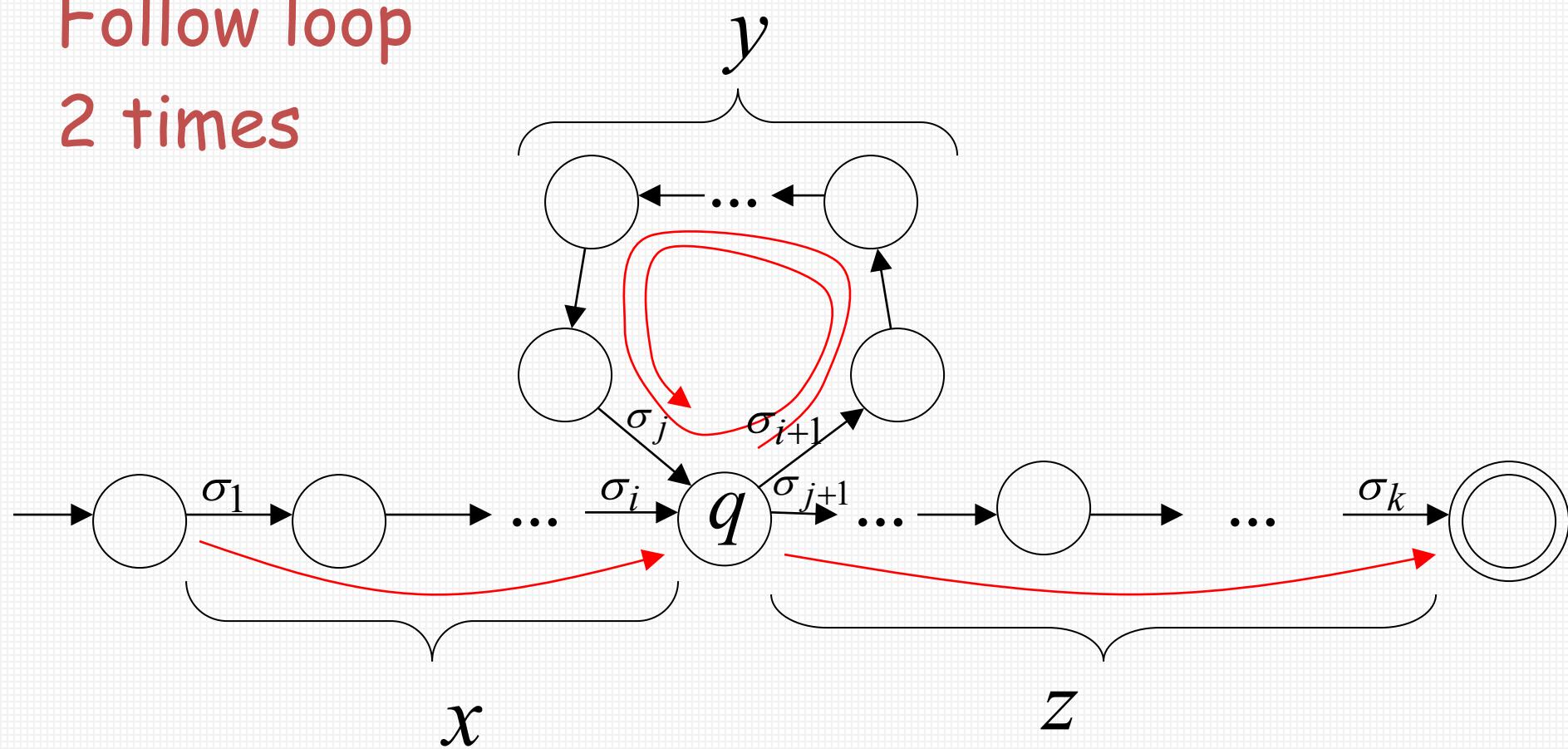
Do not follow loop



Additional string:

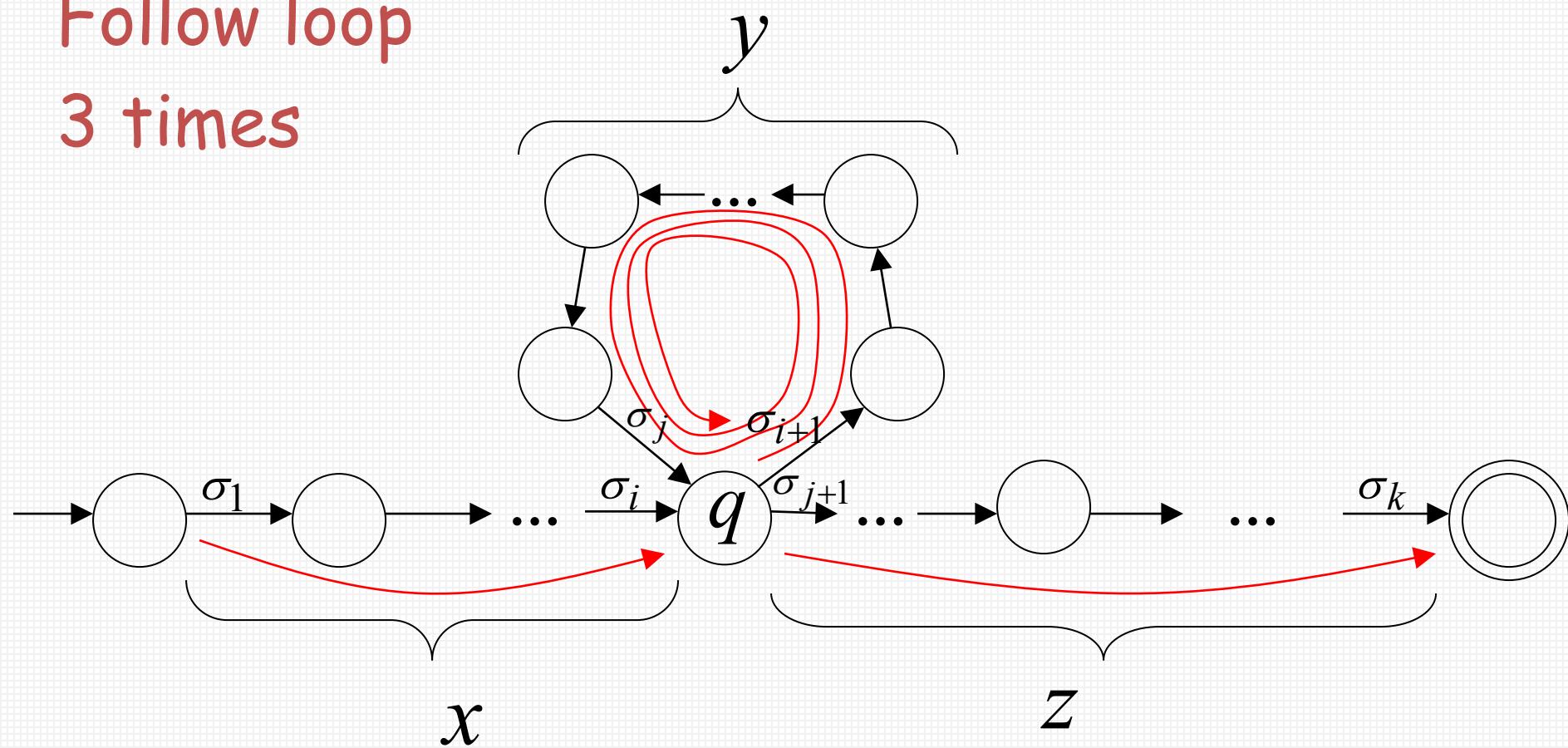
The string $x \ y \ y \ z$
is accepted

Follow loop
2 times



Additional string: The string $x y y y z$
is accepted

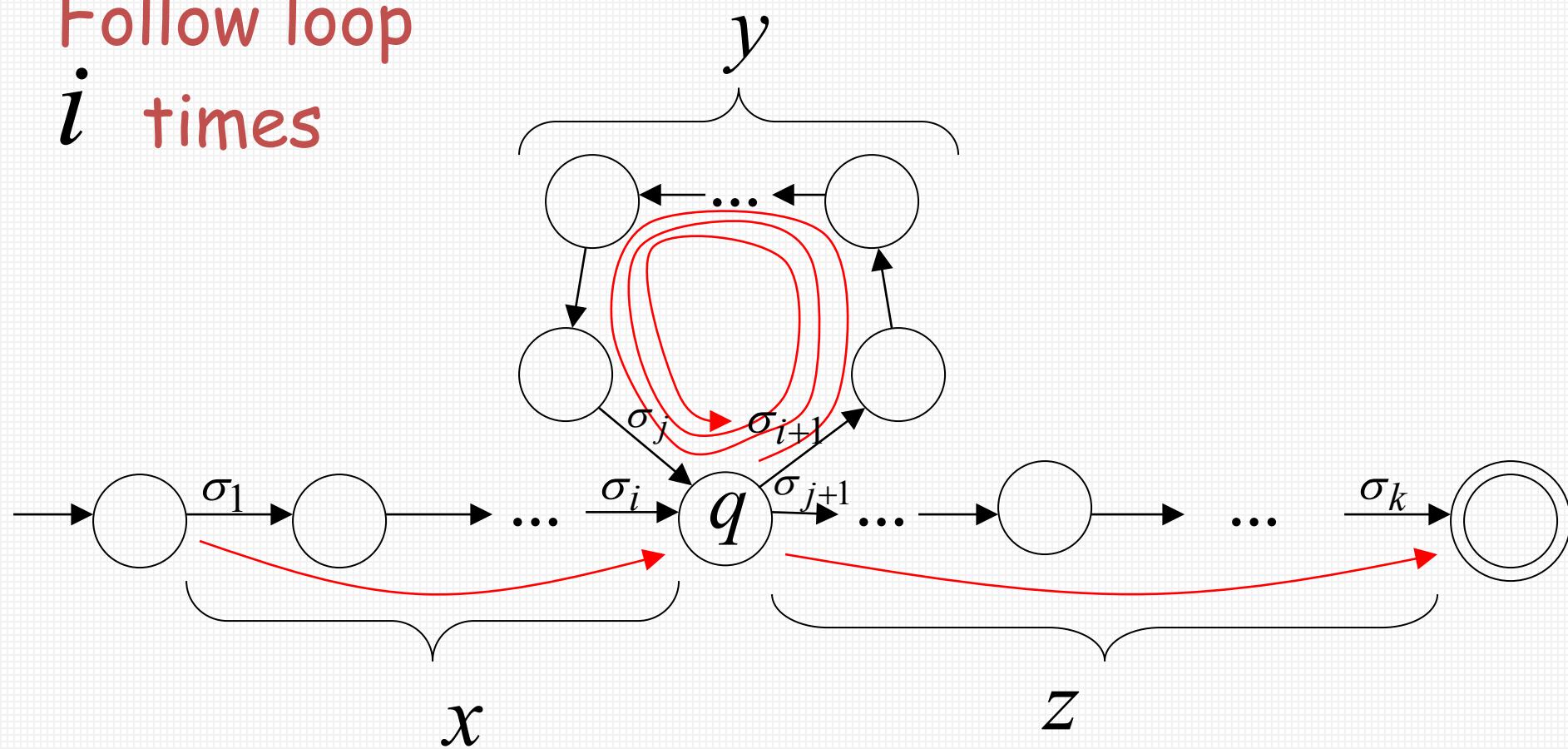
Follow loop
3 times



In General:

The string
is accepted $x \ y^i \ z$
 $i = 0, 1, 2, \dots$

Follow loop
 i times

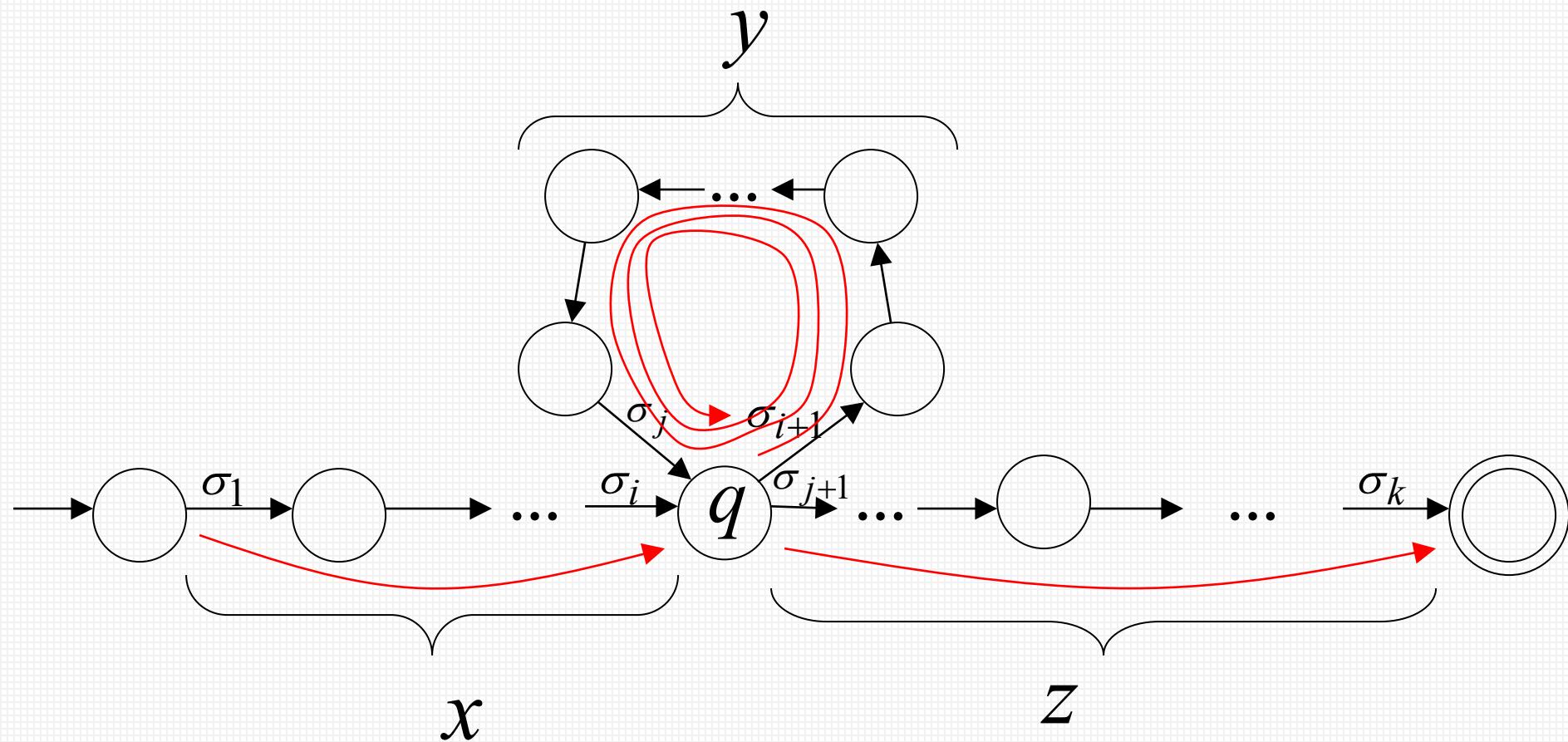


Therefore:

$$x \ y^i \ z \in L$$

$$i = 0, 1, 2, \dots$$

Language accepted by the DFA



The Pumping Lemma:

- Given a infinite regular language L there exists an integer m (critical length) for any string $w \in L$ with length $|w| \geq m$ we can write $w = x y z$ with $|x y| \leq m$ and $|y| \geq 1$
- such that: $x y^i z \in L \quad i = 0, 1, 2, \dots$

In our textbook:

Critical length m = Pumping length p

Observation:

Every language of finite size has to be regular

(we can easily construct an NFA
that accepts every string in the language)

Therefore, every non-regular language
has to be of infinite size

(contains an infinite number of strings)

Suppose you want to prove that
an infinite language L is not regular

1. Assume the opposite: L is regular
2. The pumping lemma should hold for L
3. Use the pumping lemma to obtain a contradiction
4. Therefore, L is not regular

Explanation of Step 3: How to get a contradiction

1. Let m be the critical length for L
2. Choose a particular string $w \in L$ which satisfies the length condition $|w| \geq m$
3. Write $w = xyz$
4. Show that $w' = xy^i z \notin L$ for some $i \neq 1$
5. This gives a contradiction, since from pumping lemma $w' = xy^i z \in L$

Example of Pumping Lemma application

Theorem: The language $L = \{a^n b^n : n \geq 0\}$
is not regular

Proof: Use the Pumping Lemma

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

Assume for contradiction
that L is a regular language

Since L is infinite
we can apply the Pumping Lemma

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

Let m be the critical length for L

Pick a string w such that: $w \in L$

and length $|w| \geq m$

We pick $w = a^m b^m$

From the Pumping Lemma:

we can write $w = a^m b^m = x y z$

with lengths $|x y| \leq m$, $|y| \geq 1$

$$w = xyz = a^m b^m = \overbrace{a \dots a}^m \underbrace{a \dots a}_{\text{red}} \underbrace{a \dots a}_{\text{red}} ab \dots b$$

m m

x y z

Thus: $y = a^k$, $1 \leq k \leq m$

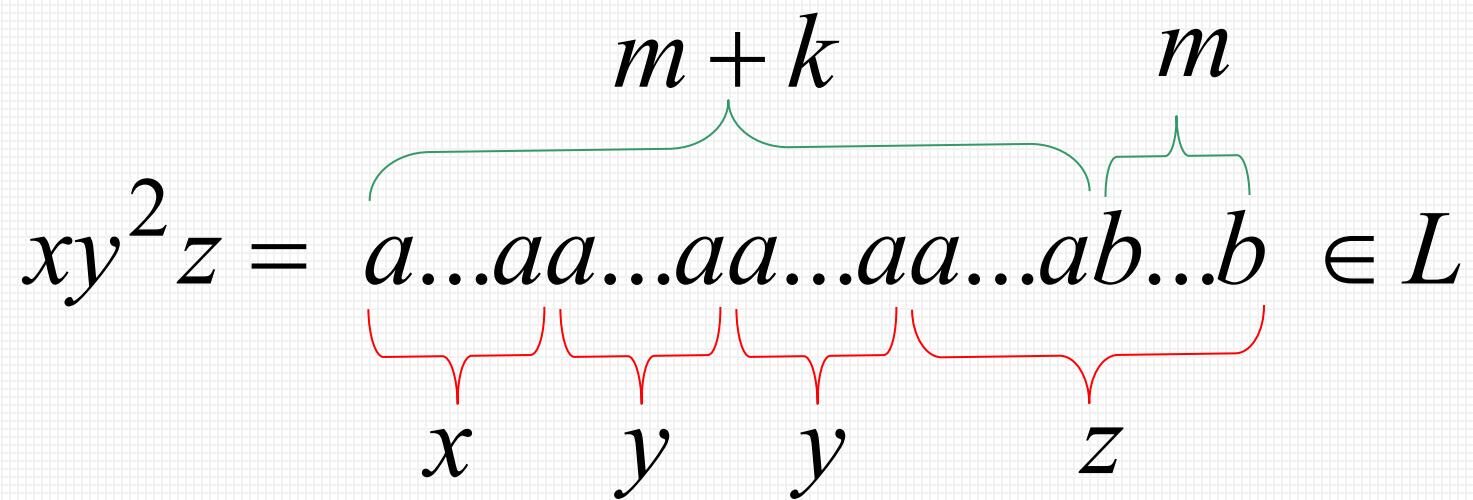
From the Pumping Lemma:

$$x \ y^i \ z \in L \quad i = 0, 1, 2, \dots$$

Thus:

$$x \ y^2 \ z \in L$$

$$x \ y \ z = a^m b^m \quad y = a^k, \quad 1 \leq k \leq m$$

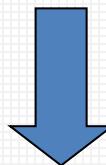


Thus: $a^{m+k}b^m \in L$

We get that

$$a^{m+k} b^m \in L \quad k \geq 1$$

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



$$a^{m+k} b^m \notin L$$

CONTRADICTION!!!

Therefore: Our assumption that L is a regular language is not true

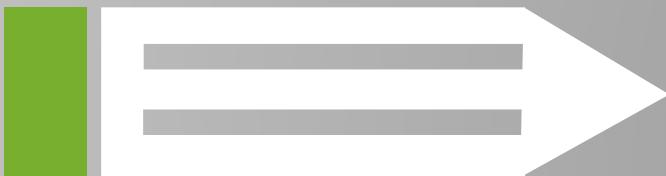
Conclusion: L is not a regular language

END OF PROOF

- ✓ Non-regular languages
- ✓ The Pigeonhole Principle
- ✓ Pumping lemma
- ✓ Applications of the Pumping Lemma

Theory of Computation

Context-Free Grammar



王轩
Wang
Xuan

- Context-Free Grammars
- Ambiguity Grammars
- Chomsky Normal Form
- Conversion to Chomsky Normal Form

Context-Free Languages

$$\{a^n b^n : n \geq 0\}$$
$$\{ww^R\}$$

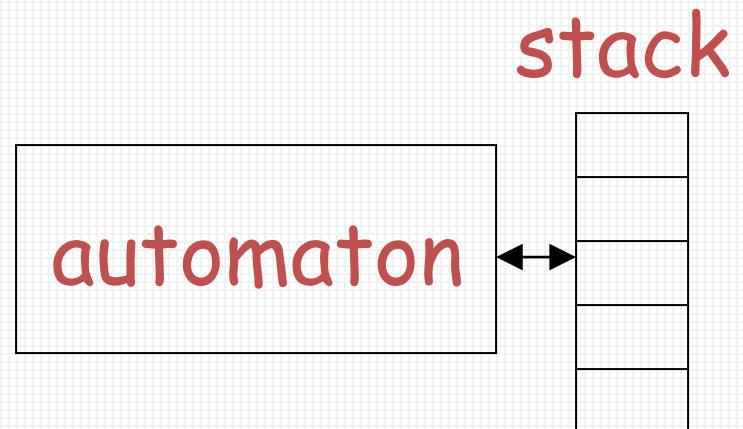
Regular Languages

$$a^* b^*$$
$$(a + b)^*$$

Context-Free Languages

Context-Free
Grammars

Pushdown
Automata



Grammars express languages

Example: the English language grammar

$$\langle \textit{sentence} \rangle \rightarrow \langle \textit{noun_phrase} \rangle \langle \textit{predicate} \rangle$$
$$\langle \textit{noun_phrase} \rangle \rightarrow \langle \textit{article} \rangle \langle \textit{noun} \rangle$$
$$\langle \textit{predicate} \rangle \rightarrow \langle \textit{verb} \rangle$$

Derivation of string “the dog sleeps”:

$\langle sentence \rangle \Rightarrow \langle noun_phrase \rangle \langle predicate \rangle$
 $\Rightarrow \langle noun_phrase \rangle \langle verb \rangle$
 $\Rightarrow \langle article \rangle \langle noun \rangle \langle verb \rangle$
 $\Rightarrow the \langle noun \rangle \langle verb \rangle$
 $\Rightarrow the \ dog \langle verb \rangle$
 $\Rightarrow the \ dog \ sleeps$

Productions

Sequence of
Terminals (symbols)

$$\langle \text{noun} \rangle \rightarrow \text{cat}$$
$$\langle \text{sentence} \rangle \rightarrow \langle \text{noun_phrase} \rangle \langle \text{predicate} \rangle$$

Variables

Sequence of Variables

Grammar: $S \rightarrow aSb$

$S \rightarrow \epsilon$

Language of the grammar:

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

A Convenient Notation

We write:

$$S \xrightarrow{*} aaabbb$$

for zero or more derivation steps

Instead of:

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$$

Grammar:

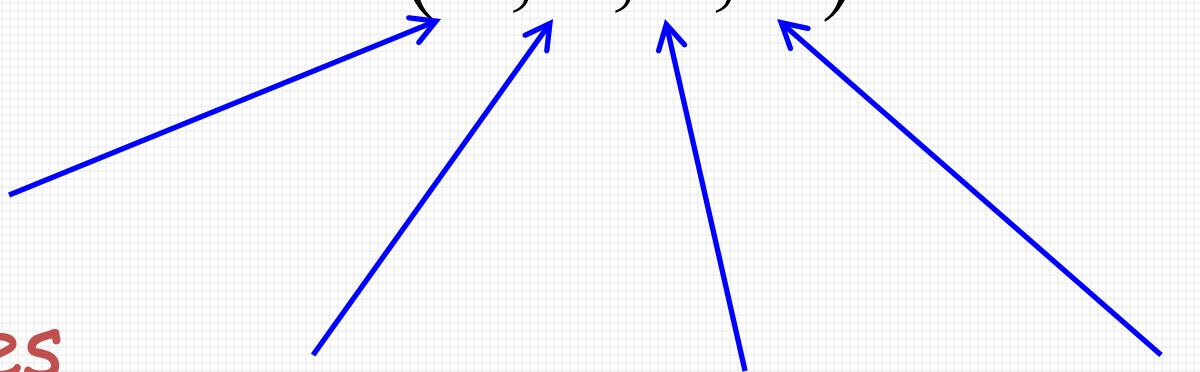
$$G = (V, T, S, P)$$

Set of
variables

Set of
terminal
symbols

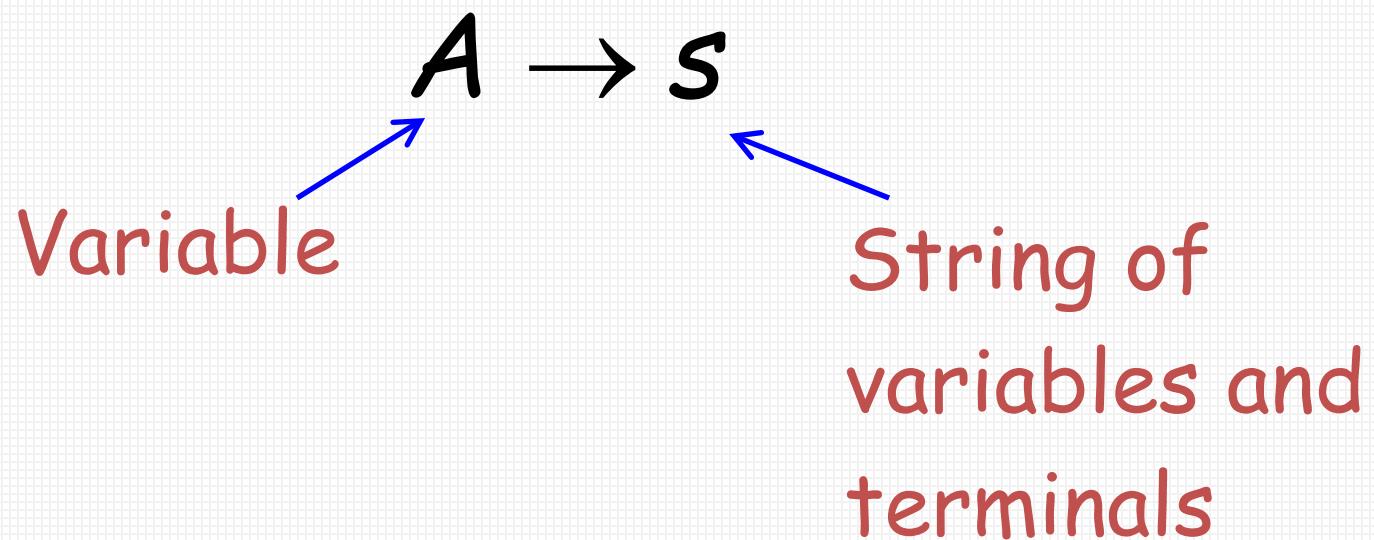
Start
variable

Set of
productions



Context-Free Grammar: $G = (V, T, S, P)$

All productions in P are of the form



Example

$$S \rightarrow aSb \mid \epsilon$$

productions

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

$$G = (V, T, S, P)$$

$V = \{S\}$
variables

$T = \{a, b\}$
terminals

start variable

Context-Free Language definition:

A language L is context-free
if there is a context-free grammar G
with $L = L(G)$

Example:

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

is a context-free language
since context-free grammar G :

$$S \rightarrow aSb \mid \epsilon$$

generates $L(G) = L$

Another Example:

Context-free grammar G :

$$S \rightarrow aSa \mid bSb \mid \epsilon$$

Example derivations:

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abba$$

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abaSaba \Rightarrow abaaba$$

Ambiguous Grammar:

A context-free grammar G is ambiguous if there is a string $w \in L(G)$ which has:
two different derivation trees or two leftmost derivations.

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

Two different derivation trees give two different leftmost derivations and vice-versa)

Grammar for mathematical expressions

$$E \rightarrow E + E \quad | \quad E * E \quad | \quad (E) \quad | \quad a$$

Example strings:

$$(a + a) * a + (a + a * (a + a))$$

Denotes any number

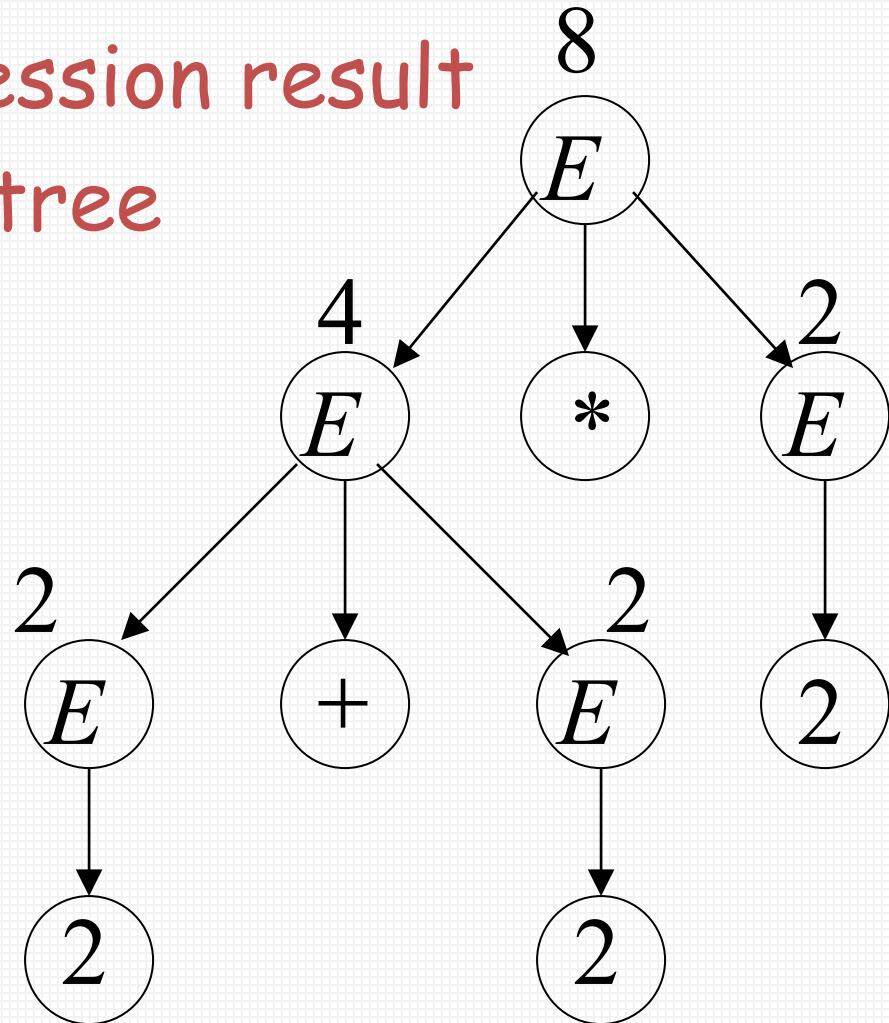
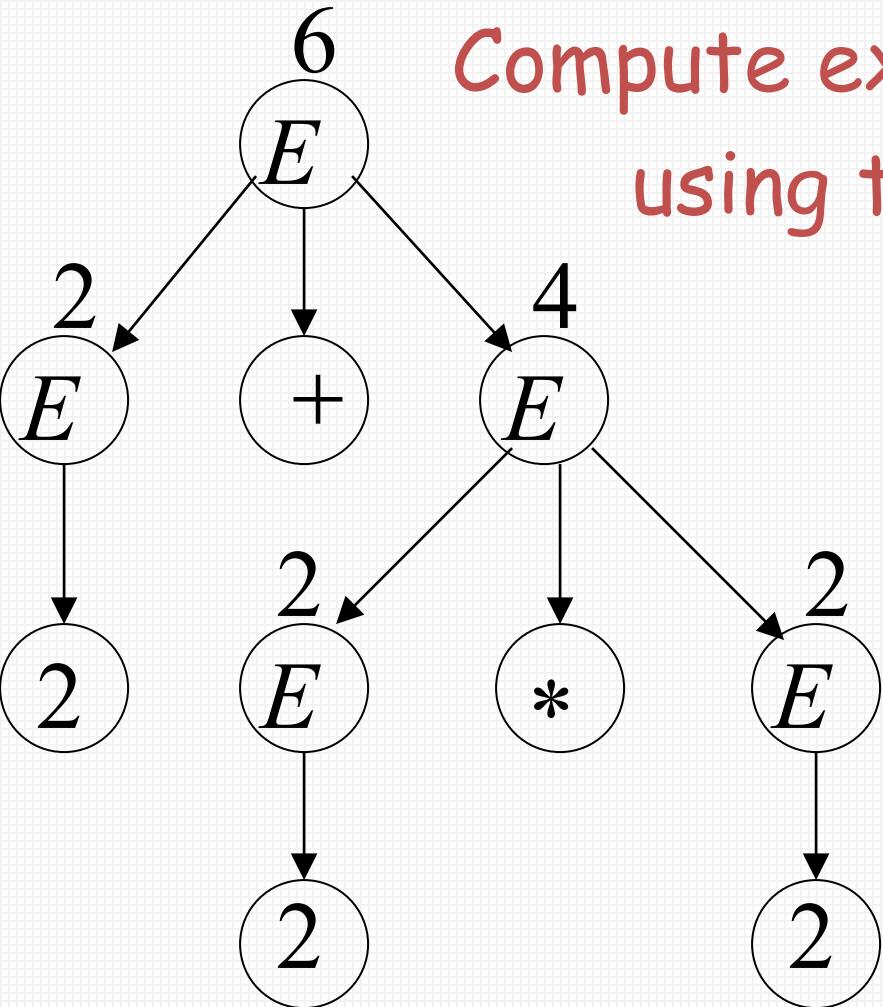
Good Tree

$$2 + 2 * 2 = 6$$

Bad Tree

$$2 + 2 * 2 = 8$$

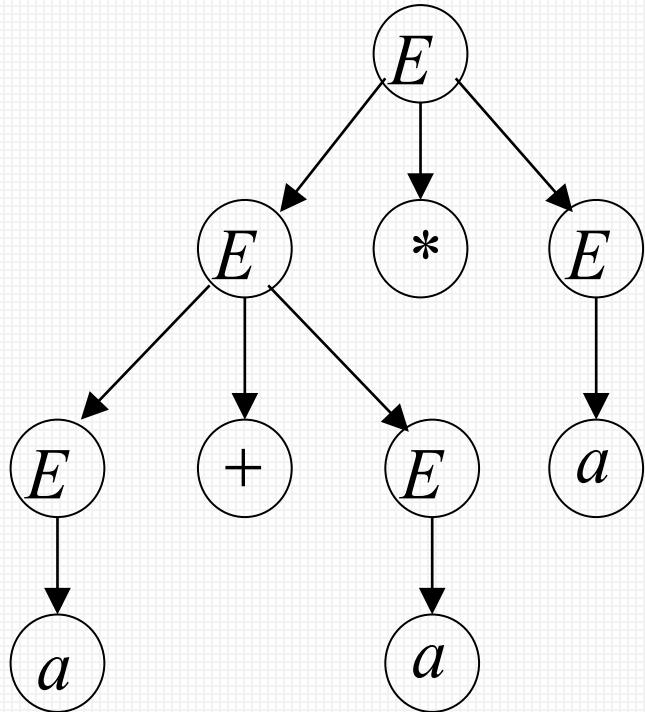
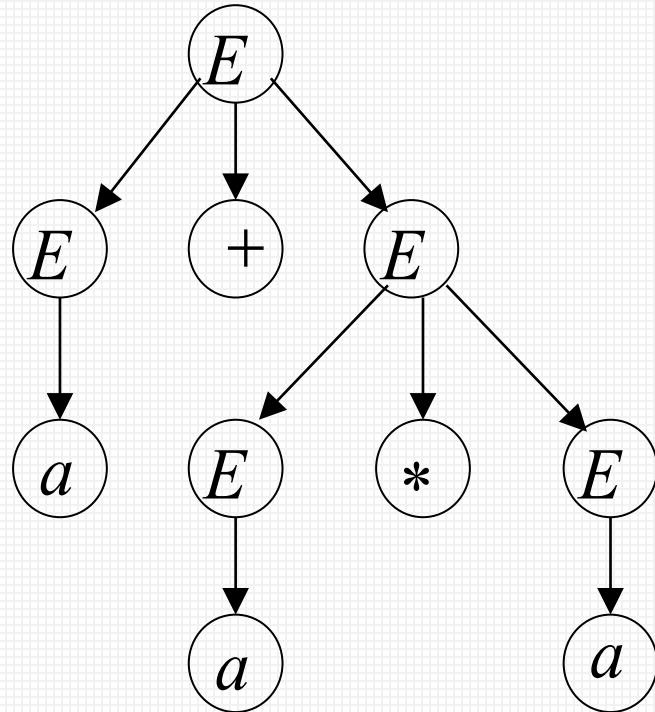
Compute expression result
using the tree



Example:

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

this grammar is ambiguous since
string $a + a * a$ has two derivation trees



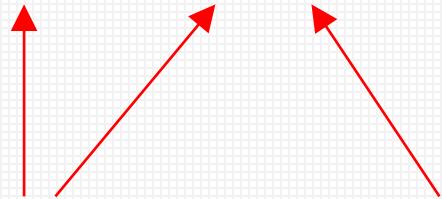
Chomsky Normal Form

Each production has form:



Chomsky

$$A \rightarrow BC$$



variable

or

$$A \rightarrow a$$



variable

terminal

Where a is any terminal and A, B, C are any variables—except that B and C may not be the start variable. In addition we permit the rule $S \rightarrow \varepsilon$, where S is the start variable

Examples:

$$S \rightarrow AS$$

$$S \rightarrow a$$

$$A \rightarrow SA$$

$$A \rightarrow b$$

Chomsky
Normal Form

$$S \rightarrow AS$$

$$S \rightarrow AAS$$

$$A \rightarrow SA$$

$$A \rightarrow aa$$

Not Chomsky
Normal Form

Conversion to Chomsky Normal Form

$$S \rightarrow ABa$$

Example: $A \rightarrow aab$

$$B \rightarrow Ac$$

Not Chomsky
Normal Form

We will convert it to Chomsky Normal Form

Step 1: Add new start state S_0 and rule $S_0 \rightarrow S$

Step 2: Remove ϵ rules, e.g. $A \rightarrow \epsilon$

Step 3: Remove Unit-Productions

Step 4: Convert into proper form

This sequence guarantees that
unwanted variables and productions
are removed

ϵ rules

ϵ – rule :

$$X \rightarrow \epsilon$$

Example:

$$S \rightarrow aMb$$

$$M \rightarrow aMb$$

$$M \rightarrow \epsilon$$



ϵ – rule

Removing ε – rule

$$S \rightarrow aMb$$

$$M \rightarrow aMb$$

$$\cancel{M \rightarrow \varepsilon}$$

Substitute

$$M \rightarrow \varepsilon$$

$$S \rightarrow aMb \mid ab$$

$$M \rightarrow aMb \mid ab$$

After we remove all the ε – rule

all the nullable productions are disappeared
(except for the start variable)

Unit-Productions

Unit Production: $X \rightarrow Y$

(a single variable in both sides)

Example:

$$S \rightarrow aA$$

$$A \rightarrow a$$

$$A \rightarrow B$$

$$B \rightarrow A$$

$$B \rightarrow bb$$

Unit Productions

Removal of unit productions:

$A -> B$

$B -> u$

Substitute

$A \rightarrow B$

$A -> u$

Unit productions of form $X \rightarrow X$

can be removed immediately

$$S \rightarrow aA \mid aB$$

$$A \rightarrow a$$

$$B \rightarrow A \mid \cancel{B}$$

$$B \rightarrow bb$$

Remove

$$B \rightarrow B$$

$$S \rightarrow aA \mid aB$$

$$A \rightarrow a$$

$$B \rightarrow A$$

$$B \rightarrow bb$$

Introduce new variables for the terminals:

$$T_a, T_b, T_c$$

$$S \rightarrow ABa$$

$$A \rightarrow aab$$

$$B \rightarrow Ac$$



$$S \rightarrow ABT_a$$

$$A \rightarrow T_a T_a T_b$$

$$B \rightarrow AT_c$$

$$T_a \rightarrow a$$

$$T_b \rightarrow b$$

$$T_c \rightarrow c$$

Replace any production $A \rightarrow C_1C_2 \cdots C_n$

with $A \rightarrow C_1V_1$

$V_1 \rightarrow C_2V_2$

...

$V_{n-2} \rightarrow C_{n-1}C_n$

New intermediate variables: V_1, V_2, \dots, V_{n-2}

Introduce new intermediate variable V_1
to break first production:

$$S \rightarrow ABT_a$$

$$A \rightarrow T_a T_a T_b$$

$$B \rightarrow AT_c$$

$$T_a \rightarrow a$$

$$T_b \rightarrow b$$

$$T_c \rightarrow c$$



$$S \rightarrow AV_1$$

$$V_1 \rightarrow BT_a$$

$$A \rightarrow T_a T_a T_b$$

$$B \rightarrow AT_c$$

$$T_a \rightarrow a$$

$$T_b \rightarrow b$$

$$T_c \rightarrow c$$

Introduce intermediate variable:

$$S \rightarrow AV_1$$

$$V_1 \rightarrow BT_a$$

$$A \rightarrow T_a T_a T_b$$

$$B \rightarrow AT_c$$

$$T_a \rightarrow a$$

$$T_b \rightarrow b$$

$$T_c \rightarrow c$$

$$V_2$$

$$S \rightarrow AV_1$$

$$V_1 \rightarrow BT_a$$

$$A \rightarrow T_a V_2$$

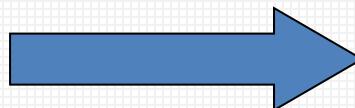
$$V_2 \rightarrow T_a T_b$$

$$B \rightarrow AT_c$$

$$T_a \rightarrow a$$

$$T_b \rightarrow b$$

$$T_c \rightarrow c$$



Final grammar in Chomsky Normal Form:

$$S \rightarrow AV_1$$

$$V_1 \rightarrow BT_a$$

$$A \rightarrow T_a V_2$$

$$V_2 \rightarrow T_a T_b$$

$$B \rightarrow AT_c$$

$$T_a \rightarrow a$$

$$T_b \rightarrow b$$

$$T_c \rightarrow c$$

Initial grammar

$$S \rightarrow ABa$$

$$A \rightarrow aab$$

$$B \rightarrow Ac$$

Observations

- Chomsky normal forms are good for parsing and proving theorems
- It is easy to find the Chomsky normal form for any context-free grammar

- Context-free grammar

$$S \rightarrow ASA \mid aB$$
$$A \rightarrow B \mid S$$
$$B \rightarrow b \mid \varepsilon$$

- 1. Add the start state and the rule $S_0 \rightarrow S$

$$S_0 \rightarrow S$$
$$S \rightarrow ASA \mid aB$$
$$A \rightarrow B \mid S$$
$$B \rightarrow b \mid \varepsilon$$

- 2. Remove ε rules, $B \rightarrow \varepsilon$

$$S_0 \rightarrow S$$

$$S \rightarrow ASA | aB$$

$$A \rightarrow B | S$$

$$B \rightarrow b | \varepsilon$$

$$S_0 \rightarrow S$$

$$S \rightarrow ASA | aB | a$$

$$A \rightarrow B | S | \varepsilon$$

$$B \rightarrow b$$

- Remove ε rules, $A \rightarrow \varepsilon$

$$S_0 \rightarrow S$$

$$S \rightarrow ASA | aB | a | SA | AS | S$$

$$A \rightarrow B | S$$

$$B \rightarrow b$$

- 3. remove $S \rightarrow S$, remove $S_0 \rightarrow S$

$S_0 \rightarrow S$

$S \rightarrow ASA|aB|a|SA|AS$

$A \rightarrow B|S$

$B \rightarrow b$

$S_0 \rightarrow ASA|aB|a|SA|AS$

$S \rightarrow ASA|aB|a|SA|AS$

$A \rightarrow B|S$

$B \rightarrow b$

- Remove $A \rightarrow B$, remove $A \rightarrow S$

$S_0 \rightarrow ASA|aB|a|SA|AS$

$S \rightarrow ASA|aB|a|SA|AS$

$A \rightarrow S|b$

$B \rightarrow b$

$S_0 \rightarrow ASA|aB|a|SA|AS$

$S \rightarrow ASA|aB|a|SA|AS$

$A \rightarrow b|ASA|aB|a|SA|AS$

$B \rightarrow b$

- Finally, add additional variables and rules

$$S_0 \rightarrow ASA|aB|a|SA|AS$$

$$S \rightarrow ASA|aB|a|SA|AS$$

$$A \rightarrow b|ASA|aB|a|SA|AS$$

$$B \rightarrow b$$

$$S_0 \rightarrow AA_1|UB|a|SA|AS$$

$$S \rightarrow AA_1|UB|a|SA|AS$$

$$A \rightarrow b|AA_1|UB|a|SA|AS$$

$$A_1 \rightarrow SA$$

$$U \rightarrow a$$

$$B \rightarrow b$$

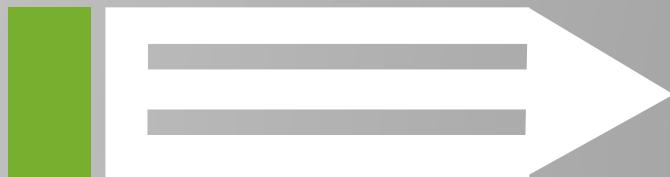
- ✓ Context-Free Grammars
- ✓ Ambiguity Grammars
- ✓ Chomsky Normal Form
- ✓ Conversion to Chomsky Normal Form

Theory of Computation

Lesson 5-1

Pushdown Automata

PDAs



王轩
Wang
Xuan

Outline

- Formalities for PDAs
- Pushing & Popping Strings
- Non-Determinism
- Example PDA
- Accept and Reject

History of PDA

Since 1968, millions of computer science students have taken a course on **algorithms** and **data structures**, typically the second course after the initial one introducing programming. One of the basic data structures in such a course is the **stack**.

The **stack has a special place** in the emergence of computing as a science, as argued by Michael Mahoney, the pioneer of the history of the theory of computing: “Between 1955 and 1970, a new agenda formed around the theory of automata and formal languages, which increasingly came to be viewed as foundational for the field as a whole”. In this process, interest arose in “devices with **more generative power than finite automata**, and **more special structure than Turing machines**.”

The **push-down automaton** which is based on a **stack** is such a device.

History of PDA

The importance of the stack in the development of computer science has been explained by Chomsky and Miller in terms of the push-down storage automaton (PDA).

This is an automaton with one read-only input tape moving in one direction only, and one storage tape, movable in both directions. Symbols are written and read on the storage tape, which is used as a push-down storage/ stack with only one symbol visible at a time.

It has been said that "...the theory of push-down automata is, in fact, essentially another version of the theory of context free grammar". The PDA automaton was introduced by Newell et al in 1959.6

History of PDA

More specifically,

The notion of the 'push-down store' was introduced by
Newell, Shaw, and Simon(1959).

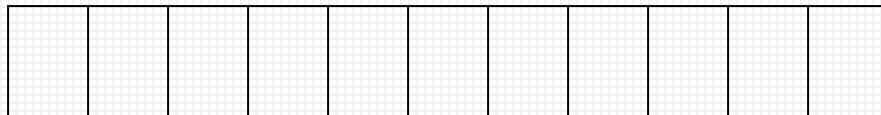
The first formulation of the relationship between push-down automata and formal languages is that of **Oettinger(1961).**

The relationship between context-free grammars and push-down automata was formulated by **Chomsky(1963)** and **Evey(1963)** more or less independently.

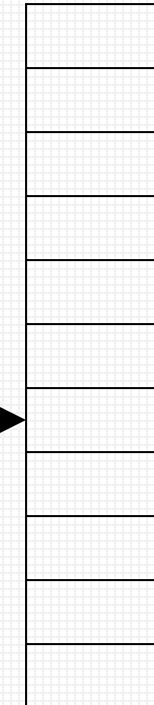
The equivalence of deterministic push-down automata and LR-grammars was proven by **Knuth(1965).**

Pushdown Automaton -- PDA

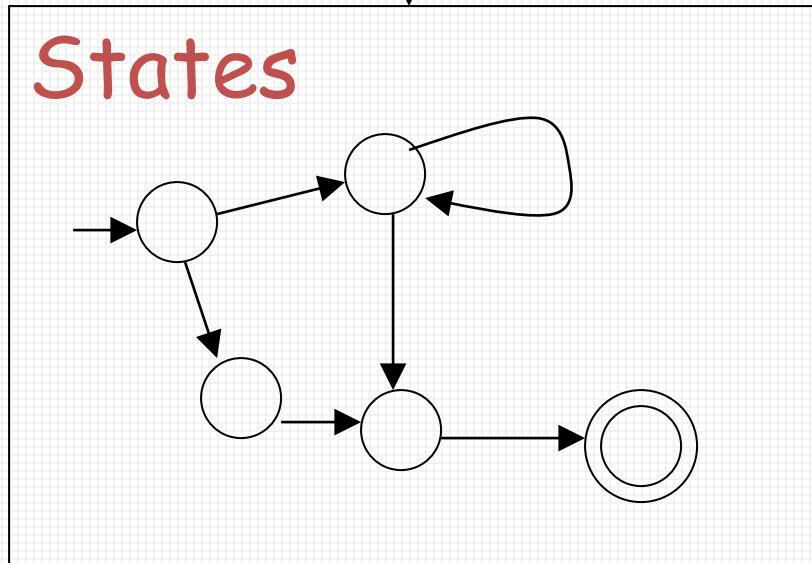
Input String



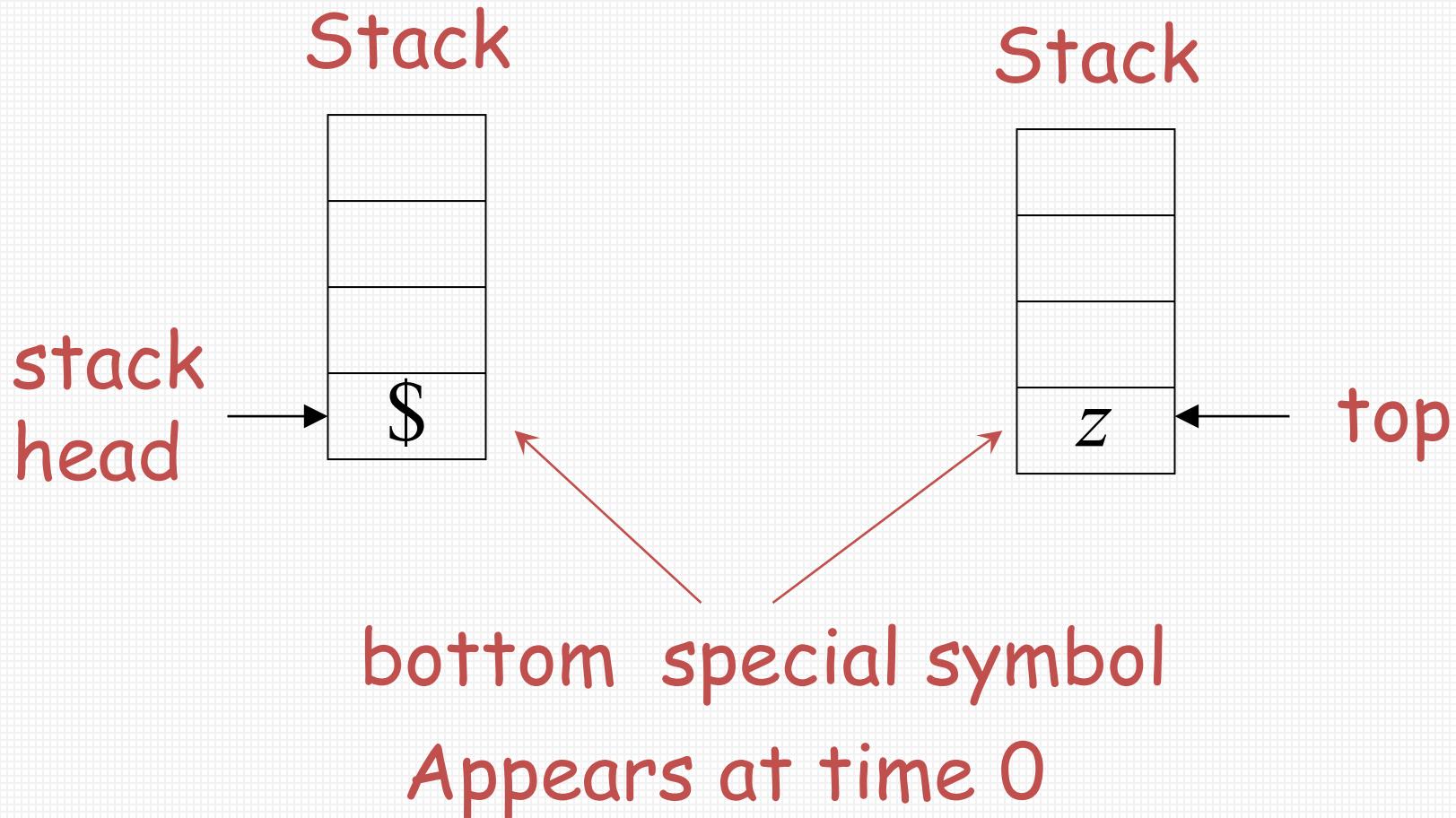
Stack



States



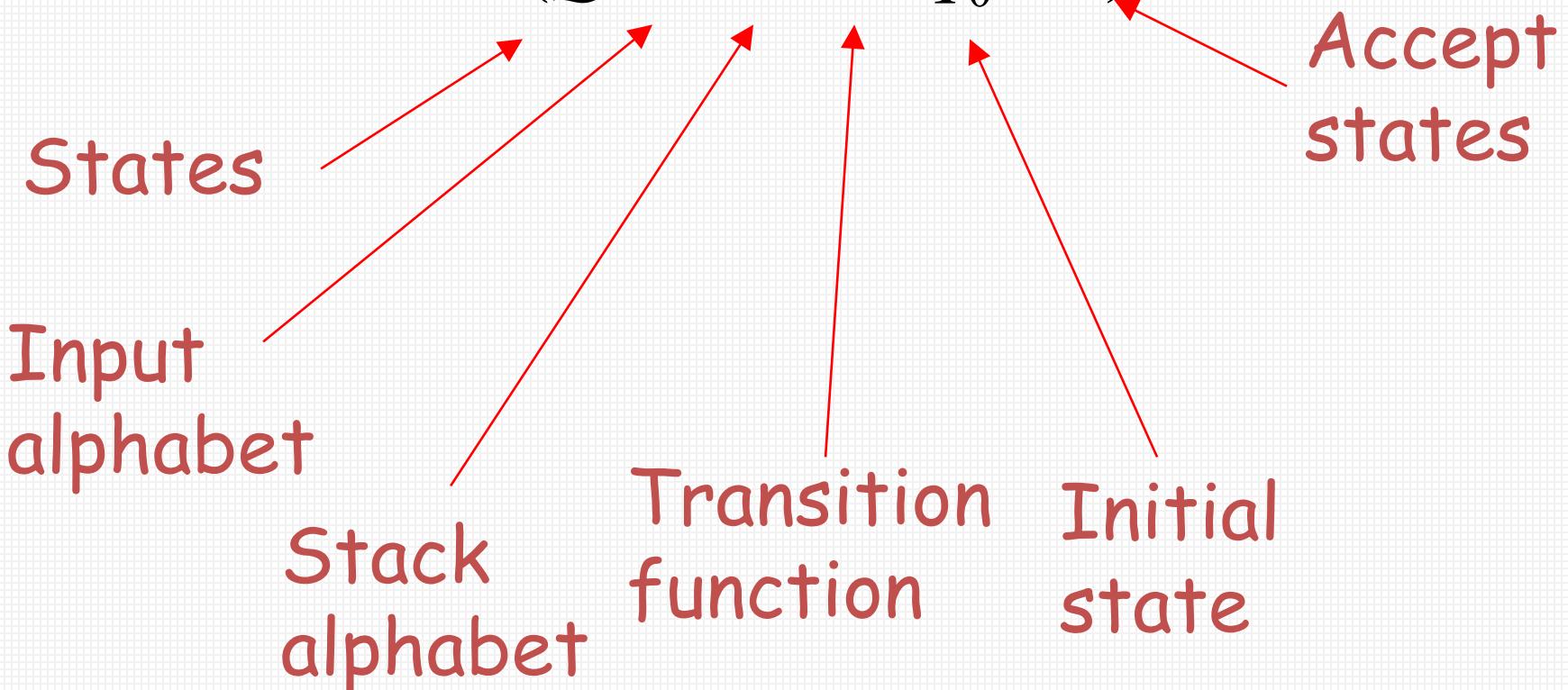
Initial Stack Symbol



Formal Definition

Pushdown Automaton (PDA)

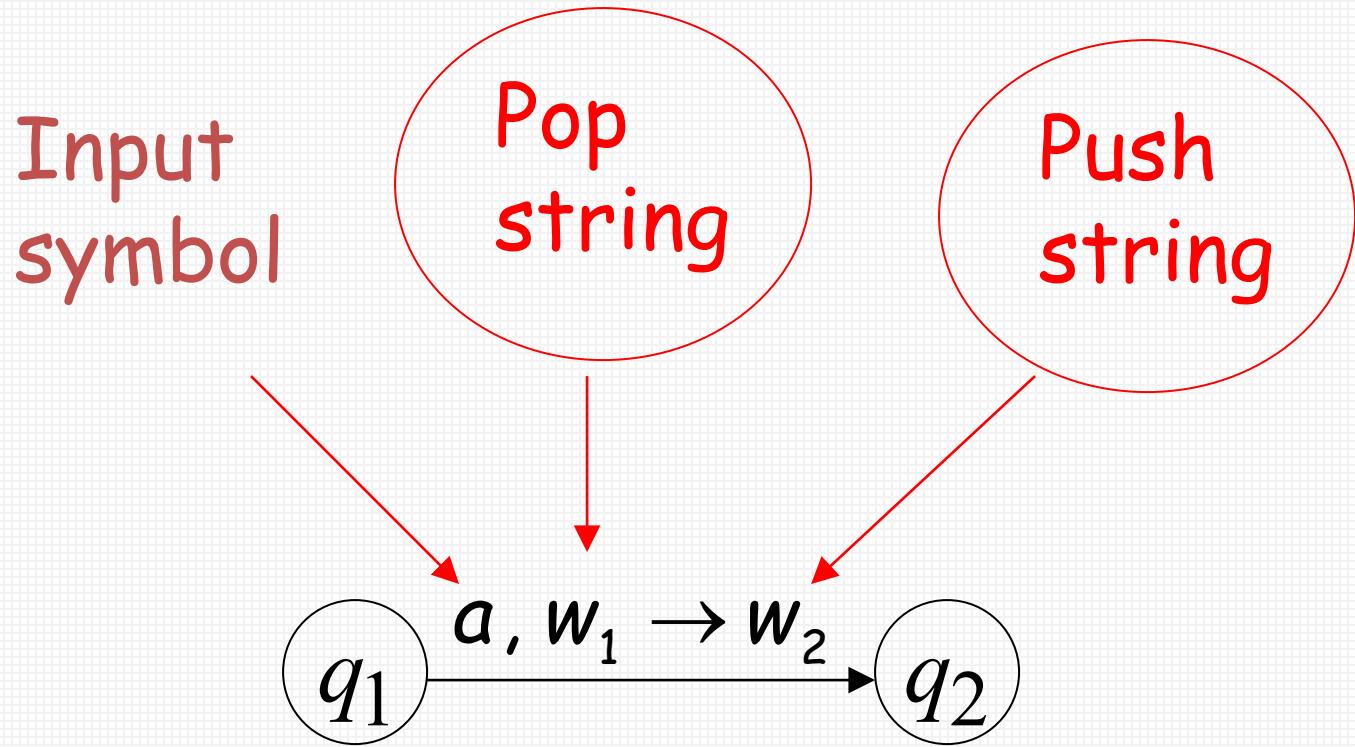
$$M = (Q, \Sigma, \Gamma, \delta, q_0, F)$$

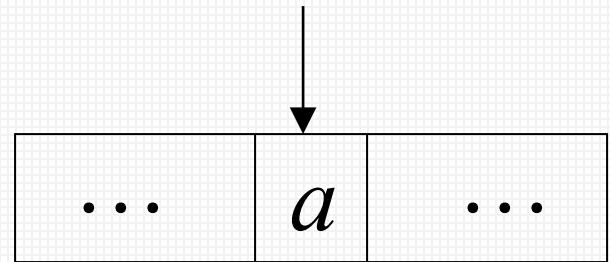
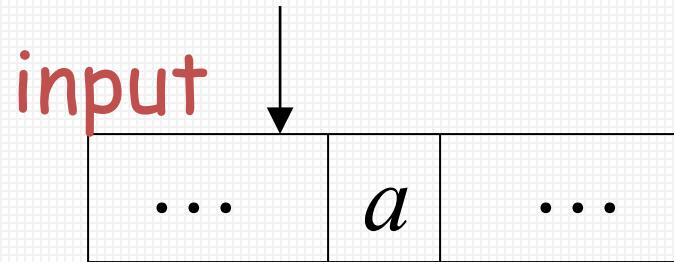
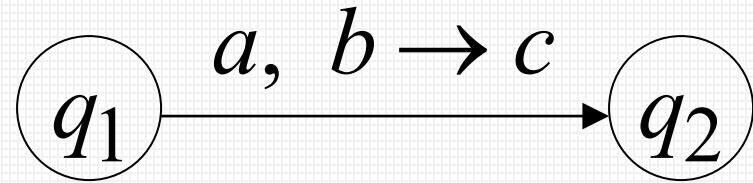


A pushdown automaton is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$

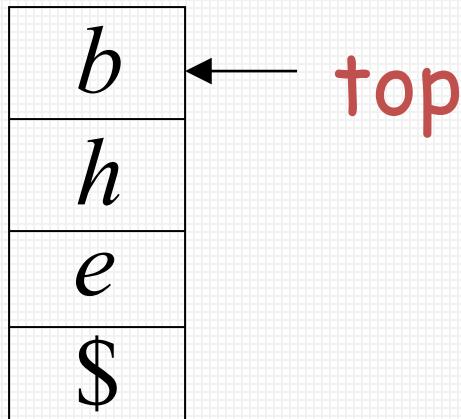
1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta: Q \times \Sigma \times \Gamma \rightarrow P(Q \times \Gamma)$ is the transition function,
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

Pushing & Popping Strings

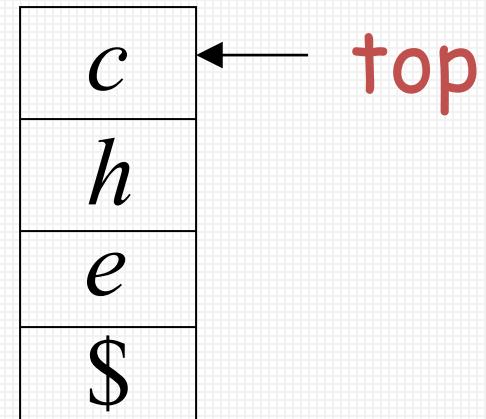


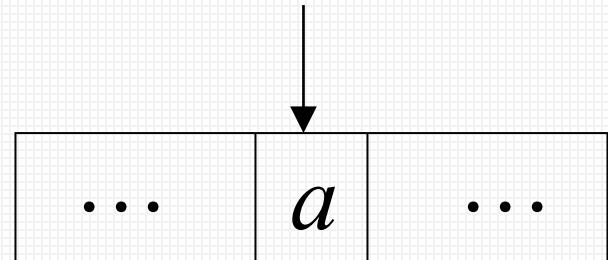
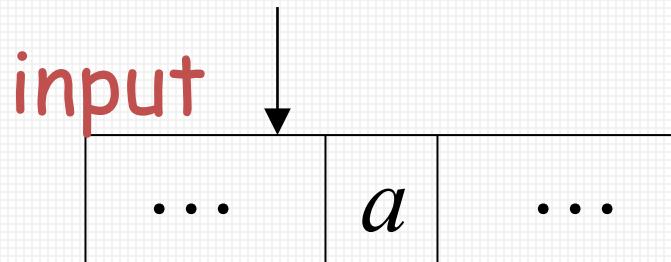
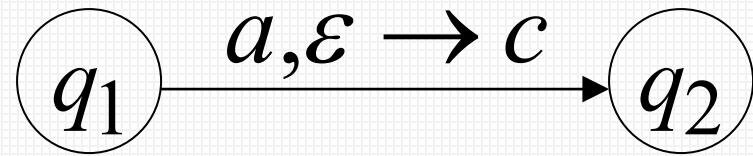


stack



Replace

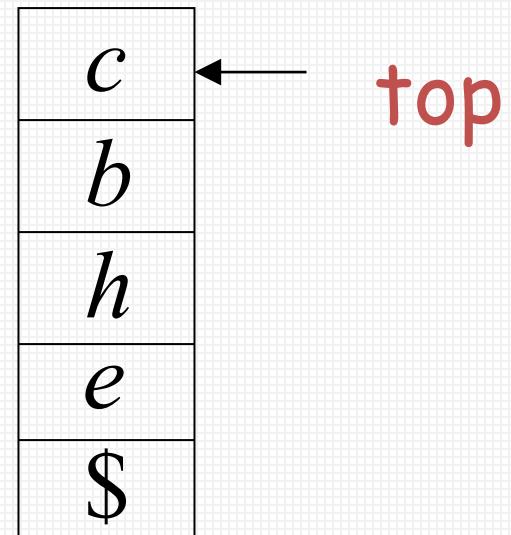
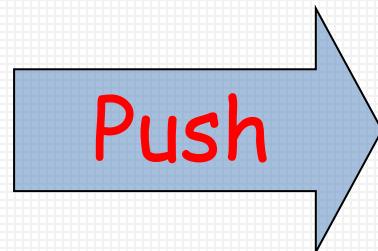


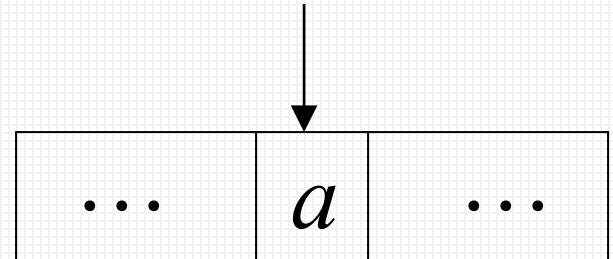
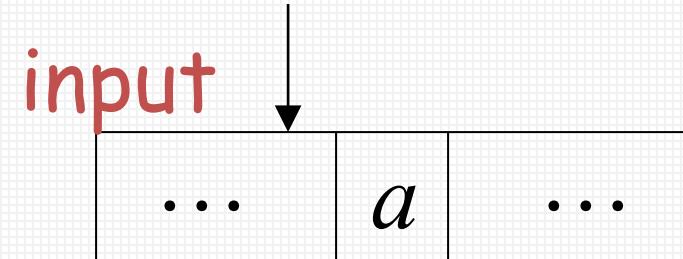
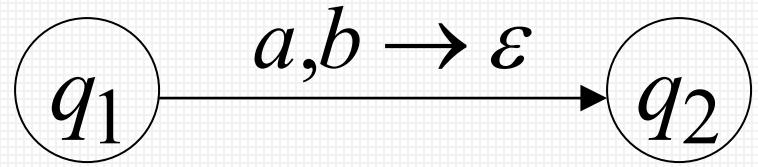


stack

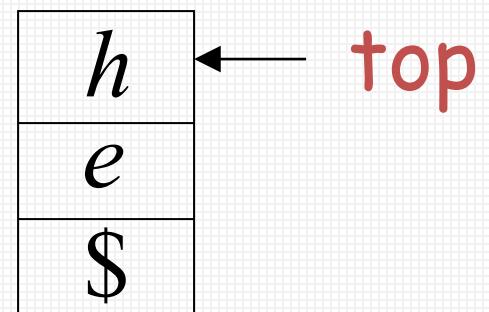
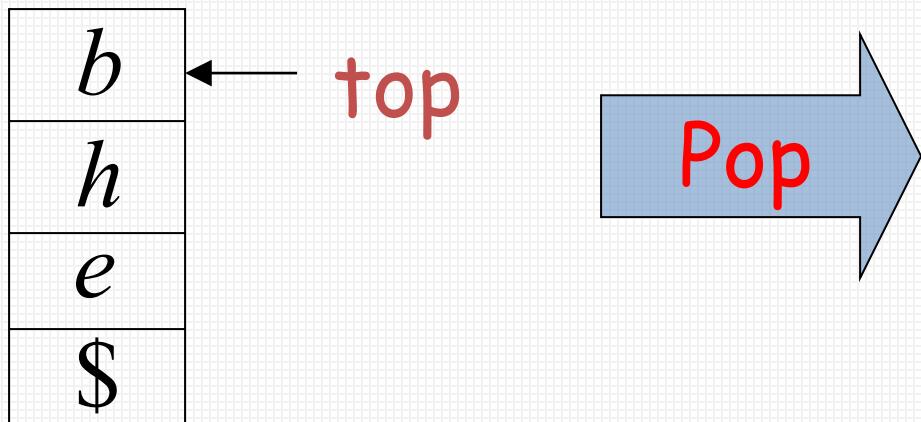
<i>b</i>
<i>h</i>
<i>e</i>
<i>\$</i>

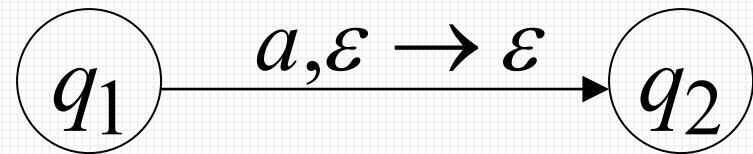
top





stack

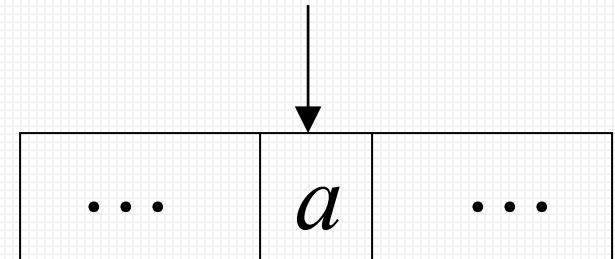




input

A horizontal input tape divided into three rectangular boxes. The first and third boxes contain three dots (...). The middle box contains the letter 'a'. A vertical arrow points down to the 'a' box, labeled 'input' in red.

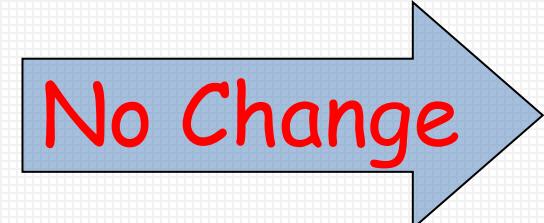
...	a	...
-----	---	-----



stack

A vertical stack structure with four horizontal boxes. From top to bottom, the boxes contain the letters 'b', 'h', 'e', and '\$'. A horizontal arrow points left from the top of the 'b' box, labeled 'top' in red.

b
h
e
\$



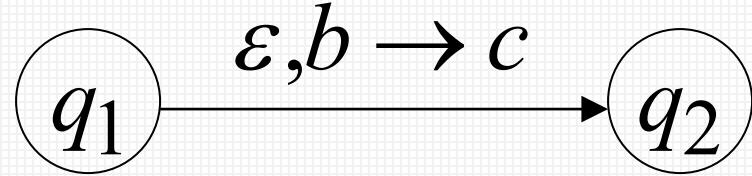
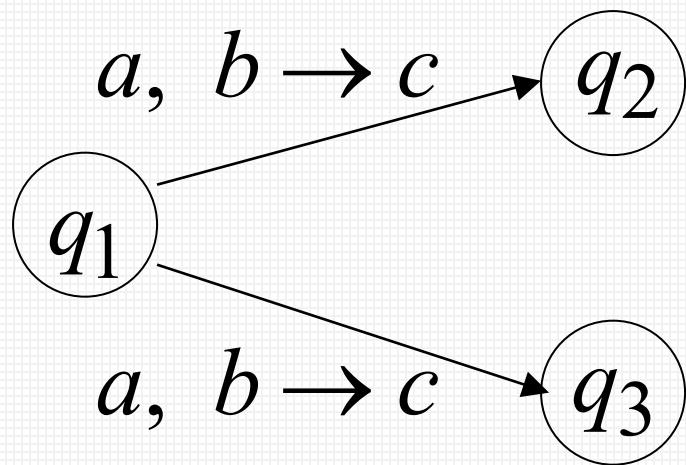
A second stack structure identical to the first, containing 'b', 'h', 'e', and '\$' from top to bottom. A horizontal arrow points left from the top of the 'b' box, labeled 'top' in red.

b
h
e
\$

Non-Determinism

PDAs are non-deterministic

Allowed non-deterministic transitions

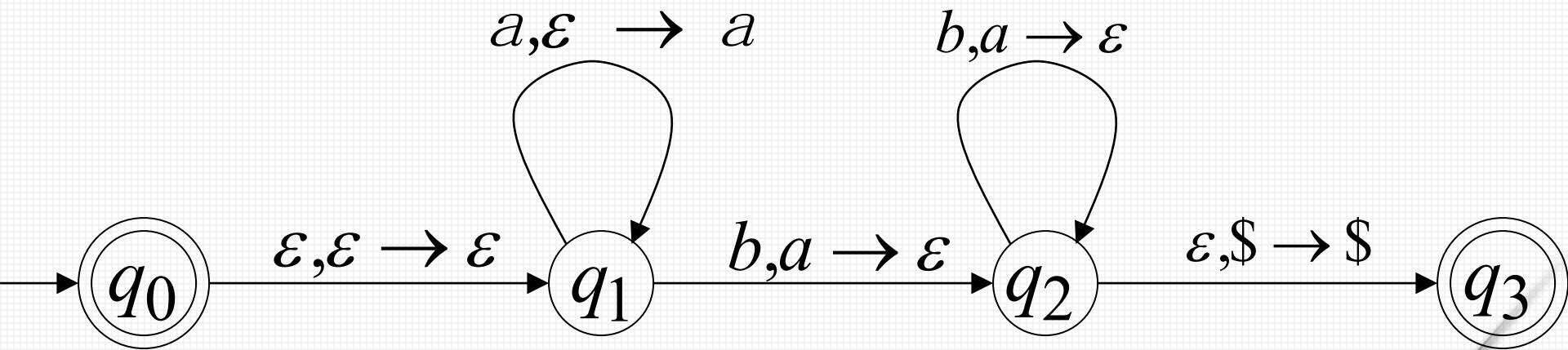


ε – transition

Example PDA

PDA M :

$$L(M) = \{a^n b^n : n \geq 0\}$$



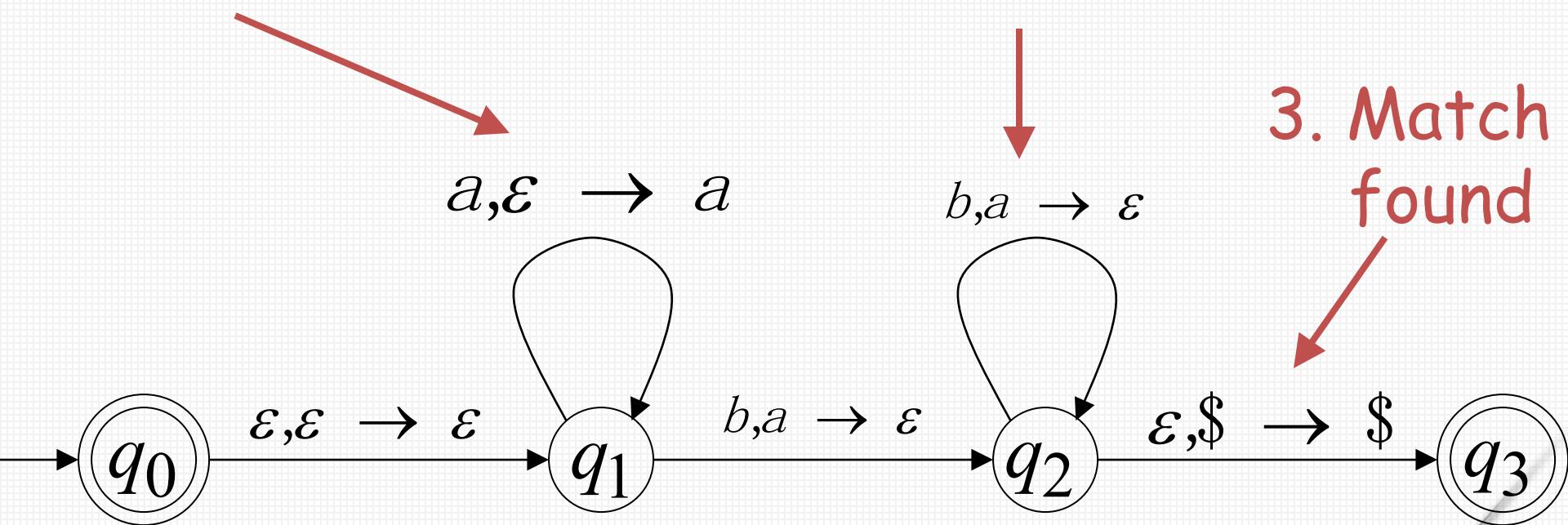
$$L(M) = \{a^n b^n : n \geq 0\}$$

Basic Idea:

1. Push the a's
on the stack

2. Match the b's on input
with a's on stack

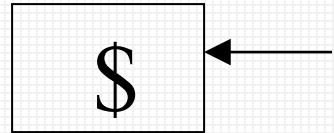
3. Match
found



Execution Example: Time 0

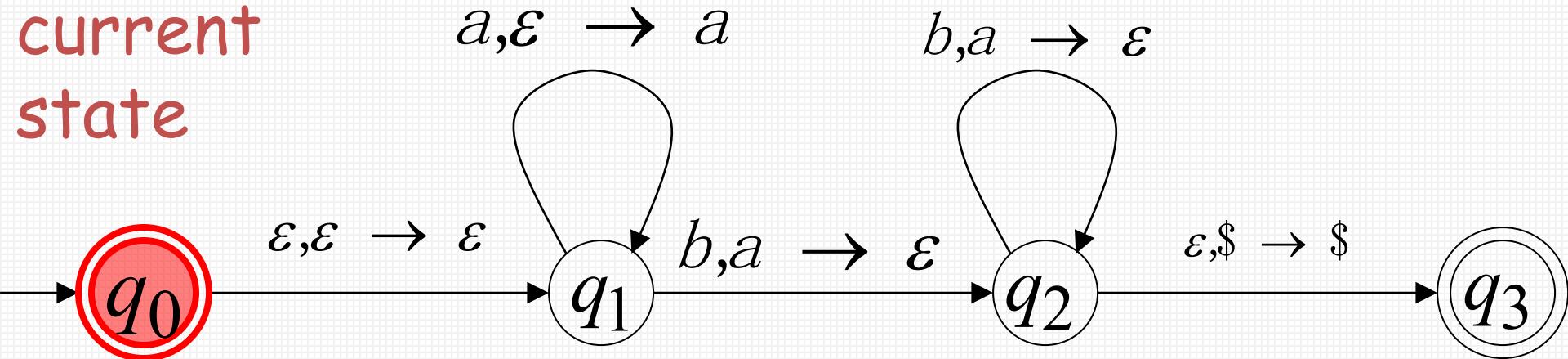
Input

a	a	a	b	b	b
-----	-----	-----	-----	-----	-----



Stack

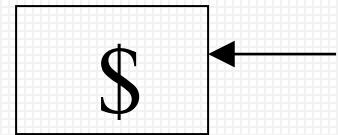
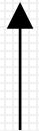
current
state



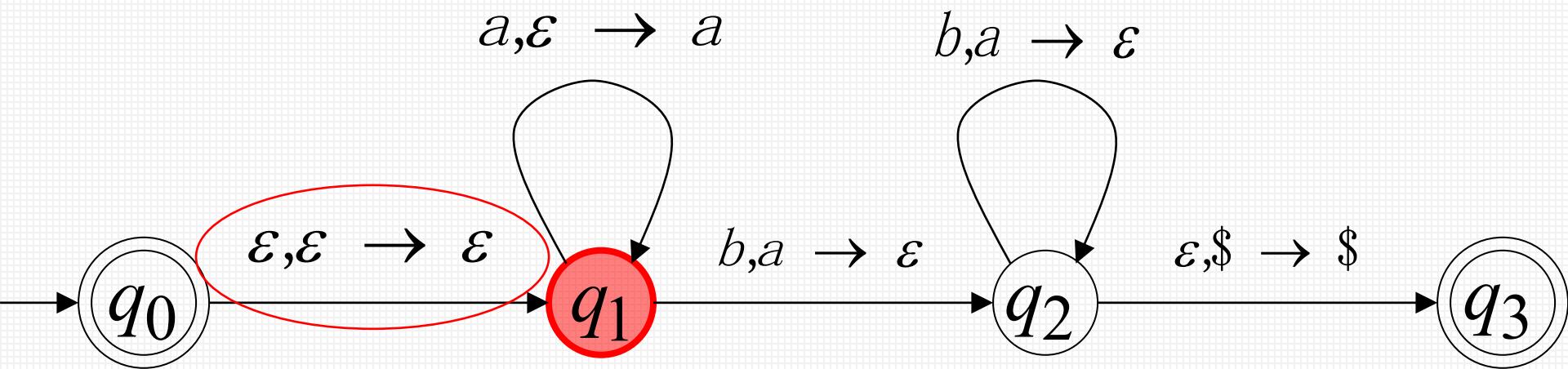
Time 1

Input

a	a	a	b	b	b
-----	-----	-----	-----	-----	-----



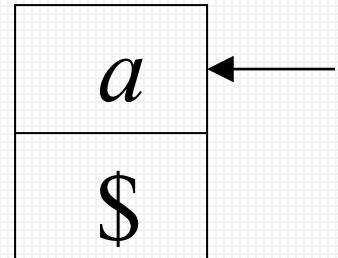
Stack



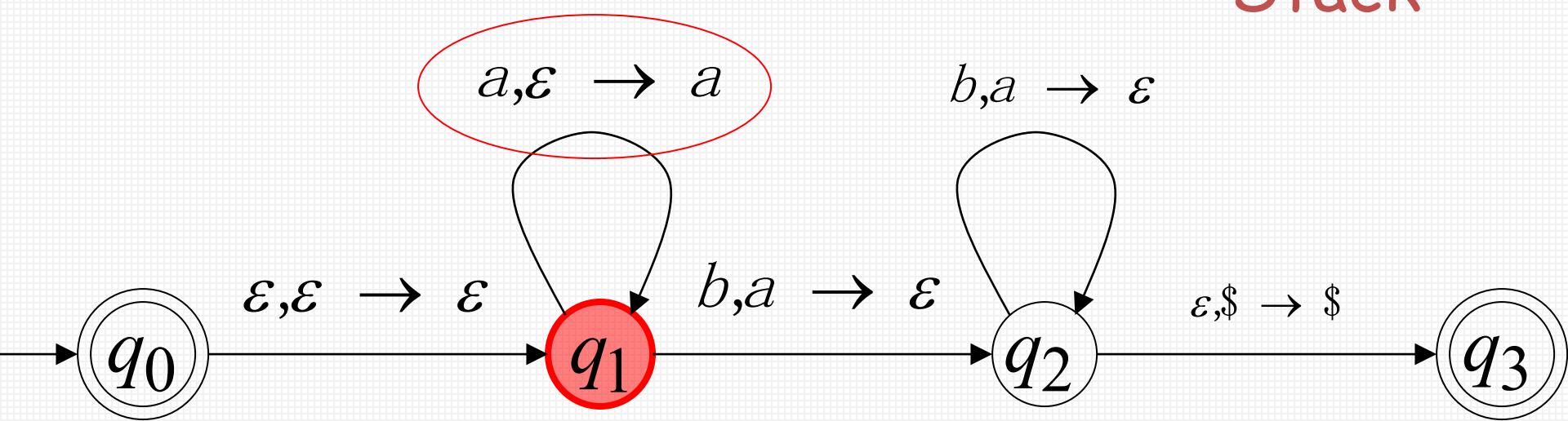
Time 2

Input

a	a	a	b	b	b
-----	-----	-----	-----	-----	-----



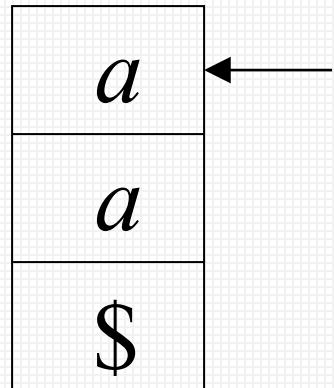
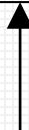
Stack



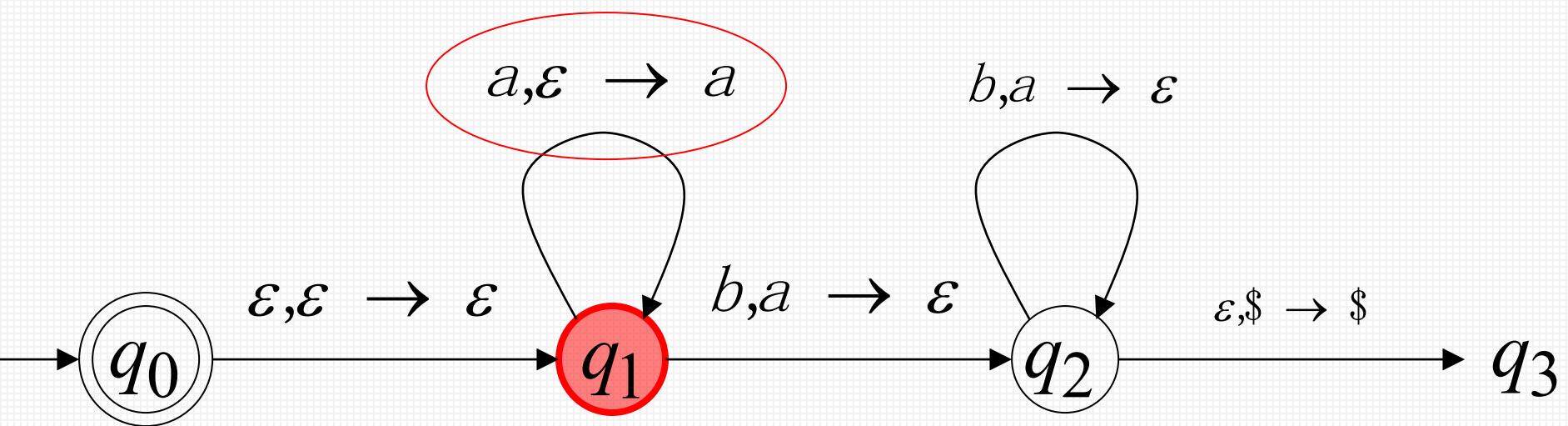
Time 3

Input

a	a	a	b	b	b
---	---	---	---	---	---



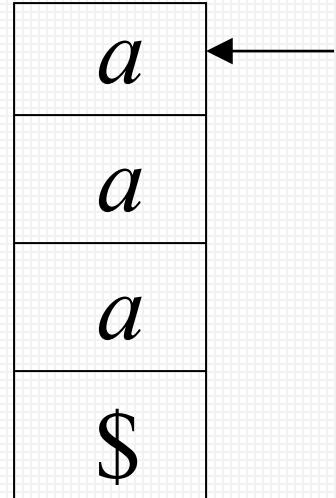
Stack



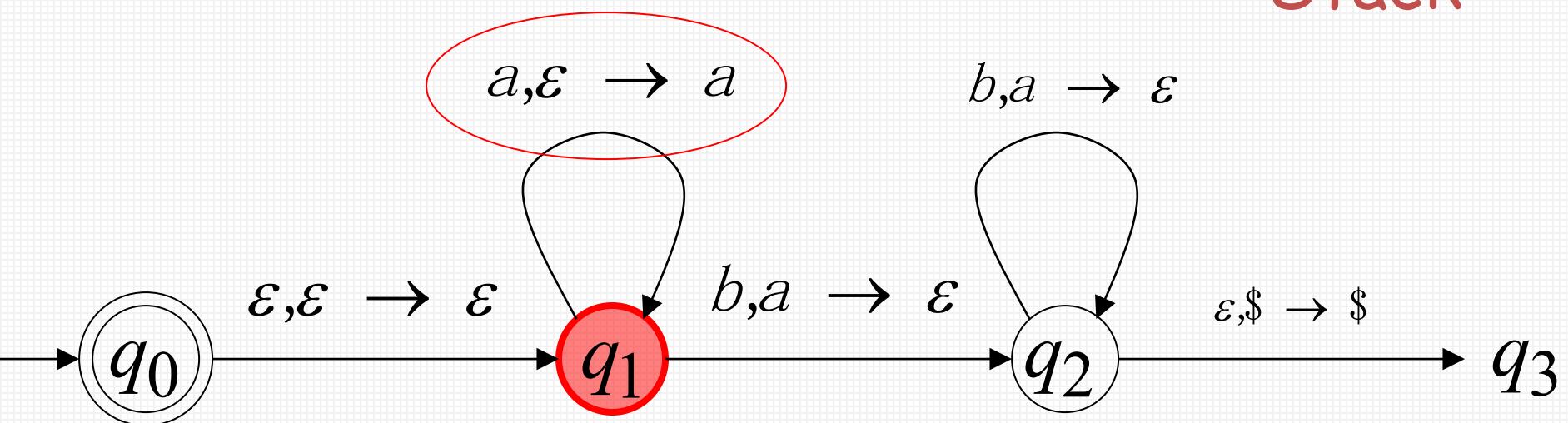
Time 4

Input

a	a	a	b	b	b
---	---	---	---	---	---



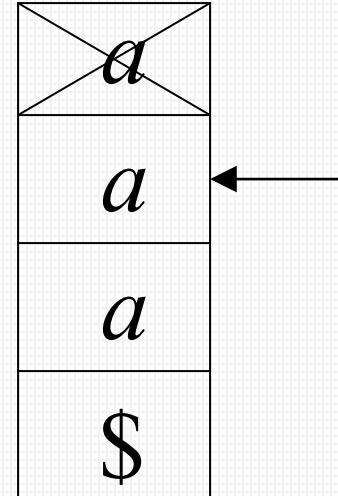
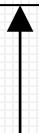
Stack



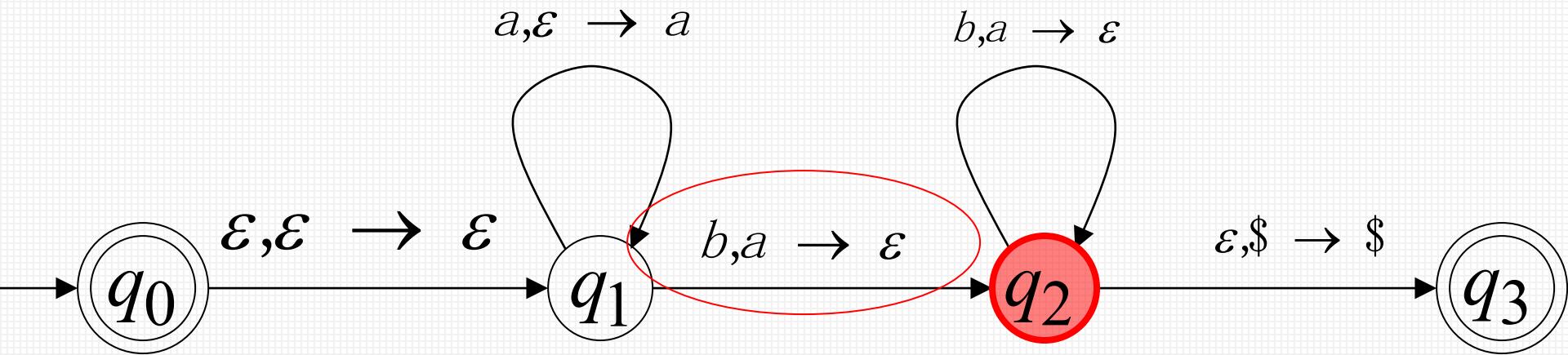
Time 5

Input

a	a	a	b	b	b
-----	-----	-----	-----	-----	-----



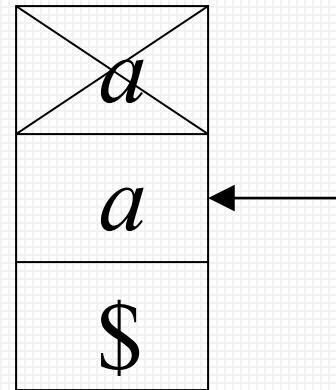
Stack



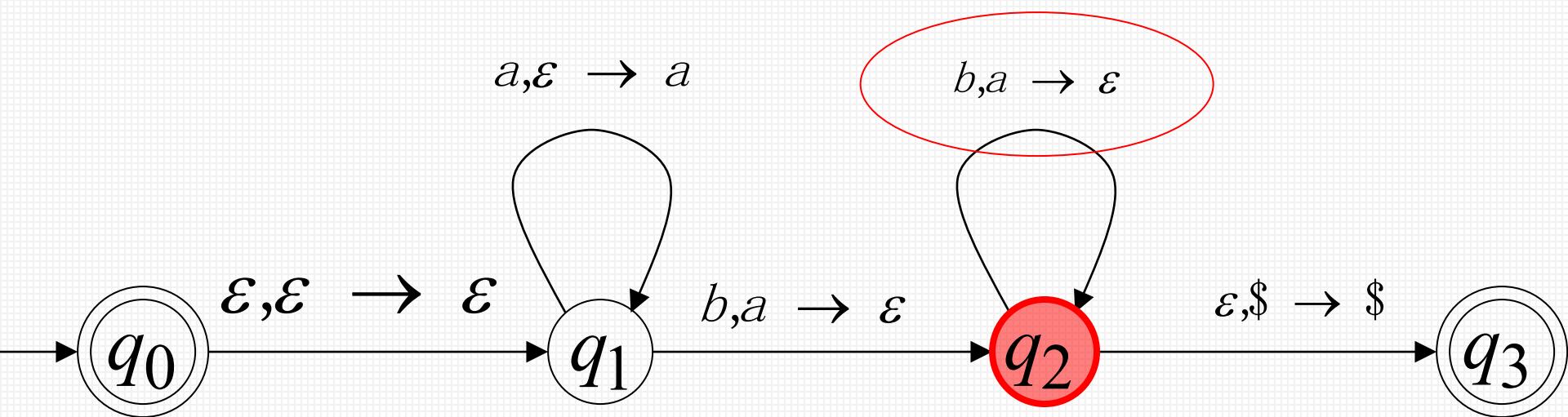
Time 6

Input

a	a	a	b	b	b
-----	-----	-----	-----	-----	-----



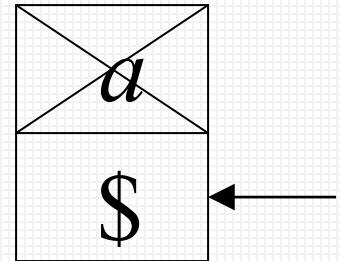
Stack



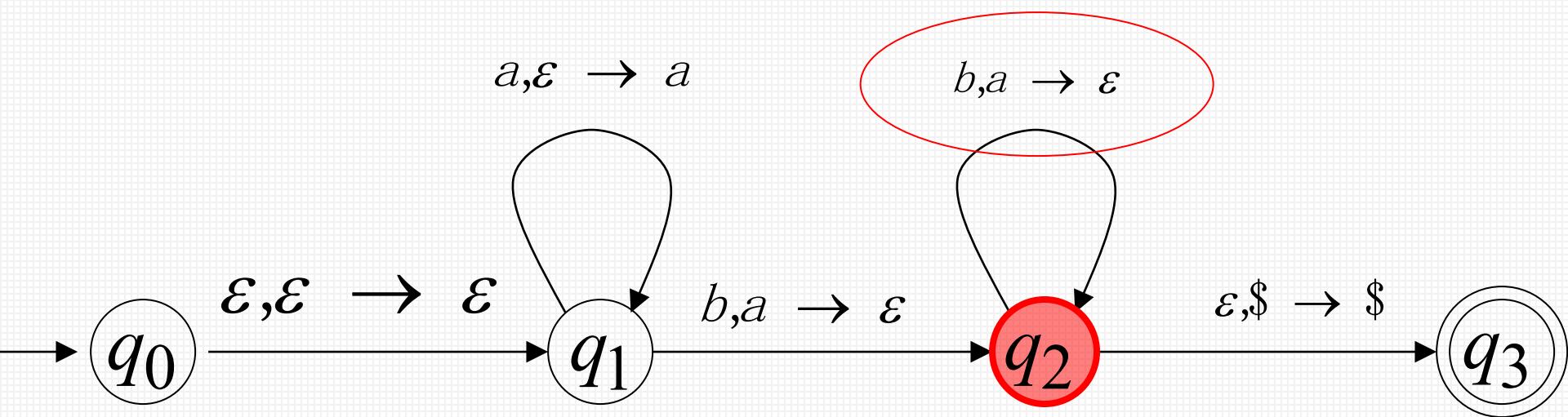
Time 7

Input

a	a	a	b	b	b
-----	-----	-----	-----	-----	-----



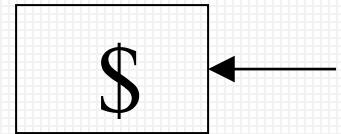
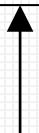
Stack



Time 8

Input

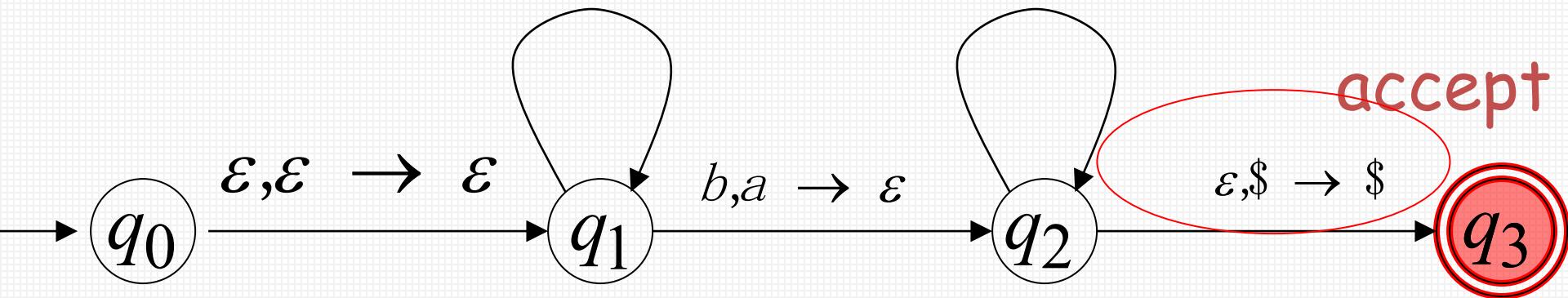
a	a	a	b	b	b
-----	-----	-----	-----	-----	-----



Stack

$$a, \epsilon \rightarrow a$$

$$b, a \rightarrow \epsilon$$



A string is accepted if there is
a computation such that:

All the input is consumed
AND

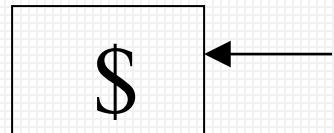
The last state is an accepting state

we do not care about the stack contents
at the end of the accepting computation

Rejection Example: Time 0

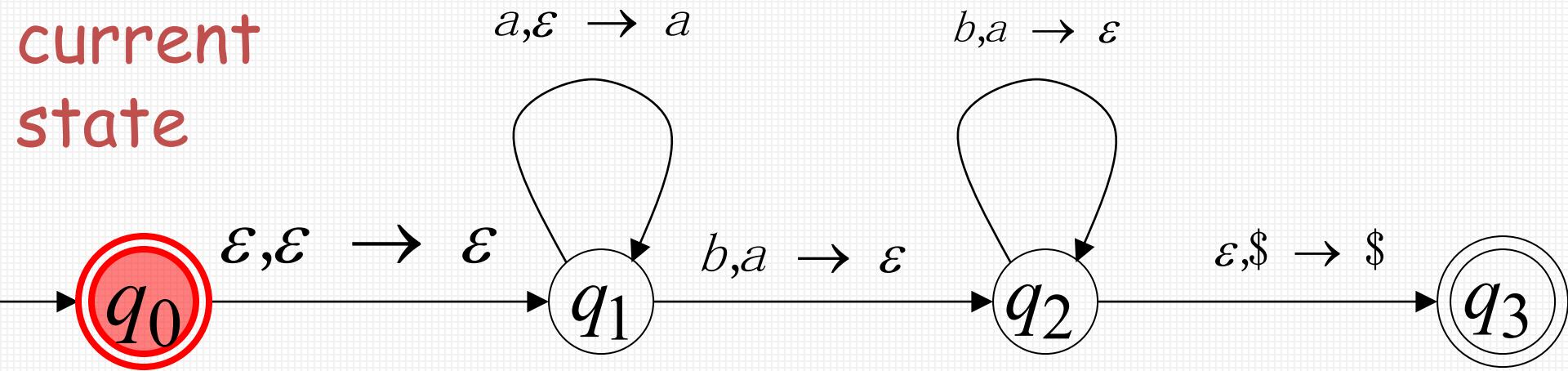
Input

a	a	b
-----	-----	-----



Stack

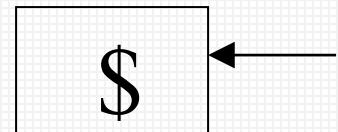
current
state



Rejection Example: Time 1

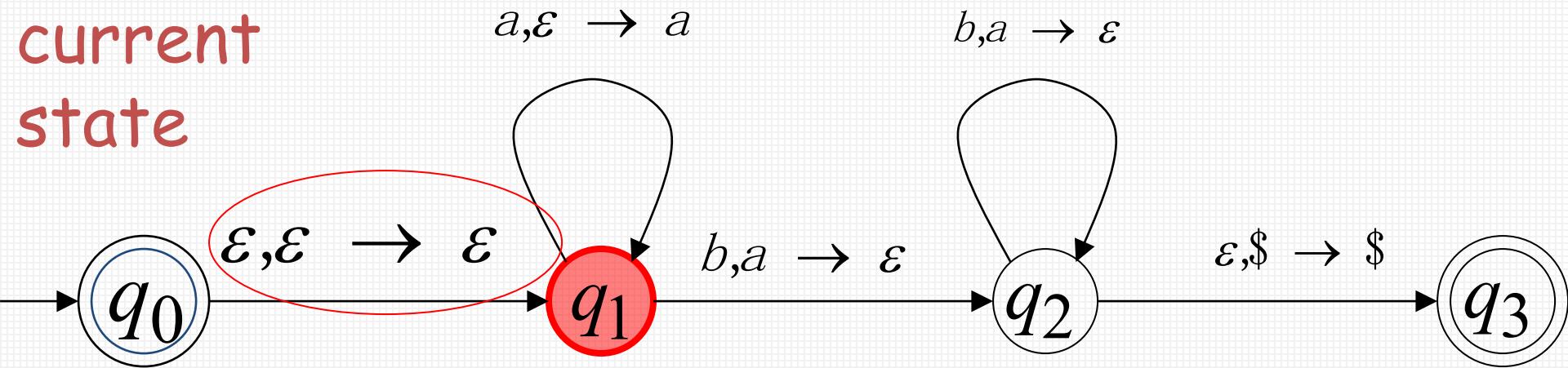
Input

a	a	b
-----	-----	-----



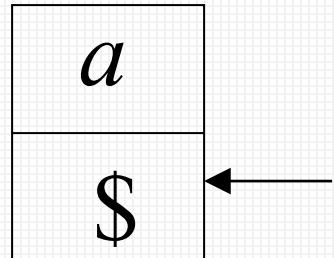
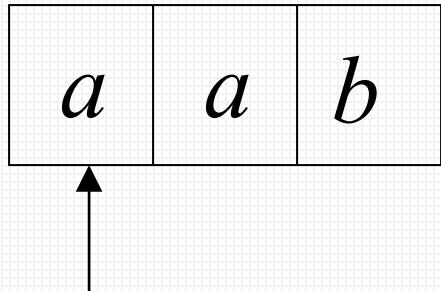
Stack

current
state



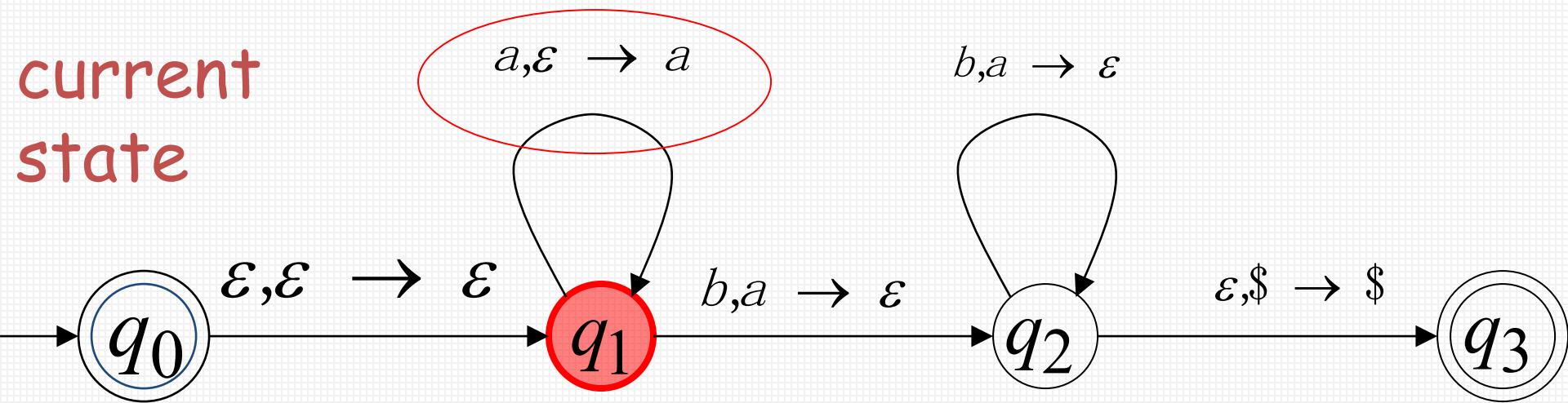
Rejection Example: Time 2

Input



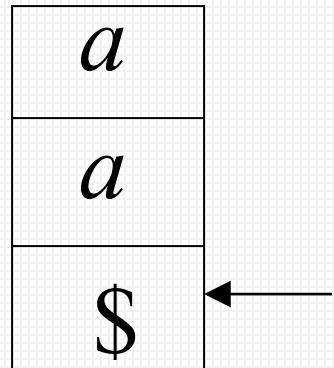
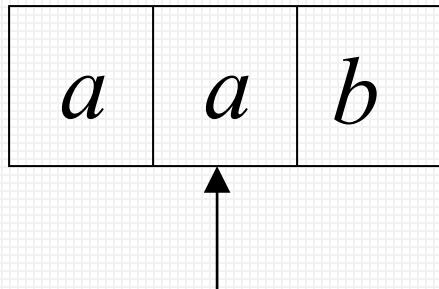
Stack

current
state



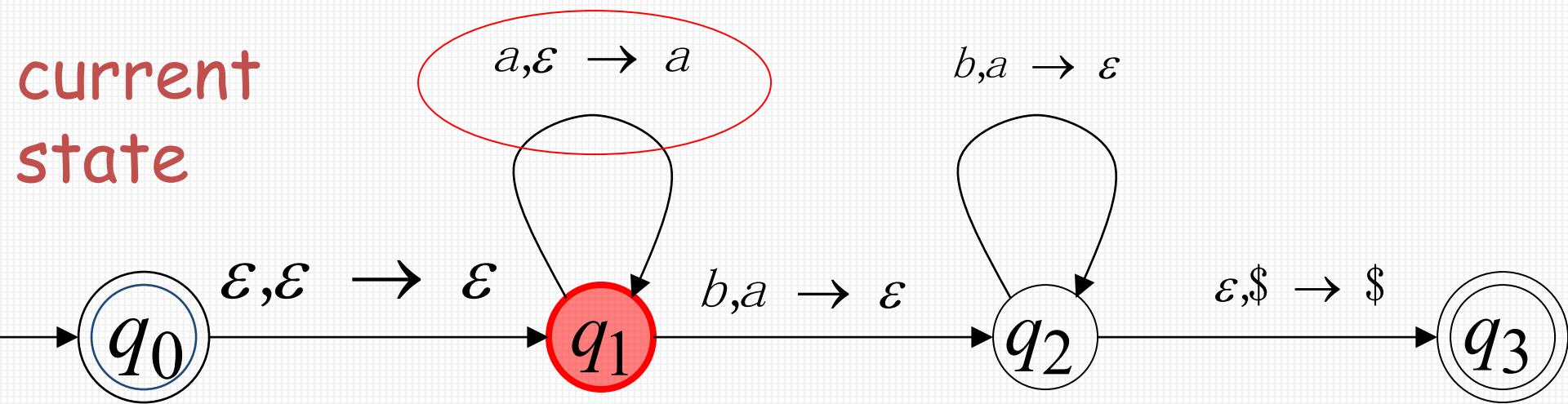
Rejection Example: Time 3

Input



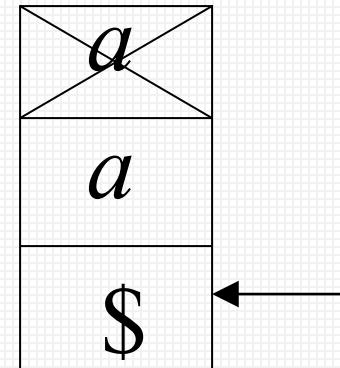
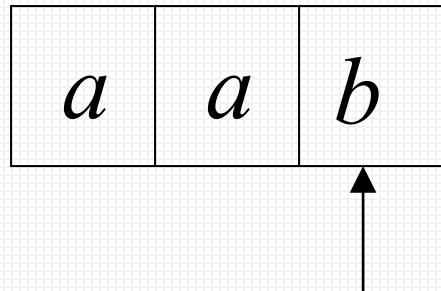
Stack

current
state



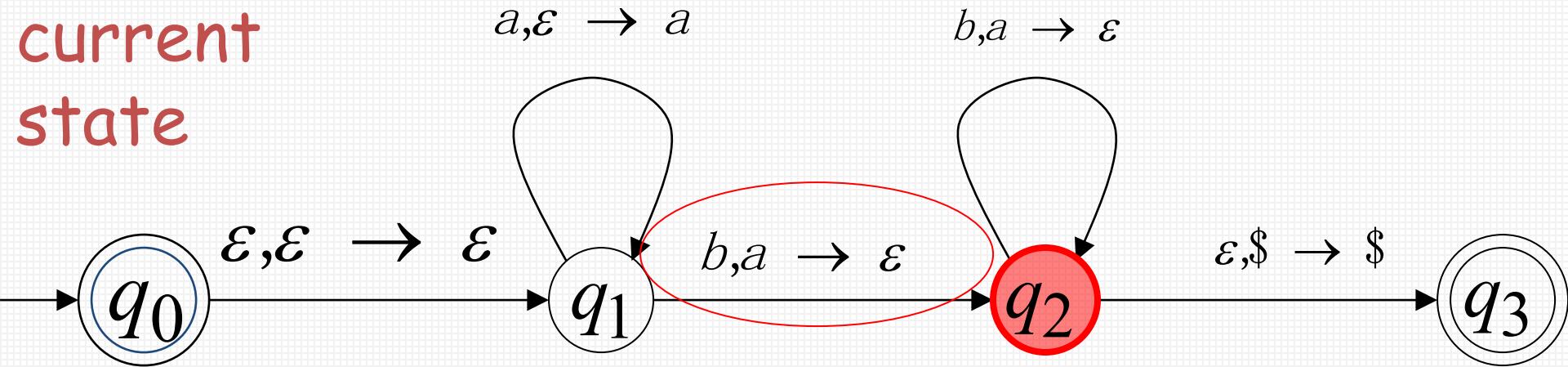
Rejection Example: Time 4

Input



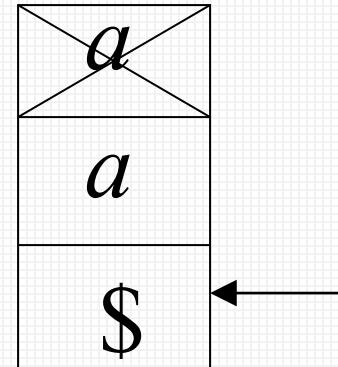
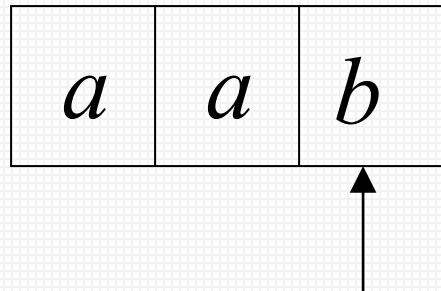
Stack

current
state



Rejection Example: Time 4

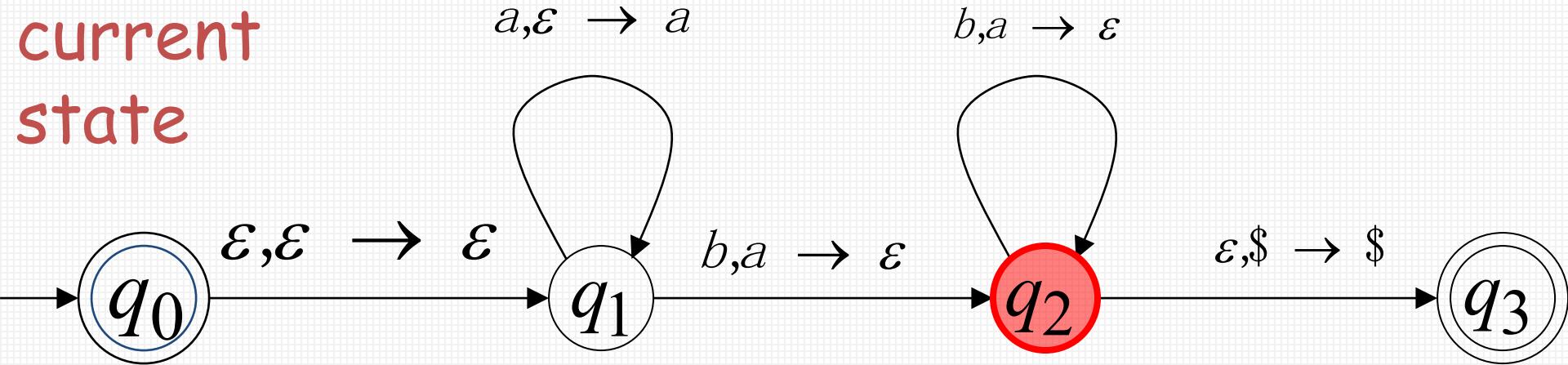
Input



Stack

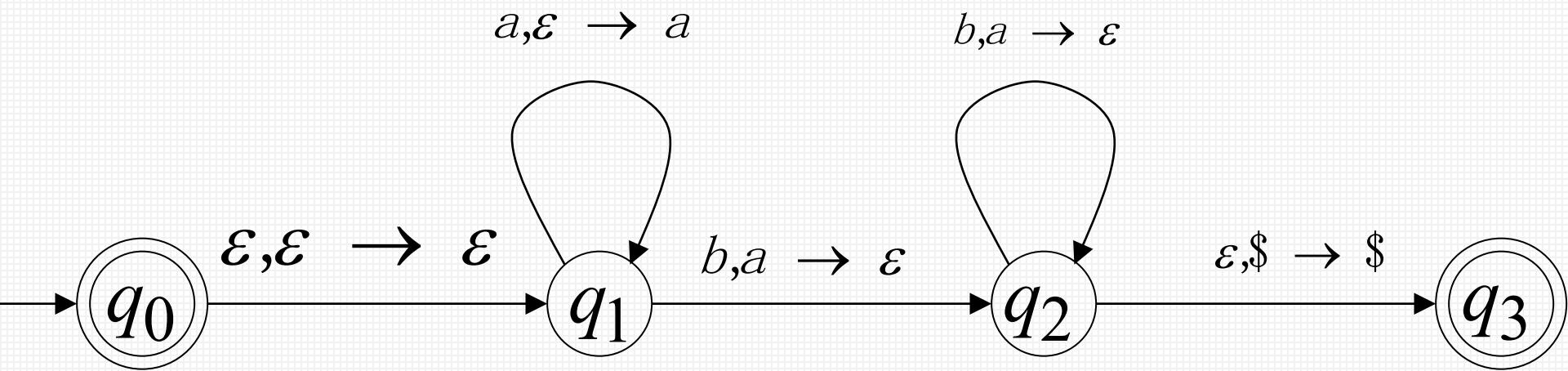
reject

current
state



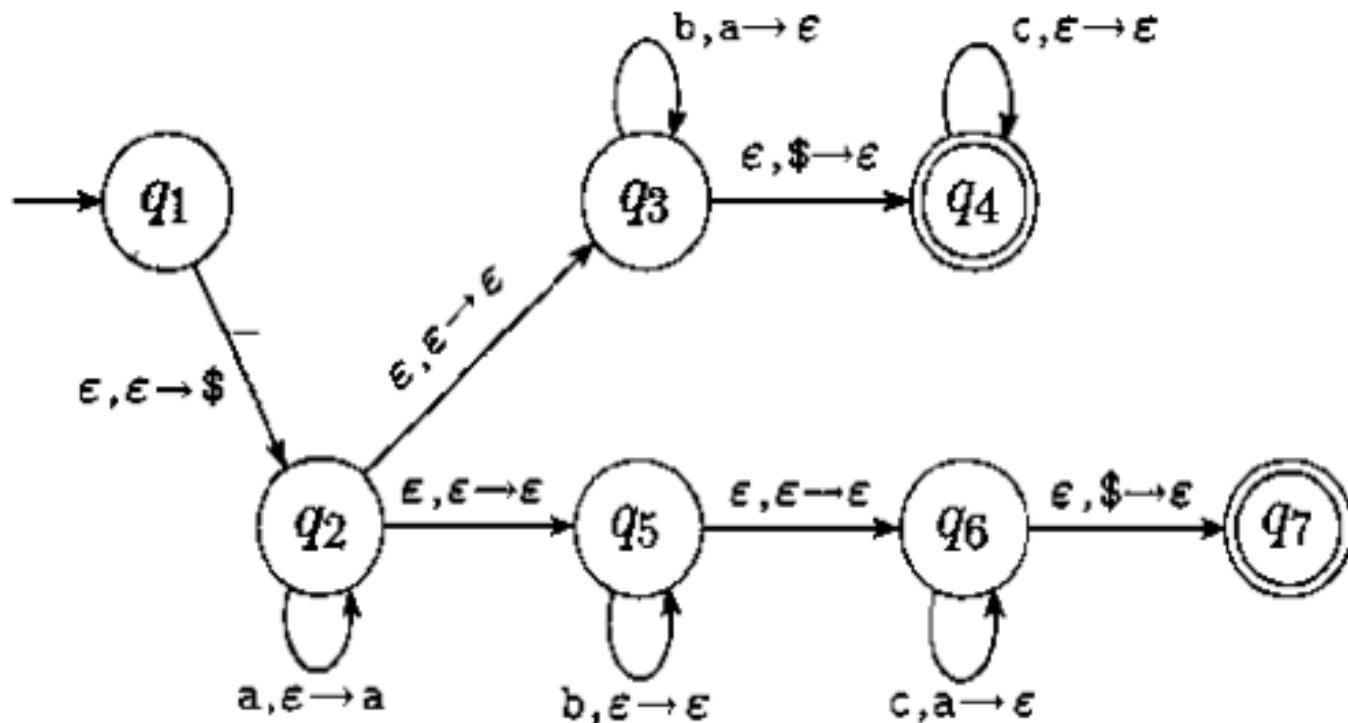
There is no accepting computation for aab

The string aab is rejected by the PDA



Example

- PDA recognizes language $\{a^i b^j c^k \mid i, j, k \geq 0 \text{ and } i = j \text{ or } i = k\}$.

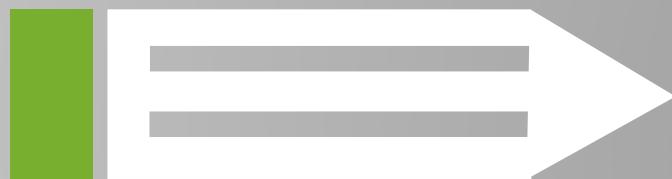


Review

- ✓ Formalities for PDAs
- ✓ Pushing & Popping Strings
- ✓ Non-Determinism
- ✓ Example PDA
- ✓ Accept and Reject

Lesson 5-2

Equivalence with Context-Free Languages



王轩
Wang
Xuan

$$\left\{ \begin{array}{l} \text{Context-Free} \\ \text{Languages} \\ (\text{Grammars}) \end{array} \right\} = \left\{ \begin{array}{l} \text{Languages} \\ \text{Accepted by} \\ \text{PDAs} \end{array} \right\}$$

$$\left\{ \begin{array}{l} \text{Context-Free} \\ \text{Languages} \\ (\text{Grammars}) \end{array} \right\} \subseteq \left\{ \begin{array}{l} \text{Languages} \\ \text{Accepted by} \\ \text{PDAs} \end{array} \right\}$$

Convert any context-free grammar G
to a PDA M with: $L(G) = L(M)$

Take an arbitrary context-free grammar G

We will convert G to a PDA M such that:

$$L(G) = L(M)$$

In another word, the main idea is to design a automaton so that it simulates the grammar.

Conversion Procedure

For each production in G

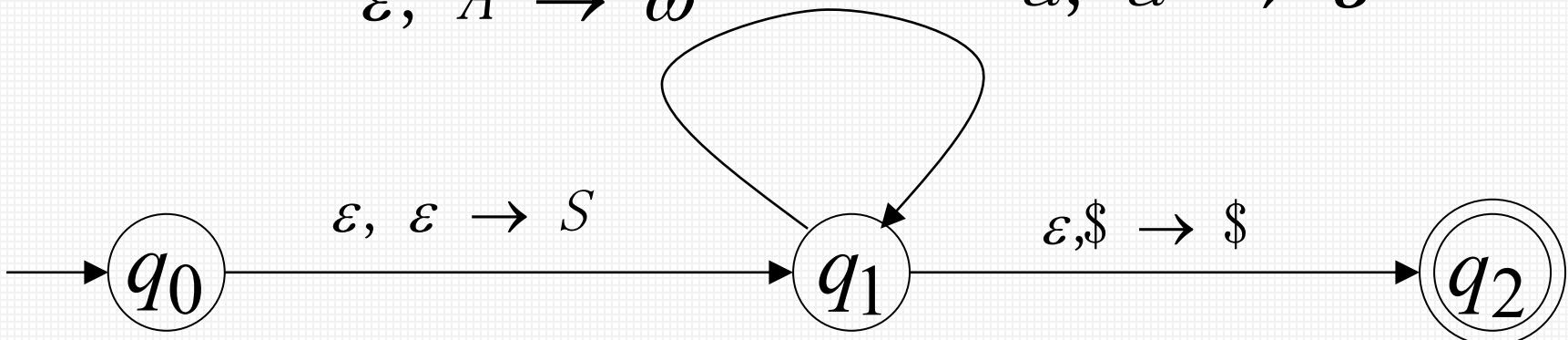
$$A \rightarrow \omega$$

Add transitions

$$\varepsilon, A \rightarrow \omega$$

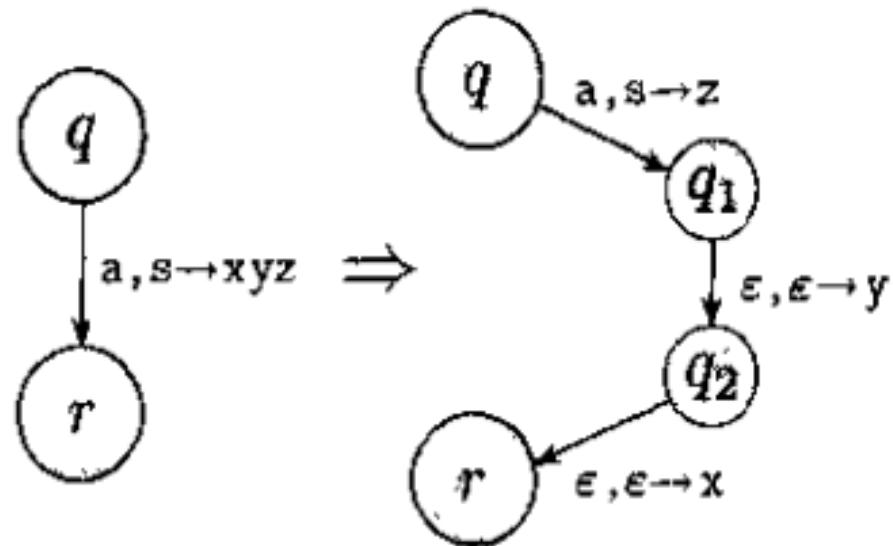
For each terminal in G

$$a, a \rightarrow \varepsilon$$



1. Place the marker symbol \$ and the start variable on the stack.
2. Repeat the following steps forever.
 - a. If the top of stack is a variable symbol A, nondeterministically select one of the rule for A and substitute A by the string on the right-hand side of the rule.
 - b. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.
 - c. If the top of stack is the symbol \$, enter the accept state. Doing so accepts the input if it has all been read.

$$(r, xyz) \in \delta(q, a, s)$$



The procedure popping s and pushing xyz , when reading a can be converted as popping s and pushing z , when reading a . Then pushing y and x in turn.

Grammar

$$S \rightarrow aSTb$$

$$S \rightarrow b$$

$$T \rightarrow Ta$$

$$T \rightarrow \varepsilon$$

Example

PDA

$$\varepsilon, S \rightarrow aSTb$$

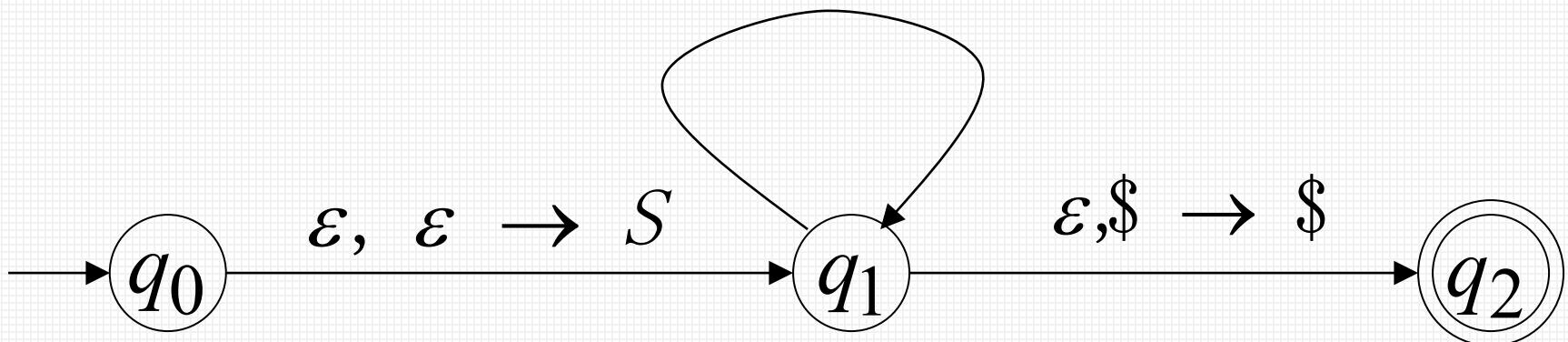
$$\varepsilon, S \rightarrow b$$

$$\varepsilon, T \rightarrow Ta$$

$$a, a \rightarrow \varepsilon$$

$$\varepsilon, T \rightarrow \varepsilon$$

$$b, b \rightarrow \varepsilon$$



Example:

Input

a	b	a	b
---	---	---	---



Time 0

$$\varepsilon, S \rightarrow aSTb$$

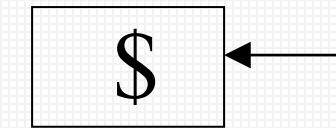
$$\varepsilon, S \rightarrow b$$

$$\varepsilon, T \rightarrow Ta$$

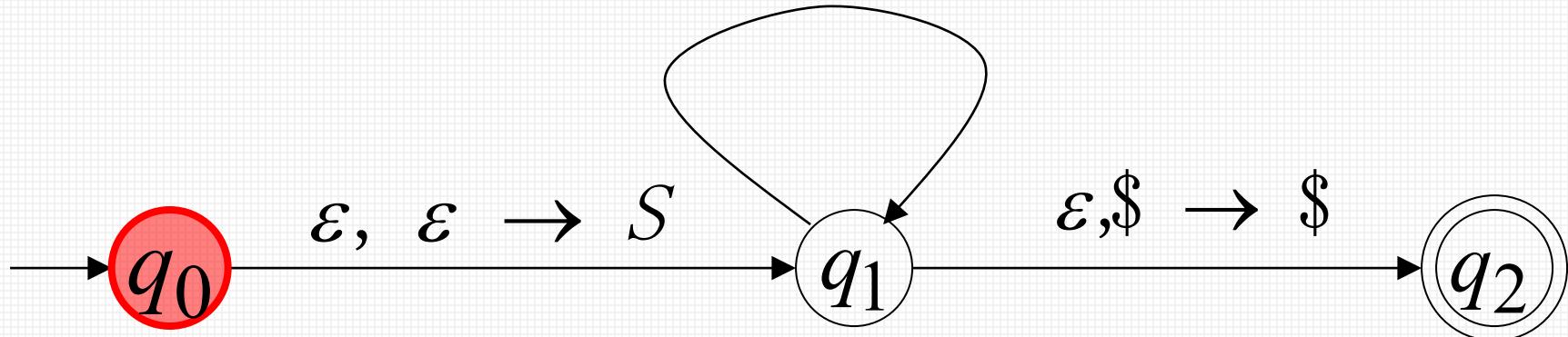
$$\varepsilon, T \rightarrow \varepsilon$$

$$a, a \rightarrow \varepsilon$$

$$b, b \rightarrow \varepsilon$$



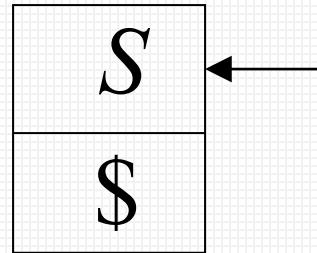
Stack



Derivation: S

Input

a	b	a	b
-----	-----	-----	-----



Time 1

$$\varepsilon, S \rightarrow aSTb$$

$$\varepsilon, S \rightarrow b$$

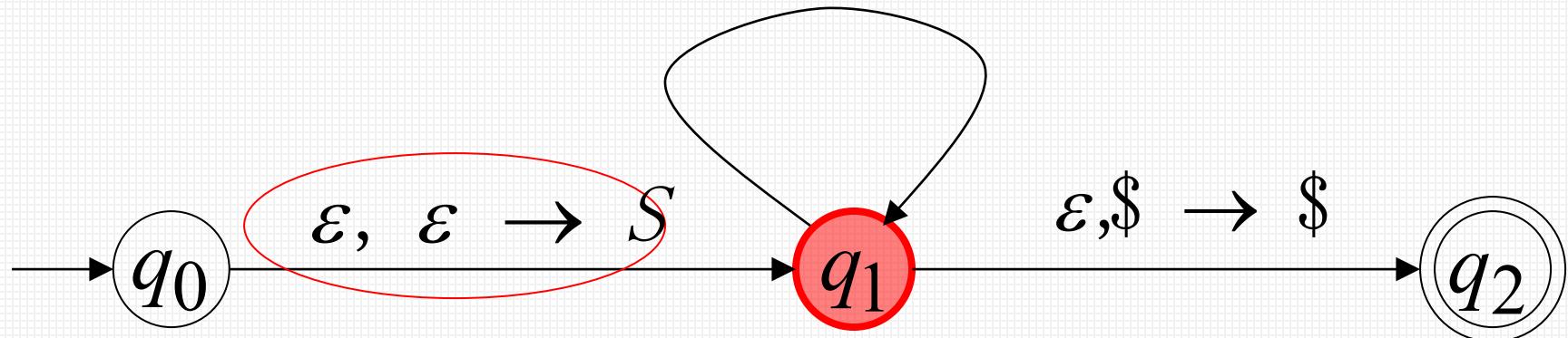
$$\varepsilon, T \rightarrow Ta$$

$$a, a \rightarrow \varepsilon$$

$$\varepsilon, T \rightarrow \varepsilon$$

$$b, b \rightarrow \varepsilon$$

Stack



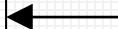
Derivation: $S \Rightarrow aSTb$

Input

a	b	a	b
---	---	---	---



a
S
T
b
\$



Time 2

$\varepsilon, S \rightarrow aSTb$

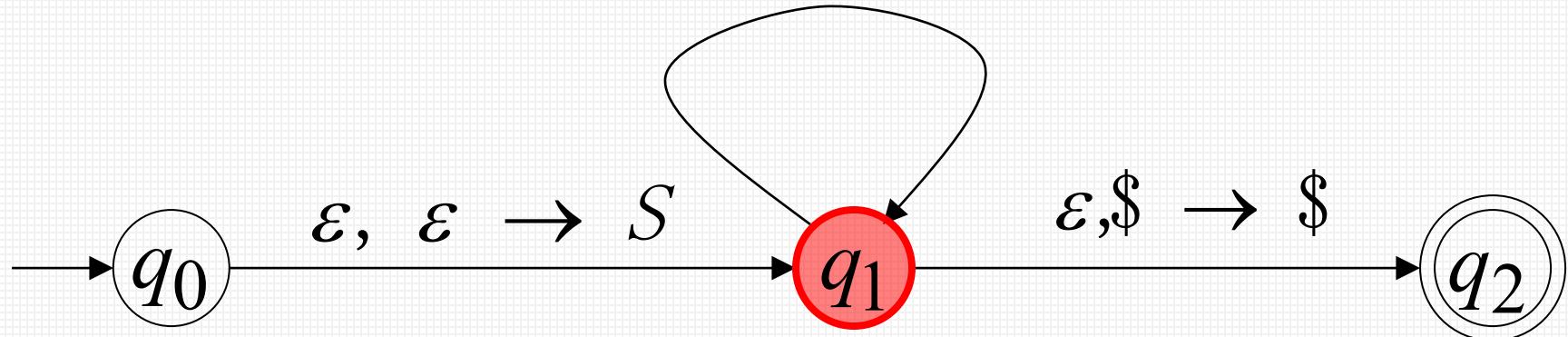
$\varepsilon, S \rightarrow b$

$\varepsilon, T \rightarrow Ta$

$\varepsilon, T \rightarrow \varepsilon$

$a, a \rightarrow \varepsilon$

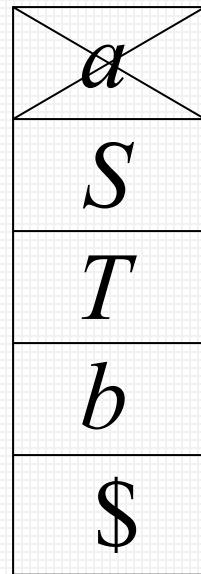
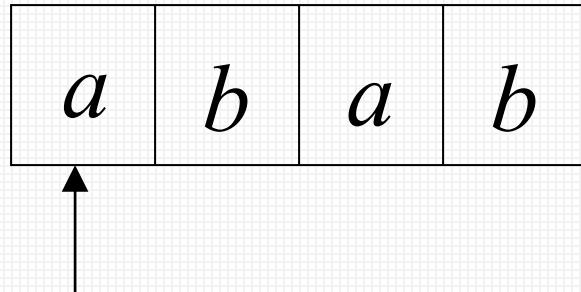
$b, b \rightarrow \varepsilon$



Stack

Derivation: $S \Rightarrow aSTb$

Input



Time 3

$$\varepsilon, S \rightarrow aSTb$$

$$\varepsilon, S \rightarrow b$$

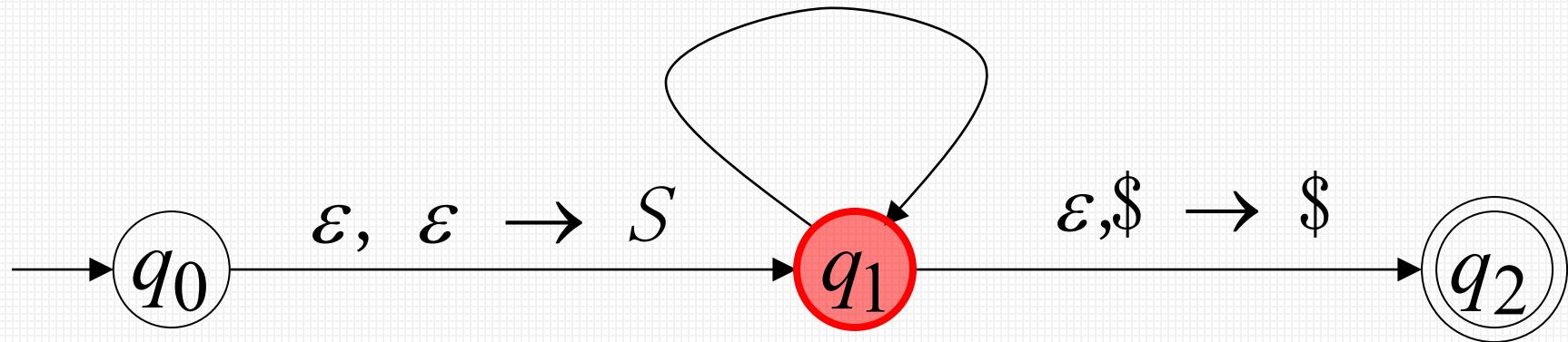
$$\varepsilon, T \rightarrow Ta$$

$$\varepsilon, T \rightarrow \varepsilon$$

$$a, a \rightarrow \varepsilon$$

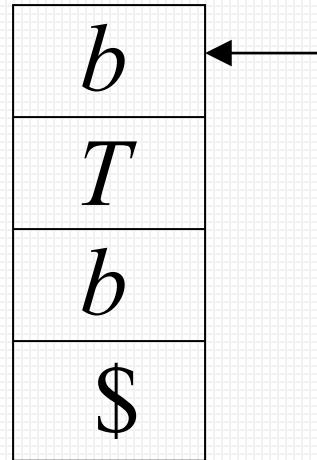
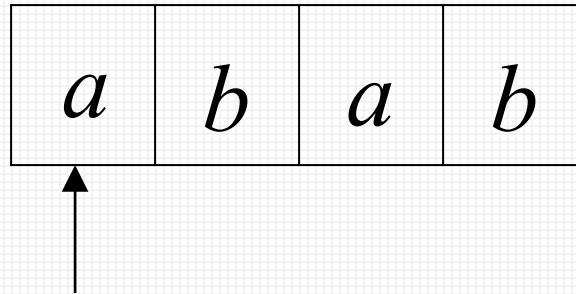
$$b, b \rightarrow \varepsilon$$

Stack



Derivation: $S \Rightarrow aSTb \Rightarrow abTb$

Input



Time 4

$$\varepsilon, S \rightarrow aSTb$$

$$\varepsilon, S \rightarrow b$$

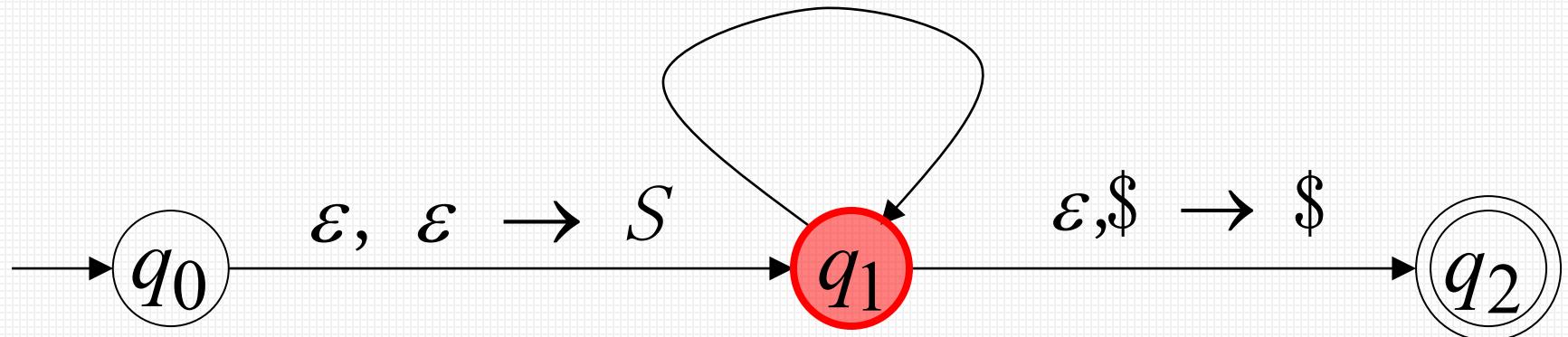
$$\varepsilon, T \rightarrow Ta$$

$$\varepsilon, T \rightarrow \varepsilon$$

$$a, a \rightarrow \varepsilon$$

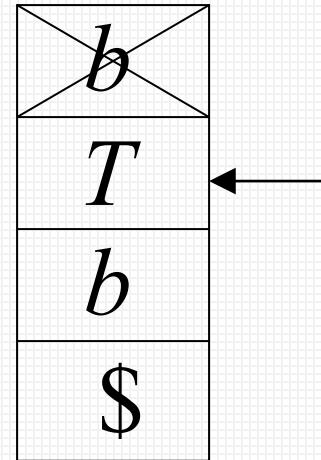
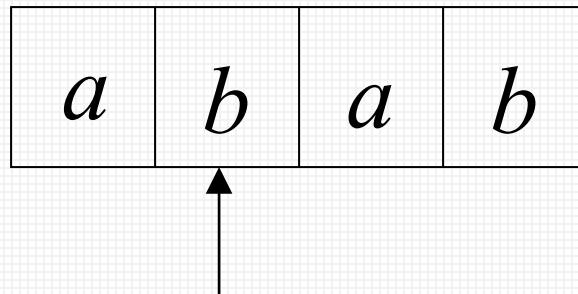
$$b, b \rightarrow \varepsilon$$

Stack



Derivation: $S \Rightarrow aSTb \Rightarrow abTb$

Input



Time 5

$$\varepsilon, S \rightarrow aSTb$$

$$\varepsilon, S \rightarrow b$$

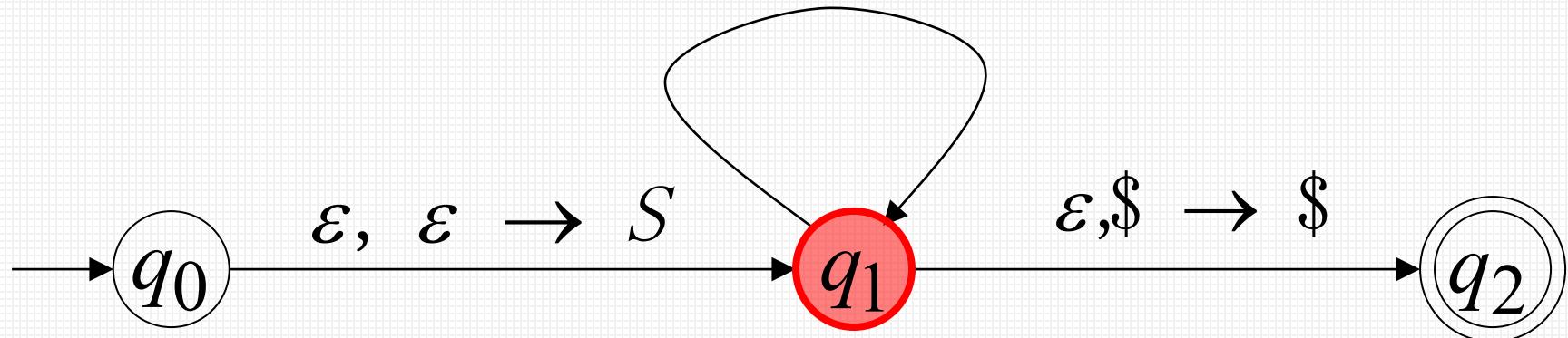
$$\varepsilon, T \rightarrow Ta$$

$$\varepsilon, T \rightarrow \varepsilon$$

$$a, a \rightarrow \varepsilon$$

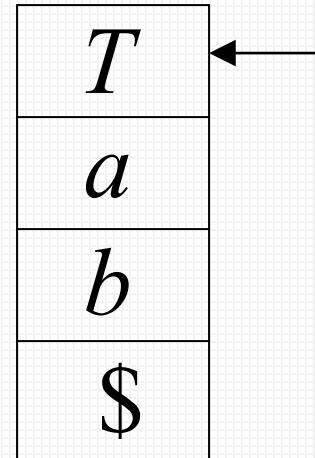
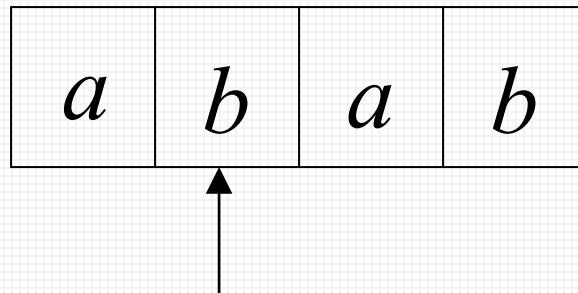
$$b, b \rightarrow \varepsilon$$

Stack



Derivation: $S \Rightarrow aSTb \Rightarrow abTb \Rightarrow abTab$

Input



Time 6

$$\varepsilon, S \rightarrow aSTb$$

$$\varepsilon, S \rightarrow b$$

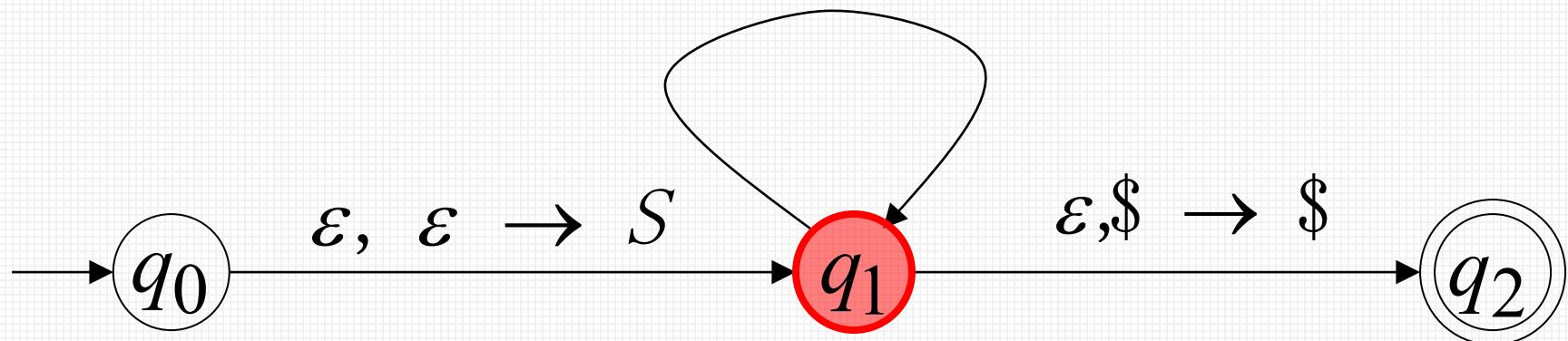
$$\varepsilon, T \rightarrow Ta$$

$$\varepsilon, T \rightarrow \varepsilon$$

Stack

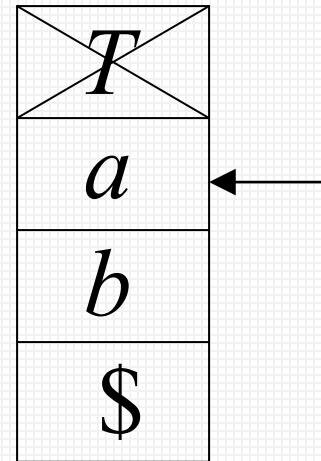
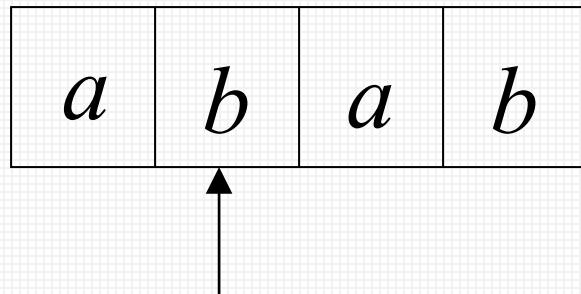
$$a, a \rightarrow \varepsilon$$

$$b, b \rightarrow \varepsilon$$



Derivation: $S \Rightarrow aSTb \Rightarrow abTb \Rightarrow abTab \Rightarrow abab$

Input



Time 7

$$\varepsilon, S \rightarrow aSTb$$

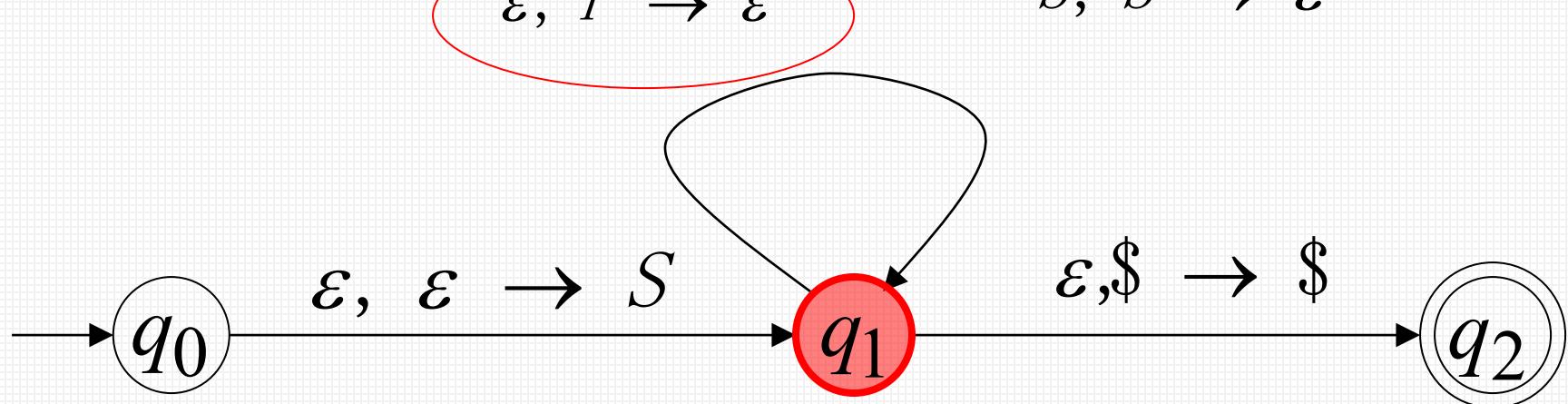
$$\varepsilon, S \rightarrow b$$

$$\varepsilon, T \rightarrow Ta$$

$$\varepsilon, T \rightarrow \varepsilon$$

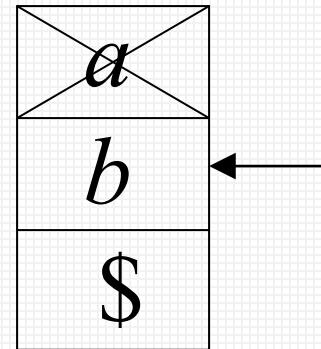
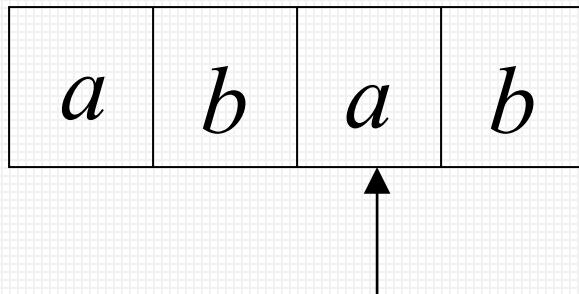
$$a, a \rightarrow \varepsilon$$

$$b, b \rightarrow \varepsilon$$



Derivation: $S \Rightarrow aSTb \Rightarrow abTb \Rightarrow abTab \Rightarrow abab$

Input



Time 8

$$\varepsilon, S \rightarrow aSTb$$

$$\varepsilon, S \rightarrow b$$

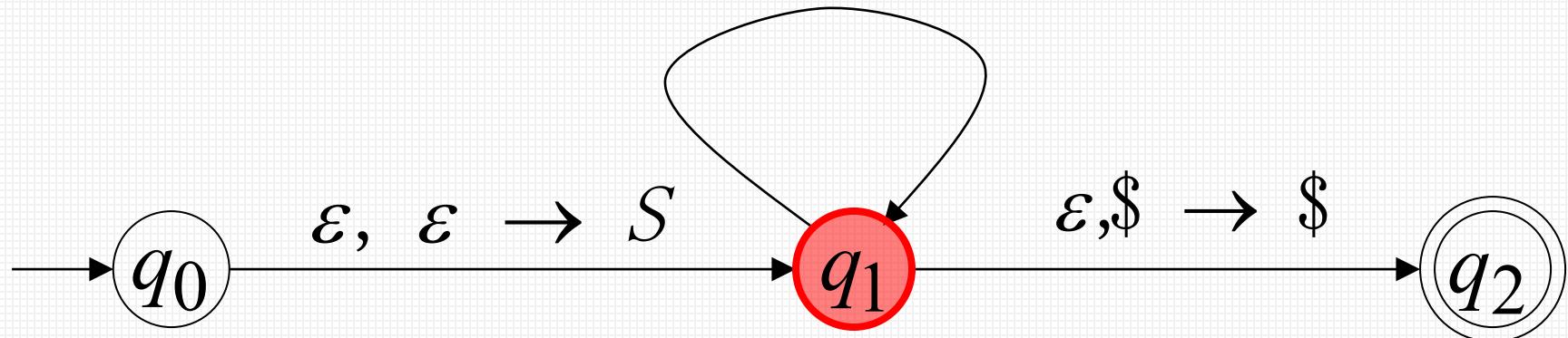
$$\varepsilon, T \rightarrow Ta$$

$$\varepsilon, T \rightarrow \varepsilon$$

Stack

$$a, a \rightarrow \varepsilon$$

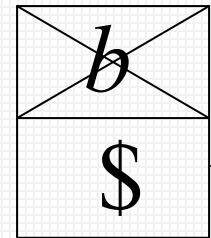
$$b, b \rightarrow \varepsilon$$



Derivation: $S \Rightarrow aSTb \Rightarrow abTb \Rightarrow abTab \Rightarrow abab$

Input

a	b	a	b
-----	-----	-----	-----



Time 9

$$\varepsilon, S \rightarrow aSTb$$

$$\varepsilon, S \rightarrow b$$

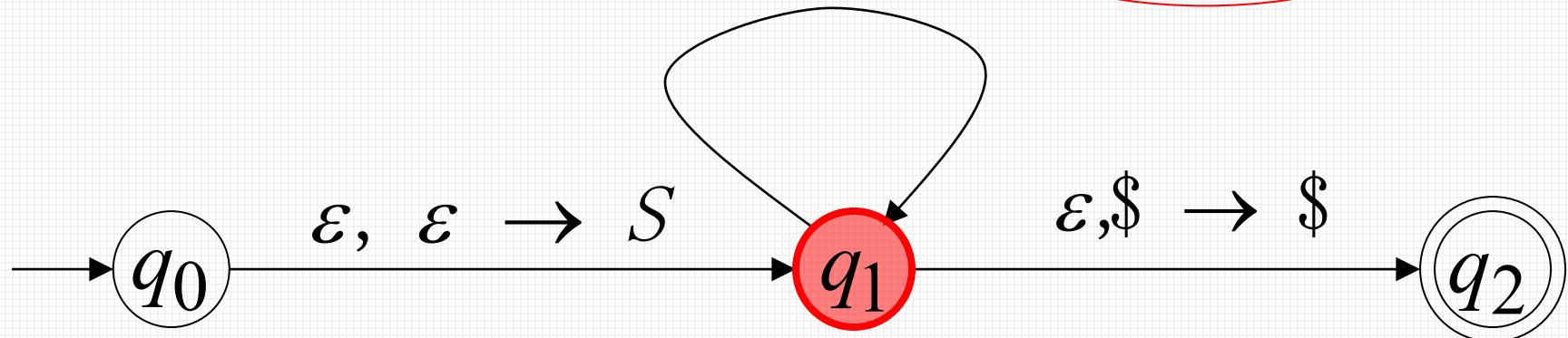
$$\varepsilon, T \rightarrow Ta$$

$$\varepsilon, T \rightarrow \varepsilon$$

$$a, a \rightarrow \varepsilon$$

$$b, b \rightarrow \varepsilon$$

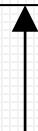
Stack



Derivation: $S \Rightarrow aSTb \Rightarrow abTb \Rightarrow abTab \Rightarrow abab$

Input

a	b	a	b
-----	-----	-----	-----



Time 10

$\varepsilon, S \rightarrow aSTb$

$\varepsilon, S \rightarrow b$

$\varepsilon, T \rightarrow Ta$

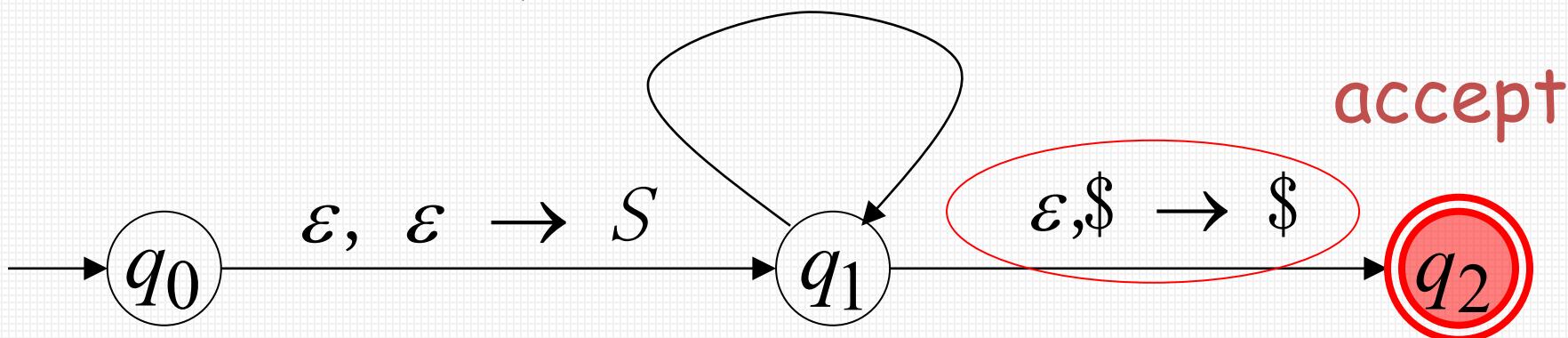
$\varepsilon, T \rightarrow \varepsilon$

$a, a \rightarrow \varepsilon$

$b, b \rightarrow \varepsilon$

\$

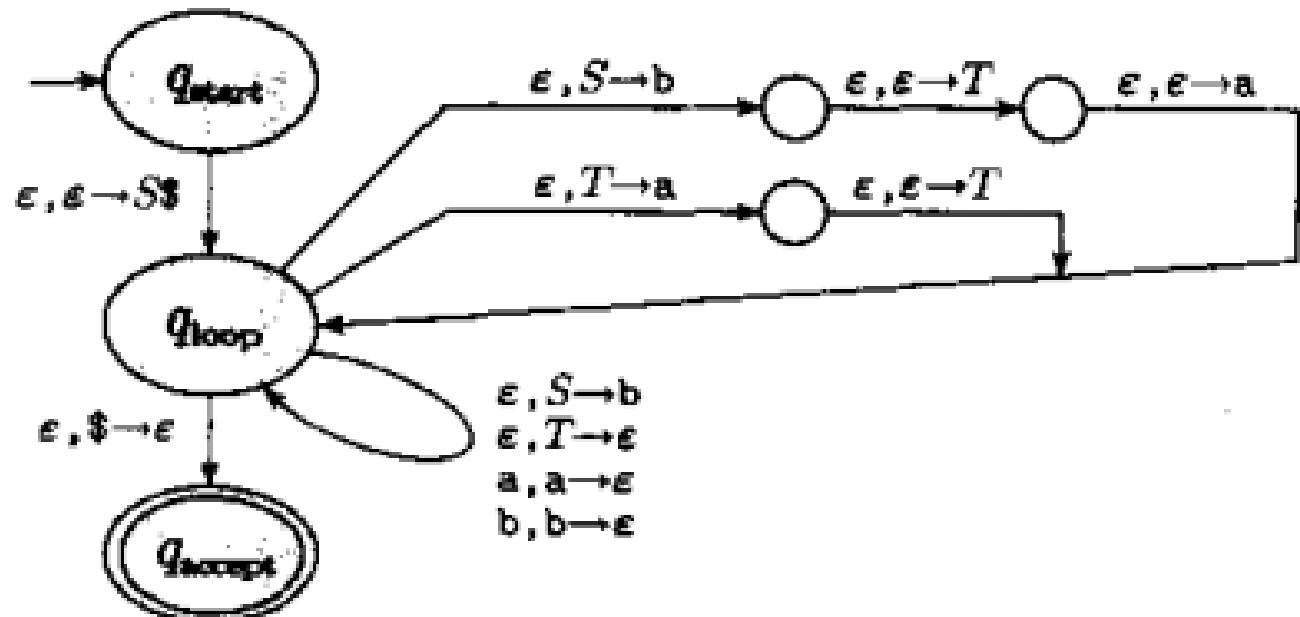
Stack



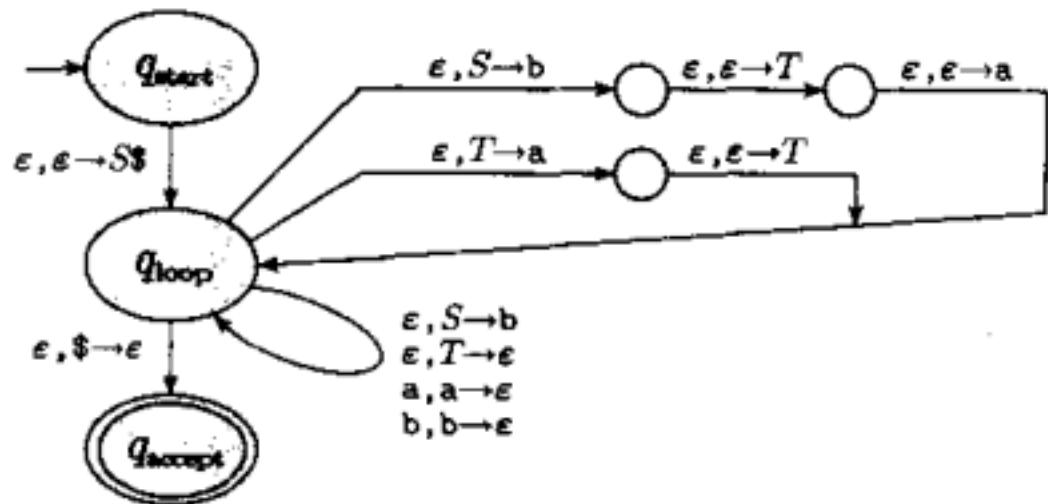
Example

$S \rightarrow aTb|b$

$T \rightarrow Ta|\epsilon$



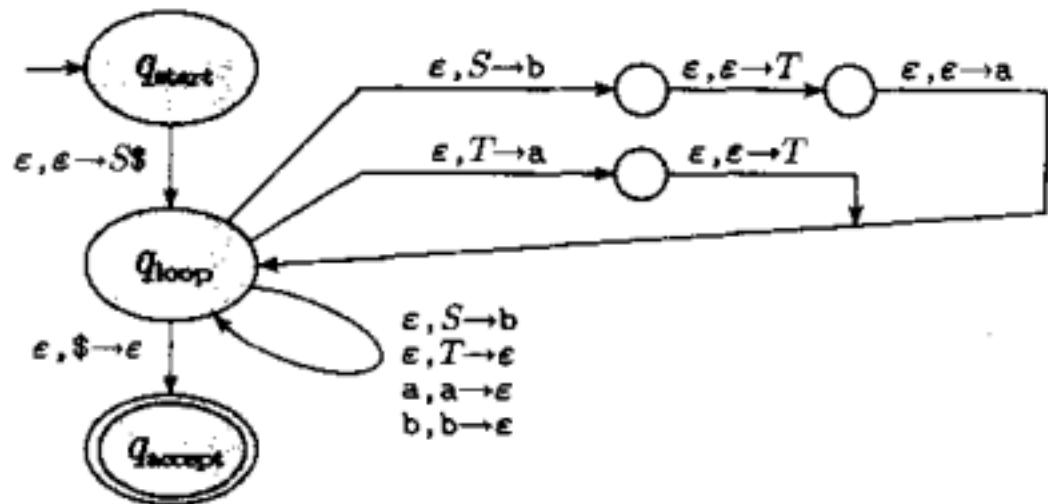
$S \rightarrow aTb|b$
 $T \rightarrow Ta| \epsilon$



$$\delta(q_{start}, \epsilon, \epsilon) = \{(q_{loop}, s\$)\}$$

$$\delta(q_{loop}, \epsilon, \$) = \{(q_{accept}, \epsilon)\}$$

$S \rightarrow aTb|b$
 $T \rightarrow Ta|\varepsilon$



$$\delta(q_{loop}, \varepsilon, S) = \{(q_{loop}, aTb), (q_{loop}, b)\}$$

$$\delta(q_{start}, \varepsilon, \varepsilon) = \{(q_{loop}, S\$)\}$$

$$\delta(q_{loop}, \varepsilon, \$) = \{(q_{accept}, \varepsilon)\}$$

$$\delta(q_{loop}, \varepsilon, T) = \{(q_{loop}, Ta), (q_{loop}, \varepsilon)\}$$

$$\delta(q_{loop}, b, b) = \{(q_{loop}, \varepsilon)\}$$

$$\left\{ \begin{array}{l} \text{Context-Free} \\ \text{Languages} \\ (\text{Grammars}) \end{array} \right\} \equiv \left\{ \begin{array}{l} \text{Languages} \\ \text{Accepted by} \\ \text{PDAs} \end{array} \right\}$$

Convert any PDA M to a context-free grammar G with: $L(G) = L(M)$

We have a PDA P , and we want to make a CFG G that generates all the strings that P accepts

We will convert a PDA M to CFG G such that:

$$L(M) = L(G)$$

Conversion Procedure

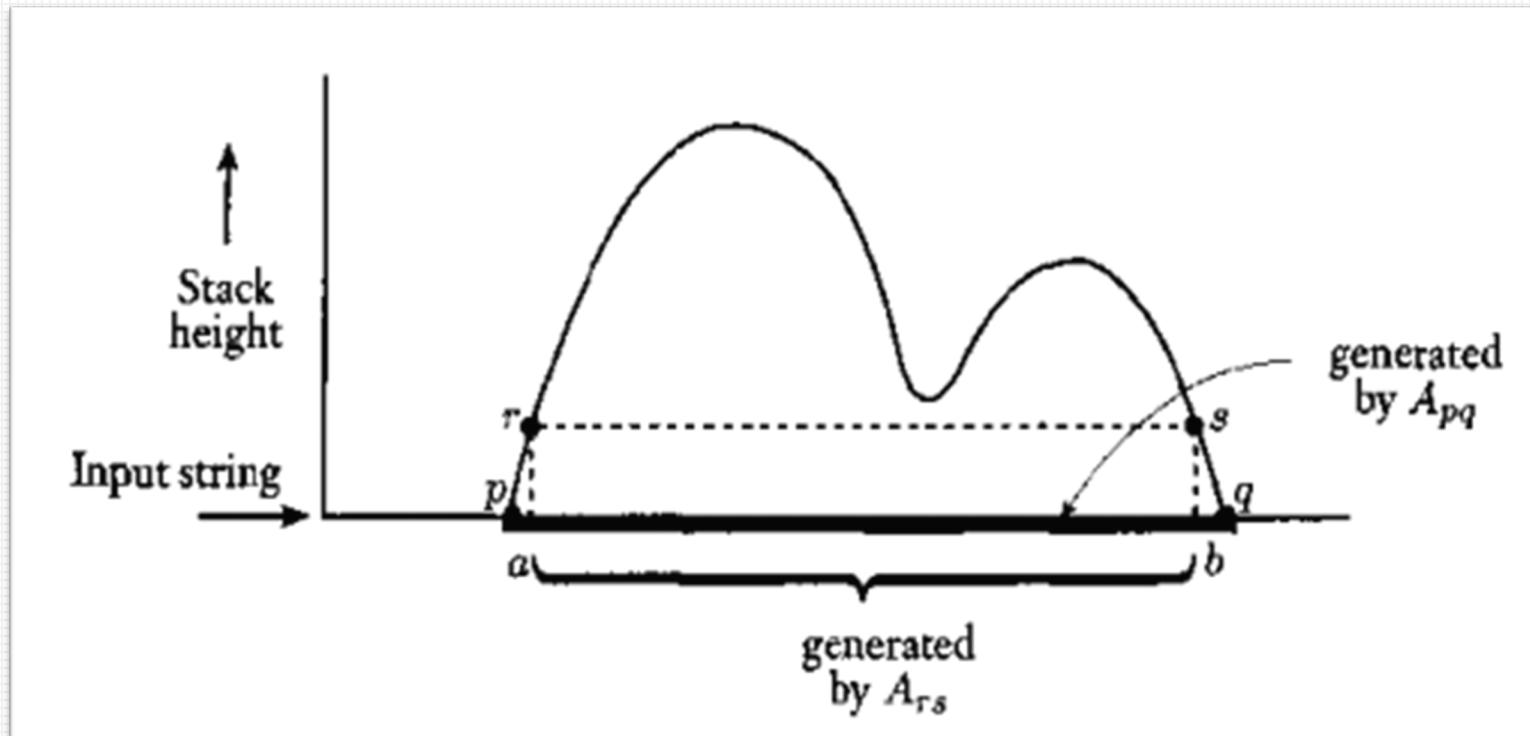
- First, we simplify our task by modifying P slightly to give it the following three features.
 - 1. It has a single accept, q_{accept} .
 - 2. It empties its stack before accepting.
 - 3. Each transition either pushes a symbol onto the stack (a push move) or pops one off the stack(a pop move), but it does not do both at the same time.

Two cases of computation

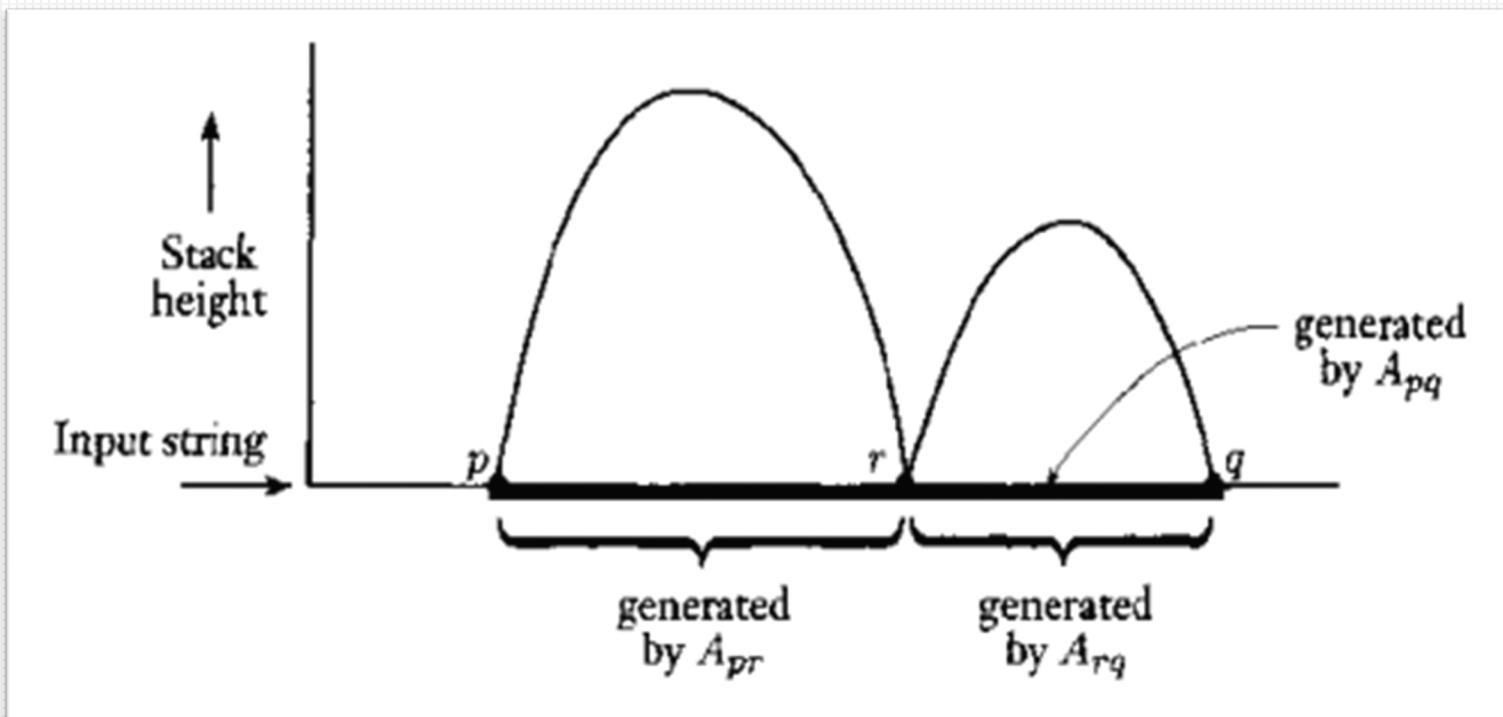
For any input string x , P 's first move on x must be a push, the last move on x must be a pop. Two cases occur during P 's computation.

- 1. The stack is empty only at the beginning and the end of P 's computation on x .
- 2. The stack becomes empty at some point.

- The stack is empty only at the beginning and the end of P 's computation on x .
- $A_{pq} \rightarrow a A_{rs} b$



- ▶ The stack becomes empty at some point.
- $A_{pq} \rightarrow A_{pr} A_{rq}$



Conversion Procedure

$$P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{\text{accept}}\})$$

- Construct $G=(V, T, S, P)$
- The variables V of G are $\{A_{pq} | p, q \in Q\}$.
- The terminal symbols T
- The start variable S is $A_{q_0, q_{\text{accept}}}$.
- The set of productions P are defined as below:
 - 1. For each $p, q, r, s \in Q, t \in \Gamma$, and $a, b \in \Sigma_\varepsilon$, if $\delta(p, a, \varepsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ε) , put the rule $A_{pq} \rightarrow a A_{rs} b$ in P .
 - 2. For each $p, q, r \in Q$, put the rule $A_{pq} \rightarrow A_{pr} A_{rq}$ in P .
 - 3. For each $p \in Q$, put the rule $A_{pp} \rightarrow \varepsilon$ in P .

- Now we prove that this construction works by demonstrating that A_{pq} generates x if and only if x can bring PDA P from p with empty stack to q with empty stack.
- Consider each direction of the proof.
 - 1. A generates $x \Rightarrow x$ can bring P from p to q with empty stack.
 - 2. x can bring P from p to q with empty stack $\Rightarrow A$ generates x .

- Proof: If A_{pq} generates x , then x can bring P from p with empty stack to q with empty stack.
 - Basis: $A_{pp} \rightarrow \epsilon$
 - Assume true for derivations of length at most k , where $k \geq 1$.
 - Suppose $A_{pq} \xrightarrow{*} x$ with $k+1$ steps. The first step is $A_{pq} \rightarrow aA_{rs}b$
or $A_{pq} \rightarrow A_{pr}A_{rq}$.
 - 1. $A_{pq} \rightarrow aA_{rs}b$: $A_{rs} \xrightarrow{*} y$ in k steps, $x = ayb$.
 - 2. $A_{pq} \rightarrow A_{pr}A_{rq}$: $A_{pr} \xrightarrow{*} y$, and $A_{rq} \xrightarrow{*} z$, $x = yz$.

- Proof: If x can bring P from p with empty stack to q with empty stack, then A_{pq} generates x .
 - Basis: the computation has 0 steps, $x = \varepsilon$, and the rule $A_{pp} \rightarrow \varepsilon$ in G .
 - Assume true for computations of length at most k , where $k \geq 0$.
 - Suppose that P has a computation wherein x brings p to q with empty stacks in $k+1$ steps.

► **1. The stack is empty only at the beginning and end.**

Let $x = ayb$, y can bring PDA P from r to s without touching the symbol t that is on the stack. Removed the first step and the last step of the $k+1$ steps_{*}, the computation of y has $k-1$ steps.

Thus $A_{rs} \rightarrow y$, hence $A_{pq} \rightarrow x$

► **2. The stack becomes empty elsewhere, too.**

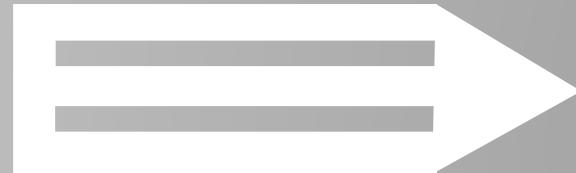
Let r be the state where the stack becomes empty, so the computation from p to r and from r to q each contain at most k steps. Let y be the input from p to r , z be the input from r to q . $A_{pr} \rightarrow y$, $A_{rq} \rightarrow z$ and $x = yz$. Because the rule $A_{pq} \rightarrow A_{pr}A_{rq}$ is in G , $A_{pq} \rightarrow x$.

End prove

Theory of Computation

Lesson 6-1

Non-Context-Free Languages



王 轩
Wang
Xuan

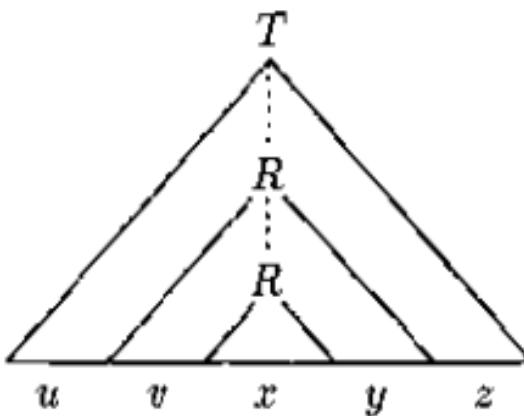
- Pumping lemma for CFG
- Applications of The Pumping Lemma

Pumping lemma for CFG

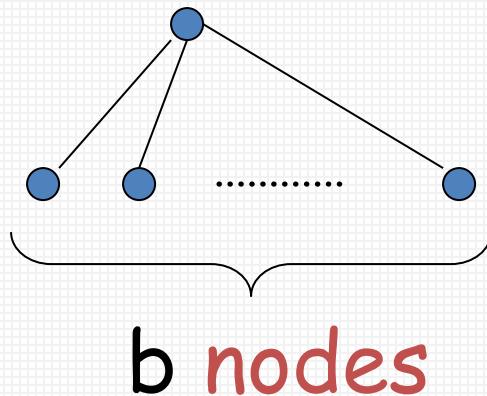
If A is a context-free language, then there is a number p (the pumping length) where, if s is any string in A of length at least p , then s may be divided into five pieces $s=uvxyz$ satisfying the conditions

1. For each $i \geq 0$, $uv^i xy^i z \in A$
2. $|vy| > 0$, and
3. $|vxy| \leq p$

- Let A be a CFL and let G be a CFG that generates it
- Let s be a very long string in A , there will be a parse tree.
- On this long path some variable symbol R must repeat because of the pigeonhole principle.



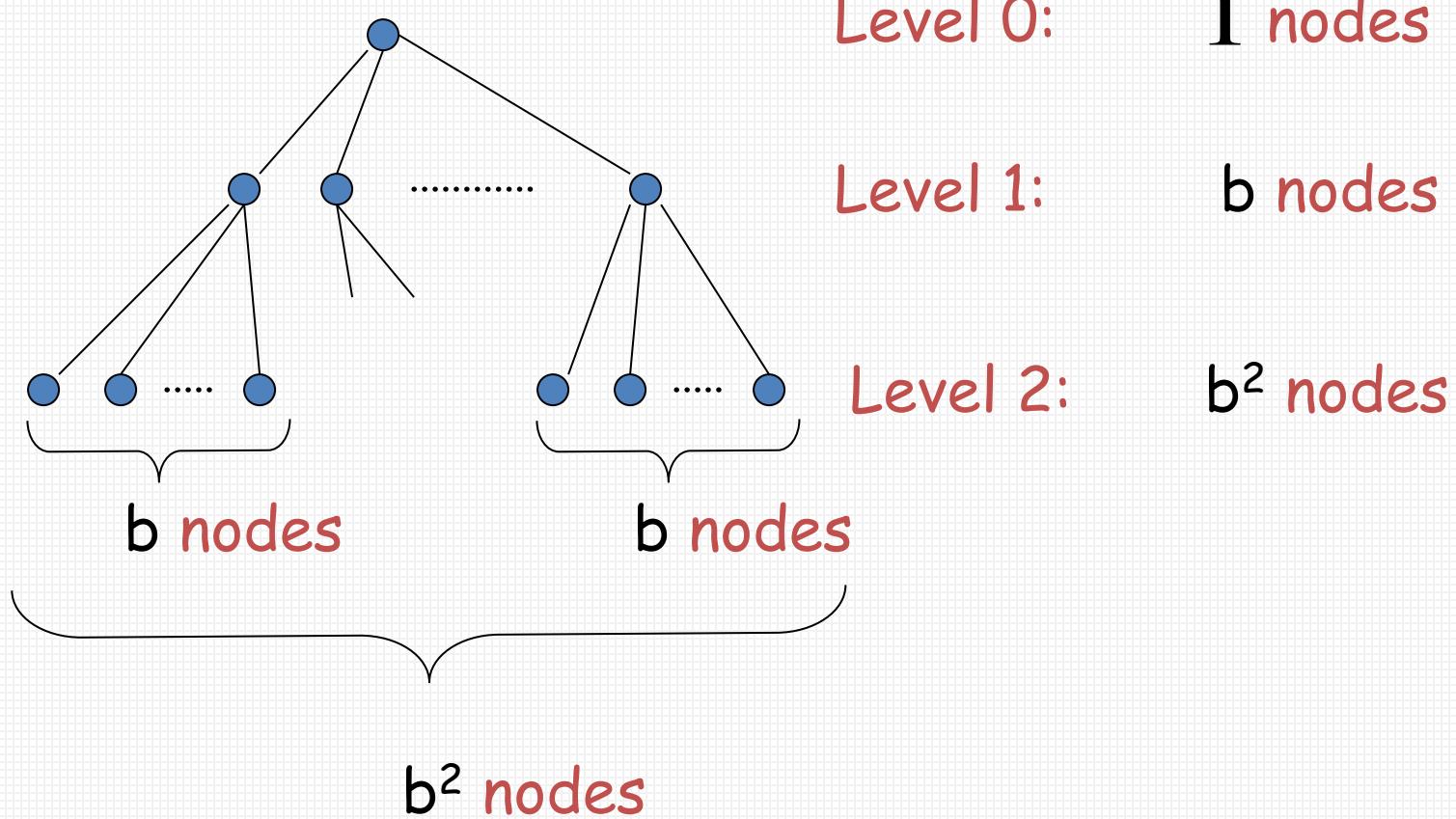
- Let b be the maximum number of symbols on the right-hand side of a rule



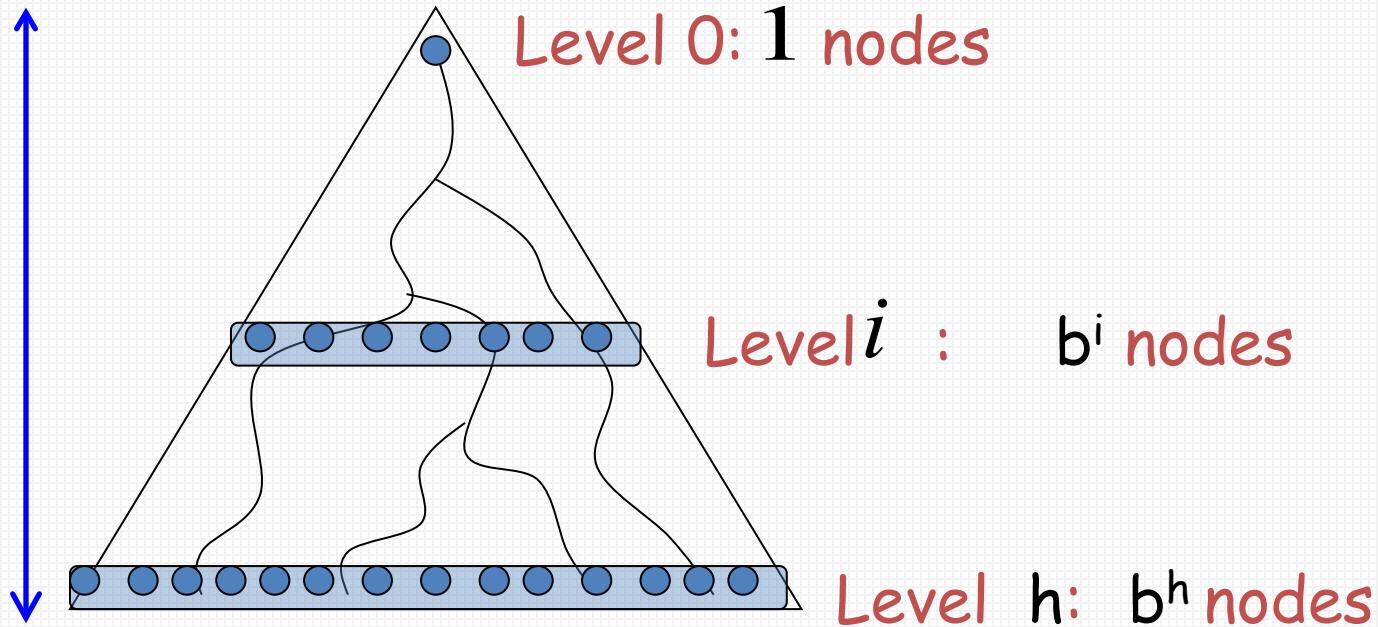
Level 0: 1 nodes

Level 1: b nodes

Maximum number of nodes per level



Maximum number of nodes per level



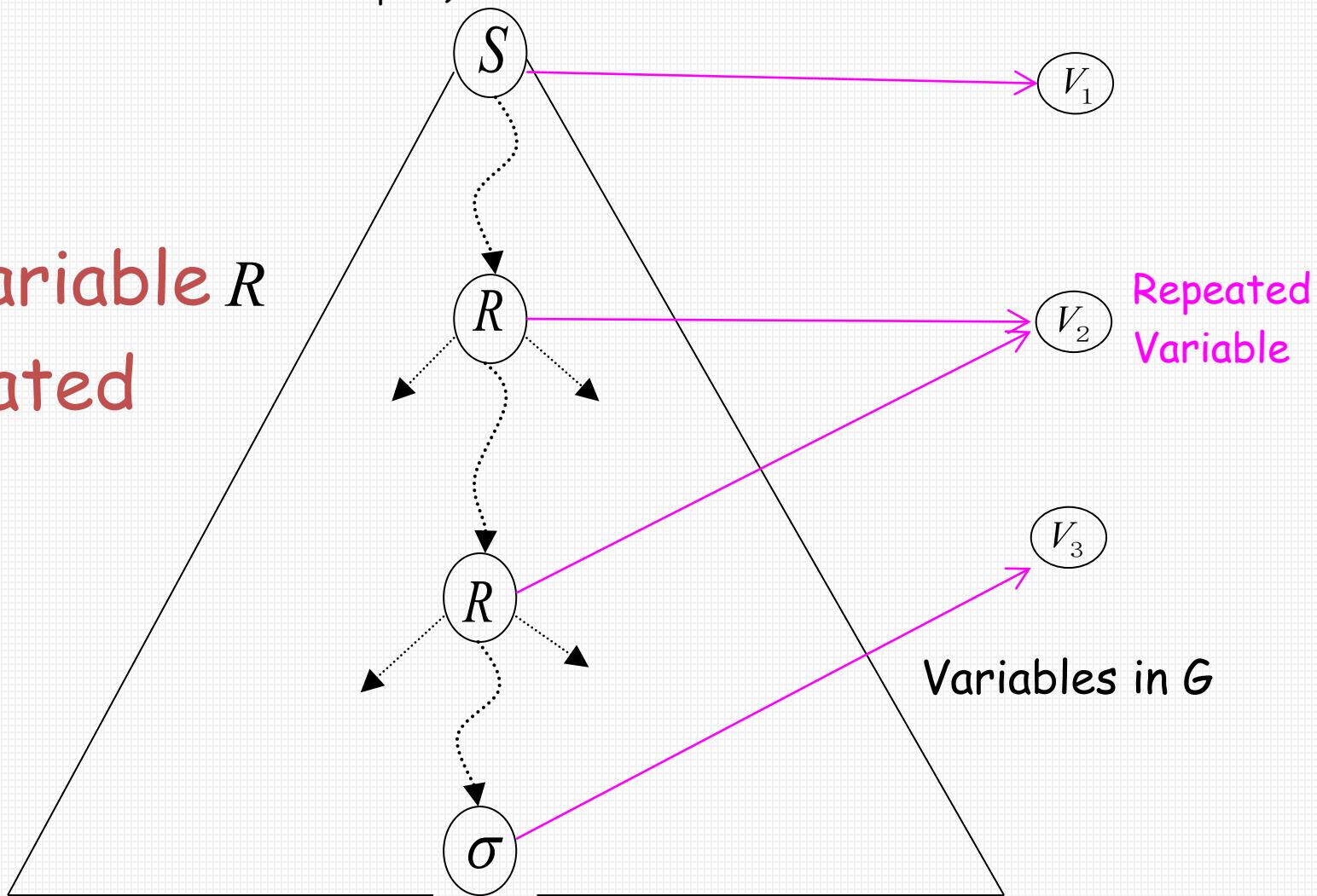
If a string is at least b^h+1 long, each of its Parse tree must be at least $h+1$ high

- Set $|V|$ is the number of variables in G .
- We set pumping length p to be $b^{|V|+1} > b^{|V|+1}$, Thus if the length string s is p or more, its parse tree must be at least $|V|+1$ high.
- Choose T is a parse tree that has the **smallest** number of nodes. Its longest path from root to leaf has length at least $|V|+1$. The path has at least $|V|+2$ nodes, at least $|V|+1$ variables.
- G have only $|V|$ variables, thus, some variable R appears more than once on that path.

Pigeons:
(Variables in the path)

Pigeonholes :
(Variables in G)

some variable R
is repeated

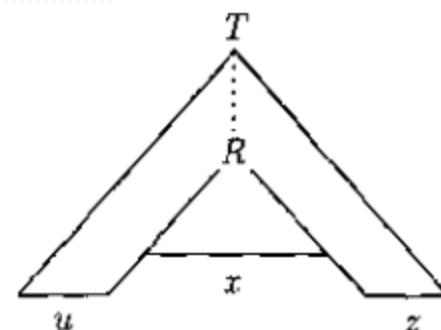
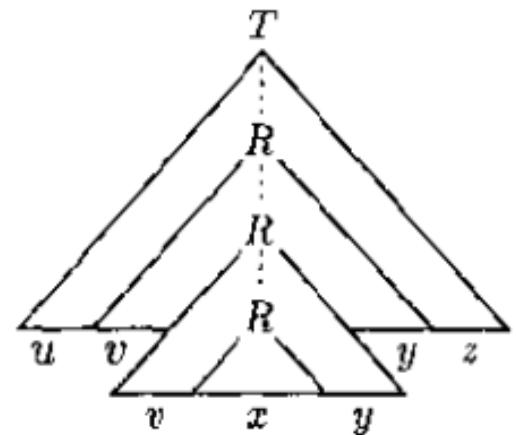
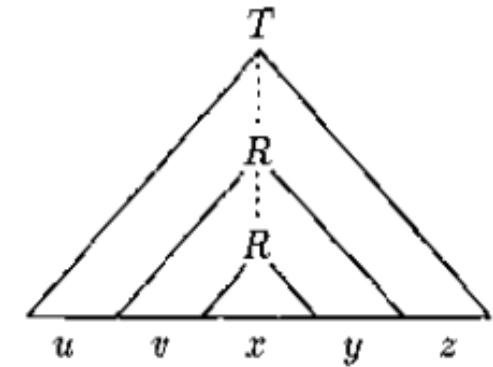


Repeated Variable
Variables in G

Derivation tree of W

- We divide s into $uvxyz$
- The occurrence of R generates vxy (*larger subtree*), x (*smaller subtree*)
- Thus, repeat R with vxy is still a valid tree, condition 1 is satisfied.

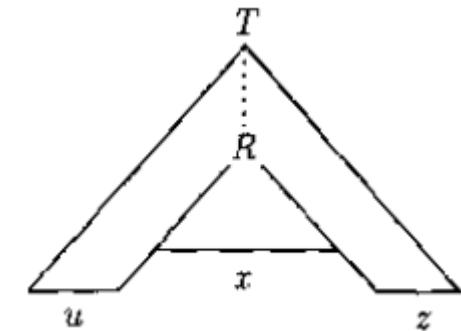
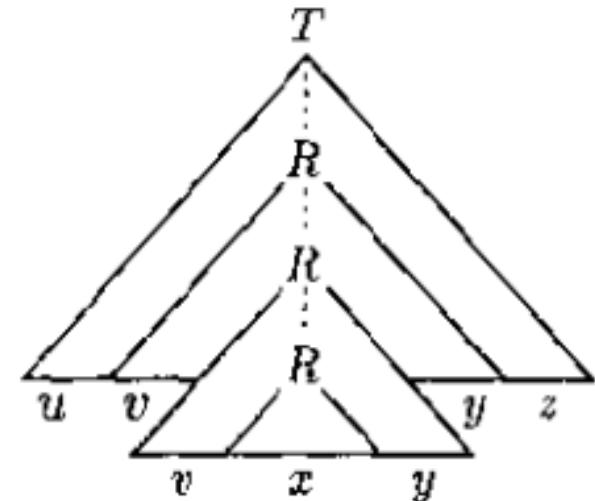
(condition 3) For each $i \geq 0$,
 $uv^i xy^i z \in A$



► (condition 2) v and y are not both ϵ . Otherwise, it generate a parse tree have fewer nodes than T .
 $|vy| > 0$

► (condition 3) We choose R that both occurrence of R within bottom $|V|+1$ variables.

► R generate vxy . So the subtree R generate vxy at most $|V|+1$ high. The length of string is less than $b^{|V|+1} = p$ $|vxy| \leq p$



Applications of The Pumping Lemma

Non-context free languages

$$\{a^n b^n c^n : n \geq 0\}$$

Context-free languages

$$\{a^n b^n : n \geq 0\}$$

Theorem: The language

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

is not context free

Proof: Use the Pumping Lemma
for context-free languages

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

Assume for contradiction that L is context-free

Since L is context-free and infinite we can apply the pumping lemma

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

Let m be the critical length
of the pumping lemma

Pick any string $w \in L$ with length $|w| \geq m$

We pick: $w = a^m b^m c^m$

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

$$w = a^m b^m c^m$$

From pumping lemma:

we can write: $w = uvxyz$

with lengths $|vxy| \leq m$ and $|vy| > 0$

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

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| > 0$$

Pumping Lemma says:

$$uv^i xy^i z \in L \quad \text{for all } i \geq 0$$

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

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| > 0$$

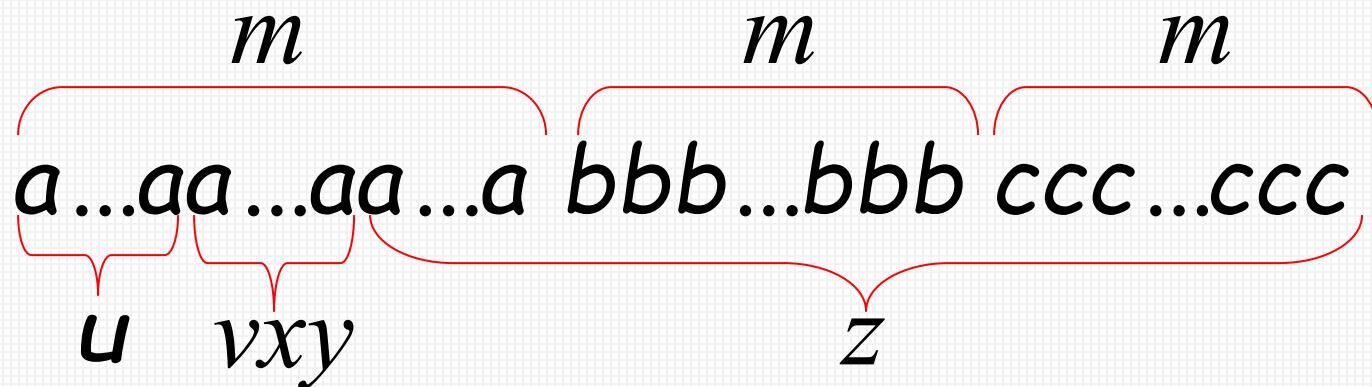
We examine all the possible locations
of string vxy in w

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

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| > 0$$

Case 1: vxy is in a^m



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

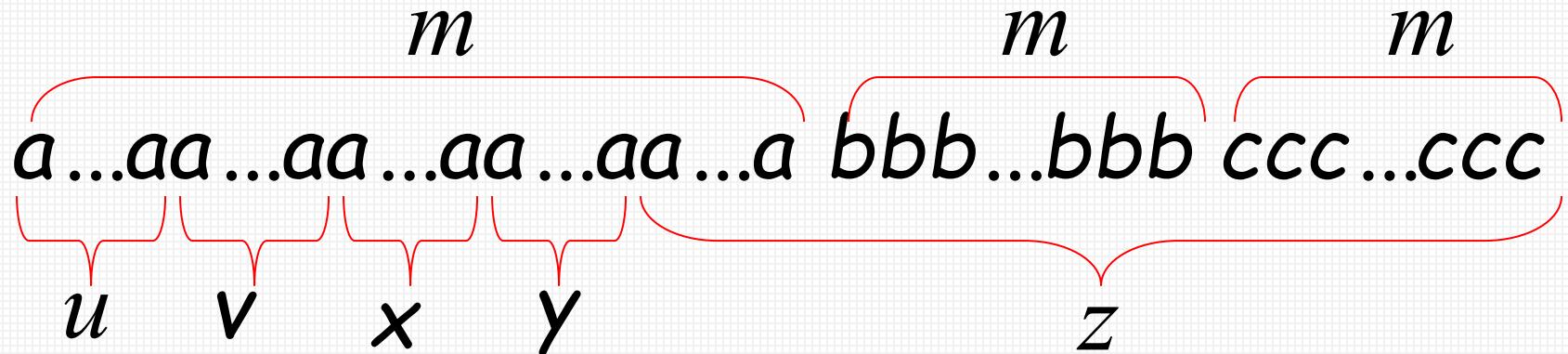
$$w = a^m b^m c^m$$

$$w = uvxyz$$

$$|vxy| \leq m$$

$$|vy| > 0$$

$$v = a^{k_1} \quad y = a^{k_2} \quad k_1 + k_2 \geq 1$$

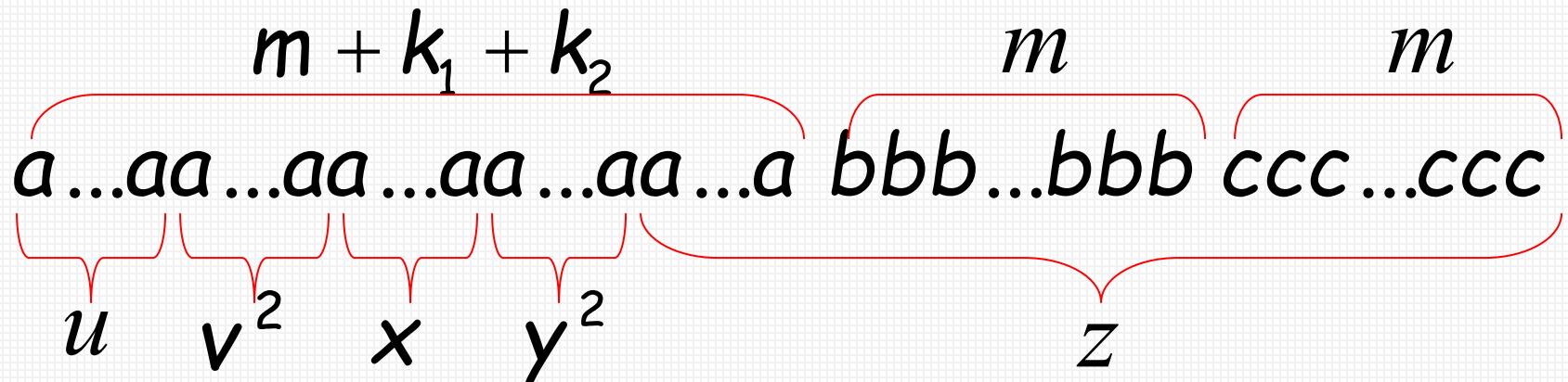


$$L = \{a^n b^n c^n : n \geq 0\}$$

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| > 0$$

$$v = a^{k_1} \quad y = a^{k_2} \quad k_1 + k_2 \geq 1$$



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

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| > 0$$

From Pumping Lemma: $uv^2xy^2z \in L$

$$k_1 + k_2 \geq 1$$

However: $uv^2xy^2z = a^{m+k_1+k_2}b^m c^m \notin L$

Contradiction!!!

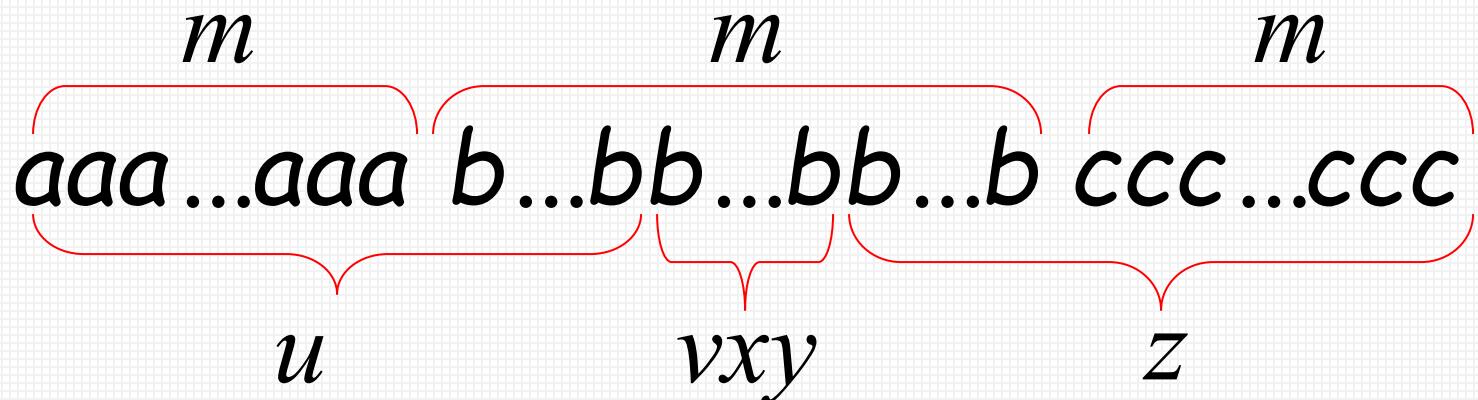
$$L = \{a^n b^n c^n : n \geq 0\}$$

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| > 0$$

Case 2: vxy is in b^m

Similar to case 1



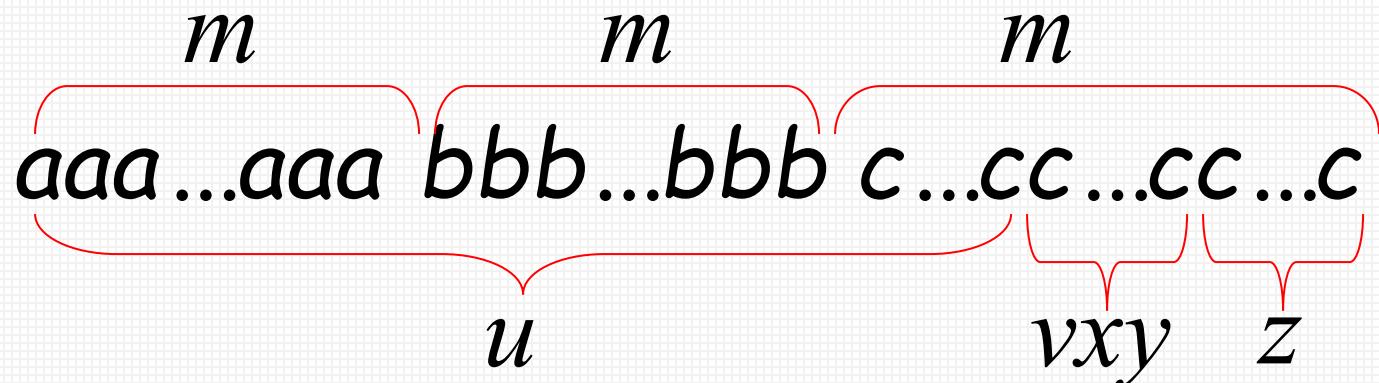
$$L = \{a^n b^n c^n : n \geq 0\}$$

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| > 0$$

Case 3: vxy is in c^m

Similar to case 1

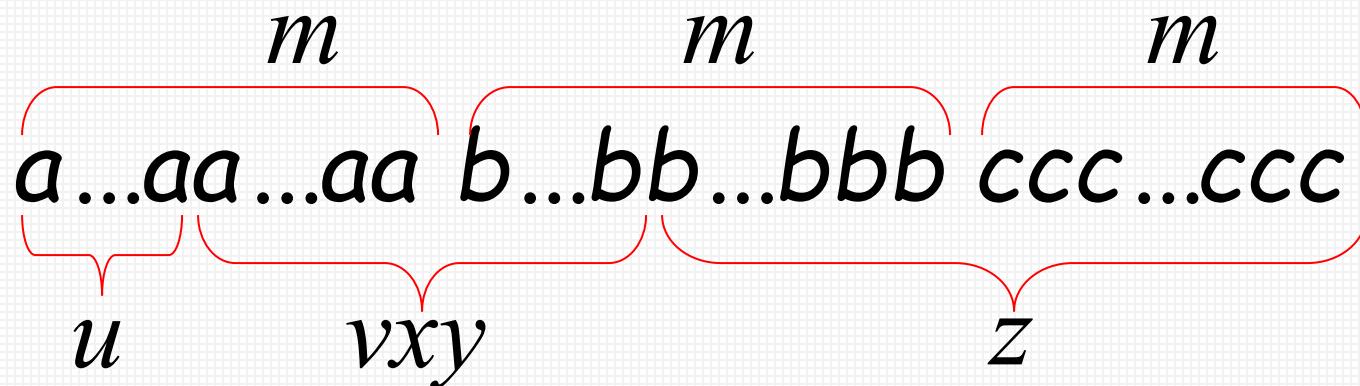


$$L = \{a^n b^n c^n : n \geq 0\}$$

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| > 0$$

Case 4: vxy overlaps a^m and b^m

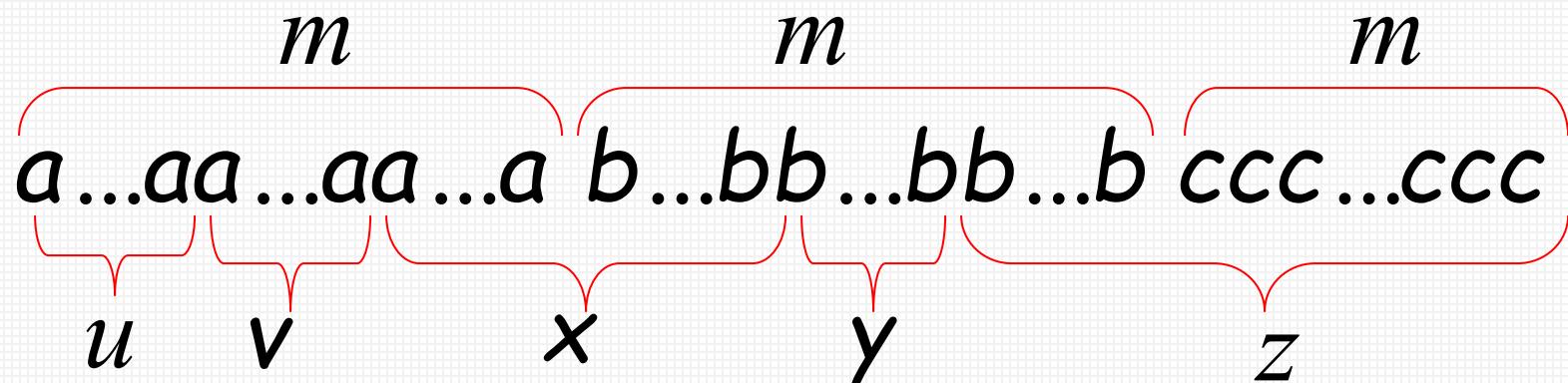


$$L = \{a^n b^n c^n : n \geq 0\}$$

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| > 0$$

Sub-case 1: v contains only a
 y contains only b



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

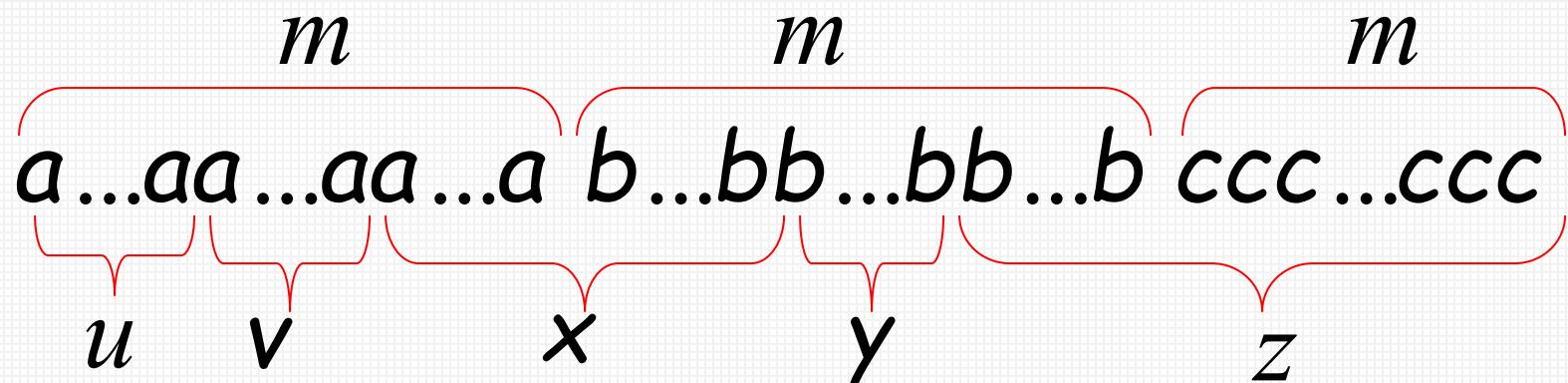
$$w = a^m b^m c^m$$

$$w = uvxyz$$

$$|vxy| \leq m$$

$$|vy| > 0$$

$$v = a^{k_1} \quad y = b^{k_2} \quad k_1 + k_2 \geq 1$$

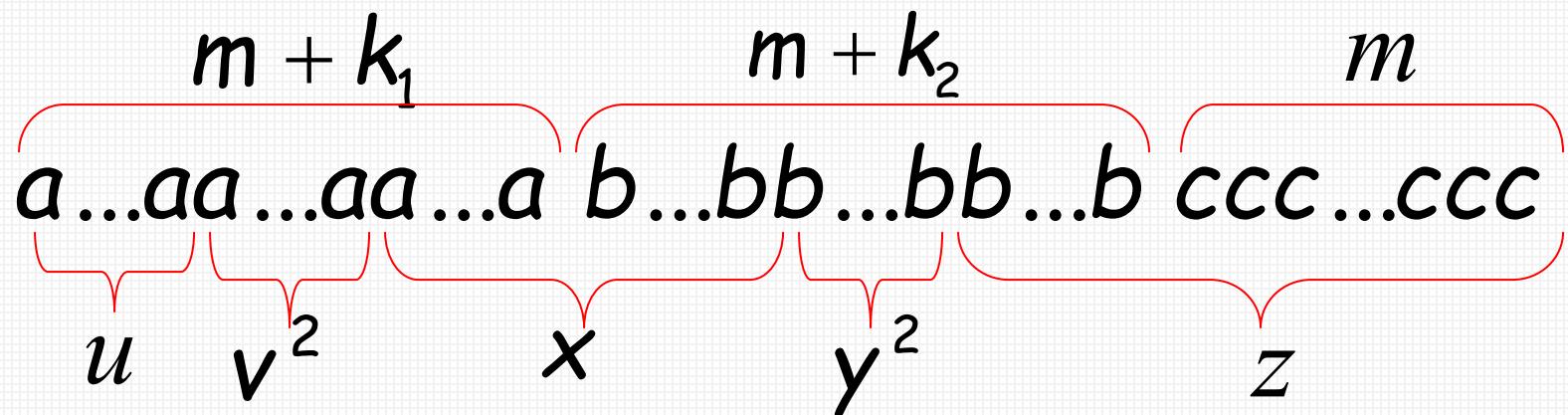


$$L = \{a^n b^n c^n : n \geq 0\}$$

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| > 0$$

$$v = a^{k_1} \quad y = b^{k_2} \quad k_1 + k_2 \geq 1$$



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

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| > 0$$

From Pumping Lemma: $uv^2xy^2z \in L$

$$k_1 + k_2 \geq 1$$

However: $uv^2xy^2z = a^{m+k_1}b^{m+k_2}c^m \notin L$

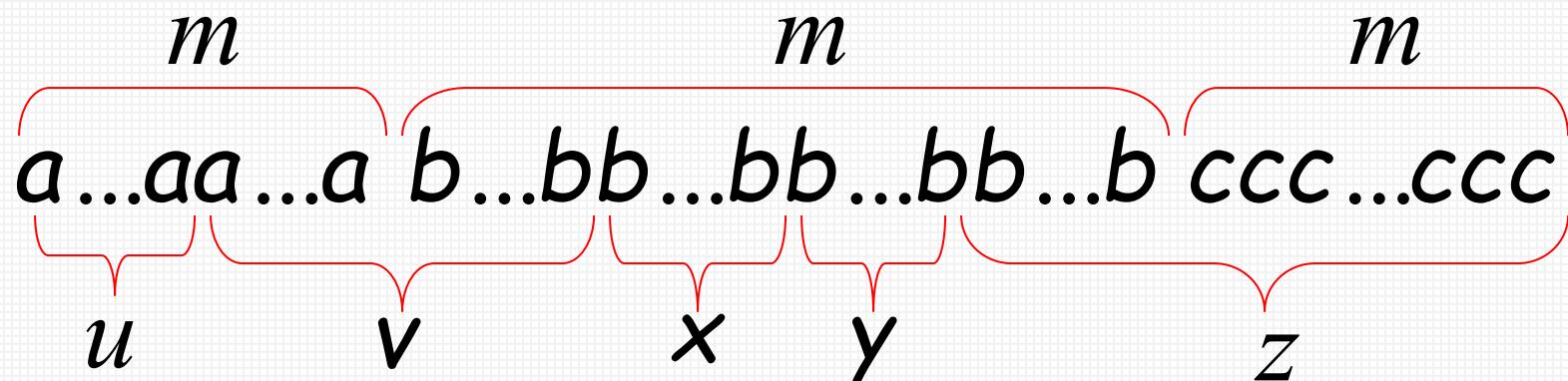
Contradiction!!!

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

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| > 0$$

Sub-case 2: v contains a and b
 y contains only b



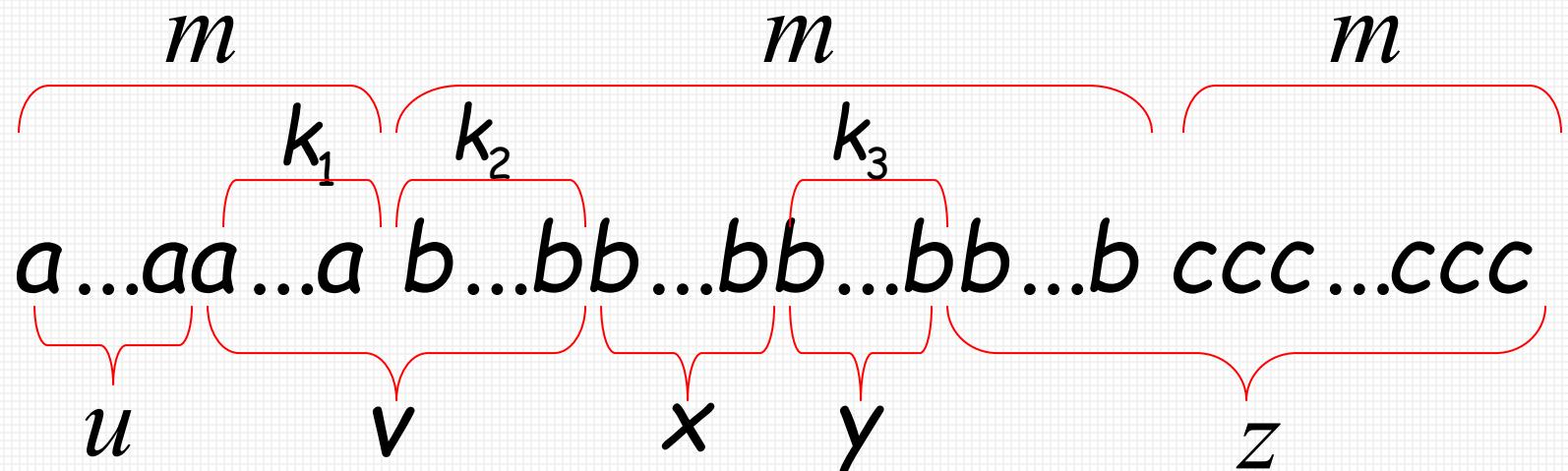
$$L = \{a^n b^n c^n : n \geq 0\}$$

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| > 0$$

By assumption

$$v = a^{k_1} b^{k_2} \quad y = b^{k_3} \quad k_1, k_2 \geq 1$$

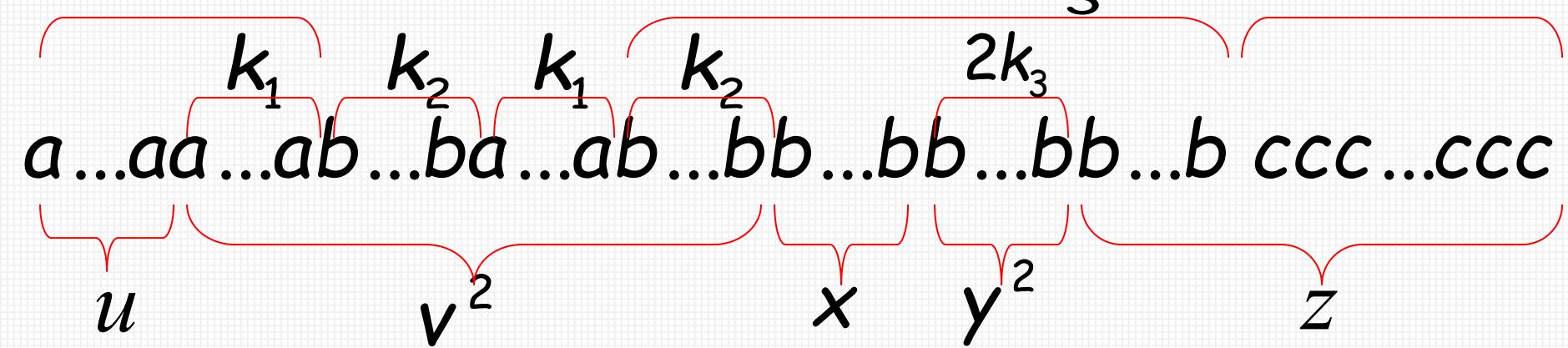


$$L = \{a^n b^n c^n : n \geq 0\}$$

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| > 0$$

$$\begin{array}{ccc} v = a^{k_1} b^{k_2} & y = b^{k_3} & k_1, k_2 \geq 1 \\ m & m + k_3 & m \end{array}$$



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

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| > 0$$

From Pumping Lemma: $uv^2xy^2z \in L$

$$k_1, k_2 \geq 1$$

However: $uv^2xy^2z = a^m b^{k_2} a^{k_1} b^{m+k_3} c^m \notin L$

Contradiction!!!

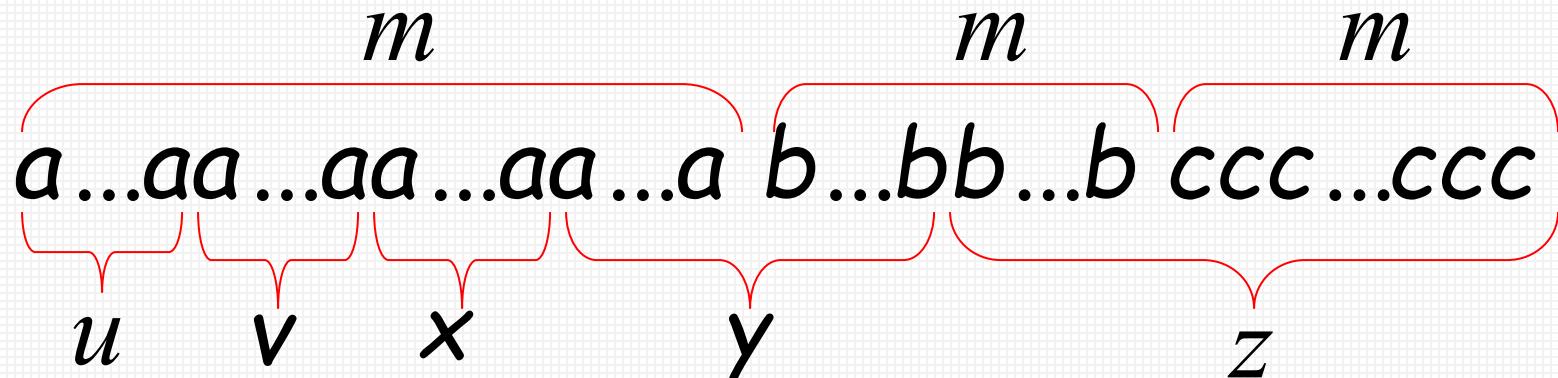
$$L = \{a^n b^n c^n : n \geq 0\}$$

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| > 0$$

Sub-case 3: v contains only a
 y contains a and b

Similar to sub-case 2



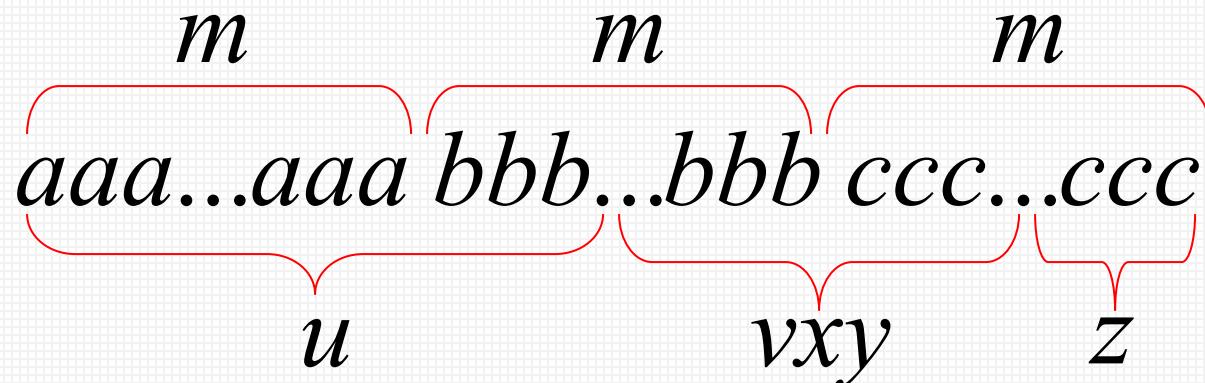
$$L = \{a^n b^n c^n : n \geq 0\}$$

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| > 0$$

Case 5: vxy overlaps b^m and c^m

Similar to case 4



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

$$w = a^m b^m c^m$$

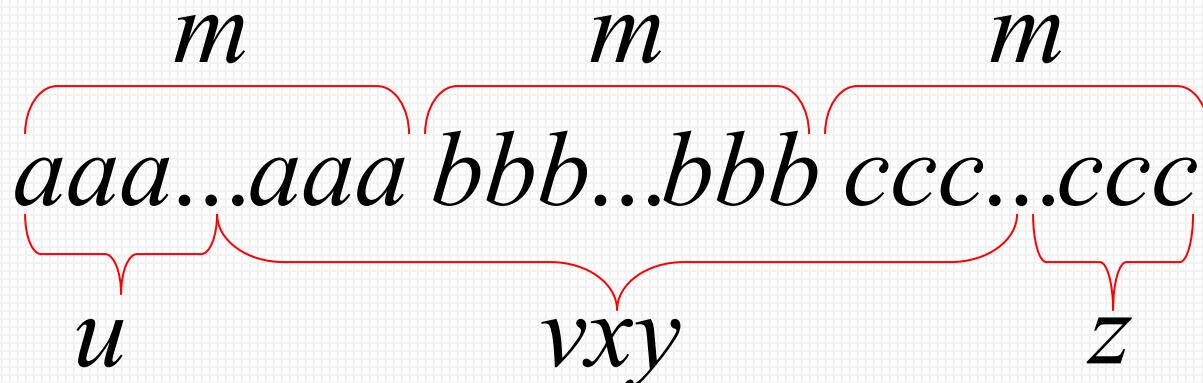
$$w = uvxyz$$

$$|vxy| \leq m$$

$$|vy| > 0$$

Case 6: vxy overlaps a^m , b^m and c^m

Impossible!



In all cases we obtained a contradiction

Therefore: the original assumption that

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

is context-free must be wrong

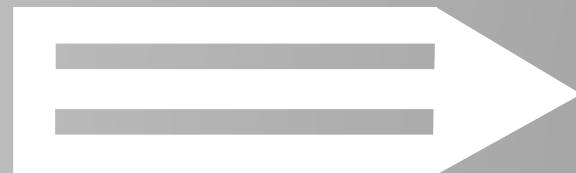
Conclusion: L is not context-free

- ✓ Pumping lemma for CFG
- ✓ Applications of The Pumping Lemma

Theory of Computation

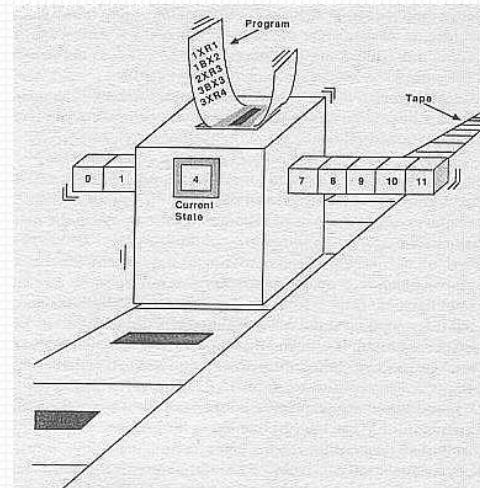
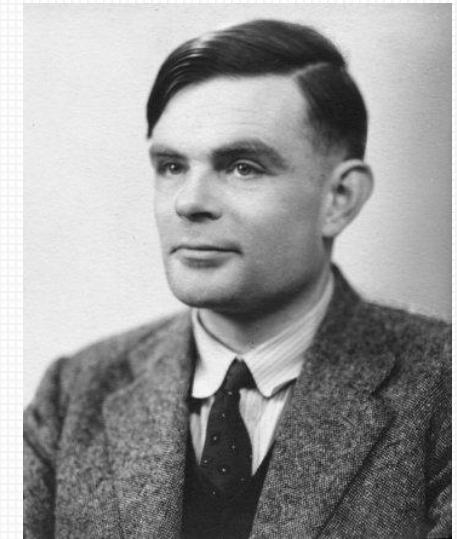
Lesson 6-2

Turing Machines



王轩
Wang
Xuan

- ▶ The Language Hierarchy
- ▶ Automata-Recognizable language Table
- ▶ Turing Machine
- ▶ Halting & Accepting
- ▶ Formal Definitions for Turing Machines
- ▶ Turing Recognizable
- ▶ Turing-decidable
- ▶ Examples



The Language Hierarchy

$a^n b^n c^n$?

ww^R ?

Context-Free Languages

$a^n b^n$

ww^R

Regular Languages

a^*

$a^* b^*$

Languages accepted by Turing Machines

$a^n b^n c^n$

ww

Context-Free Languages

$a^n b^n$

ww^R

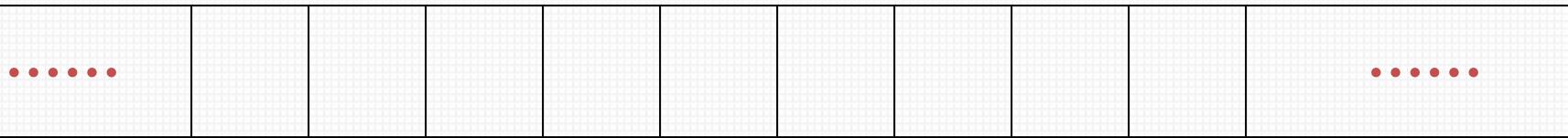
Regular Languages

a^*

$a^* b^*$

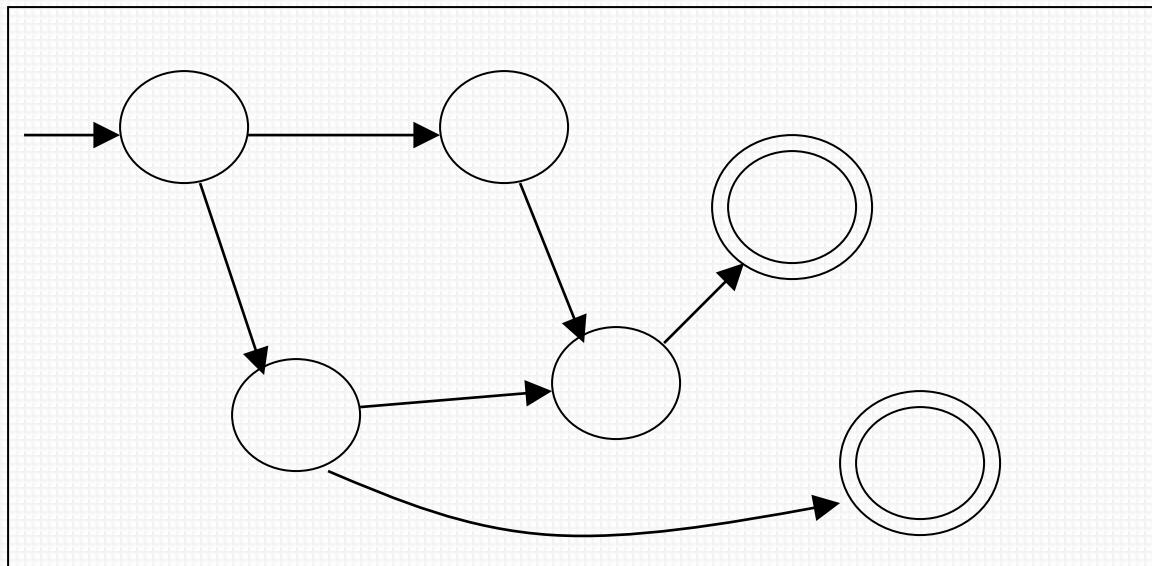
Automata	Recognizable language
Deterministic finite automata (DFA)	regular languages
Nondeterministic finite automata (NFA)	regular languages
Regular languages Pushdown automata (PDA)	context-free languages
Linear bounded automata (LBA)	context-sensitive language
Turing machines	recursively enumerable languages

A Turing Machine



Tape
Control Unit

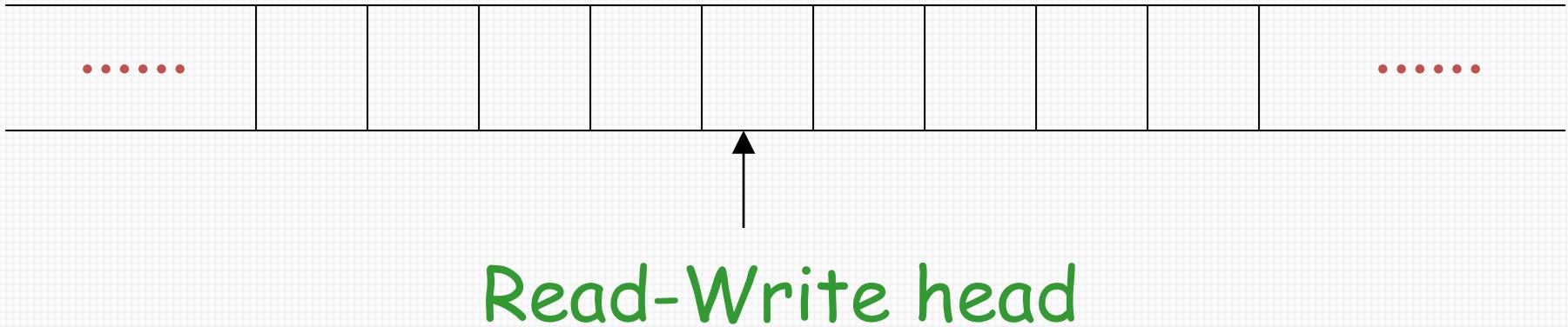
Read-Write head



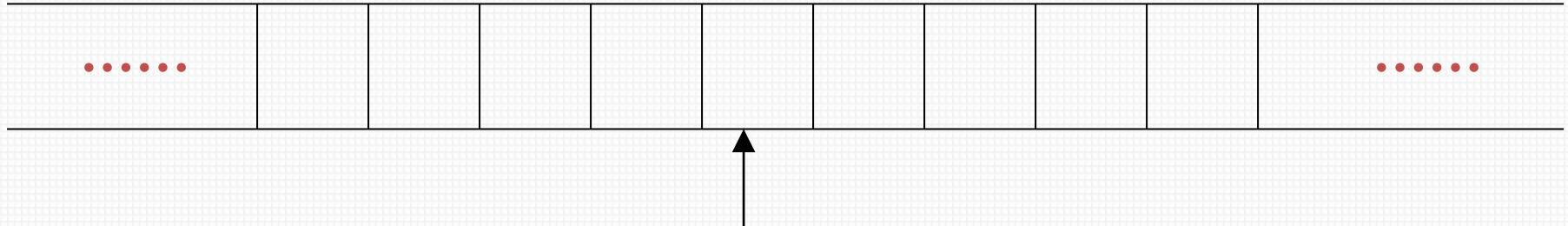
Difference Between FA & TM

- A Turing machine can both **write** on the tape and **read** from it
- The read-write head can move both to the left and to the right
- The tape is **infinite**
- The special states for **rejecting** and **accepting** take effect immediately

No boundaries -- infinite length



The head moves Left or Right



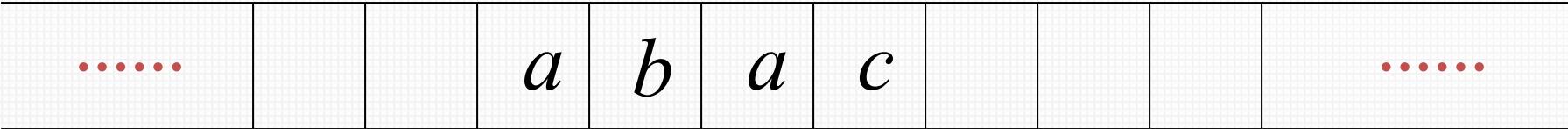
Read-Write head

The head at each transition (time step):

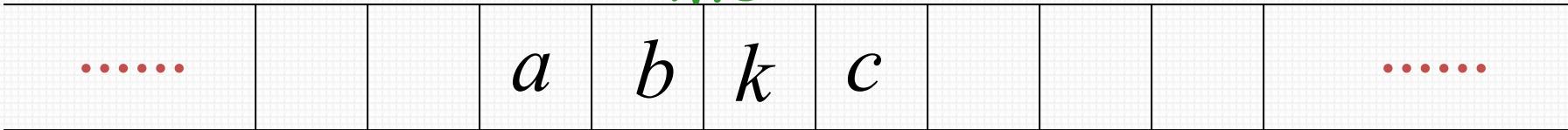
1. Reads a symbol
2. Writes a symbol
3. Moves Left or Right

Example:

Time 0



Time 1

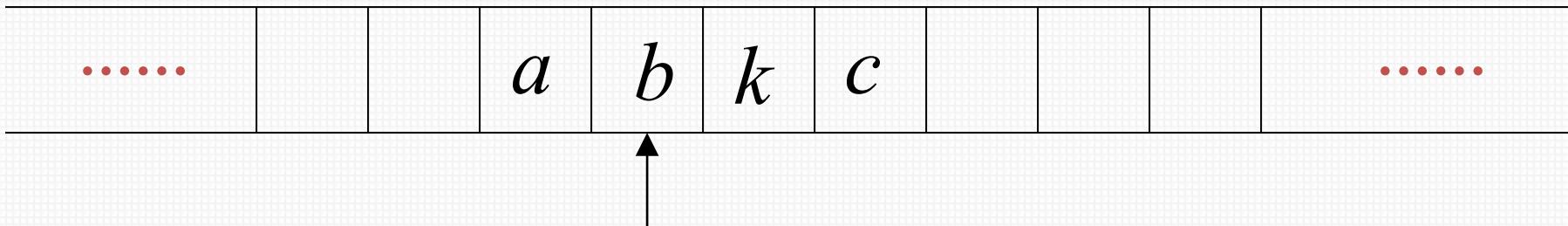


1. Reads a

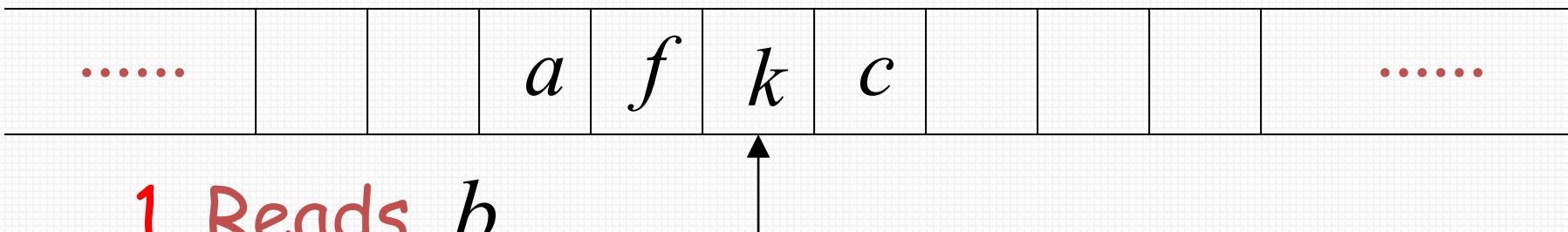
2. Writes k

3. Moves Left

Time 1



Time 2

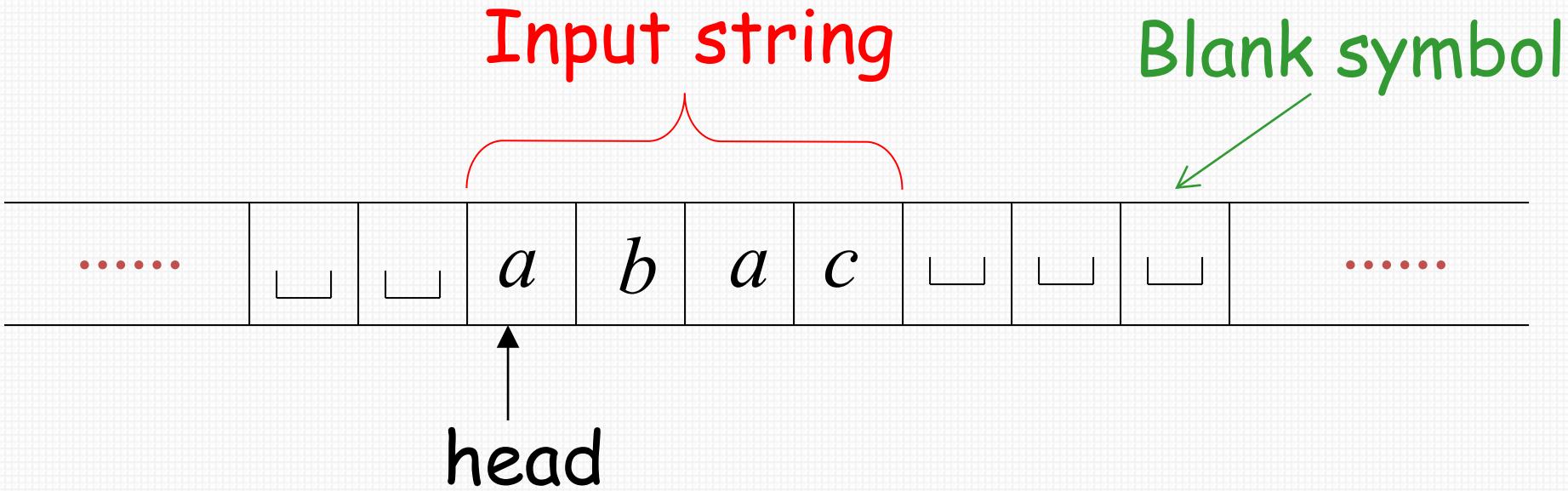


1. Reads b

2. Writes f

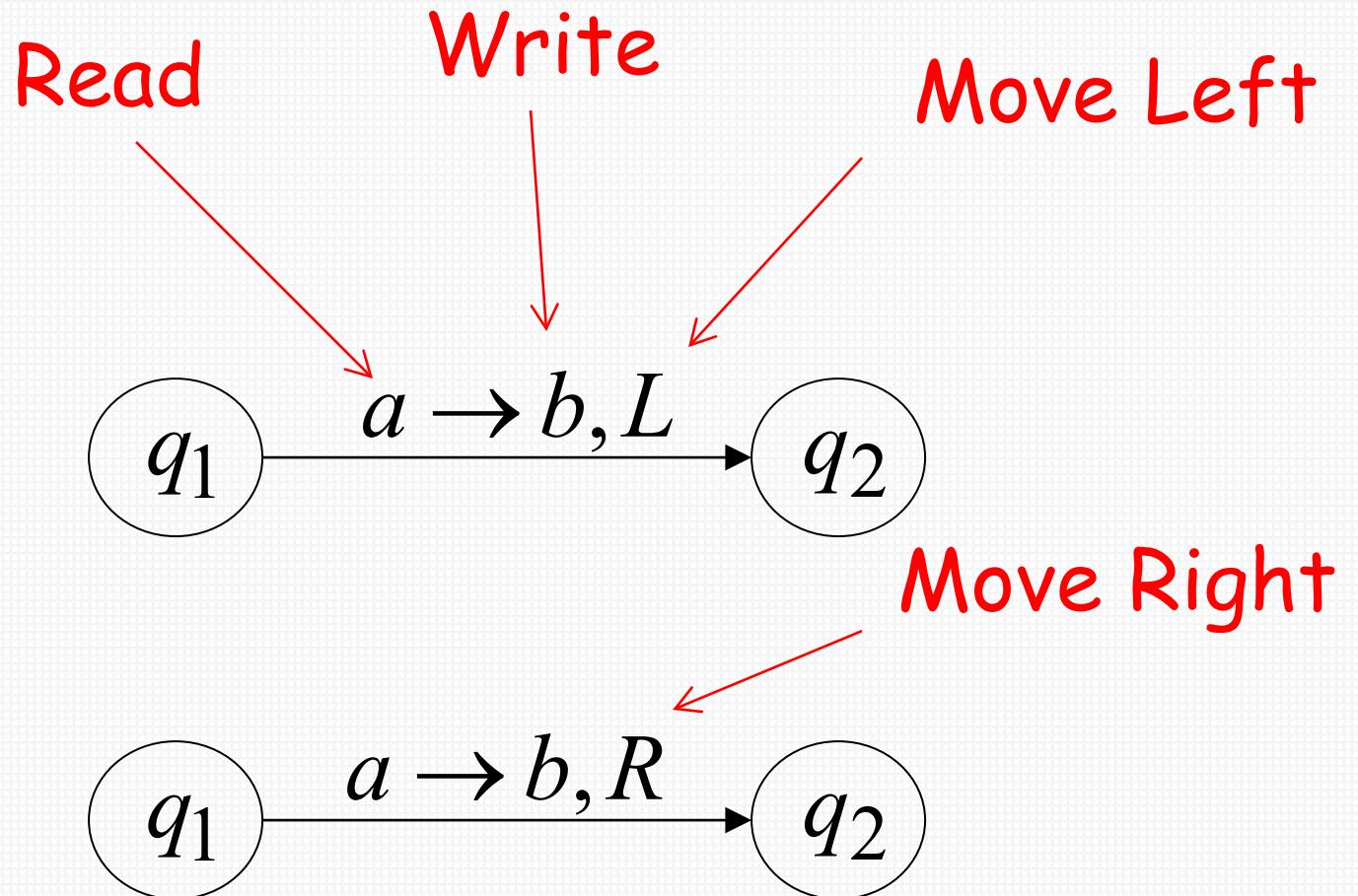
3. Moves Right

The Input String



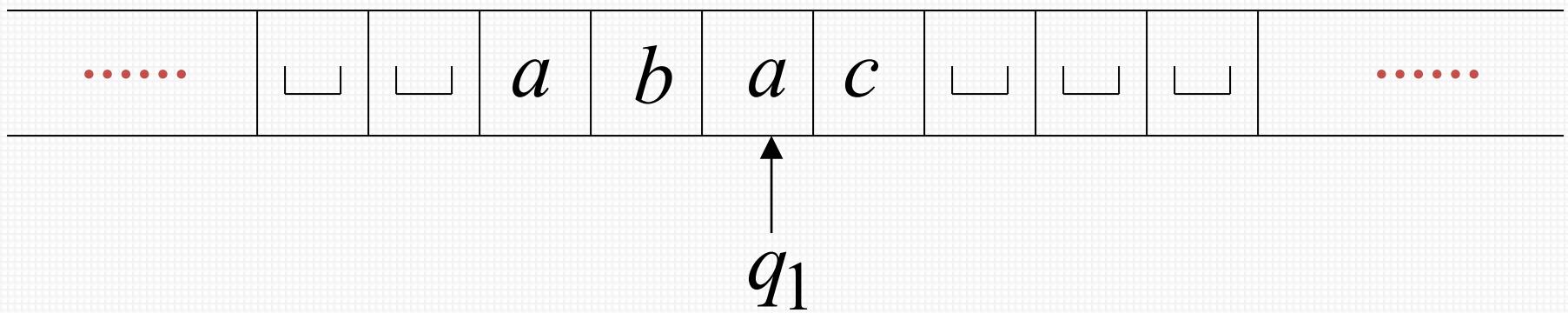
Head starts at the leftmost position
of the input string

States & Transitions

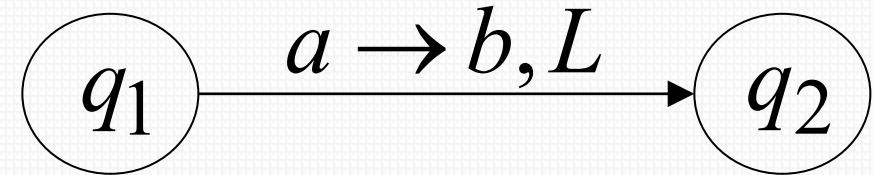
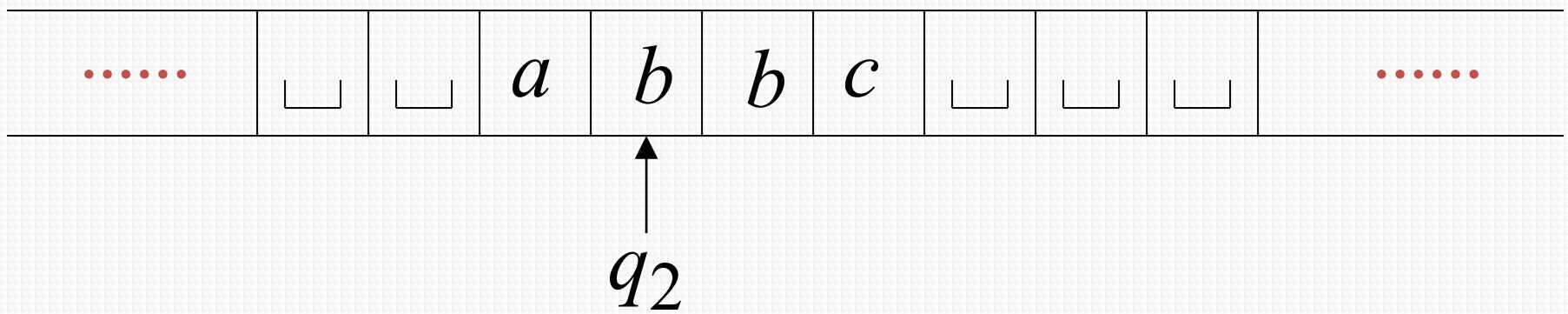


Example:

Time 1

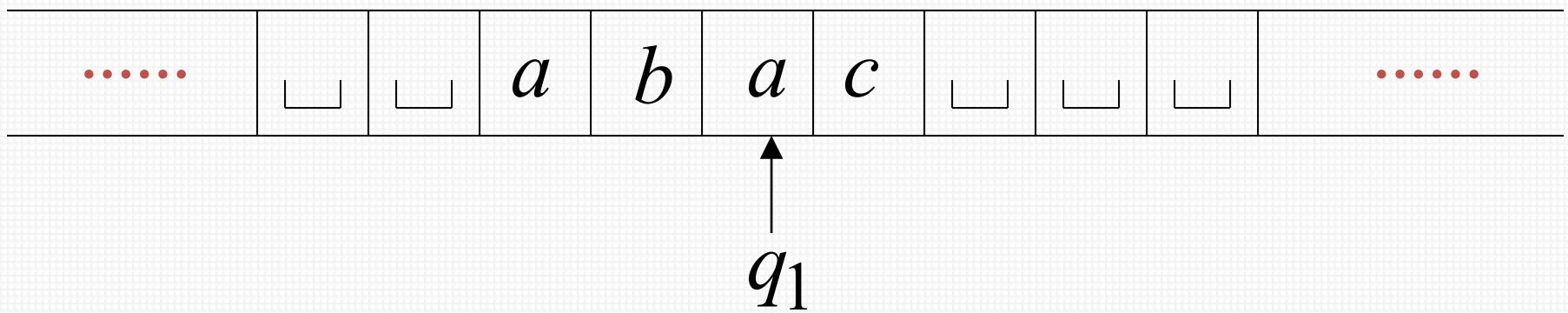


Time 2

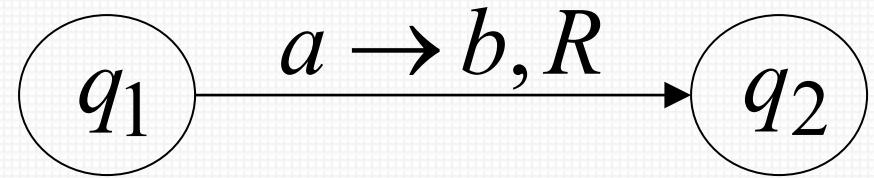
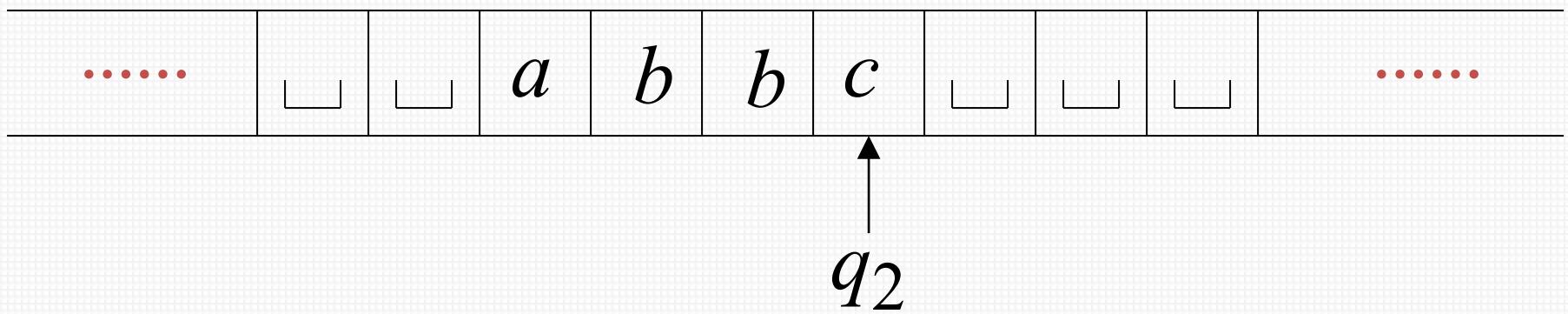


Example:

Time 1



Time 2

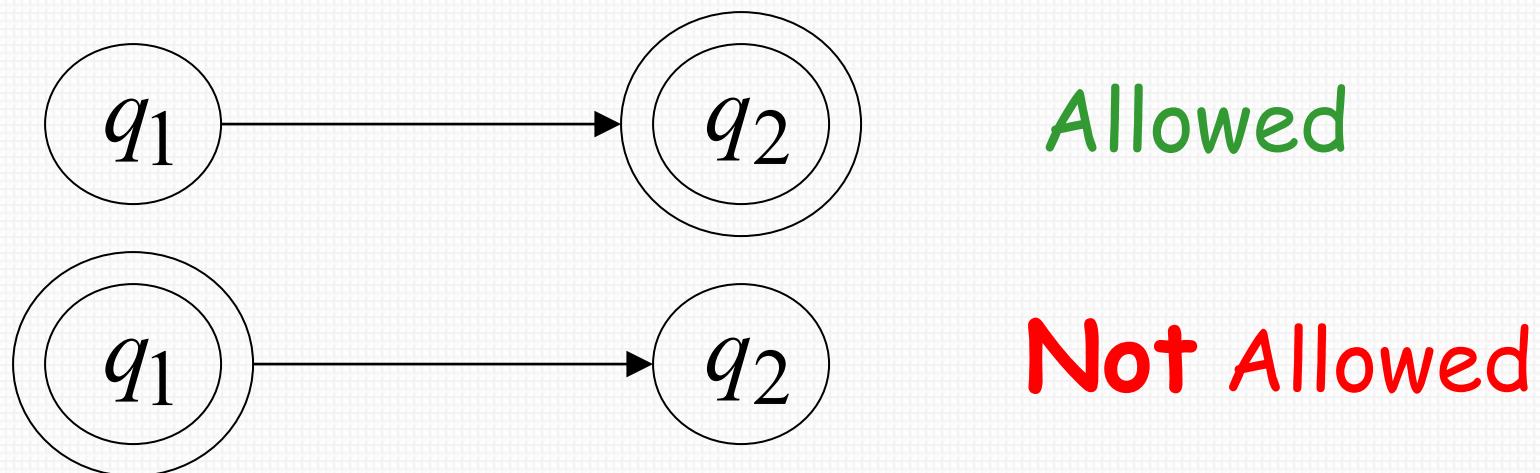


Halting

The outputs **accept** and **reject** are obtained by entering designated accepting and rejecting. If it doesn't, it will go on forever, never halting.

The machine **halts** in a state if there is no transition to follow

Accepting States



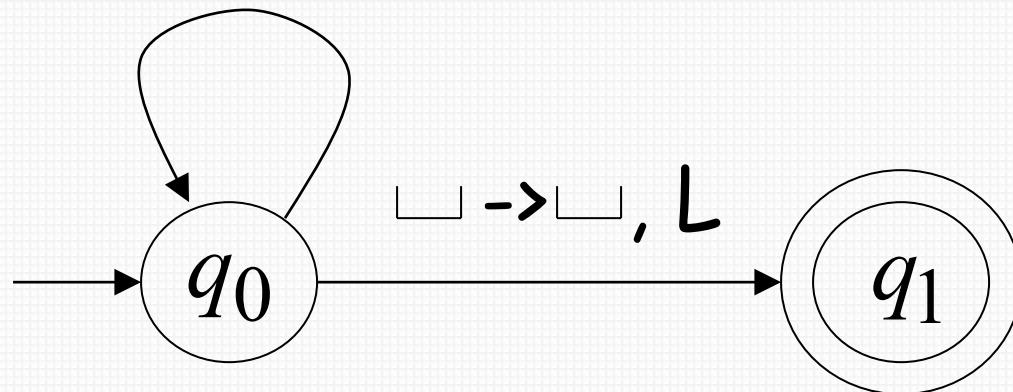
- Accepting states have no outgoing transitions
- The machine halts and accepts

Turing Machine Example

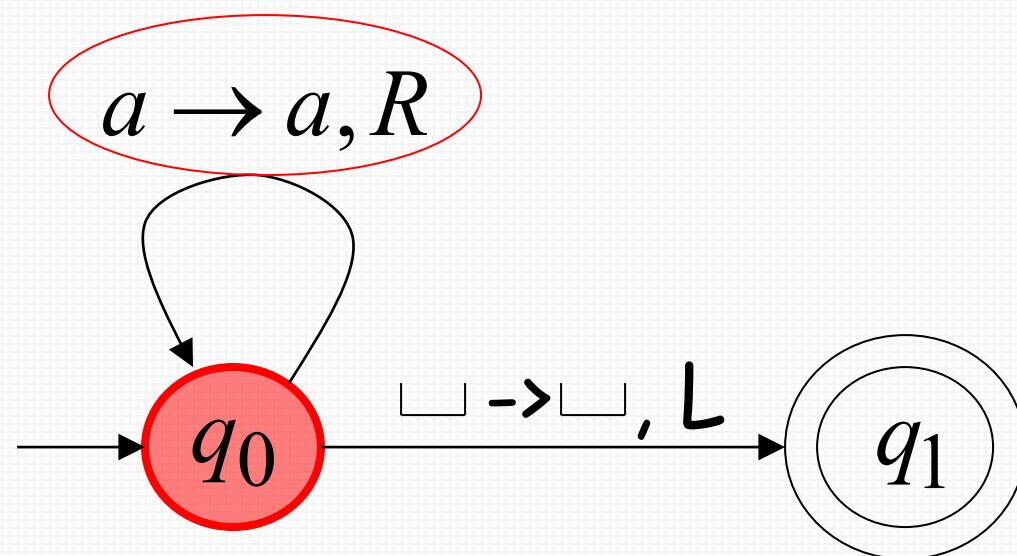
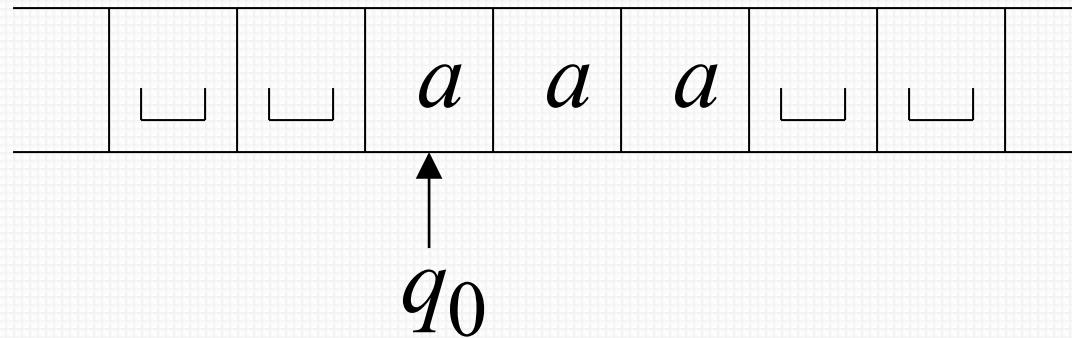
Input alphabet $\Sigma = \{a, b\}$

Accepts the language: a^*

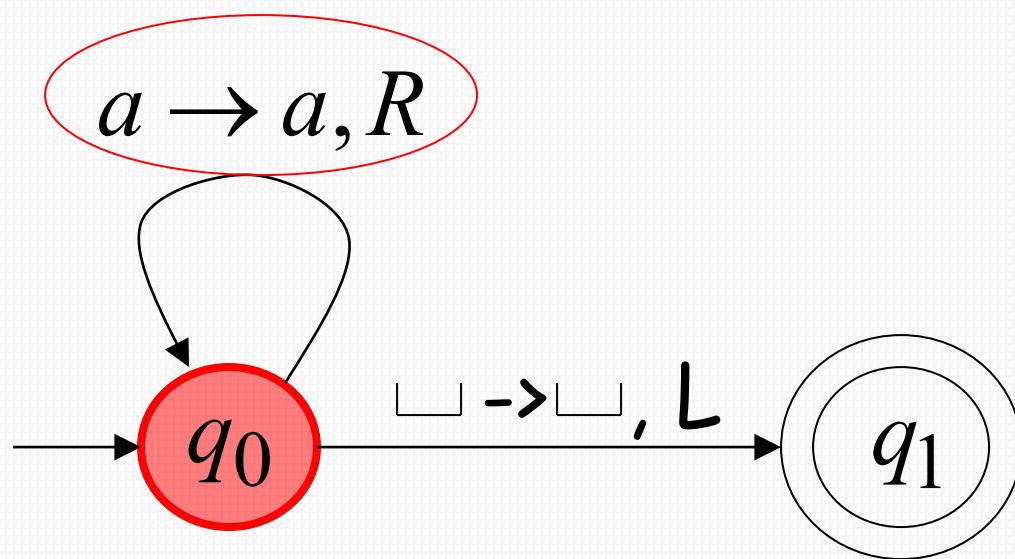
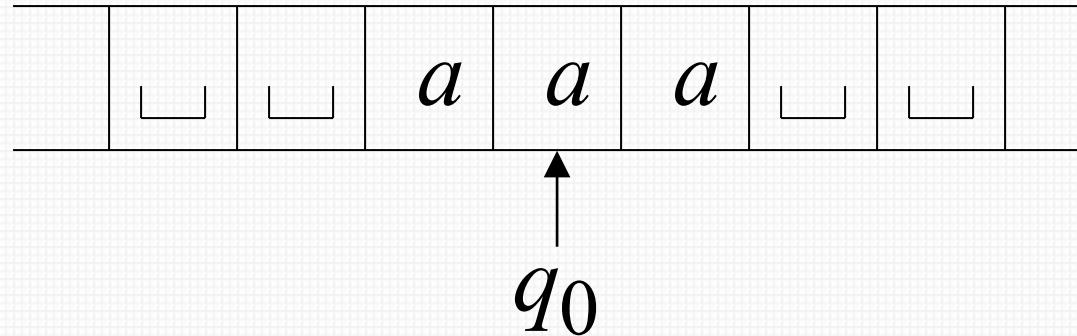
$$a \rightarrow a, R$$



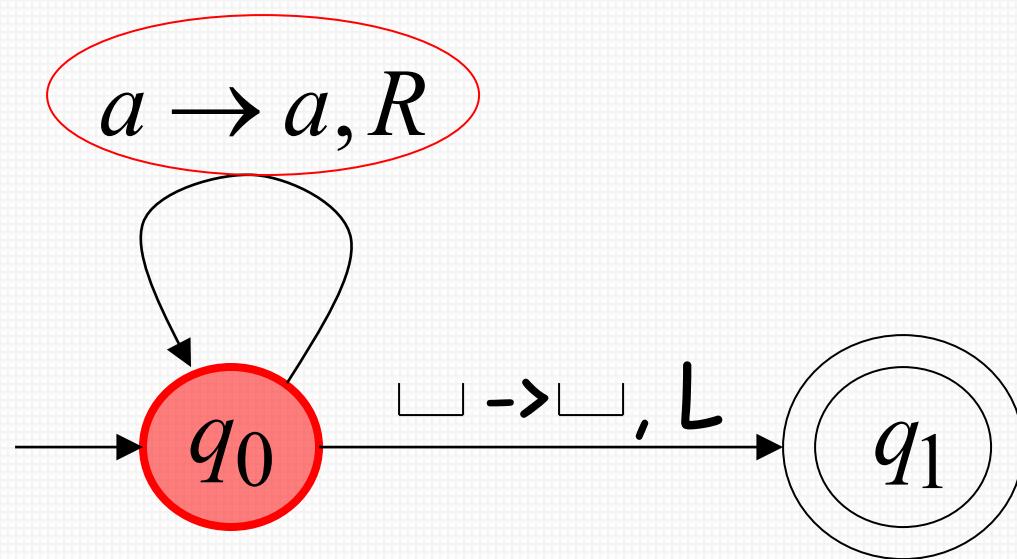
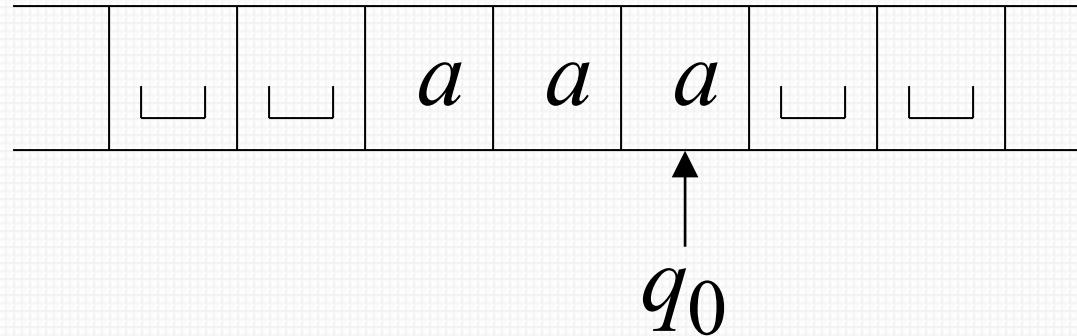
Time 0



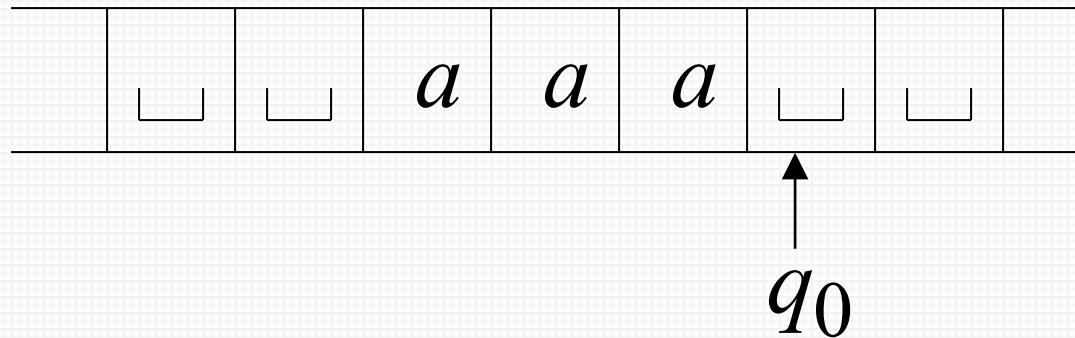
Time 1



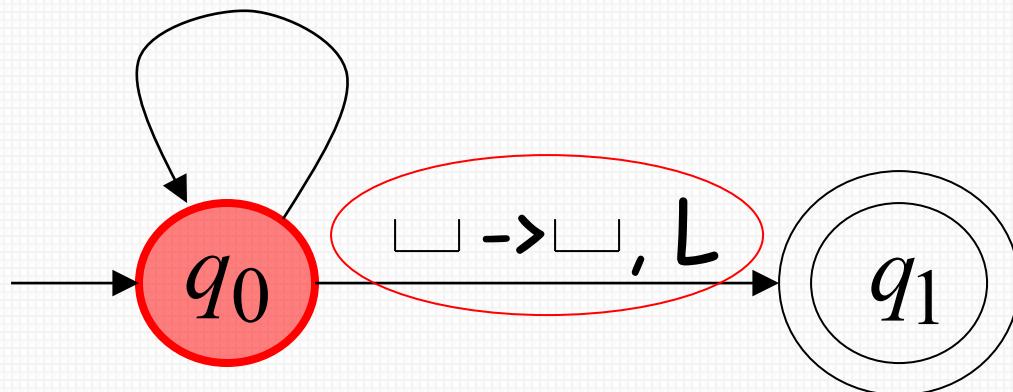
Time 2



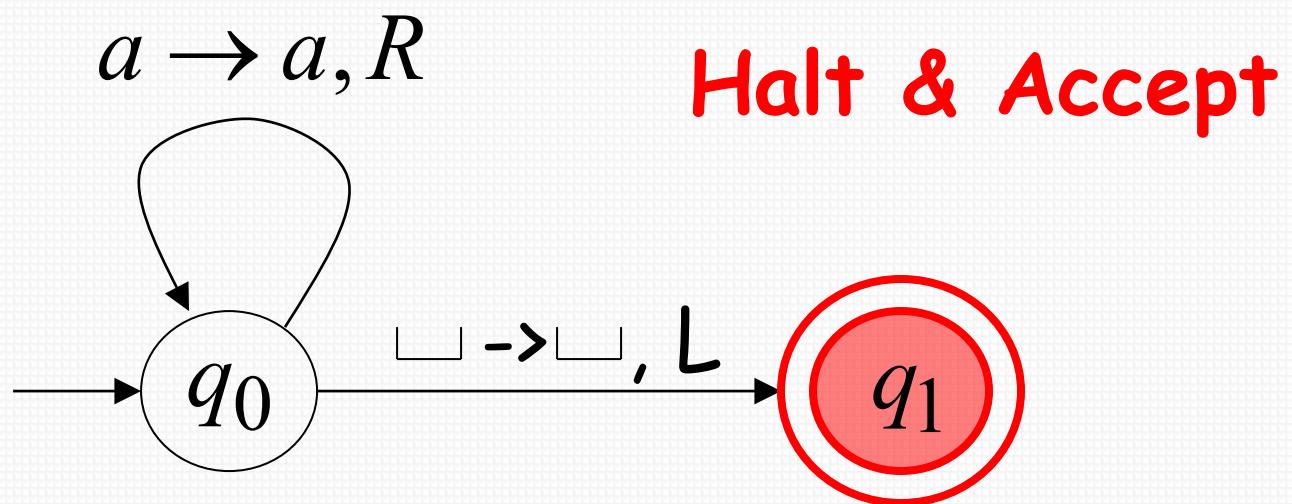
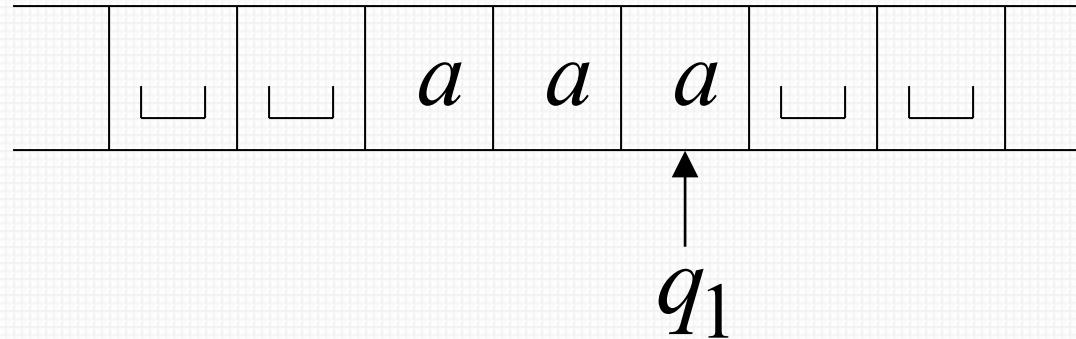
Time 3



$$a \rightarrow a, R$$

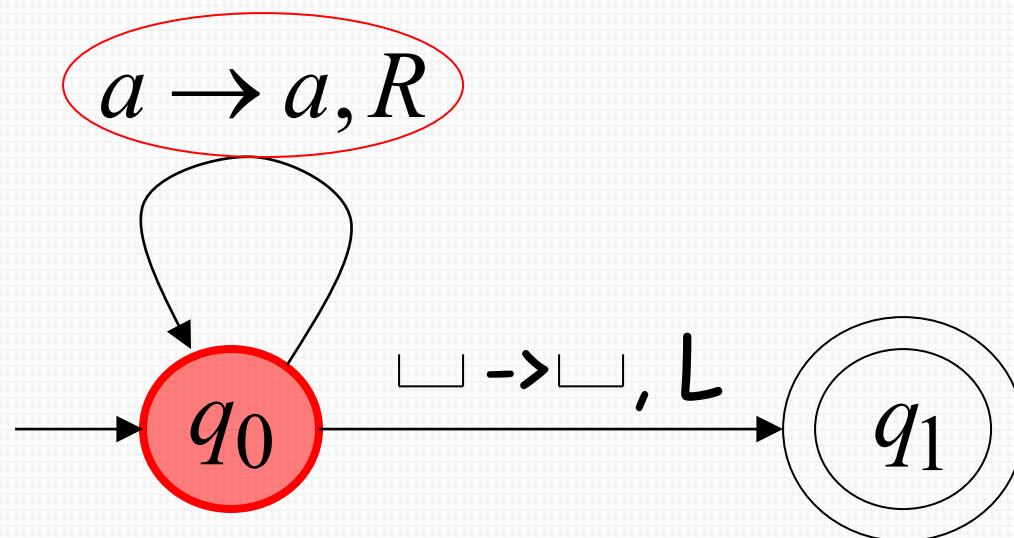
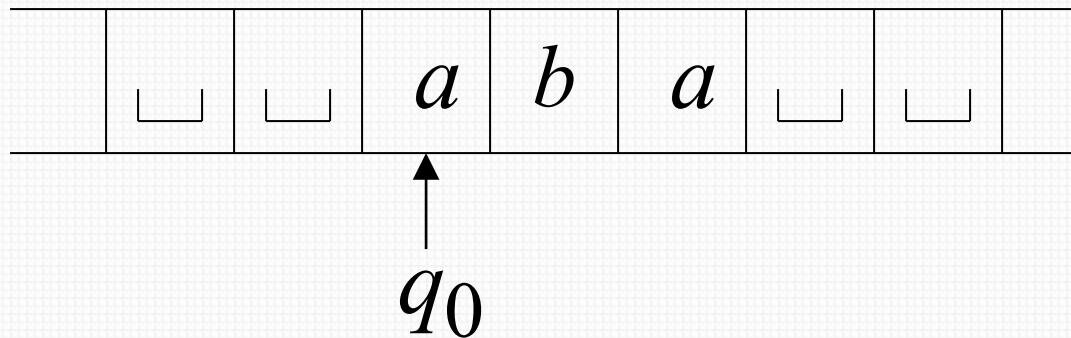


Time 4

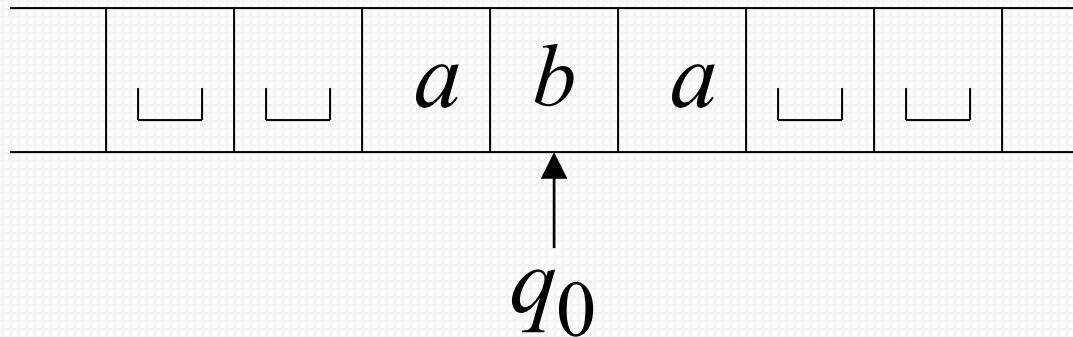


Rejection Example

Time 0



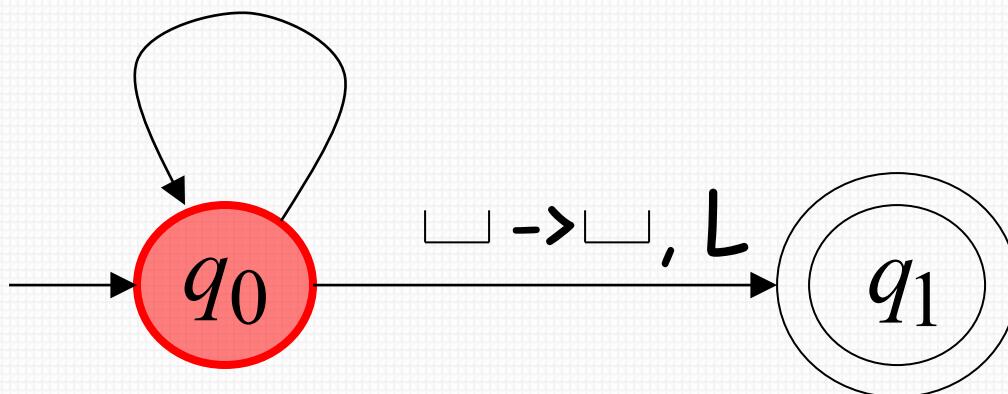
Time 1



No possible Transition

$a \rightarrow a, R$

Halt & Reject

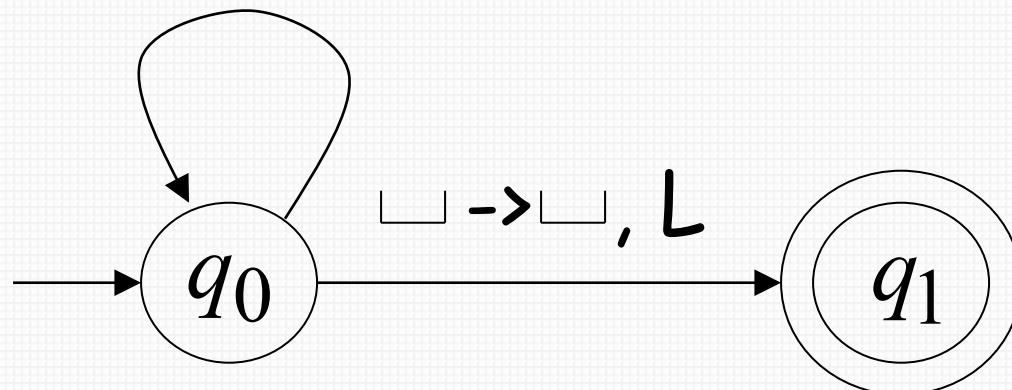


Infinite Loop Example

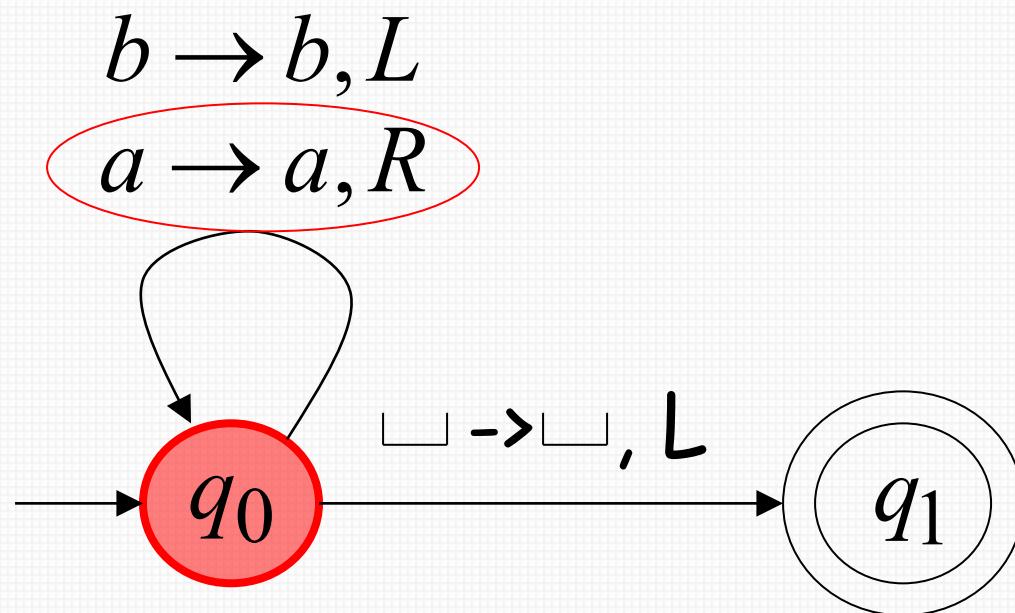
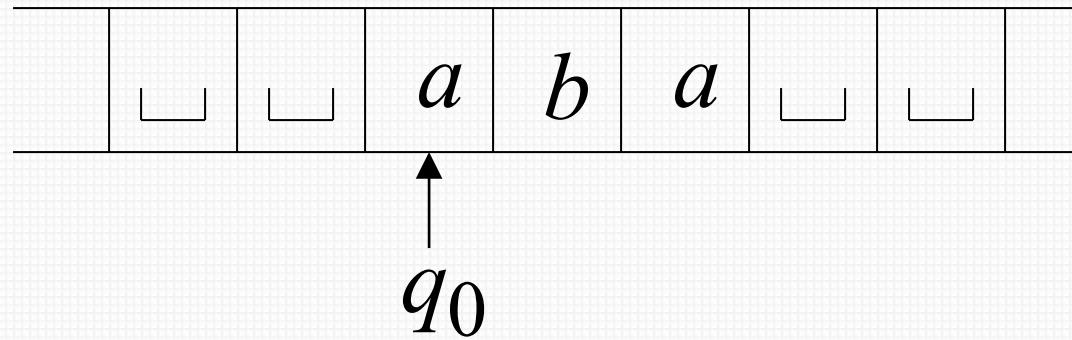
A Turing machine
for language $a^* + b(a+b)^*$

$$b \rightarrow b, L$$

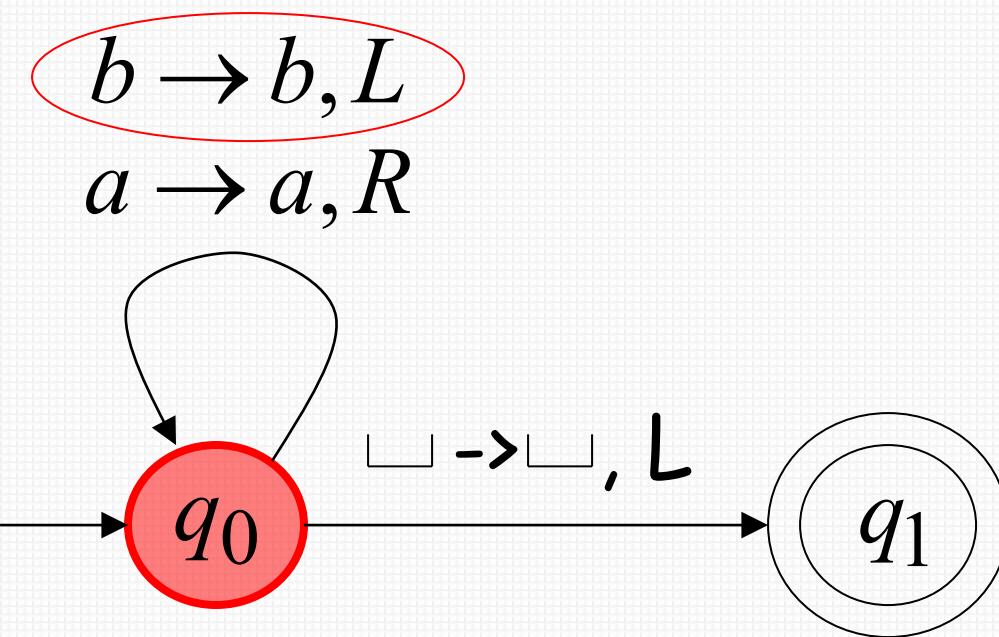
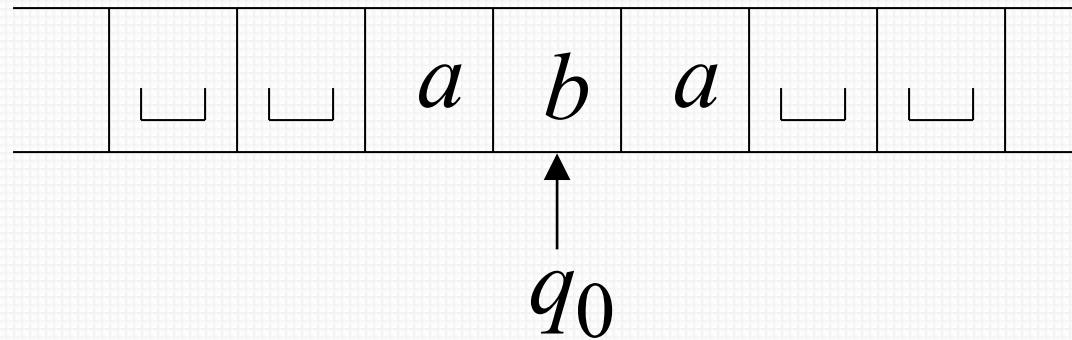
$$a \rightarrow a, R$$



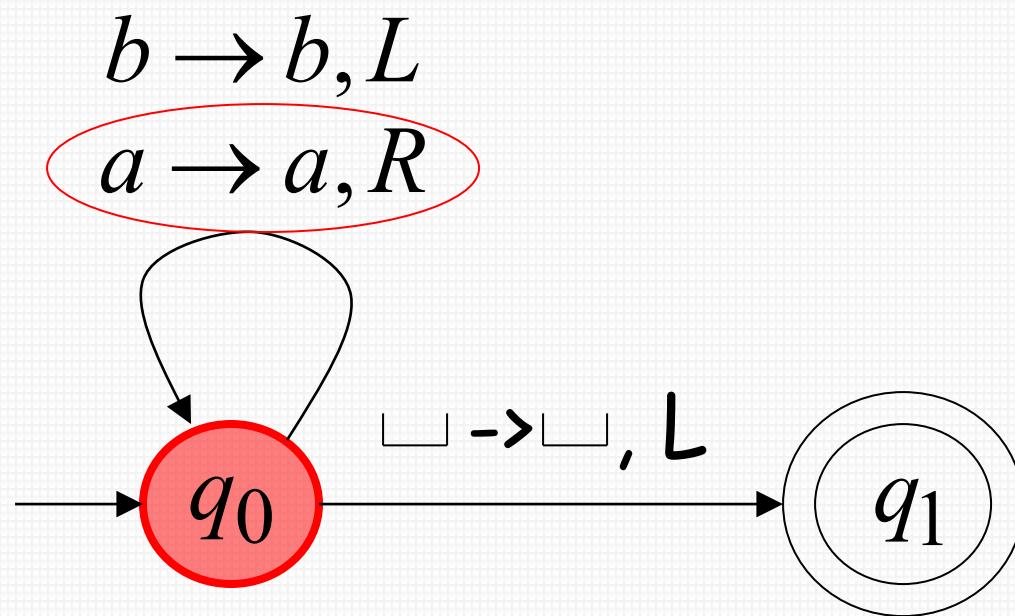
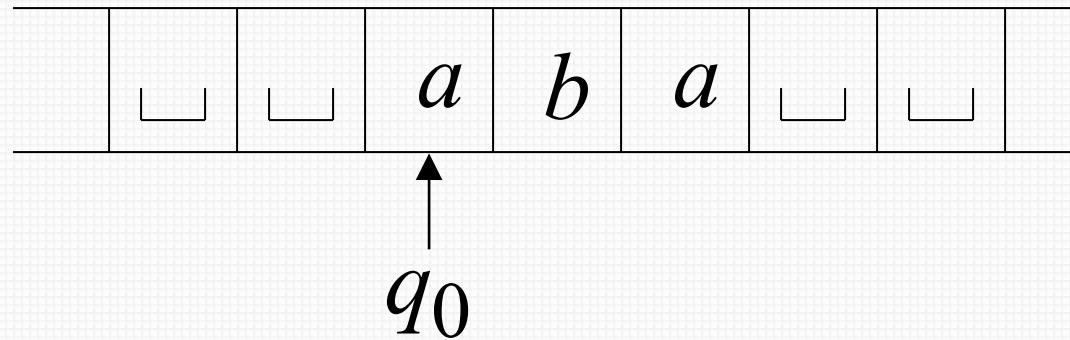
Time 0



Time 1

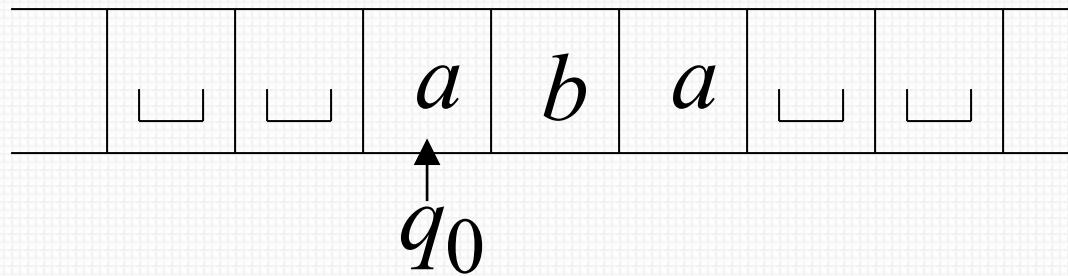


Time 2

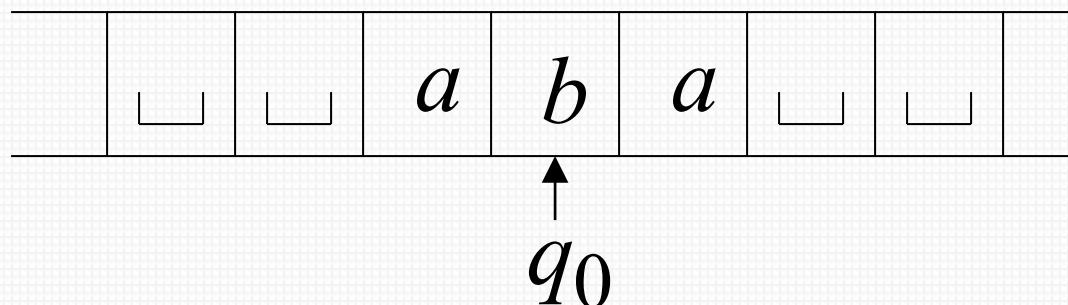


Infinite loop

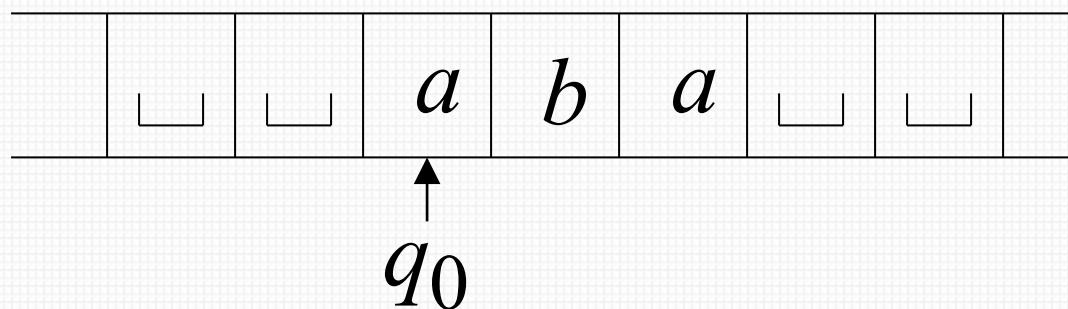
Time 2



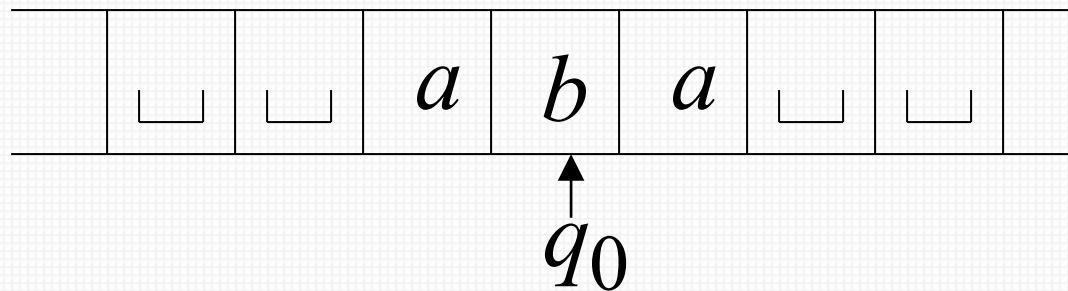
Time 3



Time 4



Time 5



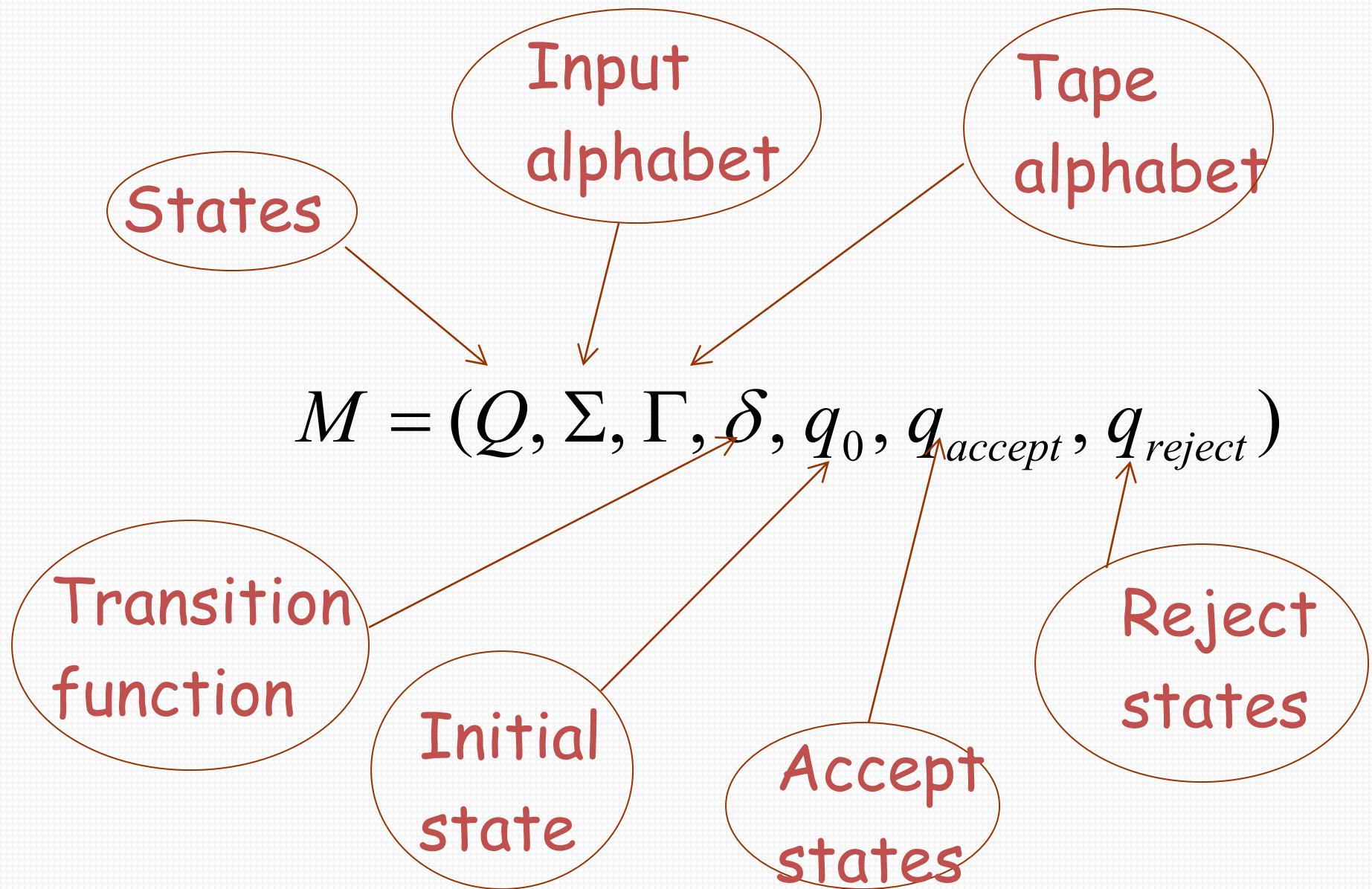
$$B = \{w\#w \mid w \in \{0,1\}^*\}$$

- M1 = “on input string w:
 - Zig-zag across the tape to corresponding positions on either side of the # symbol to check whether these positions contain the same symbol. If they do not, or if no # is found, **reject**. Cross off symbols as they are checked to keep track of which symbols correspond.
 - When all symbols to the left of the # have been crossed off, check for any remaining symbols to the right of the #. If any symbols remains, **reject**; otherwise, **accept**.

- Turing machine M computing on input
011000#011000

0 1 1 0 0 0 # 0 1 1 0 0 0 □ ...
x 1 1 0 0 0 # 0 1 1 0 0 0 □ ...
x 1 1 0 0 0 # x 1 1 0 0 0 □ ...
x 1 1 0 0 0 # x 1 1 0 0 0 □ ...
x x 1 0 0 0 # x 1 1 0 0 0 □ ...
x x x x x x # x x x x x x □ ...
accept

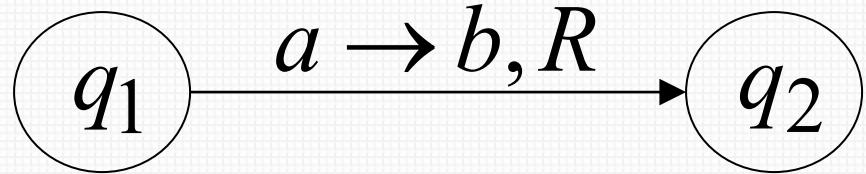
Turing Machine:



- Q is the set of states
- Σ is the input alphabet not containing the blank symbol \square
- Γ is the tape alphabet, where $\square \in \Gamma$ and $\Sigma \subset \Gamma$
- $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is the transition function
- $q_0 \in Q$ is the start state
- $q_{\text{accept}} \in Q$ is the accept state
- $q_{\text{reject}} \in Q$ is the reject state, and $q_{\text{reject}} \neq q_{\text{accept}}$.

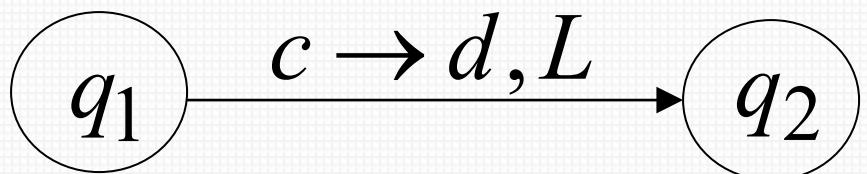
Formal Definitions for Turing Machines

Transition Function



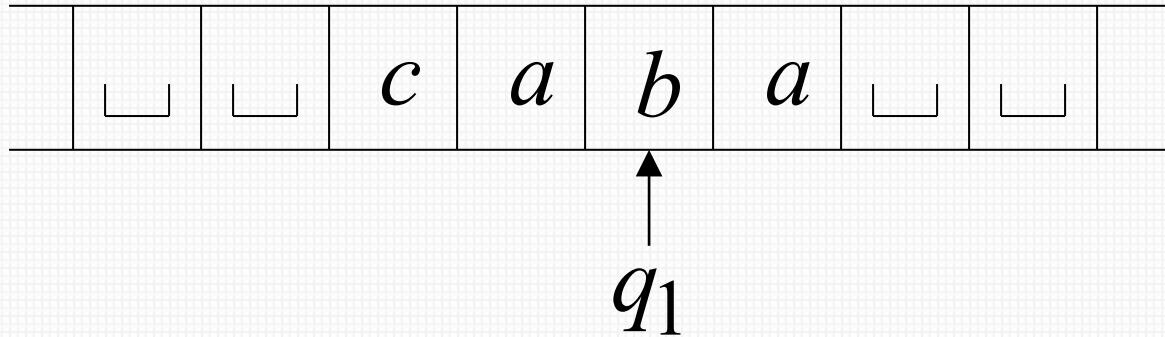
$$\delta(q_1, a) = (q_2, b, R)$$

Transition Function



$$\delta(q_1, c) = (q_2, d, L)$$

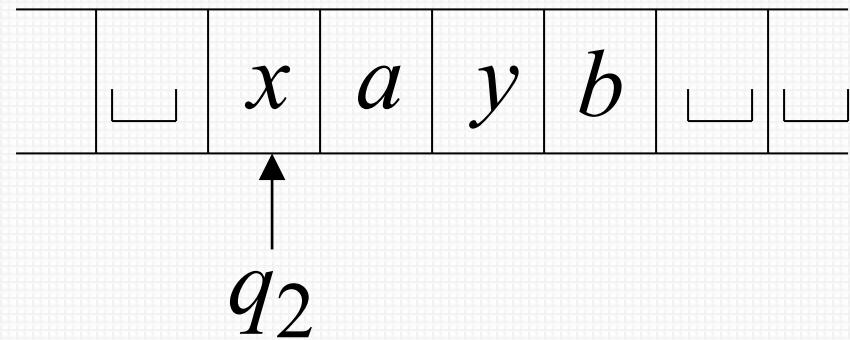
Configuration



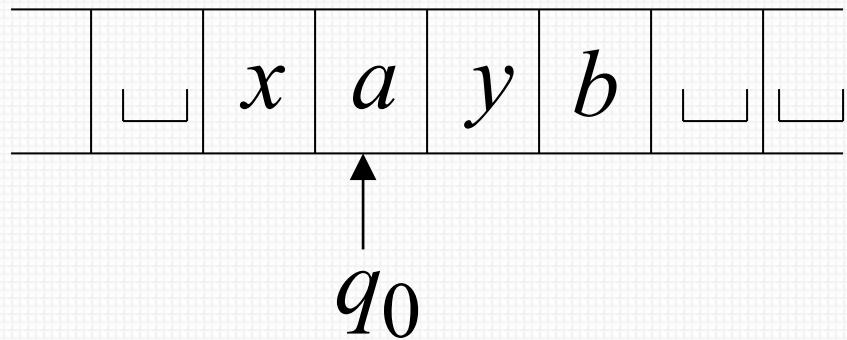
Instantaneous description: $ca\ q_1\ ba$

Configuration={current state, current tape
contents, current head location}

Time 4



Time 5

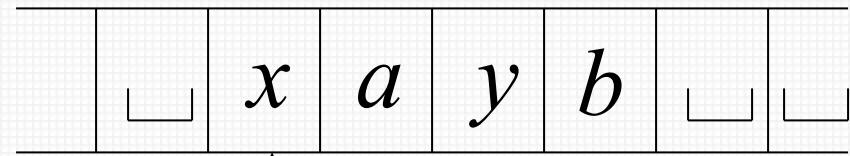


A Move:

$$q_2 \ xayb \succ x q_0 \ ayb$$

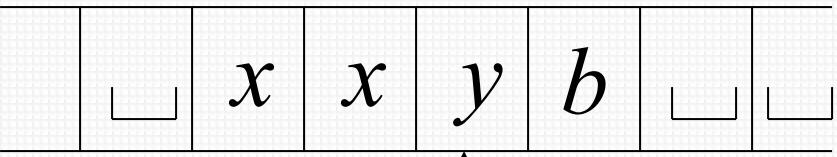
(yields)

Time 4



q_2

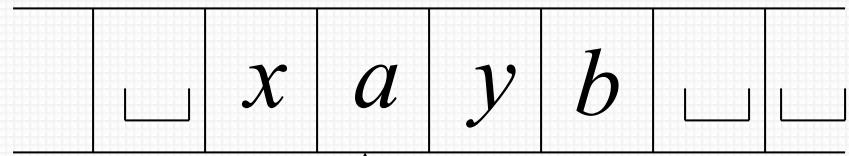
Time 6



q_1

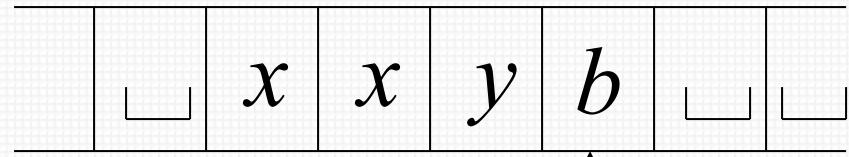
A computation

Time 5



q_0

Time 7



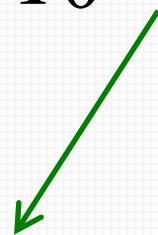
q_1

$q_2 \ xayb \succ x q_0 \ ayb \succ xx q_1 \ yb \succ xxy q_1 \ b$

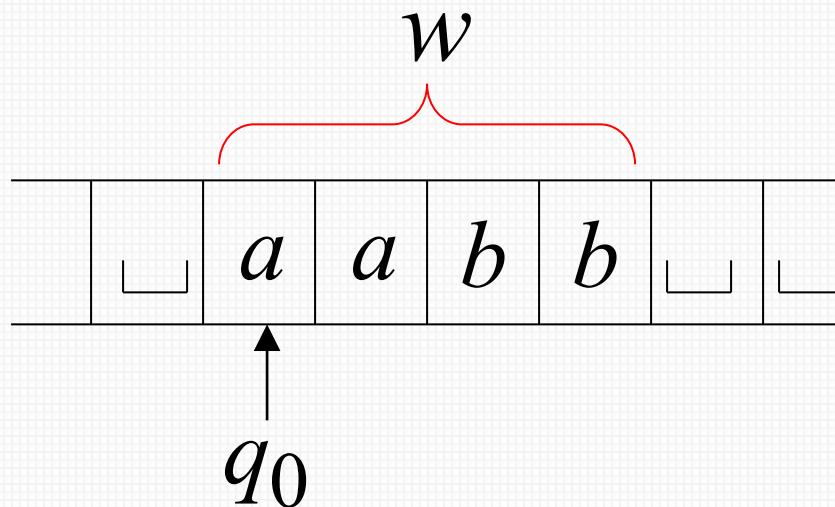
$$q_2 \ xayb \succ x q_0 \ ayb \succ xx q_1 \ yb \succ xxy q_1 b$$

Equivalent notation:
$$q_2 \ xayb \succ^* xxy q_1 b$$

Initial configuration: $q_0 \ w$



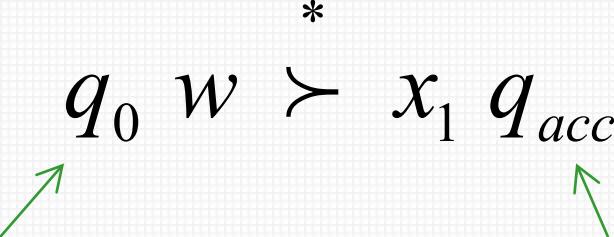
Input string



The Accepted Language

For any Turing Machine M

$$L(M) = \{w : q_0 w \xrightarrow{*} x_1 q_{accept} x_2\}$$


Initial state Accept state

The transition function

$\delta: Q' \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$, Q' is Q without
 q_{reject} and q_{accept}

Definition:

Call a language **Turing Recognizable**
If some Turing machine recognizes it

Other names used:

- Turing Acceptable
- Recursively Enumerable

A Machine may accept, or reject. These machines are called **deciders** because they always make a decision to accept or reject.

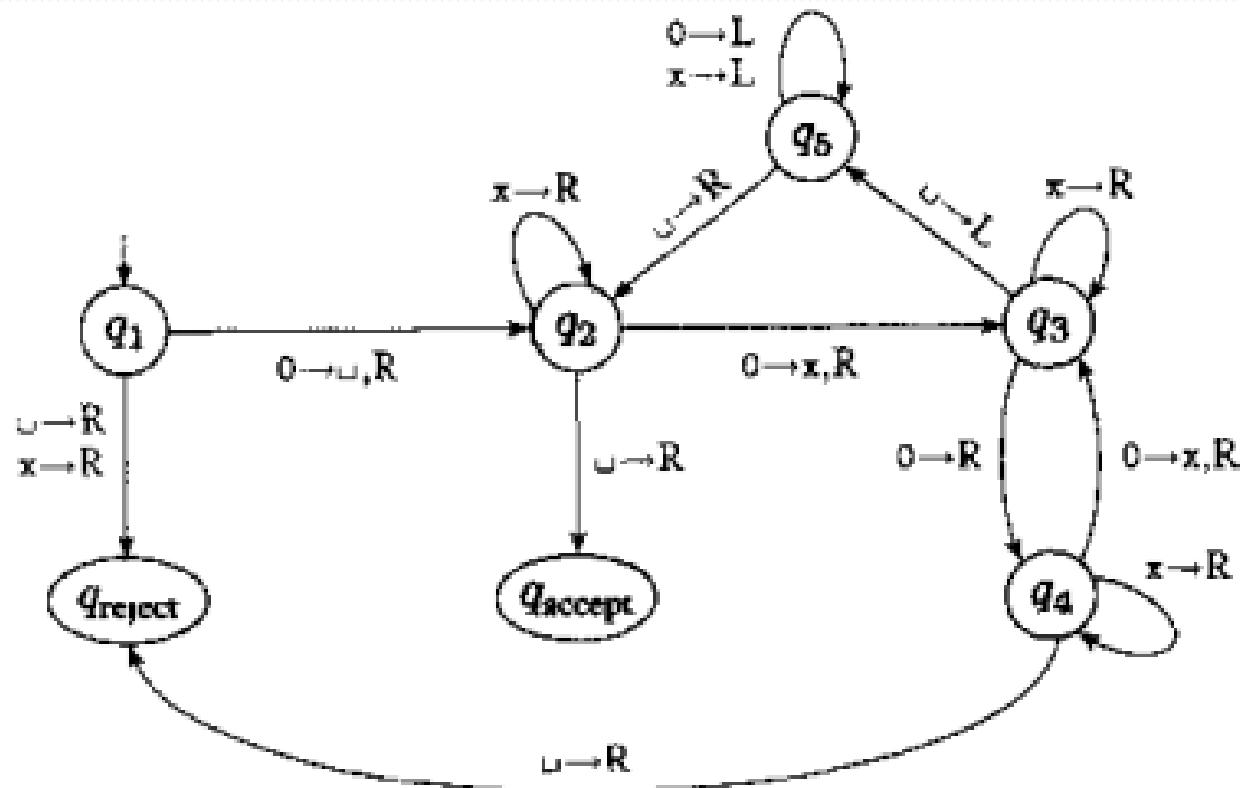
Call a language Turing-decidable or simply decidable if some Turing machine decides it.

- Describe a Turing machine(TM) M that decides

$$A = \{0^{2^n} \mid n \geq 0\}$$

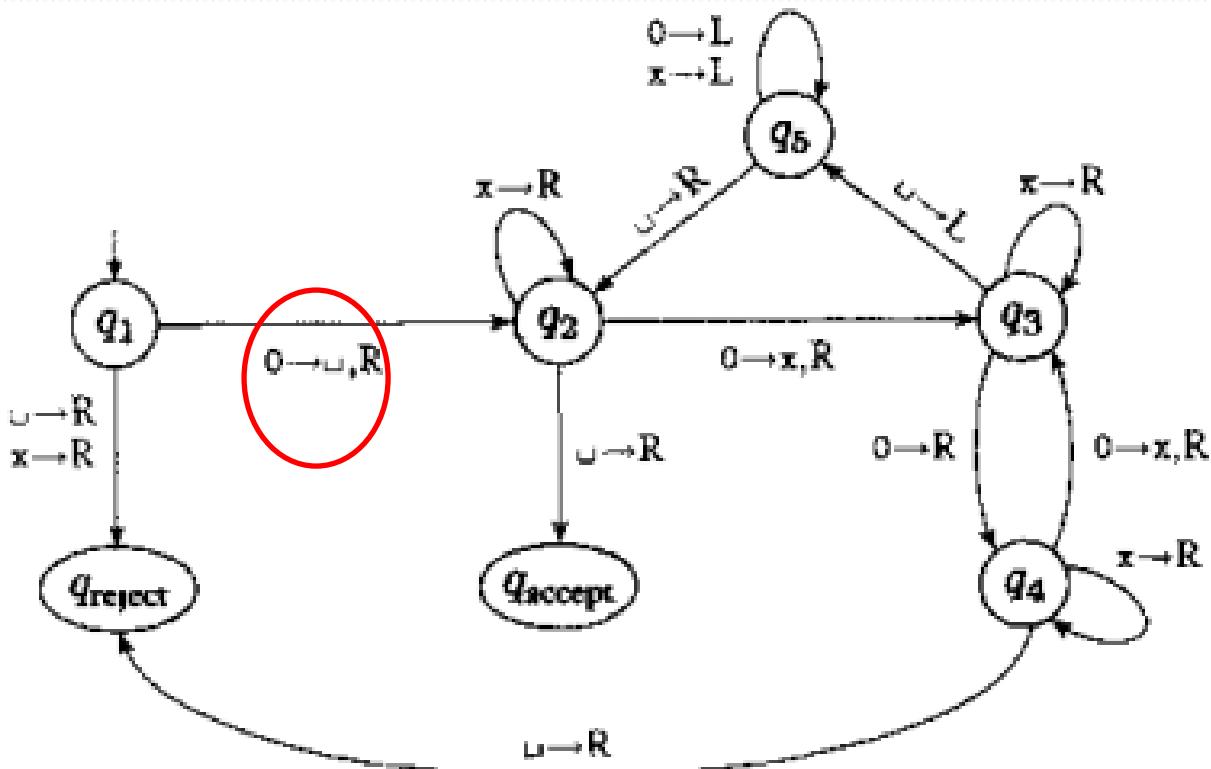
- The language consisting of all strings of 0s whose length is a power of 2
- M = “ On input string w:
 - 1. Sweep left to right across the tape, crossing off **every other 0** (隔一个)
 - 2. If in stage 1 the tape contained a single 0, accept.
 - 3. If in stage 1 the tape contained more than a single 0 and the number of 0s was odd, reject
 - 4. Return the head to the left-hand end of the tape
 - 5. Go to stage 1.

- Explanation: each iteration of stage 1 cuts the number of 0s in half.



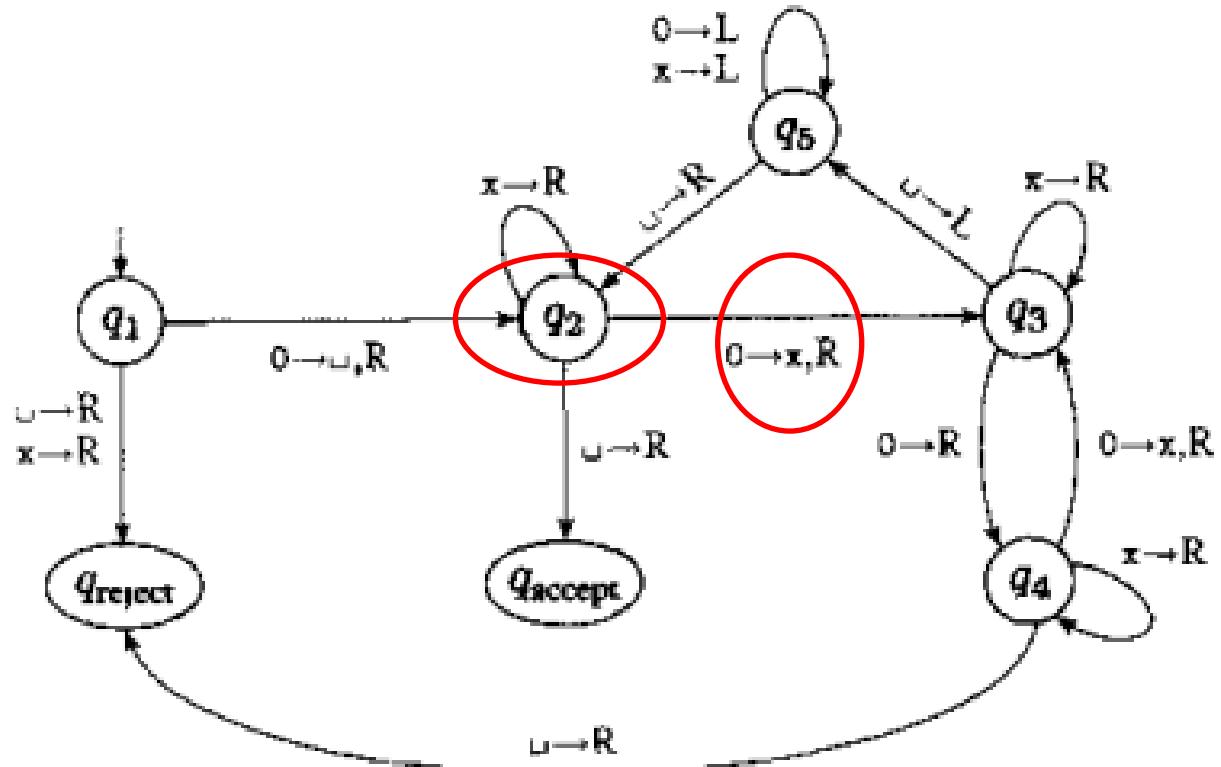
Input 0000

► $q^1 0000$



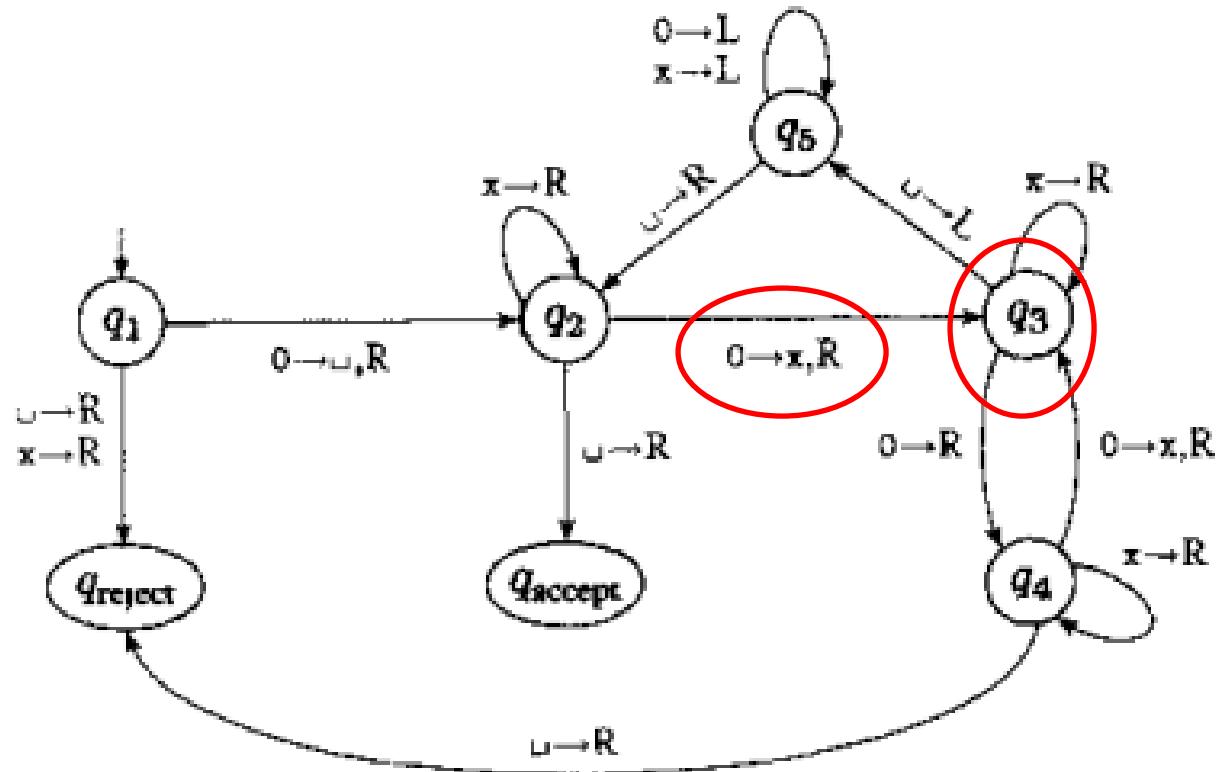
Input 0000

- $q^1 0000$
- $\sqcup q^2 000$



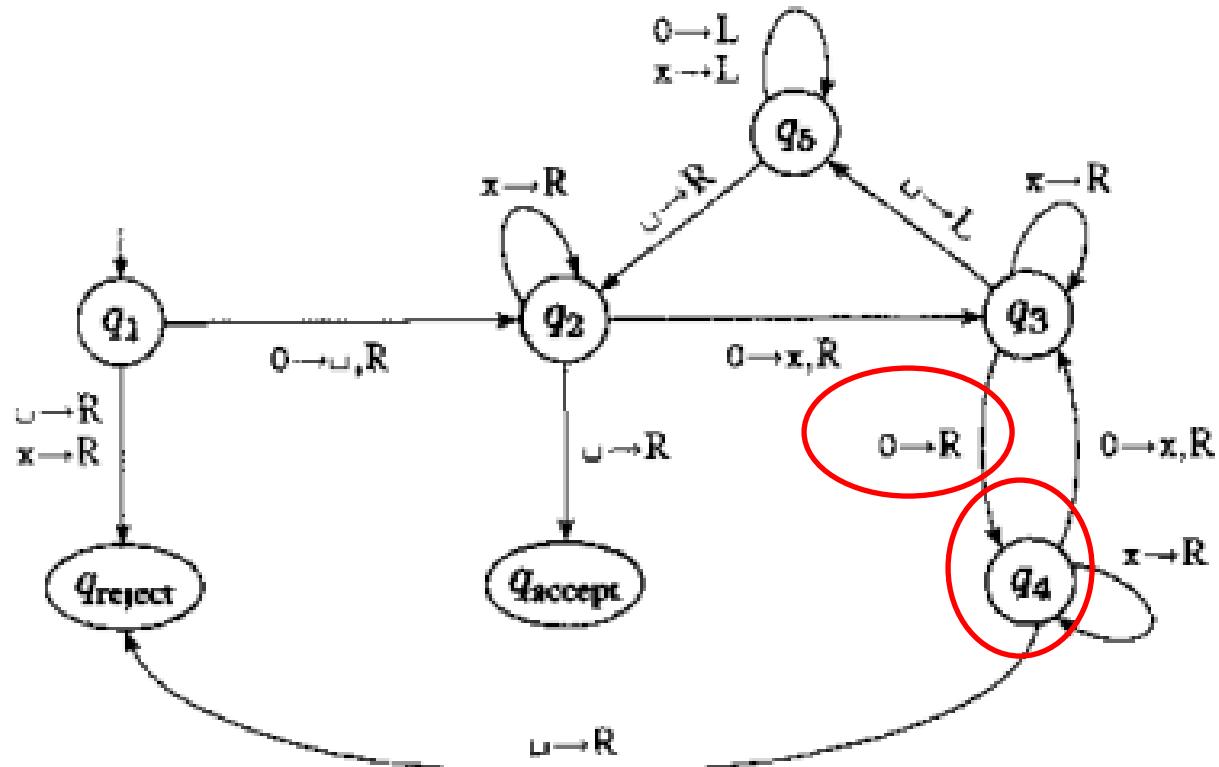
Input 0000

- ▶ $q^1 0000$
- ▶ $\sqsubset q^2 000$
- ▶ $\sqsubset x q^3 00$



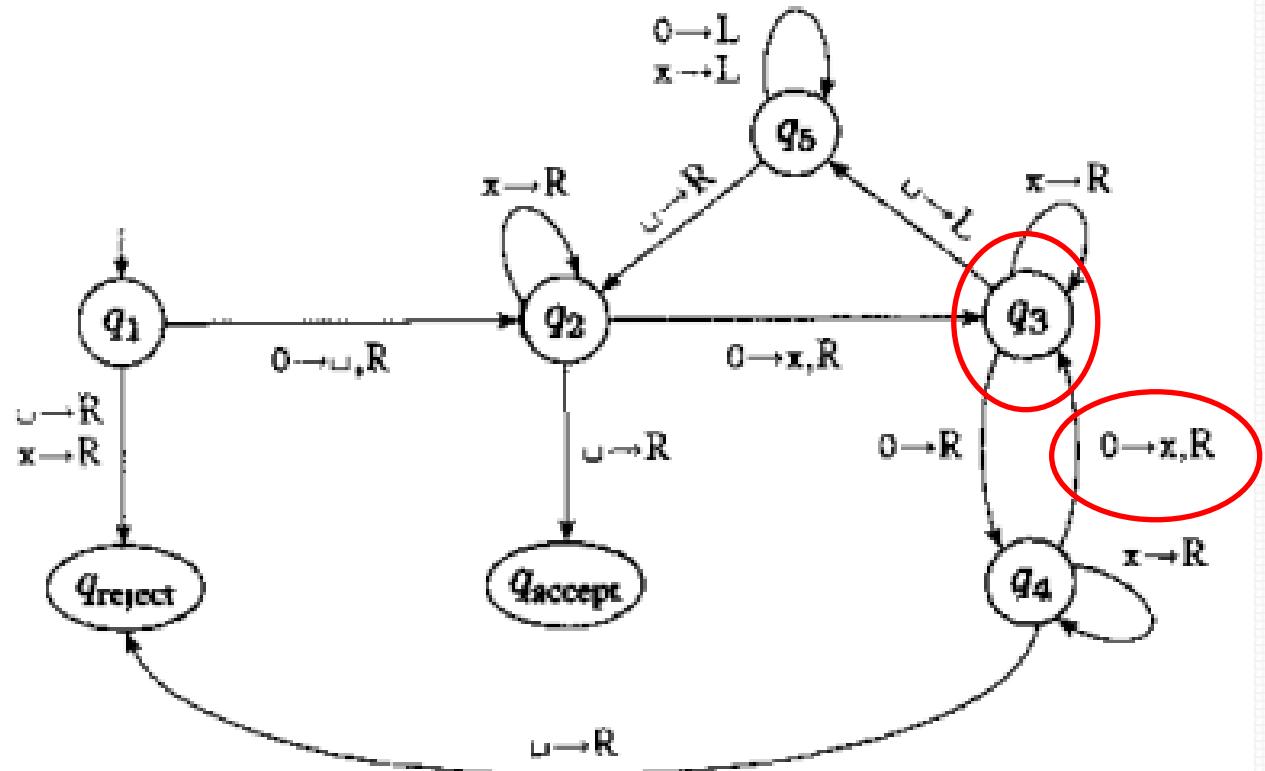
Input 0000

- ▶ $q^1 0000$
- ▶ $\sqsubset q^2 000$
- ▶ $\sqsubset x q^3 00$
- ▶ $\sqsubset x 0 q^4 0$



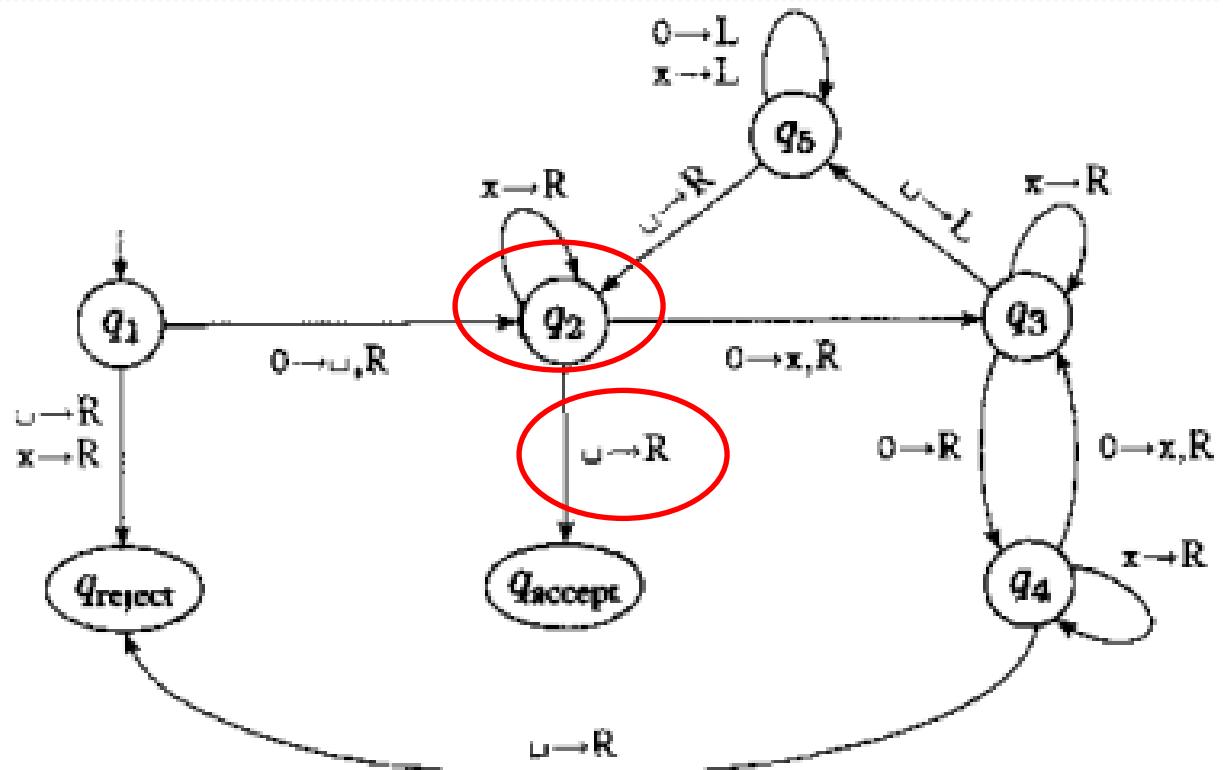
Input 0000

- ▶ $q^1 0000$
- ▶ $\sqcup q^2 000$
- ▶ $\sqcup x q^3 00$
- ▶ $\sqcup x 0 q^4 0$
- ▶ $\sqcup x 0 x q^3 0 \sqcup$



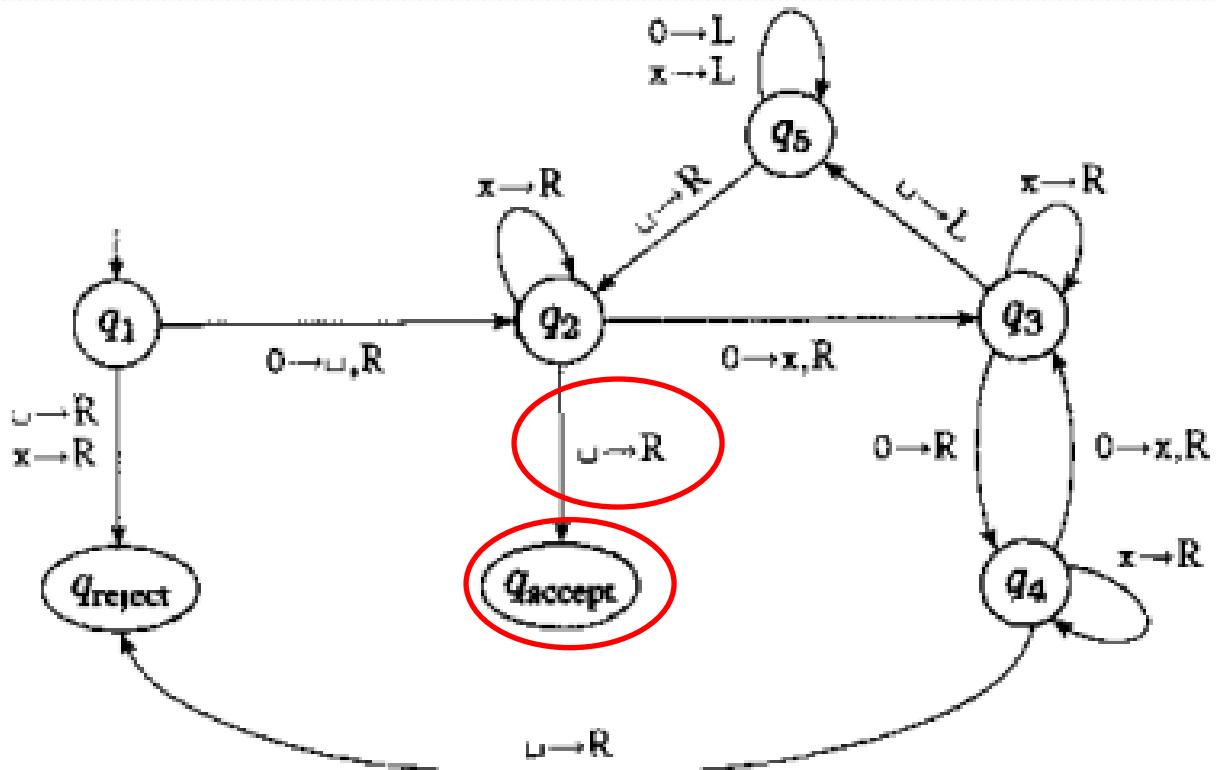
continue

- . . .
- . . .
- . . .
- . . .
- $\llbracket \text{XXXq}^2 \rrbracket$



accept

- xxxq^2
 - $\text{XXX q}_{\text{accept}}$



- Turing machine decide the language

$$C = \{a^i b^j c^k \mid i \times j = k \text{ and } i, j, k \geq 1\}$$

M = “on input string w:

1. Scan the input from left to right to determine whether it is a member of $a^+b^+c^+$ and reject if it isn’t.
2. Return the head to left-hand end of the tape

- Cross off an a and scan to the right until a b occurs. Shuttle between the b's and the c's, crossing off one of each until all b's gone. If all c's have been crossed off and some b's remain, reject.
- Restore the crossed off b's and repeat stage 3 if there is another a to cross off. If all a's have been crossed off, determine whether all c's also have been crossed off. If yes, accept; otherwise, reject.

- ✓ The Language Hierarchy
- ✓ Automata-Recognizable language Table
- ✓ Turing Machine
- ✓ Halting & Accepting
- ✓ Formal Definitions for Turing Machines
- ✓ Turing Recognizable
- ✓ Turing-decidable
- ✓ Examples

Theory of Computation

Lesson 6-3

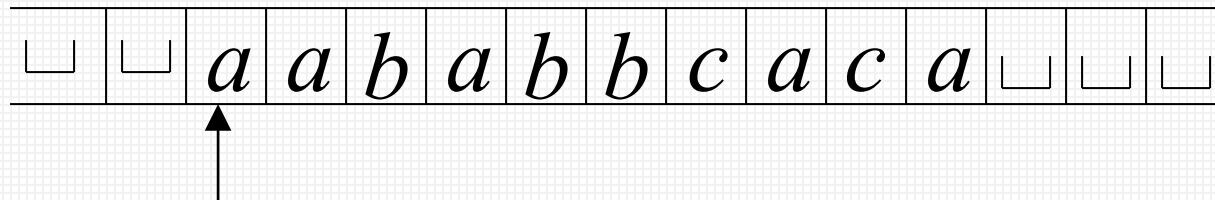
Variants of Turing Machines



王 轩
Wang
Xuan

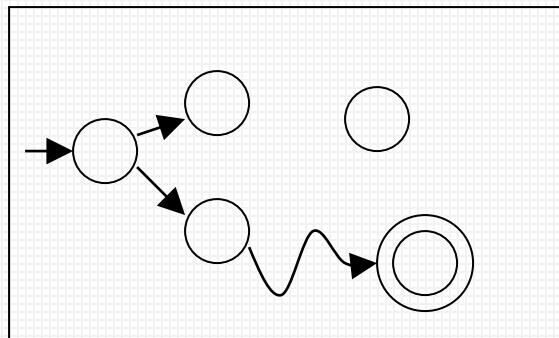
- Variations of the Standard Model
- Simulation
- Turing Machines with Stay-Option
- Multitape Turing Machines
- Nondeterministic Turing Machines
- Enumerators

Infinite Tape



Read-Write Head (Left or Right)

Control Unit



Deterministic

Variations of the Standard Model

Turing machines with:

- Stay-Option
- Multitape
- Nondeterministic
- Enumerators

Different Turing Machine Classes

Same Power of two machine classes:

both classes accept the
same set of languages

We will prove:

each new class has the same power
with Standard Turing Machine

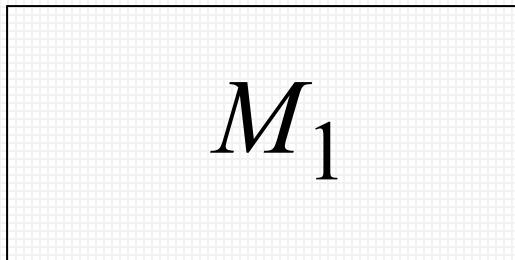
(accept Turing-Recognizable Languages)

Simulation: A technique to prove same power.

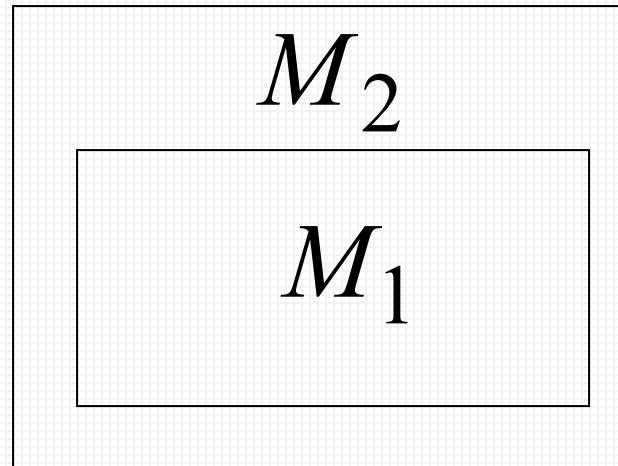
Simulate the machine of one class
with a machine of the other class

First Class

Original Machine



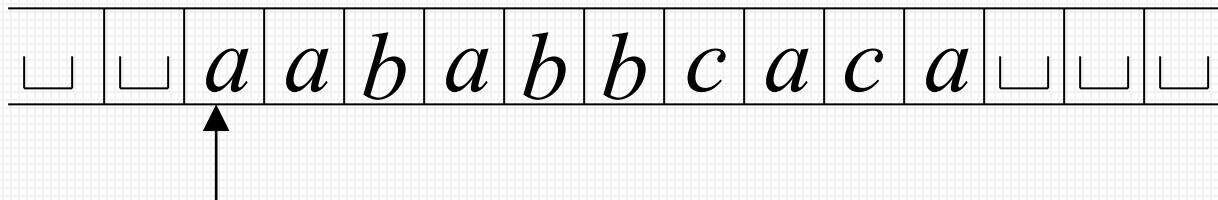
Second Class
Simulation Machine



simulates M_1

Turing Machines with Stay-Option

The head can stay in the same position



Left, Right, Stay

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

Example:

Time 1

◻ ◻ *a a b a b b c a c a* ◻ ◻ ◻



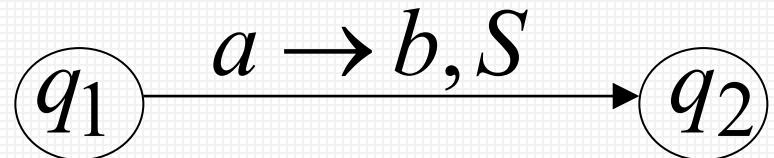
q_1

Time 2

◻ ◻ *b a b a b b c a c a* ◻ ◻ ◻



q_2



Theorem: Stay-Option machines
have the same power with
Standard Turing machines

Proof: 1. Stay-Option Machines
simulate Standard Turing machines

2. Standard Turing machines
simulate Stay-Option machines

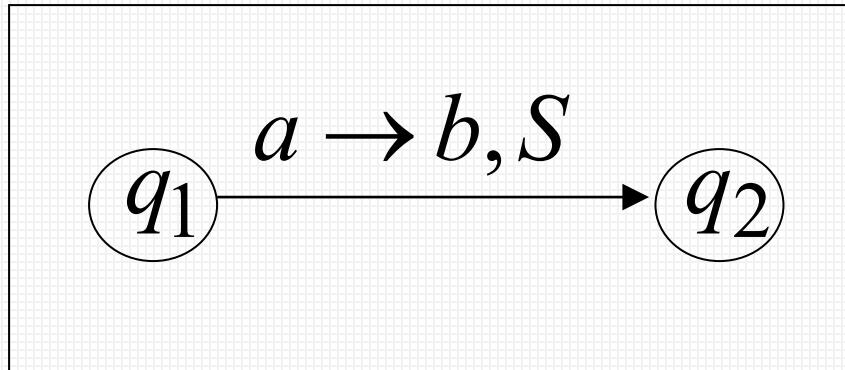
1. Stay-Option Machines simulate Standard Turing machines

Trivial: any standard Turing machine
is also a Stay-Option machine

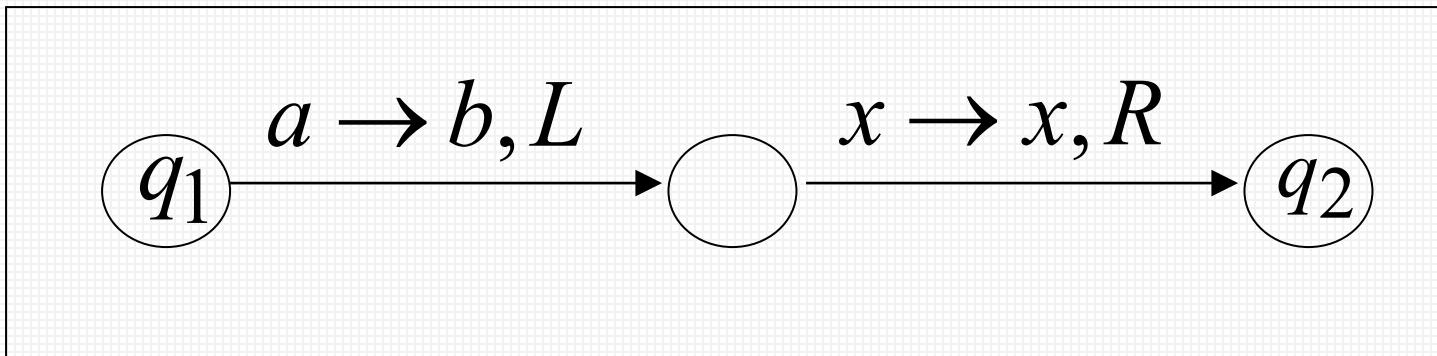
2. Standard Turing machines simulate Stay-Option machines

We need to simulate the stay head option
with two head moves, one left and one right

Stay-Option Machine



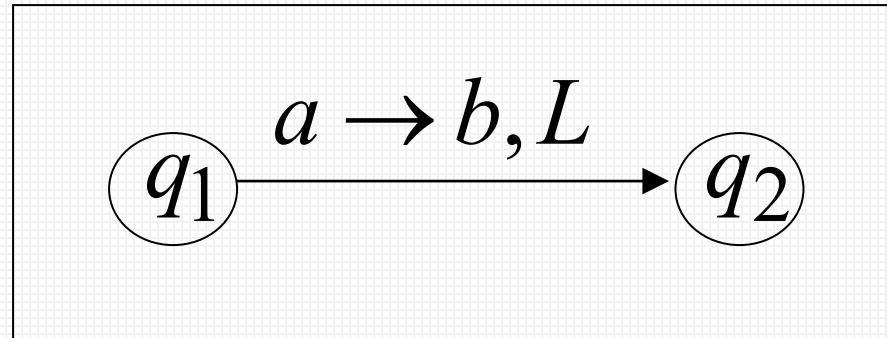
Simulation in Standard Machine



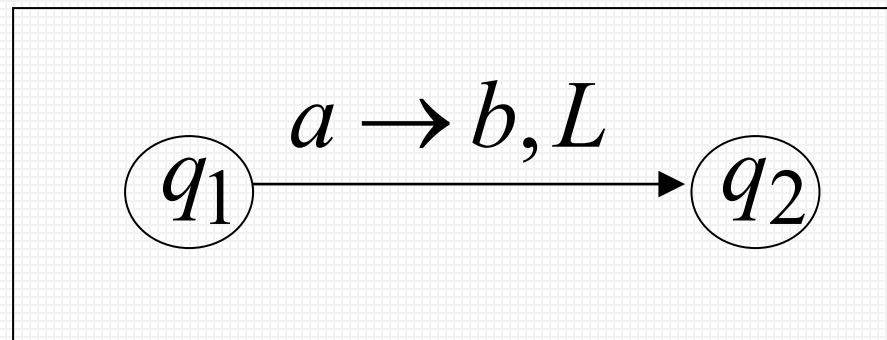
For every possible tape symbol x

For other transitions nothing changes

Stay-Option Machine

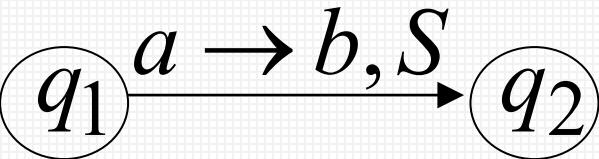


Simulation in Standard Machine

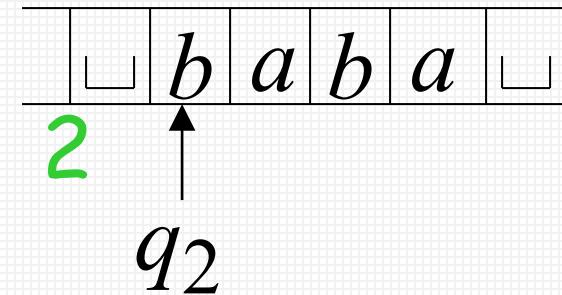
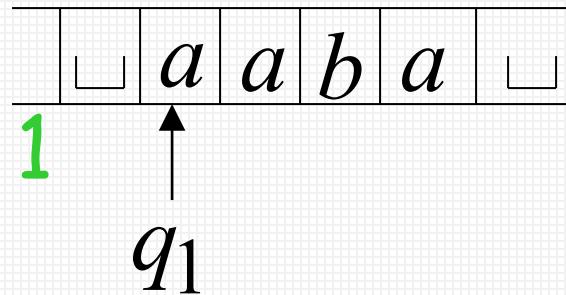


Similar for Right moves

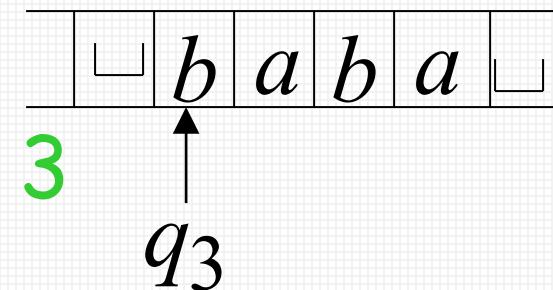
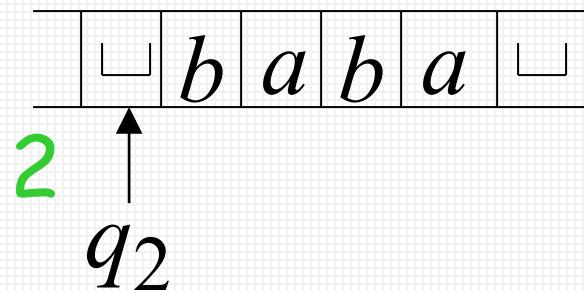
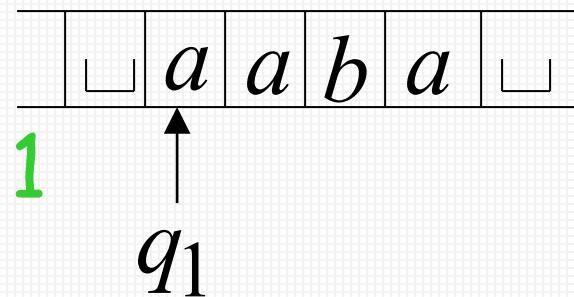
example of simulation



Stay-Option Machine:



Simulation in Standard Machine:



END OF PROOF

Multitape Turing Machines

- Transition function in Turing machine with stay option

$$\delta(Q, \Gamma) = Q \times \Gamma \times \{L, R, S\}$$

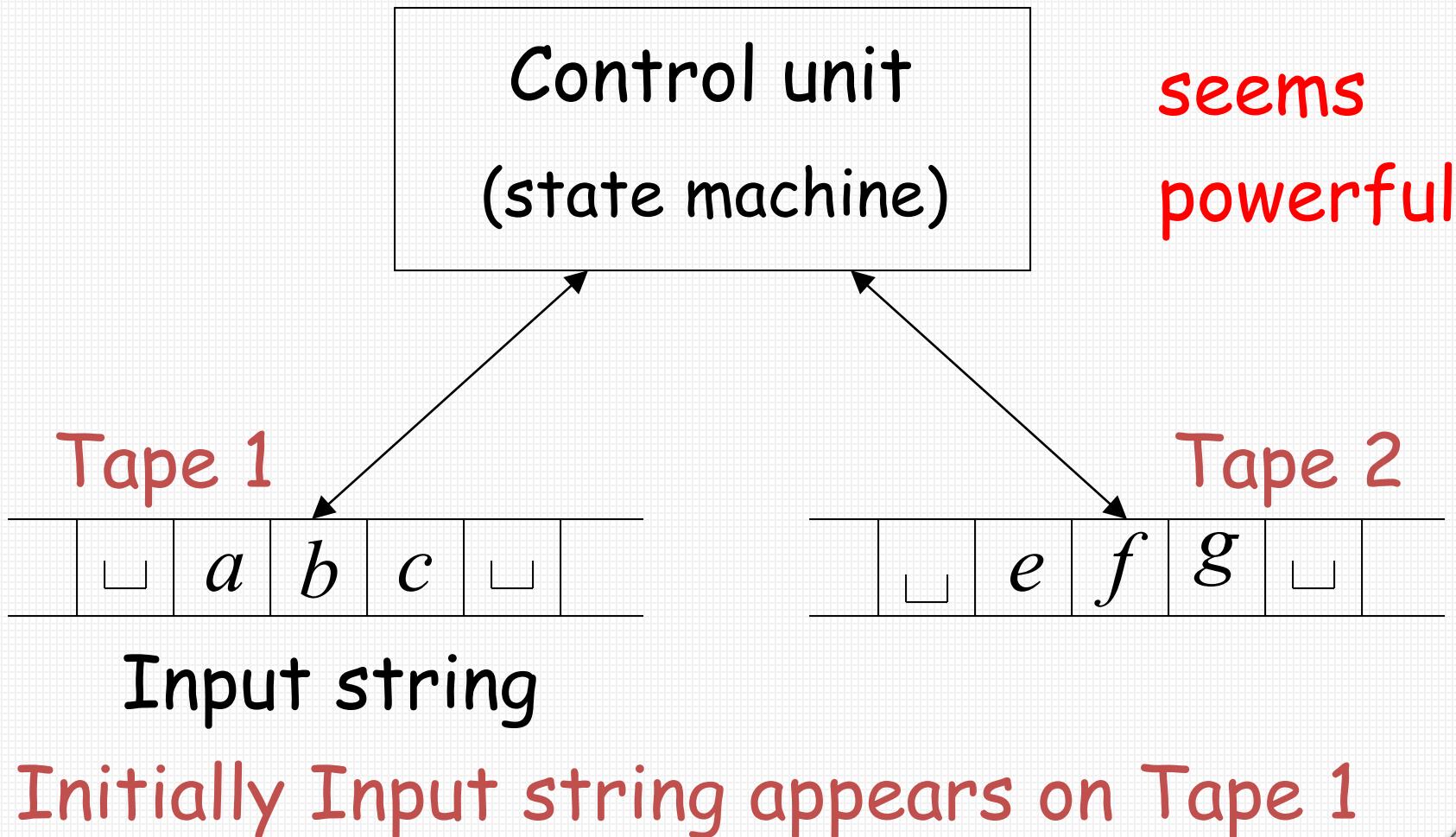
- Multitape Turing machine has several tapes, each tape has its own head for reading and writing.
Formally,

$$\delta(Q, \Gamma^k) = Q \times \Gamma^k \times \{L, R, S\}^k$$

- The expression

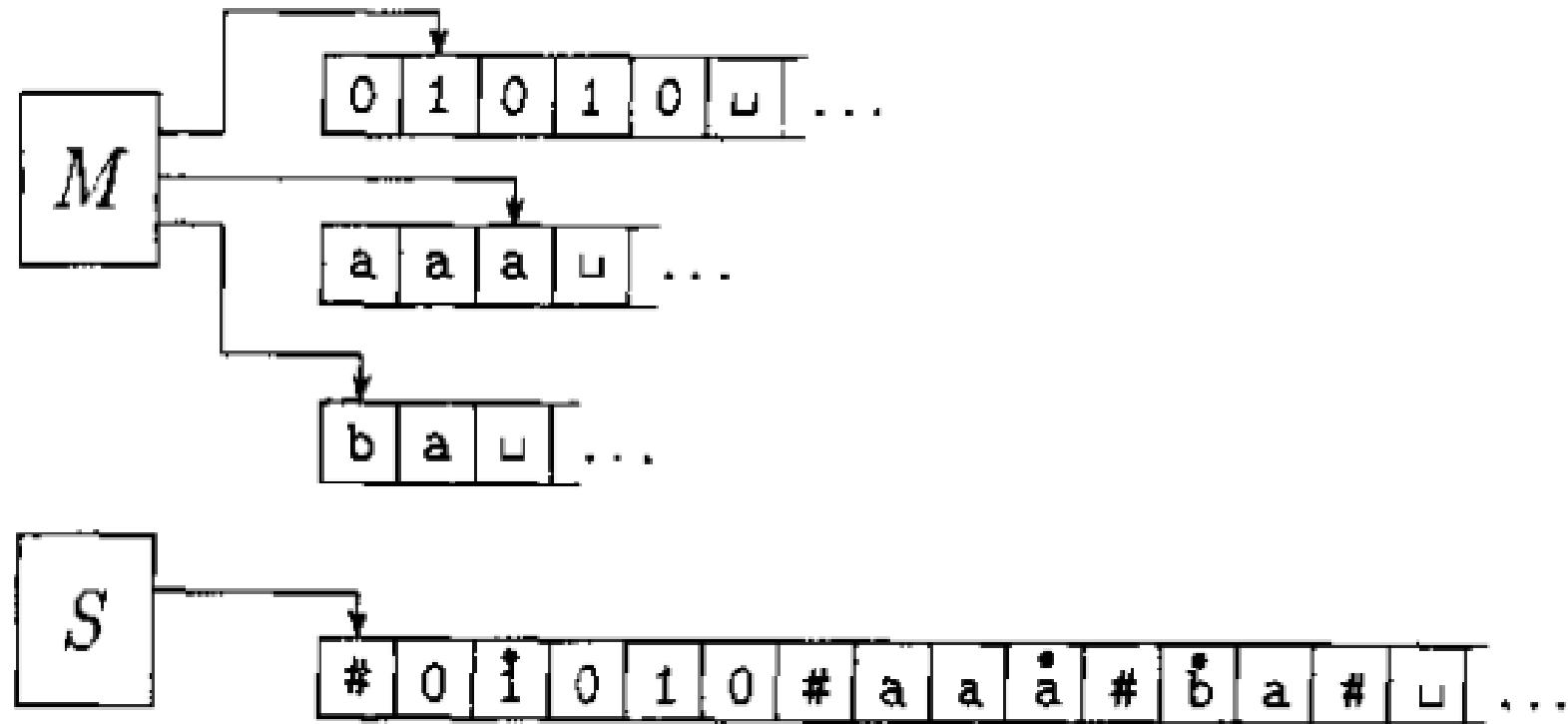
$$\delta(q_i, a_1, \dots, a_k) = (q_j, b_1, \dots, b_k, L, R, \dots, L)$$

Multitape Turing Machines



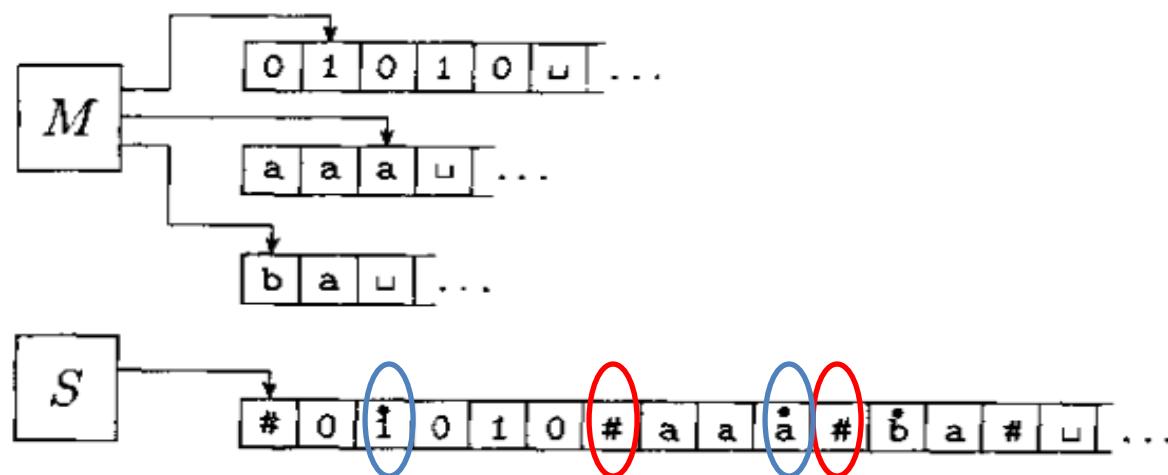
Standard Turing Machines (S)

Simulate Multi-tape machines (M)



- S simulates k tapes of M by storing their information on its single tape, using # as a delimiter to separate different tapes.
- To keep track the locations of the heads, write tap symbol with a dot above it to mark the place where the head on that tap would be, as

“vi



- $S = \text{"on input } w=w_1\dots w_n:$

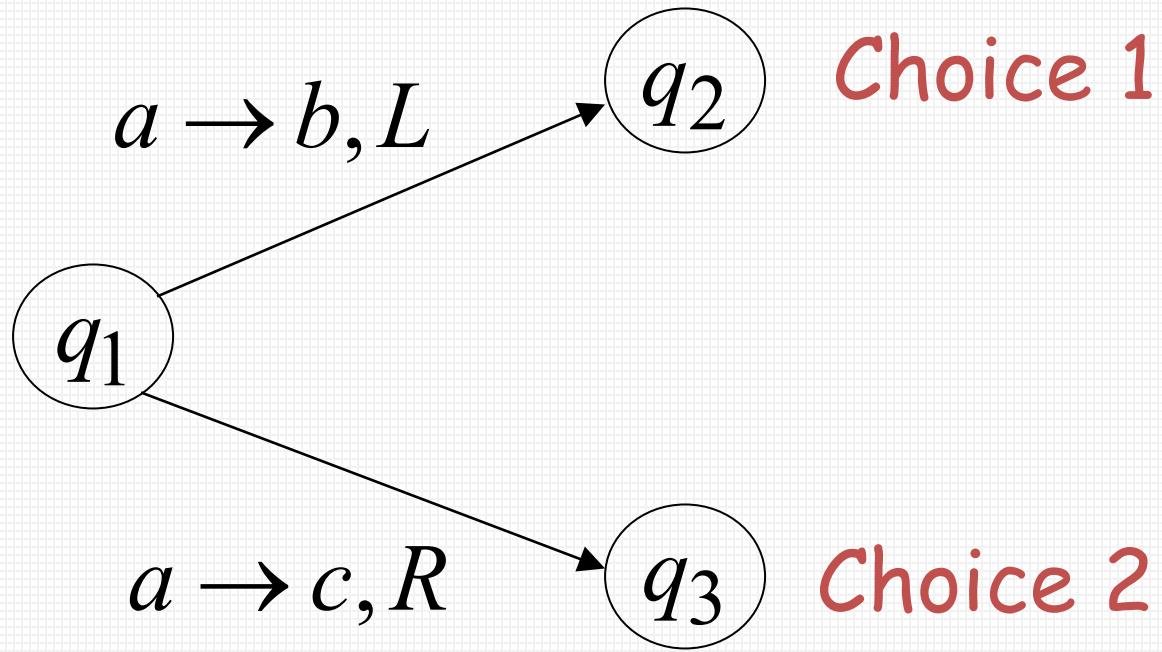
- 1. First s puts its tape into the format that represents all k tapes of M

$\# \overset{\bullet}{w_1} w_2 \dots w_n \# \overset{\bullet}{\#} \overset{\bullet}{\#} \dots \#$

- 2. To simulate, it scans from the first $\#$, to the $(k+1)_{st}\#$, in order to determine the symbols under the virtual heads. Then S makes a second pass to update the tapes according to M 's transition function.
 - 3. If S moves one of its heads to the right onto a $\#$, it signifies that M has moved the head onto the previously unread blank portion of the tape. So S writes a blank symbol on this tape and shifts the tape contents, from this cell until the rightmost $\#$, one unit to the right. Then it continues the simulation as before.

- A language is Turing-recognizable if and only if some multitape Turing machine recognizes it.
- Proof:
 - An ordinary (single-tape) Turing machine is a special case of a multitape Turing machine.
 - Every multitape Turing machine has an equivalent single-tape Turing machine.

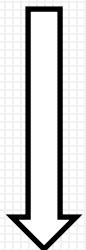
Nondeterministic Turing Machines



Allows Non Deterministic Choices

Transition Function

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

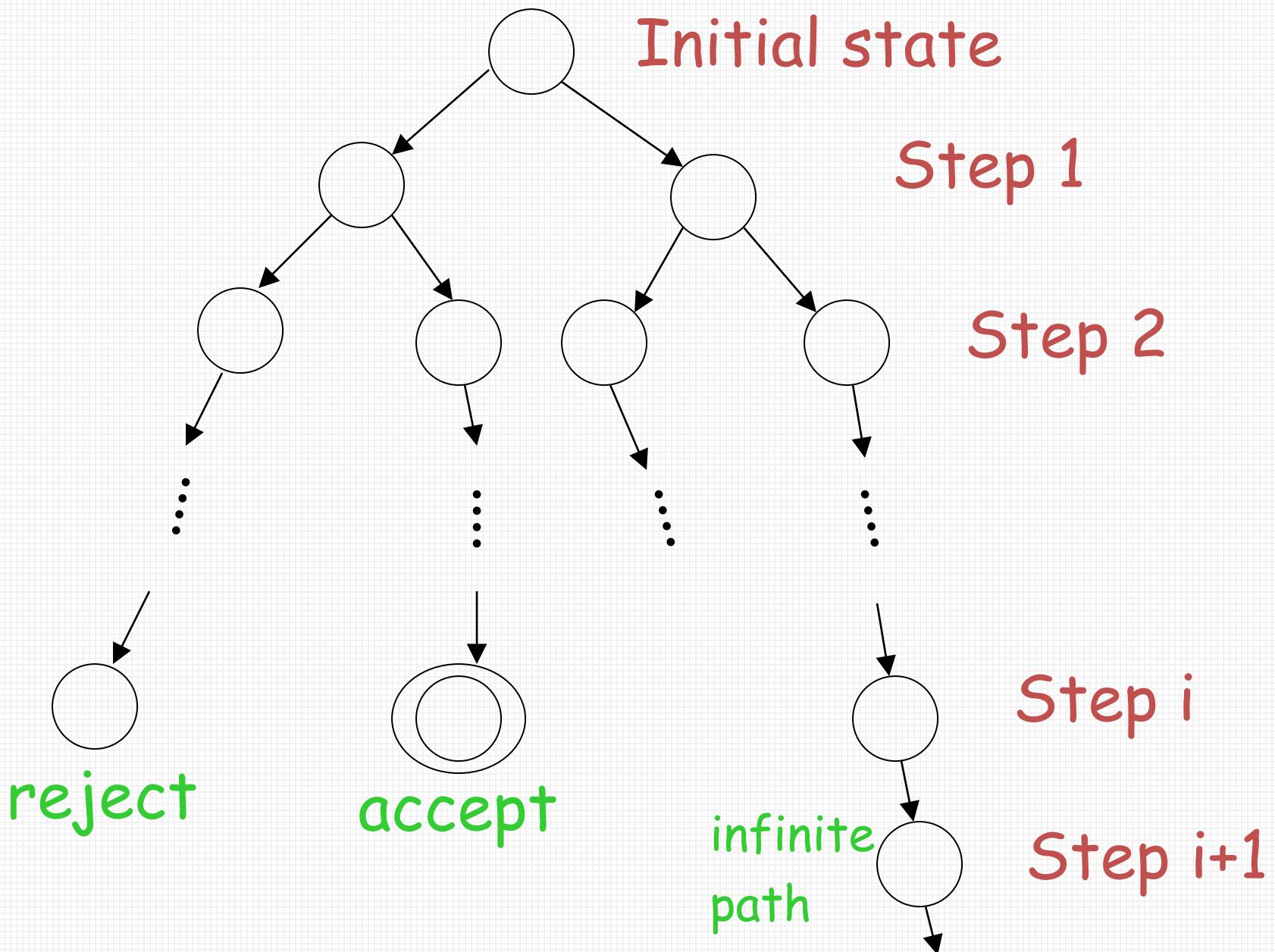


$$\delta : Q \times \Gamma = P(Q \times \Gamma \times \{L, R, S\})$$

P means the computation is a tree whose branches correspond to different possibilities for the machine

- ▶ Simulate nondeterministic TM N with a deterministic TM D. TM D try all possible branches of N. If D ever finds the accept state on one of them, D accepts.
- ▶ View N's computation on w as a tree. Each branch of the tree is one of the branches of nondeterminism. Each node of the tree is a configuration of N.
- ▶ TM D Searches the tree for an accepting configuration
 - Breadth first search: guarantee D will visit every node until it meets an accepting configuration
 - Depth-first search: may go forever on infinite branch 

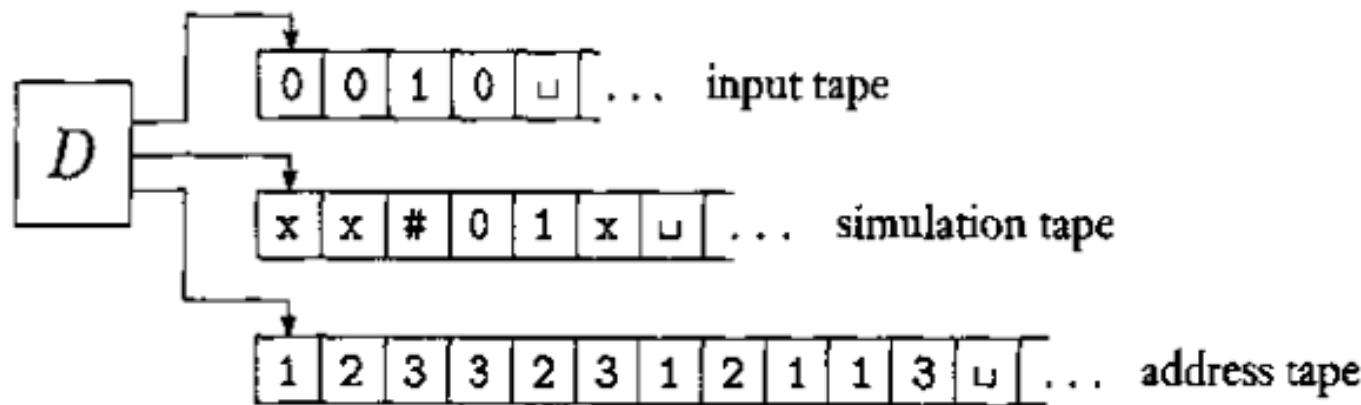
All possible computation paths



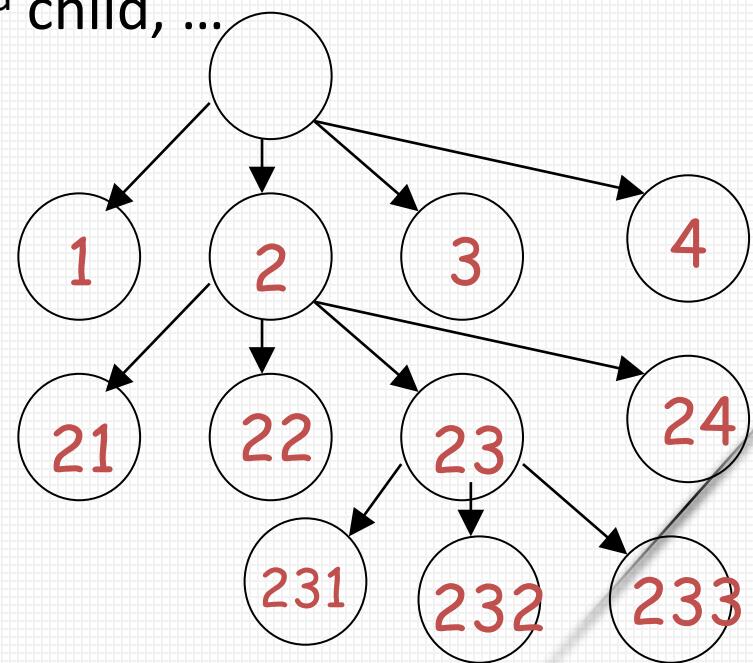
The Deterministic Turing machine
simulates all possible computation paths:

- simultaneously
- step-by-step
- in a breadth-first search fashion

- Simulating deterministic TM D has three tapes
 - Tape 1 always contains the input string and is never altered.
 - Tape2 maintains a copy of N's tape on some branch of its nondeterministic computation.
 - Tape3 keeps track of D's location in N's nondeterministic computation tree.

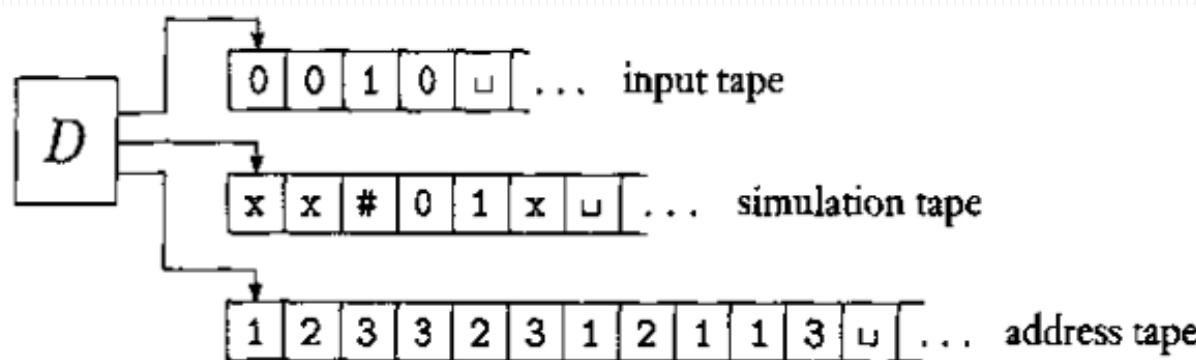


- Every node in the tree have at most b children, where b is the size of the largest set of possible choices given by N 's transition function.
- To every children in the tree we assign **an address** that is a string over the alphabet $\Sigma_b = \{1, 2, \dots, b\}$
- We assign the address 231 to the node we arrive at by starting at the root, going to its 2nd child, ...

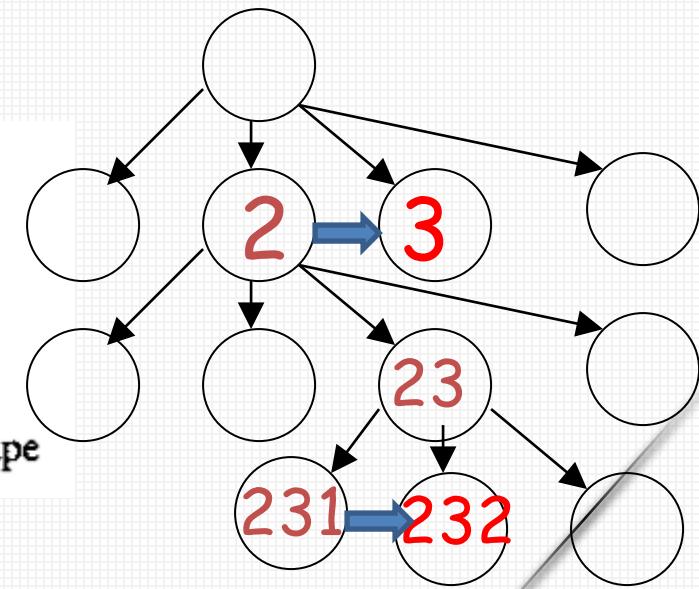
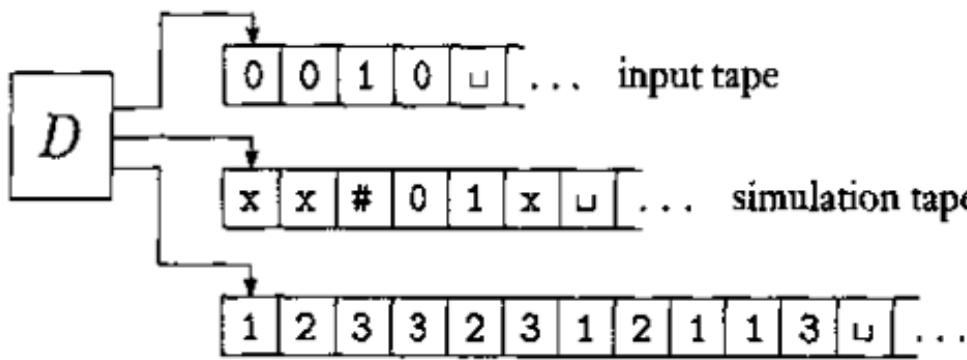


Description of TM D

- 1. Initially tape 1 contains the input w, and tape 2 and 3 are empty
- 2. Copy tape 1 to tape 2
- 3. Use tape 2 to simulate N with input w on one branch of its nondeterministic computation. Before N, consult **next symbol on tape 3** to determine which choice to make among those N's transition function.



- If no more symbol remain on tape 3 or if this deterministic choice is invalid, go to stage 4. Also go to stage 4 if a **rejecting** configuration happened. If an accepting configuration is encountered, **accept** the input
- 4. Replace the string on tape 3 with the **lexicographically next string** to simulate the next branch of N's computation by going to stage 2. (字典序)



- A language is Turing-recognizable if and only if some nondeterministic Turing machine recognizes it.
- Proof: Any deterministic Turing machine is automatically a nondeterministic Turing machine, and so one direction of this theorem follows immediately.

A language is **Turing-recognizable** if and only if some nondeterministic Turing machine recognizes it.

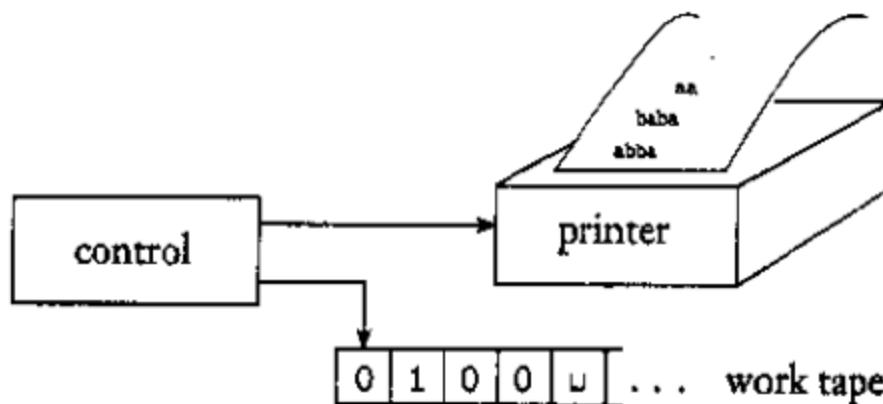
Proof.

Any deterministic TM is automatically a nondeterministic TM, and so one direction of this theorem follows immediately. The other direction we have already proved.

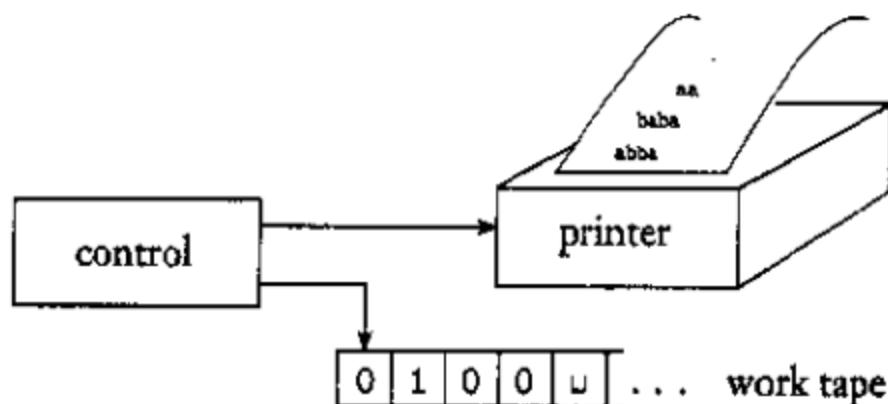
Corollary:

A language is **decidable** if and only if some nondeterministic Turing machine decides it.

- Term **Recursively enumerable language** for **Turing-recognizable language**
- An **enumerator** is a Turing machine with an attached printer. Every time Turing machine wants to add a string to the list, it sends the string to the printer.



- Enumerator E starts with a blank input tape.
- If the enumerator doesn't halt, it may print an infinite list of strings.
- The **language** enumerated by E is the collection of all the strings that is eventually prints



Theorem

A language is Turing-recognizable if and only if some enumerator enumerates it

- If we have an enumerator E that enumerates a language A , A TM M recognizes A .
- The TM M works: “on input w :
 - 1. Run E . Every time that E outputs a string, compare it with w .
 - 2. If w ever appears in the output of E , accept.”
- Clearly, Accept those strings that appear on E 's list.

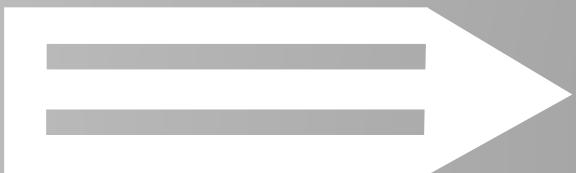
- ▶ If TM M recognizes a language A, we can construct the following enumerator E for A.
- ▶ Say that s_1, s_2, s_3, \dots is list of all possible strings in Σ^* .
- ▶ E=“Ignore the input.
 - Repeat the following for $i=1,2,3\dots$
 - Run M for i steps on each input, s_1, s_2, \dots, s_j .
 - If any computations accept, print out the corresponding s_j .”
- ▶ If M accepts a particular string s, eventually it will appear on the list generated by E.

- Pascal \Leftrightarrow Lisp
- Two computational models that satisfy certain reasonable requirements can simulate one another and hence are equivalent power.

- ✓ Variations of the Standard Model
- ✓ Simulation
- ✓ Turing Machines with Stay-Option
- ✓ Multitape Turing Machines
- ✓ Nondeterministic Turing Machines
- ✓ Enumerators

Theory of Computation

Definition of Algorithm



王轩

Wang Xuan

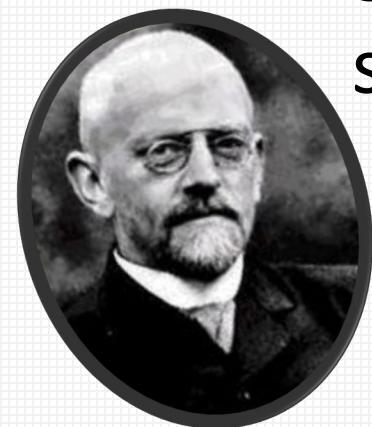
- Definition of Algorithm
- Church-Turing thesis
- Terminology
- Format of the problem

Definition of Algorithm

- Informally, An algorithm is any well-defined **computational procedure** that takes some values, as input and produces some values, as output.
- We can also view an algorithm as a tool for solving a well-specified **computational problem**.

Example----Hilbert's Problems

- Polynomial
 - $6x^3yz^2+2xy^2-x^3-10$
- Hilbert's tenth problem was to devise "a process according to which it can be determined by a finite number of operations" (**algorithm**) that tests whether a polynomial has an integer root.
- Some polynomials have an integer root and some do not.



Church-Turing thesis

- Church define algorithms using a notational system called the λ -calculus
- ▶ Lambda calculus is a formal system in mathematical logic and computer science for expressing computation by way of variable binding and substitution. It was first formulated by Alonzo Church.
- ▶ Lambda calculus has played an important role in the development of the theory of programming languages, including Lisp, ML, Haskell, etc.



Alonzo Church

The **lambda calculus** was introduced by mathematician **Alonzo Church** in the 1930s as part of an investigation into the foundations of mathematics. The **original system** was shown to be **logically inconsistent** in 1935 when Stephen Kleene and J. B. Rosser developed the Kleene–Rosser paradox.

Subsequently, in 1936 Church isolated and published just the portion relevant to computation, what is now called the **untyped lambda calculus**. In 1940, he also introduced a computationally weaker, but **logically consistent system**, known as **the simply typed lambda calculus**.

Until the 1960s when its relation to programming languages was clarified, the λ -calculus was only a **formalism**. Thanks to Richard Montague and other linguists' applications in **the semantics of natural language**, the λ -calculus has begun to enjoy a respectable place in both linguistics and computer science.

Alonzo designed Lambda calculus to represent complex computational processes in a general formal way.

Lambda calculus uses a very simple symbolic system :

{ λ , ., (,)} and variables

It can represent the calculation process of all Turing computable problems.

Example :

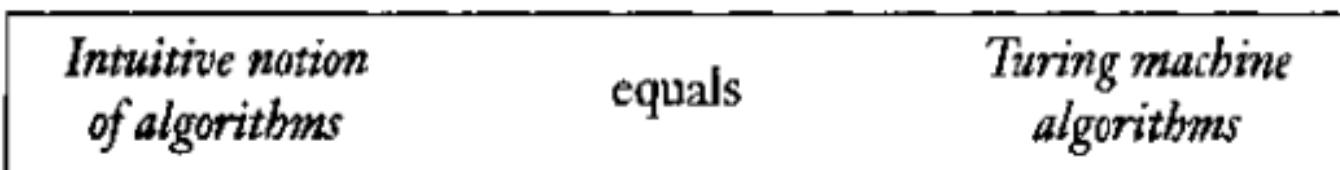
$$f(x) = x \longrightarrow \lambda x. x$$

$$f(x) = x^2 - 1 \longrightarrow \lambda x. (x^2 - 1)$$

$$f(x, y) = x + y \longrightarrow \lambda x. (\lambda y. (x+y))$$

Church-Turing thesis

- Turing define algorithms with Turing machine
- These two definitions are equivalent.
- This connection between the informal notion of algorithm and the precise definition is called Church-Turing thesis



Review—Turing-recognizable/decidable

Definition:

Call a language **Turing Recognizable** if some Turing machine recognizes it.

Call a language **Turing-decidable** or simply **decidable** if some Turing machine decides it.

- In 1970, Yuri Matijasevic building on work of Martin Davis, Hilary Putnam, and Julia Robinson, showed that no algorithm exists for testing whether a polynomial has integral roots.
- $D=\{p \mid p \text{ is polynomial with an integral root}\}$
- Hilbert's tenth problem asks in essence whether the set D is **decidable**.
- D is **Turing-recognizable, but undecidable**.

Example

- $D_1 = \{p \mid p \text{ is a polynomial over } x \text{ with an integral root}\}$
- TM M_1 that recognizes D_1 :
- $M_1 = \text{"The input is a polynomial } p \text{ over variables } x.$
 - Evaluate } p \text{ with } x \text{ set successively to the values } 0, 1, -1, 2, -2, 3, -3, \dots \text{ If at any point the polynomial evaluates to 0, accept."}
- M_1 is a recognizer not decider.

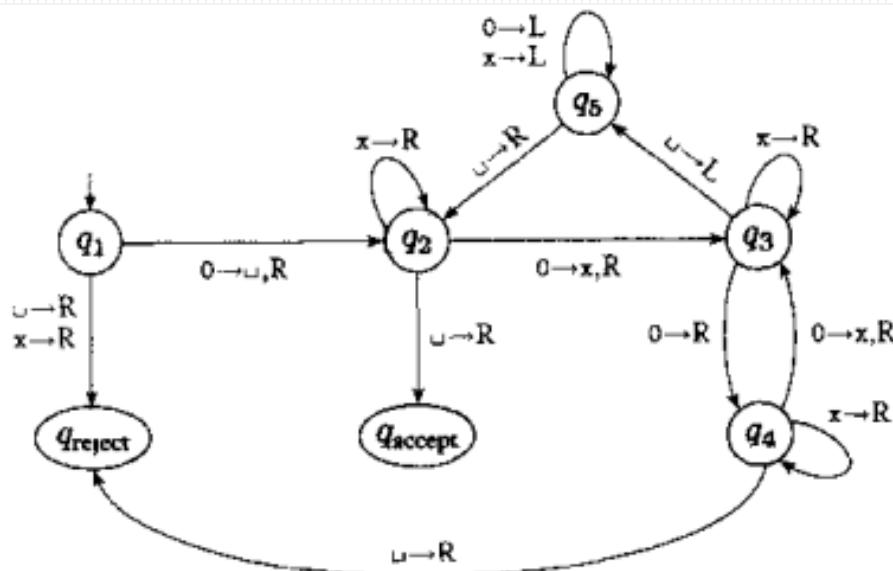
- ▶ Convert M_1 to be decider for D_1 because we can calculate bounds with which the roots of a single variable polynomial must lie and restrict the search to these bounds.

$$\pm k \frac{C_{\max}}{C_1}$$

where k is the number of terms in the polynomial, C_{\max} is the coefficient with largest absolute value, and C_1 is the coefficient of the highest order term.

- ▶ Yuri Matijasevic's theorem shows that calculating such bounds for multivariable polynomials is impossible.

- Turing machine is just a precise model for the definition of **algorithm**.
- **Three** possibilities to describe Turing machine algorithm.
- **1. Formal description:** states, transition functions, etc



► 2. **Implementation description:** use natural language to describe the way that Turing machine moves its head and the way that it stores data on its tape

► **Example:**

M = “ On input string w:

1. Sweep left to right across the tape, crossing off **every other 0**
2. If in stage 1 the tape contained a single 0, accept.
3. If in stage 1 the tape contained more than a single 0 and the number of 0s was odd, reject
4. Return the head to the left-hand end of the tape
5. Go to stage 1.

- **3. High-level description:** use natural language to describe an algorithm
- Example:
- $M = \text{"on input } w:\begin{aligned} & \text{-- 1. Run } E. \text{ Every time that } E \text{ outputs a string,} \\ & \text{compare it with } w. \\ & \text{-- 2. If } w \text{ ever appears in the output of } E, \text{ accept."}\end{aligned}$

High-level description

- Input for Turing machine:
 - Strings
 - Others: polynomials, graphs, grammars, automata, and any combination
 - Encode an object O into a string $\langle O \rangle$
 - Several objects O_1, O_2, \dots, O_k to $\langle O_1, O_2, \dots, O_k \rangle$

- We have come to a turning point in the study of the theory of computation. We continue to speak of Turing machines, but our real focus from now on is on **algorithms**.
- That is, the Turing machine merely serves as a precise model for the definition of algorithm.

Format of the problem

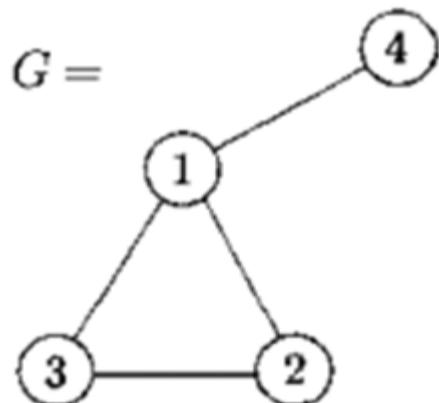
- Let A be the language consisting of the string s representing undirected graph that are connected.
- $A=\{<G> \mid G \text{ is a connected undirected graph}\}$

High-level Description

- $M = \text{"On input } \langle G \rangle, \text{ the encoding of a graph } G:$
 - 1. Select the first node of G and mark it.
 - 2. Repeat the following stage until no new nodes are marked
 - 3. For each node in G , mark it if it's attached by an edge to a node that is already marked.
 - 4. Scan all the nodes of G to determine whether they all are marked. If they are, accept; otherwise, reject.

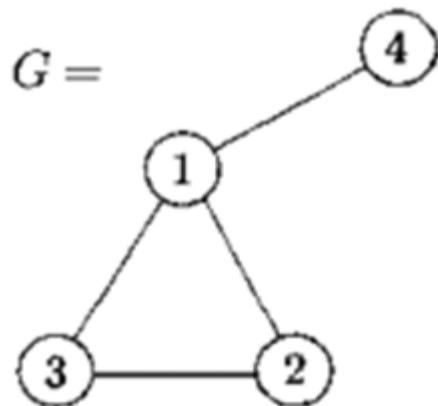
Implementation-level

- Encode the graph G as a string
 - A list of the nodes followed by a list of the edges



$\langle G \rangle = (1, 2, 3, 4) ((1, 2), (2, 3), (3, 1), (1, 4))$

- First check whether the input is a proper encoding of some graph
 - Two lists with decimal numbers
 - The first list contains no repetitions, and every node in second list must appear in the first list.



$$\langle G \rangle = (1, 2, 3, 4) ((1, 2), (2, 3), (3, 1), (1, 4))$$

$\langle G \rangle = (1, 2, 3, 4) ((1, 2), (2, 3), (3, 1), (1, 4))$

- ▶ For stage 1, M marks the first node with a dot on the leftmost digit. (1,2,3,4)
- ▶ For stage 2,
 - M scan the list of nodes to find an undotted node n1 and flag it (underlining).
 - Then M scan to find a dotted node n2 and underline it.
 - M scans the list edges, and for each edge tests whether n1 and n2 appears in the edge.
 - if they are, dots n1, goes to beginning of stage 2
 - If not, continue next edge
 - Move underline on n2 to next dotted node
 - move n1 as the next undotted node
 - Move on to stage 4 , If there are no more undotted nodes

$$\langle G \rangle = (1, 2, 3, 4) ((1, 2), (2, 3), (3, 1), (1, 4))$$

- ▶ For stage 2, $(\dot{1}, 2, 3, 4)$
 - M scan the list of nodes to find an undotted node n_1 and flag it (underlining). $(\dot{1}, \underline{2}, 3, 4)$
 - Then M scan to find a dotted node n_2 and underline it.
 - M scans the list edges, and for each edge tests whether n_1 and n_2 appears in the edge.
 - if they are, dots n_1 , goes to beginning of stage 2
 - If not, continue next edge
 - Move underline on n_2 to next dotted node
 - move n_1 as the next undotted node

$$\langle G \rangle = (1, 2, 3, 4) ((1, 2), (2, 3), (3, 1), (1, 4))$$

► For stage 2, $(\dot{1}, 2, 3, 4)$

- M scan the list of nodes to find an undotted node n1 and flag it (underlining). $(\dot{1}, \underline{2}, 3, 4)$ $(\dot{1}, \underline{2}, 3, 4)$
- Then M scan to find a dotted node n2 and underline it.
 - M scans the list edges, and for each edge tests whether n1 and n2 appears in the edge.
 - if they are, dots n1, goes to beginning of stage 2
 - If not, continue next edge
 - Move underline on n2 to next dotted node
 - move n1 as the next undotted node

$$\langle G \rangle = (1, 2, 3, 4) ((1, 2), (2, 3), (3, 1), (1, 4))$$

► For stage 2, $(\dot{1}, 2, 3, 4)$

- M scan the list of nodes to find an undotted node n1 and flag it (underlining). $(\dot{1}, \underline{2}, 3, 4)$ $(\dot{1}, \underline{2}, 3, 4)$
- Then M scan to find a dotted node n2 and underline it.
 - M scans the list edges, and for each edge tests whether n1 and n2 appears in the edge. $(1, 2)$
 - if they are, dots n1, goes to beginning of stage 2
 - If not, continue next edge
 - Move underline on n2 to next dotted node
 - move n1 as the next undotted node

$$\langle G \rangle = (1, 2, 3, 4) ((1, 2), (2, 3), (3, 1), (1, 4))$$

► For stage 2, $(\dot{1}, 2, 3, 4)$

- M scan the list of nodes to find an undotted node n1 and flag it (underlining). $(\dot{1}, \underline{2}, 3, 4)$ $(\dot{1}, \underline{2}, 3, 4)$
 - Then M scan to find a dotted node n2 and underline it.
 - M scans the list edges, and for each edge tests whether n1 and n2 appears in the edge. $(1, 2)$
 - if they are, dots n1, goes to beginning of stage 2
 - If not, continue next edge
 - Move underline on n2 to next dotted node
 - move n1 as the next undotted node

$$\langle G \rangle = (1, 2, 3, 4) ((1, 2), (2, 3), (3, 1), (1, 4))$$

► For stage 2, $(\dot{1}, \dot{2}, 3, 4)$

- M scan the list of nodes to find an undotted node n1 and flag it (underlining). $(\dot{1}, \dot{2}, \underline{3}, 4)$
 - Then M scan to find a dotted node n2 and underline it.
 - M scans the list edges, and for each edge tests whether n1 and n2 appears in the edge.
 - if they are, dots n1, goes to beginning of stage 2
 - If not, continue next edge
 - Move underline on n2 to next dotted node
 - move n1 as the next undotted node

$$\langle G \rangle = (1, 2, 3, 4) ((1, 2), (2, 3), (3, 1), (1, 4))$$

► For stage 2, $(\dot{1}, \dot{2}, 3, 4)$

- M scan the list of nodes to find an undotted node n1 and flag it (underlining). $(\dot{1}, \dot{2}, \underline{3}, 4)$
- Then M scan to find a dotted node n2 and underline it.
 - M scans the list edges, and for each edge tests whether n1 and n2 appears in the edge.
 - if they are, dots n1, goes to beginning of stage 2
 - If not, continue next edge
 - Move underline on n2 to next dotted node
 - move n1 as the next undotted node

$$\langle G \rangle = (1, 2, 3, 4) ((1, 2), (2, 3), (3, 1), (1, 4))$$

► For stage 2, $(\dot{1}, \dot{2}, 3, 4)$

- M scan the list of nodes to find an undotted node n1 and flag it (underlining). $(\dot{1}, \dot{2}, \underline{3}, 4)$ $(\underline{\dot{1}}, \dot{2}, \underline{3}, 4)$
- Then M scan to find a dotted node n2 and underline it.
 - M scans the list edges, and for each edge tests whether n1 and n2 appears in the edge. $(1, 2)$
 - if they are, dots n1, goes to beginning of stage 2
 - If not, continue next edge
 - Move underline on n2 to next dotted node
 - move n1 as the next undotted node

$$\langle G \rangle = (1, 2, 3, 4) ((1, 2), (2, 3), (3, 1), (1, 4))$$

► For stage 2, $(\dot{1}, \dot{2}, 3, 4)$

- M scan the list of nodes to find an undotted node n1 and flag it (underlining). $(\dot{1}, \dot{2}, \underline{3}, 4)$ $(\underline{\dot{1}}, \dot{2}, \underline{3}, 4)$
- Then M scan to find a dotted node n2 and underline it.
 - M scans the list edges, and for each edge tests whether n1 and n2 appears in the edge
 - if they are, dots n1, goes to beginning of stage 2
 - If not, continue next edge $(2, 3)$
 - Move underline on n2 to next dotted node
 - move n1 as the next undotted node

$$\langle G \rangle = (1, 2, 3, 4) ((1, 2), (2, 3), (3, 1), (1, 4))$$

► For stage 2, $(\dot{1}, \dot{2}, 3, 4)$

- M scan the list of nodes to find an undotted node n1 and flag it (underlining). $(\dot{1}, \dot{2}, \underline{3}, 4)$ $(\underline{\dot{1}}, \dot{2}, \underline{3}, 4)$
- Then M scan to find a dotted node n2 and underline it.
 - M scans the list edges, and for each edge tests whether n1 and n2 appears in the edge
 - if they are, dots n1, goes to beginning of stage 2
 - If not, continue next edge $(3, 1)$
 - Move underline on n2 to next dotted node
 - move n1 as the next undotted node

$$\langle G \rangle = (1, 2, 3, 4) ((1, 2), (2, 3), (3, 1), (1, 4))$$

► For stage 2, $(\dot{1}, \dot{2}, 3, 4)$

- M scan the list of nodes to find an undotted node n1 and flag it (underlining). $(\dot{1}, \dot{2}, \underline{3}, 4)$ $(\underline{1}, \dot{2}, \underline{3}, 4)$
- Then M scan to find a dotted node n2 and underline it.
 - M scans the list edges, and for each edge tests whether n1 and n2 appears in the edge
 - if they are, dots n1, goes to beginning of stage 2 $(\dot{1}, \dot{2}, \dot{3}, 4)$
 - If not, continue next edge $(3, 1)$
 - Move underline on n2 to next dotted node
 - move n1 as the next undotted node

$$\langle G \rangle = (1, 2, 3, 4) ((1, 2), (2, 3), (3, 1), (1, 4))$$

► For stage 2, $(\dot{1}, \dot{2}, \dot{3}, 4)$

- M scan the list of nodes to find an undotted node n1 and flag it (underlining). $(\dot{1}, \dot{2}, \dot{3}, \underline{4})$
 - Then M scan to find a dotted node n2 and underline it.
 - M scans the list edges, and for each edge tests whether n1 and n2 appears in the edge.
 - if they are, dots n1, goes to beginning of stage 2
 - If not, continue next edge
 - Move underline on n2 to next dotted node
 - move n1 as the next undotted node

$$\langle G \rangle = (1, 2, 3, 4) ((1, 2), (2, 3), (3, 1), (1, 4))$$

► For stage 2, $(\dot{1}, \dot{2}, \dot{3}, 4)$

- M scan the list of nodes to find an undotted node n1 and flag it (underlining). $(\dot{1}, \dot{2}, \dot{3}, \underline{4})$ $(\underline{1}, \dot{2}, \dot{3}, \underline{4})$
- Then M scan to find a dotted node n2 and underline it.
 - M scans the list edges, and for each edge tests whether n1 and n2 appears in the edge. $(1, 2)$
 - if they are, dots n1, goes to beginning of stage 2
 - If not, continue next edge
 - Move underline on n2 to next dotted node
 - move n1 as the next undotted node

$$\langle G \rangle = (1, 2, 3, 4) ((1, 2), (2, 3), (3, 1), (1, 4))$$

► For stage 2, $(\dot{1}, \dot{2}, \dot{3}, 4)$

- M scan the list of nodes to find an undotted node n1 and flag it (underlining). $(\dot{1}, \dot{2}, \dot{3}, \underline{4})$ $(\underline{1}, \dot{2}, \dot{3}, \underline{4})$
- Then M scan to find a dotted node n2 and underline it.
 - M scans the list edges, and for each edge tests whether n1 and n2 appears in the edge.
 - if they are, dots n1, goes to beginning of stage 2
 - If not, continue next edge $(2, 3)$
 - Move underline on n2 to next dotted node
 - move n1 as the next undotted node

$$\langle G \rangle = (1, 2, 3, 4) ((1, 2), (2, 3), (3, 1), (1, 4))$$

► For stage 2, $(\dot{1}, \dot{2}, \dot{3}, 4)$

- M scan the list of nodes to find an undotted node n1 and flag it (underlining). $(\dot{1}, \dot{2}, \dot{3}, \underline{4})$ $(\underline{\dot{1}}, \dot{2}, \dot{3}, \underline{4})$
- Then M scan to find a dotted node n2 and underline it.
 - M scans the list edges, and for each edge tests whether n1 and n2 appears in the edge.
 - if they are, dots n1, goes to beginning of stage 2
 - If not, continue next edge $(3, 1)$
 - Move underline on n2 to next dotted node
 - move n1 as the next undotted node

$$\langle G \rangle = (1, 2, 3, 4) ((1, 2), (2, 3), (3, 1), (1, 4))$$

► For stage 2, $(\dot{1}, \dot{2}, \dot{3}, 4)$

- M scan the list of nodes to find an undotted node n1 and flag it (underlining). $(\dot{1}, \dot{2}, \dot{3}, \underline{4})$ $(\underline{1}, \dot{2}, \dot{3}, \dot{4})$
- Then M scan to find a dotted node n2 and underline it.
 - M scans the list edges, and for each edge tests whether n1 and n2 appears in the edge.
 - if they are, dots n1, goes to beginning of stage 2 $(\dot{1}, \dot{2}, \dot{3}, \dot{4})$
 - If not, continue next edge $(1, 4)$
 - Move underline on n2 to next dotted node
 - move n1 as the next undotted node

$$\langle G \rangle = (1, 2, 3, 4) ((1, 2), (2, 3), (3, 1), (1, 4))$$

► For stage 2, $(\dot{1}, \dot{2}, \dot{3}, 4)$

- M scan the list of nodes to find an undotted node n1 and flag it (underlining). $(\dot{1}, \dot{2}, \dot{3}, \underline{4})$ $(\underline{1}, \dot{2}, \dot{3}, \underline{4})$
- Then M scan to find a dotted node n2 and underline it.
 - M scans the list edges, and for each edge tests whether n1 and n2 appears in the edge.
 - if they are, dots n1, goes to beginning of stage 2
 - If not, continue next edge $(1, 4)$
 - Move underline on n2 to next dotted node
 - move n1 as the next undotted node

- ▶ Stage 4, M scan the list of nodes to determine whether all are dotted. If they are, it enters the accept state; otherwise it enters the reject state.

▶ completion

- ✓ Definition of Algorithm
- ✓ Church-Turing thesis
- ✓ Terminology
- ✓ Format of the problem

Theory of Computation

Decidable Languages



王轩

Wang Xuan

- ▶ Decidable Languages
- ▶ Decidable Problems Concerning Regular language

- Church-Turing thesis:

Relation between the informal notion of algorithm and the precise definition

- Turing-decidable

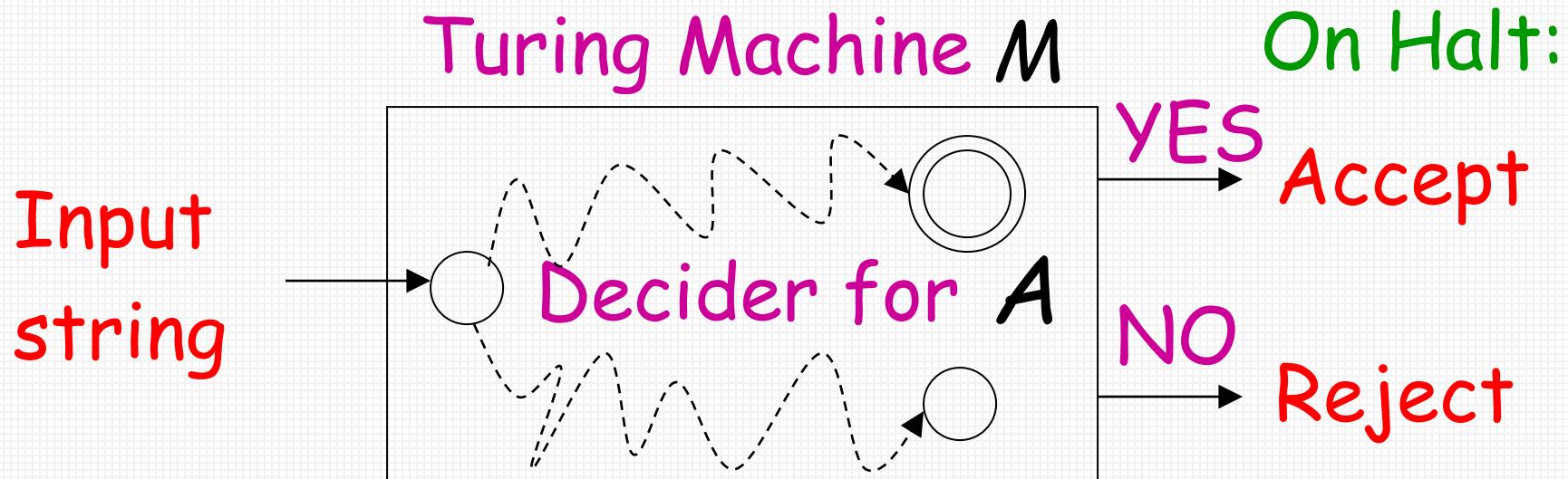
Call a language Turing-decidable or simply decidable if some Turing machine decides it.

Decidable Languages

Recall that:

A language A is decidable,
if there is a Turing machine M (decider)
that accepts the language A and
halts on every input string.

Decision



A computational problem is decidable
if the corresponding language is decidable

We also say that the problem is solvable

- The acceptance problem for DFAs of testing whether a particular deterministic finite automaton accepts a given string can be expressed as a language, A_{DFA}
- $A_{DFA} = \{<B, w> \mid B \text{ is a DFA that accepts input string } w\}$
- The problem of testing whether a given finite automaton accepts a given string is decidable.

A_{DFA} is a decidable language

- Idea: construct a TM M that decides A_{DFA}
- $M =$ “on input $\langle B, w \rangle$, where B is a DFA and w is a string:
 - 1. Simulate B on input w .
 - 2. if the simulation ends in an accept state, accept.
If it ends in a nonaccepting state, reject.”

Proof (implementation details)

- Examine the input $\langle B, w \rangle$
 - B is a DFA, five-tuple.
 - w is a string
- M carries out the simulation
 - Keep track B 's current state and current position.
 - Update according to transition function.
 - When M finished processing the last symbol of w ,
 M accepts if B is in an accepting state; M rejects
the input if B is in nonaccepting state.

A_{NFA} is a decidable language

- $A_{NFA} = \{<B, w> \mid B \text{ is a NFA that accepts input string } w\}$
- Proof:
 - Present a TM N that decides A_{NFA} .
 - N use M as a subroutine.
 - N first converts the NFA it receives as input to a DFA before passing it to M .
 - $N =$ “on input $<B, w>$, where B is a NFA and w is a string:
 - 1. Convert NFA B to equivalent DFA C .
 - 2. Run TM M on input $<C, w>$.
 - 3. if M accepts, accept. Otherwise, reject.”

A_{REX} is a decidable language

- $A_{REX} = \{<R,w> \mid R \text{ is regular expression that generates string } w\}$
- Proof:
 - P=“ on input $<R,w>$, where R is a regular expression and w is a string:
 - 1. Convert regular expression R to an equivalent NFA A.
 - 2. Run TM N on input $<A,w>$.
 - 3. if N accepts, accept. Otherwise, reject.”

E_{DFA} is decidable language

- Emptiness testing:

Whether a finite automaton doesn't accept any strings at all.

- $E_{DFA} = \{< A > \mid A \text{ is a DFA and } L(A) = \emptyset\}$
- $T =$ “on input $< A >$, where A is a DFA:
 - 1. Mark the start state in A
 - 2. Repeat until no new states get marked:
 - 3. Mark any state that has a transition coming into it from any state that is already marked.
 - 4. if no accept state is marked, accept. Otherwise, reject.”

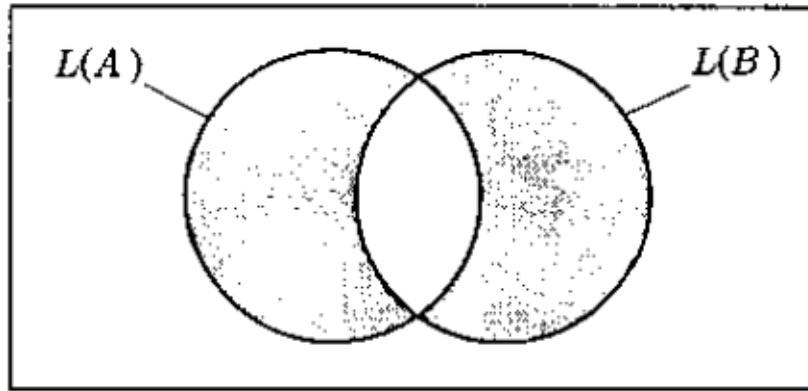
EQ_{DFA} is decidable language

- $\text{EQ}_{\text{DFA}} = \{\langle A, B \rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$
- Determine whether two DFAs recognize the same language is decidable
- Proof:
- Construct a new DFA C from A and B, called the symmetric difference of $L(A)$ and $L(B)$

$$L(C) = (L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap L(B))$$

- If A and B recognize the same language, C will accept nothing.

$$L(C) = (L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap L(B))$$



- C is still DFA (closed)
- Test whether $L(C)$ is empty. If it is, $L(A)$ and $L(B)$ must be equal

- $F = " \text{on input } \langle A, B \rangle, \text{ where } A, B \text{ are DFAs:}$
 - 1. Construct DFA C as described.
 - 2. Run TM T on input $\langle C \rangle$
 - 3. If T accepts, accept. If T rejects, reject."

- ✓ Decidable Languages
- ✓ Decidable Problems Concerning Regular language

Theory of Computation

Decidable Problems Concerning Context Free Languages



王轩

Wang Xuan

- A_{CFG}
- E_{CFG}
- Every CFG
- EQ_{CFG}
- Relationship

Context-free languages

- $A_{CFG} = \{<G, w> \mid G \text{ is a CFG that generates string } w\}$
- A_{CFG} is decidable language
- Idea 1 :
 - Go through ~~all~~ derivations. If G doesn't generate w , this algorithm **maybe never halt**. Only Turing-recognizable

- Idea 2 :

In Chomsky normal form, any derivation of w has $2n-1$ steps, where n is the length of w . Check only derivations with $2n-1$ steps to determine whether G generates w would be sufficient.

$$A \rightarrow BC$$

$$A \rightarrow a$$

Chomsky Normal Form



Chomsky

A_{CFG} is decidable language

- Proof
- $S = " \text{on input } \langle G, w \rangle, \text{ where } G \text{ are CFG and } w \text{ is a string:}$
 - 1. Convert G to an equivalent grammar in Chomsky normal form.
 - 2. List all derivation with $2n-1$ steps, where n is the length of w , except if $n=0$, then instead list all derivations with 1 step.
 - 3. If one of these derivations generate w , accept; if not, reject."
- **A_{PDA} is decidable language**

E_{CFG} is a decidable language

- $E_{CFG} = \{<G> \mid G \text{ is a CFG and } L(G) = \emptyset\}$
- Idea 1:
 - Going through all possible w's doesn't work 
- Idea 2:
 - We need to determine whether the start variable can generate a string of terminals
 - Mark all terminal symbols
 - Scan all rules of the grammar, the right of production is all marked, then mark the left variable
 - Continue until it cannot mark any more variables

E_{CFG} is a decidable language

- Proof:
- $R = \text{" on input } \langle G \rangle, \text{ where } G \text{ are CFG:}$
 - 1. Mark all terminal symbols in G .
 - 2. Repeat until no new variables get marked:
 - Mark any variable A where G has a rule $A \rightarrow U_1 U_2 \dots U_k$ and each U_1, \dots, U_k has already been marked
 - 3. If the start variable is not marked, accept; otherwise, reject."

Every context-free language is decidable

- Bad idea
 - Convert a PDA for a CFL A directly into a TM.
Simulating a stack with TM's tape is easy.
 - However, some branches of PDA may go forever(loop). The simulating TM may also have some non-halting branches. (not decider)
- Good one:

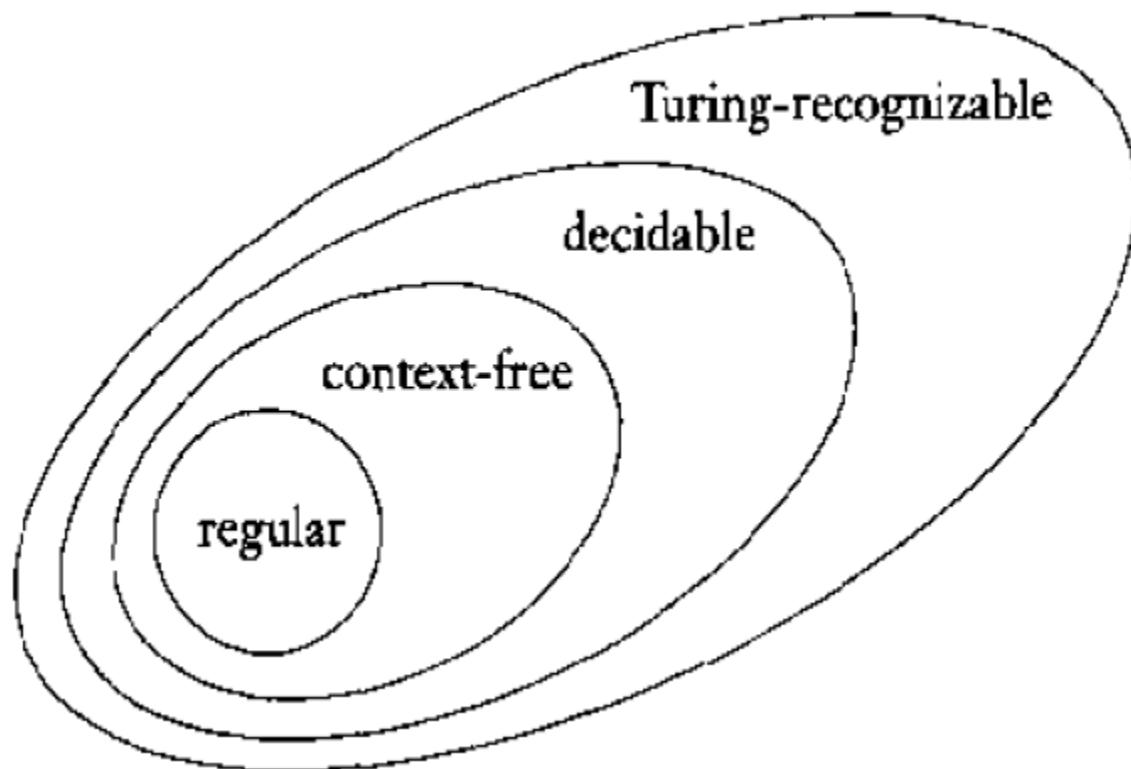
Using the decider to decide $A_{CFG} \text{----} S$

- Let G be a CFG for A and design a TM M_G that decides A . We build a copy of G into M_G .
- $M_G = \text{"On input } w:$
 - 1. Run TM S on input $\langle G, w \rangle$.
 - 2. If this machine accepts, accept; if it rejects, reject."

EQ_{CFG} is a not decidable

- $EQ_{CFG} = \{<G, H> \mid G \text{ and } M \text{ are CFGs and } L(G) = L(H)\}$
- EQ_{DFA} is decidable
- CFG is not closed under complementation and intersection. So we can't use the same method as EQ_{DFA}
- In fact, EQ_{DFA} is not decidable. The technique for proving so is presented in the next chapter.

- We have learned regular, context free, decidable, and Turing-recognizable so far. The figure depicts this relationship.



- ✓ A_{CFG}
- ✓ E_{CFG}
- ✓ EQ_{CFG}
- ✓ Every CFG
- ✓ Relation

Theory of Computation

Lesson 8-1

The Halting Problem



王 轩
Wang
Xuan

- ▶ Unsolvable
- ▶ Undecidable Languages
- ▶ A_{TM} is Turing-recognizable
- ▶ Universal Turing Machine
- ▶ The Halting Problem is Undecidable
- ▶ \mathbb{R} is uncountable
- ▶ The set of all languages is uncountable
- ▶ Diagonalization method
- ▶ Turing-unrecognizable Language
- ▶ \overline{A}_{TM} is not Turing-recognizable

- Several ordinary problems are **computationally unsolvable**
- In this section and Chapter 5 you will encounter several computationally unsolvable problems.
- Our objectives are to help you develop a feel for the types of problems that are unsolvable and to learn techniques for proving unsolvability.

undecidable language = not decidable language

There is no decider:

There is no Turing Machine which accepts the language and makes a decision (**halts**) for **every** input string

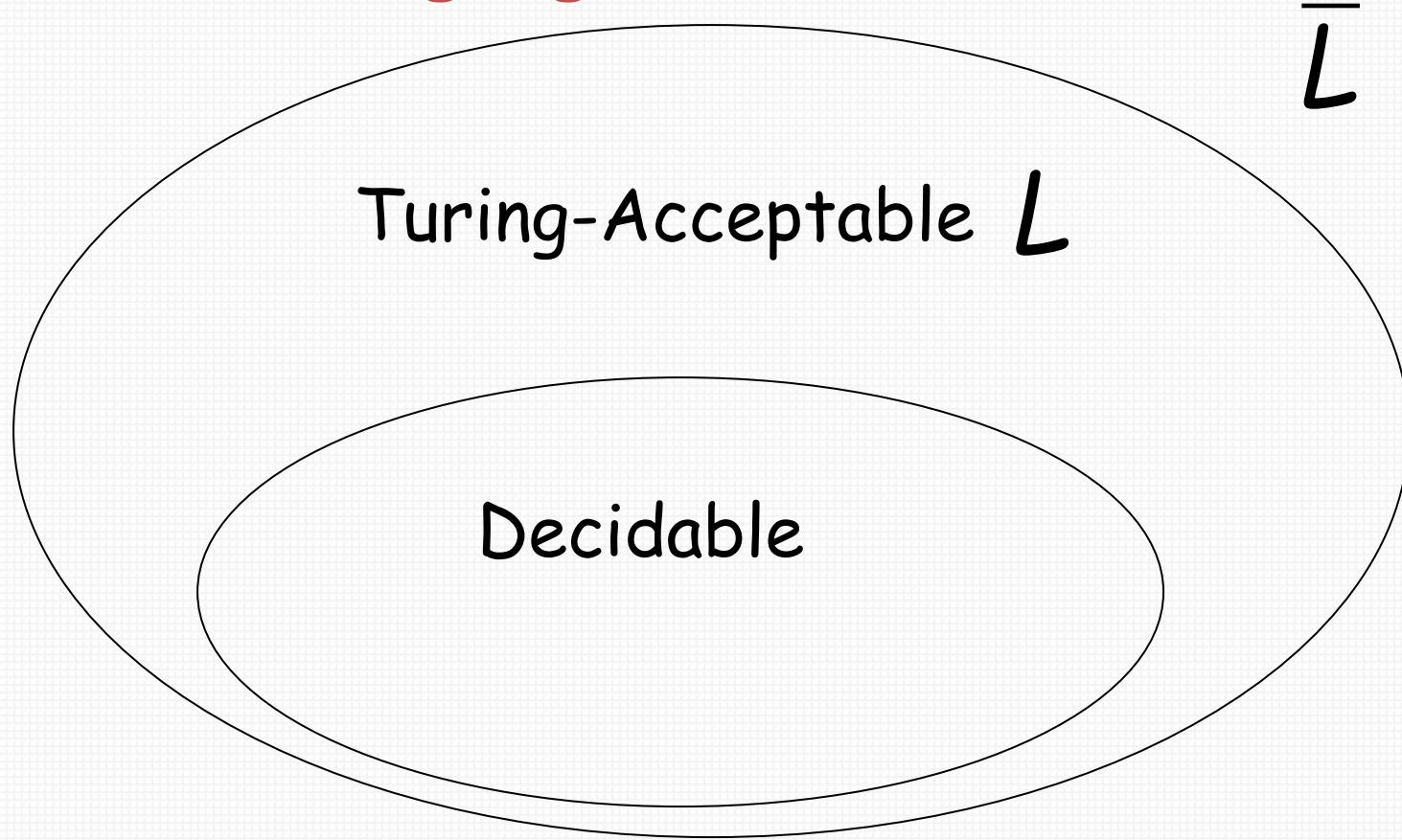
(machine may make decision for some input strings)

For an **undecidable** language, the corresponding problem is **undecidable (unsolvable)**:

there is no Turing Machine (Algorithm) that gives an answer (yes or no) for every input instance

(answer may be given for some input instances)

We have shown before that there are undecidable languages:



L is Turing-Acceptable and undecidable

Input: • Turing Machine M
• String w

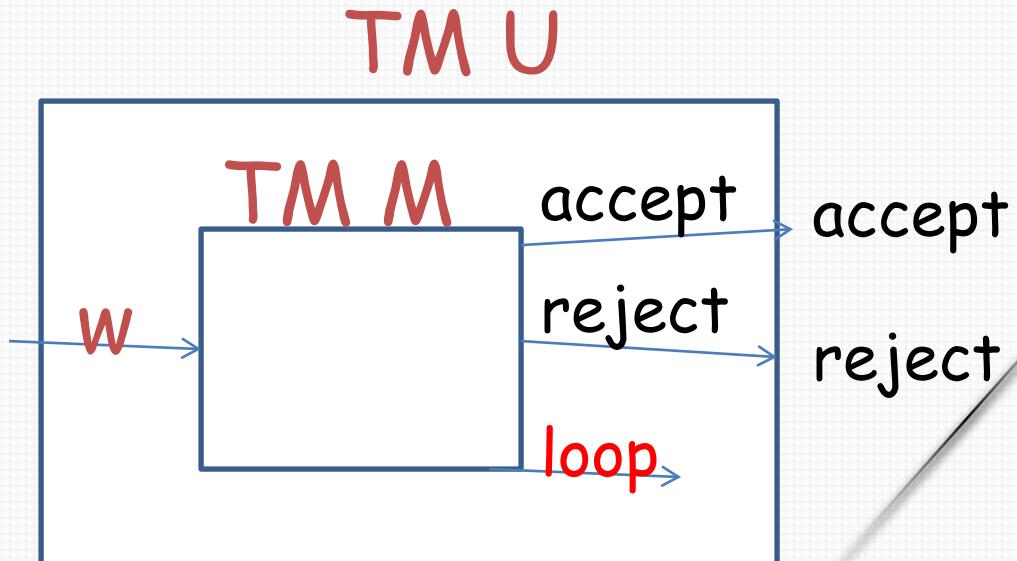
Question: Does M accept w ?
 $w \in L(M)$?

Corresponding language:

$A_{TM} = \{\langle M, w \rangle : M \text{ is a Turing machine and}$
 $M \text{ accepts string } w\}$

A_{TM} is Turing-recognizable

- Turing machine U **recognizes** A_{TM}
- U = “on input $\langle M, w \rangle$, where M is a TM and w is a string:
 - 1. Simulate M on input w .
 - 2. If M ever enters its accept state, accept; if M ever enters its reject state, reject.”



Halting Problem

- This machine loops on input $\langle M, w \rangle$ if M loops on w , which is why this machine does not decide A_{TM} .
- A_{TM} is sometimes called the halting problem.
- A_{TM} is undecidable and A_{TM} is recognizable. Thus recognizers are more powerful than deciders.

Universal Turing Machine

- ▶ Turing machine U is an example of Universal Turing Machine (U_{TM})
- ▶ U_{TM} is capable of **simulating** any other Turing machine from the description of that machine.
- ▶ It plays an important early role in stimulating the development of stored-program computers

The Halting Problem is Undecidable

- $A_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$.
- Proof **by contradiction**.
- Suppose that H is a decider for A_{TM} .

$$H(\langle M, w \rangle) = \begin{cases} \text{accept} & \text{if } M \text{ accepts } w \\ \text{reject} & \text{if } M \text{ does not accept } w. \end{cases}$$

- Construct Turing machine D with H as a subroutine. D calls H to determine what M does when the input to M is its own description $\langle M \rangle$.

- The description of D

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

1. Run H on input $\langle M, \langle M \rangle \rangle$.
2. Output the opposite of what H outputs; that is, if H accepts, reject and if H rejects, accept."

- We obtain

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

- Run D with its own description $\langle D \rangle$. We get

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

- No matter what D does, it is forced to do the opposite.
 - D rejects $\langle D \rangle$ exactly when D accepts $\langle D \rangle$.
- There is a contradiction.

The Diagonalization Method

- Georg Cantor in 1873
- Measuring the sizes of infinite set
 - Finite sets is easy
 - Infinite sets:
 - Set of natural numbers {1,2,3...}
 - Set of even natural numbers {2,4,6...}
- Two finite sets have the same size if the elements of one set can be **paired** with the elements of the other set.
- Extending to infinite sets



- Sets A and B, function f from A to B
- f is **one-to-one**, if $f(a) \neq f(b)$ whenever $a \neq b$.
- f is **onto**, if for every $b \in B$ there is an $a \in A$ such that $f(a) = b$.
- A and B are the **same size** if there is a one-to-one, onto function $f: A \rightarrow B$.
- Function that is both one-to-one, and onto is called **correspondence**.

Example

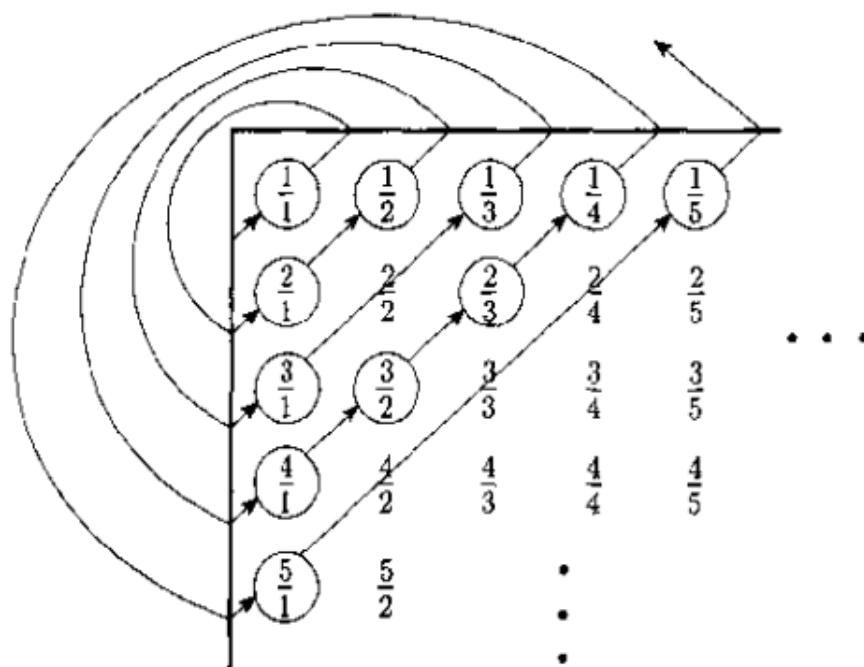
- ▶ Natural numbers $N \{1,2,3,\dots\}$ & Even natural numbers $\{2, 4, 6, \dots\}$
- ▶ Same size. Correspondence f map N to the set of even natural numbers is $f(n)=2n$.

n	$f(n)$
1	2
2	4
3	6
:	:

- ▶ A set A is countable if either it is finite or it has the same size as N .

Another Example

- $Q = [m/n \mid m, n \in N]$ be the set of positive rational numbers.
- We need to give a correspondence with N to show that Q is countable



- ▶ For some infinite sets no correspondence with N exists. Such sets are called **uncountable**.
- ▶ Example: The set of real numbers, R
- ▶ The number $\pi = 3.1415926\ldots$ and $\sqrt{2} = 1.4142135\ldots$

- Show that no correspondence exists between N and R.
- Proof by **contradiction**.
- Suppose that a correspondence f existed between N and R. We can find an x in R that is not paired with anything in N.
- Suppose

n	$f(n)$
1	3.14159...
2	55.55555...
3	0.12345...
4	0.50000...
:	:

Cont.

- Construct x by giving its decimal representation.
- To ensure $x \neq f(1)$, let the first digit of x be different from the first fractional digit 1 of $f(1)=3.\underline{1}4159$. Let it be 4
- To ensure $x \neq f(2)$, let the second digit of x be different from the second fractional digit 5 of $f(2)=55.\underline{5}55555$. Let it be 6.
- Continue ...

n	$f(n)$	
1	3. <u>1</u> 4159...	
2	55. <u>5</u> 5555...	
3	0.1 <u>2</u> 345...	$x = 0.4641\dots$
4	0.500 <u>0</u> ...	
:	:	

n	$f(n)$
1	3. <u>1</u> 4159...
2	55. <u>5</u> 5555...
3	0. <u>123</u> 45...
4	0.5 <u>000</u> 0...
:	:

$$x = 0.4641 \dots$$

- We get x is not $f(n)$ for any n because it differs from $f(n)$ in the n th fractional digit.

- Done

IDEA: Some languages are not Turing-recognizable

- 1. The set of all strings Σ^* is countable, for any alphabet Σ .

- 2. The set of all Turing machines is countable.

- 3. The set of all languages is uncountable.
- Thus, some languages are not recognized by any Turing machine.

- The set of all strings Σ^* is countable, for any alphabet Σ
 - Form a list of Σ^* by writing all strings of length 0, length 1, length 2, and so on
- The set of all Turing machines is countable
 - Each Turing machine M has an encoding into a string $\langle M \rangle$. **Turing Machine Encoding**
 - Remove the illegal encodings

The set of all languages is uncountable

- The set of all languages
 - Let B be the set of all infinite binary sequences.
- Let L be the set of all languages over alphabet Σ .
- Suppose L is a correspondence with B .
- Let $\Sigma^* = \{s_1, s_2, s_3, \dots\}$. Each language $A \in L$ has a unique sequence in B . The i^{th} bit of that sequence is a 1 if $s_i \in A$ and is a 0 if $s_i \notin A$. We call it characteristic sequence x_A of A .

- If A is the language of all strings starting with a 0 over the alphabet {0,1}, its characteristic sequence x_A would be

$$\begin{aligned}\Sigma^* &= \{ \epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots \}; \\ A &= \{ 0, 00, 01, 000, 001, \dots \}; \\ x_A &= 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ \dots .\end{aligned}$$

- The function $f: L \rightarrow B$, where $f(A)$ equals the characteristic sequence of A and is a correspondence. Therefore, as B is uncountable, L is uncountable as well.

- The set of all languages is uncountable.
- The set of all Turing machines is countable.
- Thus, some languages are not recognized by any Turing machine.

Turing-unrecognizable Language

- A language is **co-Turing-recognizable** if it is the complement of a Turing-recognizable language.
- **Theorem:** A language is decidable iff it is Turing-recognizable and co-Turing-recognizable.
- In other words, a language is decidable if and only if both it and its complement are Turing-recognizable.

Proof (two directions)

- If A is decidable, then both A and \bar{A} are Turing-recognizable.
- If both A and \bar{A} are Turing-recognizable, we let M_1 be the recognizer for A and M_2 be the recognizer for \bar{A} .
 - The Turing Machine M is a decider for A
 M = “On input w :
 1. Run both M_1 and M_2 on input w in parallel.
 2. If M_1 accepts, accept; if M_2 accepts, reject.”
 - Every string w is either A or \bar{A} . Therefore either M_1 or M_2 must accept w . Because M halts whenever M_1 or M_2 accept, M always halts and so it is decider.

\overline{A}_{TM} is not Turing-recognizable

- Proof
 - A_{TM} is Turing-recognizable.
 - If \overline{A}_{TM} is also Turing-recognizable, then A_{TM} is decidable.
 - We already know that A_{TM} is not decidable, thus \overline{A}_{TM} is not Turing-recognizable.

- ✓ Unsolvable
- ✓ Undecidable Languages
- ✓ A_{TM} is Turing-recognizable
- ✓ Universal Turing Machine
- ✓ The Halting Problem is Undecidable
- ✓ \mathbb{R} is uncountable
- ✓ The set of all languages is uncountable
- ✓ Diagonalization method
- ✓ Turing-unrecognizable Language
- ✓ \overline{A}_{TM} is not Turing-recognizable

Theory of Computation

Lesson 8-2

Turing Machine Encoding



王 轩
Wang
Xuan

Outline

- Some languages are not Turing-recognizable
- $\overline{A_{TM}}$ is not Turing-recognizable

IDEA: Some languages are not Turing-recognizable

- 1. The set of all strings Σ^* is countable, for any alphabet Σ .
- 2. The set of all Turing machines is countable.
- 3. The set of all languages is uncountable.
- Thus, some languages are not recognized by any Turing machine.

Proof

- 1. The set of all strings Σ^* is countable, for any alphabet Σ
 - Form a list of Σ^* by writing all strings of length 0, length 1, length 2, and so on.
 - Σ^* over the alphabet {0,1}

$$\Sigma^* = \{ \ \epsilon \ , \ 0 \ , \ 1 \ , \ 00 \ , \ 01 \ , \ 10 \ , \ 11 \ , 000, 001, \dots \ } ;$$

Proof

- 2. The set of all Turing machines is countable.
 - Each Turing machine M has an encoding into a string $\langle M \rangle$. **Turing Machine Encoding**

$$M = (Q, \Sigma, \Gamma, \delta, q_{start}, q_{accept}, q_{reject})$$

Turing Machine Encoding

- Q is the set of states $\{q_1, q_2, q_3, q_4, \dots\}$
- Σ is the input alphabet $\{a_1, a_2, a_3, a_4, \dots\}$
- Γ is the tape alphabet $\{b_1, b_2, b_3, b_4, \dots\}$
- $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is the transition function
- $q_{\text{start}} \in Q$ is the start state
- $q_{\text{accept}} \in Q$ is the accept state
- $q_{\text{reject}} \in Q$ is the reject state

Turing Machine Encoding

- Beginning encode $0^i 1 0^j 1 0^k 1 0^m 1 0^n 1 0^r 1 0^t$
- $|Q| = i$
- $|\Sigma| = j$
- $|\Gamma| = k$
- $q_{\text{start}} = q_m$
- $q_{\text{accept}} = q_n$
- $q_{\text{reject}} = q_r$
- Blank symbol \sqcup is b_t

separator

Turing Machine Encoding

- Transition function encode:
- $\delta_1 11 \delta_2 11 \delta_3 11 \dots 11 \delta_n$


separator
- For each transition function:
- $\delta(q_i, a_j) = (q_k, b_m, D_n) \rightarrow 0^i 1 0^j 1 0^k 1 0^m 1 0^n$
- D_n : L/R
 - R(D_1): $0^1 = 0$
 - L(D_2): $0^2 = 00$

Turing Machine Encoding

Input encode:

$a_i a_j \dots a_n \rightarrow 0^i 1 0^j 1 \dots 0^n$

Total turing machine encoding:

Beginning encode + 111 + Transition function encode + 111 + Input encode

separator

Each Turing machine M has an encoding into a string $\langle M \rangle$.

Example

$$M = (Q, \Sigma, \Gamma, \delta, q_{start}, q_{accept}, q_{reject})$$

- TM M=($\{q_1, q_2, q_3, q_4\}$, $\{a, b\}$, $\{a, b, \sqcup\}$, δ , q_4 , q_3 , q_2)
- δ :
 - $\delta_1(q_4, a) = (q_4, a, R)$
 - $\delta_2(q_4, b) = (q_1, b, R)$
 - $\delta_3(q_1, b) = (q_2, b, R)$
 - $\delta_4(q_2, b) = (q_3, \#, L)$

Beginning encode

- TM $M = (\{q_1, q_2, q_3, q_4\}, \{a, b\}, \{a, b, \sqcup\}, \delta, q_4, q_3, q_2)$
 - $|Q| = 4$
 - $|\Sigma| = 2$
 - $|\Gamma| = 3$
 - $q_{start} = q_4$
 - $q_{accept} = q_3$
 - $q_{reject} = q_2$
 - Blank symbol \sqcup is b_3
- Beginning encode is $0^4 10^2 10^3 10^4 10^3 10^2 10^3$

Transition function encode

– $\delta_1(q_4, a) = (q_4, a, R) \rightarrow 0^4 1 0^1 1 0^4 1 0^1 1 0^1$

– $\delta_2(q_4, b) = (q_1, b, R) \rightarrow 0^4 1 0^2 1 0^1 1 0^2 1 0^1$

– $\delta_3(q_1, b) = (q_2, b, R) \rightarrow 0^1 1 0^2 1 0^2 1 0^2 1 0^1$

– $\delta_4(q_2, b) = (q_3, \#, L) \rightarrow 0^2 1 0^2 1 0^3 1 0^3 1 0^2$

– $0^4 1 0^1 1 0^4 1 0^1 1 0^1 1 1 0^4 1 0^2 1 0^1 1 0^2 1 0^1 1 1 0^1 1 0^2 1 0^2 1 0^2 1$

$0^1 1 1 0^2 1 0^2 1 0^3 1 0^3 1 0^2$

separator

Input encode

- Input encode: ababa → 0¹10²10¹10²10¹

Total turing machine encoding:

- Beginning encode+111+Transition function
encode+111+Input encode
- $0^4 10^2 10^4 10^4 10^3 10^2 10^3 \text{111} 0^4 10^1 10^4 10^1 10^1 1$
 $10^4 10^2 10^1 10^2 10^2 110^1 10^2 10^2 10^2 10^2$
 $110^2 10^2 10^3 10^3 10^2 \text{111} 0^1 10^2 10^1 10^2 10^1$ 

separator
- ▶ Turing machine M has an encoding into a string <M>

Proof

- 3. The set of all languages is uncountable
 - Let B be the set of all infinite binary sequences.
 - Let L be the set of all languages over alphabet Σ .
 - Suppose L is a correspondence with B .
 - Let $\Sigma^* = \{s_1, s_2, s_3, \dots\}$. Each language $A \in L$ has a unique sequence in B . The i^{th} bit of that sequence is a ‘1’ if $s_i \in A$ and is a ‘0’ if $s_i \notin A$. We call it characteristic sequence x_A of A .

Cont.

- If A is the language of all strings starting with a 0 over the alphabet {0,1}, its character sequence x_A would be

$$\begin{aligned}\Sigma^* &= \{ \epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots \} ; \\ A &= \{ 0, 00, 01, 000, 001, \dots \} ; \\ x_A &= 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad \dots .\end{aligned}$$

- The function $f: L \rightarrow B$, where $f(A)$ equals the characteristic sequence of A and is a correspondence. Therefore, as B is uncountable, L is uncountable as well.

- The set of all languages is uncountable.
- The set of all Turing machines is countable.
- Thus, some languages are not recognized by any Turing machine.

Turing-unrecognizable Language

- A language is **co-Turing-recognizable** if it is the complement of a Turing-recognizable language.
- **Theorem:** A language is decidable iff it is Turing-recognizable and co-Turing-recognizable.
- In other words, a language is decidable if and only if both it and its complement are Turing-recognizable.

Proof (two directions)

- If A is decidable, then both A and \bar{A} are Turing-recognizable.
- If both A and \bar{A} are Turing-recognizable, we let M_1 be the recognizer for A and M_2 be the recognizer for \bar{A} .
 - The Turing Machine M is a decider for A
 M = “On input w :
 1. Run both M_1 and M_2 on input w in parallel.
 2. If M_1 accepts, accept; if M_2 accepts, reject.”
 - Every string w is either A or \bar{A} . Therefore either M_1 or M_2 must accept w . Because M halts whenever M_1 or M_2 accept, M always halts and so it is decider.

$\overline{A_{TM}}$ is not Turing-recognizable

- Proof
 - A_{TM} is Turing-recognizable.
 - If $\overline{A_{TM}}$ is also Turing-recognizable, then A_{TM} is decidable.
 - We already know that A_{TM} is not decidable, thus $\overline{A_{TM}}$ is not Turing-recognizable.

Review

- ✓ The set of all languages is uncountable
- ✓ Turing-unrecognizable Language
- ✓ $\overline{A_{TM}}$ is not Turing-recognizable

Theory of Computation

Lesson 9

Reducibility



王轩

Wang Xuan

- Introduction
- Undecidable Problems from Language Theory
- Computation history method

- Introduction
- Undecidable Problems from Language Theory
- Computation history method

Reducibility

Problem X is reduced to problem Y



If we can solve problem Y
then we can solve problem X

Theorem:

If: a: Language A is reduced to B

b: Language A is undecidable

Then: B is undecidable

Proof: Suppose B is decidable

Using the decider for B

build the decider for A

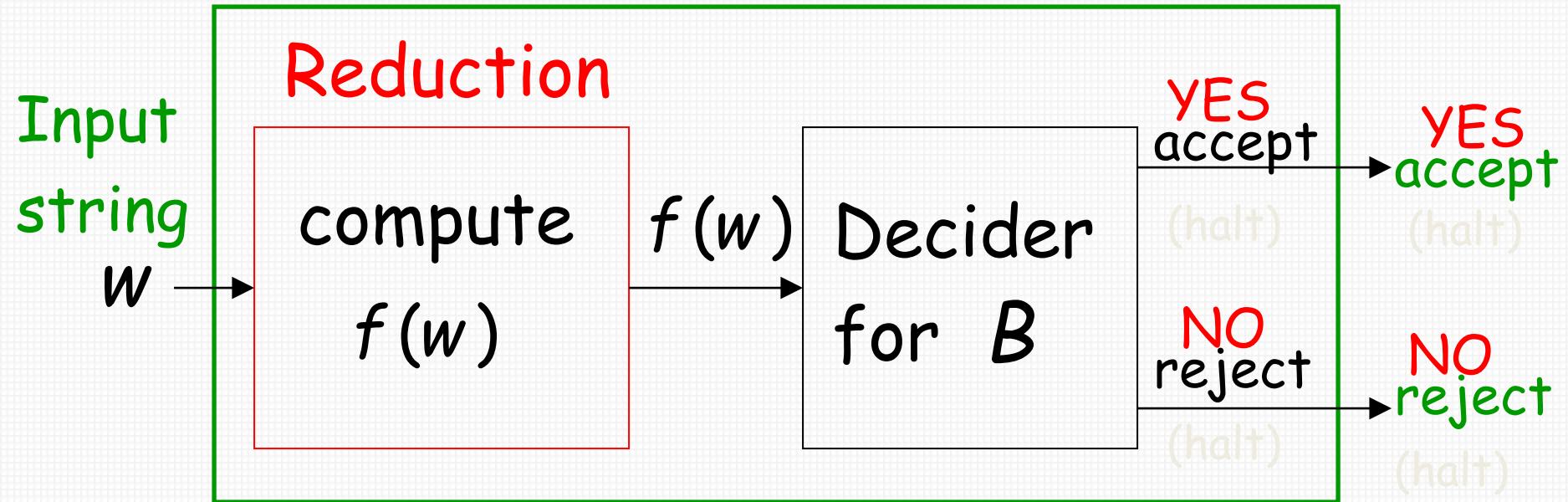
Contradiction!

Observation:

In order to prove
that some language B is undecidable
we only need to reduce a
known undecidable language A
to B

If B is decidable then we can build:

Decider for A



$$w \in A \Leftrightarrow f(w) \in B$$

CONTRADICTION!

END OF PROOF

- Introduction
- Undecidable Problems from Language Theory
- Computation history method

HALT_{TM} is undecidable

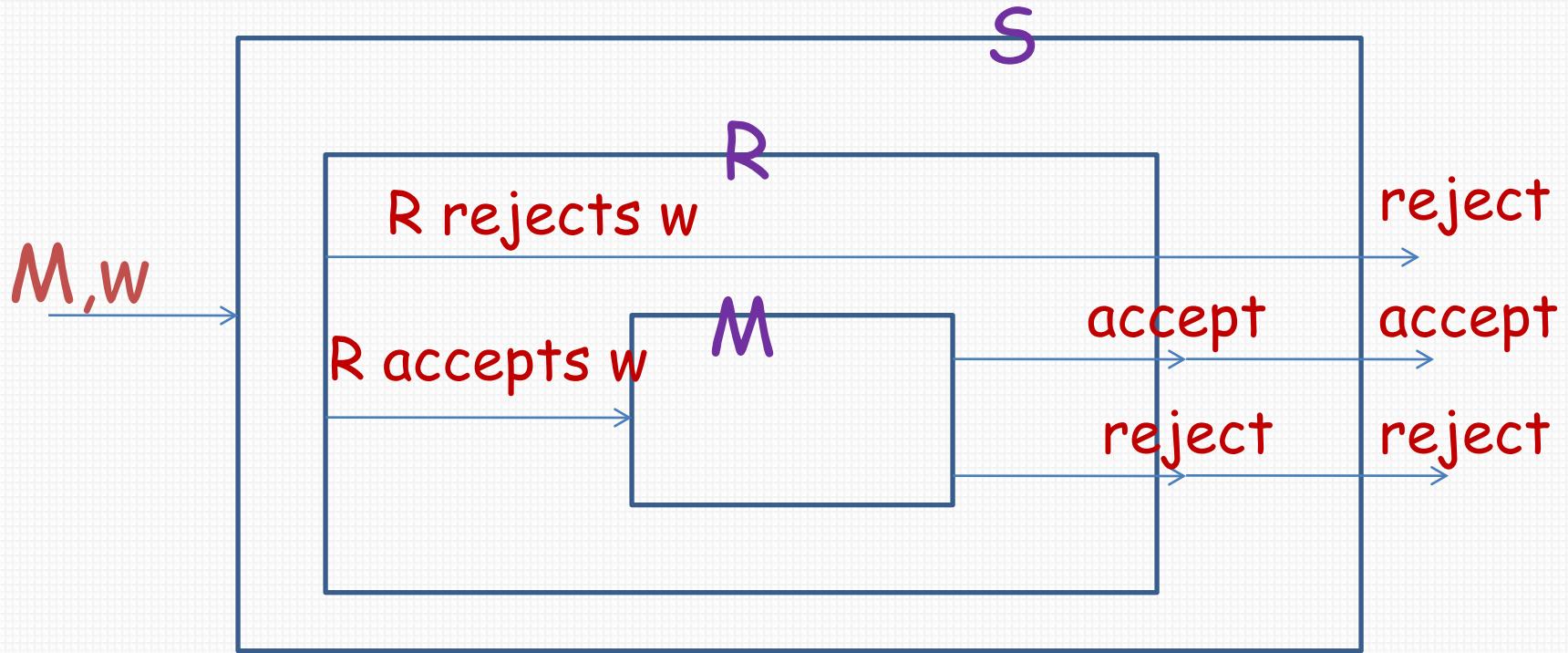
- $\text{HALT}_{\text{TM}} = \{\langle M, w \rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$
- Idea: Proof by contradiction.
 - Assume HALT_{TM} is decidable.
 - Prove that A_{TM} is reducible to HALT_{TM} . (important)
 - Because A_{TM} is undecidable. Contradiction.

- TM R decides HALT_{TM} . We construct TM S to decide A_{TM} .

S = “On input $\langle M, w \rangle$, an encoding of a TM M and a string w :

1. Run TM R on input $\langle M, w \rangle$.
2. If R rejects, *reject*.
3. If R accepts, simulate M on w until it halts.
4. If M has accepted, *accept*; if M has rejected, *reject*.”

- Because A_{TM} is undecidable, HALT_{TM} must be undecidable.

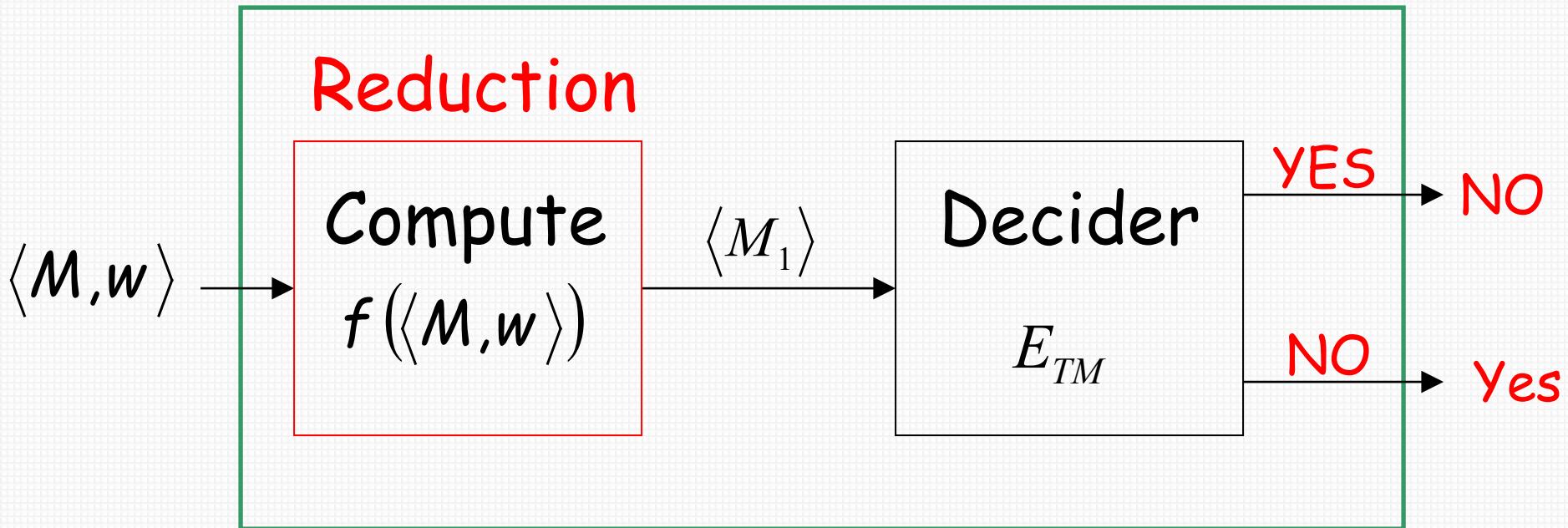


- TM S decides A_{TM} , R decides HALT_{TM} .

E_{TM} is undecidable

- $E_{TM} = \{<M,w> \mid M \text{ is a TM and } L(M) = \emptyset\}$
- Idea: Proof by contradiction. (**Same**)
 - Assume E_{TM} is decidable.
 - Prove that A_{TM} is reducible to E_{TM} .
 - Because A_{TM} is undecidable. Contradiction.

Decider for A_{TM}



Given the reduction,
if E_{TM} is decidable,
then A_{TM} is decidable

A contradiction!
since A_{TM}
is undecidable

- ✓ TM R decides E_{TM} . We construct TM S to decide A_{TM} .
- ✓ Run R on input $\langle M \rangle$
 - Accept. $L(M)$ is empty. M does not accept any string.
 - Reject. $L(M)$ is not empty. **But do not know what $L(M)$ is.** 

$$A_{TM} = \{(M, w) \mid M \text{ accepts } w\}$$

① R accept $\langle M_1 \rangle \Rightarrow L(M_1) = \emptyset \Rightarrow M_1 \text{ does not accept } w$.

② R reject $\langle M_1 \rangle \Rightarrow L(M_1) \neq \emptyset \Rightarrow L(M_1) = \{w \mid M_1 \text{ accepts } w\}$

- ✓ Run R on a modification of $\langle M \rangle$, call it M_1
 - To reject all strings except w , but on w it works as usual

M_1 = “On input x :

1. If $x \neq w$, reject.
2. If $x = w$, run M on input w and accept if M does.”

- Use R to determine whether M_1 recognizes the empty language

- Run R on input $\langle M, w \rangle$
 - M does not accept any string, including w ($L(M) = \emptyset$). $L(M)$ is empty. Accept.
 - M accepts w ($L(M) = w$). Reject.
- S = “On input $\langle M, w \rangle$, an encoding of a TM M and a string w .
 1. Use the description of M and w to construct the TM M_1 just described.
 2. Run R on input $\langle M_1 \rangle$.
 3. If R accepts, reject; if R rejects, accept.”

Contradiction

REGULAR_{TM} is undecidable

- $\text{REGULAR}_{\text{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is a regular language}\}$
- Similar as E_{TM}
- Construct modified machine M_2 to recognize $\{0^n 1^n \mid n \geq 0\}$ if M doesn't accept w , and to recognize regular language if M accepts w .

- Let R be a TM that decides $\text{REGULAR}_{\text{TM}}$ and construct TM S to decide A_{TM} .
- S works in the following manner:

S = “On input $\langle M, w \rangle$, where M is a TM and w is a string:

1. Construct the following TM M_2 .

M_2 = “On input x :

1. If x has the form $0^n 1^n$, accept.

2. If x does not have this form, run M on input w and accept if M accepts w . ”

2. Run R on input $\langle M_2 \rangle$.

3. If R accepts, accept; if R rejects, reject.”

- $\text{CFG}_{\text{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is a context-free language}\}$ is undecidable
- Generally, Rice's theorem
- Testing **any property** of the languages recognized by Turing machines if undecidable.

► A_{TM}

- Reduce A_{TM} to E_{TM}
- Reduce A_{TM} to $REGULAR_{TM}$

► Thus we can use E_{TM} , $REGULAR_{TM}$ for other problems

- An example: Reduce E_{TM} to EQ_{TM}

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

- E_{TM} : the language of a TM is empty
- EQ_{TM} : the languages of two TMs are the same
- For EQ_{TM} , if one of these languages is \emptyset , the language of another machine is empty- E_{TM} .
- In a sense, E_{TM} is a special problem of EQ_{TM} wherein one of the machines recognize the empty language.

- Let TM R decide EQ_{TM} and construct TM S to decide E_{TM} as follow.

$S = \text{"On input } \langle M \rangle, \text{ where } M \text{ is a TM:}$

1. Run R on input $\langle M, M_1 \rangle$, where M_1 is a TM that rejects all inputs.
2. If R accepts, accept; if R rejects, reject."

- If R decides EQ_{TM} , S decides E_{TM} .
- Thus EQ_{TM} is undecidable.

- Introduction
- Undecidable Problems from Language Theory
- Computation history method

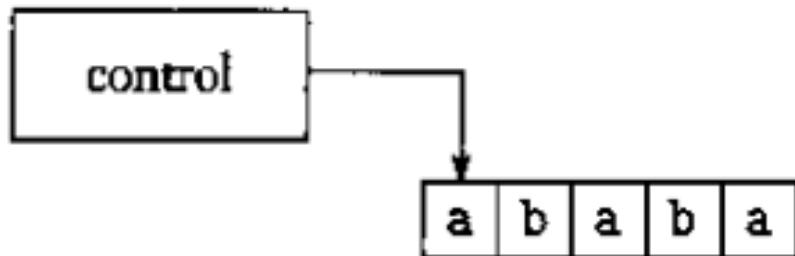
Computation history method

- The computation history for a Turing machine on an input is simply the **sequence of configurations** that the machine goes through as it processes the input.

- An **accepting computation history** for TM M on input w is a **sequence of configurations** C_1, C_2, \dots, C_l
 - C_1 is the start configuration of M on w,
 - C_l is an accepting configuration of M
 - Each C_i legally follows from C_{i-1} according to rules of M
- A **rejecting computation history** for M on w is defined similarly, except that C_l is a rejecting configuration.
- Computation Histories are finite sequences.

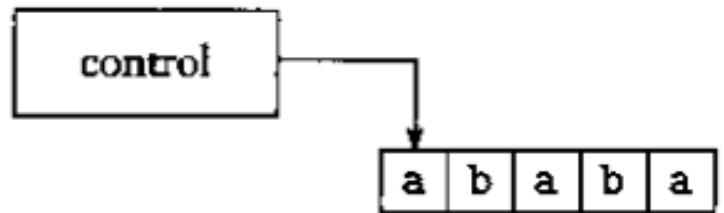
Linear bounded automation LBA

- LBA is a restricted type of TM.
 - Tape head is limited in the portion of the tape containing the tape.
 - If it tries to move its head off either end of the input, stays where it is.



- LBAs are quite powerful
 - Deciders for A_{DFA} , A_{CFG} , E_{DFA} , and E_{CFG} all are LBAs.
 - Every CFL can be decided by an LBA.
- $A_{LBA} = \{ \langle M, w \rangle \mid M \text{ is an LBA that accepts string } w \}$
- Prove A_{LBA} is decidable.

- Let M be an LBA with q states and g symbols in the tape alphabet. There are exactly qng^n distinct configurations of M for a tape of length n .
- Proof
 - A configuration contains
 - the state of the control (q states),
 - position of the head (n positions),
 - contents of the tape (g^n possible strings of tape symbol).
 - The product of these three quantities is the total number of different configurations.



- Idea
 - If M halts, ok
 - If M loops: how to detect
 - Number of configuration is finite
- Proof: the algorithm that decides A_{LBA} is as follows.

$L =$ “On input $\langle M, w \rangle$, where M is an LBA and w is a string:

1. Simulate M on w for qng^n steps or until it halts.
2. If M has halted, accept if it has accepted and reject if it has rejected. If it has not halted, reject.”

E_{LBA} is undecidable

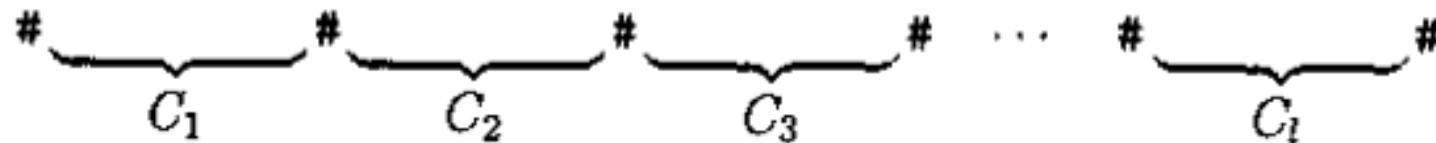
- $E_{LBA} = \{ \langle M \rangle \mid M \text{ is an LBA and } L(M) = \emptyset \}$
- Idea: how to construct TM S to decide A_{TM}
 - For a TM M and an input w , determine whether M accept w by construct a LBA B and testing whether $L(B)$ is empty
 - $L(B)$ comprises all accepting computation histories for M on w
 - If M accepts $w \Leftrightarrow$ exist a accepting computation history for M on $w \Leftrightarrow L(B)$ is not empty
 - If M doesn't accept $w \Leftrightarrow$ not exist a accepting computation history for M on $w \Leftrightarrow L(B)$ is empty
 - Such, if we decide whether $L(B)$ is empty, we determine whether M accepts $w \Rightarrow A_{TM}$ is decidable. **Contradiction!**

IDEA: Construct LBA B

- Construct B for M and w using computation histories.

$B = \{ x \mid x \text{ is an accepting computation history for } M \text{ on } w \}$

- An **accepting computation history x** is the sequence C_1, C_2, \dots, C_l that M goes through as it accepts some string w.



- Why B is a LBA?

$M \text{ accepts } w \Rightarrow M \text{ stops on accepting states within finite steps} \Rightarrow$
The length of x is finite $\Rightarrow B$ is a LBA.

- When B receives an input x , B is supposed to accept if x is an accepting computation history for M on w.
 - First, B breaks up x into strings C_1, C_2, \dots, C_l .
 - Then B determine whether C_i satisfy the three conditions of an accepting computation history:
 - C_1 is the start configuration for M on w.
 - Each C_{i+1} legally follows from C_i .
 - C_l is an accepting configuration for M.



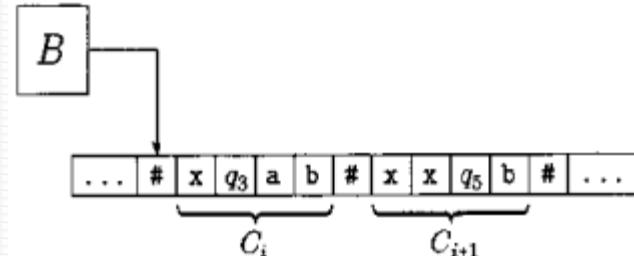
- Suppose that TM R decides E_{LBA} . Construct TM S that decides A_{TM} as follows.

S = “On input $\langle M, w \rangle$, where M is a TM and w is a string:

1. Construct LBA B from M and w as described in the proof idea.
2. Run R on input $\langle B \rangle$.
3. If R rejects, accept; if R accepts, reject.”

- If R accepts $\langle B \rangle \Rightarrow \langle B \rangle \in E_{\text{LBA}} \Rightarrow B$ is a LBA, and $L(B) = \emptyset \Rightarrow M$ has no accepting computation history on $w \Rightarrow M$ doesn’t accept $w \Rightarrow S$ rejects $\langle M, w \rangle$.
- If R rejects $\langle B \rangle \Rightarrow \langle B \rangle \notin E_{\text{LBA}} \Rightarrow L(B) \neq \emptyset$ or B is not a LBA (as we have proved B is a LBA) $\Rightarrow L(B) \neq \emptyset \Rightarrow$ there is at least one an accepting computation history for M on $w \Rightarrow M$ accept $w \Rightarrow S$ accepts $\langle M, w \rangle$.

S is a decider for A_{TM} . Contradiction!



ALL_{CFG} is undecidable

- ALL_{CFG} = {<G> | G is a CFG and L(G) = Σ*}.
- For a TM M and an input w, we construct a CFG G that generates all strings if and only if M does not accept w.
- If M does accept w, G doesn't generate the accepting computation history for M on w.
- G is designed to generate all strings that not accepting computation histories for M on w.

- Make CFG G generate all strings that fail to be an accepting computation history for M on w, follow the strategy
 - An accepting computation history for M on w appears as
$$\#C_1\#C_2\#\dots\#C_l\#$$
 - Then , G generates all strings that
 1. *do not start with* C_1 ,
 2. *do not end with an accepting configuration*, or
 3. *where some* C_i *does not properly yield* C_{i+1} *under the rules of M.*
- If M doesn't accept w, no accepting computation history exists, so all strings fail in one way or another. Therefore G would generate all string, as desired.

- ✓ Introduction
- ✓ Undecidable Problems from Language Theory
- ✓ Computation history method

Theory of Computation

Lesson 10-1

Post Correspondence Problem



王轩
Wang
Xuan

Dominos and Match

- Domino, containing two strings on each side

$$\left[\begin{array}{c} a \\ \hline ab \end{array} \right]$$

- Collection of dominos

$$\left\{ \left[\begin{array}{c} b \\ \hline ca \end{array} \right], \left[\begin{array}{c} a \\ \hline ab \end{array} \right], \left[\begin{array}{c} ca \\ \hline a \end{array} \right], \left[\begin{array}{c} abc \\ \hline c \end{array} \right] \right\}$$

- **Match:** make a list of these dominos (repetitions permitted) so that the symbols on the top is **same** as the symbol on the bottom.

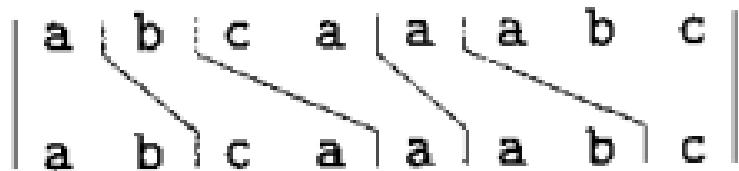
- An example

$$\left[\begin{array}{c} a \\ \hline ab \end{array} \right] \left[\begin{array}{c} b \\ \hline ca \end{array} \right] \left[\begin{array}{c} ca \\ \hline a \end{array} \right] \left[\begin{array}{c} a \\ \hline ab \end{array} \right] \left[\begin{array}{c} abc \\ \hline c \end{array} \right]$$

- symbols: abcaaabc

Example

- Deform the dominos so that corresponding symbols from top and bottom line up.



- For some collections of dominos, there is no match. For example

$$\left\{ \left[\frac{abc}{ab} \right], \left[\frac{ca}{a} \right], \left[\frac{acc}{ba} \right] \right\}$$

Post Correspondence Problem

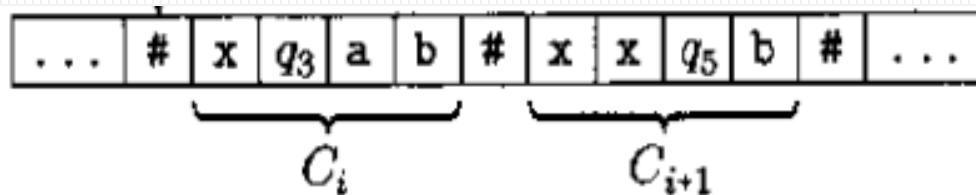
- PCP is to determine whether a collection of dominos has a match. It's **unsolvable** by algorithms.
 - Formally, an instance of the PCP is a collection P of dominos:
$$P = \left\{ \left[\begin{smallmatrix} t_1 \\ b_1 \end{smallmatrix} \right], \left[\begin{smallmatrix} t_2 \\ b_2 \end{smallmatrix} \right], \dots, \left[\begin{smallmatrix} t_k \\ b_k \end{smallmatrix} \right] \right\},$$
 - and a match is a sequence i_1, i_2, \dots, i_l , where $t_{i_1}t_{i_2}\dots t_{i_l} = b_{i_1}b_{i_2}\dots b_{i_l}$. The problem is to determine whether P has a match.
- $\text{PCP} = \{<P> \mid P \text{ is an instance of the Post correspondence problem with a match } \}$

PCP is undecidable

- Idea: Reduction from A_{TM} via accepting computation histories
- From any TM M and input w , construct an instance P where a match is an accepting computation history for M on w .
 - If we could determine whether the instance has a match, we would be able to determine whether M accepts w .

How to construct P

- ▶ Choose the dominos in P so that making a match forces a simulation of M to occur.
 - Each domino links a position in one configuration with the corresponding one in the next configuration.



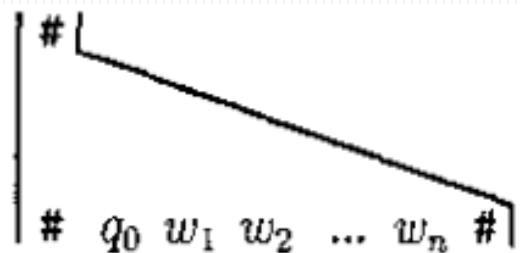
- ▶ Modify PCP to require that a match starts with the first domino, $\left[\frac{t_1}{b_1}\right]$
- ▶ Modified Post Corresponding Problem (MPCP)
 - MPCP={ $\langle P \rangle$ | P is an instance of the Post correspondence problem with a match that starts with the first domino}

- Let TM R decide the PCP and construct S deciding A_{TM} . Let

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$$

- S construct an instance of PCP p that has a match iff M accepts w. We first construct an instance P' of MPCP.
- There are 7 parts, each of which accomplishes a particular aspect of simulating M on w.

- The construction **begins** as
- Put $\left[\frac{\#}{\#q_0w_1w_2 \dots w_n \#} \right]$ into P' as the first domino $\left[\frac{t_1}{b_1} \right]$.
- The bottom string begins correctly with $C1 = q_0w_1w_2 \dots w_n$, the first configuration in the accepting computation history for M on w .



- ▶ Extending the top string to **match** the bottom string, we provide additional dominos to allow this extension.
 - Part2 handles head motions to the right
 - Part3 handles head motions to the left
 - Part4 handles the tape cells not adjacent to the head
- ▶ These additional dominos cause M's next configuration to appear at the extension of the bottom by forcing a single-step simulation of M.

Part 2. For every $a, b \in \Gamma$ and every $q, r \in Q$ where $q \neq q_{\text{reject}}$,

if $\delta(q, a) = (r, b, R)$, put $\left[\frac{qa}{br} \right]$ into P' .

Part 3. For every $a, b, c \in \Gamma$ and every $q, r \in Q$ where $q \neq q_{\text{reject}}$,

if $\delta(q, a) = (r, b, L)$, put $\left[\frac{cq a}{rc b} \right]$ into P' .

Part 4. For every $a \in \Gamma$,

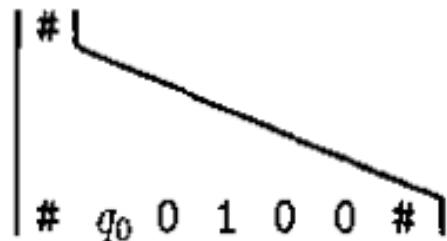
put $\left[\frac{a}{a} \right]$ into P' .

Put $\left[\frac{\#}{\#} \right]$ and $\left[\frac{\#}{\square\#} \right]$ into P' .

- The first marks the separation of the configurations
- The second add a blank symbol \square at the end of the configuration to simulate the infinitely many blanks.

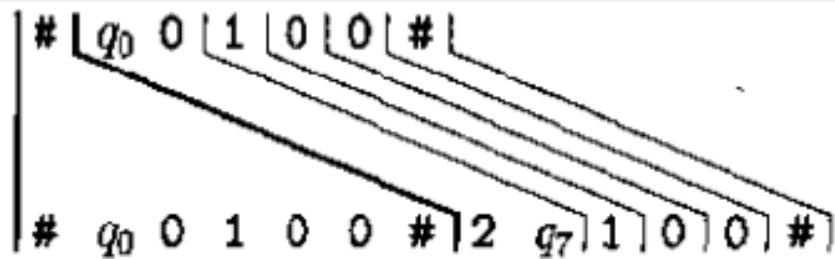
An example

- Let $\Gamma = \{0, 1, 2, \sqcup\}$.
- w is the string 0100 , and the start state of M is q_0 .
- $\delta(q_0, 0) = (q_7, 2, R)$
- Part 1 place the domino $\left[\frac{\#}{\#_{q_0} 0100} \right] = \left[\frac{t_1}{b_1} \right]$ P' , the match begins



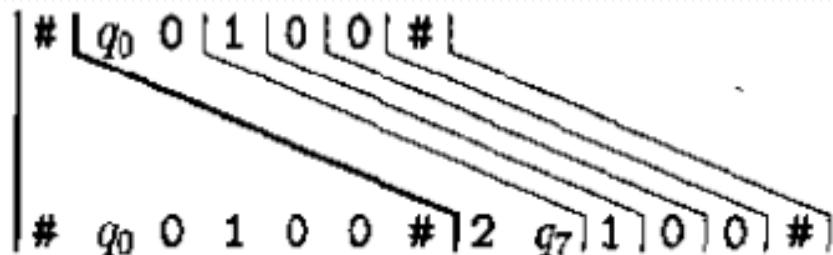
- Then, part 2 places the domino $\left[\frac{q_0 0}{2q_7} \right]$ as in $\delta(q_0, 0) = (q_7, 2, R)$

- Part 4 places the dominos $\left[\begin{smallmatrix} 0 \\ 0 \end{smallmatrix}\right]$, $\left[\begin{smallmatrix} 1 \\ 1 \end{smallmatrix}\right]$, $\left[\begin{smallmatrix} 2 \\ 2 \end{smallmatrix}\right]$, and $\left[\begin{smallmatrix} u \\ v \end{smallmatrix}\right]$ P' .
- Together with part 5, allow us to extend the match to

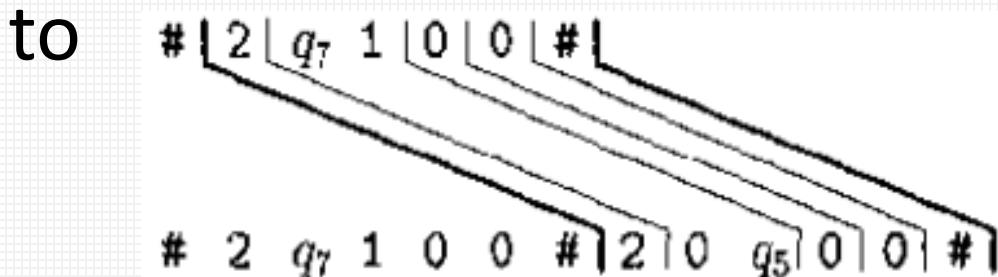


Example

- Continuing the example, it's in state q_7 .



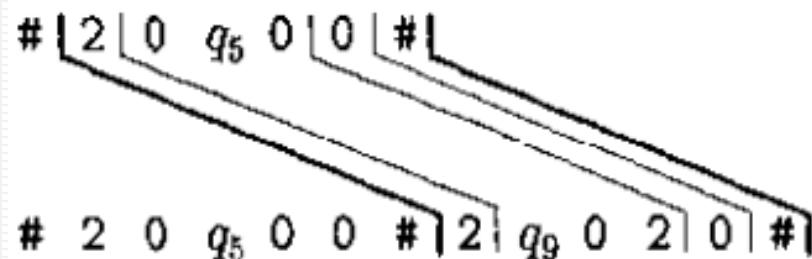
- There is a transition $\delta(q_7, 1) = (q_5, 0, R)$. Then we have the domino $\left[\frac{q_7 1}{0 q_5} \right]$ in P' . The match extends to



- We have transition $\delta(q_5, 0) = (q_9, 2, L)$. Then we have the dominos

$$\left[\frac{0q_50}{q_902} \right], \left[\frac{1q_50}{q_912} \right], \left[\frac{2q_50}{q_922} \right], \text{ and } \left[\frac{\sqcup q_50}{q_9\sqcup 2} \right].$$

- The first one is relevant. The match extends to



- The process continues until M reaches a halting state.

For every $a \in \Gamma$,

put $\left[\frac{a}{q_{\text{accept}}} \right]$ and $\left[\frac{q_{\text{accept}} a}{q_{\text{accept}}} \right]$ into P' .

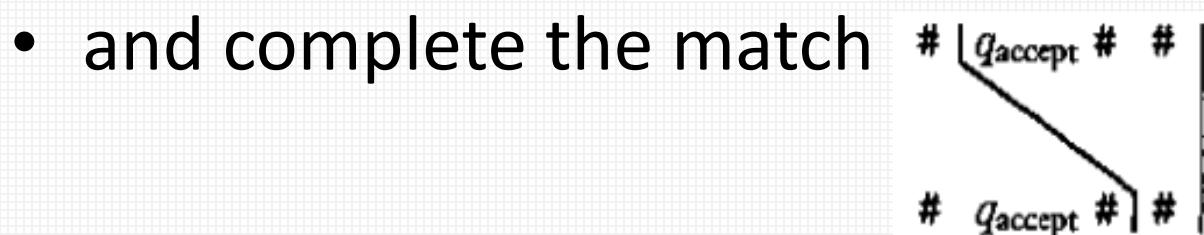
- This step adding “pseudo-step” after it has halted, where the head “eats” adjacent symbols until none are left.

- We have . Extend the match to

|
2 1 q_{accept} 0 2 #|

| 2 | 1 | q_{accept} 0 | 2 | # | ... | # |
 . . .
 # 2 1 q_{accept} 0 2 # | 2 | 1 | q_{accept} | 2 | # | ... | # | q_{accept} | # |

- Add the domino $\left[\frac{q_{\text{accept}} \# \#}{\#} \right]$



- That **concludes** the construction of P' .
- P' is an instance of MPCP whereby the match simulates the computation of M on w .
- We need to convert P' to P , an instance of the PCP that simulates M on w .

- The idea is to build the requirement of **starting with the first domino** using **some** notation so that starting the explicit requirement is unnecessary.
- Let $u=u_1u_2\dots u_n$ be any string of length n .
Define
 - $\star u = *u_1*u_2*\dots*u_n$
 - $u \star = u_1*u_2*\dots*u_n*$
 - $\star u \star = *u_1*u_2*\dots*u_n*$

- If P' are the collection $\left\{ \left[\frac{t_1}{b_1} \right], \left[\frac{t_2}{b_2} \right], \left[\frac{t_3}{b_3} \right], \dots, \left[\frac{t_k}{b_k} \right] \right\}$
- Let P be the collection

$$\left\{ \left[\frac{*t_1}{*b_1*} \right], \left[\frac{*t_1}{b_1*} \right], \left[\frac{*t_2}{b_2*} \right], \left[\frac{*t_3}{b_3*} \right], \dots, \left[\frac{*t_k}{b_k*} \right], \left[\frac{*◇}{◇} \right] \right\}$$
- Considering P as an instance of PCP, the starting domino must be $\left[\frac{*t_1}{*b_1*} \right]$.
- The domino $\left[\frac{*◇}{◇} \right]$ is added to the end of the match to balance number of *.

Example

- TM

$$M = (\{q_1, q_2, q_3\}, \{0, 1\}, \{0, 1, B\}, \delta, q_1, B, \{q_3\})$$

the transition function δ is as follow:

q_i	$\delta(q_i, 0)$	$\delta(q_i, 1)$	$\delta(q_i, B)$
q_1	$(q_2, 1, R)$	$(q_2, 0, L)$	$(q_2, 1, L)$
q_2	$(q_3, 0, L)$	$(q_1, 0, R)$	$(q_2, 0, R)$
q_3	--	--	--

$$w = 01.$$

$w = 01$

We simulate Turing Machine M, the computation history is:

→

$q_1 01 \#$ →

$q_1 01 \# 1 q_2 1 \#$ →

$q_1 01 \# 1 q_2 1 \# 10 q_1 B \#$ →

$q_1 01 \# 1 q_2 1 \# 10 q_1 B \# 1 q_2 01 \#$ →

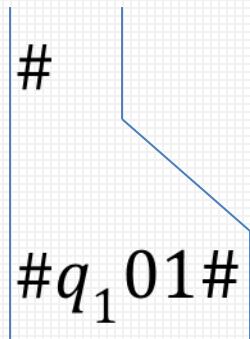
$q_1 01 \# 1 q_2 1 \# 10 q_1 B \# 1 q_2 01 \# q_3 101 \#$

Then M accept w.

Example — MPCP

Part 1:

Construct the first domino:



Example — MPCP

Part 2, 3, 4

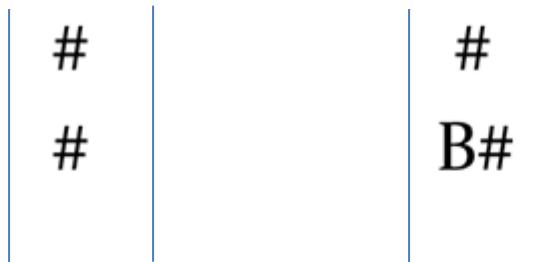
q_i	$\delta(q_i, 0)$	$\delta(q_i, 1)$	$\delta(q_i, B)$
q_1	$(q_2, 1, R)$	$(q_2, 0, L)$	$(q_2, 1, L)$
q_2	$(q_3, 0, L)$	$(q_1, 0, R)$	$(q_2, 0, R)$
q_3	--	--	--

$q_1 0$	$\$ q_2 0$	$\$ q_1 0$	$q_2 1$	$\$ q_1 B$	$q_2 B$	0	1	B
$1q_2$	$q_3 \$ 0$	$q_2 \$ 0$	$0q_1$	$q_2 \$ 1$	$0q_2$	0	1	B

Here \$ can be 0|1|B

Example – MPCP

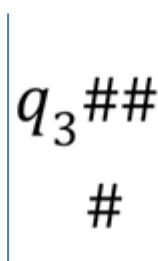
Part 5



Part 6



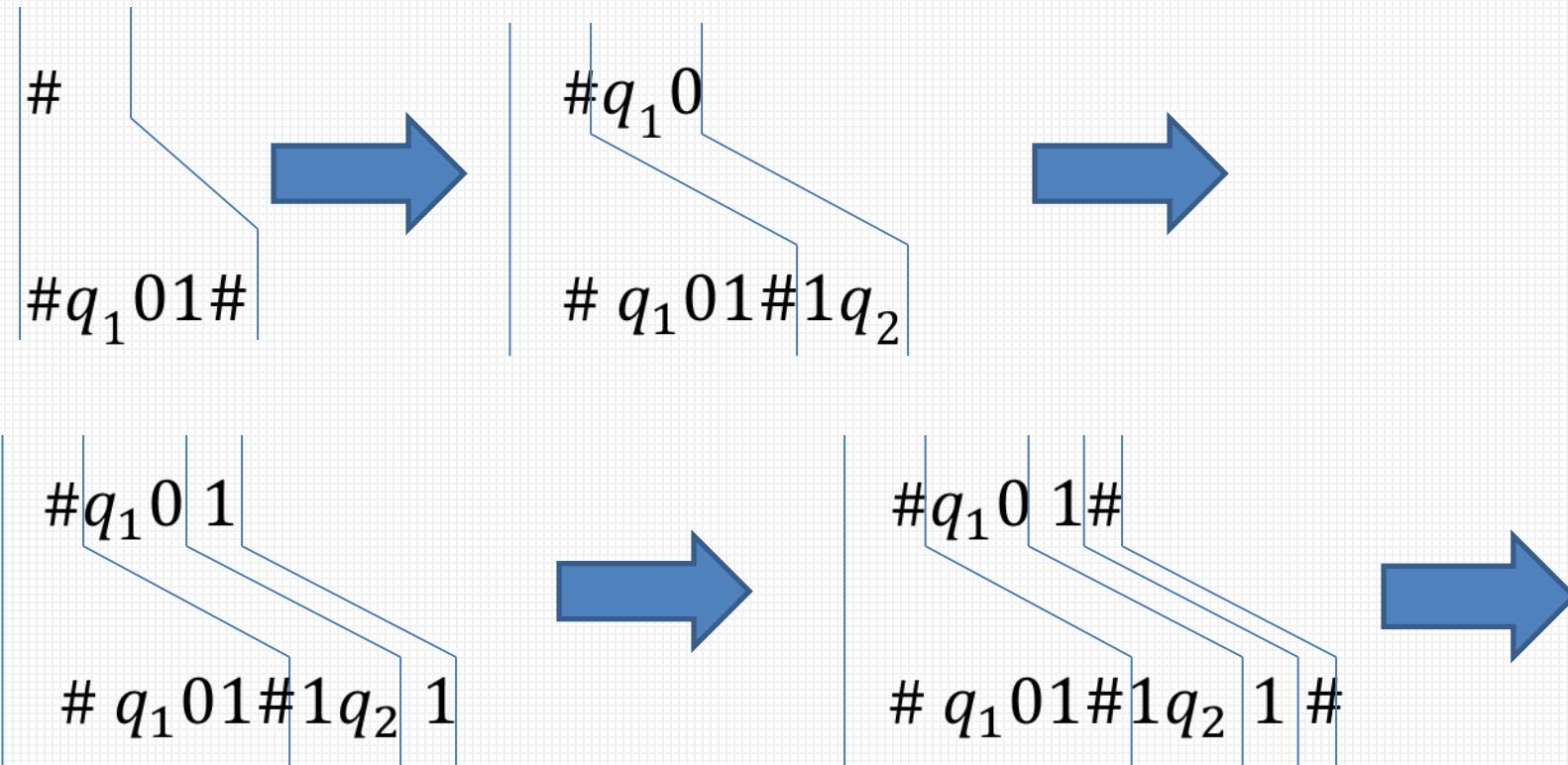
Part 7



We have got all
the dominos.

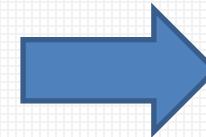
Example – MPCP

- Then we begin to simulate the MPCP:

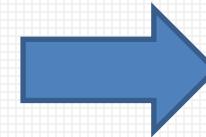


Example – MPCP

q_1 0 1#1
q_1 0 1#1 q_2 1 #1

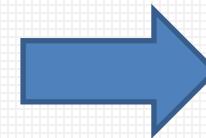


q_1 0 1#1 q_2 1
q_1 0 1#1 q_2 1 #10 q_1 B

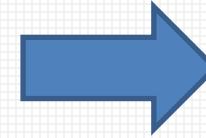


Example – MPCP

$q_1 0 1 \# 1 q_2 1 \#$
$q_1 0 1 \# 1 q_2 1 \# 1 0 q_1 B \#$



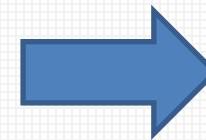
$q_1 0 1 \# 1 q_2 1 \# 1$
$q_1 0 1 \# 1 q_2 1 \# 1 0 q_1 B \# 1$



Example – MPCP

$q_1 0 1 \# 1 q_2 1 \# 1 0 q_1 B$

$q_1 0 1 \# 1 q_2 1 \# 1 0 q_1 B \# 1 q_2 0 1$



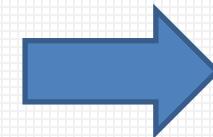
$q_1 0 1 \# 1 q_2 1 \# 1 0 q_1 B \#$

$q_1 0 1 \# 1 q_2 1 \# 1 0 q_1 B \# 1 q_2 0 1 \#$



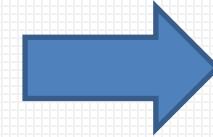
Example — MPCP

$q_1 0 1 \# 1 q_2 1 \# 1 0 q_1 B \# 1 q_2 0$



$q_1 0 1 \# 1 q_2 1 \# 1 0 q_1 B \# 1 q_2 0 1 \# q_3 1 0$

$q_1 0 1 \# 1 q_2 1 \# 1 0 q_1 B \# 1 q_2 0 1$

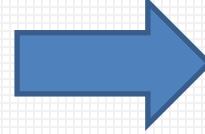


$q_1 0 1 \# 1 q_2 1 \# 1 0 q_1 B \# 1 q_2 0 1 \# q_3 1 0 1$

Example – MPCP

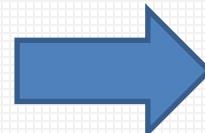
$q_1 01 \# 1 q_2 1 \# 1 0 q_1 B \# 1 q_2 01 \#$

$q_1 01 \# 1 q_2 1 \# 1 0 q_1 B \# 1 q_2 01 \# q_3 101 \#$

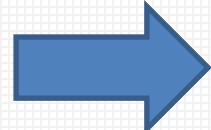


$q_1 01 \# 1 q_2 1 \# 1 0 q_1 B \# 1 q_2 01 \# q_3 0$

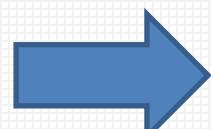
$q_1 01 \# 1 q_2 1 \# 1 0 q_1 B \# 1 q_2 01 \# q_3 101 \# q_3$



Example – MPCP

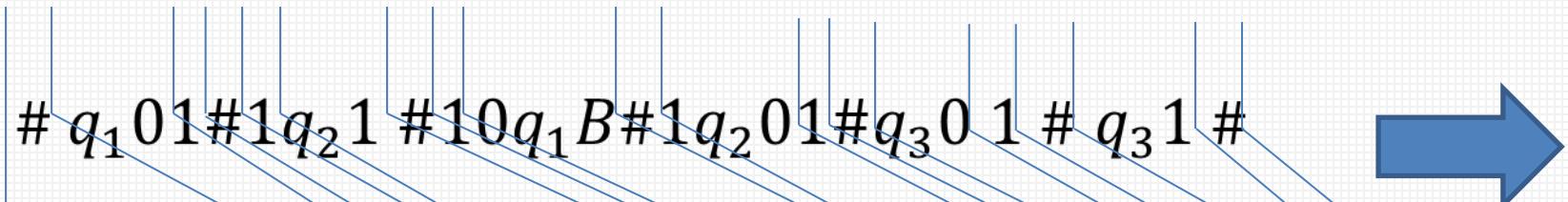
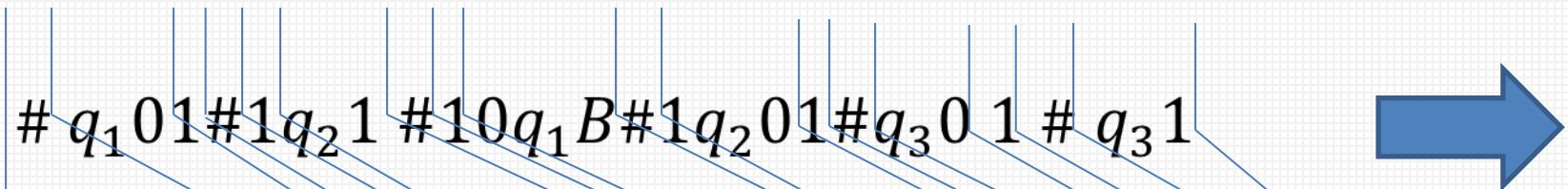
$q_1 01 \# 1 q_2 1 \# 1 0 q_1 B \# 1 q_2 01 \# q_3 01$ 

$q_1 01 \# 1 q_2 1 \# 1 0 q_1 B \# 1 q_2 01 \# q_3 1 0 1 \# q_3 1$

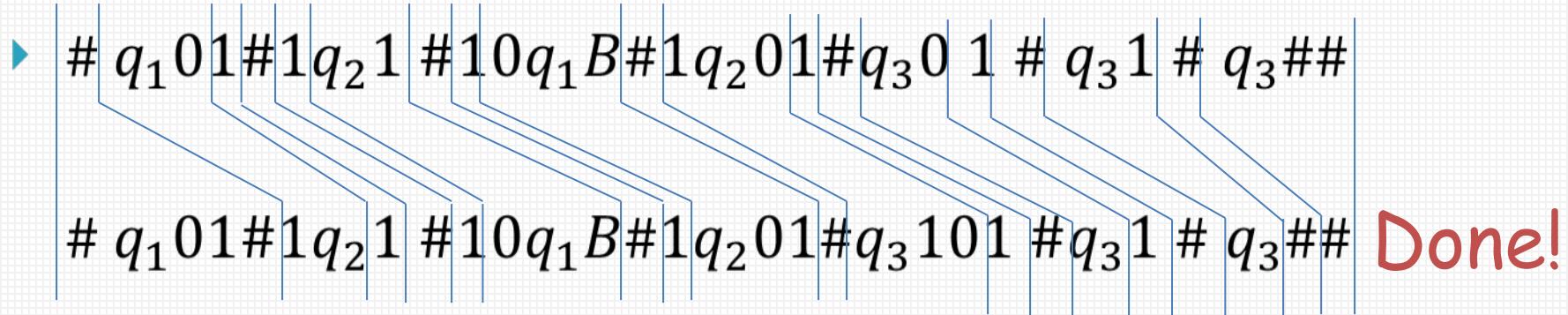
$q_1 01 \# 1 q_2 1 \# 1 0 q_1 B \# 1 q_2 01 \# q_3 0 1 \#$ 

$q_1 01 \# 1 q_2 1 \# 1 0 q_1 B \# 1 q_2 01 \# q_3 1 0 1 \# q_3 1 \#$

Example – MPCP



Example — MPCP

▶  Done!

Summary:

So when we can find a Turing Machine M accept w,
there is a PCP(MPCP) P can be found. 

In another word, if TM R decide the PCP, we can
construct a TM S deciding A_{TM} . 

For A_{TM} is undecidable, PCP is undecidable.

Theory of Computation

Lesson 10-2

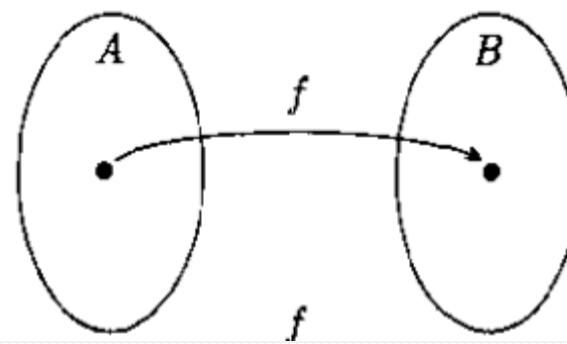
Mapping Reducibility



王轩
Wang
Xuan

Mapping Reducibility

- Reduce problem A to problem B by using a **mapping reducibility** means that a **computation function** exists that converts instances of A to instance of B.
- We can solve A with a solver for B using the conversion function (reduction)
 - Convert any instance of A to an instance of B
 - Applying the solver for B

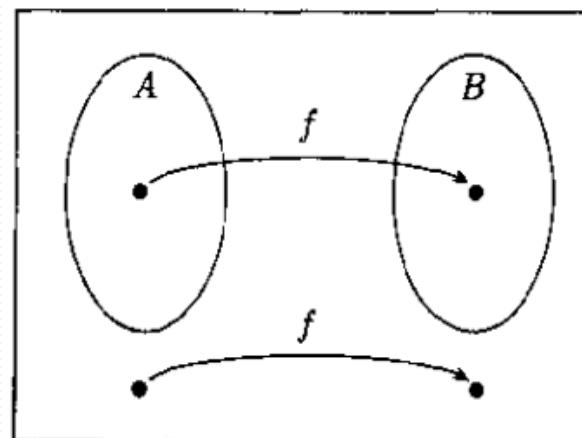


Computable functions

- Definition: A function $f: \Sigma^* \rightarrow \Sigma^*$ is a **computable function** if some Turing machine M , on every input w , halts with just $f(w)$ on its tape.
- Example
 - All usual arithmetic operations on integers are computable functions. For example, a machine takes $\langle m, n \rangle$ as input and returns $m+n$.

Formal Definition

- Language A is mapping reducible to language B, written $A \leq_m B$, if there is a computable function $f: \Sigma^* \rightarrow \Sigma^*$, where for every w ,
 - $w \in A \Leftrightarrow f(w) \in B$
- The function f is called the reduction of A to B.
- Mapping reducibility is transitive.



Membership Testing

- Mapping reduction provides a way to convert **membership testing** in A to membership testing in B
 - To testing whether $w \in A$
 - Use the reduction f to map w to $f(w)$
 - Test whether $f(w) \in B$
- If problem A is mapping reducible to a solved problem B, we can obtain the solution of A.

- If $A \leq_m B$ and B is decidable, then A is decidable.
- Proof:
 - Let M be the decider for B and f be the reduction from A to B. We describe a decider N for A as follow.
 - N = “On input w:
 - 1. Compute $f(w)$
 - 2. Run M on input $f(w)$ and output whatever M outputs.”
 - If M accept $f(w) \Rightarrow f(w) \in B \Rightarrow w \in A \Rightarrow N$ accept w;
if M reject $f(w) \Rightarrow f(w) \notin B \Rightarrow w \notin A \Rightarrow N$ reject w.

- Corollary: If $A \leq_m B$ and A is undecidable, then B is undecidable.
- Example: HALT_{TM} is undecidable.
 - Assume $A = A_{\text{TM}}$, $B = \text{HALT}_{\text{TM}}$
 - Mapping reducible from A_{TM} to HALT_{TM}
 - There is a computation function f that takes input of the form $\langle M, w \rangle \in A_{\text{TM}}$ if and only if $\langle M', w' \rangle \in \text{HALT}_{\text{TM}}$ the form $\langle M', w' \rangle$, where

- The following machine F computes a reduction f .
- $F = \text{"On input } \langle M, w \rangle:$
 - 1. Construct the following machine M' .
 - $M' = \text{"On input } x:$
 - 1. Run M on x .
 - 2. If M accepts, accept.
 - 3. If M rejects, enter a loop."
 - 2. Output $\langle M', w \rangle$."

- a. If $\langle M, w \rangle \in A_{TM} \Rightarrow M \text{ accepts } w \Rightarrow M' \text{ halts} \Rightarrow \langle M', w \rangle \in \text{HALT}_{TM}$
- b. If $\langle M, w \rangle \notin A_{TM} \Rightarrow M \text{ doesn't accept } w \Rightarrow M' \text{ enters a loop} \Rightarrow \langle M', w \rangle \notin \text{HALT}_{TM}$

- Post corresponding problem (PCP)
 - $A_{TM} \leq_m MPCP, MPCP \leq_m PCP$
 - Mapping reducibility is transitive
 - $A_{TM} \leq_m PCP$
- $E_{TM} \leq EQ_{TM}$

- If $A \leq_m B$ and B is Turing-recognizable, then A is Turing-recognizable. (same as before)
 - If $A \leq_m B$ and A is not Turing-recognizable, then B is not Turing-recognizable.
-
- $A \leq_m B \iff \overline{A} \leq_m \overline{B}$
 - $(w \in \overline{A} \iff w \notin A \iff f(w) \notin B \iff f(w) \in \overline{B})$

Theorem

- $\overline{\text{EQ}_{\text{TM}}}$ is neither Turing-recognizable nor co-Turing-recognizable.
- 1. Prove $\overline{\text{EQ}_{\text{TM}}}$ is not Turing-recognizable.
 - We reducing $\overline{A_{\text{TM}}}$ to $\overline{\text{EQ}_{\text{TM}}}$.
 - $\overline{A_{\text{TM}}} \leq_m \overline{\text{EQ}_{\text{TM}}} \Leftrightarrow A_{\text{TM}} \leq_m \overline{\text{EQ}_{\text{TM}}}$
 - Since $\overline{A_{\text{TM}}}$ is not Turing-recognizable, $\overline{\text{EQ}_{\text{TM}}}$ is not Turing-recognizable.

- The reduction function f works as follows.

F = “On input $\langle M, w \rangle$ where M is a TM and w a string:

1. Construct the following two machines M_1 and M_2 .

M_1 = “On any input:

1. *Reject.*”

M_2 = “On any input:

1. Run M on w . If it accepts, *accept.*”

2. Output $\langle M_1, M_2 \rangle$.”

- $\langle M_1, M_2 \rangle \in EQ_{TM} \Rightarrow L(M_1) = L(M_2) \Rightarrow L(M_2) = \emptyset \Rightarrow M$ doesn't accept w .
- $\langle M_1, M_2 \rangle \notin EQ_{TM} \Rightarrow L(M_1) \neq L(M_2) \Rightarrow L(M_2) \neq \emptyset \Rightarrow M$ accepts w .

- 2. Prove $\overline{\text{EQ}_{\text{TM}}}$ is not co-Turing-recognizable we need to reducing A_{TM} to the complement of $\overline{\text{EQ}_{\text{TM}}}$, $\overline{\text{EQ}_{\text{TM}}}$. Thus to prove $A_{\text{TM}} \leq_m \overline{\text{EQ}_{\text{TM}}}$.
- $A_{\text{TM}} \leq_m \text{EQ}_{\text{TM}} \Leftrightarrow \overline{A_{\text{TM}}} \leq_m \overline{\text{EQ}_{\text{TM}}}$
- Since $\overline{A_{\text{TM}}}$ is not Turing-recognizable, $\overline{\text{EQ}_{\text{TM}}}$ is not Turing-recognizable, so EQ_{TM} is not co-Turing-recognizable.

- The reduction function g is

$G =$ “The input is $\langle M, w \rangle$ where M is a TM and w a string.

1. Construct the following two machines M_1 and M_2 :

M_1 = “On any input:

1. *Accept.*”

M_2 = “On any input:

1. Run M on w .

2. If it accepts, *accept.*”

2. Output $\langle M_1, M_2 \rangle$.

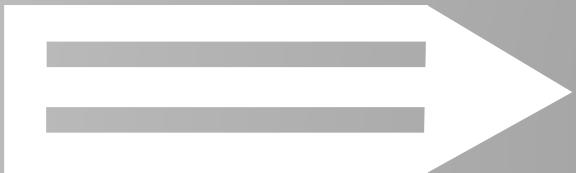
- $\langle M_1, M_2 \rangle \in EQ_{TM} \Rightarrow L(M_1) = L(M_2) \Rightarrow M$ accepts $w \Rightarrow \langle M, w \rangle \in A_{TM}$
- $\langle M_1, M_2 \rangle \notin EQ_{TM} \Rightarrow L(M_1) \neq L(M_2) \Rightarrow M$ doesn't accept $w \Rightarrow \langle M, w \rangle \notin A_{TM}$

$$A_{TM} \leq_m EQ_{TM}$$

Theory of Computation

Lesson 11-1

The Recursion Theorem



王轩

Wang Xuan

- Introduction
- Self-Reference
- The Recursion Theorem
- Applications

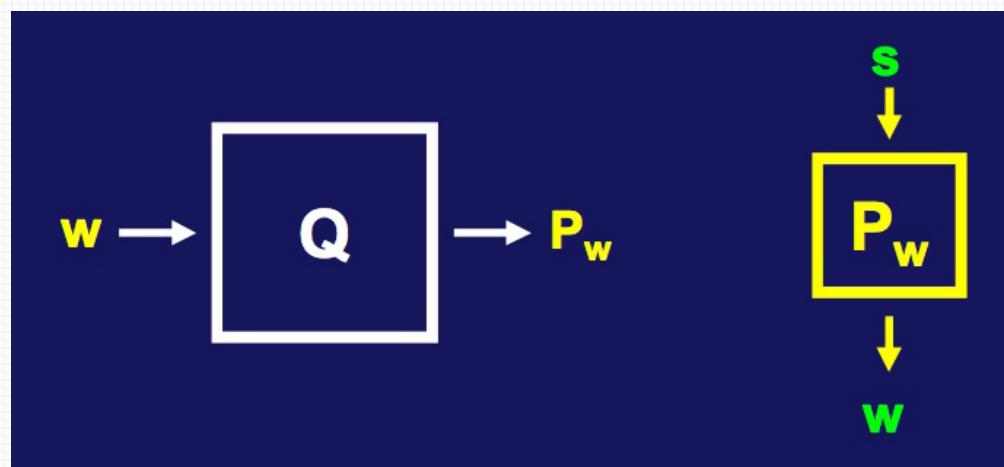
- The recursion theorem is a mathematical result that plays an important role in advanced work in the theory of computability.
- It has connections to mathematical logic, the theory of self-reproducing systems and even computer viruses.
- Making machines that reproduce themselves is possible.

- Introduction
- Self-Reference
- The Recursion Theorem
- Applications

- Let's begin by making a Turing machine **SELF**, that ignores its input and prints out a copy of its own description.
- To help describe SELF, we need the following lemma.
- Lemma:
 - There is a computable function $q: \Sigma^* \rightarrow \Sigma^*$
 - Where if w is any string, $q(w)$ is the description of a Turing machine P_w that prints out w and halt.

- The following TM Q computes $q(w)$.
- $Q = \text{"On input string } w:$
 - 1. Construct the following Turing machine P_w
 - $P_w = \text{"On any input } s:$
 - » 1. Erase input.
 - » 2. Write w on tape.
 - » 3. Halt."
 - 2. Output $\langle P_w \rangle$."

✓ $\langle P_w \rangle = q(w)$



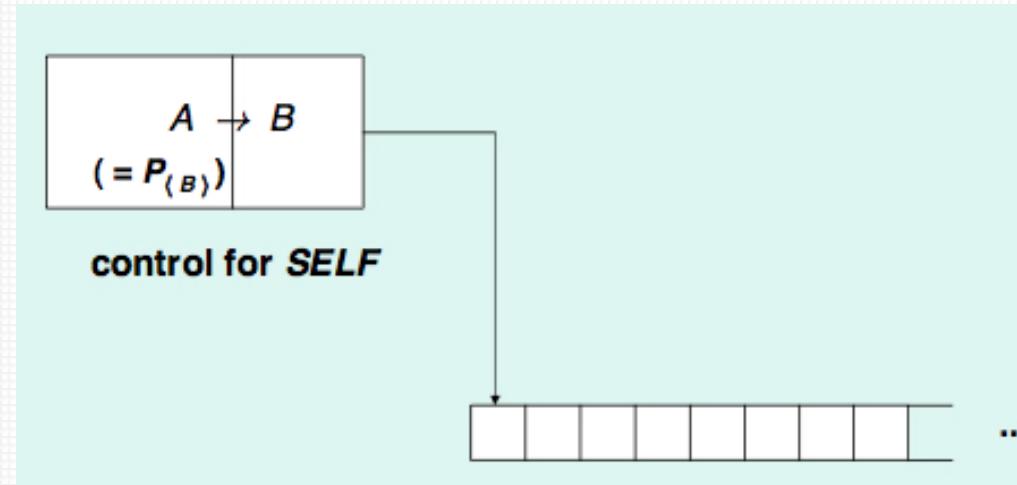
A TM That Prints Itself

- Next we build a TM, SELF, that ignores its own input and prints out a copy of its description.
- The Turing machine SELF is in two parts, A and B. We think of A and B as being two separate procedures that go together to make up SELF.

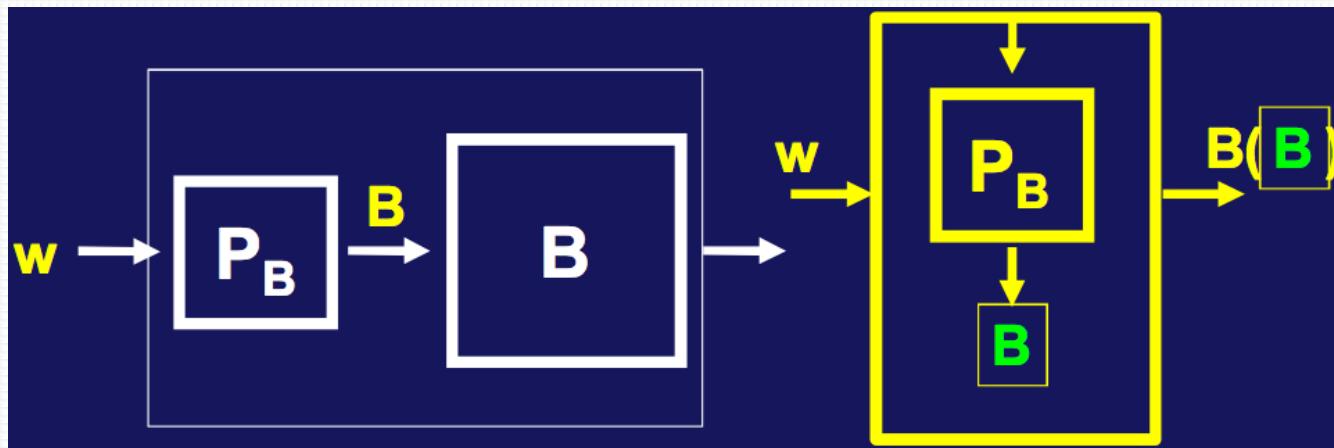
- Part A runs first and upon completion passes control to part B.
- The job of A is to print a description of B on the tape ,hence $A = P_{\langle B \rangle}(P_w \text{ where } w=\langle B \rangle)$.
- The job of B is to print out a description of A.
- The tasks are similar, but are carried out differently.

- For A we use the machine $P_{\langle B \rangle}$, described by $q(\langle B \rangle)$.
- $q(\langle B \rangle)$ means applying the function q to $\langle B \rangle$.
- Our description of A depends on having a description of B. So we can't complete the description of A until we construct B.

- If B can obtain $\langle B \rangle$, it can apply q to that and obtain $\langle A \rangle$.
- What how can B obtain $\langle B \rangle$?
- Well, it was printed on the tape, just before A passed control to B.
- So, B computes $q(\langle B \rangle) = \langle A \rangle$ and combines these and writes its own description $\langle AB \rangle$.



- $A = P_{\langle B \rangle}$: A is the TM that prints out the description of B (But we do not have B yet!)
- B = “On input $\langle M \rangle$ where M is a portion of a TM:
 1. Compute $q(\langle M \rangle)$, (find the description of the machine which prints $\langle M \rangle$)
 2. Combine the result with $\langle M \rangle$ to make a complete TM.
 3. Print the description of this TM and halt.”



How Self Behaves

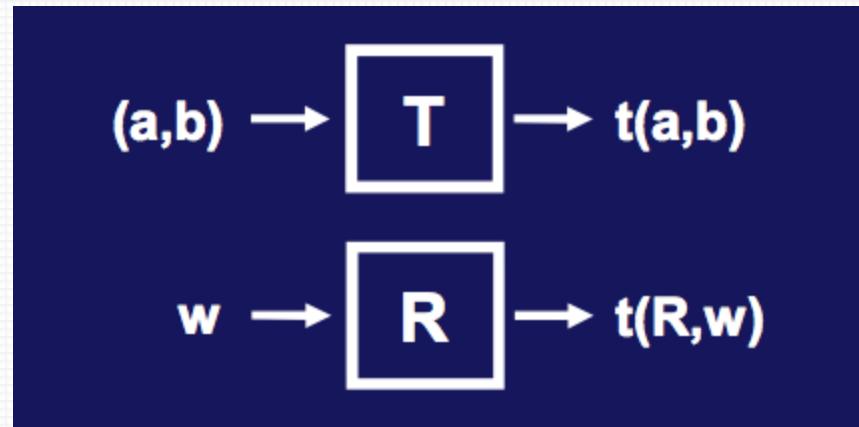
- 1. First A runs. It prints $\langle B \rangle$.
- 2. B starts. It looks at the tape and finds its input $\langle B \rangle$.
- 3. B computes $q(\langle B \rangle) = \langle S \rangle$ and combines that with $\langle B \rangle$ into a TM description $\langle \text{SELF} \rangle$.
- 4. B prints this description and halts.

- Introduction
- Self-Reference
- The Recursion Theorem
- Applications

The Recursion Theorem

- Let T be a TM that computes a function t : $\Sigma^* \times \Sigma^* \rightarrow \Sigma^*$. There is a TM R that computes r , where for every w ,

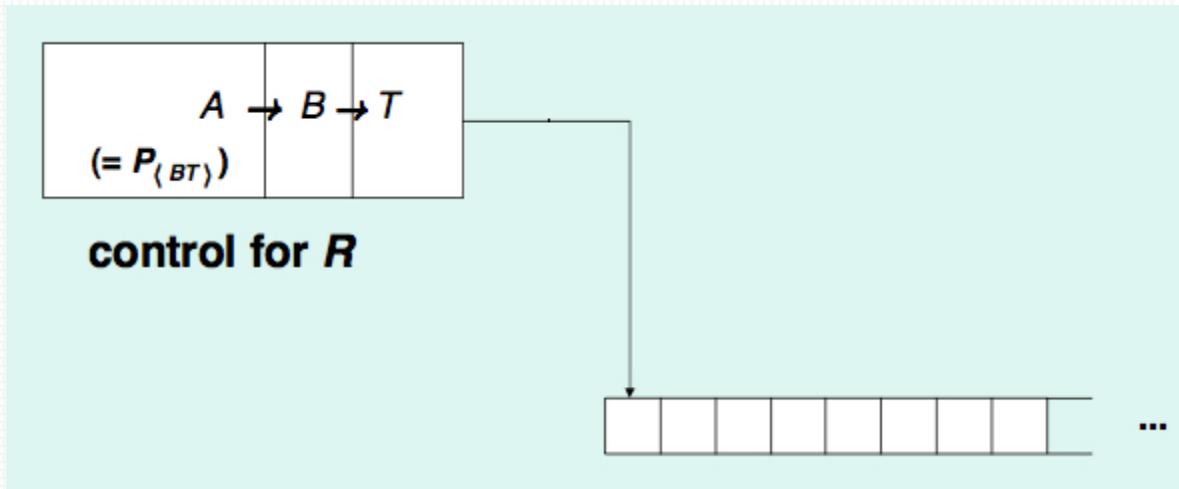
$$r(w) = t(R, w)$$



- What is this Theorem saying?
- Informally, a TM can obtain its own description and compute with it.
- To make a TM, that can obtain its own description and then compute with it
- 1. Make a TM T that receives the description of itself as an extra input.
- 2. Then the recursion theorem produces a new machine, R which operates as T does, with R's description filled in automatically.

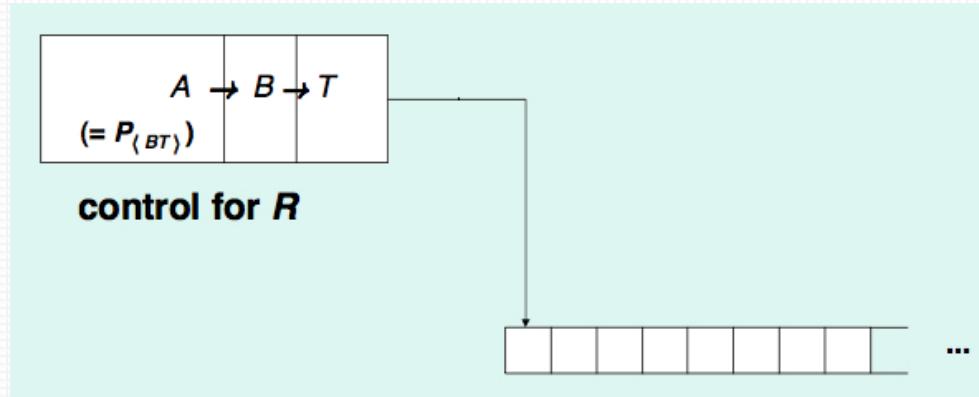
Proof Of The Recursion Theorem

- We construct a machine with 3 parts: A,B and T.



Proof Of The Recursion Theorem

- A is the TM $P_{\langle BT \rangle}$, described by $q(\langle BT \rangle)$
 - Technical point: We redesign q so that $P_{\langle BT \rangle}$ writes its output following any preexisting string on the tape.
- So, after A runs, the tape contains $w\langle BT \rangle$
- B examines the tape and applies q to $\langle BT \rangle$ getting $\langle A \rangle$.
- B then combines A, B and T into a single machine and obtains its description $\langle ABT \rangle = \langle R \rangle$
- It encodes these as $\langle R, w \rangle$ and places it on the tape and passes the control to T.



- Introduction
- Self-Reference
- The Recursion Theorem
- Applications

Significance Of The Recursion Theorem

- A_{TM} is undecidable.
- Suppose H decides A_{TM} , we construct B :
- $B = \text{"On input } w:$
 - 1 Obtain, via the recursion theorem, own description $\langle B \rangle$.
 - 2 Run H on input $\langle B, w \rangle$.
 - 3 Do the opposite of what H says.
 - accept if H rejects.
 - reject if H accepts.
- B conflicts with itself – hence can not exist.
- H can not exist.

- A computer virus is a computer program that is designed to spread itself among computers. When placed appropriately in a host computer, computer viruses can become activated and transmit copies of themselves to other accessible machines.
- In order to carry out its primary task of self-replication, a virus may contain the construction described in the proof of the recursion theorem.

- ✓ Introduction
- ✓ Self-Reference
- ✓ The Recursion Theorem
- ✓ Applications

Theory of Computation

Lesson 11-2

Turing Reducibility



王 轩
Wang
Xuan

- Introduction
- Oracle
- Turing Reducibility

- Reducibility: If A is reducible to B then we can solve A by solving B.
- Mapping Reducibility ($A \leq_m B$) : Use a computable mapping f to transform an instance of A to an instance of B.

- It turns out that Mapping Reducibility is not general enough!
 - Consider A_{TM} and \overline{A}_{TM}
 - Clearly the solution to one can be used as a solution to the other, by simply reversing the answer.
 - But \overline{A}_{TM} is **not** mapping reducible to A_{TM} because A_{TM} is Turing-recognizable while \overline{A}_{TM} is not.
 - We need a more general notion of reducibility

- Introduction
- Oracle
- Turing Reducibility

- An oracle for a language A is an external device that is capable of answering the question “Is $w \in A$?”
- We aren’t concerned with the way the oracle determines its responses. We use the term oracle to connote a magical ability and consider oracles for languages that aren’t decidable by ordinary algorithms.
- An oracle TM M^A is a modified TM, that has the capability of querying an oracle for language A.

Turing Reducibility

- Definition:
- Language A is **Turing reducible** to language B, written as $A \leq_T B$, if A is decidable relative to B (that is, using an oracle for B to decide A)
- Turing reducibility is a generalization of mapping reducibility $A \leq_M B$

- **Proof:**

Idea: Consider there is an oracle for A_{TM} , the following oracle TM P decides E_{TM} .

Input: $\langle M \rangle$ ($TM\ M$)

1. Construct the following TM N :

- $N = "On\ any\ Input"$
 - (a) Run M in parallel on all strings in Σ^*
 - (b) If M accepts any of these strings, accept."

2. Query the oracle to determine whether $\langle N, 0 \rangle \in A_{TM}$;

3. If the oracle returns "YES", reject; if "NO", accept.

- If M 's language isn't empty, N will accept every input and, in particular, input 0. Hence the oracle will answer YES, and oracle TM P will reject.
- If M 's language is empty, N will reject all inputs. Hence the oracle will answer NO, and oracle TM P will accept.
- Thus oracle TM P decides E_{TM} . We say that E_{TM} is Turing reducible to A_{TM} .

- Theorem

If $A \leq_T B$ and B is decidable, then A is decidable.

- Proof

If B is decidable, then replace the oracle by the TM for B. Thus we may replace the oracle Turing machine that decides A by an ordinary Turing machine that decides A.

Theory of Computation

A Definition of Information



王轩

Wang Xuan

- A Definition of Information
- Minimal Length Descriptions
- Compressible

A Definition of Information

- Consider the following two binary sequences
 - $A = 01$
 - $B = 1110010110100011101010000111010011010111$
- Intuitively, A contains little information: it is merely a repetition of the pattern 01 twenty times
- In contrast, B appears to contain more information

- We define the quantity of information contained in an object to be the size of that object's smallest representation or description
 - Sequence A contains little information: it has a small description
 - Sequence B apparently contains more information because it seems to have no concise description

- A Definition of Information
- Minimal Length Descriptions
- Compressible

Minimal Length Descriptions

- We describe a binary string x with a Turing machine M and a binary input w to M
- The length of the description is the combined length of representing M and w : $\langle M, w \rangle = \langle M \rangle_w$

- Let x be a binary string. The minimal description of x , denoted $d(x)$, is the **shortest string $\langle M, w \rangle$** where TM M on input w halts with x on its tape.
- If several such strings exist, select the lexicographically first among them.

- The descriptive complexity of x is

$$K(x) = |d(x)|$$

In other words, $K(x)$ is the length of the minimal description of x .

- It is intended to capture our intuition for the amount of information in the string x .
- The descriptive complexity of a string is at most a fixed constant more than its length.
- The constant is universal, it is not dependent on the string.

- Theorem 6.24 $\exists c \forall x [K(x) \leq |x| + c]$
- Proof:
 - Consider the following description of the string x .
 - Let M be a TM that halts as soon as it is started. This machine computes the identity function – its output is the same as input. A description of x is simply $\langle M \rangle x$. Letting c be the length of $\langle M \rangle$, completes the proof.

- Theorem 6.25 $\exists c \forall x [K(xx) \leq K(x) + c]$
- Proof:
- Consider the following Turing machine, which expects an input of the form $\langle N, w \rangle$, where N is a TM and w is an input for it.
- $M = \text{"On input } \langle N, w \rangle, \text{ where } N \text{ is a TM and } w \text{ is a string:}$
 - 1. Run N on w until it halts and produces an output string s .
 - 2. Output the string ss .
- A description of xx is $\langle M \rangle d(x)$. Recall that $d(x)$ is the minimal description of x . The length of this description is $|\langle M \rangle| + |d(x)|$, which is $c + K(x)$, where c is the length of $\langle M \rangle$.

- Theorem 6.26 $\exists c \forall x, y [K(xy) \leq 2K(x) + K(y) + c]$

- Proof:

- We construct a TM M that breaks its input into two separate descriptions. The bits of the first description $d(x)$ are **doubled** and terminated with string **01** before the second description $d(y)$ appears.
- The length of this description of **xy** is clearly $2K(x) + K(y) + c$.

$x = 1010010110, y = 001011$, double x and terminated with 01, then add y .

1100110000110011110001001011
Double x 01 y

- Theorem 6.26 $\exists c \forall x, y [K(xy) \leq 2K(x) + K(y) + c]$
- A more efficient method
- Instead of doubling the bits of $d(x)$, we may prepend the length of $d(x)$ as a binary integer that has been **doubled** to differentiate it from $d(x)$.
- The description still contains enough information to decode it into the two descriptions of x and y .
- The description now has length at most $2\log_2 K(x) + K(x) + K(y) + c$

$x = 1010010110, y = 001011$, length of x is 1010. Double the length of x and terminated with 01, then add x and y

11001100011010010110001011
Double of 01 x y
length of x

- A Definition of Information
- Minimal Length Descriptions
- Compressible

Compressible

- A string's minimal description is never much longer than the string itself.
- Of course for some strings, the minimal description may be much shorter if the information in the string appears sparsely or redundantly.
- Let x be a string. Say that x is c -compressible if $K(x) \leq |x| - c$.
- If x is not c -compressible, we say that x is **incompressible by c** .
- If x is incompressible by 1 , we say that x is **incompressible**.

- Incompressible strings of every length exist.
- IDEA:
 - The number of strings of length n is greater than the number of descriptions of length less than n .
 - Each description describes at most one string.
 - Therefore some string of length n is not described by any description of length less than n . That string is incompressible.

- ▶ Consider binary strings of length n .
- ▶ The number of binary strings of length is 2^n .
- ▶ Each description is a binary string, so the number of descriptions of length less than n is at most the sum of the number of strings of each length up to $n-1$.
- ▶ $\sum_{0 \leq i \leq n-1} 2^i = 1 + 2 + 3 + 4 + \dots + 2^{n-1} = 2^n - 1$
- ▶ Therefore at least one string of length n is incompressible.

- At least $2^n - 2^{n-c+1} + 1$ strings of length n are incompressible by c .
- Proof:
- At most $2^{n-c+1} - 1$ strings of length n are c -compressible, because at most that many descriptions of length at most $n-c$ exist.
- The remaining $2^n - (2^{n-c+1} - 1)$ are incompressible by c .

Theory of Computation

Lesson 12

Time Complexity



王轩

Wang Xuan

- Measuring Complexity
- BIG-O and SMALL-O notation
- Analyzing Algorithms
- Complexity Relationships Among Models

- Language $A=\{0^k1^k \mid k \geq 0\}$.
- How much time does a single-tape Turing machine need to decide A?

M_1 = “On input string w :

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat the following if both 0s and 1s remain on the tape.
 3. Scan across the tape, crossing off a single 0 and a single 1.
 4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*.”

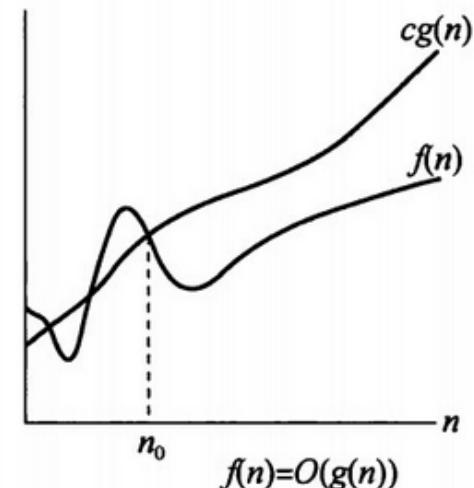
Time complexity

- Let M be a deterministic Turing machine that halts on all inputs.
- The **running time** or **time complexity** of M is the function $f: N \rightarrow N$, where $f(n)$ is the maximum number of steps that M uses on any input of length n .
- M is an $f(n)$ time Turing machine.

- Measuring Complexity
- BIG-O and SMALL-O notation
- Analyzing Algorithms
- Complexity Relationships Among Models

Big-O Notation

- Asymptotic analysis
 - $f(n) = 6n^3 + 2n^2 + 2n + 45$
 - f is asymptotically at most n^3 . The asymptotic notation or big-O notation for describing this relation $f(n)=O(n^3)$.
- Definition
 - Let f and g be functions $f, g: \mathbb{N} \rightarrow \mathbb{R}^+$. Say that $f(n)=O(g(n))$ if positive integers c and n_0 exist such that for every integer $n \geq n_0$, $f(n) \leq c g(n)$.
 - When $f(n)=O(g(n))$ we say that $g(n)$ is an **upper bound** for $f(n)$, or **asymptotic upper bound**.



Example

► $f_1(n) = 5n^3 + 2n^2 + 22n + 6.$

- Let $c=6$, $n_0=12$. Then $5n^3 + 2n^2 + 22n + 6 < 6n^3$.
- $f_1(n) = O(n^3)$.
- In addition, $f_1(n) = O(n^4)$.

► $f_2(n) = 3n\log_2 n + 5n\log_2 \log_2 n + 2$

- $f_2(n) = O(n \log n)$

► $f(n) = O(n^2) + O(n)$

- $f(n) = O(n^2)$

► Bounds of the form n^c for $c \geq 0$, called **polynomial bounds**

► Bounds of the form $2^{(n^\delta)}$ for $\delta \geq 0$, called **exponential bounds**

Small-O Notation

- Let f and g be function $f, g: N \rightarrow R^+$. Say that $f(n)=o(g(n))$ if

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

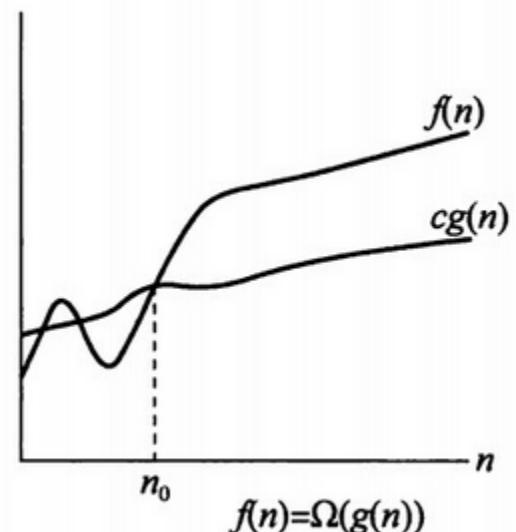
- In other word, $f(n)=o(g(n))$ means that, for any real number $c>0$, a number n_0 exists, where $f(n) < cg(n)$ for all $n \geq n_0$.
- Example

- $-\sqrt{n} = o(n)$.

- $n=o(n \log(\log n))$.

- $n \log(\log n)=o(n \log n)$

- $n \log n=o(n^2)$.



- Measuring Complexity
- BIG-O and SMALL-O notation
- Analyzing Algorithms
- Complexity Relationships Among Models

Analyzing algorithms

- Language $A=\{0^k 1^k \mid k \geq 0\}$.

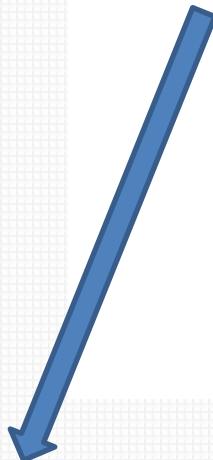
M_1 = “On input string w :

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat the following if both 0s and 1s remain on the tape.
 3. Scan across the tape, crossing off a single 0 and a single 1.
 4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*.”

- Language $A = \{0^k 1^k \mid k \geq 0\}$.

M_1 = “On input string w :

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat the following if both 0s and 1s remain on the tape.
 3. Scan across the tape, crossing off a single 0 and a single 1.
 4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*.”

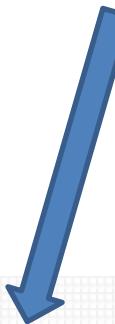


- In stage 1, use $2n$ steps. We say it use $O(n)$ steps.

- Language $A = \{0^k 1^k \mid k \geq 0\}$.

M_1 = “On input string w :

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat the following if both 0s and 1s remain on the tape.
 3. Scan across the tape, crossing off a single 0 and a single 1.
 4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*.”



- In stage 2 and 3, each scan use $O(n)$ steps.
- At most $n/2$ scans can occur.
- So total time by stage 2 and 3 is $O(n^2)$ steps.

► Language $A = \{0^k 1^k \mid k \geq 0\}$.

M_1 = “On input string w :

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat the following if both 0s and 1s remain on the tape.
 3. Scan across the tape, crossing off a single 0 and a single 1.
 4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*.”



- In stage 4, make a single scan. The time is $O(n)$.
- Thus the total time of M_1 on input of length n is $O(n) + O(n^2) + O(n) = O(n^2)$.

Time complexity class

- Let $t: N \rightarrow R^+$ be a function. Define the **time complexity class $\text{TIME}(t(n))$** , to be the collection of all languages that are decidable by an $O(t(n))$ time Turing machine.

Other example problems in the same class

TIME(n)

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

$$\{ab^n aba : n, k \geq 0\}$$

$$\{b^n : n \text{ is even}\}$$

$$\{b^n : n = 3k\}$$

Examples in class:

$\text{TIME}(n^2)$

$\{a^n b^n : n \geq 0\}$

$\{ww^R : w \in \{a,b\}^*\}$

$\{ww : w \in \{a,b\}^*\}$

Examples in class:

TIME (n^3)

CYK algorithm

$L_2 = \{\langle G, w \rangle : w \text{ is generated by}$
context - free grammar $G\}$

Matrix multiplication

$L_3 = \{\langle M_1, M_2, M_3 \rangle : n \times n \text{ matrices}$
and $M_1 \times M_2 = M_3\}$

- Example:
 - $A = \{0^k 1^k \mid k \geq 0\}$.
 - $A \in \text{TIME}(n^2)$, $\text{TIME}(n^2)$ contains all languages that can be decided in $O(n^2)$ time.
- Can another TM decide A faster?

M_2 = "On input string w :

1. Scan across the tape and *reject* if a 0 is found to the right of a 1. $\rightarrow O(n)$
 2. Repeat the following as long as some 0s and some 1s remain on the tape. $\rightarrow 1 + \log_2 n$
 3. Scan across the tape, checking whether the total number of 0s and 1s remaining is even or odd. If it is odd, *reject*. $\rightarrow O(n)$
 4. Scan again across the tape, crossing off every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1.
 5. If no 0s and no 1s remain on the tape, *accept*. Otherwise, *reject*." $\rightarrow O(n)$
- $O(n) \quad O(n \log_2 n)$

- The running time of M_2 is $O(n) + O(n \log n) = O(n \log n)$
- $A \in \text{TIME}(n \log n)$

Multitape Turing Machine

M_3 = “On input string w :

1. Scan across the tape and *reject* if a 0 is found to the right of a 1. $\rightarrow O(n)$
2. Scan across the 0s on Tape 1 until the first 1. At the same time, $\rightarrow O(n)$ copy the 0s onto Tape 2.
3. Scan across the 1s on Tape 1 until the end of the input. For each 1 read on Tape 1, cross off a 0 on Tape 2. If all 0s are crossed off $\rightarrow O(n)$ before all the 1s are read, *reject*.
4. If all the 0s have now been crossed off, *accept*. If any 0s remain, $\rightarrow O(n)$ *reject*.”

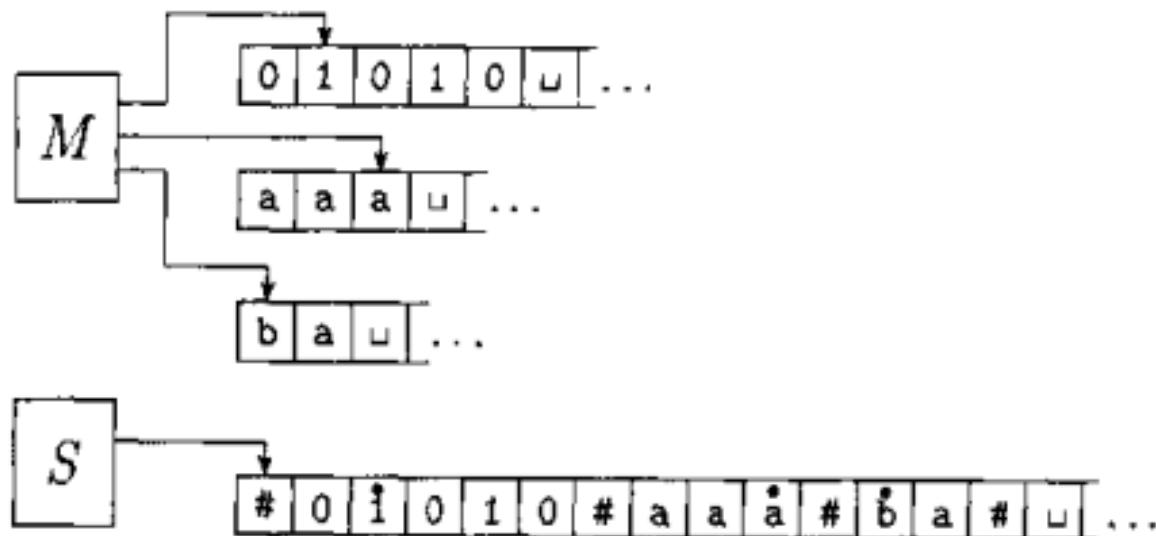
- The total time is $O(n)$.
- We observe:
 - A single-tape TM decides A in $O(n \log n)$ time.
 - A two-tape TM decides A in $O(n)$ time.

- Measuring Complexity
- BIG-O and SMALL-O notation
- Analyzing Algorithms
- Complexity Relationships Among Models

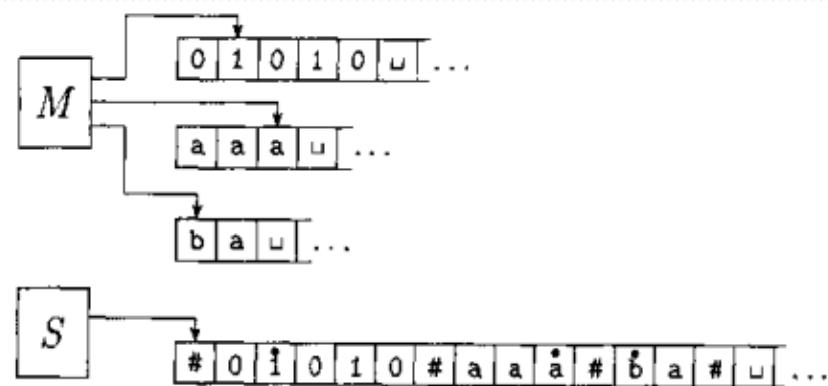
- The same language may have different time requirements on different models.
 - Single-tape Turing machine
 - Multitape Turing machine
 - Nondeterministic Turing machine
- Let $t(n)$ be a function, where $t(n) \geq n$.
 - 1. Every $t(n)$ time multitape Turing machine has an equivalent $O(t^2(n))$ time single-tape Turing machine.
 - 2. Every $t(n)$ time nondeterministic single-tape Turing machine has an equivalent $2^{O(t(n))}$ time single-tape Turing machine.

Multitape & single-tape TM

- Let M be a k -tape TM that runs in $t(n)$ time.
We construct a single-tape TM that runs in $O(t^2(n))$ time.
- Machine S operates by simulating M .



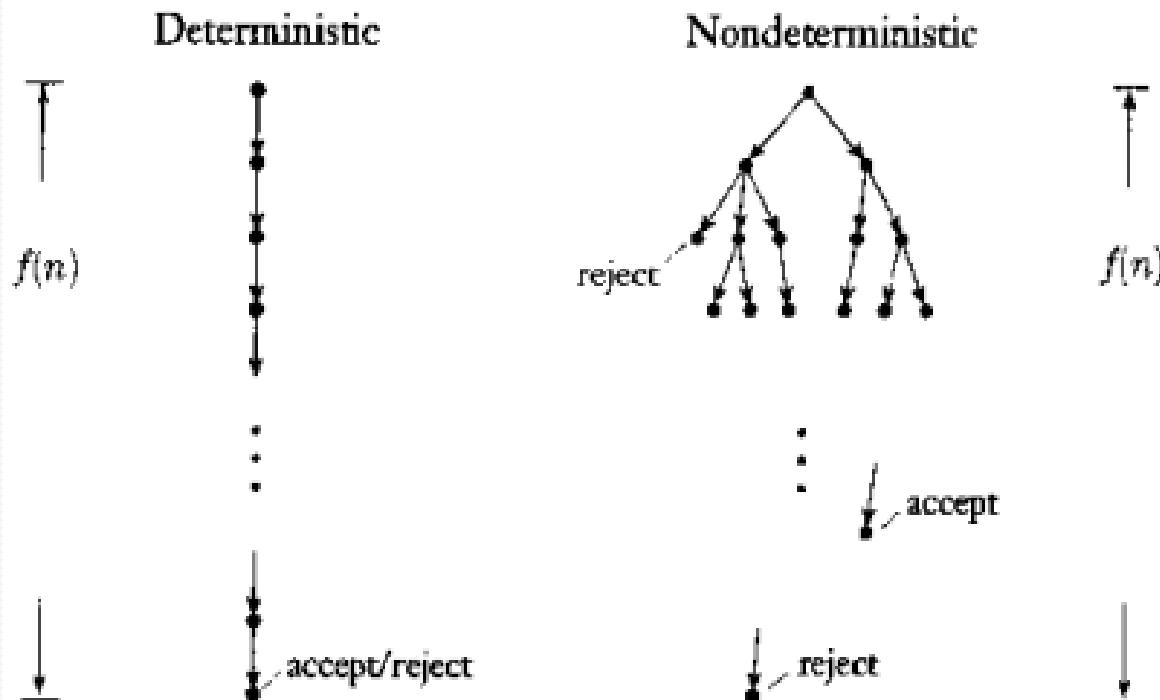
- For each step of M
 - S scan all the information on its tape to determine the symbols under M 's tape heads.
 - Scan again to update the tape content and head position.
- The length of active portions of S 's tape is $O(t(n))$
 - Each of the active portions of M 's k tapes has length at most $t(n)$. Because M use $t(n)$ tape cells in $t(n)$ step if the head moves rightward at every step and even fewer if a head ever moves left.
 - S 's tape = $kt(n)=O(t(n))$



- Simulation
 - The initial stage, where S put its tape into the proper format, uses $O(n)$ steps.
K-tape TM: $\{n_1, n_2, \dots, n_k\} \leq kn_{max} = O(n)$
 - S simulates each of the $t(n)$ steps of M, using $O(t(n))$ steps, so its part of simulation uses $t(n)*O(t(n))=O(t^2(n))$.
- Therefore the entire simulation of M uses $O(t^2(n))$.

Nondeterministic TM

- Let N be a nondeterministic TM that is a decider.
- The running time of N is the function $f: N \rightarrow N$, where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n .



Proof idea :

1. Change nondeterministic TM to a multitape TM.(Theorem 3.16)
2. Change multitape TM to a single-tape TM

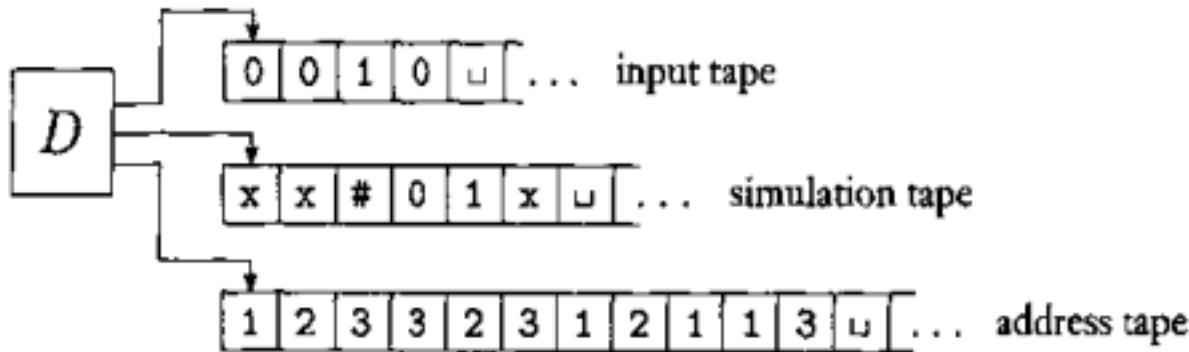
Proof: Step-1

- Let N be a NTM running in $t(n)$ time. We construct a deterministic TM D that simulates N .
 - On an input of length n , each branch of N' tree has a length of at most $t(n)$.
 - Every node can have at most b children.
 - The total number of leaves in the tree is at most $b^{t(n)}$.
- Simulation by exploring the tree first.
 - The total number of nodes in the tree is less than twice the maximum number of leaves, $O(b^{t(n)})$.
 - The time for starting from the root to a node is $O(t(n))$.



The total number of nodes is no more than $1+b+b^2+\dots+b^{t(n)} = \frac{b^{t(n)+1}-1}{b-1}$,

$$\begin{aligned}\frac{b^{t(n)+1}-1}{b-1} &\leq 2b^{t(n)} \Leftrightarrow b^{t(n)+1}-1 \leq 2b^{t(n)}(b-1) \Leftrightarrow b^{t(n)+1}-1 \leq 2b^{t(n)+1}-2b^{t(n)} \\ &\Leftrightarrow b^{t(n)+1}-2b^{t(n)}+1 \geq 0 \Leftrightarrow (b-2) \cdot b^{t(n)}+1 \geq 0\end{aligned}$$



- Thus the running time of D is $O(t(n)b^{t(n)}) = 2^{O(t(n))}$.

$$\log(t(n)b^{t(n)}) = \log(t(n)) + t(n)\log(b)$$

$$\text{so } O(t(n)b^{t(n)}) = 2^{O(t(n))}$$

$O(t(n) \cdot b^{t(n)}) = 2^{O(t(n))} \Leftrightarrow \log_2 O(t(n) \cdot b^{t(n)}) = O(t(n))$

$$\log_2 O(t(n) \cdot b^{t(n)}) = O(\log_2(t(n)) + t(n) \cdot \log_2 b) =$$

$$O(\log_2(t(n))) + \log_2 b \cdot O(t(n)) = O(t(n))$$

- TM D has three tapes. Converting to a single-tape TM at most squares the running time. Thus the running time of the single-tape simulator is $(2^{O(t(n))})^2 = 2^{O(2t(n))} = 2^{O(t(n))}$.

Theory of Computation

Lesson 13

The Class P



王轩

Wang Xuan

Polynomial time

- Polynomial difference between the time complexity of problems measured on deterministic single-tape and multitape TMs.
- Exponential difference between the time complexity of problems measured on deterministic and nondeterministic TMs.
- All reasonable deterministic computational models are polynomially equivalent.

- P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$P = \bigcup_{k>0} \text{TIME}(n^k)$$

- P is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape Turing machine.
- P roughly corresponds to the class of problems that are realistically solvable on a computer.

Class P

$\{a^n b\}$

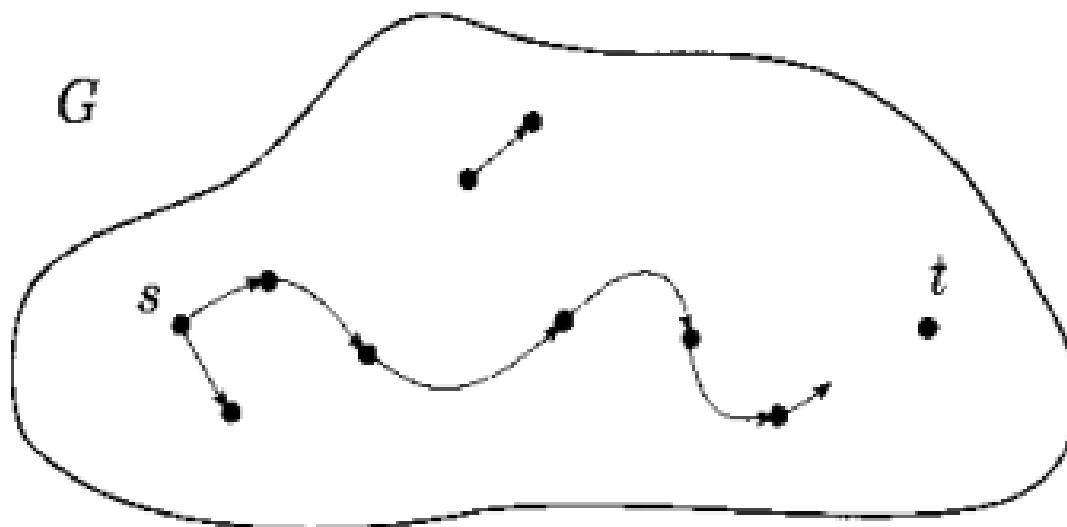
$\{a^n b^n\}$

$\{ww\}$

CYK-algorithm

Matrix multiplication

- PATH={ $\langle G, s, t \rangle$ | G is a directed graph that has a directed path from s to t}.



A solution: search exhaustively all paths

$L = \{<G,s,t>: \text{there is a path in } G \text{ from } s \text{ to } t\}$

$$L \in \text{TIME}(m^m)$$

m is the number of nodes in G .

Exponential time

Intractable problem

Polynomial time algorithm

- A polynomial time algorithm M for PATH operates

$M =$ “On input $\langle G, s, t \rangle$ where G is a directed graph with nodes s and t :

1. Place a mark on node s .
2. Repeat the following until no additional nodes are marked.
3. Scan all the edges of G . If an edge (a, b) is found going from a marked node a to an unmarked node b , mark node b .
4. If t is marked, accept. Otherwise, reject.”

- The total number of stages is at most $1+1+m$, giving a polynomial in the size of G .

- RELPRIME={ $\langle x, y \rangle \mid x \text{ and } y \text{ are relatively prime}$ }.
 - Two numbers are relatively prime if 1 is the largest integer that evenly divides them both.
- Proof
 - The Euclidean algorithm E is as follows.

E = “On input $\langle x, y \rangle$, where x and y are natural numbers in binary:

1. Repeat until $y = 0$.
2. Assign $x \leftarrow x \bmod y$.
3. Exchange x and y .
4. Output x .

- Algorithm R solves RELPRIME, using E as a subroutine

R = "On input $\langle x, y \rangle$, where x and y are natural numbers in binary:

 1. Run E on $\langle x, y \rangle$.
 2. If the result is 1, accept. Otherwise, reject."
- For every execution of stage 2 in E , cut the value of x by at least half.
- Thus, the maximum number of time that stage 2 and 3 are executed is the lesser of $2\log_2 x$ and $2\log_2 y$.
- The total running time is polynomial.

$$x \bmod y \leq \frac{x}{2}$$

①. $y \leq \frac{x}{2}$, $x \bmod y < y \leq \frac{x}{2}$

②. $y > \frac{x}{2}$, $2y > x \Leftrightarrow x \bmod y = x - y \leq \frac{x}{2}$



- Every context-free language is a member of P.
- Let L be a CFL generated by CFG G that is in Chomsky normal form.
 - All derivation of a string w has $2n-1$ steps.
 - The decider tries all possible derivations with $2n-1$ steps .
 $S \rightarrow A_1 A_2 A_3 \dots \dots A_k$, if there are m terminal symbols, the number of derivation is m^k
- It doesn't run in polynomial time
- Instead, we use **dynamic programming** technique.
 - Solve subproblems, and record them.

- The algorithm enters the solution to this subproblem in an $n \times n$ table.
- For $i \leq j$ the (i,j) entry of the table contains the **collection of variables** that generate the substring $w_i w_{i+1} \dots w_j$. For $i > j$ the table entries are unused.

$i \backslash j$			

- Let G be a CFG in Chomsky normal form generating the CFL L . Assume that S is the start variable.

D = “On input $w = w_1 \cdots w_n$:

1. If $w = \epsilon$ and $S \rightarrow \epsilon$ is a rule, accept. [handle $w = \epsilon$ case]
2. For $i = 1$ to n , [examine each substring of length 1]
3. For each variable A ,
4. Test whether $A \rightarrow b$ is a rule, where $b = w_i$.
5. If so, place A in $\text{table}(i, i)$.
6. For $l = 2$ to n , [l is the length of the substring]
7. For $i = 1$ to $n - l + 1$, [i is the start position of the substring]
8. Let $j = i + l - 1$, [j is the end position of the substring]
9. For $k = i$ to $j - 1$, [k is the split position]
10. For each rule $A \rightarrow BC$,
11. If $\text{table}(i, k)$ contains B and $\text{table}(k + 1, j)$ contains C , put A in $\text{table}(i, j)$.
12. If S is in $\text{table}(1, n)$, accept. Otherwise, reject.”

- Let G be a CFG in Chomsky normal form generating the CFL L . Assume that S is the start variable.

D = “On input $w = w_1 \cdots w_n$:

1. If $w = \epsilon$ and $S \rightarrow \epsilon$ is a rule, accept. [handle $w = \epsilon$ case]
 2. For $i = 1$ to n , [examine each substring of length 1]
 3. For each variable A ,
 4. Test whether $A \rightarrow b$ is a rule, where $b = w_i$. $\rightarrow O(n)$
 5. If so, place A in $table(i, i)$.
 6. For $l = 2$ to n , [l is the length of the substring] $\rightarrow O(n)$
 7. For $i = 1$ to $n - l + 1$, [i is the start position of the substring] $\rightarrow O(n)$
 8. Let $j = i + l - 1$, [j is the end position of the substring]
 9. For $k = i$ to $j - 1$, [k is the split position] $\rightarrow O(n)$
 10. For each rule $A \rightarrow BC$, $\rightarrow r$ times(r is the number of rules)
 11. If $table(i, k)$ contains B and $table(k + 1, j)$ contains C , put A in $table(i, j)$.
 12. If S is in $table(1, n)$, accept. Otherwise, reject.”
- $O(n^3)$

Theory of Computation

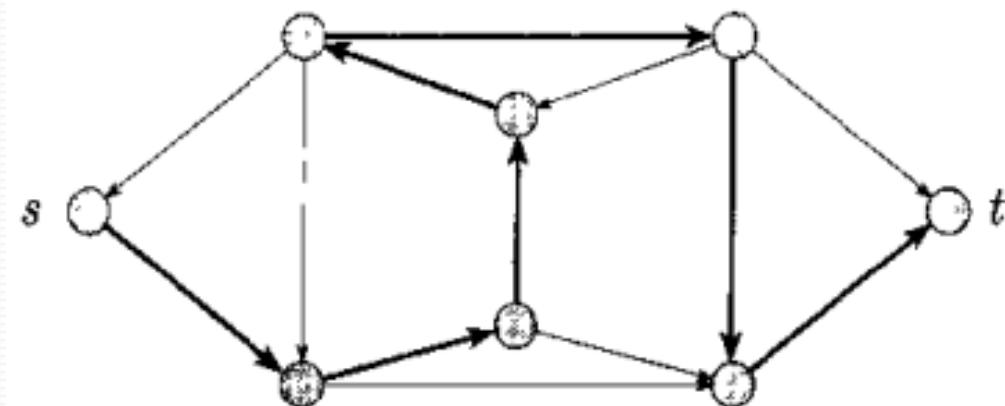
The Class NP



王轩

Wang Xuan

- A **Hamiltonian path** in a directed graph G is a directed path that goes through each node exactly once.
- $\text{HAMPATH} = \{ \langle G, s, t \rangle \mid G \text{ is a directed graph with a Hamiltonian path from } s \text{ to } t \}$.



- We can obtain an **exponential** time algorithm for the HAMPATH problem.
- Don't know whether HAMPATH is solvable in **polynomial** time.
- But we can check to verify whether the potential path is Hamiltonian in **polynomial** time.
- Verifying the existence of a Hamiltonian path may be easier than determining its existence.

- A natural number is **composite** if it is the product of two integers greater than 1. (合数)
- COMPOSITES={ $x \mid x = pq$, for integers $p, q > 1$ }
- It is a **polynomially verifiable** problem.
- But some are not polynomially verifiable.
 - for example $\overline{HAMPATH}$

- ▶ A **Verifier** for a language A is an algorithm V, where
 $A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}$.
- ▶ A **polynomial time verifier** runs in polynomial time in the length of w.
- ▶ A language A is **polynomially verifiable** if it has a polynomial time verifier.
- ▶ The information represented by c is called a **certificate**, or **proof**, of membership of A. (polynomial length)
 - A certificate for the composite number x simply is one of its divisors.

- NP is the class of languages that have polynomial time verifiers.
 - HAMPATH and COMPOSITES are members of NP.
 - COMPOSITES is a member of P which is a subset of NP.
- NP comes from **nondeterministic polynomial time**, and is derived from an alternative characterization by using **nondeterministic polynomial time Turing machines**.

- A nondeterministic TM decides the HAMPATH problem in nondeterministic polynomial time.

N_1 = “On input $\langle G, s, t \rangle$, where G is a directed graph with nodes s and t :

1. Write a list of m numbers, p_1, \dots, p_m , where m is the number of nodes in G . Each number in the list is nondeterministically selected to be between 1 and m .
2. Check for repetitions in the list. If any are found, *reject*.
3. Check whether $s = p_1$ and $t = p_m$. If either fail, *reject*.
4. For each i between 1 and $m - 1$, check whether (p_i, p_{i+1}) is an edge of G . If any are not, *reject*. Otherwise, all tests have been passed, so *accept*.”

- ▶ A language is in NP iff it is decided by some nondeterministic polynomial time Turing machine.
- ▶ Idea:
- ▶ How to convert a polynomial time verifier to an equivalent polynomial time NTM and vice versa.
 - NTM simulates the verifier by guessing the certificate.
 - The verifier simulates NTM by using the accepting branch as the certificate.

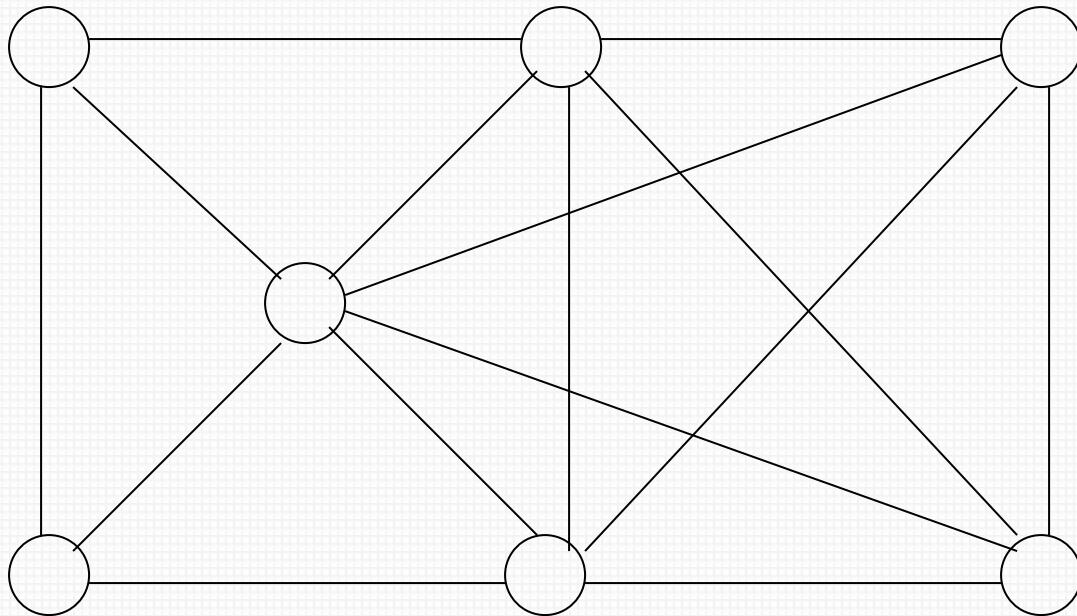
- Let V be the polynomial time verifier for language A in NP. Assume that V is a TM that runs in time n^k and construct N as follow.
- $N = \text{"On input } w \text{ of length } n:$
 - 1. Nondeterministically select string c of length at most n^k .
 - 2. Run V on input $\langle w, c \rangle$.
 - 3. If V accepts, accept; otherwise, reject."

- Assume that A is decided by a polynomial time NTM N and construct a polynomial time verifier V as follows.
- V=“On input $\langle w, c \rangle$, where w and c are strings:
 - 1. Simulate N on input w, treating each symbol of c as a description of the nondeterministic choice to make at each step.
 - 2. If this branch of N’s computation accepts, accept; otherwise, reject.”

- $\text{NTIME}(t(n)) = \{ L \mid L \text{ is a language decided by an } O(t(n)) \text{ time nondeterministic Turing machine}\}$.

$$NP = \bigcup_k \text{NTIME}(n^k)$$

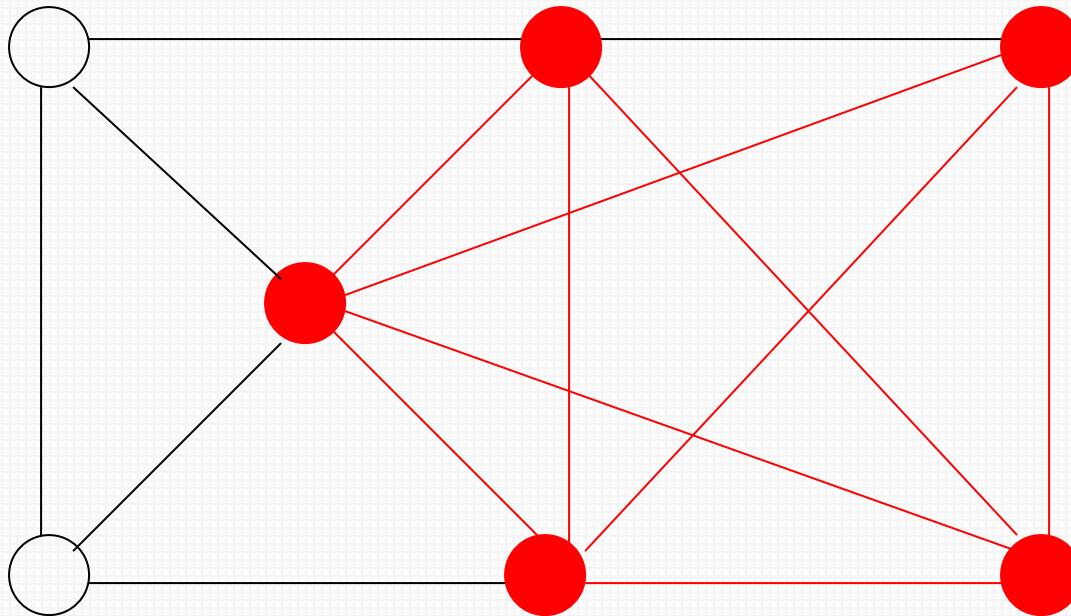
The clique problem



Clique: a clique in an undirected graph is subgraph, wherein every two nodes are connected by an edge.
A k -clique is a clique that contains k nodes.

Does there exist a clique of size 5?

The clique problem



Does there exist a clique of size 5?

- ▶ CLIQUE = { $\langle G, k \rangle$ | G is undirected graph with a k -clique}.
- ▶ Proof:
- ▶ Construct a polynomial time verifier V for CLIQUE.
- ▶ $V =$ “On input $\langle\langle G, k \rangle, c \rangle$:
 - 1. Test whether c is a set of k nodes in G .
 - 2. Test whether G contains all edges connecting nodes in C .
 - 3. If both pass, accept; otherwise, reject.

- Prove using nondeterministic polynomial time Turing machines.
- $N = \text{"On input } \langle G, k \rangle, \text{ where } G \text{ is a graph:}$
 - 1. Nondeterministically select a subset c of k nodes of G .
 - 2. Test whether G contains all edges connecting nodes in c .
 - 3. If yes, accept; otherwise, reject."

- ▶ SUBSET-SUM = { $\langle S, t \rangle$ | $S = \{x_1, \dots, x_k\}$ and for some $\{y_1, \dots, y_l\} \subseteq \{x_1, \dots, x_k\}$, we have $\sum y_i = t$ }.
- $\{x_1, \dots, x_k\}$ and $\{y_1, \dots, y_l\}$ are **multisets** and allow repetition of elements.
- ▶ For example
 - $\langle \{4, 11, 16, 21, 27\}, 25 \rangle \in \text{SUBSET-SUM}$ because $4 + 21 = 25$.

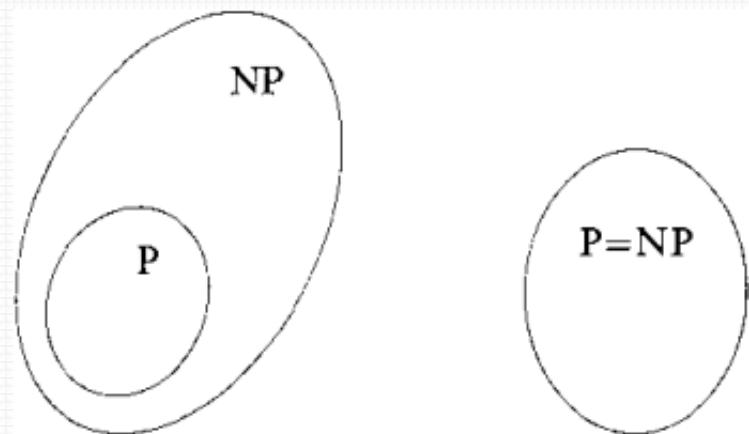
► Construct a polynomial time verifier V

- V=“On input $\langle\langle S, t \rangle, c \rangle$:
 - 1. Test whether c is a collection of numbers that sum to t.
 - 2. Test whether S contains all the numbers in c.
 - 3. If both pass, accept; otherwise, reject.

► Construct a nondeterministically polynomial time TM.

- N=“ On input $\langle S, t \rangle$:
 - 1. Nondeterministically select a subset c of the numbers in S.
 - 2. Test whether c is a collection of numbers that sum to t.
 - 3. If the test passes, accept; otherwise, reject.

- P = the class of languages for which membership can be **decided** quickly.
- NP = the class of languages for which membership can be **verified** quickly.
- P=NP ? Two possibilities.



- ▶ The question of whether P=NP is one of the greatest unsolved problems in theoretical computer science and contemporary mathematics.
- ▶ The best method known for solving languages in NP deterministically uses exponential time. In other words, we can prove that:

$$\text{NP} \subseteq \text{EXPTIME} = \bigcup_k \text{TIME}(2^{n^k})$$

- ▶ But we don't know whether NP is contained in a smaller deterministic time complexity class.

Theory of Computation

NP-Completeness



王轩

Wang Xuan

- There are **certain problems in NP** whose complexity is related to that of the entire class.
- If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable.
- These problems are call **NP-complete**.

Satisfiability problem

- A **Boolean formula** is an expression involving Boolean variables and operations
 - An example: $\phi = (\bar{x} \vee y) \wedge (x \vee \bar{z})$
 - Boolean variables: take on values TRUE and FALSE
 - Operations: AND, OR and NOT
- A Boolean formula is **satisfiable** if some assignment of 0s and 1s to variables makes the formula equals 1.
 - ϕ is satisfiable if $x=0, y=1, z=0$.

- The **satisfiability problem** is to test whether a Boolean formula is satisfiable.
 - $SAT = \{<\phi> \mid \phi \text{ is a satisfiable Boolean formula}\}$.
- **Satisfiability problem** is a NP-complete problem.
- Cook-Levin theorem: $SAT \in P$ iff $P=NP$.

Polynomial Time Reducibility

A function $f : \Sigma^* \rightarrow \Sigma^*$ is a
Polynomial time Computable function

if some polynomial time Turing machine M
exists that halts with just $f(w)$ on its tape,
when started on any input w .

Definition:

Language A

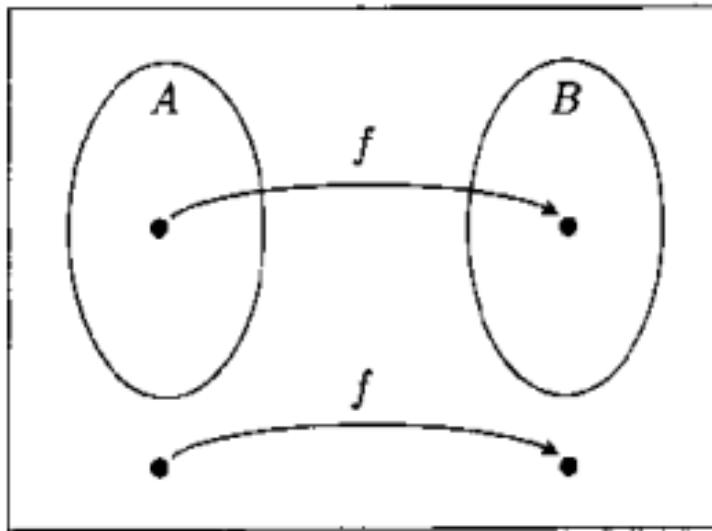
is polynomial time reducible to

language B , written $A \leq_p B$

if there is a polynomial time computable
function f such that:

$$w \in A \iff f(w) \in B$$

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



- To test $w \in A$, we use the reduction f to map w to $f(w)$ and test whether $f(w) \in B$.

Theorem:

If $A \leq_p B$ and $B \in P$, then $A \in P$.

Proof:

Let M be the machine that decides B
in polynomial time

Machine M' to decide A in polynomial time:

On input string w :

1. Compute $f(w)$
2. Run M on input $f(w)$
3. If $f(w) \in B$ accept w

A language B is NP-complete if:

- B is in NP, and
- Every language A in NP
is polynomial time reduced to B

Observation:

If a NP-complete language
is proven to be in P then:

$$P = NP$$

If B is NP-complete and $B \leq_p C$ for C in NP, then C is NP-complete

- B is NP-complete.
- Every language A in NP is polynomial time reducible to B
- B is polynomial time reducible to C .
- Then, every language A in NP is polynomial time reducible to C .

Theory of Computation

Lesson14-1

Cook-Levin Theorem



王轩

Wang Xuan

Cook-Levin theorem



Cook in 1968



Cook in 2008

Stephen Arthur Cook

Stephen Arthur Cook, born December 14, 1939 is an American-Canadian computer scientist and mathematician who has made major contributions to the fields of **complexity theory** and **proof complexity**.

Stephen Cook is considered one of the forefathers of **computational complexity theory**. During his PhD, Cook worked on complexity of functions, mainly on multiplication. In his seminal 1971 paper "The Complexity of Theorem Proving Procedures", Cook formalized the notions of **polynomial-time reduction** and **NP-completeness**, and proved the existence of an **NP-complete problem** by showing that the Boolean satisfiability problem (usually known as **SAT**) is NP-complete. This theorem was proven independently by **Leonid Levin** in the Soviet Union, and has thus been given the name the **Cook-Levin theorem**.

In **1982**, Cook received the **Turing award** for his contributions to complexity theory.

Cook-Levin theorem



Leonid Levin in 2010

Leonid Levin

Leonid Anatolievich Levin, born November 2, 1948 is a Soviet-American computer scientist.

He is known for his work in randomness in computing, algorithmic complexity and intractability, algorithmic probability, theory of computation, and information theory.

He and Stephen Cook independently discovered the existence of **NP-complete problems**. This NP-completeness theorem, often called the Cook–Levin theorem, was a basis for **one of the seven Millennium Prize Problems** declared by the Clay Mathematics Institute with a \$1,000,000 prize offered. The Cook–Levin theorem was a breakthrough in computer science and an important step in the development of the **theory of computational complexity**.

Levin was awarded the **Knuth Prize** in 2012 for his discovery of **NP-completeness** and the development of **average-case complexity**.

Theorem (Cook-Levin): SAT is NP-complete

Corollary: $\text{SAT} \in \text{P}$ if and only if $\text{P} = \text{NP}$

- Boolean formula: an expression involving Boolean variable and operation.

$$\emptyset = (\bar{x} \wedge y) \vee (x \wedge \bar{z})$$

- Satisfiable: if some assignment of 0s and 1s to the variables makes the formula evaluate to 1.
- Define:

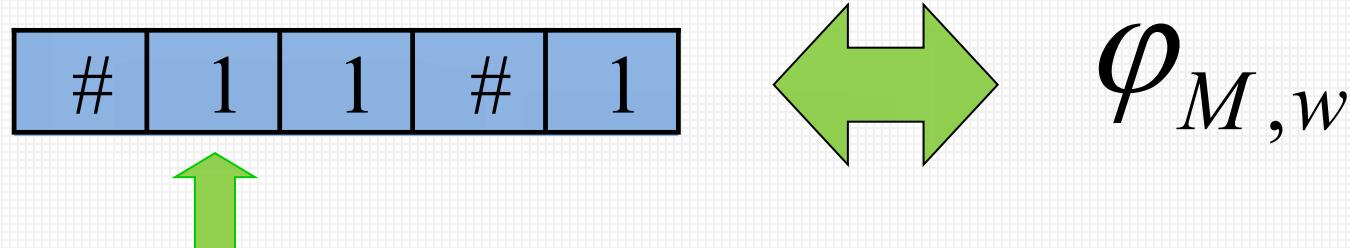
SAT= $\{\{\emptyset\} | \emptyset \text{ is a satisfiable Boolean formula}\}$

- Problem: To decide if the formula is satisfiable.

- Step 1: SAT is in NP
- Idea: A nondeterministic polynomial time machine can guess an assignment to a given formula \emptyset and accept if the assignment satisfies \emptyset .

- Step 2: SAT is NPC
- Proof idea:

For any NP machine M and any input string w , we construct a Boolean formula $\varphi_{M,w}$ which is satisfiable iff M accepts w .



Representing a Computation by a Configurations Table

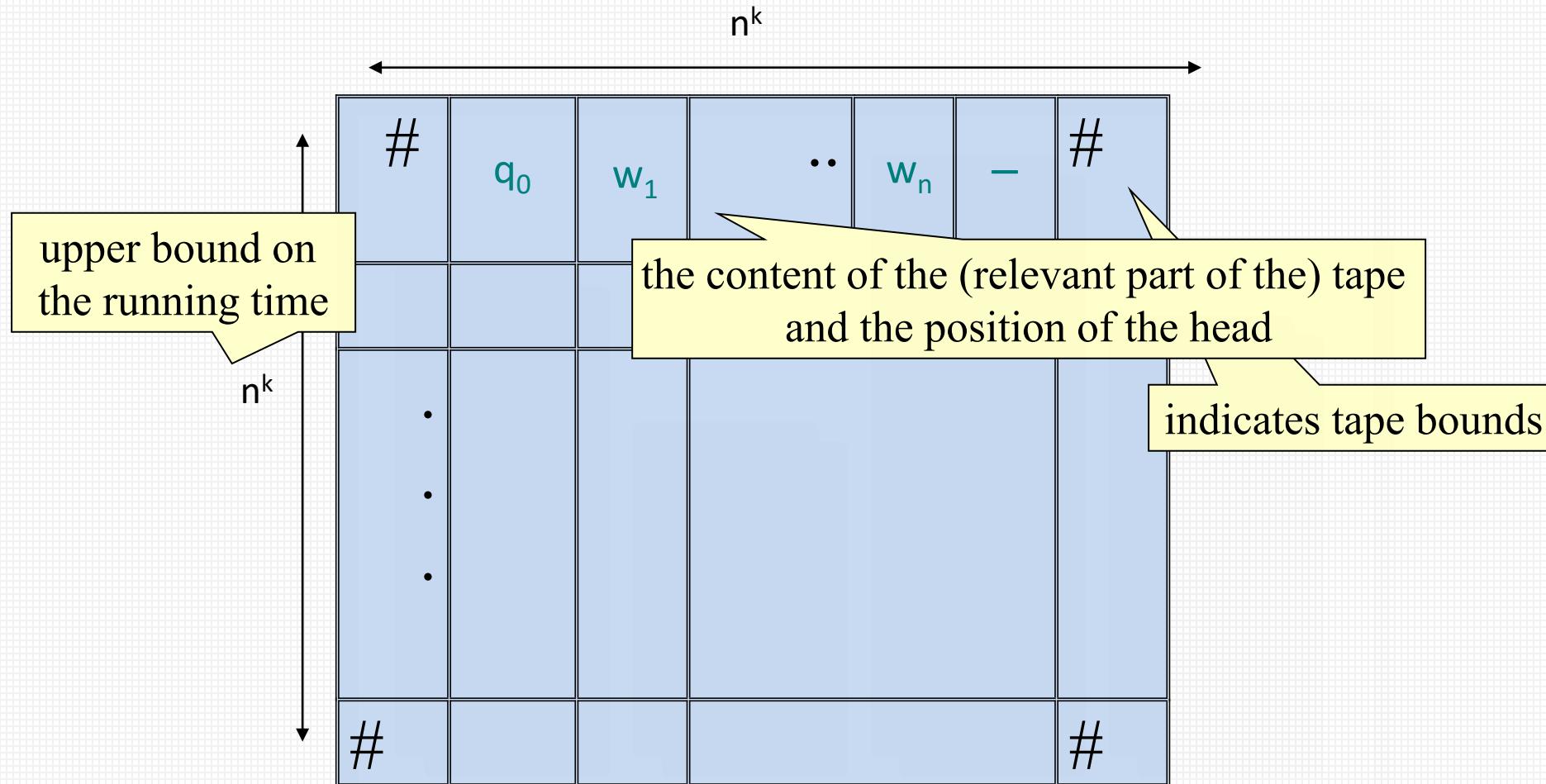


Tableau: Example

- TM:
 - $Q = \{q_0, q_{\text{accept}}, q_{\text{reject}}\}$
 - $\Sigma = \{1\}$
 - $\Gamma = \{1, _\}$
 - $\delta(q_0, 1) = \{(q_0, _, R)\}$
 - $\delta(q_0, _) = \{(q_{\text{accept}}, L)\}$

Q: what does this machine compute?

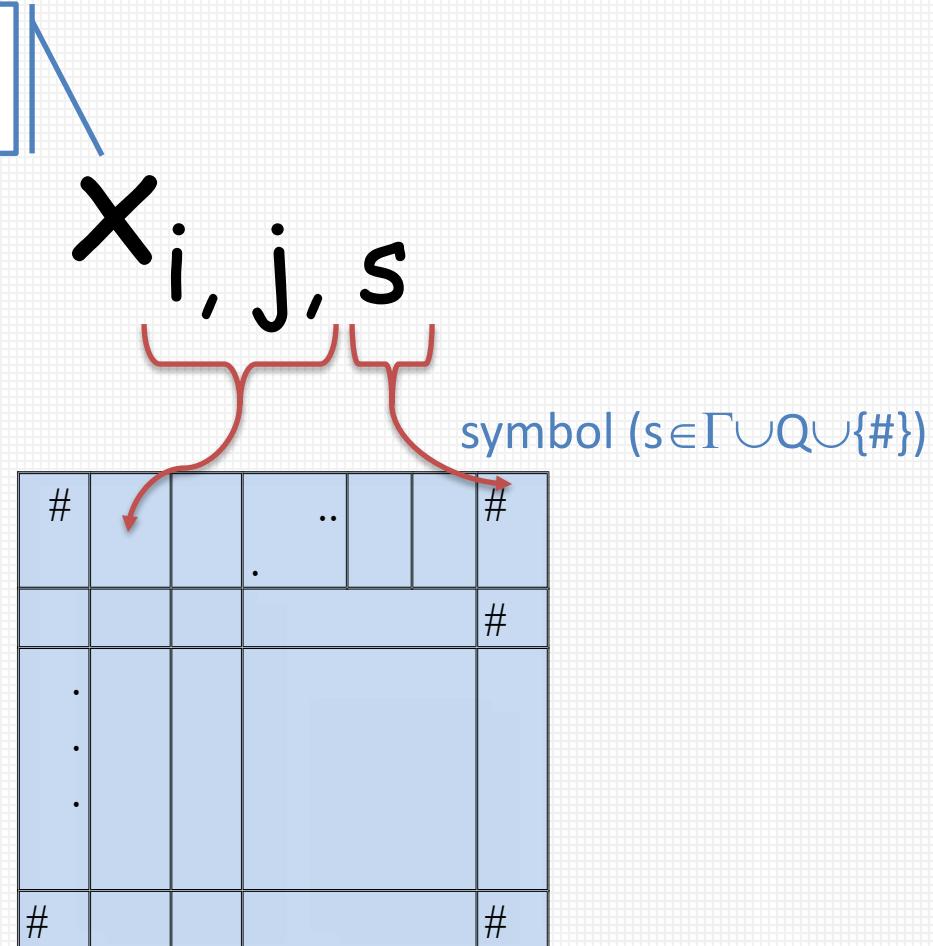
- tableau (input 11)

#	q_0	1	1	-	#
#	-	q_0	1	-	#
#	-	-	q_0	-	#
#	-	q_{acc}	-	-	#

The Variables of the Formula

stands for: “Is s the content of cell (i,j) ?”

position in the tableau ($1 \leq i,j \leq n^k$)



The Formula φ

$$\varphi_{M,w} = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}}$$

cell content
consistency

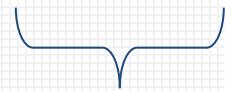
input consistency

machine accepts

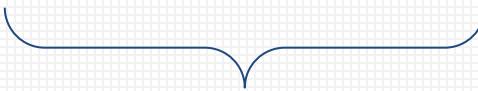
transition legal

Ensuring Unique Cell Content

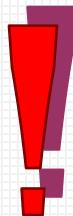
$$\varphi_{\text{cell}} = \bigwedge_{1 \leq i, j \leq n^k} \left[\left(\bigvee_{s \in C} X_{i,j,s} \right) \wedge \left(\bigwedge_{s \neq t \in C} (\overline{X_{i,j,s}} \vee \overline{X_{i,j,t}}) \right) \right]$$



The (i,j) cell
must contain
some symbol



It shouldn't contain
different symbols.



Note: the length of this formula is polynomial in n .

Ensuring Initial Configuration Corresponds to Input

Observe: we can explicitly state the desired configuration in the first step. Assuming the input string is $w_1 w_2 \dots w_n$,

$$\varphi_{\text{start}} = x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge \dots \wedge x_{1,n+3,_} \wedge \dots \wedge x_{1,n^k-1,_} \wedge x_{1,n^k,\#}$$

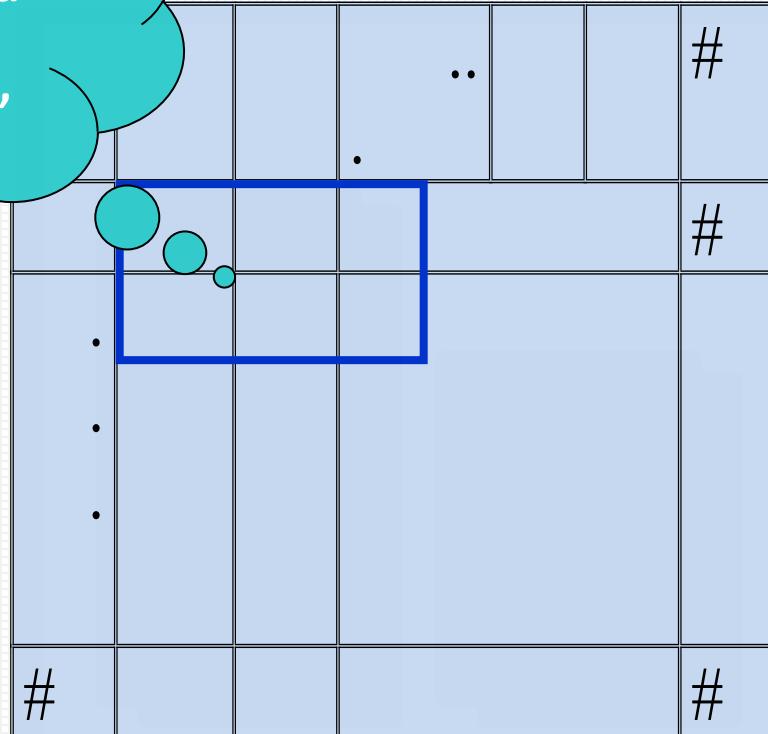
Ensuring the Computation Accepts

The accepting state is visited during the computation.

$$\Phi_{\text{accept}} = \bigvee_{1 \leq i, j \leq n^k} X_{i,j,q_{\text{accept}}}$$

Ensuring Every Transition is Legal

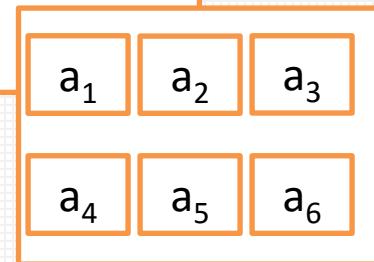
Local: only need
to examine
 2×3 “windows”



Ensuring Every Transition is Legal

$$\varphi_{\text{move}} = \bigwedge_{1 \leq i, j \leq n^k} \bigvee_{a_1, \dots, a_6} \left(x_{i-1, j, a_1} \wedge \dots \wedge x_{i+1, j+1, a_6} \right)$$

for any a_1, \dots, a_6 s.t.
this is a legal
window



Which Windows are Legal in the Following Example?

- TM:
 - $Q = \{q_0, q_{\text{accept}}, q_{\text{reject}}\}$
 - $\Sigma = \{1\}$
 - $\Gamma = \{1, _\}$
 - $\delta(q_0, 1) = \{(q_0, _, R)\}$
 - $\delta(q_0, _) = \{(q_{\text{accept}}, L)\}$

1	q ₀	1
q _{acc}	-	-

-	q ₀	1
-	-	q ₀

1	q ₀	1
1	-	q ₀

#	q ₀	1
#	-	q ₀

1	q ₀	1
1	1	q ₀

1	q ₀	-
q _{acc}	-	-

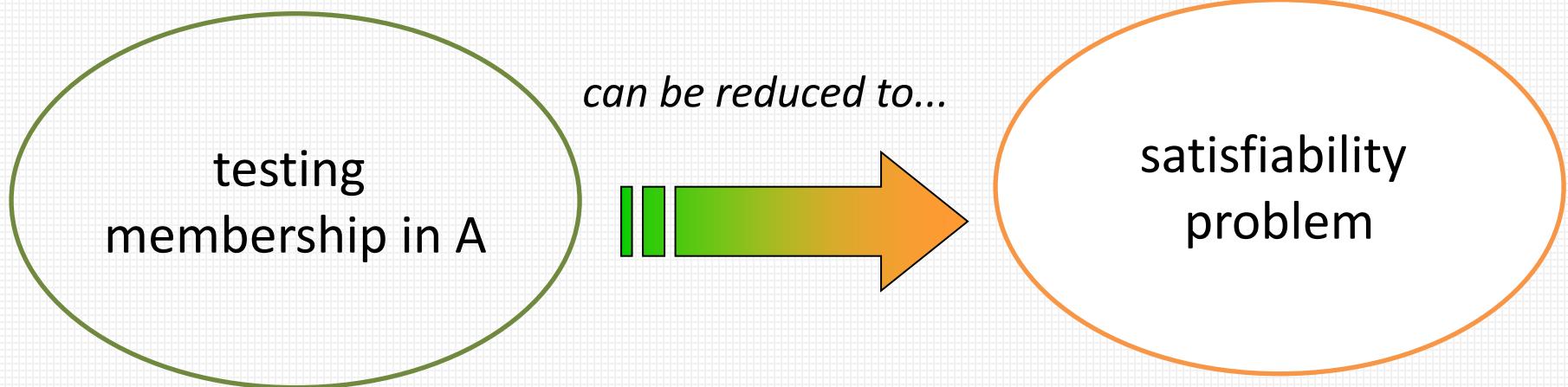
The Bottom Line

$$\varphi_{M,w} = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}}$$

φ , which is of size polynomial in n -
Check! - is satisfiable iff the TM accepts
the input string.

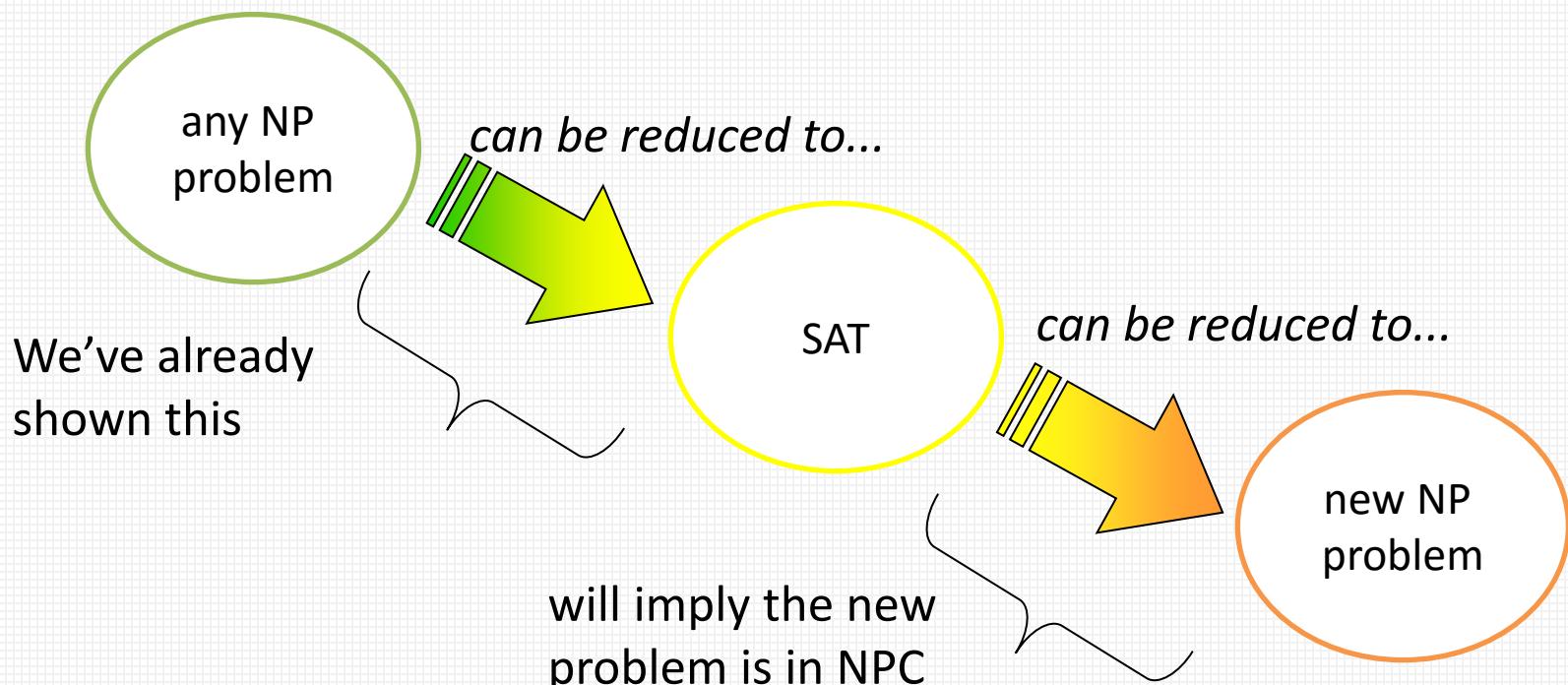
Conclusion: SAT is NP-Complete

For any language A in NP,



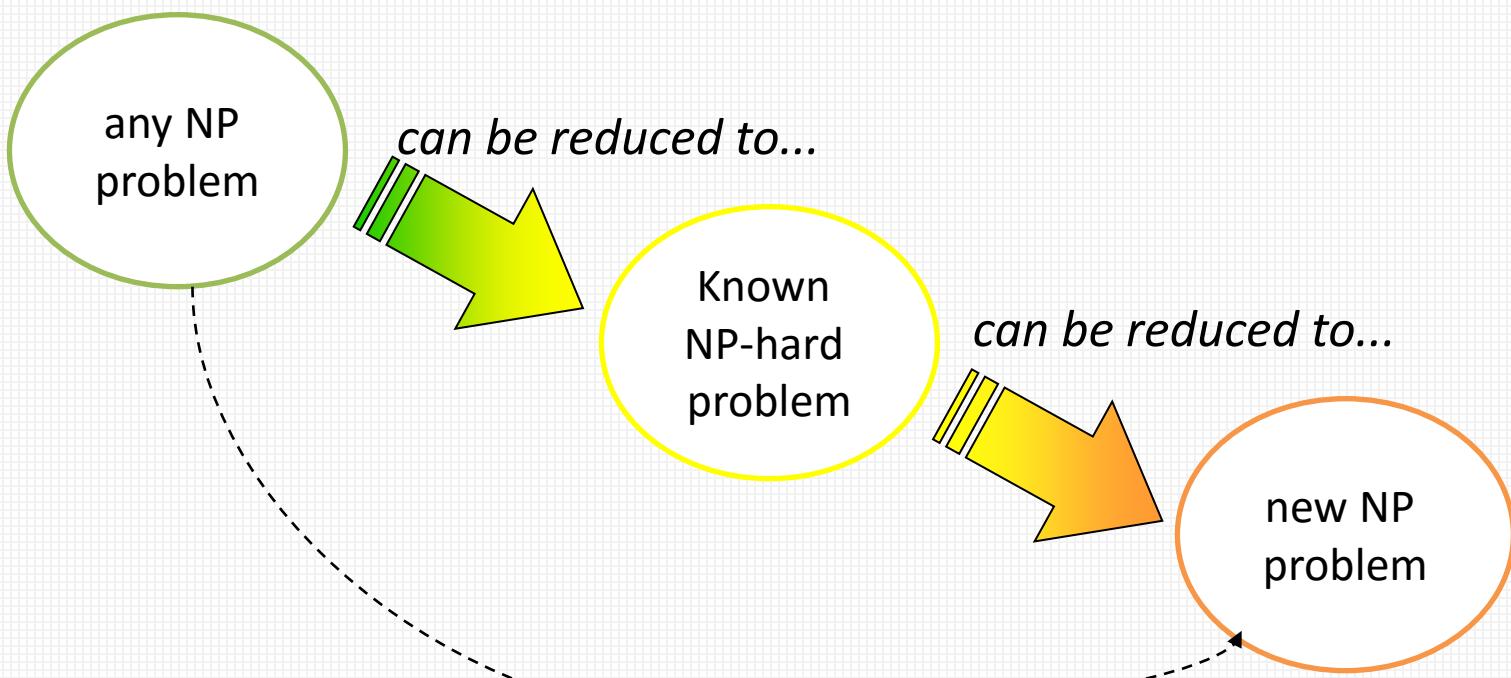
Looking Forward

From now on, in order to show some NP problem is NP-Complete, we merely need to reduce SAT to it.

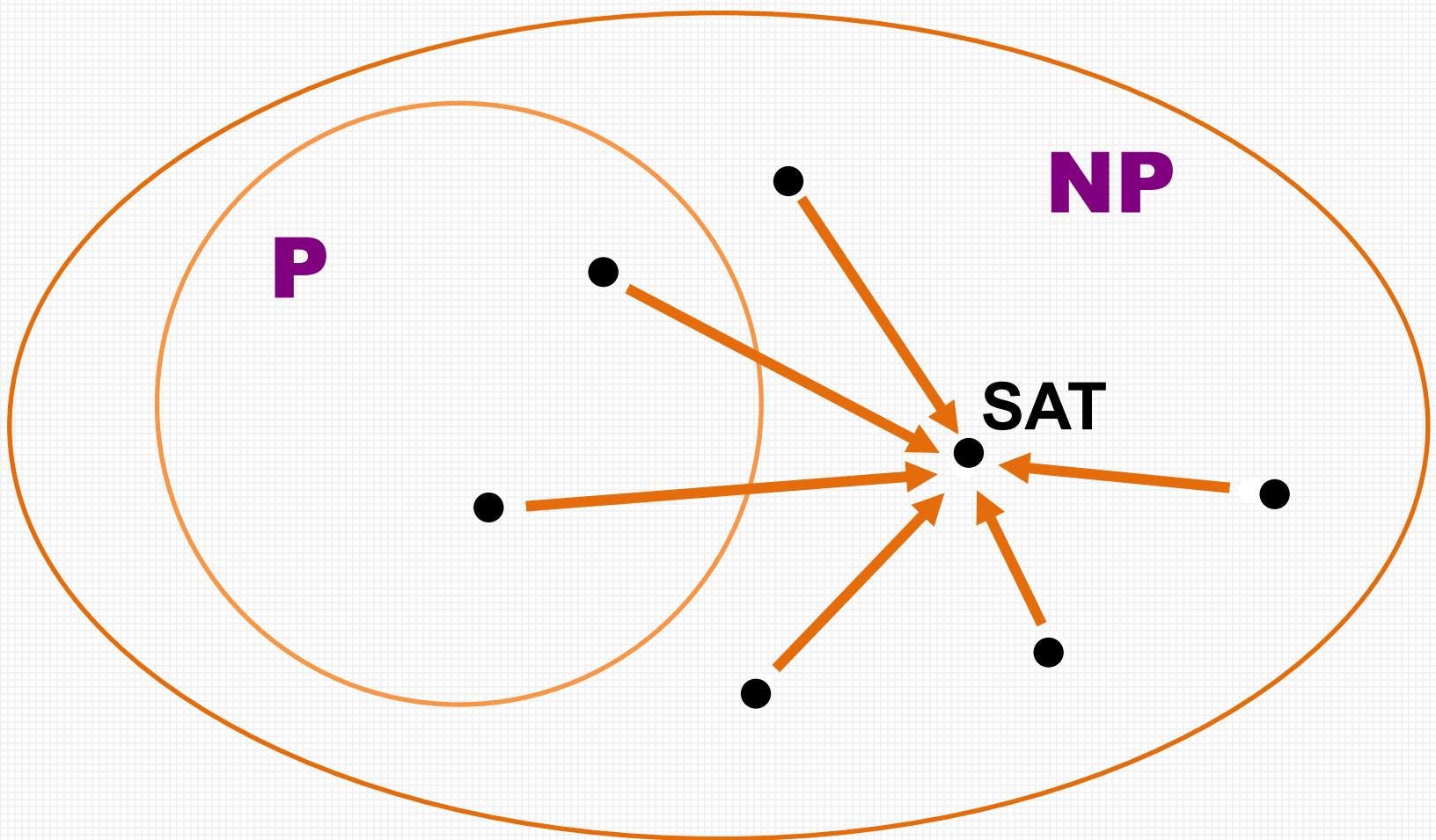


... and Beyond!

Moreover, every **NP-Complete** problem we discover, provides us with a new way for showing problems to be **NP-Complete**.



Any thing in $\text{NP} \leq_p \text{SAT}$



You can think of \rightarrow as “easier than”.
SAT is the hardest problem in NP.

3SAT = { ϕ | $\exists y$ such that y is a satisfying assignment to ϕ and ϕ is in 3cnf }

- Problem: Given a CNF where each clause has 3 variables, decide whether it is satisfiable or not.
- $(x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$

Idea:

We can convert (in polynomial time) a given SAT instance S into a 3SAT instance S' such that

- If S is satisfiable, then S' is satisfiable.
- If S' is satisfiable, then S is satisfiable.

Key Observation:

$x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5$ is satisfiable

if and only if

$(x_1 \vee x_2 \vee z_1) \wedge (\neg z_1 \vee x_3 \vee z_2) \wedge (\neg z_2 \vee x_4 \vee z_3) \wedge (\neg z_3 \vee x_4 \vee x_5)$
is satisfiable.

Polynomial Time Reduction

- Clause in SAT

x_1

$x_1 \vee \neg x_2$

$x_1 \vee x_2 \vee x_3$

$x_1 \vee x_2 \vee x_3 \vee x_4$

$x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5$

- Clauses in 3SAT

$x_1 \vee x_1 \vee x_1$

$x_1 \vee x_1 \vee \neg x_2$

$x_1 \vee x_2 \vee x_3$

$(x_1 \vee x_2 \vee z_1) \wedge (\neg z_1 \vee x_3 \vee x_4)$

$(x_1 \vee x_2 \vee z_1) \wedge (\neg z_1 \vee x_3 \vee z_2) \wedge$

$(\neg z_2 \vee x_4 \vee z_3) \wedge (\neg z_3 \vee x_4 \vee x_5)$

- CLIQUE = { $\langle G, k \rangle$ | G is undirected graph with a k -clique}.
- Idea:
 1. CLIQUE is NP (proved)
 2. $3\text{SAT} \leq_p \text{CLIQUE}$

Brute Force Algorithm:

Try out all $\{n \text{ choose } k\}$ possible locations for the k clique

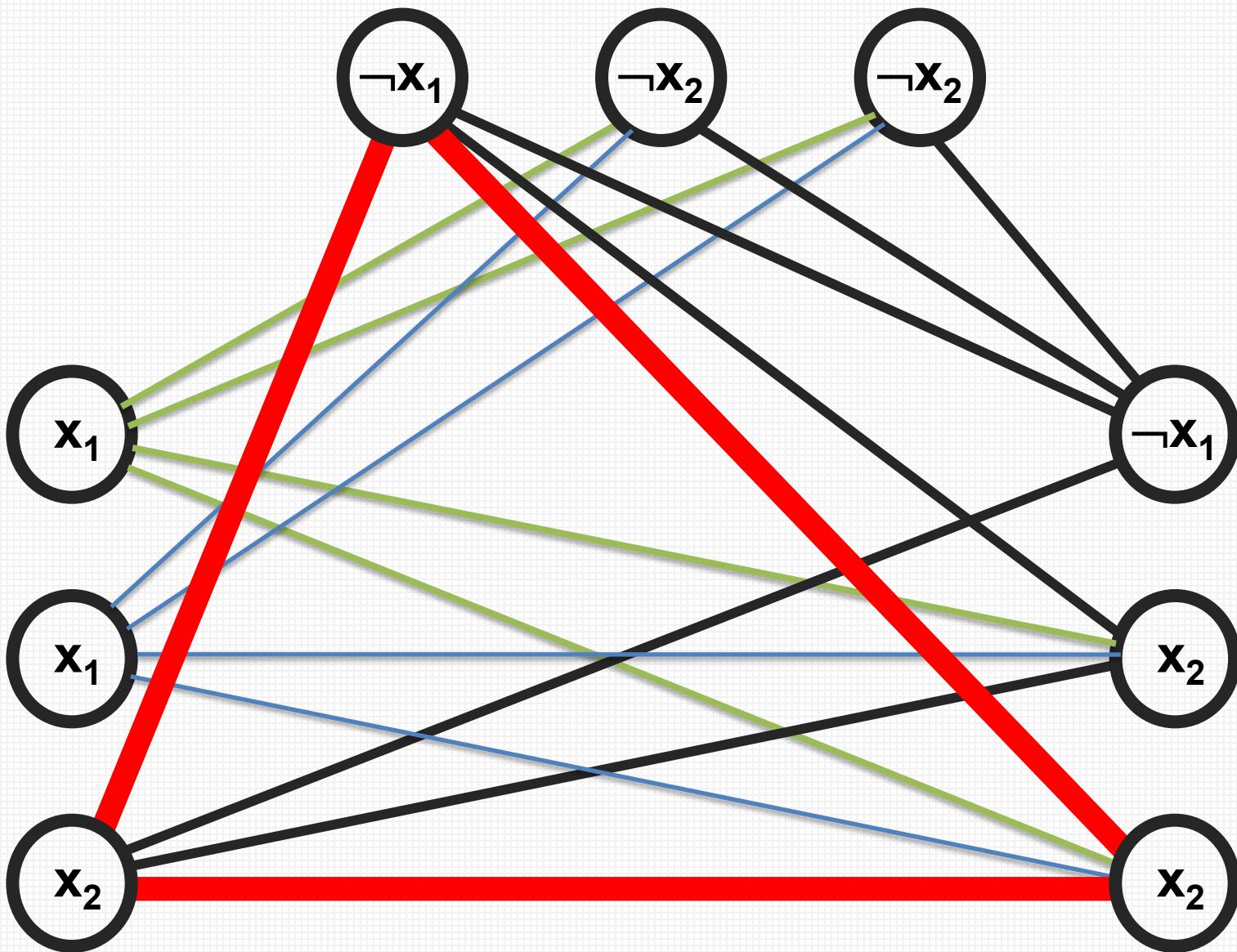
- We transform a 3-cnf formulation Φ into (G,k) such that

$$\phi \in 3SAT \Leftrightarrow (G, k) \in CLQUE$$

- If ϕ has m clauses, we create a graph with m clusters of 3 nodes each, and set $k=m$
 - Each cluster corresponds to a clause.
 - Each node in a cluster is labeled with a literal from the clause.
- We do not connect any nodes in the same cluster. We connect nodes in different clusters whenever they are not contradictory

$$(x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$$

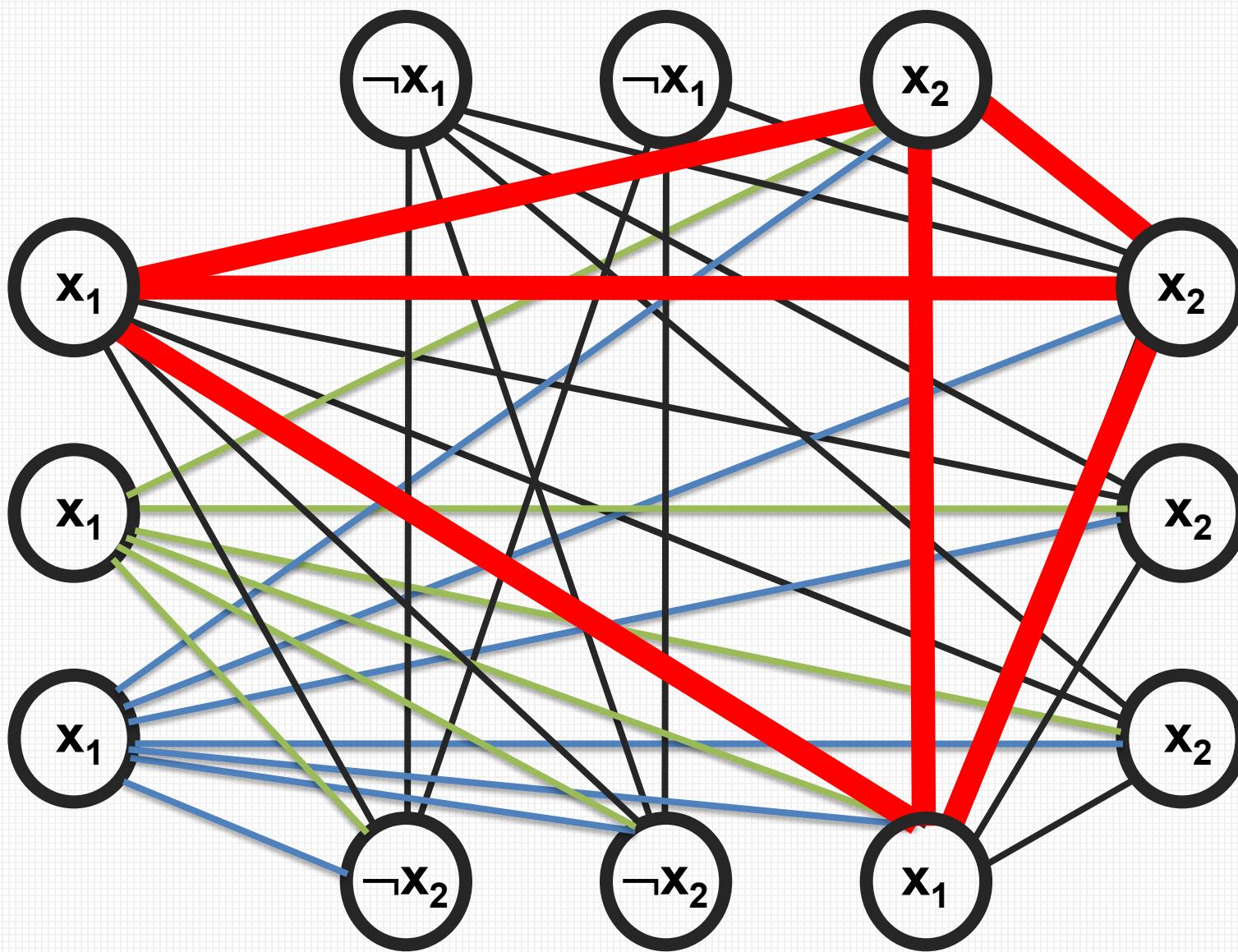
c
l
a
u
s
e



#nodes = 3(# clauses)

k = #clauses

$$\begin{aligned} & (\mathbf{x}_1 \vee \mathbf{x}_1 \vee \mathbf{x}_1) \wedge (\neg \mathbf{x}_1 \vee \neg \mathbf{x}_1 \vee \mathbf{x}_2) \wedge \\ & (\mathbf{x}_2 \vee \mathbf{x}_2 \vee \mathbf{x}_2) \wedge (\neg \mathbf{x}_2 \vee \neg \mathbf{x}_2 \vee \mathbf{x}_1) \end{aligned}$$



Other NP-Complete Problem

- Vertex Cover
- Max Cut

Theory of Computation

Lesson14-2

Approximation Algorithms



王轩
Wang Xuan

- Introduction
- MIN-VERTEX-COVER
- MAX-CUT

Optimization problem

- In certain problems called optimization problems we seek to find the best solution among a collection of possible solutions. When an optimization problem is **NP-hard**, as is the case with the first two of these types of problems, no polynomial time algorithm exists that finds the best solution unless **P=N_P**.
- In practice, we may not need the absolute best or **optimal solution** to a problem. A solution that is nearly optimal may be good enough and may be much easier to find. As its name implies, an **approximation algorithm** is designed to find such approximately optimal solutions.

- Question description:

We aim to produce one of the smallest vertex covers among all possible vertex covers in the input graph.

A = “On input $\langle G \rangle$, where G is an undirected graph:

1. Repeat the following until all edges in G touch a marked edge:
2. Find an edge in G untouched by any marked edge.
3. Mark that edge.
4. Output all nodes that are endpoints of marked edges.”

- Theorem 1:

A is a polynomial time algorithm that produces a vertex cover of G that is no more than twice as large as a smallest vertex cover.

Proof:

- 1) A obviously runs in polynomial time.
- 2) X touches all edges in G. (X be the set of nodes that it outputs)
- 3) $X \leq 2Y$

$$X = 2H$$

$$H \leq Y$$



$$X \leq 2Y$$

H be the set of edges that it marks.

- MIN-VERTEX-COVER is an example of a **minimization problem** because we aim to find the smallest among the collection of possible solutions. In maximization problem we seek the largest solution.
- An approximation algorithm for a minimization problem is **k-optimal** if it always finds a solution that is not more than k times optimal. For a maximization problem a k-optimal approximation algorithm always finds a solution that is at least $1/k$ times the size of the optimal.

- Question description:
- The MAX-CUT problem ask for a largest cut in the input graph G . (A cut in an approximation algorithm)
- The following algorithm approximates MAX-CUT within a factor of 2:

B = “On input $\langle G \rangle$, where G is an undirected graph with nodes V :

1. Let $S = \emptyset$ and $T = V$.
2. If moving a single node, either from S to T or from T to S , increases the size of the cut, make that move and repeat this stage.
3. If no such node exists, output the current cut and halt.”

- Theorem 10.2:

B is a polynomial time, 2-optimal approximation algorithm for MAX-CUT.

Proof:

1) B runs in polynomial time. Because the size of the cut \leq the total number of edges in G.

2) Output $\geq \frac{1}{2}$ Max-Cut 

Cut edges  \geq uncut edges output $\geq \frac{1}{2}$ total number of edges
in G Output $\geq \frac{1}{2}$ Max-Cut

Theory of Computation

Lesson14-3

Probabilistic Algorithms



王轩
Wang Xuan

Probabilistic Algorithms

- A **probabilistic algorithm** is an algorithm designed to use the outcome of a random process.
- Typically, such an algorithm would contain an instruction to “flip a coin” and the result of that coin flip would influence the algorithm's subsequent execution and output.
- Certain types of problems seem to be more easily solvable by probabilistic algorithms than by deterministic algorithms.

- We begin our formal discussion of probabilistic computation by defining a model of a **probabilistic Turing machine**.

DEFINITION 10.3

A **probabilistic Turing machine** M is a type of nondeterministic Turing machine in which each nondeterministic step is called a **coin-flip step** and has two legal next moves. We assign a probability to each branch b of M 's computation on input w as follows. Define the probability of branch b to be

$$\Pr[b] = 2^{-k},$$

where k is the number of coin-flip steps that occur on branch b . Define the probability that M accepts w to be

$$\Pr[M \text{ accepts } w] = \sum_{\substack{b \text{ is an} \\ \text{accepting branch}}} \Pr[b].$$

- In other words, the probability that M accepts w is the probability that we would reach an accepting configuration if we simulated M on w by flipping a coin to determine which move to follow at each coin-flip step. We let

$$\Pr[M \text{ rejects } w] = 1 - \Pr[M \text{ accepts } w].$$

- When a probabilistic Turing machine decides a language, it must accept all strings in the language and reject all strings out of the language as usual.
 - Now we allow the machine a small probability of error. For $0 \leq \epsilon < 1/2$, we say that M decides language A with error probability ϵ if

1. $w \in A$ implies $\Pr[M \text{ accepts } w] \geq 1 - \epsilon$, and
2. $w \notin A$ implies $\Pr[M \text{ rejects } w] \geq 1 - \epsilon$.

- We measure the time and space complexity of a **probabilistic Turing machine** in the same way we do for a **nondeterministic Turing machine**: by using **the worst case computation branch** on each input.

DEFINITION 10.4

BPP is the class of languages that are decided by probabilistic polynomial time Turing machines with an error probability of $\frac{1}{3}$.

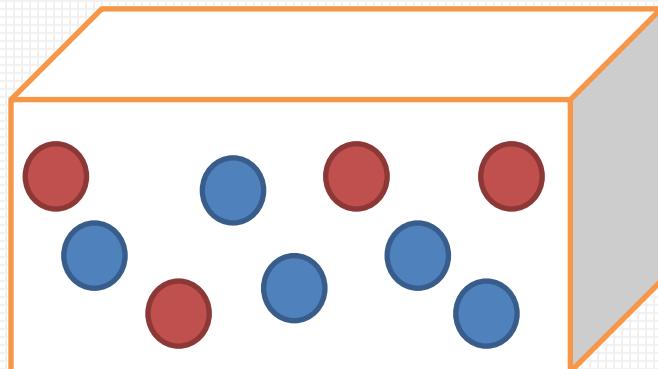
LEMMA 10.5

Let ϵ be a fixed constant strictly between 0 and $\frac{1}{2}$. Then for any polynomial $p(n)$, a probabilistic polynomial time Turing machine M_1 that operates with error probability ϵ has an equivalent probabilistic polynomial time Turing machine M_2 that operates with an error probability of $2^{-p(n)}$.

- PROOF IDEA

M2 simulates M1 by running it a polynomial number of times and taking the majority vote of the outcomes. The probability of error decreases exponentially with the number of runs of M1 made.

Consider the case where $\varepsilon = 1/3$.



It corresponds to a box that contains many red and blue balls. We know that $2/3$ of the balls are of one color and that the remaining $1/3$ are of the other color, but we don't know which color is predominant.

We can test for that color by sampling several—say, 100—balls at random to determine which color comes up most frequently.
the predominant color in the box  the most frequent one in the sample

- PROOF IDEA

balls  branches of M_1 's computation

-   accepting
-   rejecting

M_2 samples the color by running M_1 .

A calculation shows that M_2 errs with exponentially small probability if it runs M_1 a polynomial number of times and outputs the result that comes up most often.

PROOF Given TM M_1 deciding a language with an error probability of $\epsilon < \frac{1}{2}$ and a polynomial $p(n)$, we construct a TM M_2 that decides the same language with an error probability of $2^{-p(n)}$.

M_2 = “On input x :

1. Calculate k (see analysis below).
2. Run $2k$ independent simulations of M_1 on input x .
3. If most runs of M_1 accept, then *accept*; otherwise, *reject*.”