Bangladesh Govt. & UGC Approved

UNIVERSITY OF GLOBAL VILLAGE (UGV), BARISHAL.

THE FIRST SKILL BASED HI-TECH UNIVERSITY IN BANGLADESH

THEORY OF COMPUTATION

MD. MAHADI HASAN SHAON

Theory of Computation

Course Code:	CSE-0613-2101	Credits:	03
		CIE Marks:	90
Exam Hours:	03	SEE Marks:	60

Course Learning Outcome (CLOs): After Completing this course successfully, the student will be able to…

CLO 1	Describe the concepts and formalisms used in formal language theory and automata theory, including the notions of regular and context-free languages, finite automata, pushdown automata, and Turing machines.
CLO 2	Understand the fundamental results and techniques in automata theory and complexity theory, such as the pumping lemma, decidability, NP-completeness, polynomial-time reductions, and approximation algorithms.
CLO 3	Create and design automata, regular expressions, context-free grammars, and Turing machines to solve problems and recognize languages.
CLO 4	Apply the concepts and techniques of automata theory to solve problems in various areas, such as computer science, linguistics, and engineering.
CLO 5	Identify the limitations of automata and formal language theory, and appreciate the broader implications of these limitations on the theory and practice of computer science.

SUMMARY OF COURSE CONTENT

Sl. No.	COURSE CONTENT	HRs	CLOs
1.	Introduction: Formal language theory, Formal proof, Inductive proofs and Central concepts of automata theory.	6	CLO1 CLO2
1.	Finite Automata: Deterministic finite automata, Nondeterministic finite automata, Finite automata with ε-transitions, Equivalence and conversion of deterministic and nondeterministic finite automata.	6	CLO3
1.	Regular Expressions and Languages: Regular expressions, Algebraic laws for regular expressions, Regular languages, Pumping lemma.	6	CLO3
1.	Context Free Grammar and Languages: Context free grammars, Parsing (or derivation) and parse trees, Ambiguity in grammars and languages, Normal forms for context-free grammars, Pumping lemma for CFL's.	6	CLO3 CLO4
1.	Push Down Automata: Push down automata, Acceptance by empty store and final state, Equivalence between pushdown automata and context-free grammars, Deterministic push down automata.	6	CLO4
1.	Turing Machines: Turing machines, The church-Turing machine, Techniques for Turing machine construction, Configurations, Computing with Turing machines, Restricted Turing machines, Turing machines and computers, Combining Turing machines.	6	CLO4 CLO5
1.	Undecidability and Complexity Theory: Recursively enumerable language, the undecidability of the halting problem, Undecidable problems about Turing machines, Post's correspondence problem, Complexity theory including the classes P, NP, examples of problems in these classes, NP completeness, Polynomial time reducibility, The Cook-Levin theorem, examples of NP complete problems, approximation algorithms, and probabilistic algorithms.	6	CLO5

Recommended Books:

1. Introduction to Automata Theory, Languages, and Computation by John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman
2. Theory of Computation by Michael Sipser

ASSESSMENT PATTERN

CIE- Continuous Internal Evaluation (90 Marks)

Bloom's Category Marks (out of 90)	Tests (45)	Assignments (15)	Quizzes (15)	Attendance (15)
Remember	5	03		
Understand	5	04	05	
Apply	15	05	05	
Analyze	10			
Evaluate	5	03	05	
Create	5			

SEE- Semester End Examination (60 Marks)

Bloom's Category	Test
Remember	7
Understand	7
Apply	20
Analyze	15
Evaluate	6
Create	5

Week No.	Topics	Teaching Learning Strategy(s)	Assessment Strategy(s)	Alignment to CLO
1	Introduction to Theory of Computation	Lecture, multimedia, group discussion	Feedback, Q&A, assessment of LOs	CLO1
2	Finite Automata	Explain DFA and NFA, conversion, and applications through interactive sessions and exercises	Feedback, Q&A, quizzes	CLO2
3	Equivalence of NFA & DFA	Explore equivalence proofs through lectures and problem-solving workshops	Group activities, quizzes, assignments	CLO2
4	Regular Languages and Expressions	Lecture, multimedia, interactive sessions	Feedback, Q&A, group discussions	CLO3
5	Context-Free Grammars and Languages	Lecture, multimedia, and practical examples on CFG creation and parsing techniques	Midterm Quiz, hands-on practice	CLO4
6	Pushdown Automata	Lecture and problem-solving sessions focusing on PDA and CFG relationships	Feedback, Q&A, assessment of LOs	CLO5
7	Turing Machines	Implement Turing machines through hands-on practice and explore their significance	Group assignments, feedback, quizzes	CLO6
8	Revision and Recap	Review through group discussions and interactive sessions to consolidate knowledge	Feedback, problem-solving assessments	CLO1-8
9	Practical Applications of Theory	Hands-on exercises applying concepts to real-world computational problems	Case studies, group work	CLO9

Week No.	Topics	Teaching Learning Strategy(s)	Assessment Strategy(s)	Alignment to CLO
10	Mid Examination	Summative evaluation covering the entire course content	Written exam, case studies	CLO1-9
11	Decidability and Computability	Interactive discussions on undecidability problems and hands-on exploration of key topics	Feedback, Q&A, Midterm Exam	CLO7
12	Complexity Theory	Lecture, problem-solving sessions on P, NP, NP-complete, and NP-hard problems	Feedback, Q&A, quizzes	CLO8
13	Equivalence of NFA & DFA (Revisited)	Deep understanding with further problem-solving on equivalence	Practical exercises, quizzes	CLO2
14	Revision and Recap	Consolidate topics with a focus on problem-solving and exam preparation	Final Term quizzes, feedback	CLO1-8
15	Feedback and Future Directions	Reflective discussions on the course and exploration of research opportunities	Participation feedback	CLO10
16	Final Documentation/Presentation	Showcase application of Theory of Computation concepts in a practical or research project	Presentations, project assessment	CLO1-10
17	Final Examination	Summative evaluation covering the entire course content	Written exam, case studies	CLO1-10

WEEK 1

THEORY OF COMPUTATION

- The ***Theory of Computation*** is the branch of computer science that deals with how efficiently problems can be solved on a model of computation, using an algorithm.
- The field is divided into three major branches:
- Automata theory and language
- Computability theory
- Complexity theory

COMPLEXITY THEORY

- The main question asked in this area is “What makes some problems computationally *hard* and other problems *easy*?”
- A problem is called “easy”, if it is efficiently solvable.

Examples of “easy” problems are (i) sorting a sequence of, say, 1,000,000 numbers, (ii) searching for a name in a telephone directory.

- A problem is called “hard”, if it cannot be solved efficiently, or if we don’t know whether it can be solved efficiently.

Examples of “hard” problems are (i) factoring a 300-digit integer into its prime factors.

Central Question in *Complexity Theory*: Classify problems according to their degree of “difficulty”. Give a proof that problems that seem to be “hard” are really “hard”.

COMPUTABILITY THEORY

- Computability theory In the 1930's, G¨odel, Turing, and Church discovered that some of the fundamental mathematical problems cannot be solved by a "computer".
- To attack such a problem, we need formal definitions of the notions of *computer*, *algorithm*, and *computation*.
- The *theoretical models* that were proposed in order to understand *solvable* and *unsolvable* problems led to the development of real computers.

Central Question in *Computability Theory*: Classify problems as being solvable or unsolvable.

AUTOMATA THEORY

Automata Theory deals with definitions and properties of different types of "*computation models*". Examples of such models :

Finite Automata :These are used in text processing, compilers, and hardware design.

Context-Free Grammars: These are used to define programming languages and in Artificial Intelligence.

Turing Machines: These form a simple abstract model of a "real" computer, such as your PC at home.

Central Question in *Automata Theory*: Do these models have the same power, or can one model solve more problems than the other?.

THEORY OF COMPUTATION

Purpose and motivation :

- What are the mathematical properties of computer hardware and software ?
- What is a *computation* and what is an *algorithm*? Can we give mathematical definitions of these notions?
- What are the *limitations* of computers? Can "everything" be computed?

Purpose of the TOC: Develop formal mathematical models of computation that reflect real-world computers.

WEEK 2

FINITE AUTOMATA

- We will use several different models, depending on the features we want to focus on. Begin with the simplest model, called the finite automaton.
- Good models for computer with an extremely limited amount of memory. For example, various household appliances such as dishwashers and electronic thermostats, as well as parts of digital watches and calculators.
- The design of such devices requires keeping the methodology and terminology of finite automata in mind.
- Next we will analyze an example to get an idea.

FINITE STATE MACHINE

- The finite state machine represents a mathematical model of a system with certain input.
- The model finally gives certain output.
- The input is processed by various states, these states are called as intermediate states.
- The finite state system is very good design tool for the programs such as TEXT EDITORS and LEXICAL ANALYZERS.

DEFINITION OF FINITE AUTOMATA

A finite automata is a collection of 5-tuple(Q,Σ,δ,q_0,F) Where,

Q is finite set of states, which is non empty.

Σ is input alphabet, indicates input set.

δ is transition function or mapping function. We can determine the next state using this function.

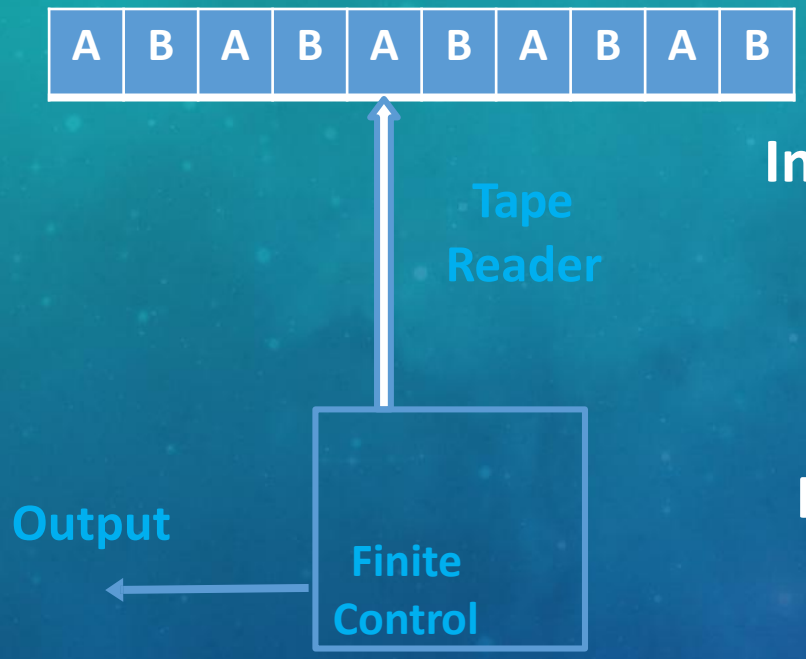
q_0 is an initial state and is in

F is set of final states.

δ = Delta

FINITE AUTOMATA MODEL

- The finite automata can be represented as follows:



Input Tape: It is a linear tape having some number of cells. Each input symbol is placed in each cell.

Finite Control: The finite control decides the next state on receiving particular input from input tape.

EXAMPLE

Figure: Top view of an automatic door

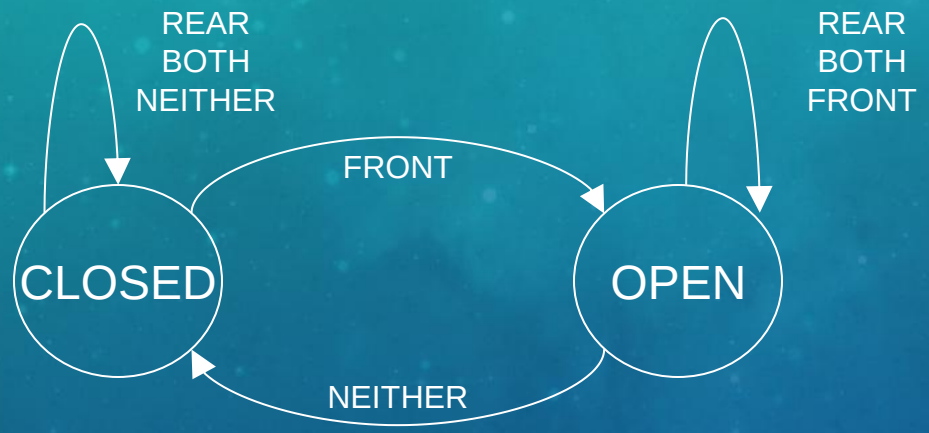


Figure: State diagram for Automatic door controller

	Input Signal			
State	NEITHER	FRONT	REAR	BOTH
CLOSED	CLOSED	OPEN	CLOSED	CLOSED
OPEN	CLOSED	OPEN	OPEN	OPEN

Figure: State Transition table for automatic door controller

- Automatic doors swing open when sensing that a person is approaching.
- An automatic door has a pad in front to detect the presence of a person about to walk through the doorway.
- Another pad is located to the rear of the doorway so that –
  - The controller can hold the door long enough for the person to pass all the way through.
  - The door does not strike someone standing behind it as it opens.

TERMINOLOGY

Figure: A finite automaton called M_1 that has three states.

- The above figure is called *state diagram* of M_1.
- It has three *states*, labeled q_1, q_2, and q_3.
- The *start state* is q_1, indicated by the arrow pointing at it from no where.
- The *accept state*, q_2, is the one with a double circle.
- The arrow going from one state from another are called *transitions*.
- The symbol(s) along the transition is called *label*.

M_1 works as follows –

The automaton receives the symbols from the input string one by one from left to right.

After reading each symbol, M_1 moves from one state to another along the transition that has the symbol as its label.

When it reads the last symbol, M_1 produces the output.

The output is ACCEPT if M_1 is now in an accept state and REJECT if it is not.

SIMULATION

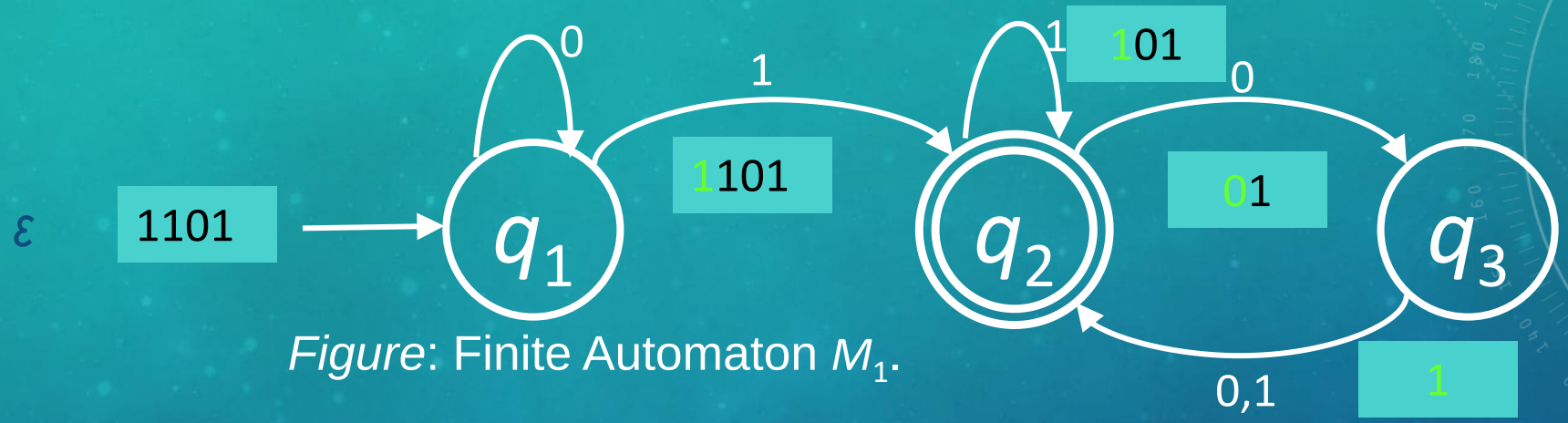


Figure: Finite Automaton M_1.

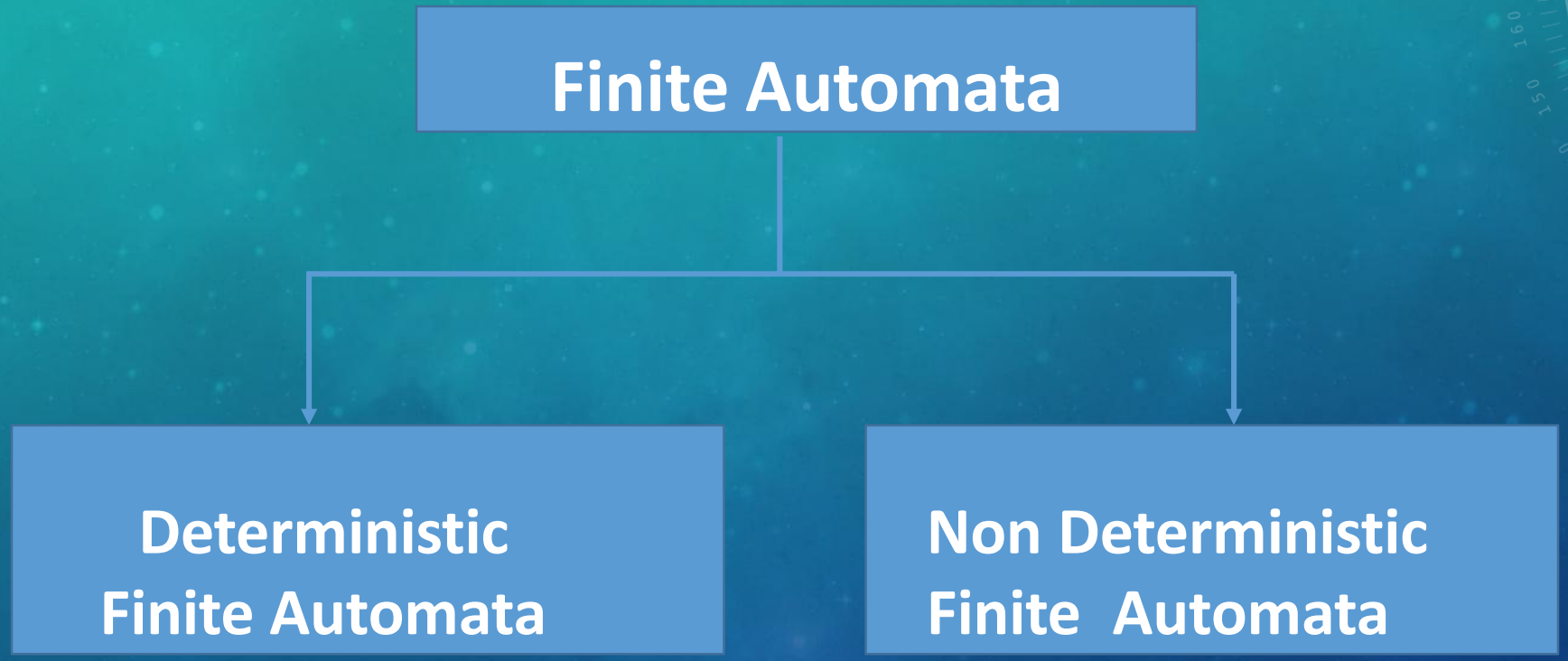
After feeding the input string 1101 to the above machine, the processing proceeds as follows –

Start in state q_1;

- Read 1, follow transition from q_1 to q_2;
- Read 1, follow transition from q_2 to q_2;
- Read 0, follow transition from q_2 to q_3;
- Read 1, follow transition from q_3 to q_2;

Accept, as the machine M_1 is in an accept state q_2 at the end of the input string.

TYPES OF AUTOMATA



TYPES OF AUTOMATA

- **Deterministic Finite Automata:** The Finite Automata is called Deterministic Finite Automata if there is only one path for a specific input from current state to next state.

It can be represented as follows:

- A machine M = (Q,Σ,δ,q_0,F) Where ,

  Q is finite set of states, which is non empty.

  Σ is input alphabet, indicates input set.

  δ is transition function or mapping function. We can determine the next state using this function.

  q_0 is an initial state and is in Q

  F is set of final states.

Where δ:**Q X** Σ **-> Q**

TYPES OF AUTOMATA

- **Non Deterministic Finite Automata:** The Finite Automata is called Non Deterministic Finite Automata if there are more than one path for a specific input from current state to next state.

It can be represented as follows:

- A machine M = (Q,Σ,δ,q_0,F) Where ,

  Q is finite set of states, which is non empty.

  Σ is input alphabet, indicates input set.

  δ is transition function or mapping function. We can determine the next state using this function.

  q_0 is an initial state and is in Q

  F is set of final states.

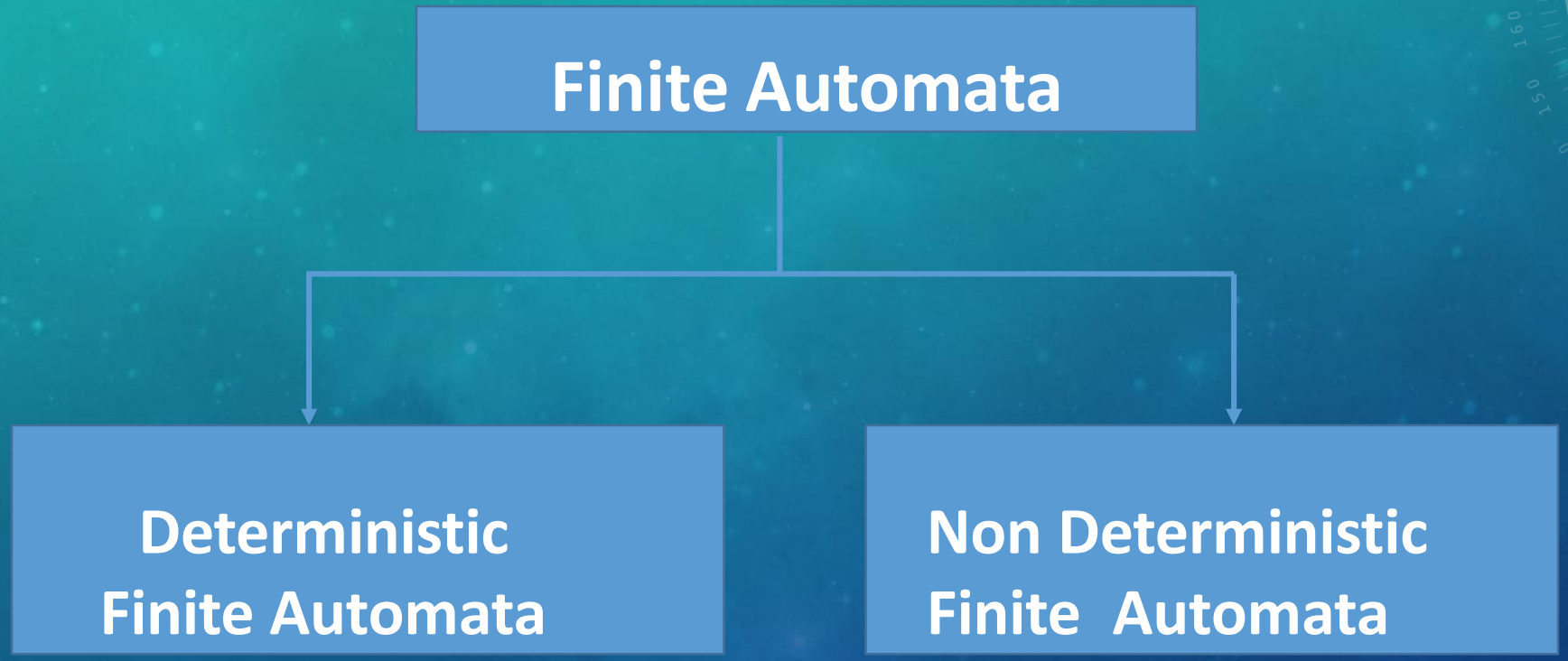
Where δ:**Q X** Σ **-> 2^Q**

DIFFERENCE BETWEEN DFA & NFA

Deterministic Finite Automata	Non Deterministic Finite Automata
For Every symbol of the alphabet, there is only one state transition in DFA.	We do not need to specify how does the NFA react according to some symbol.
DFA cannot use Empty String transition.	NFA can use Empty String transition.
DFA can be understood as one machine.	NFA can be understood as multiple little machines computing at the same time.
DFA will reject the string if it end at other than accepting state.	If all of the branches of NFA dies or rejects the string, we can say that NFA reject the string.
Backtracking is allowed in DFA.	Backtracking is not always allowed in NFA.
DFA is more difficult to construct.	NFA is easier to construct.

WEEK 3

TYPES OF AUTOMATA



TYPES OF AUTOMATA

- **Deterministic Finite Automata:** The Finite Automata is called Deterministic Finite Automata if there is only one path for a specific input from current state to next state.

It can be represented as follows:

- A machine M = (Q,Σ,δ,q_0,F) Where ,

  Q is finite set of states, which is non empty.

  Σ is input alphabet, indicates input set.

  δ is transition function or mapping function. We can determine the next state using this function.

  q_0 is an initial state and is in Q

  F is set of final states.

Where δ:**Q X** Σ **-> Q**

TYPES OF AUTOMATA

- **Non Deterministic Finite Automata:** The Finite Automata is called Non Deterministic Finite Automata if there are more than one path for a specific input from current state to next state.

It can be represented as follows:

- A machine M = (Q,Σ,δ,q_0,F) Where ,

  Q is finite set of states, which is non empty.

  Σ is input alphabet, indicates input set.

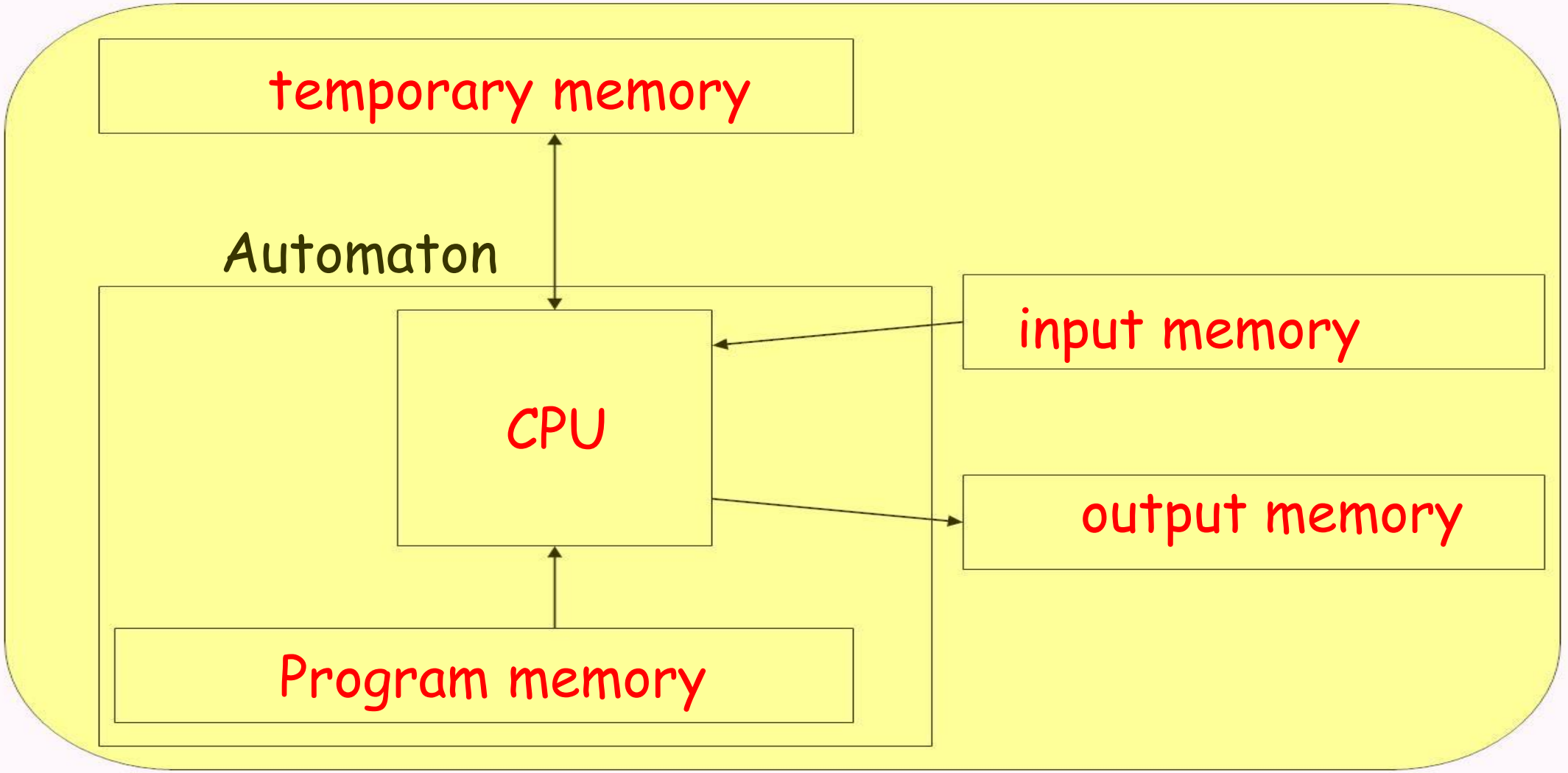
  δ is transition function or mapping function. We can determine the next state using this function.

  q_0 is an initial state and is in Q

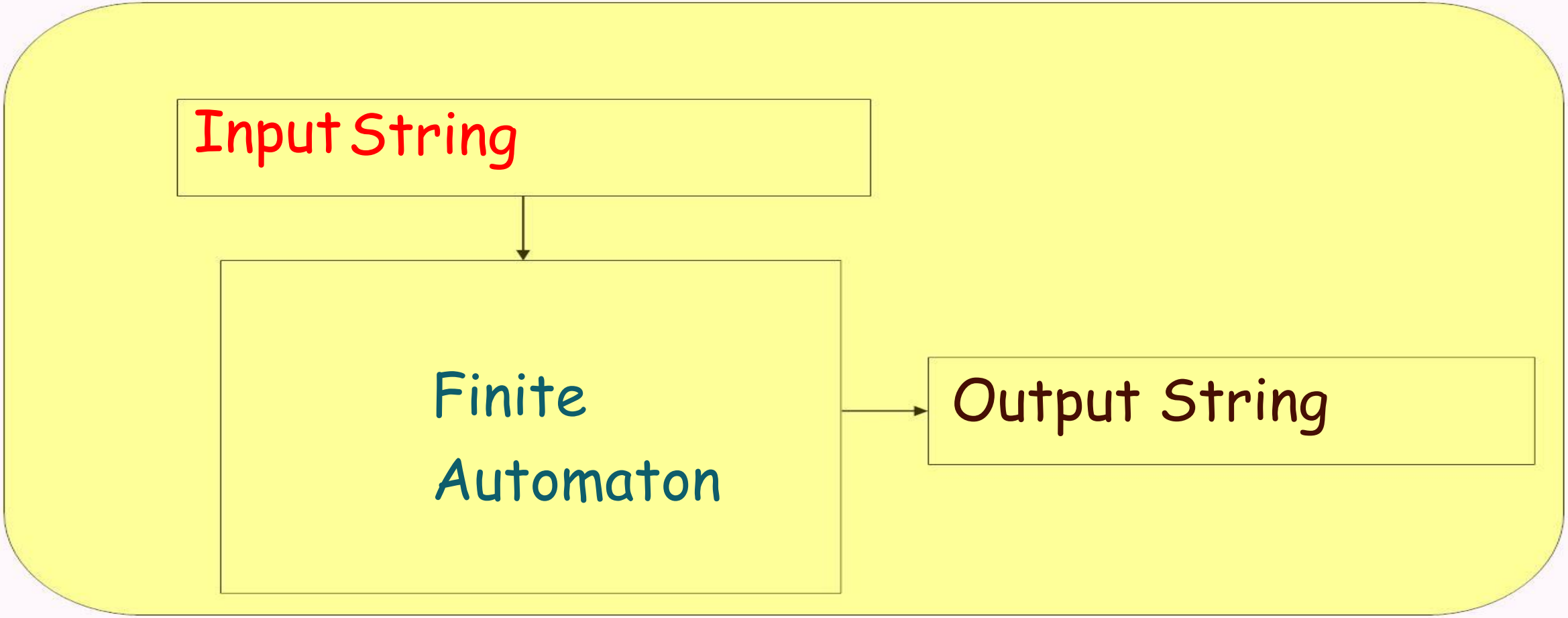
  F is set of final states.

Where δ:**Q X** Σ **-> 2^Q**

Automaton


temporary memory
Automaton
CPU
input memory
output memory
Program memory

Finite Automaton



DFA

- DFA: Deterministic Finite Automaton.
- Every step of a computation follows in a unique way from the preceding step.
- When the machine is in a given state and reads the next input symbol, we know the next state will be – it is determined.
- We call this deterministic computation.

FORMAL DEFINITION OF A DFA

- A DFA is a five-tuple:

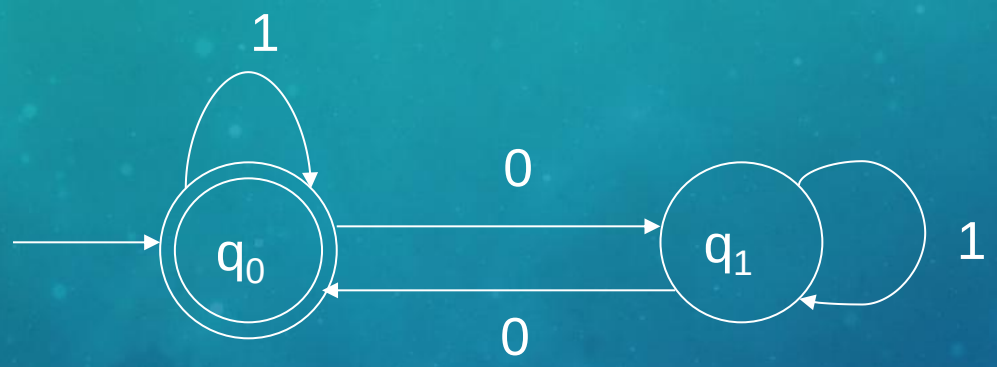
$M = (Q, \Sigma, \delta, q_0, F)$

Q A finite set of states
Σ A finite input alphabet
q_0 The initial/starting state, q_0 is in Q
F A set of final/accepting states, which is a subset of Q
δ A transition function, which is a total function from Q x Σ to Q

$\delta: (Q \times \Sigma) \rightarrow Q$ δ is defined for any q in Q and s in Σ, and
$\delta(q,s) = q'$ is equal to some state q' in Q, could be q'=q

Intuitively, δ(q,s) is the state entered by M after reading symbol s while in state q.

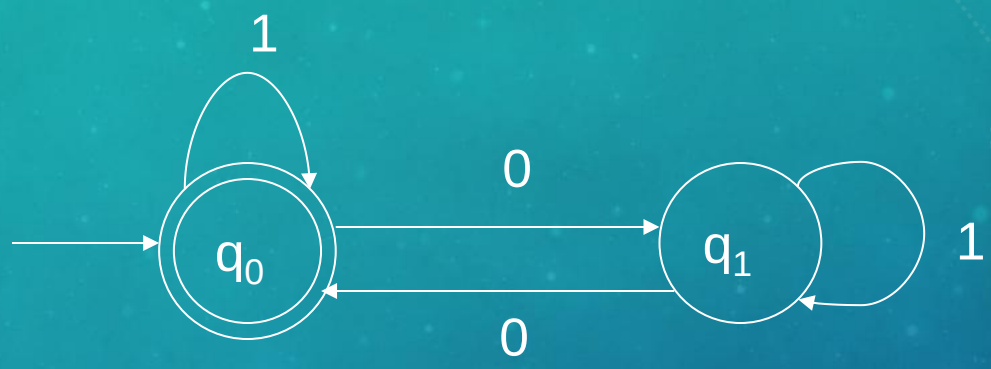
- The finite control can be described by a transition diagram or table:



1		0	0	1	1	
q_0	q_0	q_1	q_0	q_0	q_0	accpeted

- One state is final/accepting, all others are rejecting.
- The above DFA accepts those strings that contain an even number of 0's, including the *null* string, over *Sigma* = {0,1}

  L = {all strings with zero or more 0's}
- Note, the DFA must reject all other strings



$Q = \{q_0, q_1\}$

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

Start state is q_0

$F = \{q_0\}$

δ:

	0	1
q_0	q_1	q_0
q_1	q_0	q_1

NONDETERMINISTIC FINITE STATE AUTOMATA (NFA)

- An NFA is a five-tuple:

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

  Q A finite set of states

  Σ A finite input alphabet

  q_0 The initial/starting state, q_0 is in Q

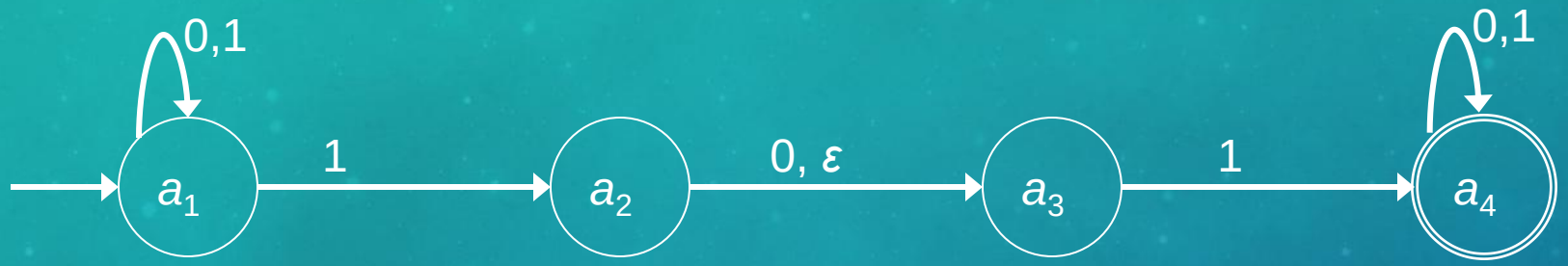
  F A set of final/accepting states, which is a subset of Q

  δ A transition function, which is a total function from Q x Σ to 2^Q

  δ: (Q x Σ) –> **2^Q** :2^Q is the power set of Q, the set of *all subsets* of Q δ(q,s) :The set of all states p such that there is a transition labeled s from q to p

  δ(q,s) is a function from Q x S to 2^Q (but not only to Q)

PROPERTIES OF NFA

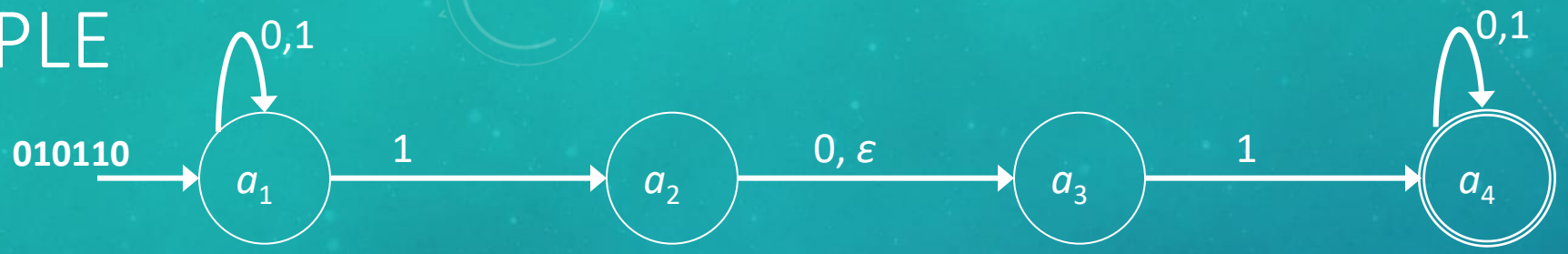


- We already know DFA, so it would be sufficient to look into the differences of properties between the two.
- In NFA a state may have –
  - Zero or more exiting arrows for each alphabet symbol.
  - Zero or more exiting arrows with the label ε.
- So we can see that, not all steps of a computation follows in a unique way from the preceding step. There can be multiple choices to move from one state to another with a symbol. That's the reason it's computation is called nondeterministic.

RUNNING AN NFA

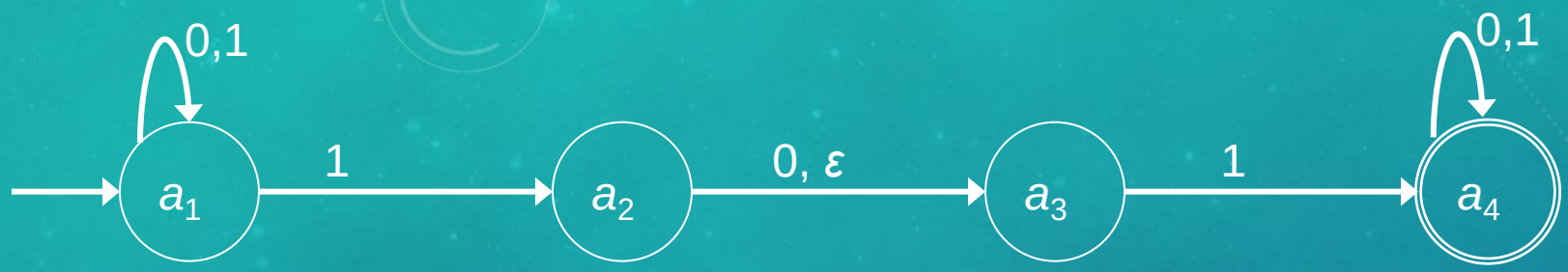
- If we encounter a state with multiple ways to proceed –
  - The machine splits into multiple copies of itself and follows all the possibilities in parallel.
  - Each copy of the machine takes one of the possible ways to proceed and continues as before.
  - If there are subsequent choices, the machine splits again.
- If a state with an ε symbol on an exiting arrow is encountered without reading any input, the machine splits into multiple copies,
  - one following each of the exiting ε-labeled arrows and
  - one staying in the current state.
- If the next input symbol doesn't appear on any of the arrows exiting the state occupied by a copy of the machine, that copy of the machine dies, along with the branch of the computation associated with it.
- If any one of these copies of the machine is in an accept state at the end of the input, the NFA accepts the input strings.
- So, nondeterminism may be viewed as a kind of parallel computation wherein several processes can be running concurrently.
- If at least one of these processes accepts then the entire computation accepts.

EXAMPLE



WEEK 4

EXAMPLE

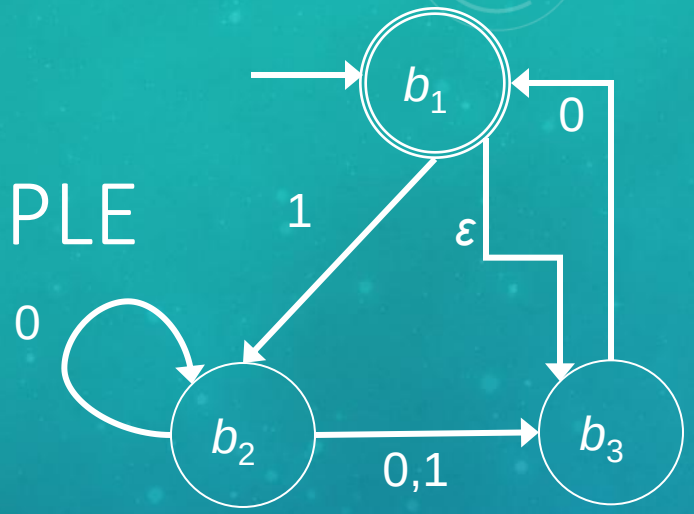


- Let, the above NFA $N_1 = (Q_1, \Sigma, \delta_1, a_1, F_1)$.
  - $Q_1 = \{a_1, a_2, a_3, a_4\}$.
  - $\Sigma = \{0, 1\}$.
  - δ_1 is given as –

	0	1	ε
a_1	$\{a_1\}$	$\{a_1, a_2\}$	ϕ
a_2	$\{a_3\}$	ϕ	$\{a_3\}$
a_3	ϕ	$\{a_4\}$	ϕ
a_4	$\{a_4\}$	$\{a_4\}$	ϕ

  - a_1 is the start state.
  - $F_1 = \{a_4\}$.

EXAMPLE



- Let, the above NFA $N_2 = (Q_2, \Sigma, \delta_2, b_1, F_2)$.
  - $Q_2 = \{b_1, b_2, b_3\}$.
  - $\Sigma = \{0, 1\}$.
  - δ_2 is given as –

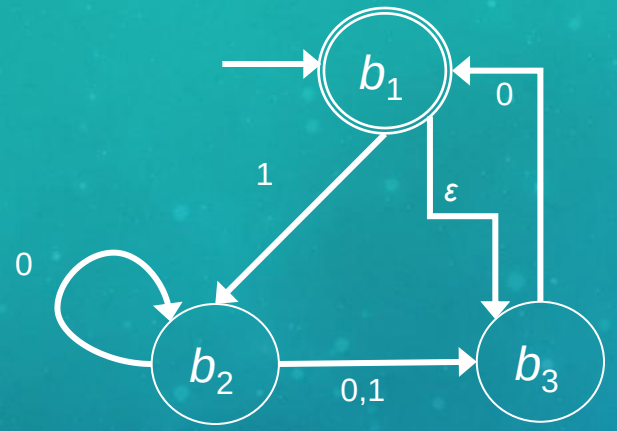
	0	1	ε
b_1	ϕ	$\{b_2\}$	$\{b_3\}$
b_2	$\{b_2, b_3\}$	$\{b_3\}$	ϕ
b_3	$\{b_1\}$	ϕ	ϕ

  - b_1 is the start state.
  - $F_2 = \{b_1\}$.

Equivalence between NFA & DFA

- Every NFA has an equivalent DFA.
- Let $N = (Q, \Sigma, \delta, q_0, F)$ be the NFA recognizing some language A.
- Construct a DFA $M = (Q', \Sigma, \delta', q_0', F')$ which recognizes A.
  - $Q' = \mathcal{P}(Q)$, power set of Q.
    - Every state of M is a set of states of N.
  - Let $E(R) = \{q \mid q$ can be reached from $R \subseteq Q$ by traveling along 0 or more ε arrows, including the members of R themselves$\}$.
  - For $B \in Q'$ and $a \in \Sigma$, $\delta'(B, a) = \{q \in Q' \mid q \in E(\delta(r, a))$ for some $r \in B\}$.
    - Each state B may go to a set of states after reading any symbol a. So, we take the union of all these sets.
  - $q_0' = E(\{q_0\})$.
    - M starts at the state corresponding to the collection containing all the possible states that can be reached from the start state of N along with the ε arrows.
  - $F' = \{D \in Q' \mid D$ contains an accept state of $N\}$.

NFA-DFA Equivalence



Let, the above NFA $N_2=(Q_2, \Sigma, \delta_2, b_1, F_2)$.

$Q_2 = \{b_1,b_2,b_3\}$; $\Sigma = \{0,1\}$;

b_1 = start state; $F_2 = \{b_1\}$.

δ_2 is given as –

	0	1	ε
b_1	ϕ	$\{b_2\}$	$\{b_3\}$
b_2	$\{b_2, b_3\}$	$\{b_3\}$	ϕ
b_3	$\{b_1\}$	ϕ	ϕ

Equivalent DFA $M = (Q, \Sigma, \delta, q_0, F)$.

$Q = \{b_1,b_2,b_3\} = \mathcal{P}(Q)$

$Q = \{ \phi, \{b_1\}, \{b_2\}, \{b_3\}, \{b_1, b_2\}, \{b_1, b_3\}, \{b_2, b_3\}, \{b_1, b_2, b_3\} \}$;

δ is given as –

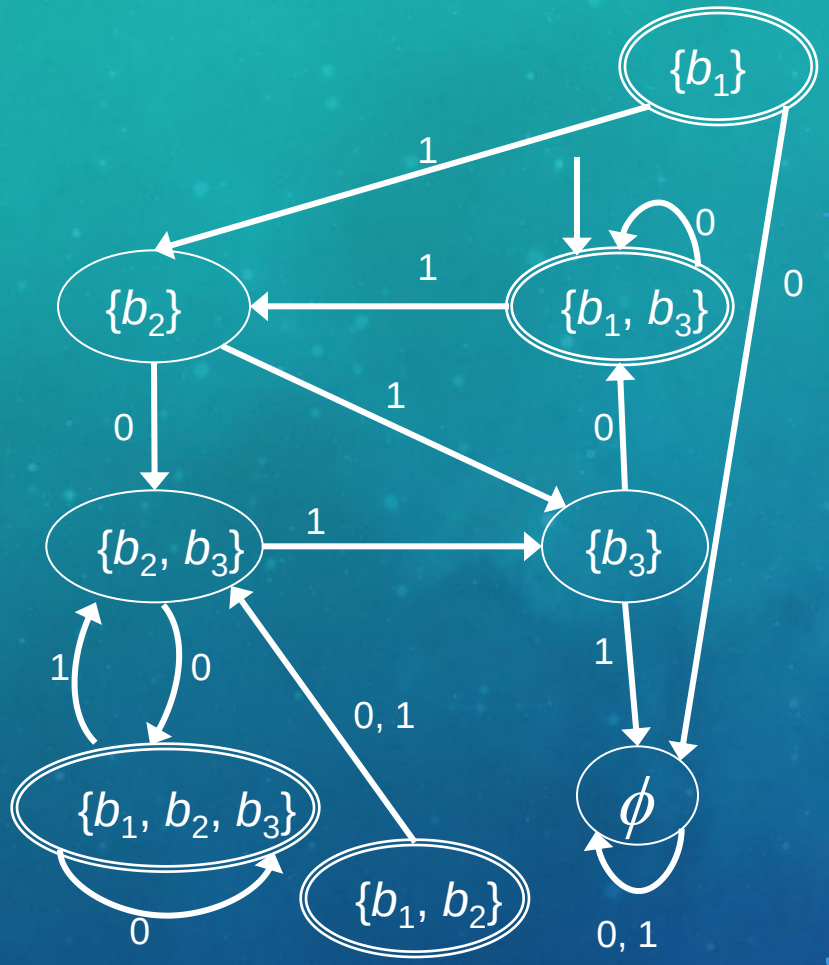
	δ 0	1
ϕ	δ ϕ	ϕ
$\{b_1\}$	δ ϕ	$\{b_2\}$
$\{b_2\}$	δ $\{b_2, b_3\}$	$\{b_3\}$
$\{b_3\}$	δ $\{b_1, b_3\}$	ϕ
$\{b_1, b_2\}$	δ $\{b_2, b_3\}$	$\{b_2, b_3\}$
$\{b_1, b_3\}$	δ $\{b_1, b_3\}$	$\{b_2\}$
$\{b_2, b_3\}$	δ $\{b_1, b_2, b_3\}$	$\{b_3\}$
$\{b_1, b_2, b_3\}$	δ $\{b_1, b_2, b_3\}$	$\{b_2, b_3\}$

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

$q_0 = E(\{b_1\}) = \{b_1, b_3\}$ is the start state;

$F = \{\{b_1\}, \{b_1, b_2\}, \{b_1, b_3\}, \{b_1, b_2, b_3\}\}$.

NFA-DFA Equivalence



Remove the states with no incoming arrows.

Equivalent DFA $M = (Q, \Sigma, \delta, q_0, F)$.

$Q = \{b_1, b_2, b_3\} = \mathcal{P}(Q)$

$Q = \{ \phi, \{b_1\}, \{b_2\}, \{b_3\}, \{b_1, b_2\}, \{b_1, b_3\}, \{b_2, b_3\}, \{b_1, b_2, b_3\} \};$

δ is given as –

δ	0	1
ϕ	ϕ	ϕ
$\{b_1\}$	ϕ	$\{b_2\}$
$\{b_2\}$	$\{b_2, b_3\}$	$\{b_3\}$
$\{b_3\}$	$\{b_1, b_3\}$	ϕ
$\{b_1, b_2\}$	$\{b_2, b_3\}$	$\{b_2, b_3\}$
$\{b_1, b_3\}$	$\{b_1, b_3\}$	$\{b_2\}$
$\{b_2, b_3\}$	$\{b_1, b_2, b_3\}$	$\{b_3\}$
$\{b_1, b_2, b_3\}$	$\{b_1, b_2, b_3\}$	$\{b_2, b_3\}$

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

$q_0 = E(\{b_1\}) = \{b_1, b_3\}$ is the start state;

$F = \{\{b_1\}, \{b_1, b_2\}, \{b_1, b_3\}, \{b_1, b_2, b_3\}\}$.

Week 5

Regular Expression

- Regular expression describes languages.
- Regular expression can be build up using regular operations.
- Precedence order: * • ∪
- Example:
  - $(0 \cup 1)0^* = (\{0\} \cup \{1\}) \bullet \{0\}^* = \{0,1\} \bullet \{0\}^*$
    $A = \{w \mid \text{string } w \text{ starts with a 0 or a 1 followed by zero or more 0's}\}$
  - $(0 \cup 1)^* = (\{0\} \cup \{1\})^* = \{0,1\}^*$
    $A = \{\text{all possible string with 0s and/or 1s}\}$.

Formal Definition of Regular Expression

- ***R*** is a regular expression if ***R*** is –
  - a for some $a \in \Sigma$, represents the language $\{a\}$.
  - ε, represents the language $\{\varepsilon\}$ containing a single string, namely, the empty string.
  - ϕ, represents the empty language that doesn't contain any string. $L(\phi^*) = \{\varepsilon\}$.
  - $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
    - $R \cup \phi = R$, but $R \cup \varepsilon$ may not be equal to R.
  - $(R_1 \bullet R_2)$, where R_1 and R_2 are regular expressions,
    - $R \bullet \varepsilon = R$, but $R \bullet \phi$ may not be equal to R.
  - (R_1^*), where R_1 is a regular expressions,

EQUIVALENCE WITH FINITE AUTOMATA

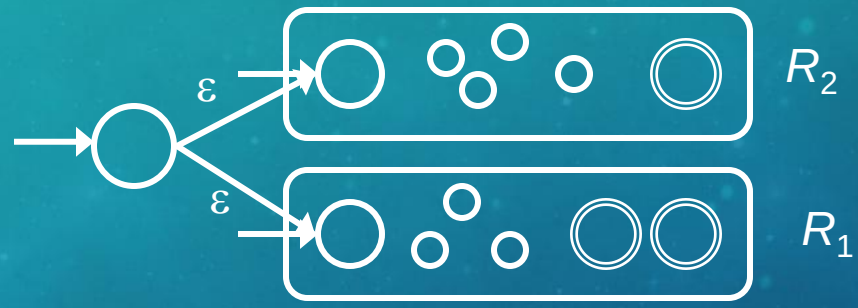
- Let convert regular language R into an NFA considering the six cases in the formal definition of regular language.
  - $R = a$, $a \in \Sigma$. Then $L(R)=\{a\}$, and the NFA that recognizes $L(R)$ is –

  - $R = \varepsilon$. Then $L(R)=\{\varepsilon\}$, and the NFA that recognizes $L(R)$ is –

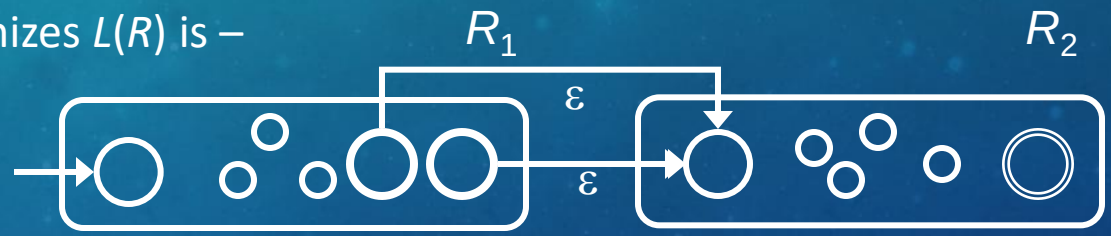
  - $R = \phi$. Then $L(R)= \phi$, and the NFA that recognizes $L(R)$ is –

EQUIVALENCE WITH FINITE AUTOMATA

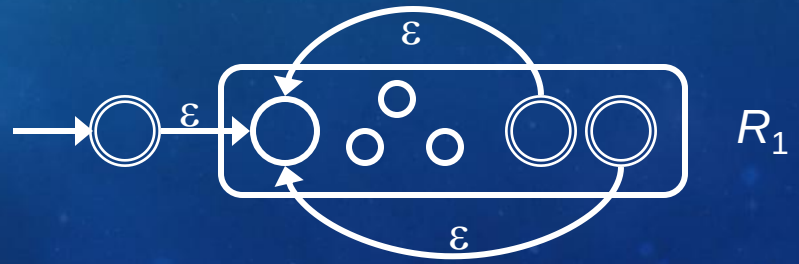
- $R = R_1 \cup R_2$. Then $L(R)=\{R_1,R_2\}$, and the NFA that recognizes $L(R)$ is –



- $R = R_1 \bullet R_2$. Then $L(R)=\{R_1R_2\}$, and the NFA that recognizes $L(R)$ is –

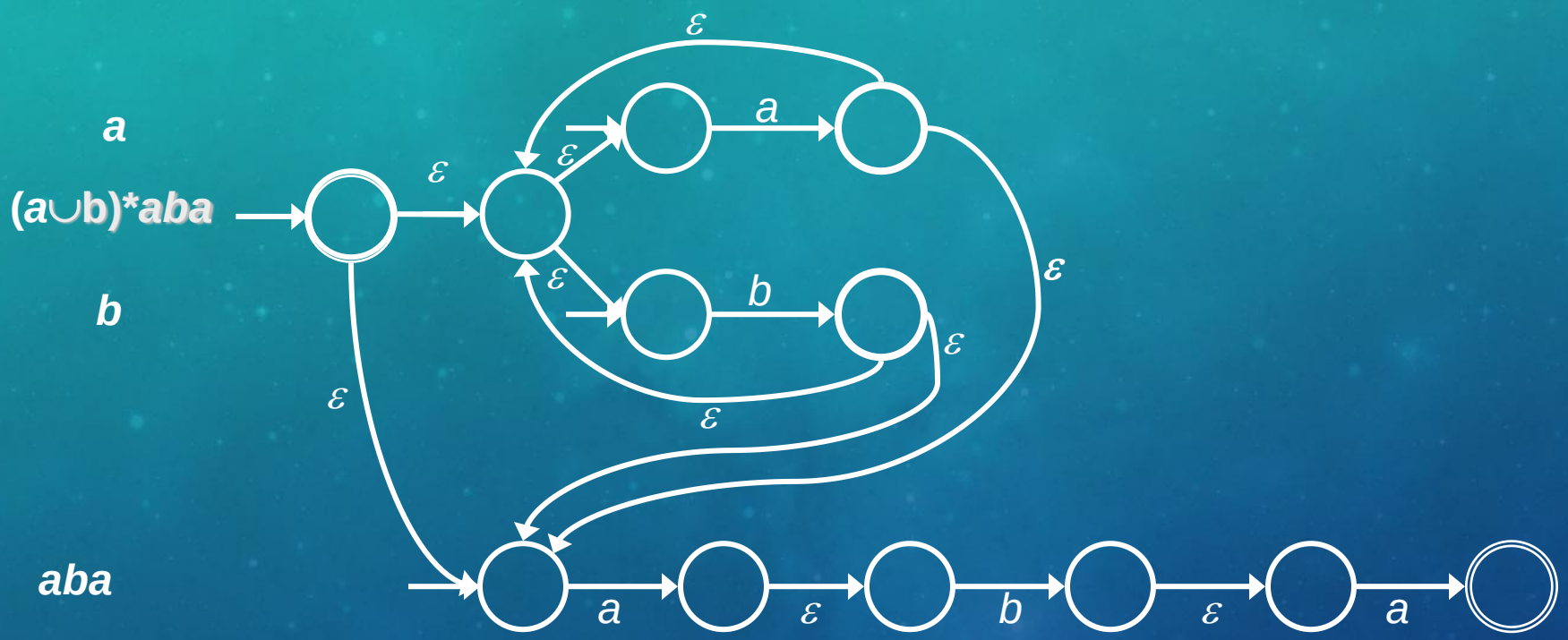


- $R = R_1^*$. Then $L(R)=\{R_1\}^*$, and the NFA that recognizes $L(R)$ is –



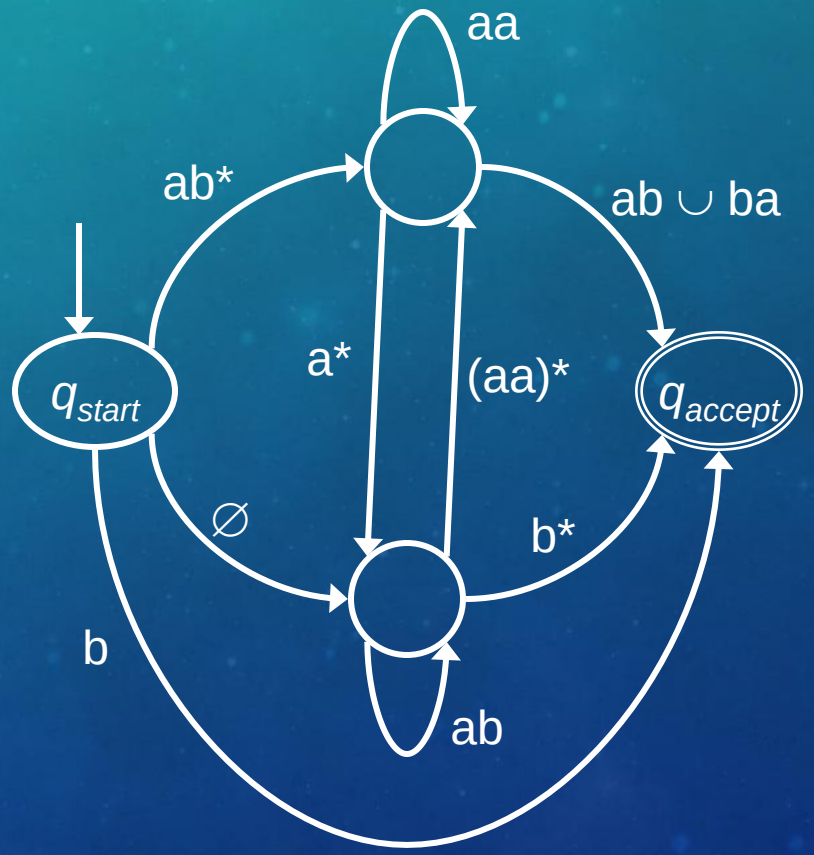
CONVERTING A REGULAR EXPRESSION TO AN NFA

- Building an NFA from regular expression: $(a \cup b)^* aba$



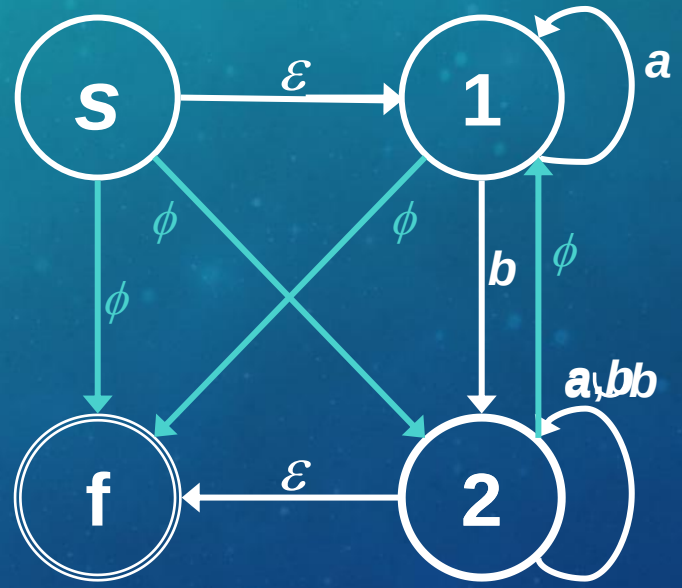
Converting a DFA to a regular expression

- This can be done in two parts. For this we introduce a new type of finite automata called ***generalized nondeterministic automaton***, GNFA.
  - First, we will convert a DFA to GNFA, and
  - then GNFA to regular expression.
- GNFA has the following special form –
  - Transition labels might be in regular expression form.
  - The start state doesn't have any incoming arrow from any other state.
  - There is only one accept state, and it doesn't have any outgoing arrow to any other state.
  - Start state is never the same as accept state.
  - There is only one outgoing arrow to any other state and to itself, except the start and accept states. We will consider ϕ labeled outgoing arrows, if no transition exists between any two states.



Converting a DFA to GNFA

- Add a new start state with an ε arrow to the old start state.
- Add new accept state with ε arrows from the old accept states.
- If any arrows have multiple labels, union the previous labels into one label.
- Add arrows with ϕ label between states where there are no arrows. This won't change the language as ϕ label arrows can never be used.
  - Even we might ignore adding such arrows, as these are arrows which can be assumed to be there with no use.



Formal Definition of GNFA

- A generalized nondeterministic finite automaton is a 5-tuple, $(Q, \Sigma, \delta, q_{start}, q_{accept})$ where –
  - Q is the finite set of states,
  - Σ is the input alphabet,
  - $\delta: (Q - \{q_{start}\}) \times (Q - \{q_{accept}\}) \rightarrow \mathcal{R}$ is the transition function,
  - q_{start} is the start state,
  - q_{accept} is the accept state.
- A GNFA accepts a string w in Σ^* if $w = w_1 w_2 \ldots w_k$, where each w_i is in Σ^* and a sequence of states $q_0, q_1, \ldots q_k$ exists such that –
  - $q_0 = q_{start}$ is the start state,
  - $q_k = q_{accept}$ is the accept state, and
  - For each i, we have $w_i \in L(R_i)$, where $R_i = \delta(q_{i-1}, q_i)$; i.e., R_i is the expression on the arrow from q_{i-1} to q_i.

Week 6

CFL

SYMBOL

Symbol: Is a basic building block of Theory of Computation.

e.g. a,b,….z(Latters)

0,1,….9(Digit)

ALPHABET

Alphabet: Is a finite set of symbols.

Σ(Sigma)

e.g. Σ={a,b}

Σ={0,1}

Σ={0,1,….9}

Σ={a,b,c}

This all are finite set

STRING

String: Is a finite sequence of symbol.

ɯ(String)

e.g. ɯ={0,1}

ɯ=0110

ɯ=1010

LENGTH OF STRING

Length of String: |ɯ|

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

$$ɯ = ababba = 6$$

$$| ɯ | = 6 \text{ (length is 6)}$$

EMPTY STRING

Empty String: **ε(Epsilon) or λ(Lambda)**

Σ={0,1}

0 is a string over the Σ of length 1

10 is a string over the Σ of length 2

101 is a string over the Σ of length 3

e.g. (Ø/{} = Empty Set)

LANGUAGE

Language: Is collection of strings.

(**It can be finite/ infinite**)

e.g. $\Sigma = \{a, b\}$

L1= Set of all strings over Σ of length 2

={aa,ab,ba,bb} **Finite Set**

L2= Set of all strings over Σ of length 3

={aaa,aab,aba,abb,baa,bab,bba,bbb} **Finite Set**

L3= Set of all strings over Σ where each string starts with ‘a’

={a,aa,ab,aaa,aba,aaaa,.....} **Infinite set**

POWER OF Σ

Σ={a,b}

Σ1 = Set of all strings over this Σ of length 1

={a,b}

Σ2 = Set of all strings over this Σ of length 2

={aa,ab,ba,bb}

Σ3 = Set of all strings over this Σ of length 3

={aaa,aab,aba,abb,baa,bab,bba,bbb}

Σ° = Set of all strings over this Σ of length 0

={ε}

Σ*

Σ* = **Set of all possible strings**.

$$\Sigma^* = \Sigma^\circ \cup \Sigma 1 \cup \Sigma 2 \cup \Sigma 3$$

$$=\{\varepsilon\} \cup \{a,b\} \cup \{aa,ab,ba,bb\} \ldots\ldots$$

Previous,

$$L1 \subseteq \Sigma^*$$

$$L2 \subseteq \Sigma^*$$

$$L3 \subseteq \Sigma^*$$

So, all language is subset of **Σ***

e.g. ⊆ = Subset

SET

Set: **is a collection of objects.**

S={a,b,c,h,d}

S={1,2,5,6}

Set

1. Empty set S = Ø/{}
2. Not Empty Set S ≠ Ø

Not Empty Set

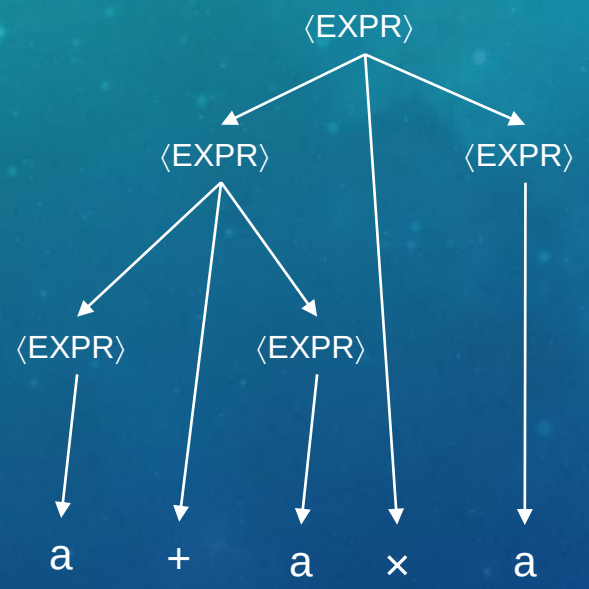
1. Finite Set
2. Infinite Set

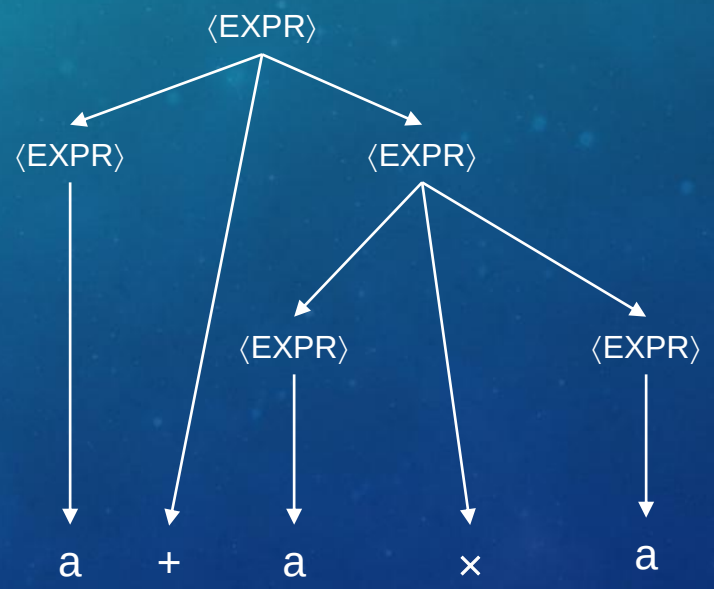
CFG

Ambiguity – Parse Tree

- If a grammar generates the same string in several different ways, we say that the string is derived ***ambiguously*** in the grammar.
- If a grammar generates some string ambiguously we say that the grammar is ***ambiguous***.
- Example: Grammar G, ⟨EXPR⟩→⟨EXPR⟩+⟨EXPR⟩|⟨EXPR⟩×⟨EXPR⟩| (⟨EXPR⟩) |a

Two parse trees for the string a+a×a in G





Ambiguity - Derivation

- When we say that a grammar generates a string ambiguously, we mean that the string has two different parse trees, not two different derivations.
- A derivation of string ***w*** in a grammar ***G*** is a ***leftmost derivation*** if at every step the leftmost remaining variable is the one replaced.
- Then we can say, a string ***w*** is derived ***ambiguously*** in CFG ***G*** if it has two or more different leftmost derivations.
- Grammar ***G*** is ***ambiguous*** if it generates some string ambiguously.
- Some CFLs can only be generated by ambiguous grammars. Such languages are called ***inherently ambiguous***.
  Example: $\{0^i1^j2^k \mid i=j \text{ or } j=k\}$

Chomsky Normal Form

- It is often convenient to have CFGs in simplified form. One such form is Chomsky normal form.
- A context free grammar is in Chomsky normal form if every rule is of the form

$$A \rightarrow BC$$
$$A \rightarrow a$$

where ***a*** is any terminal and ***A***, ***B***, and ***C*** are any variables – except that ***B*** and ***C*** may not be the start variable.

In addition $S \rightarrow \varepsilon$ is permitted, where S is the start variable.

Convert any grammar *G* to Chomsky Normal Form

- Add a new start symbol S_0 and the new rule $S_0 \rightarrow S$, where S was the original start symbol.
- Eliminate all ε rules of the form $A \rightarrow \varepsilon$, where A is not the start symbol.
  - Add rule $R \rightarrow uv$ for every rule of the form $R \rightarrow uAv$, where u and v are strings of variables and terminals.
  - Add such rules for every occurrence of A. for example, add $R \rightarrow uvAw$, $R \rightarrow uAvw$, $R \rightarrow uvw$ for the rule of the form $R \rightarrow uAvAw$.
  - Add $R \rightarrow \varepsilon$ for the rule of the form $R \rightarrow A$, unless we have previously removed the rule $R \rightarrow \varepsilon$.

Convert any grammar *G* to Chomsky Normal Form

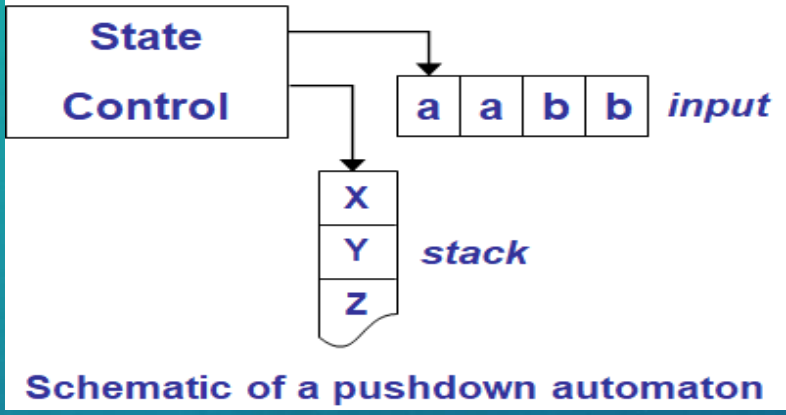
- Eliminate all unit rules of the form $A \rightarrow B$.
  - Add rule $A \rightarrow u$ for the rule of the form $B \rightarrow u$, unless this was a unit rule previously removed.
  - Here u is a string of variables and terminals.
- Convert remaining rules into proper form, $R \rightarrow PQ$ and $R \rightarrow u$.
  - We replace each rule of the form $A \rightarrow u_1 u_2 \ldots u_k$ with the rules $A \rightarrow u_1 A_1$, $A_1 \rightarrow u_2 A_2$, $A_2 \rightarrow u_3 A_3$, … , $A_{k-2} \rightarrow u_{k-1} u_k$.
  - Here $k \geq 3$ and each u_i is a variable or terminal symbol, and A_i's are new variables.
  - If $k \geq 2$, replace any terminal u_i in the preceding rule(s) with the new variable U_i and the rule $U_i \rightarrow u_i$.
- The above procedure converts a Grammar to a Chomsky normal form. Next, we will go through an example.

Week 7

Recognizing Context Free Languages

- Regular Languages (RL) are recognized by the computational model Finite Automaton (FA), examples: DFA, NFA.
- A computational model is required that can recognize some Context Free Language (CFL).
- Based on the definition of the language to be recognized, additional memory with rule of access is required to construct such computational model.
- Push Down Automata (PDA) is the computational model that can recognize some Context Free Language (CFL).
- PDA contains additional memory with the LIFO (Last In First Out) access rule. That is, it maintains a stack where an element is pushed down the stack.
- Hence the name Push Down Automata.

PUSHDOWN AUTOMATA(NON-REGULAR LANGUAGES)



Schematic of a pushdown automaton

- Have an extra component called stack.
- Stack provides additional memory beyond the finite amount available in the control.
- Schematic of a pushdown automaton
  - Control represents the states and transition function
  - The arrow on the tape, containing the input string, represents the input head, pointing at the next input symbol to be read.
  - The arrow on the stack points the top element.
  - Writing symbol on the stack is referred to as ***pushing*** down the symbols.
  - Removing a symbol is referred to as ***popping*** up.
  - The top symbol of the stack can be read and removed at any time.

Formal Definition

- A pushdown automaton is a 6-touple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are all finite sets and
  - Q is the set of states,
  - Σ is the set of alphabet,
    - $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$
  - Γ is the stack alphabet,
    - $\Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$
  - $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$
    - Domain of the transition function is the current state, next input symbol read, and top symbol of the stack.
    - Because of the nondeterminism, i.e. several legal next moves, δ returns a set of members, each containing the next state and the next stack symbol.
  - $q_0 \in Q$ is the start state, and
  - $F \subseteq Q$ is the set of accept states

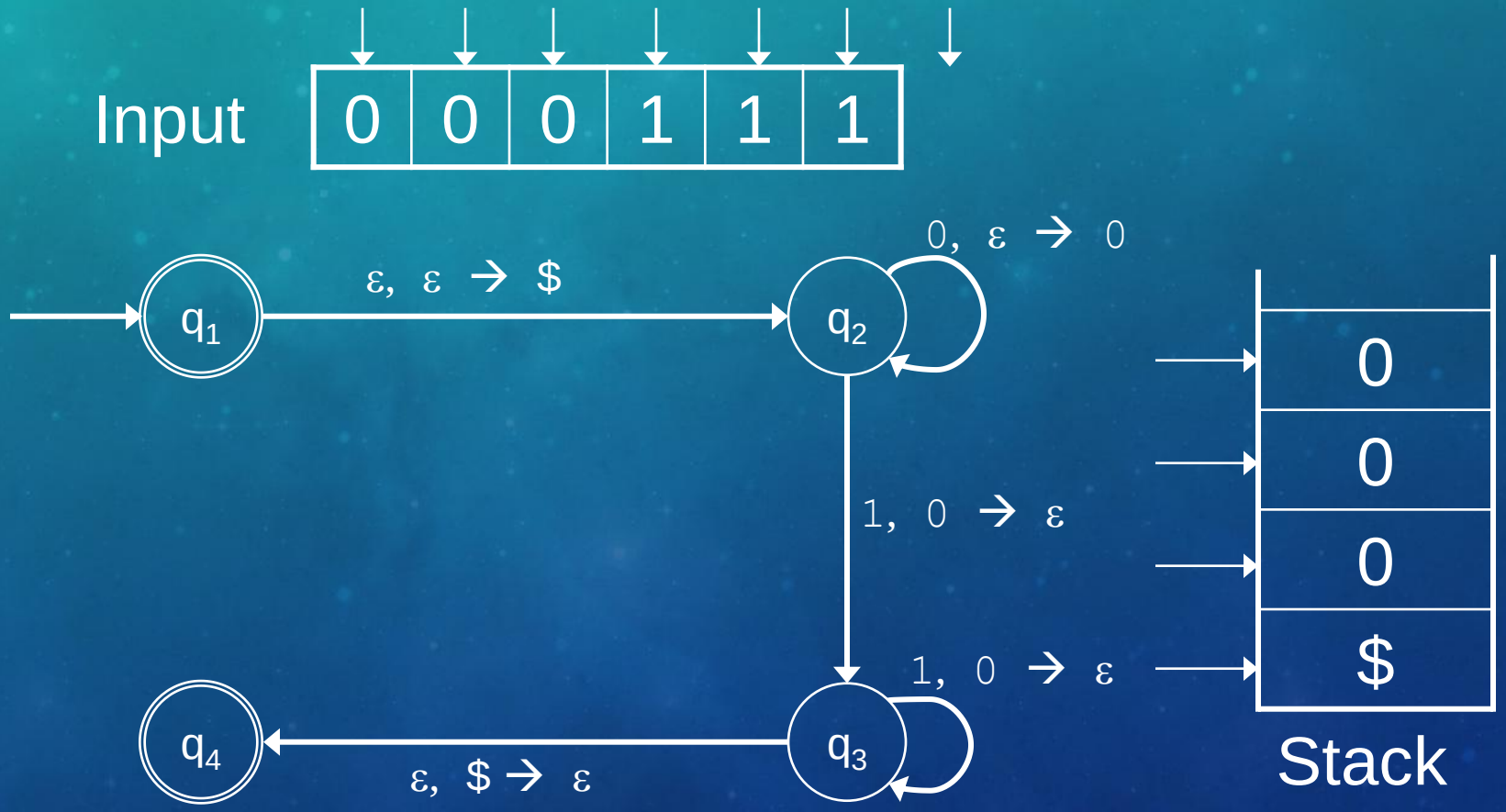
EXAMPLE: PDA

- $L = \{0^n1^n \mid n \geq 0\}$
- $M = (Q, \Sigma, \Gamma, \delta, q_1, F)$, where
  - $Q = \{q_1, q_2, q_3, q_4\}$,
  - $\Sigma = \{0, 1\}$,
  - $\Gamma = \{0, \$\}$,
    - Test for an empty stack is done by initially placing a special symbol $ on the stack. If it ever sees the sign $ again, it knows that the end of stack effectively is empty.
  - $F = \{q_1, q_4\}$,
  - δ is given in the following table, wherein blank entries signify $\varnothing$.

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

Example - State Diagram

- We write "***a***, ***b*** → ***c***" to signify that when the machine is reading an ***a*** from the input it may replace the symbol ***b*** on the top of the stack with a ***c***.
- State diagram for the PDA **M** that recognizes $\{0^n1^n \mid n \geq 0\}$



Transition Table – State Diagram

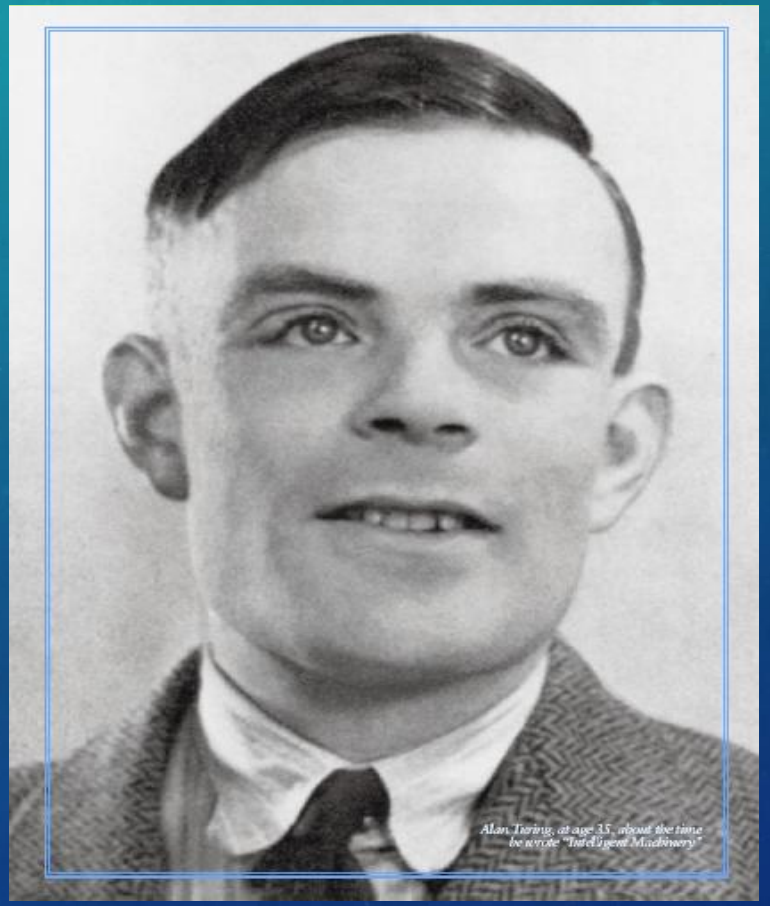
- δ is given in the following table, wherein blank entries signify $\varnothing$.

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

Week 8

FOUNDATIONS

- The theory of computation and the practical application it made possible — the computer — was developed by an Englishman called Alan Turing.



THE TURING MACHINE

•Turing's machine — which came to be called the Turing machine — was this:

•(1) A tape of infinite length

•(2) Finitely many squares of the tape have a single symbol from a finite language.

•(3) Someone (or something) that can read the squares and write in them.

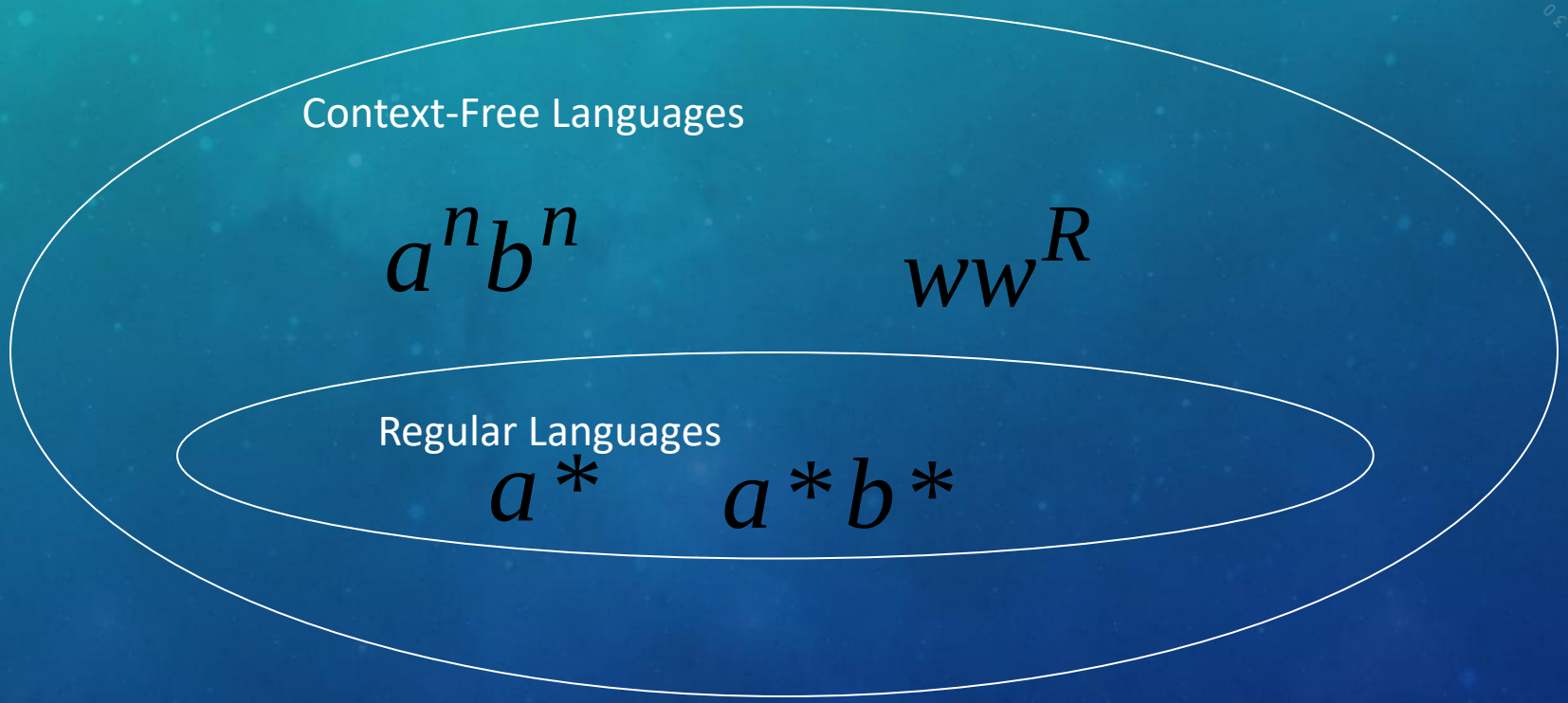
- (4) At any time, the machine is in one of a finite number of internal states.
- (5) The machine has instructions that determine what it does given its internal state and the symbol it encounters on the tape. It can
  - change its internal state;
  - change the symbol on the square;
  - move forward;
  - move backward;
  - halt (i.e. stop).

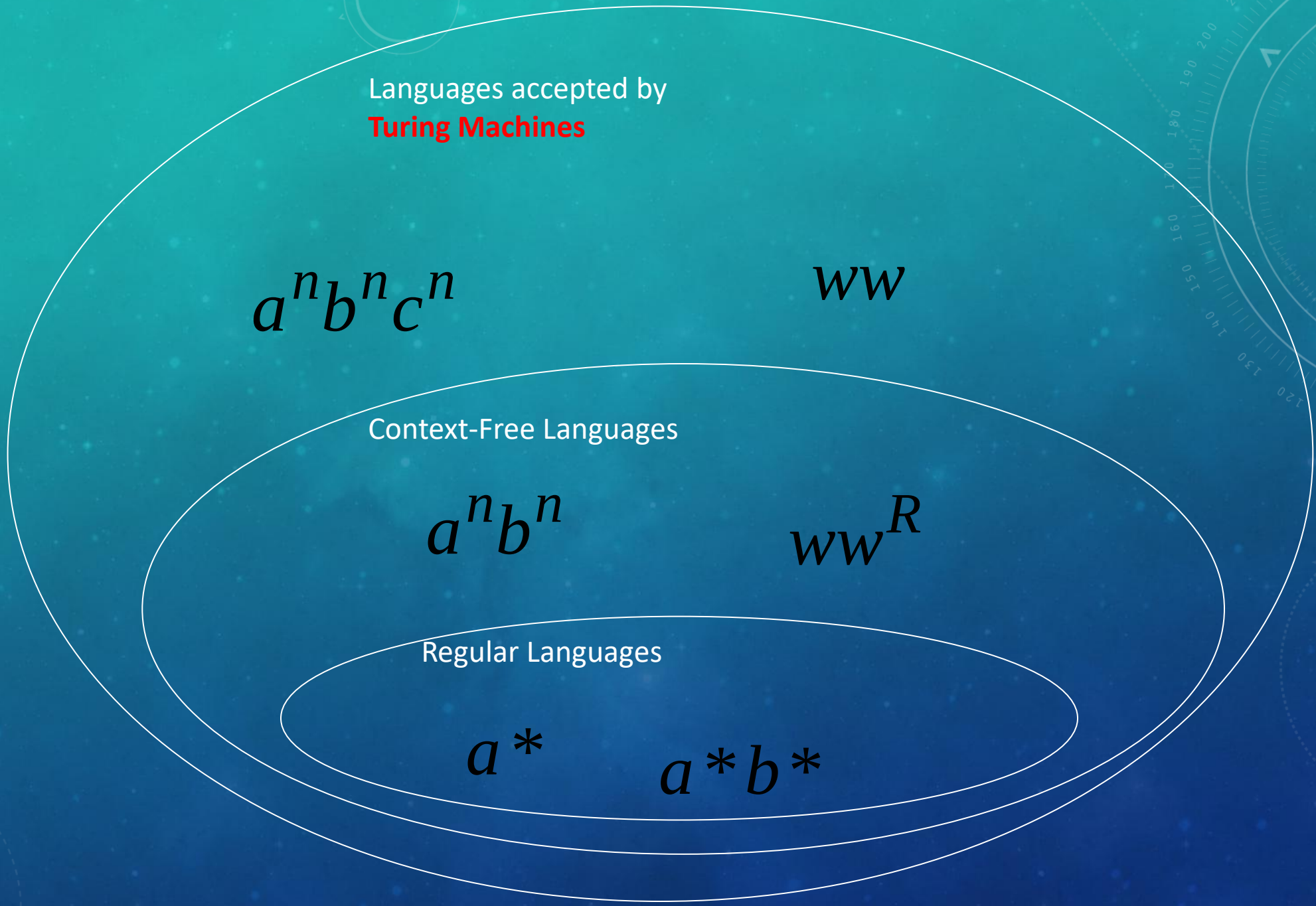
TURING MACHINES

THE LANGUAGE HIERARCHY

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

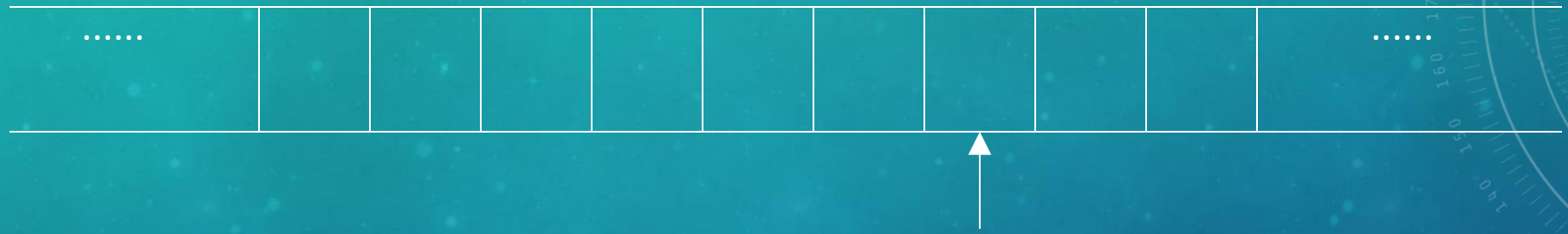
ww ?



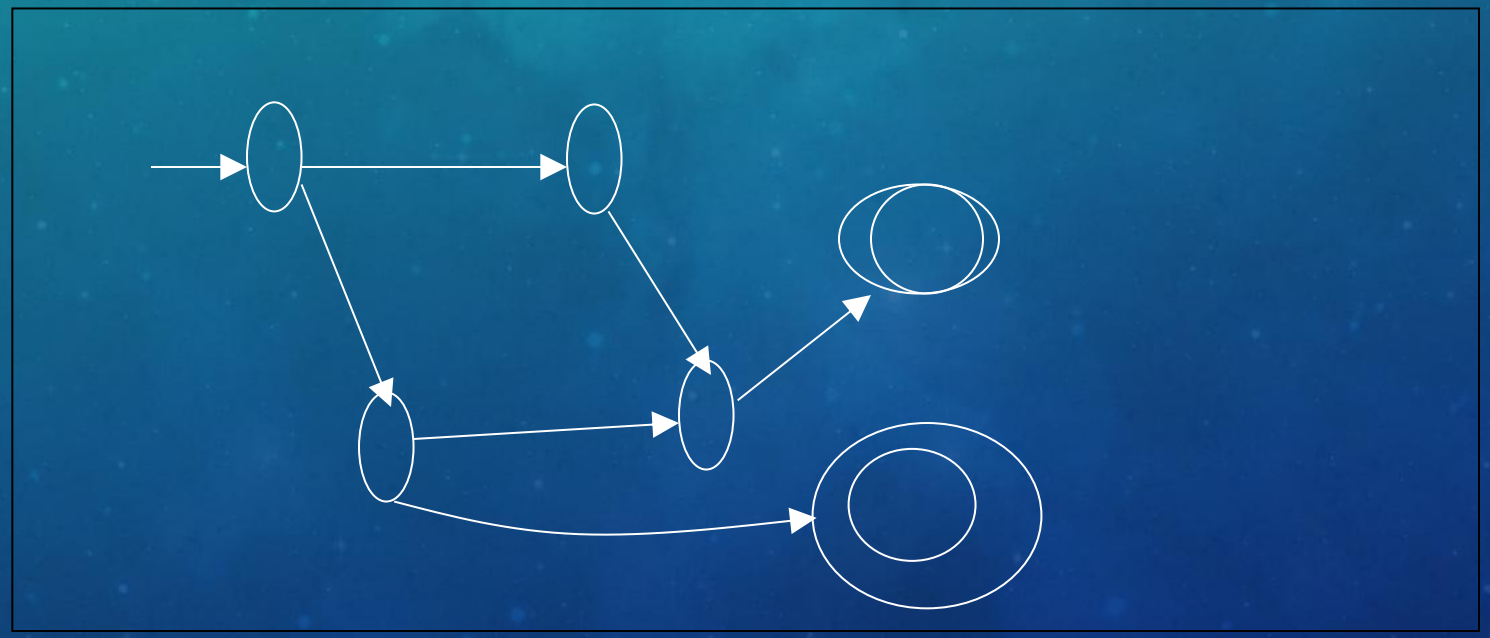

Languages accepted by
Turing Machines
$a^n b^n c^n$
ww
Context-Free Languages
$a^n b^n$
ww^R
Regular Languages
$a*$
$a*b*$

A TURING MACHINE

Tape



Control Unit

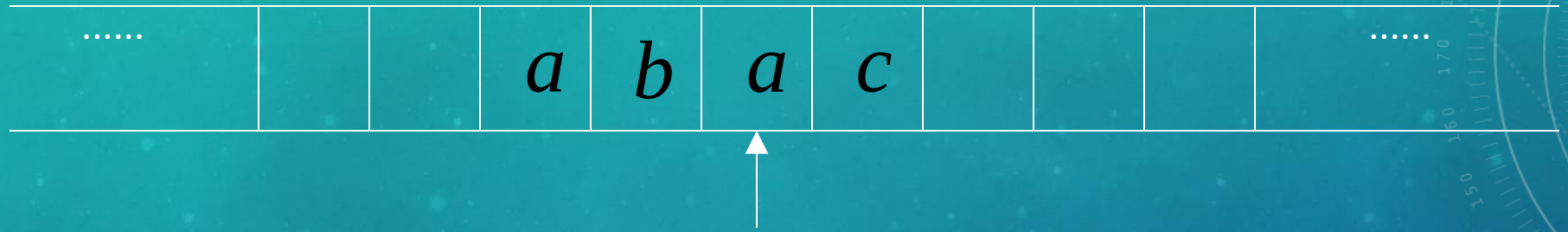


The head at each time step:

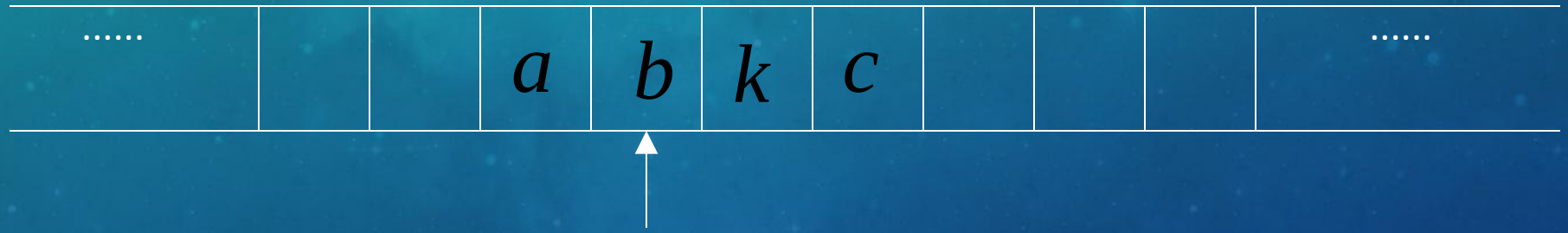
1. Reads a symbol
2. Writes a symbol
3. Moves Left or Right

Example:

Time 0

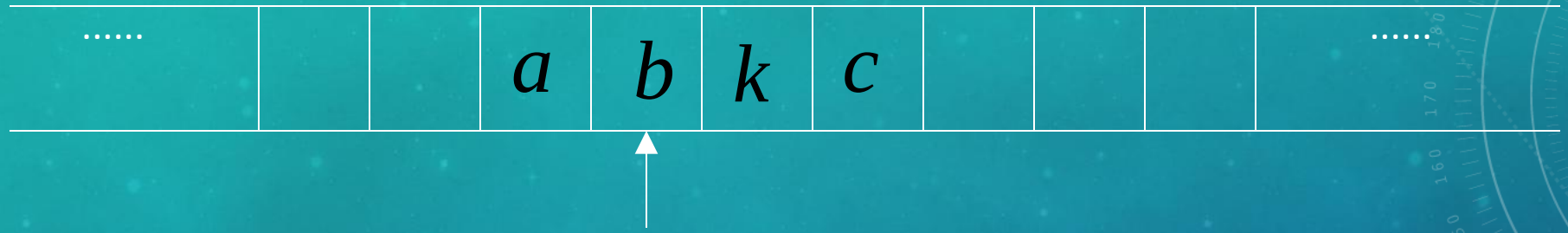


Time 1

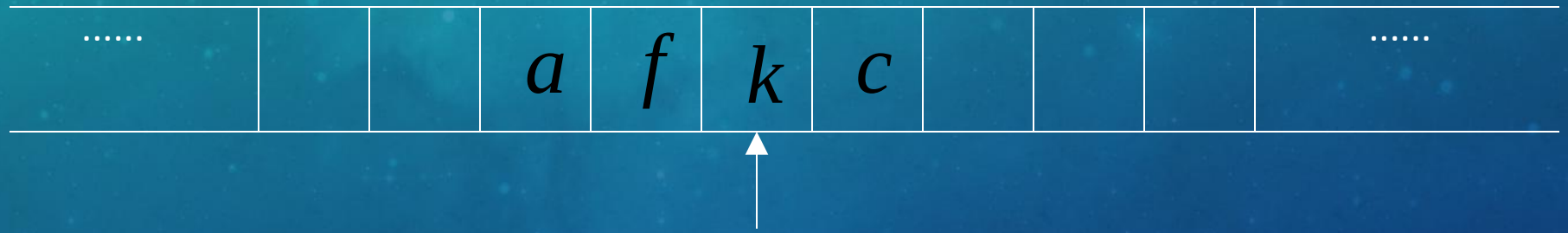


1. Reads a
2. Writes k
3. Moves Left

Time 1

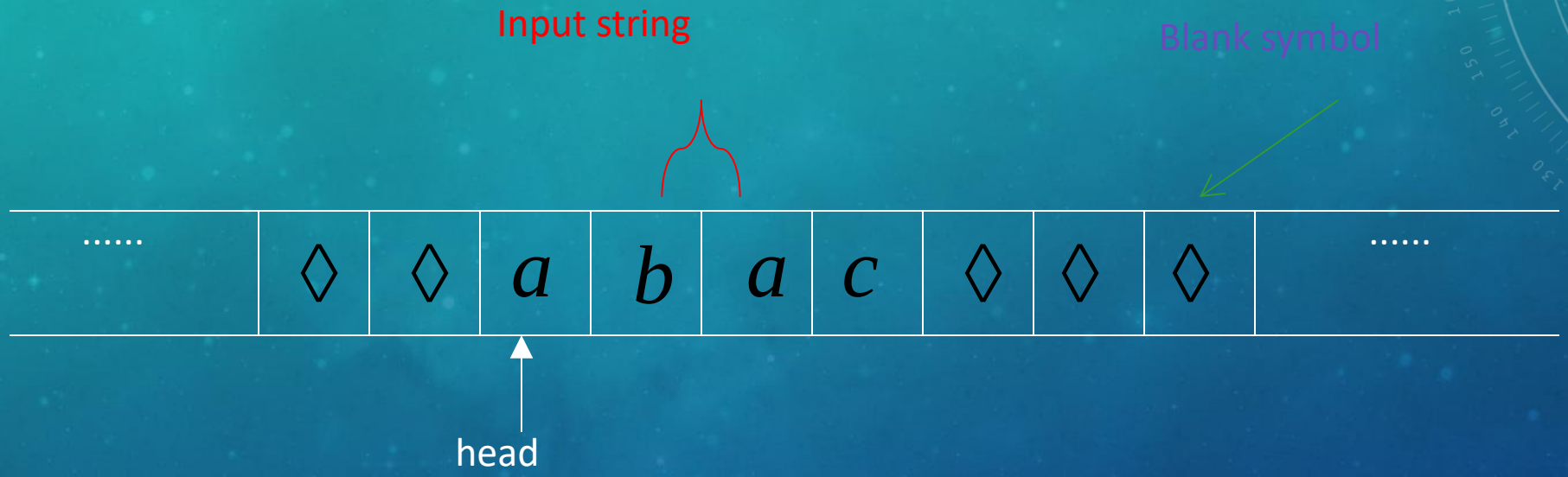


Time 2

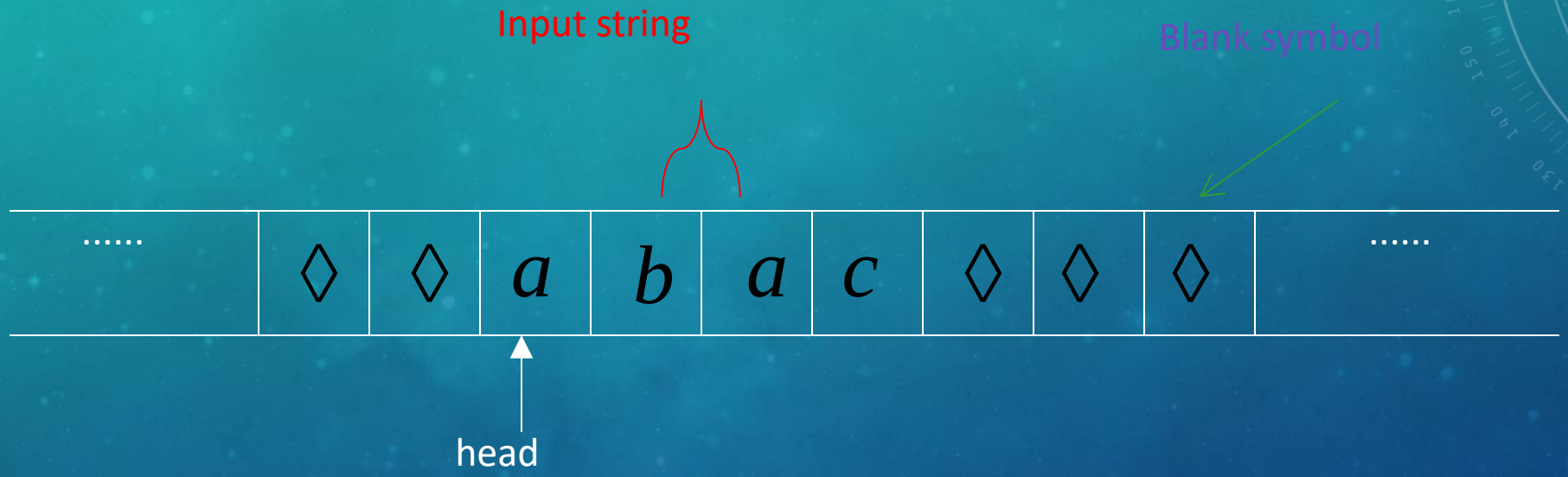


1. Reads b
2. Writes f
3. Moves Right

THE INPUT STRING

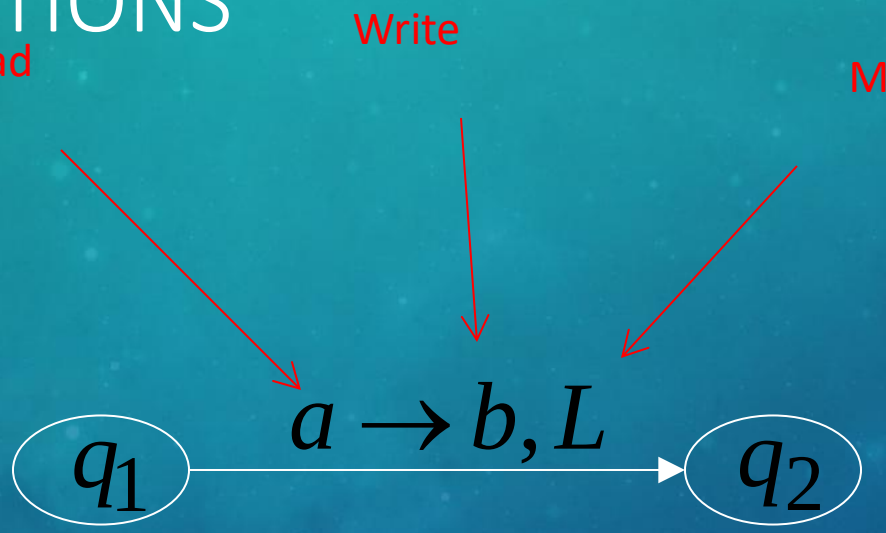


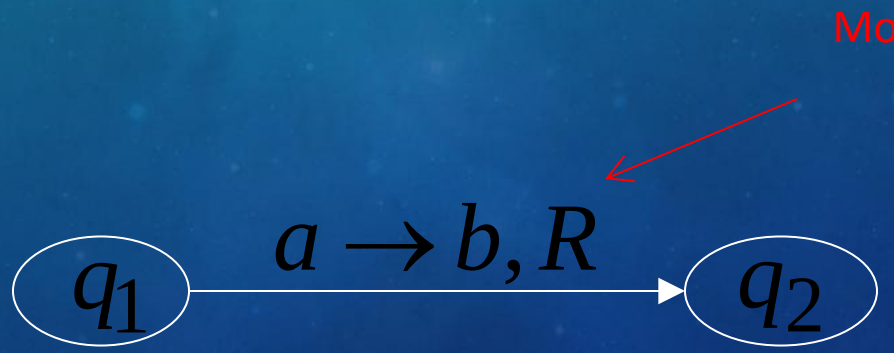
Head starts at the leftmost position of the input string



Remark: the input string is never empty

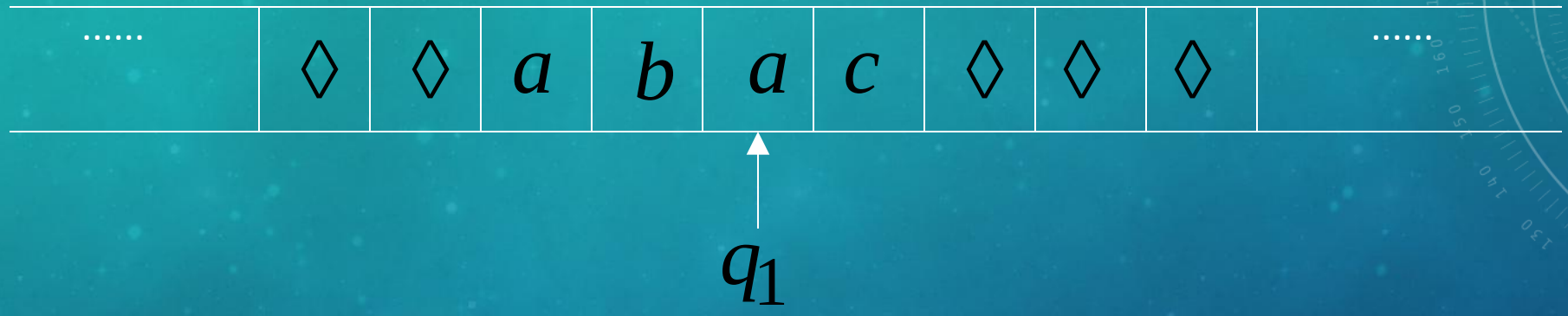
STATES & TRANSITIONS

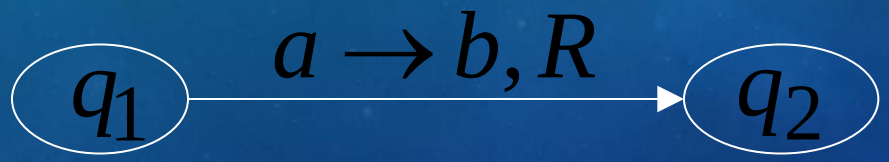




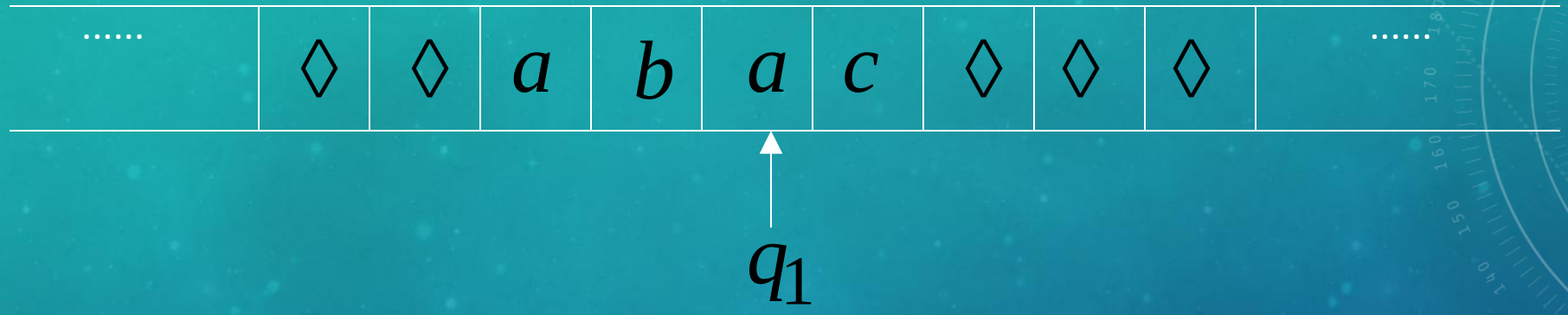
Example:

Time 1

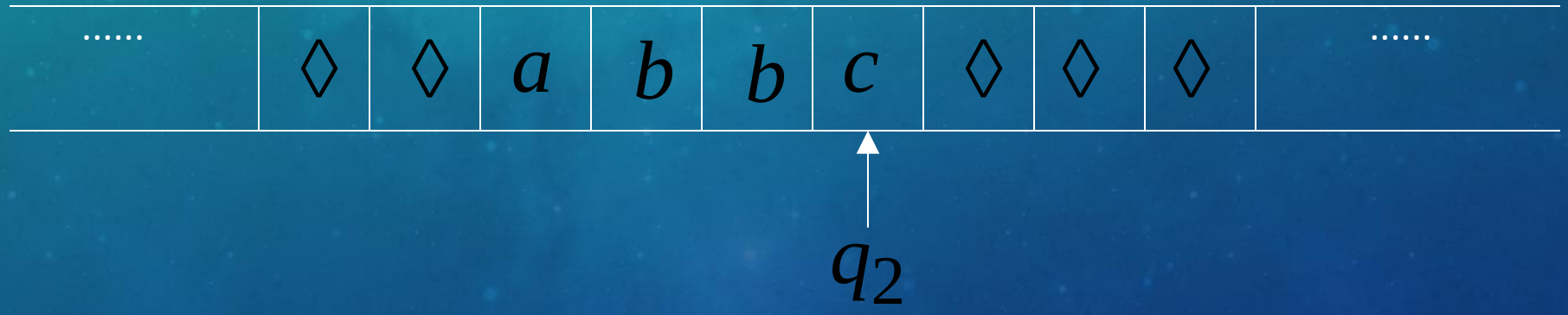


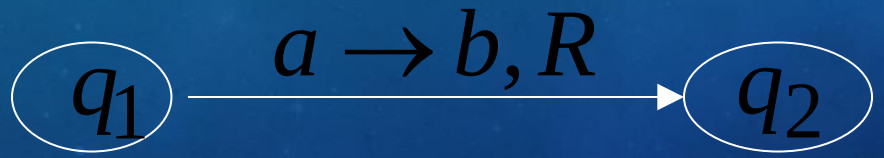


Time 1



Time 2

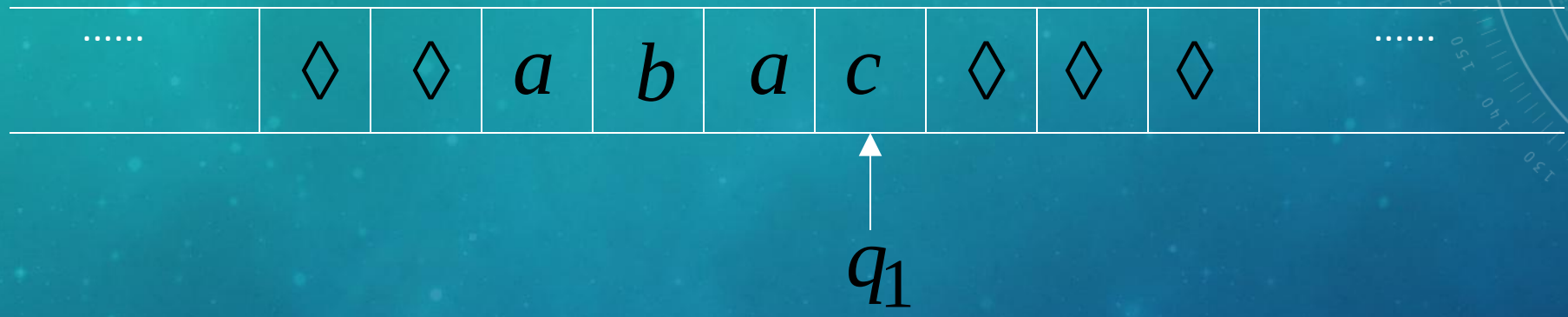


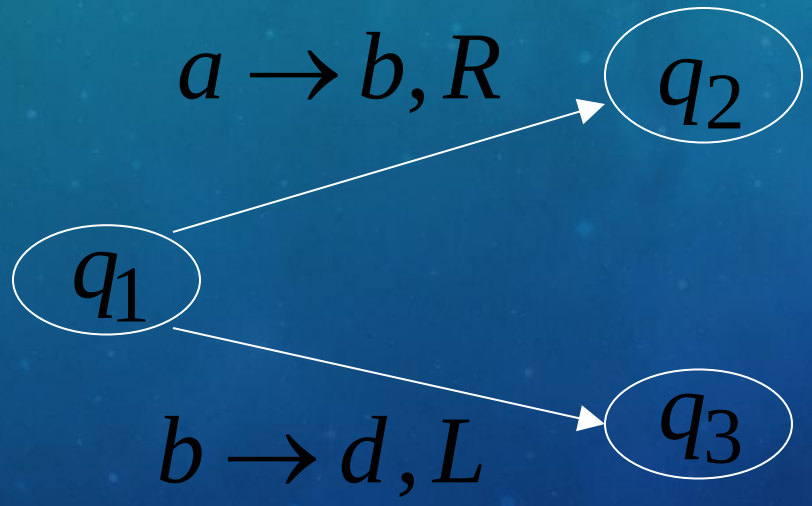


HALTING

The machine ***halts*** if there are no possible transitions to follow

Example:





No possible transition

HALT!!!

FINAL STATES

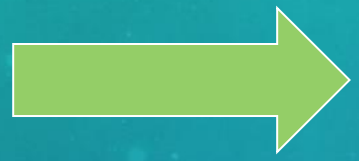
Allowed

Not Allowed

- Final states have no outgoing transitions
- In a final state the machine halts

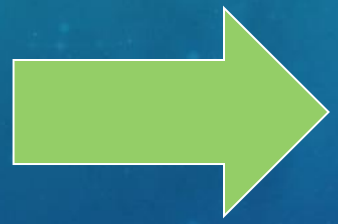
ACCEPTANCE

Accept Input



If machine halts
in a final state

Reject Input



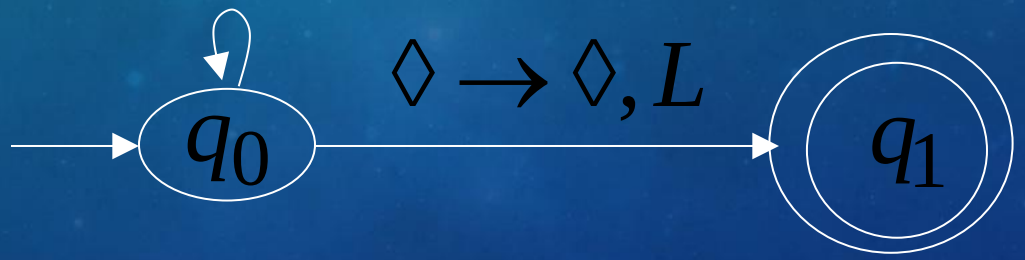
If machine halts
in a non-final state
or
If machine enters
an *infinite loop*

TURING MACHINE EXAMPLE

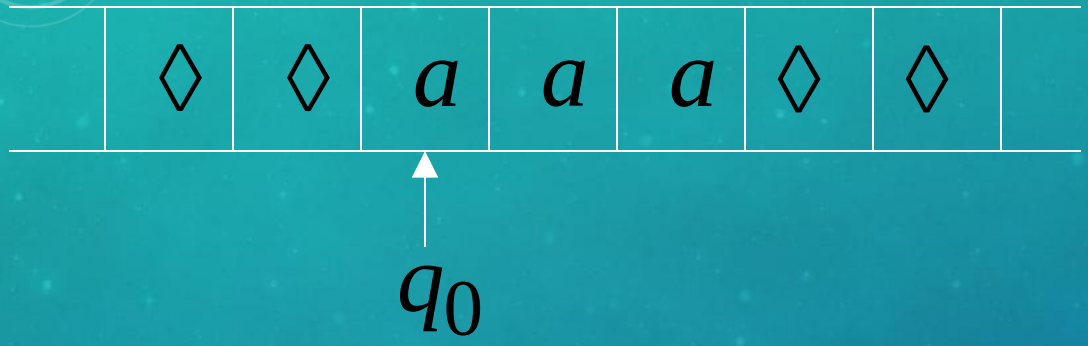
A Turing machine that accepts the language:

$$aa*$$

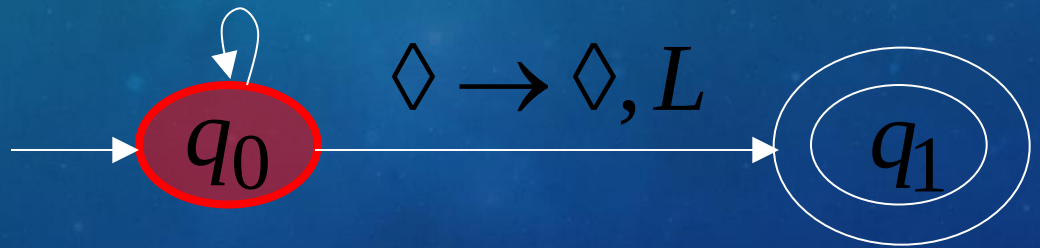
$$a \rightarrow a, R$$



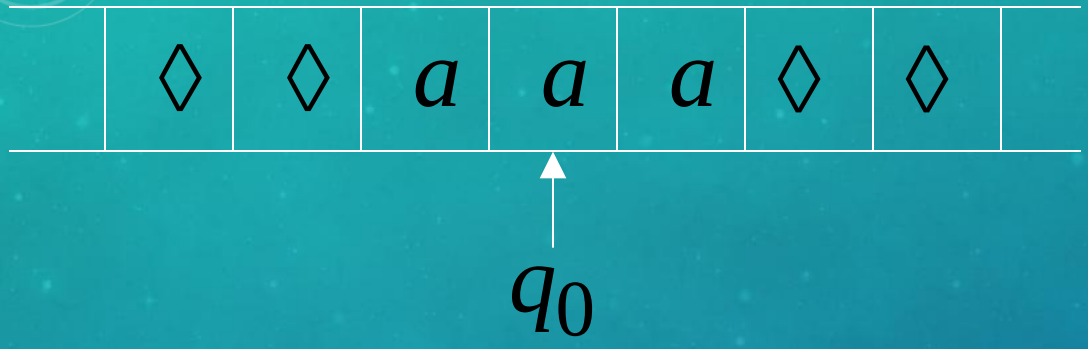
Time 0



$a \rightarrow a, R$

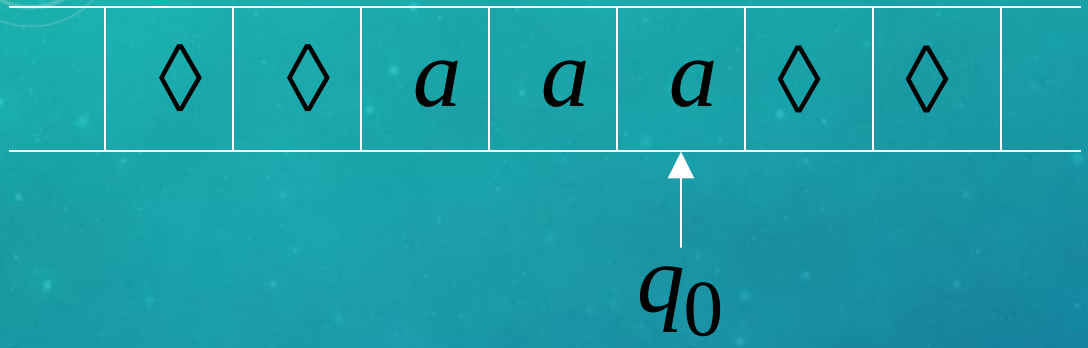


Time 1

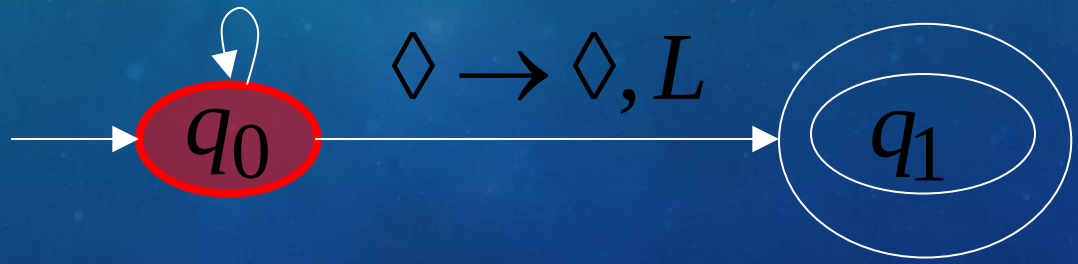


$a \rightarrow a, R$

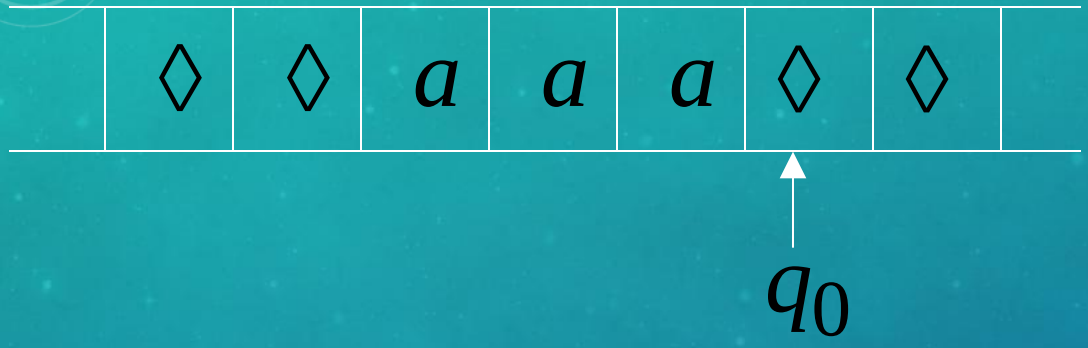
Time 2



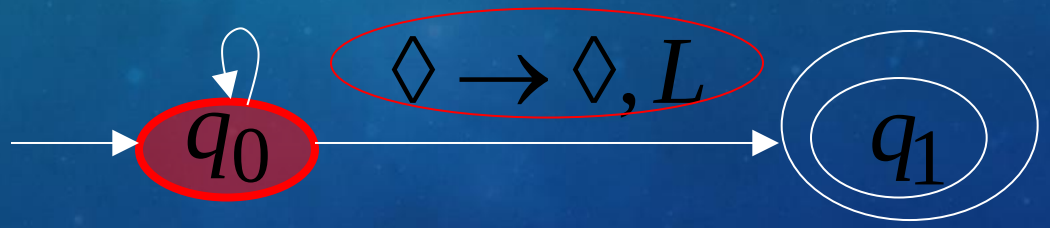
$a \rightarrow a, R$



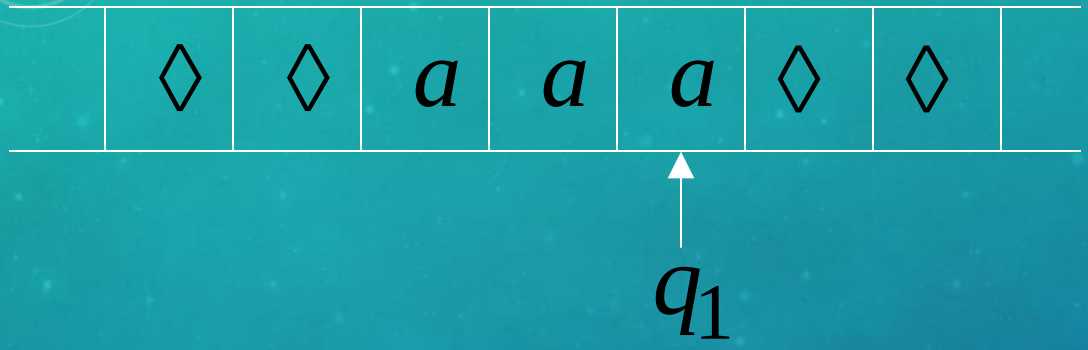
Time 3



$$a \rightarrow a, R$$

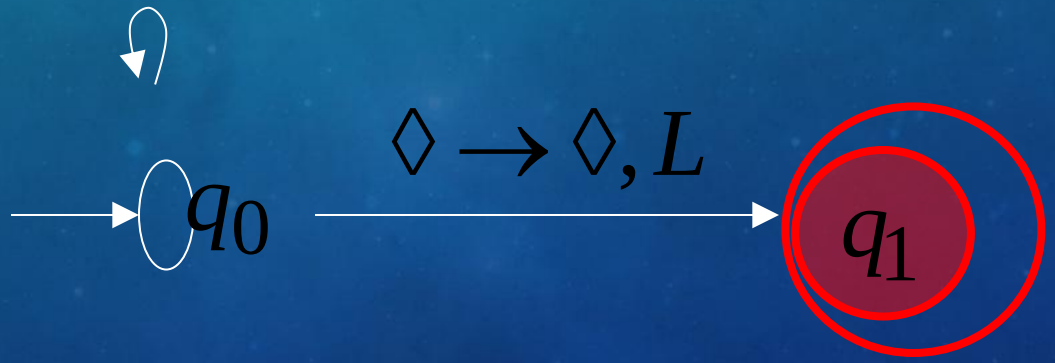


Time 4



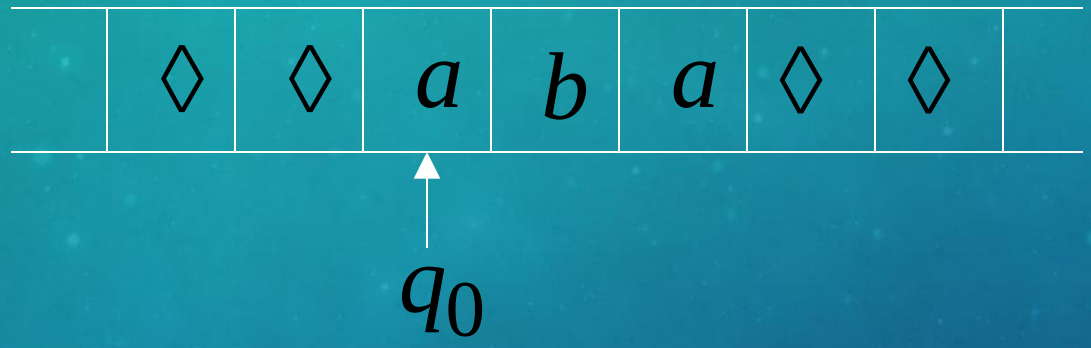
$a \rightarrow a, R$

Halt & Accept

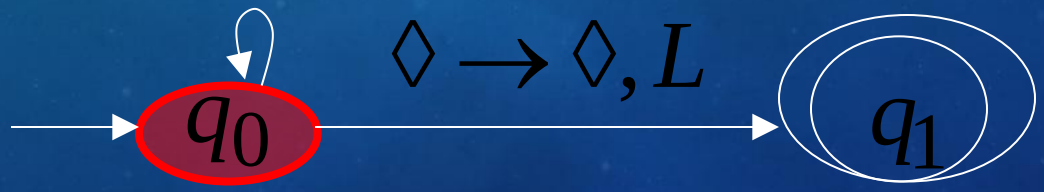


Rejection Example

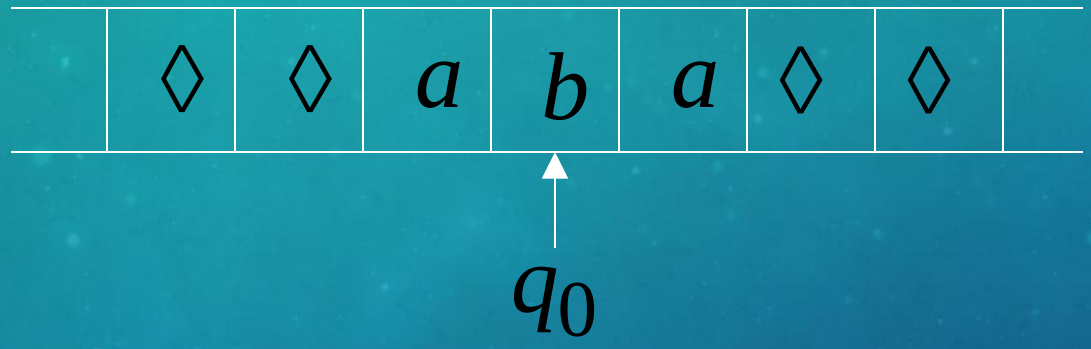
Time 0



$a \rightarrow a, R$



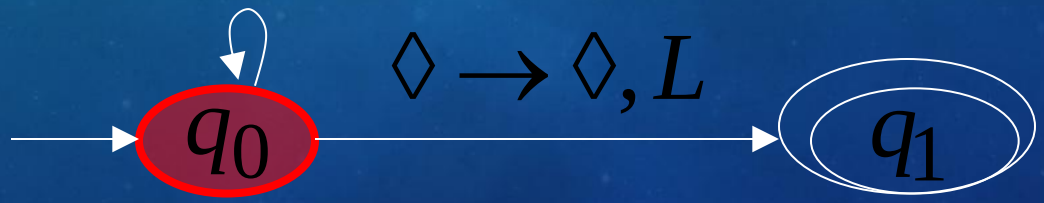
Time 1



No possible Transition

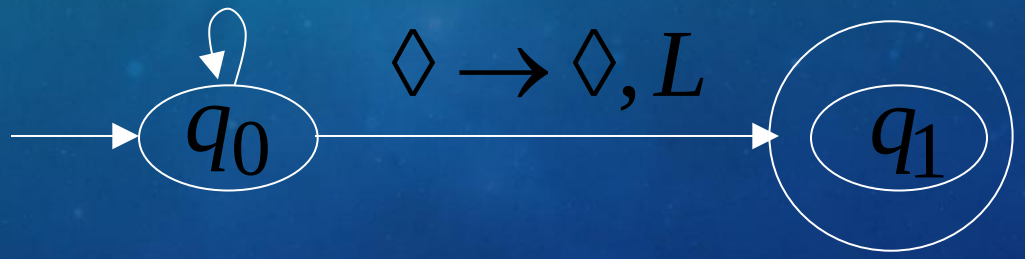
Halt & Reject

$$a \rightarrow a, R$$

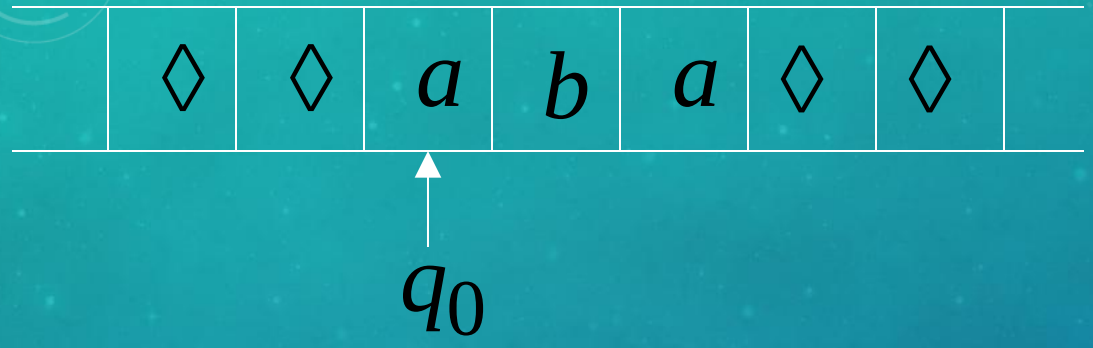


INFINITE LOOP EXAMPLE

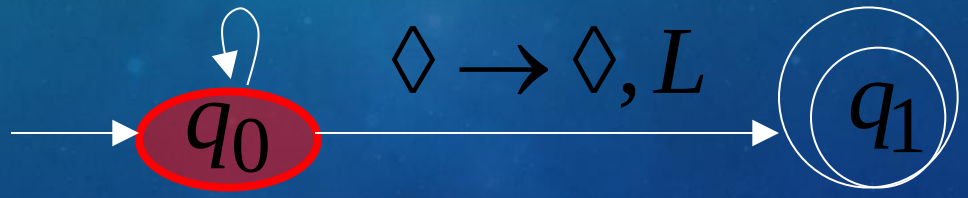
$$b \rightarrow b, L$$
$$a \rightarrow a, R$$



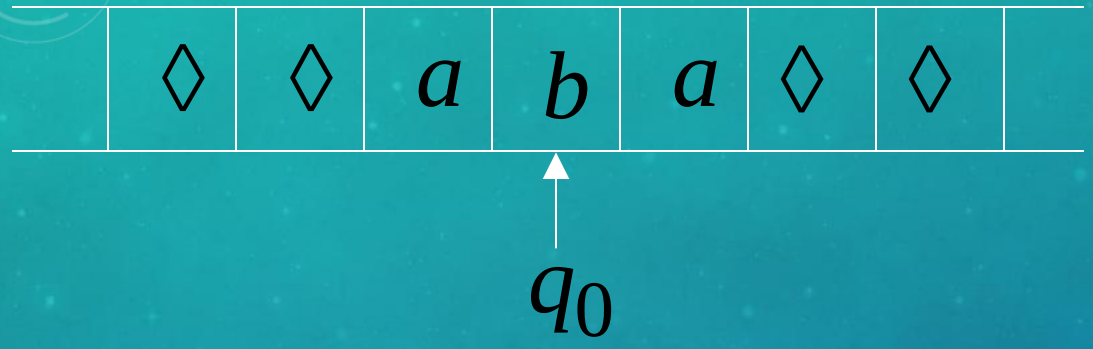
Time 0



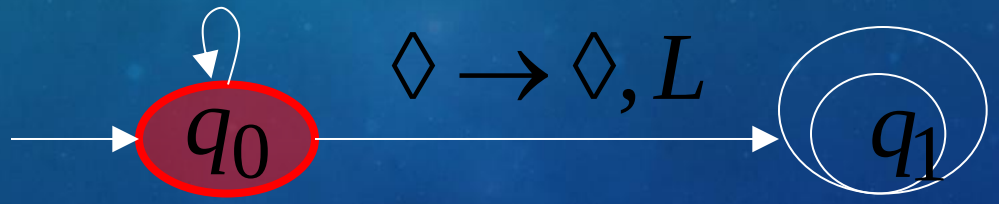
$$b \rightarrow b, L$$
$$a \rightarrow a, R$$



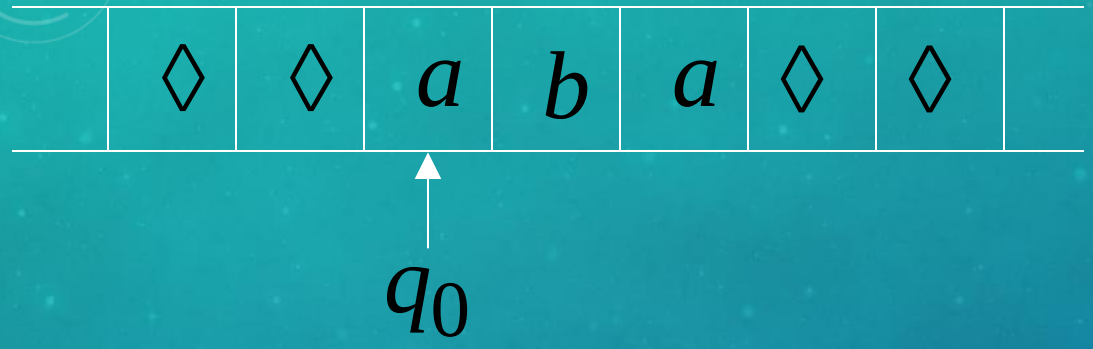
Time 1



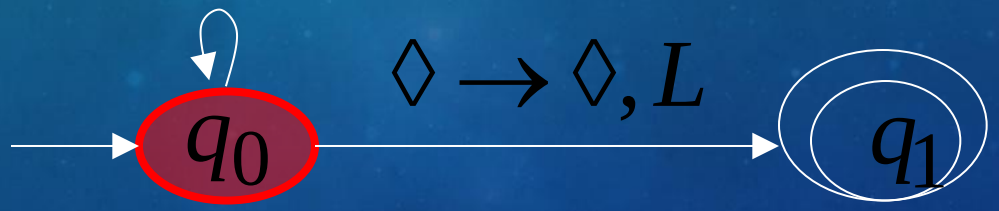
$$b \to b, L$$
$$a \to a, R$$



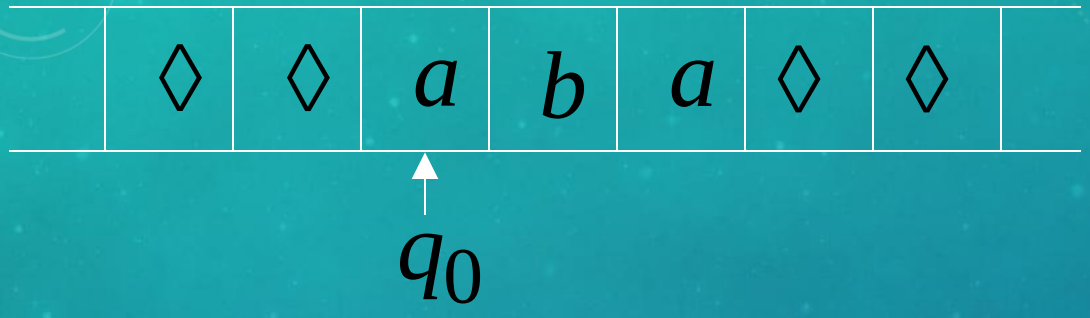
Time 2



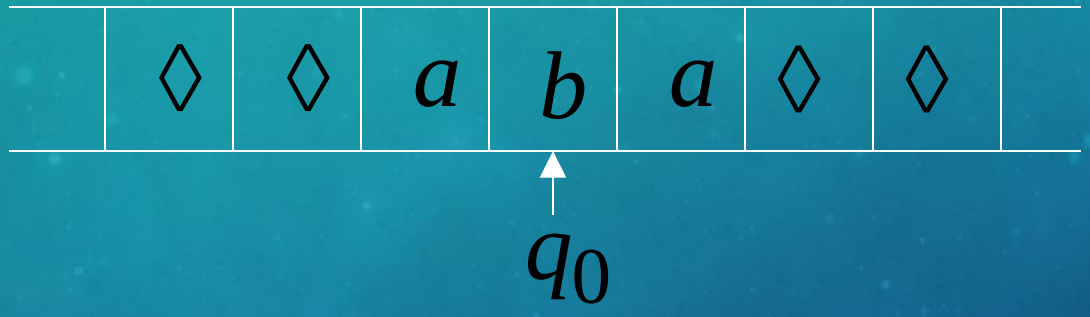
$$b \rightarrow b, L$$
$$a \rightarrow a, R$$



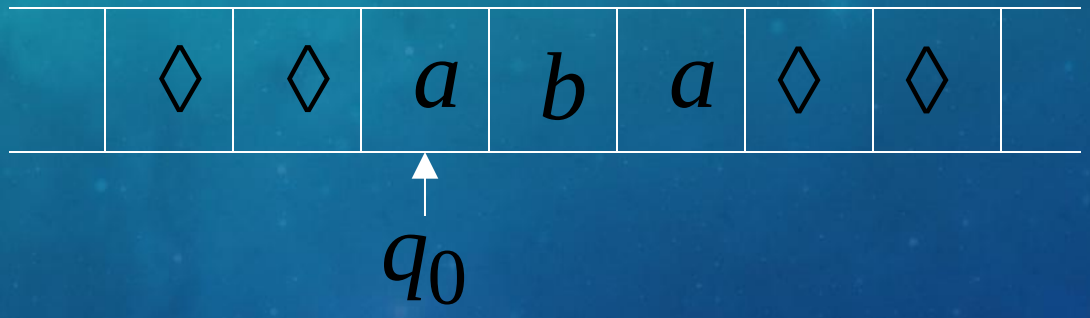
Time 2



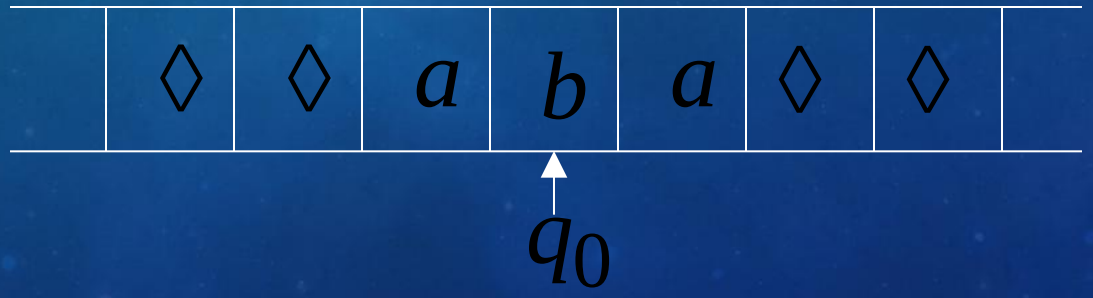
Time 3



Time 4



Time 5



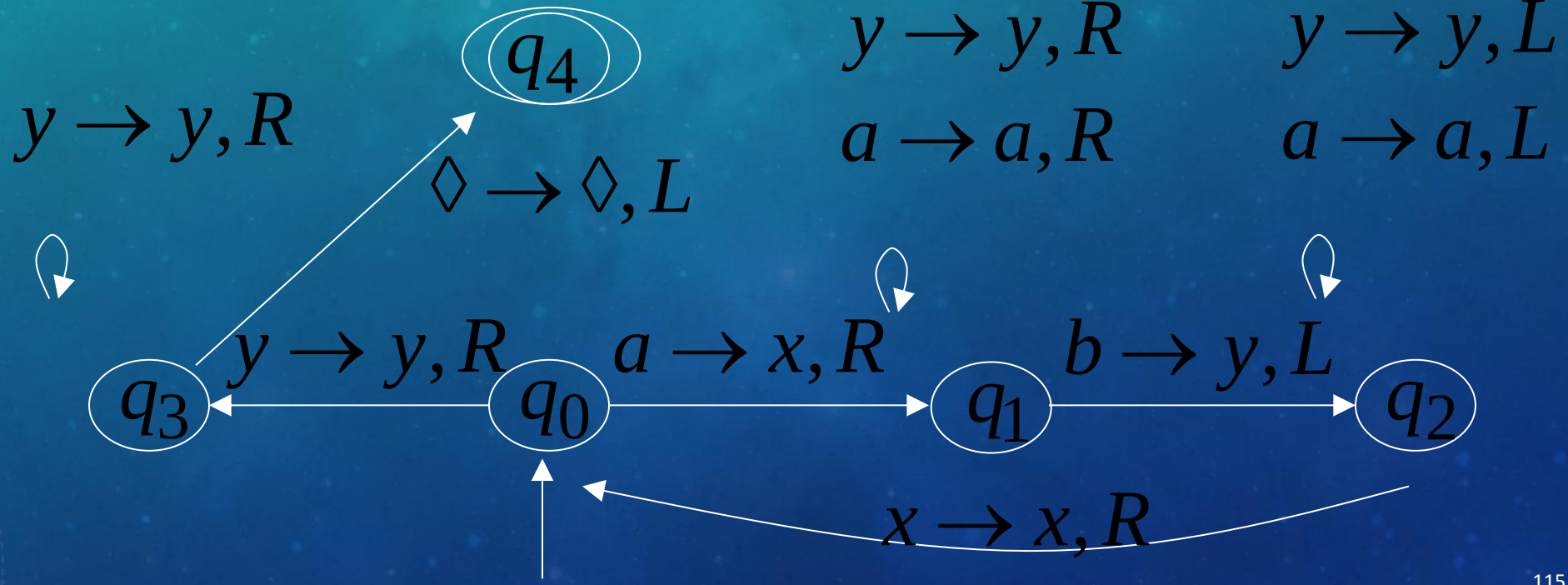
Infinite loop

Because of the infinite loop:

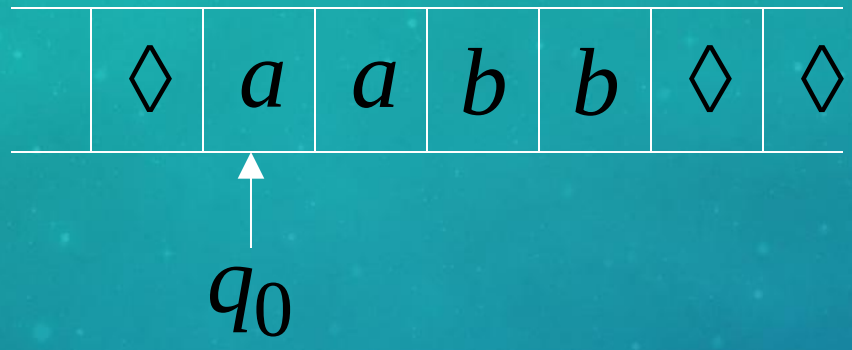
- The final state cannot be reached
- The machine never halts
- The input is not accepted

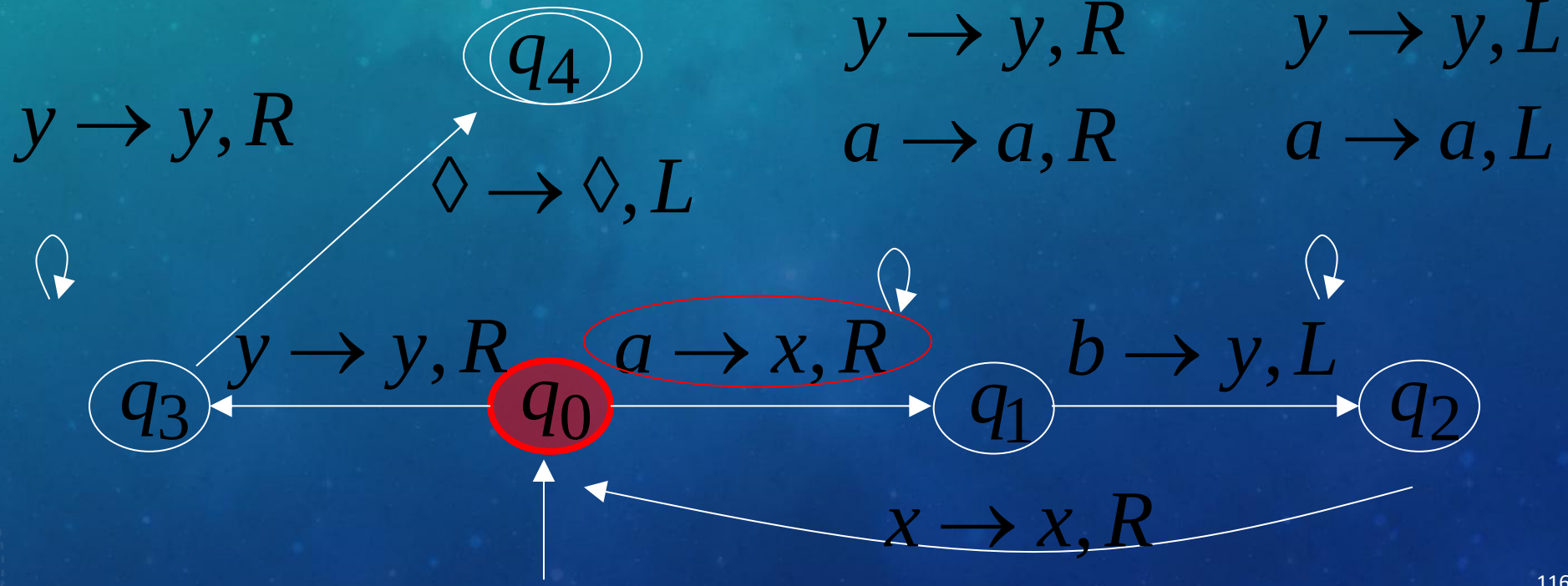
ANOTHER TURING MACHINE EXAMPLE

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

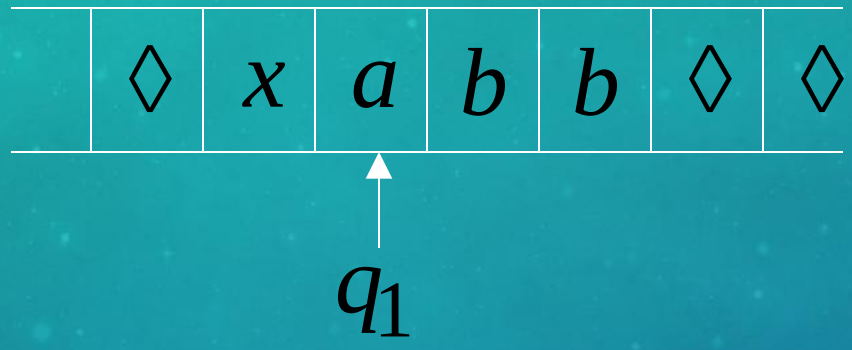


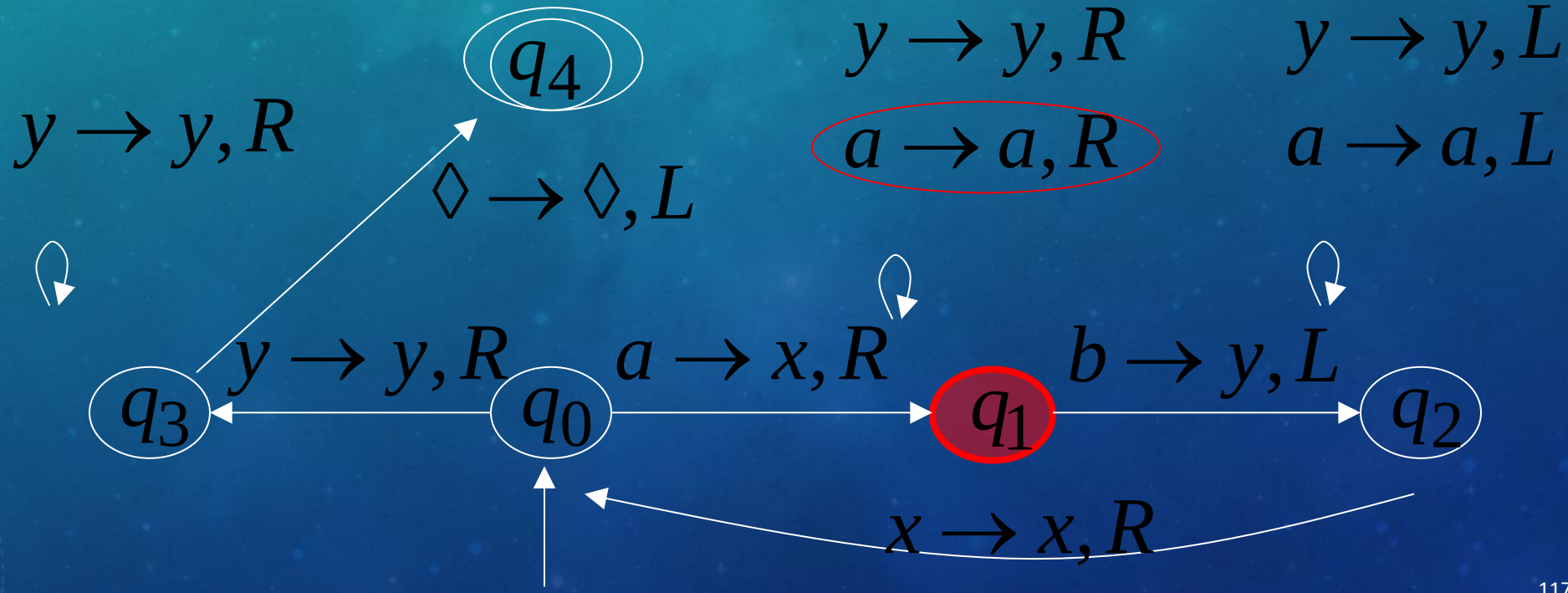
Time 0


◊ a a b b ◊ ◊
q_0

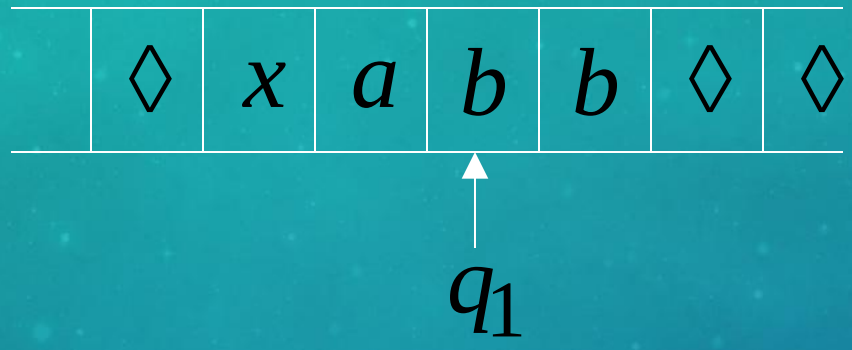

q_4
$y \to y, R$
$y \to y, R$
$a \to a, R$
$y \to y, L$
$a \to a, L$
$\Diamond \to \Diamond, L$
$y \to y, R$
$a \to x, R$
$b \to y, L$
q_3
q_0
q_1
q_2
$x \to x, R$

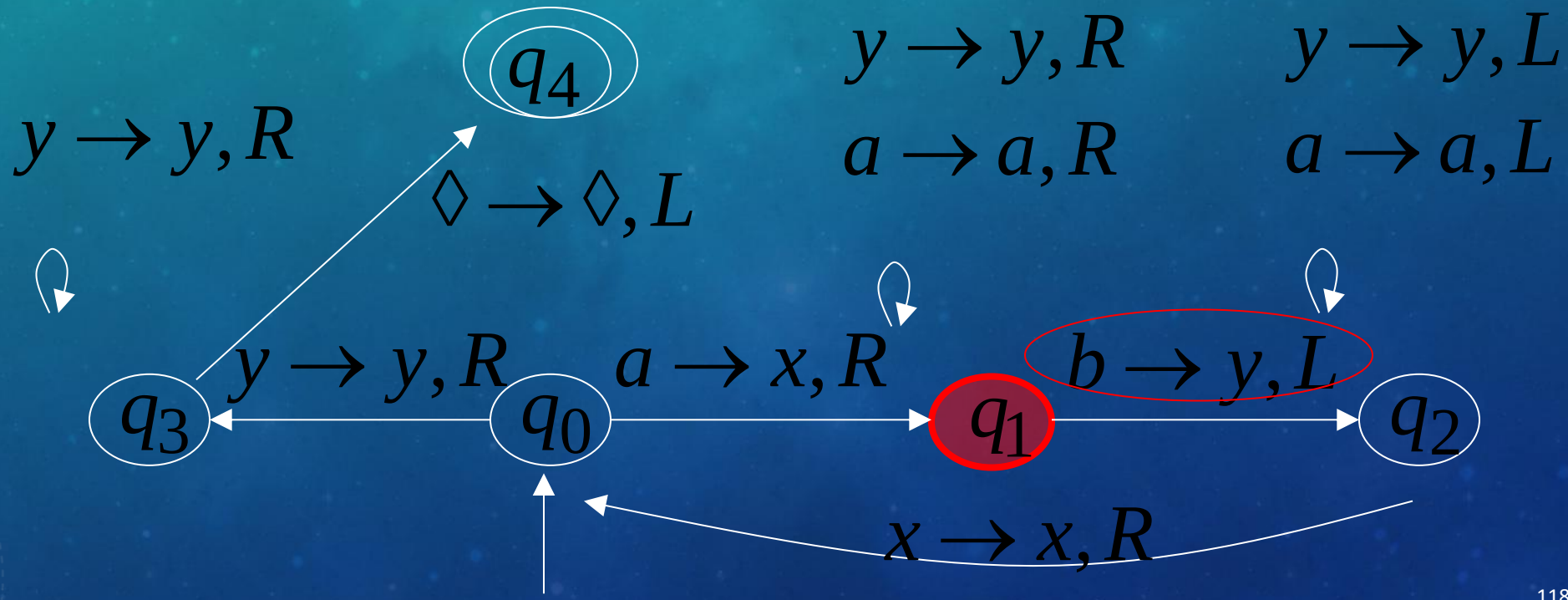
Time 1


◊ x a b b ◊ ◊
q_1

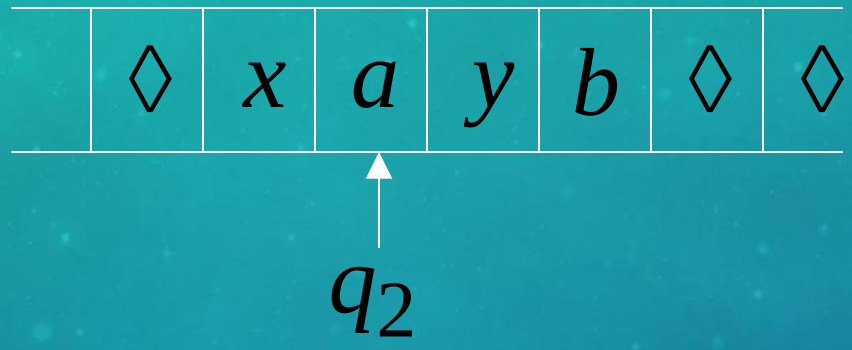

q_4
$y \to y, R$
$y \to y, L$
$a \to a, R$
$a \to a, L$
$y \to y, R$
$\lozenge \to \lozenge, L$
$y \to y, R$
$a \to x, R$
$b \to y, L$
q_3
q_0
q_1
q_2
$x \to x, R$

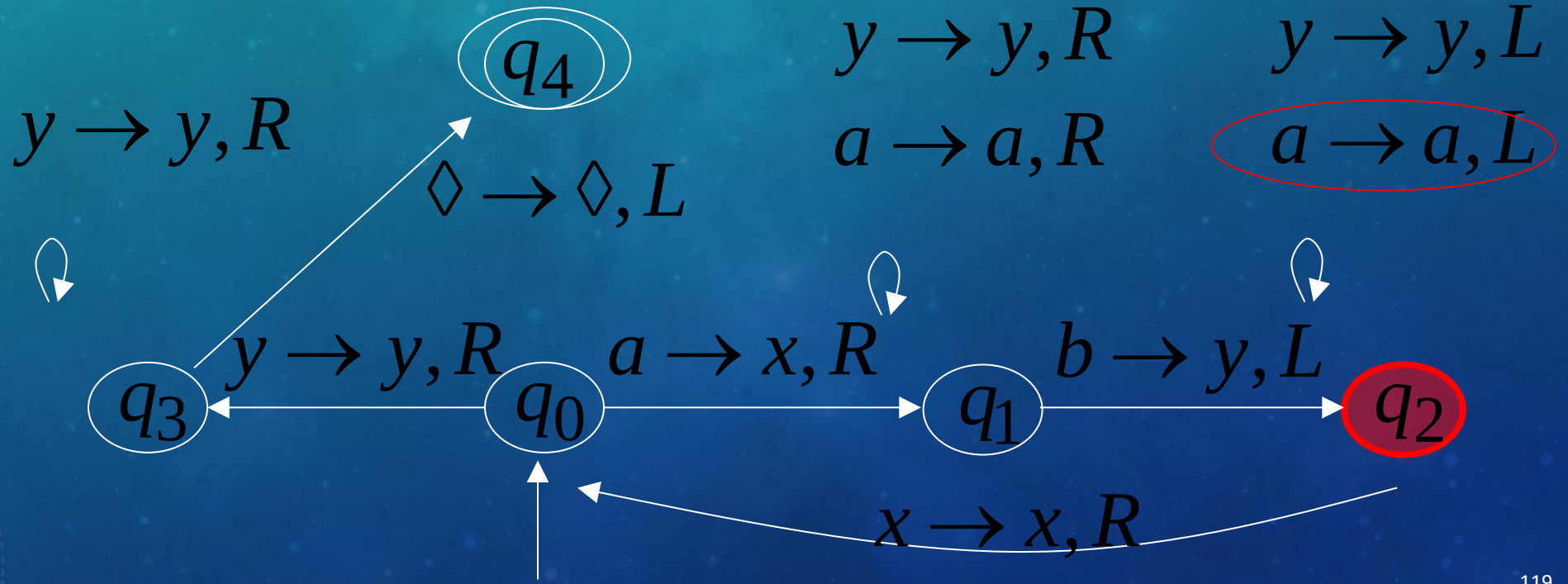
Time 2


◊ x a b b ◊ ◊
q_1

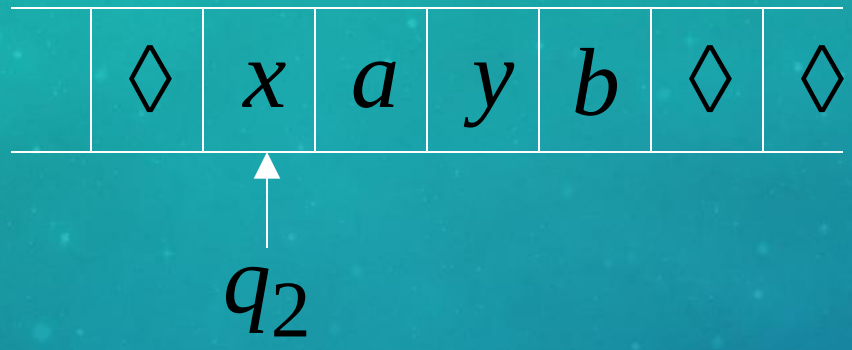

q_4
$y \to y, R$
$y \to y, R$
$a \to a, R$
$y \to y, L$
$a \to a, L$
$\Diamond \to \Diamond, L$
$y \to y, R$
$a \to x, R$
$b \to y, L$
q_3
q_0
q_1
q_2
$x \to x, R$

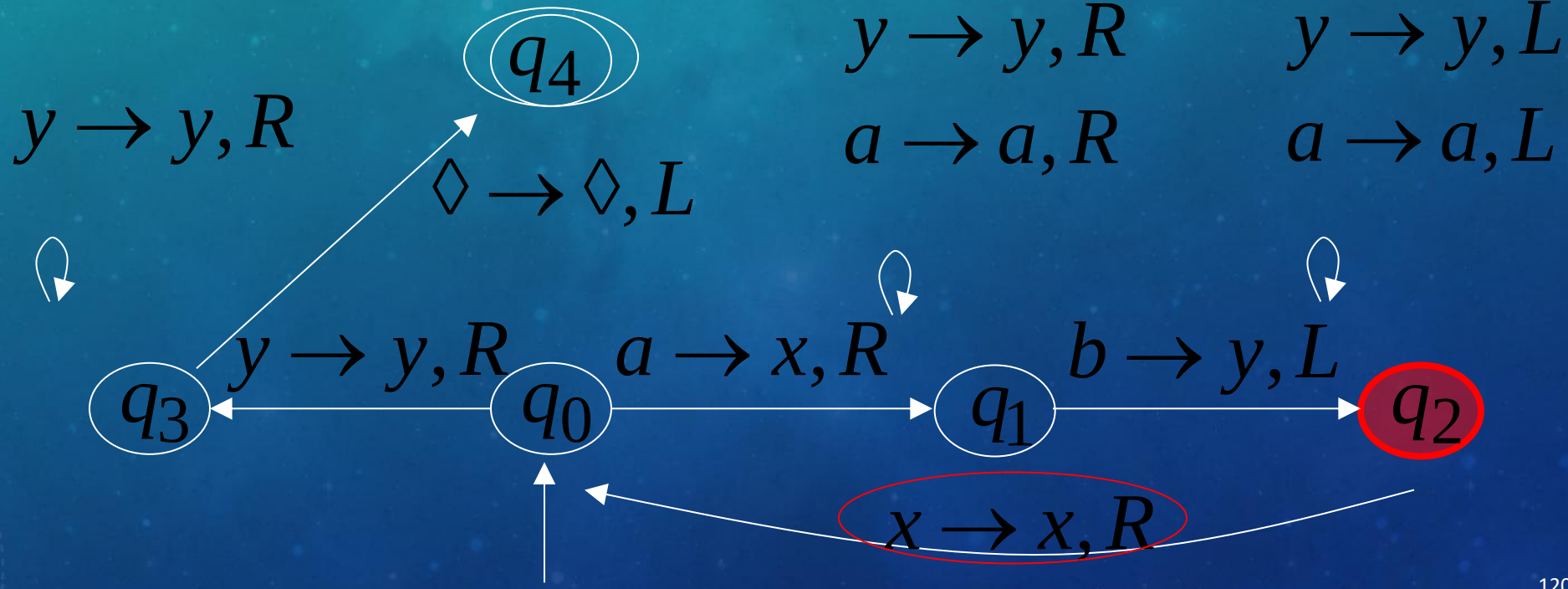
Time 3


◊ x a y b ◊ ◊
q_2

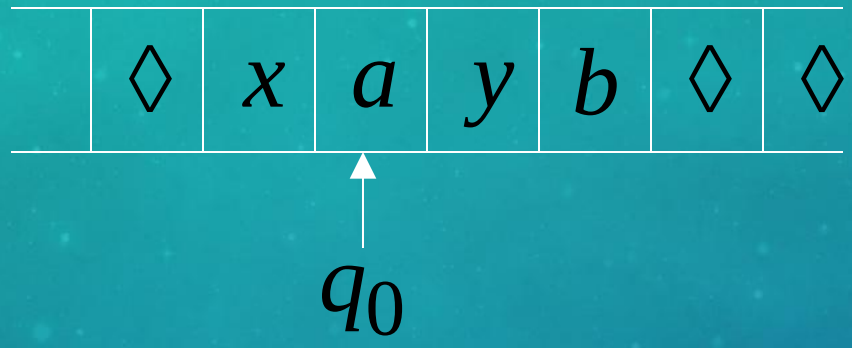

q_4
$y \to y, R$
$y \to y, R$
$a \to a, R$
$y \to y, L$
$a \to a, L$
$\Diamond \to \Diamond, L$
$y \to y, R$
$a \to x, R$
$b \to y, L$
q_3
q_0
q_1
q_2
$x \to x, R$

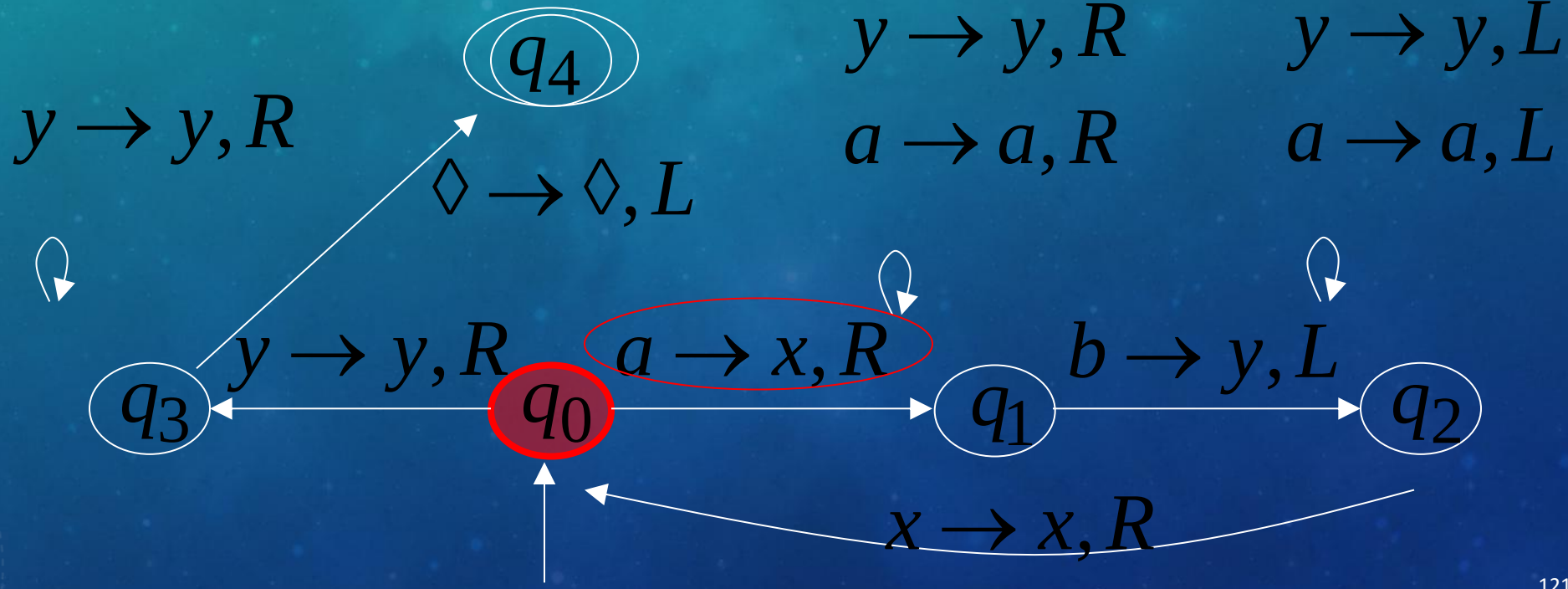
Time 4


◊ x a y b ◊ ◊
q_2

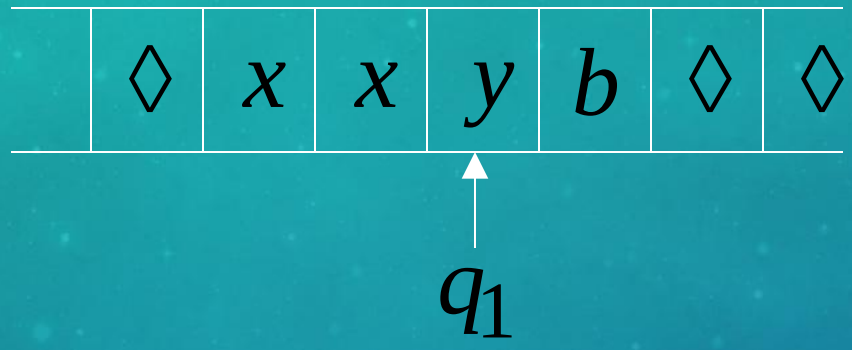

q_4
$y \to y, R$
$y \to y, R$
$a \to a, R$
$y \to y, L$
$a \to a, L$
$\Diamond \to \Diamond, L$
$y \to y, R$
$a \to x, R$
$b \to y, L$
q_3
q_0
q_1
q_2
$x \to x, R$

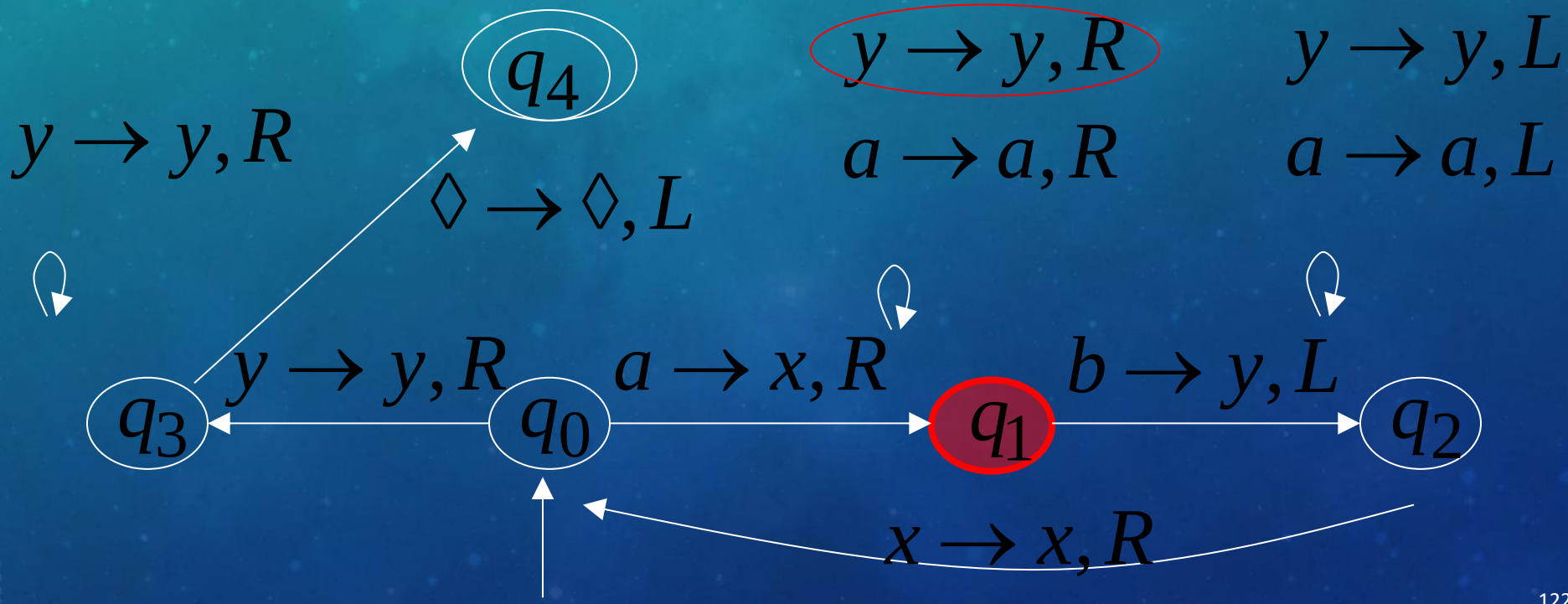
Time 5


◊ x a y b ◊ ◊
q_0

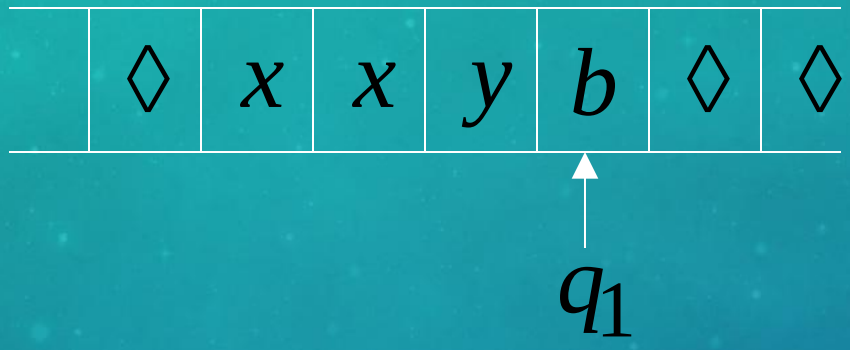

q_4
$y \to y, R$
$y \to y, R$
$a \to a, R$
$y \to y, L$
$a \to a, L$
$\Diamond \to \Diamond, L$
$y \to y, R$
$a \to x, R$
$b \to y, L$
q_3
q_0
q_1
q_2
$x \to x, R$

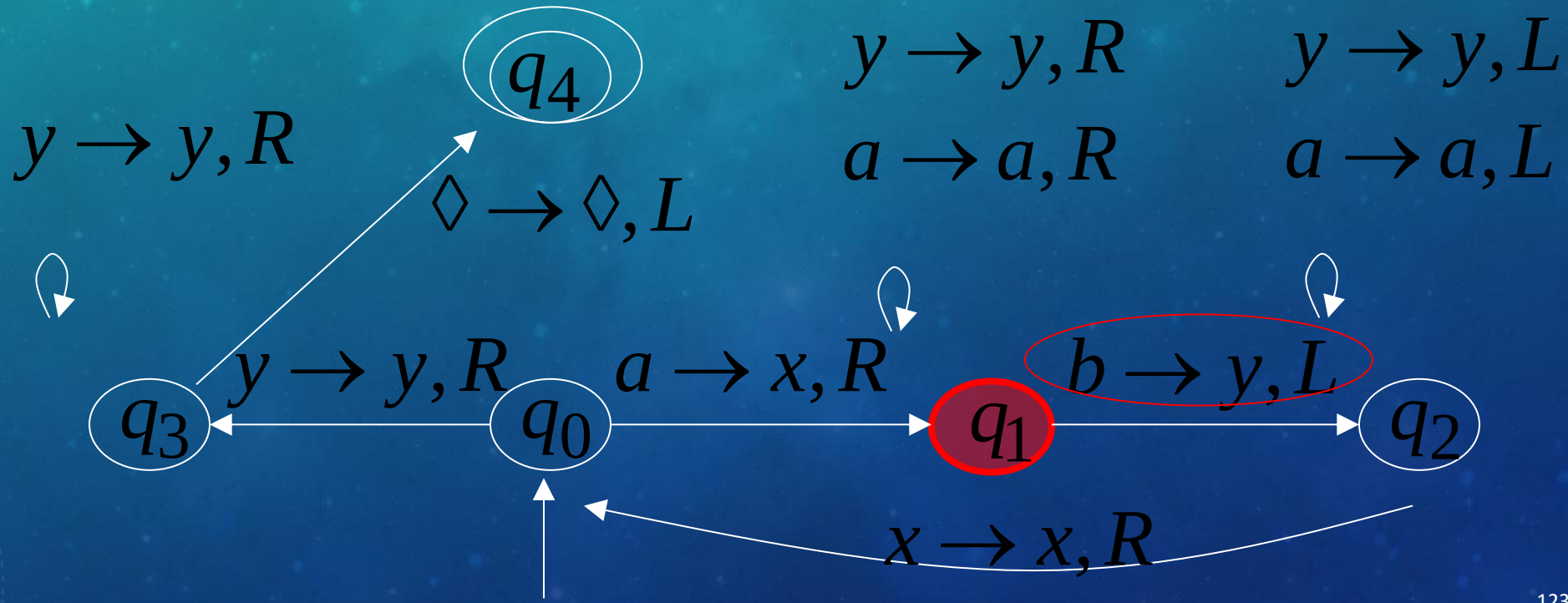
Time 6


◊ x x y b ◊ ◊
q_1

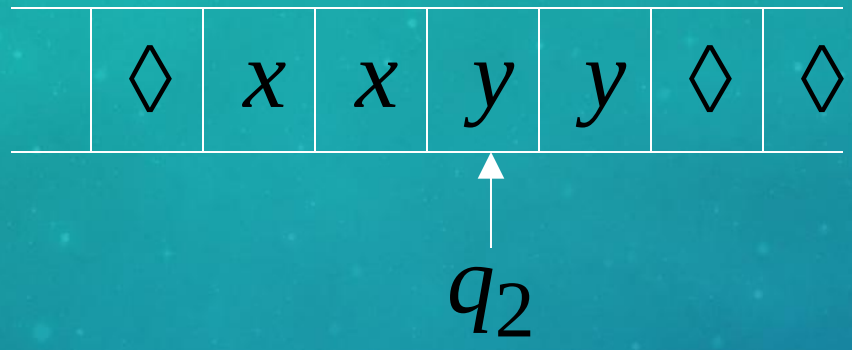

q_4
$y \to y, R$
$y \to y, R$
$a \to a, R$
$y \to y, L$
$a \to a, L$
$\Diamond \to \Diamond, L$
$y \to y, R$
$a \to x, R$
$b \to y, L$
q_3
q_0
q_1
q_2
$x \to x, R$

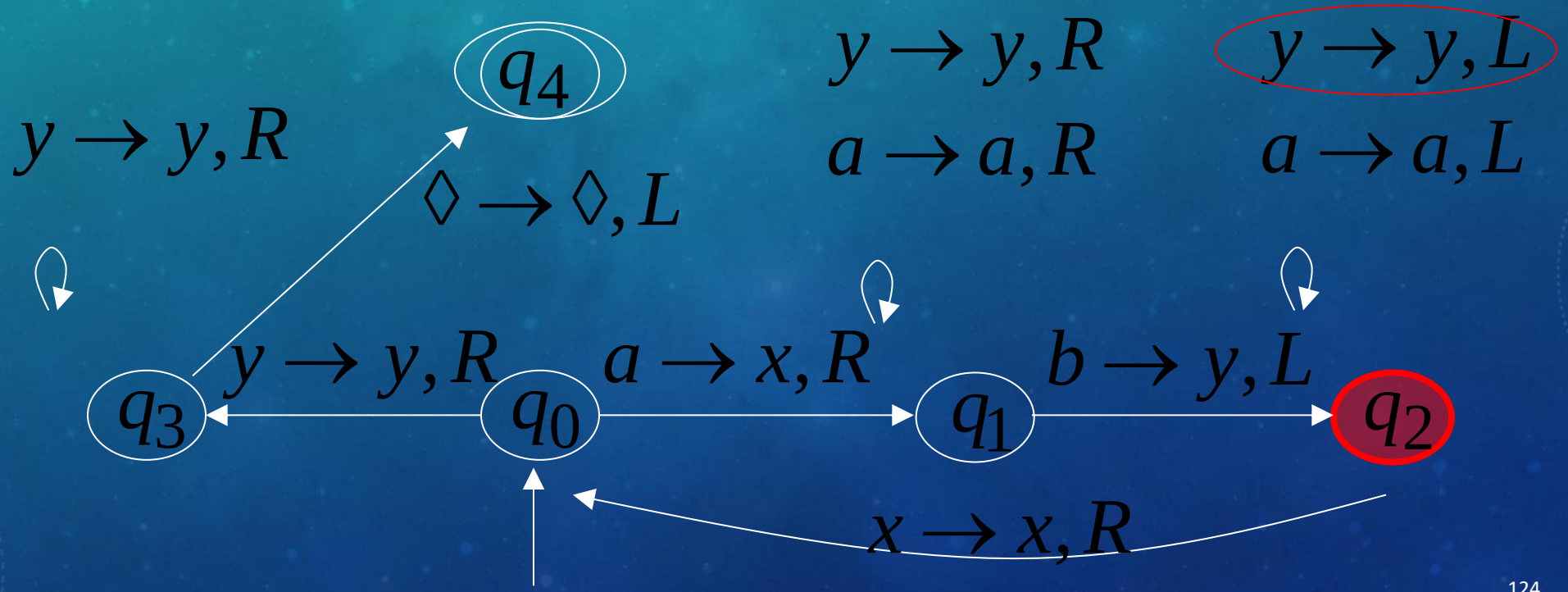
Time 7


◊ x x y b ◊ ◊
q_1

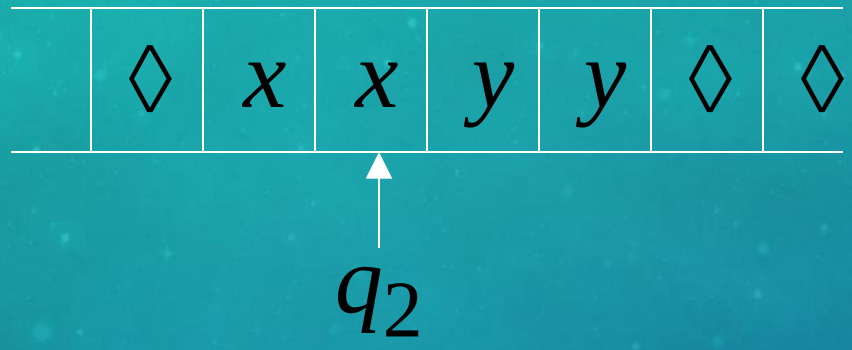

q_4
$y \to y, R$
$y \to y, R$
$a \to a, R$
$y \to y, L$
$a \to a, L$
$\Diamond \to \Diamond, L$
$y \to y, R$
$a \to x, R$
$b \to y, L$
q_3
q_0
q_1
q_2
$x \to x, R$

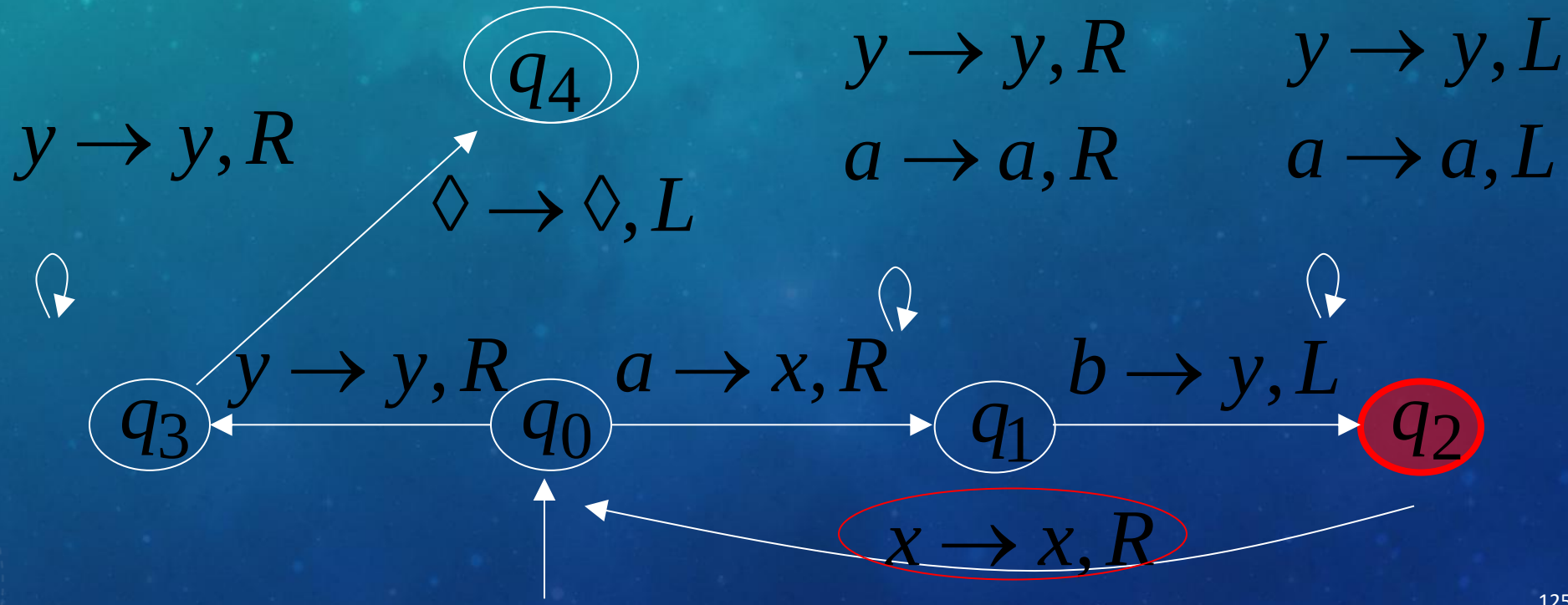
Time 8


◊ x x y y ◊ ◊
q_2

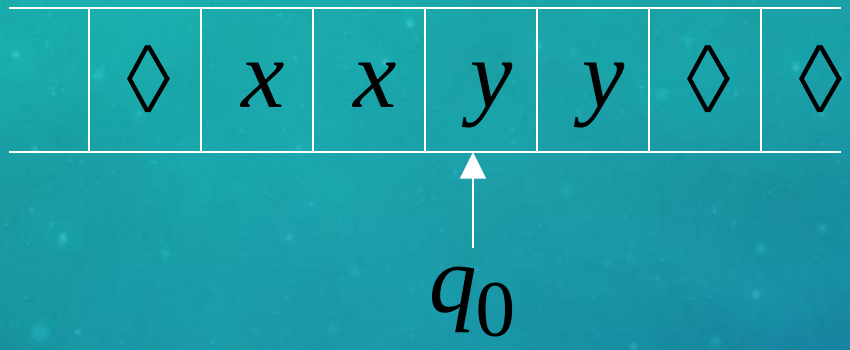

q_4
$y \to y, R$
$y \to y, R$
$a \to a, R$
$y \to y, L$
$a \to a, L$
$\Diamond \to \Diamond, L$
$y \to y, R$
$a \to x, R$
$b \to y, L$
q_3
q_0
q_1
q_2
$x \to x, R$

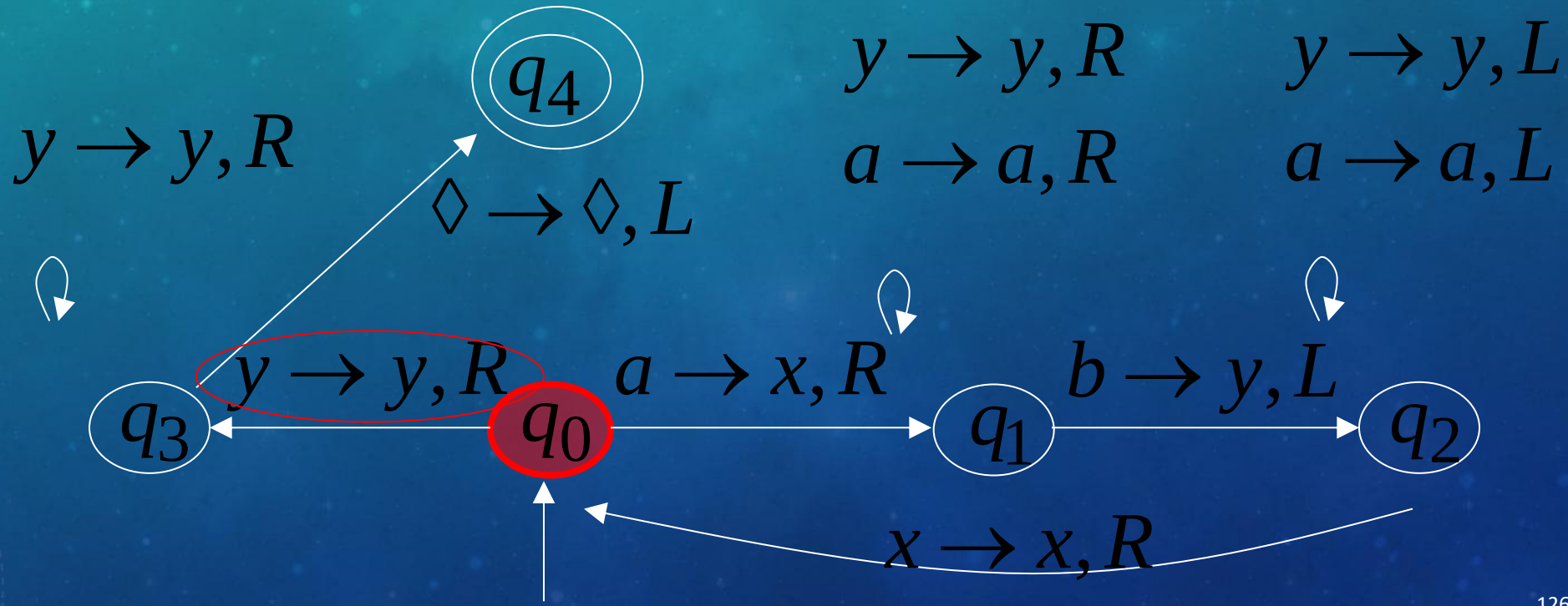
Time 9


◊ x x y y ◊ ◊
q_2

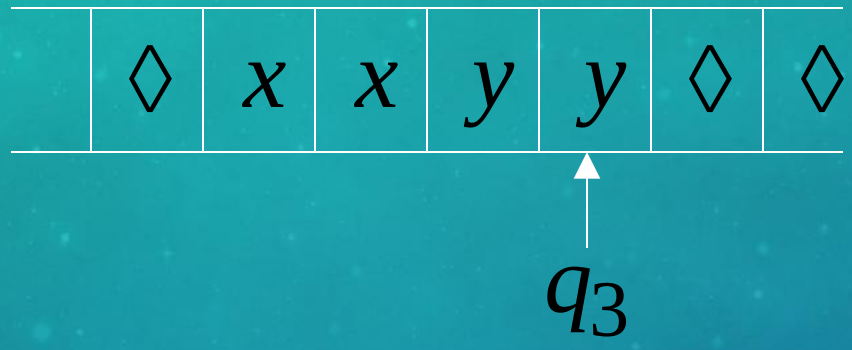

q_4
$y \to y, R$
$y \to y, R$
$a \to a, R$
$y \to y, L$
$a \to a, L$
$\Diamond \to \Diamond, L$
$y \to y, R$
$a \to x, R$
$b \to y, L$
q_3
q_0
q_1
q_2
$x \to x, R$

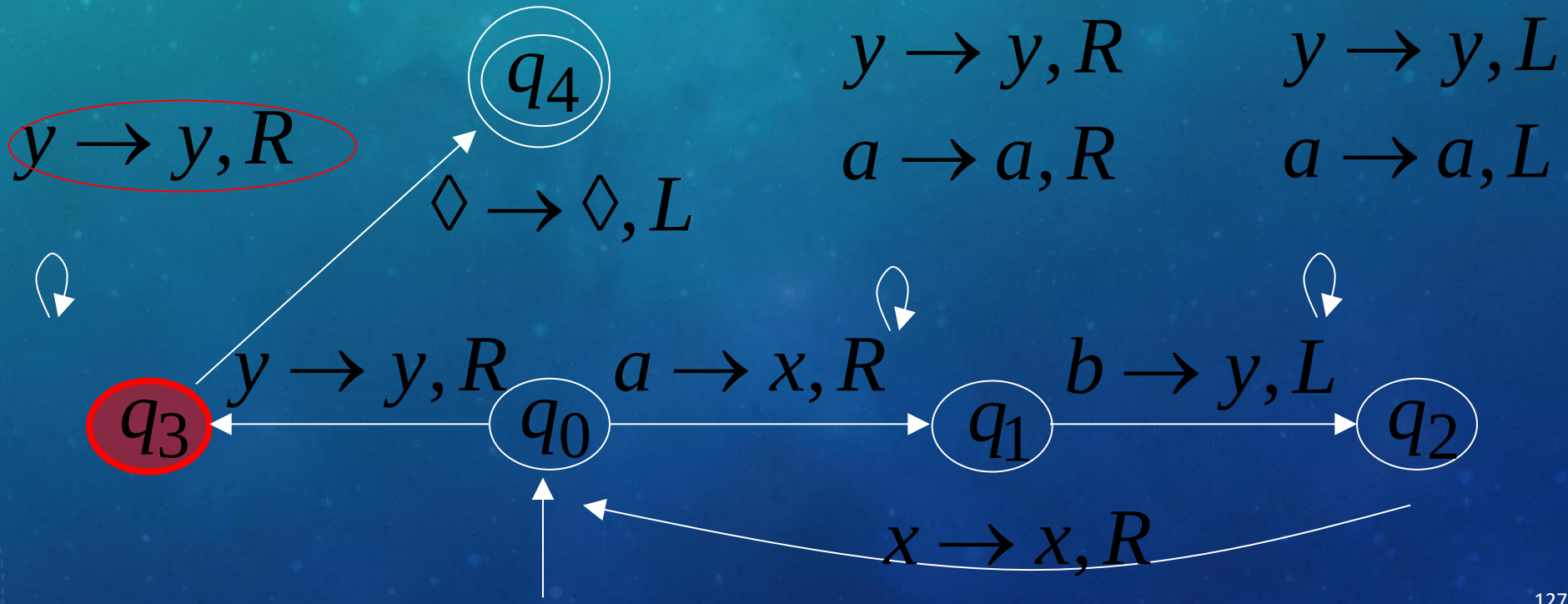
Time 10


◊ x x y y ◊ ◊
q_0

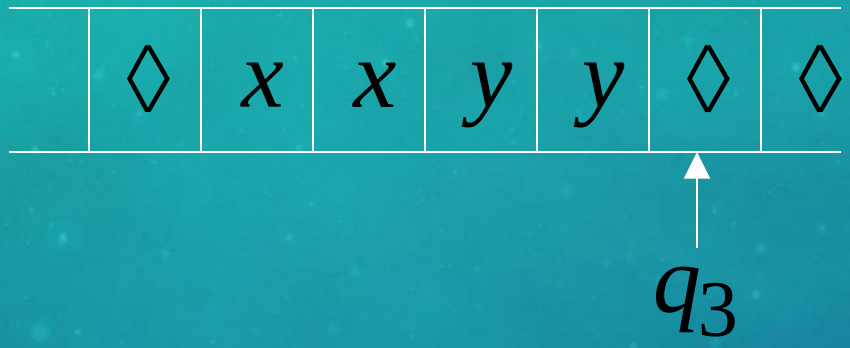

q_4
$y \to y, R$
$y \to y, R$
$a \to a, R$
$y \to y, L$
$a \to a, L$
$\Diamond \to \Diamond, L$
$y \to y, R$
$a \to x, R$
$b \to y, L$
q_3
q_0
q_1
q_2
$x \to x, R$

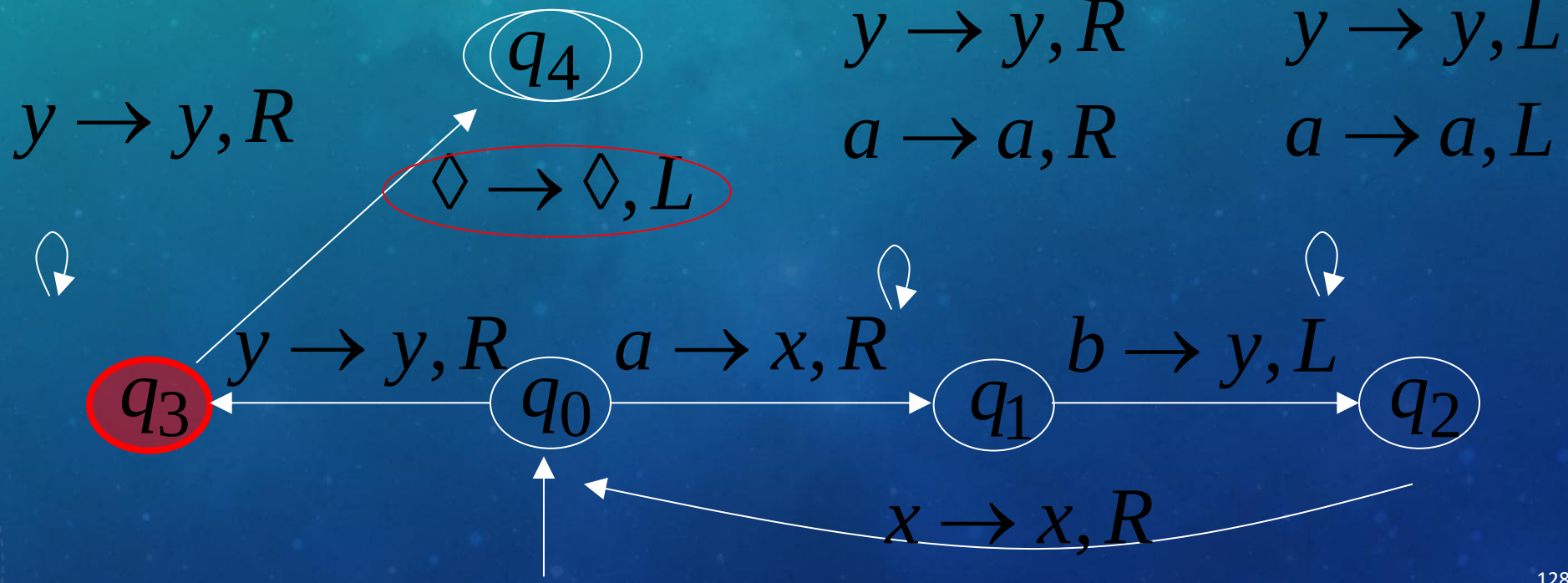
Time 11


◊ x x y y ◊ ◊
q_3

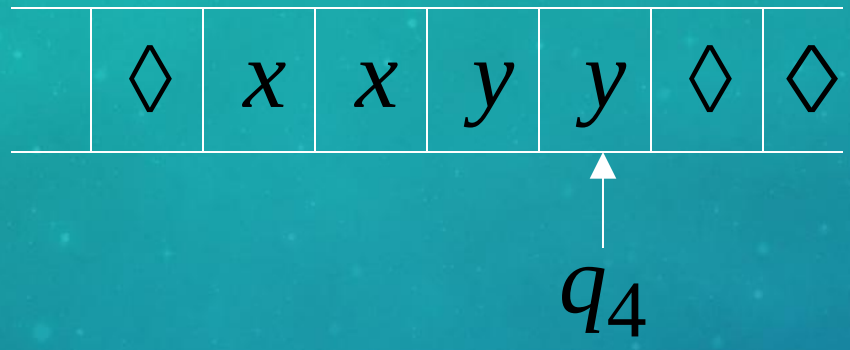

$y \to y, R$
q_4
$y \to y, R$
$a \to a, R$
$y \to y, L$
$a \to a, L$
$\Diamond \to \Diamond, L$
q_3
$y \to y, R$
q_0
$a \to x, R$
q_1
$b \to y, L$
q_2
$x \to x, R$

Time 12

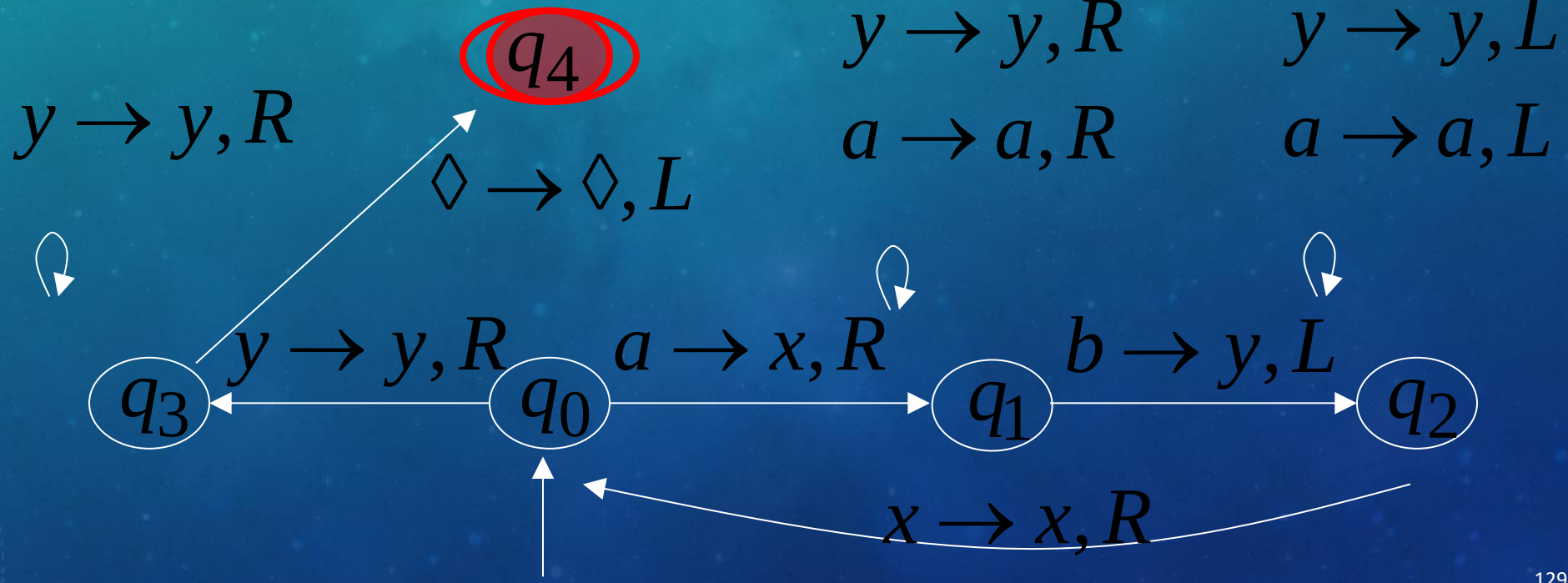

◊ x x y y ◊ ◊
q_3


$y \to y, R$
q_4
$y \to y, R$
$a \to a, R$
$y \to y, L$
$a \to a, L$
$\Diamond \to \Diamond, L$
$y \to y, R$
$a \to x, R$
$b \to y, L$
q_3
q_0
q_1
q_2
$x \to x, R$

Time 13



Halt & Accept



Week 9

Revision and Recap and Practical Applications of Theory

1. Finite Automata and Pattern Matching

Finite automata, both deterministic (DFA) and non-deterministic (NFA), are widely used in **pattern matching**, which powers tools like:

- **Text editors:** Identifying specific patterns in text.
- **Spam filters:** Detecting harmful phrases or URLs in emails.
- **Programming languages:** Lexical analyzers in compilers use finite automata to recognize tokens.

2. Regular Expressions in Search and Validation

Regular languages are implemented as **regular expressions**, critical for tasks like:

- **Web development:** Validating email addresses, phone numbers, and user input in forms.
- **Data extraction:** Extracting structured information, such as dates or IDs, from unstructured data.

3. Context-Free Grammars in Language Processing

Context-free grammars (CFGs) form the backbone of **parsers**, which analyze hierarchical structures:

- **Programming:** Compilers rely on CFGs to parse source code and ensure it follows language syntax.
- **Chatbots:** CFGs help in processing user inputs grammatically for better responses.

4. Turing Machines and Universal Computation

Turing machines model what computers can and cannot do, guiding the development of:

- **Algorithms:** Understanding the limits of computation helps optimize solutions.
- **Cryptography:** Encoding and decoding systems often rely on Turing-complete algorithms.

WEEK 10

Mid Term Examination

WEEK 11

Computability Theory

- A mathematical problem is ***computable*** if it can be solved in principle by a computing device.
- In the 1930's, well before there were computers, Gödel, Turing, and Church showed that not all mathematical problems are computable in a computing device.
- There is an extensive study and classification of
  - Which mathematical problems are ***computable***, and which are not.
  - Computable problems into computational ***complexity*** classes according to how much computation is needed to answer that instance, as a function of the size of the problem instance.
- Some common synonyms for "computable" are "solvable", "decidable", and "recursive".

Computability History

- David Hilbert's Tenth Problem in 1900 states that a given Diophantine equation (polynomial equation with integral coefficients) is **solvable** in rational integers using a finite number of operations.
- Hilbert came up with the term "entscheidungsproblem" (decision problems) which is the pre-version to the NP-problem that we currently know as SAT (satisfiability problem) in computing science, in 1928.
- In 1930s, various mathematicians – Alonzo Church, Kurt Gödel, Stephen Kleene, Markov, Emil Post, and Alan Turing, independently defined what it means to be **computable**.
- They defined Lambda calculus, Recursive functions, Formal systems, Markov algorithms, Post (abstract) machine, and Turing (abstract) machine models, which are equivalent to each other.
- In 1930 & 1931, Mr. Gödel gave his Completeness and Incompleteness theorem. A few years later, Church and Turing independently proved that the entscheidungsproblem is **unsolvable**.

DEFINITION: TURING MACHINE

- A Turing Machine is a 7-tuple
  $T = (Q, \Sigma, \Gamma, \delta, q_0, q_A, q_R)$, where:
  - Q is a finite set of states
  - Σ is the input alphabet, where $\square \notin \Sigma$
  - Γ is the tape alphabet, where $\square \in \Gamma$ and $\Sigma \subseteq \Gamma$
  - $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L,R\}$
  - $q_0 \in Q$ is the start state
  - $q_A \in Q$ is the accept state
  - $q_R \in Q$ is the reject state, and $q_R \neq q_A$

EMPTINESS TESTING

- E_{DFA} = {⟨A⟩ | A is a DFA with L(A) = ∅ }; This language concerns the behavior of the DFA A on all possible strings
- Proof for DFA-Emptiness:
  - Algorithm for E_{DFA} on input A=(Q,Σ,δ,q_0,F):
  - If A is not proper DFA then “reject”
  - Make set S with initially S={q_0}
  - Repeat |Q| times:
    - If S has an element in F then “reject”
    - Otherwise, add to set S, all the elements that can be δ-reached from S. i.e., “If $\exists q_i \in S$ and $\exists u \in \Sigma$ with $\delta(q_i,u)=q_j$, then q_j added to S”.
  - If final $S \cap F = \emptyset$ then “accept”

DFA-EQUIVALENCE

- A problem that deals with two DFAs:
  $EQ_{DFA} = \{\langle A,B\rangle \mid L(A) = L(B)\}$
- Theorem: EQ_{DFA} is TM-decidable.
- **Proof**: Look at the *symmetric difference* between the two languages: $(L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap L(B))$
- Note: "L(A)=L(B)" is equivalent with an empty symmetric difference between L(A) and L(B). This difference is expressed by standard DFA transformations: union, intersection, complement.

DFA-EQUIVALENCE

- Proof Theorem for $EQ_{DFA} = \{\langle A,B\rangle \mid L(A) = L(B)\}$
- **Algorithm on given** $\langle A,B\rangle$:
  - If A or B are not proper DFA then “reject”
  - Construct a third DFA C that accepts the language

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

  - Decide with the TM of the previous theorem whether or not $C \in E_{DFA}$
  - If $C \in E_{DFA}$ then “accept”;
    If $C \notin E_{DFA}$ then “reject”

- Similarly we can decide on the following languages:
- A_{DFA} = { (B,w) | B is a DFA that accepts string w }
  - Proof Idea: Simulate B on w.
- A_{NFA} = { (B,w) | B is an NFA that accepts string w }
  - Proof Idea:
    - Transform B into DFA C.
    - Simulate C on w.

DECIDABLE AND UNDECIDABLE PROBLEMS IN THEORY OF COMPUTATION

- In the Theory of Computation, problems are categorized as **decidable** or **undecidable**. Decidable problems have algorithms that provide correct solutions in finite time, while undecidable problems lack algorithms that can solve them for all inputs.

DECIDABLE

- A problem is decidable if there exists an algorithm that always provides a correct answer. For example, finding all prime numbers between 1000 and 2000 can be solved using a straightforward algorithm. In terms of a Turing machine, a problem is decidable if the machine halts on every input with a "yes" or "no" answer, making it Turing Decidable.

SEMI DECIDABLE PROBLEMS

- Semi-decidable problems are those for which a Turing machine halts on the input accepted by it but it can either halt or loop forever on the input which the Turing Machine rejects. Such problems are termed as Turing Recognizable problems.

UNDECIDABLE PROBLEMS

- **Undecidable problems** are those for which no algorithm can provide a correct answer in finite time. While they may be partially decidable, there will always be cases where a Turing machine enters an infinite loop without producing a result. For example, Fermat's Theorem is an undecidable problem because a Turing machine attempting to find a contradiction for the equation an+bn=cna^n + b^n = c^nan+bn=cn (where n>2n > 2n>2) might run indefinitely without reaching a conclusion.

COMPARISON: DECIDABLE VS. UNDECIDABLE PROBLEMS

Aspect	Decidable Problems	Undecidable Problems
Definition	Problems that can be solved by an algorithm that always gives a correct answer in a finite time.	Problems where no algorithm can give a solution for all possible cases.
Solvability	Always solvable using a step-by-step process (algorithm).	Cannot be solved for all inputs using a single algorithm.
Algorithm	There is an algorithm that works for every input and always finishes with an answer.	No algorithm can solve the problem for every input.
Halting	The algorithm stops (halts) and gives an answer for every input.	The algorithm might never stop for some inputs, or no algorithm exists.
Examples	Problems like checking if a number is even or odd, or if a string belongs to a regular language (like finding a match in a search).	Examples include the Halting Problem, where you can't always tell if a program will finish running or run forever.
Decision Procedure	There's a clear method to always reach a correct conclusion.	No guaranteed method exists to solve the problem in every case.
Complexity	May be complex but can always be computed.	Too complex to compute in general, and no universal solution exists.
Applications	Useful in practical computing tasks like compiling code or searching for text patterns.	Helps understand the limits of what computers can do, showing what problems are beyond computation.

WEEK 12

- Complexity Theory
  - Easy problems (sort a million items in a few seconds)
  - Hard problems (schedule a thousand classes in a hundred years)
  - What makes some problems hard and others easy (computationally) and how do we make hard problems easier?
  - Complexity Theory addresses these questions

Computability Theory

- In the first half of the 20th century, mathematicians such as Kurt Gödel, Alan Turing, and Alonzo Church discovered that certain basic problems cannot be solved by computers
  - ➢ determine whether a mathematical statement is true or false
- Complexity Theory: classify problems as easy ones and hard ones
- Computability Theory: classify problems are solvable and not solvable

AUTOMATA THEORY

- Deals with the definitions and properties of mathematical models of computation
- Finite automaton (used in text processing, compilers, hardware design)
- Context-free grammar (used in programming languages and artificial intelligence)

COMPLEXITY ANALYSIS

Why do we write programs?

- to perform some specific tasks
- to solve some specific problems
- We will focus on “solving problems”
- What is a “problem”?
- We can view a problem as a mapping of “inputs” to “outputs

COMPLEXITY CLASSES
P AND NP

RECAP: DECISION PROBLEMS

- In the initial part of this course, we'll focus primarily on decision problems.
- Decision problems can be naturally identified with boolean functions, i.e. functions from $\{0,1\}^*$ to $\{0,1\}$.
- Boolean functions can be naturally identified with sets of $\{0,1\}$ strings, also called languages.

RECAP: DECISION PROBLEMS

Decision problems ⟷ Boolean functions ⟷ Languages

- Definition. We say a TM M *decides* a language $L \subseteq \{0,1\}^*$ if M computes f_L, where $f_L(x) = 1$ if and only if $x \in L$.

RECAP: COMPLEXITY CLASS P

- Let T: N N be some function.
- Definition: A language L is in DTIME(T(n)) if there's a TM that decides L in time O(T(n)).
- Defintion: Class P = $\bigcup_{c>0}$ DTIME (n^c).

Deterministic polynomial-time

COMPLEXITY CLASS P : EXAMPLES

- Cycle detection
- Solvabililty of a system of linear equations
- Perfect matching
- Primality testing *(AKS test 2002)*
  - Check if a number is prime

POLYNOMIAL TIME TURING MACHINES

- Definition. A TM M is a *polynimial time* TM if there's a polynomial function $q: \mathbb{N} \to \mathbb{N}$ such that for every input $x \in \{0,1\}^*$, M halts within $q(|x|)$ steps.

  Polynomial function. $q(n) = n^c$ for some constant c

CLASS (FUNCTIONAL) P

- What if a problem is not a decision problem? Like the task of adding two integers.

CLASS (FUNCTIONAL) P

- What if a problem is not a decision problem? Like the task of adding two integers.
- One way is to focus on the i-th bit of the output and make it a decision problem.

(Is the i-th bit, on input x, 1?)

CLASS (FUNCTIONAL) P

- What if a problem is not a decision problem? Like the task of adding two integers.
- One way is to focus on the i-th bit of the output and make it a decision problem.

- Alternatively, we define a class called functional P.

CLASS (FUNCTIONAL) P

- What if a problem is not a decision problem? Like the task of adding two integers.
- One way is to focus on the i-th bit of the output and make it a decision problem.

- We say that a problem or a function $f: \{0,1\}^* \longrightarrow \{0,1\}^*$ is in FP (functional P) if there's a polynomial-time TM that computes f.

COMPLEXITY CLASS FP : EXAMPLES

- Greatest Common Divisor *(Euclid ~300 BC)*
  - Given two integers a and b, find their gcd.

COMPLEXITY CLASS FP : EXAMPLES

- Greatest Common Divisor
- Counting paths in a DAG *(homework)*
  - Find the number of paths between two vertices in a directed acyclic graph.

COMPLEXITY CLASS FP : EXAMPLES

- Greatest Common Divisor
- Counting paths in a DAG
- Maximum matching *(Edmonds 1965)*
  - Find a maximum matching in a given graph

COMPLEXITY CLASS NP

- Solving a problem is generally *harder* than verifying a given solution to the problem.

COMPLEXITY CLASS NP

- Solving a problem is generally *harder* than verifying a given solution to the problem.
- Class NP captures the set of decision problems whose solutions are *efficiently verifiable*.

COMPLEXITY CLASS NP

- Solving a problem is generally *harder* than verifying a given solution to the problem.
- Class NP captures the set of decision problems whose solutions are *efficiently verifiable*.

Nondeterministic polynomial-time

COMPLEXITY CLASS NP

- Definition. A language $L \subseteq \{0,1\}^*$ is in NP if there's a polynomial function $p: \mathbb{N} \to \mathbb{N}$ and a polynomial time TM M (called the *verifier*) such that for every x,

$$x \in L \iff \exists u \in \{0,1\}^{p(|x|)} \quad \text{s.t.} \quad M(x, u) = 1$$

COMPLEXITY CLASS NP

- Definition. A language $L \subseteq \{0,1\}^*$ is in NP if there's a polynomial function $p: \mathbb{N} \longrightarrow \mathbb{N}$ and a polynomial time TM M (called the *verifier*) such that for every x,

$$x \in L \iff \exists u \in \{0,1\}^{p(|x|)} \text{ s.t. } M(x, u) = 1$$

u is called a *certificate or witness* for x (w.r.t L and M) if $x \in L$

COMPLEXITY CLASS NP

- Definition. A language $L \subseteq \{0,1\}^*$ is in NP if there's a polynomial function $p: \mathbb{N} \to \mathbb{N}$ and a polynomial time TM M (called the *verifier*) such that for every x,

$$x \in L \iff \exists u \in \{0,1\}^{p(|x|)} \quad \text{s.t.} \quad M(x, u) = 1$$

- It follows that verifier M cannot be fooled!

COMPLEXITY CLASS NP

- Definition. A language $L \subseteq \{0,1\}^*$ is in NP if there's a polynomial function $p: \mathbb{N} \to \mathbb{N}$ and a polynomial time TM M (called the *verifier*) such that for every x,

$$x \in L \iff \exists u \in \{0,1\}^{p(|x|)} \quad \text{s.t.} \quad M(x, u) = 1$$

- Class NP contains those problems (languages) which have such efficient verifiers.

CLASS NP : EXAMPLES

- Vertex cover
  - Given a graph G and an integer k, check if G has a vertex cover of size k.

CLASS NP : EXAMPLES

- Vertex cover
- 0/1 integer programming
  - Given a system of linear (in)equalities with integer coefficients, check if there's a 0-1 assignment to the variables that satisfy all the (in)equalities.

CLASS NP : EXAMPLES

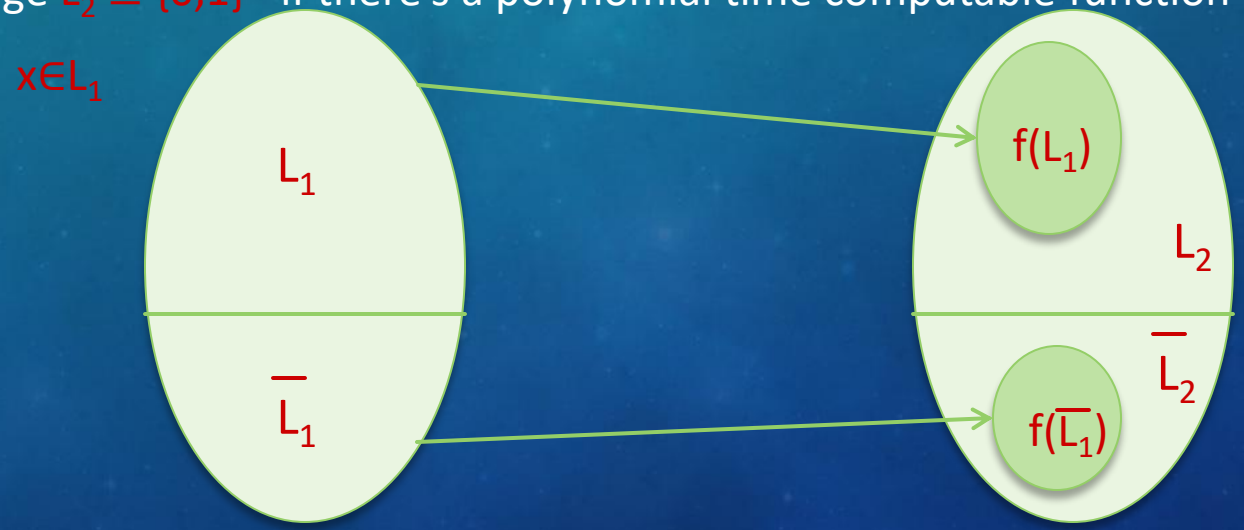
- Vertex cover
- 0/1 integer programming
- Integer factoring
  - Given 2 numbers n and U, check if n has a nontrivial factor less than equal to U.

CLASS NP : EXAMPLES

- Vertex cover
- 0/1 integer programming
- Integer factoring
- Graph isomorphism
  - Given 2 graphs, check if they are isomorphic

POLYNOMIAL TIME REDUCTION

- Definition. We say a language $L_1 \subseteq \{0,1\}^*$ is *polynomial time (Karp) reducible* to a language $L_2 \subseteq \{0,1\}^*$ if there's a polynomial time computable function f s.t.



POLYNOMIAL TIME REDUCTION

- Definition. We say a language $L_1 \subseteq \{0,1\}^*$ is *polynomial time (Karp) reducible* to a language $L_2 \subseteq \{0,1\}^*$ if there's a polynomial time computable function f s.t.

  $x \in L_1 \quad f(x) \in L_2$ ⟺

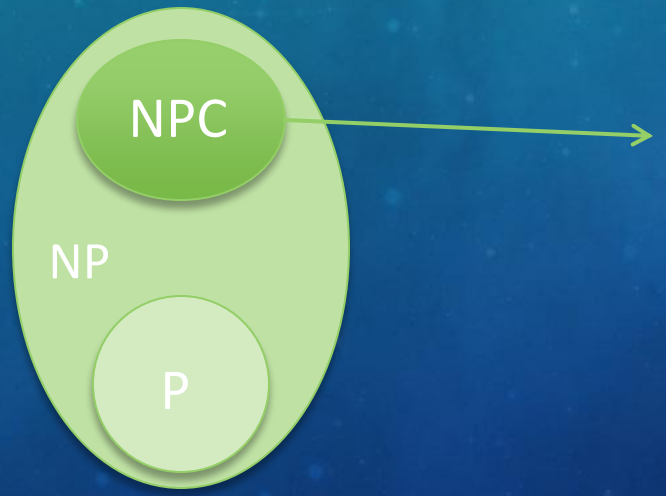
- Notation. $L_1 \leq_p L_2$

- Observe. If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$ then $L_1 \leq_p L_3$.

NP-COMPLETENESS

- Definition. A language L' is *NP-hard* if for every L in NP, $L \leq_p L'$. Further, L' is *NP-complete* if L' is in NP and is NP-hard.

- Observe. If L' is NP-hard and L' is in P then P = NP. If L' is NP-complete then L' in P if and only if P = NP.



Hardest problems inside NP in the sense that if one NPC problem is in P then all problems in NP is in P.

NP-COMPLETENESS

- Definition. A language L' is *NP-hard* if for every L in NP, $L \leq_p L'$. Further, L' is *NP-complete* if L' is in NP and is NP-hard.
- Observe. If L' is NP-hard and L' is in P then P = NP. If L' is NP-complete then L' in P if and only if P = NP.
- Exercise. Let $L_1 \subseteq \{0,1\}^*$ be any language and L_2 be a language in NP. If $L_1 \leq_p L_2$ then L_1 is also in NP.

CLASS P AND NP : EXAMPLES

- Vertex cover (NP-complete)
- 0/1 integer programming (NP-complete)
- Integer factoring (unlikely to be NP-complete)
- Graph isomorphism (Quasi-P)
- Primality testing (P)
- Linear programming (P)

A NATURAL NP-COMPLETE PROBLEM

- Definition. A *boolean formula* on variables $x_1, \ldots, x_n$ consists of AND, OR and NOT operations.

  e.g. $\phi = (x_1 \vee x_2) \wedge (x_3 \vee \neg x_2)$

- Definition. A boolean formula ϕ is *satisfiable* if there's a $\{0,1\}$-assignment to its variables that makes ϕ evaluate to 1.

A NATURAL NP-COMPLETE PROBLEM

- Definition. A boolean formula is in *Conjunctive Normal Form* (*CNF*) if it is an AND of OR of literals.

e.g. $\phi = (x_1 \vee x_2) \wedge (x_3 \vee \neg x_2)$

A NATURAL NP-COMPLETE PROBLEM

- Definition. A boolean formula is in *Conjunctive Normal Form* (*CNF*) if it is an AND of OR of literals.

e.g. $\phi = (x_1 \vee x_2) \wedge (x_3 \vee \neg x_2)$

literals

A NATURAL NP-COMPLETE PROBLEM

- Definition. A boolean formula is in *Conjunctive Normal Form* (*CNF*) if it is an AND of OR of literals.

  e.g. $\phi = (x_1 \vee x_2) \wedge (x_3 \vee \neg x_2)$

- Definition. Let SAT be the language consisting of all *satisfiable CNF formulae*.

A NATURAL NP-COMPLETE PROBLEM

- Definition. A boolean formula is in *Conjunctive Normal Form (CNF)* if it is an AND of OR of literals.

  e.g. $\phi = (x_1 \vee x_2) \wedge (x_3 \vee \neg x_2)$

- Definition. Let SAT be the language consisting of all *satisfiable CNF formulae.*

- Theorem. *(Cook-Levin)* SAT is NP-complete.

A NATURAL NP-COMPLETE PROBLEM

- Definition. A boolean formula is in *Conjunctive Normal Form (CNF)* if it is an AND of OR of literals.

  e.g. $\phi = (x_1 \vee x_2) \wedge (x_3 \vee \neg x_2)$

- Definition. Let SAT be the language consisting of all *satisfiable CNF formulae.*

- Theorem. *(Cook-Levin)* SAT is NP-complete.

Easy to see that SAT is in NP.

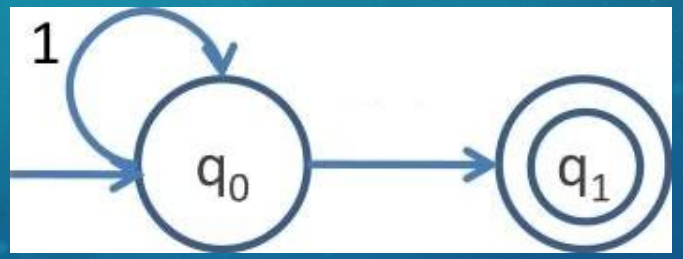
Need to show that SAT is NP-hard.

WEEK 13

NFA, - NFA - DFA EQUIVALENCE

WHAT IS AN NFA

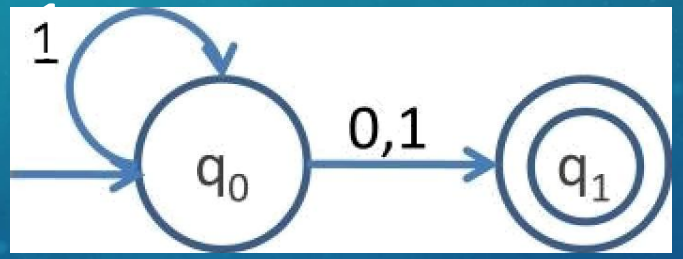
- An NFA is an automaton that its states might have none, one or more outgoing arrows under a specific symbol.



- From q_0 with 1 we can be either in q_0 or q_1.
- No outgoing arrows under 0 or 1 from q_1.

WHAT IS AN NFA

- An NFA is an automaton that its states might have none, one or more outgoing arrows under a specific symbol.



- A DFA is by definition an NFA (each state has exactly one outgoing arrow under each symbol).

WHAT IS AN NFA

- An NFA, is an NFA that might have c-moves. In an c-move we can transport from one state to the other without having any symbols.

- From q_0 with 1 we can be either in q_0 or q_2.

WHAT IS AN NFA

- An NFA, is an NFA that might have c-moves. In an c-move we can transport from one state to the other without having any symbols.

- An NFA is by definition an NFA, (but with no c-moves).

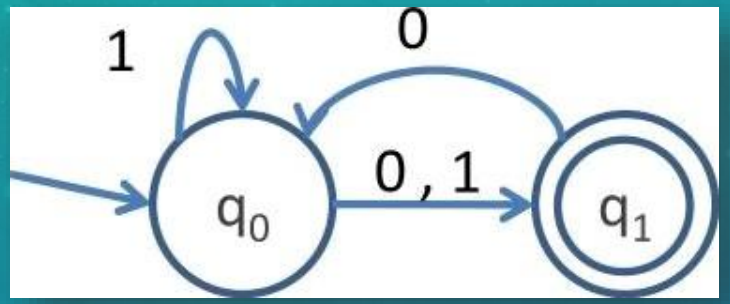
NFA – NFA, COMPUTATION

- An Non-Deterministic FA can have choices. If there are two possible transitions under a specific symbol, it can choose either of them and follow it.
- Given some input string, there might be more than one paths to follow. Some of them might fail but, in order to accept, it suffices to find one that succeeds.

NFA —NFA, COMPUTATION

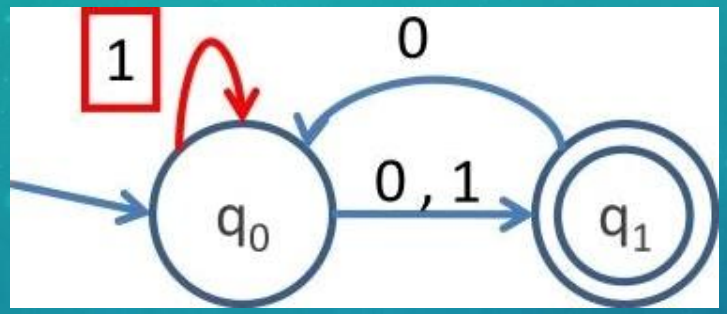
An Non-Deterministic Finite Automaton accepts an input string s if there exists a path following transitions under the symbols of s consecutively, that leads to an accepting state.

EXAMPLE



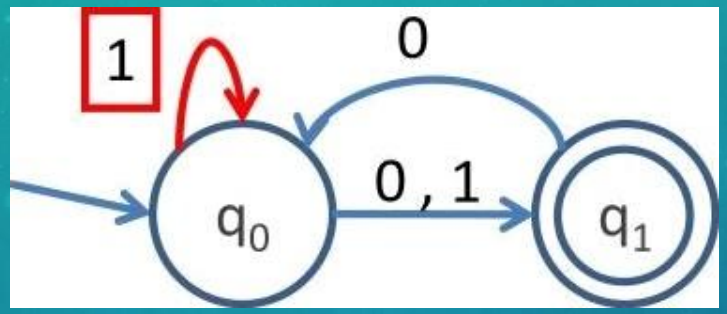
- This automaton accepts the string 1110 because there is a path under 1110 that takes us to an accepting state
- (the path $q_0 4^1 q_0 4^1 q_0 4^1 q_0 4^0 q_1$).

EXAMPLE



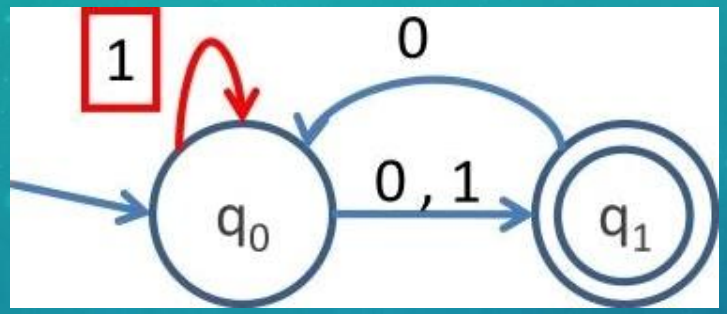
- This automaton accepts the string 1110 because there is a path under 1110 that takes us to an accepting state
- (the path $q_0 4^1 q_0 4^1 q_0 4^1 q_0 4^0 q_1$).

EXAMPLE



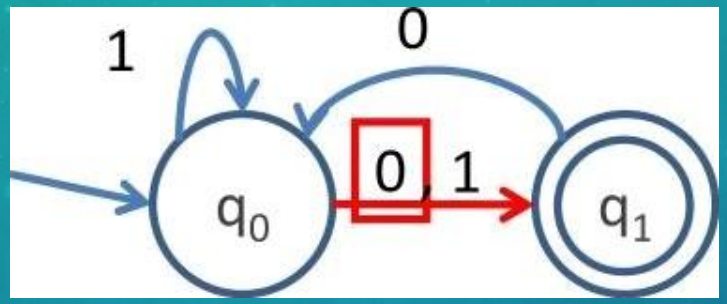
- This automaton accepts the string 1110 because there is a path under 1110 that takes us to an accepting state
- (the path $q_0 4^1 q_0 4^1 q_0 4^1 q_0 4^0 q_1$).

EXAMPLE



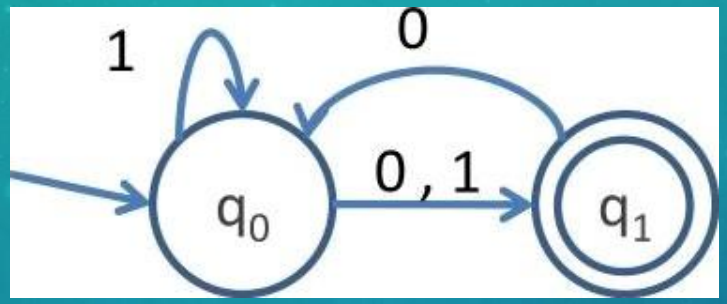
- This automaton accepts the string 1110 because there is a path under 1110 that takes us to an accepting state
- (the path $q_0 4^1 q_0 4^1 q_0 4^1 q_0 4^0 q_1$).

EXAMPLE



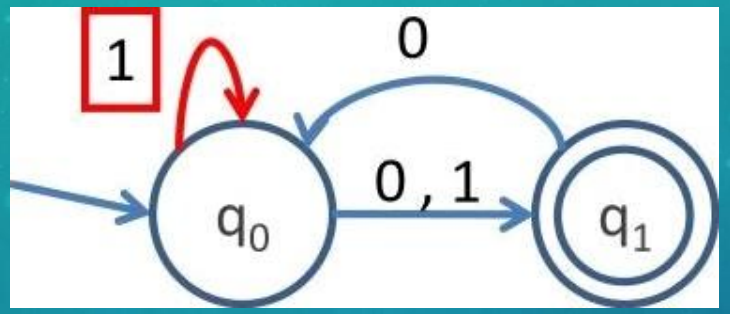
- This automaton accepts the string 1110 because there is a path under 1110 that takes us to an accepting state
- (the path $q_0 4^1 q_0 4^1 q_0 4^1 q_0 4^0 q_1$).

EXAMPLE



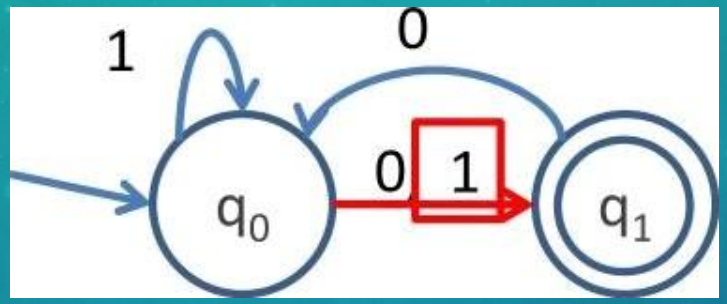
- And that is so despite the fact that there are some other paths under 1110 which do not lead to an accepting state
  (for example the path $q_0 4^1 q_0 4^1 q_0$).

EXAMPLE



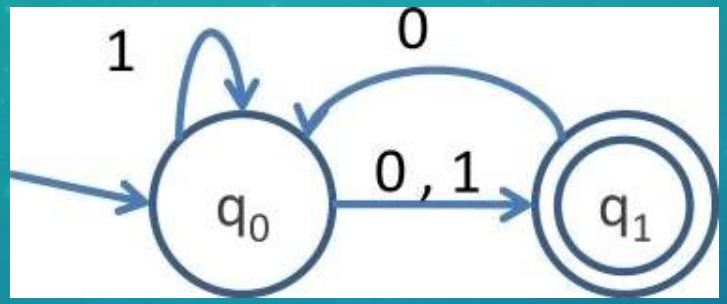
- And that is so despite the fact that there are some other paths under 1110 which do not lead to an accepting state
  (for example the path $q_0 4^1 q_0 4^1 q_g$).

EXAMPLE



- And that is so despite the fact that there are some other paths under 1110 which do not lead to an accepting state
  (for example the path $q_0 4^1 q_0 4^1 q_0$).

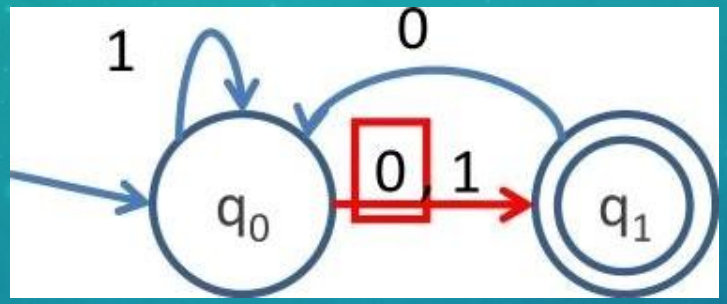
EXAMPLE



- However it doesn't accept the string 00 because there are no paths under 00 to take us to q_1.

  (only possible path is $q_0 \xrightarrow{0} q_1 \xrightarrow{0} q_0$).

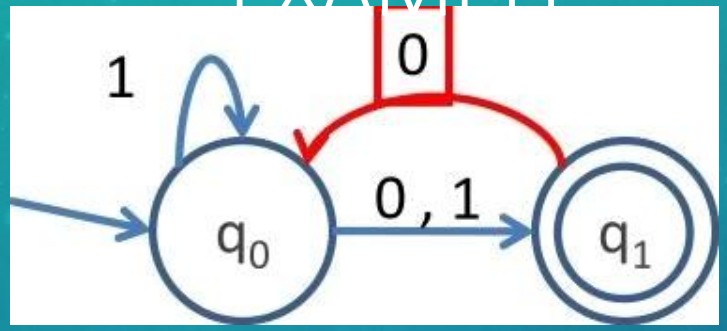
EXAMPLE



- However it doesn't accept the string 00 because there are no paths under 00 to take us to q_1.

  (only possible path is q 4 ^{0} q_1 4 ^{0} q_0).

EXAMPLE



- However it doesn't accept the string 00 because there are no paths under 00 to take us to q_1.

  (only possible path is $q \xrightarrow{0} q_1 \xrightarrow{0} q_0$).

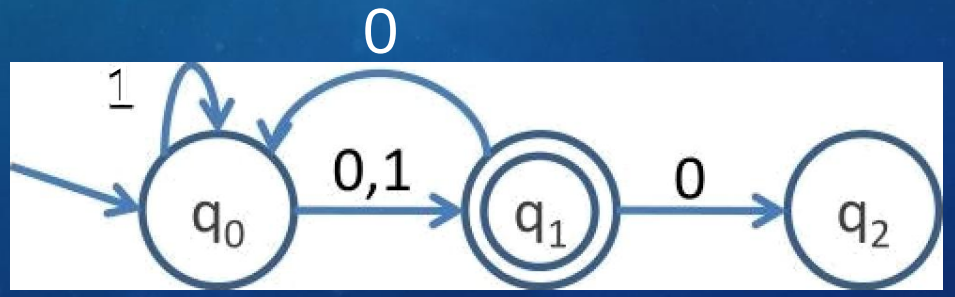
NFA - NFA_C ACCEPTANCE

- The language that a Non—Deterministic FA recognizes is the set of strings which accepts.
- To see whether a string can get accepted or not, it suffices to find the set of all possible states in which the automaton can be following all possible transitions suggested by this string as an input and see if a final state is contained in this set.

NFA ACCEPTANCE

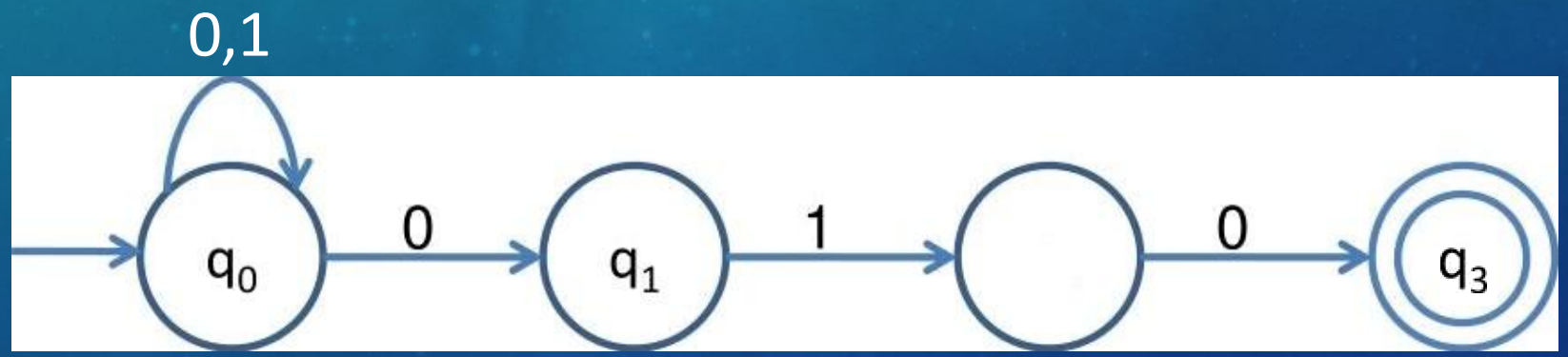
- Whenever an arrow is followed, there is a set of possible following states that the NFA can be. This set of states is a subset of Q.
- For example with input being 0010 I have the following sequence of sets of states:

$$q_0 \xrightarrow{0} \{q_1\} \xrightarrow{0} \{q_0, q_2\} \xrightarrow{1} \{q_0, q_1\} \xrightarrow{0} \{q_0, q_1, q_2\}$$



IS NFA MORE POWERFUL THAN DFA?

- Designing an NFA is sometimes mush easier than constructing a DFA. For example, the following NFA recognizes the language of all binary strings that end with 010.

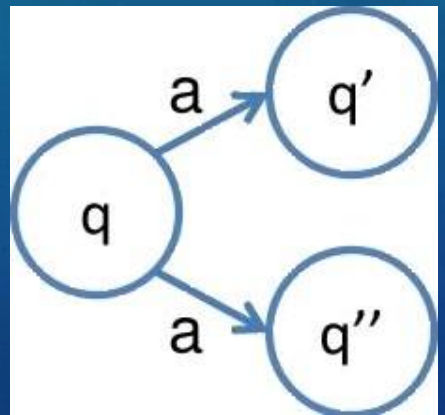


NFA DFA EQUIVALENCE

- It is obvious that a DFA is also an NFA.
- Somebody would expect the NFA to be more powerful. We will see that this is not the case!

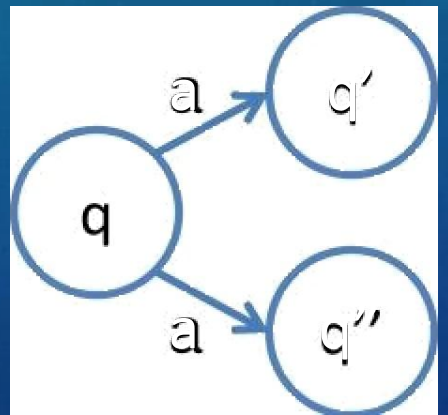
NFA —DFA EQUIVALENCE

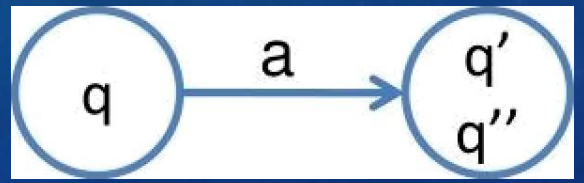
- An NFA might have more than one or no transitions under some symbol of the alphabet.



NFA – DFA EQUIVALENCE

- An NFA might have more than one or no transitions under some symbol of the alphabet.
- I can simulate all possible transitions in one new state. This state should contain all the reachable states under the same symbol.





NFA —DFA EQUIVALENCE

- The new DFA can possibly contain one state for each subset of states of the NFA.
- Since all the subsets of Q are 2! | total, this should be a finite (6 2!D) number of subsets!
- The NFA accepts if there is at least one path that takes us to an accepting state. Thus, each subset-state of the DFA containing an accepting state of the NFA should be an accepting one.

NFA –2 DFA

Suppose that you want to find an equivalent DFA for an NFA . The algorithm is the following:

- Starting from the start state and for each symbol in the alphabet, find the subset of states that can be reached after following this symbol and create a new state for each subset.

NFA –2 DFA

Suppose that you want to find an equivalent DFA for an NFA . The algorithm is the following:

- Repeat the same process for every new subset-state that you are creating...

NFA –2 DFA

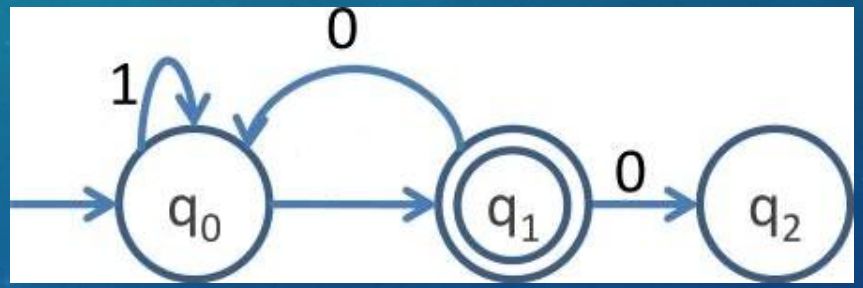
Suppose that you want to find an equivalent DFA for an NFA . The algorithm is the following:

- Repeat the same process for every new subset-state that you are creating...
- .. until no new subset-state can be created.

NFA —DFA EQUIVALENCE (EXAMPLE)

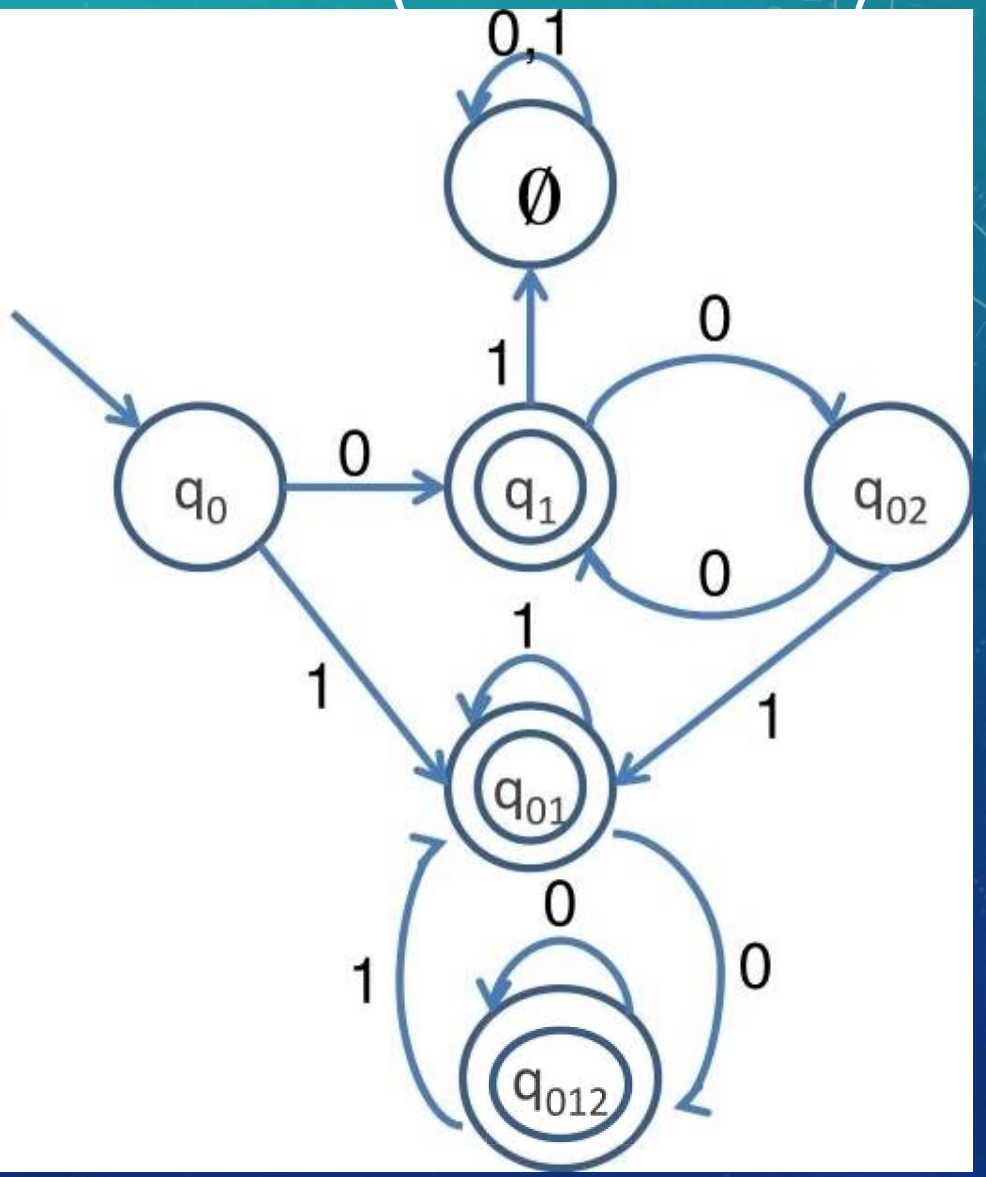
To find an equivalent DFA to the NFA of the figure we complete the following table:

	0	
q	lq l	lq₀ q₁l
Isib	lq q2l	»
I k q I	I k q i2l	t*‹ all
\q q2t	Incl	(Bo all
bw q q2l	lq q q2\	lq all



NFA ¯ DFA EQUIVALENCE(EXAMPLE)

	0	1
q_0	$\{q_1\}$	$\{q_0, q_1\}$
$\{q_1\}$	$\{q_0, q_2\}$	$\emptyset$
$\{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$
ïq q2ï	$\{q_1\}$	$\{q_0, q_1\}$
$\emptyset$	$\emptyset$	$\emptyset$
$\{q_0, q_1, q_2\}$	ïq q q2ï	ïq q ï

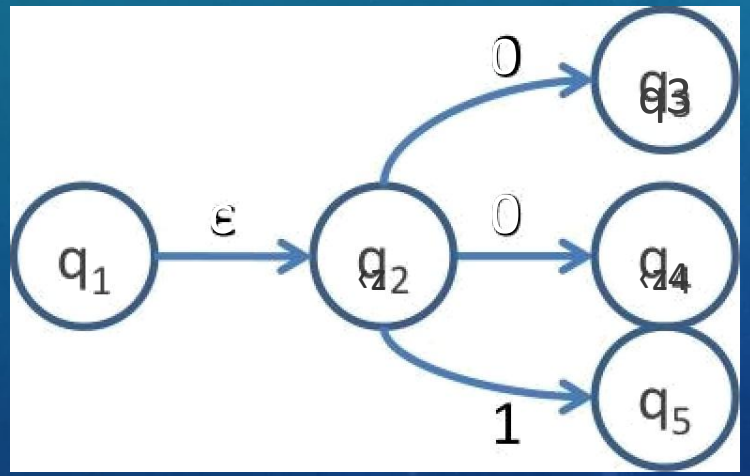


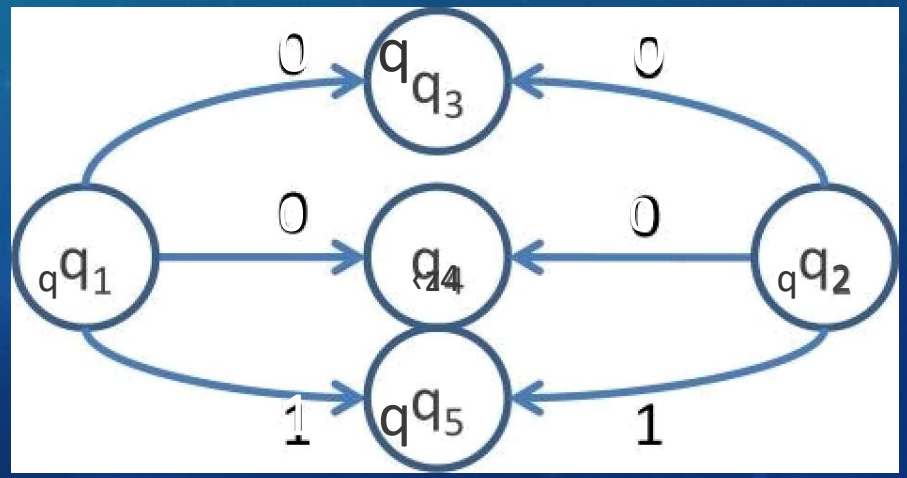
NFA, —NFA EQUIVALENCE

- An NFA, is an NFA which might have c-moves.
- Again, somebody would think that this attribute can make NFA, more powerful than NFA. This is not the case since I can get rid of the c-moves.

NFA, — NFA EQUIVALENCE

- Suppose that I have an c-move like the one shown in the figure. Since an c-move is like teleporting from q_1 to q_2, I can remove the c-move and add transitions from q directly to every neighbor of q_2.

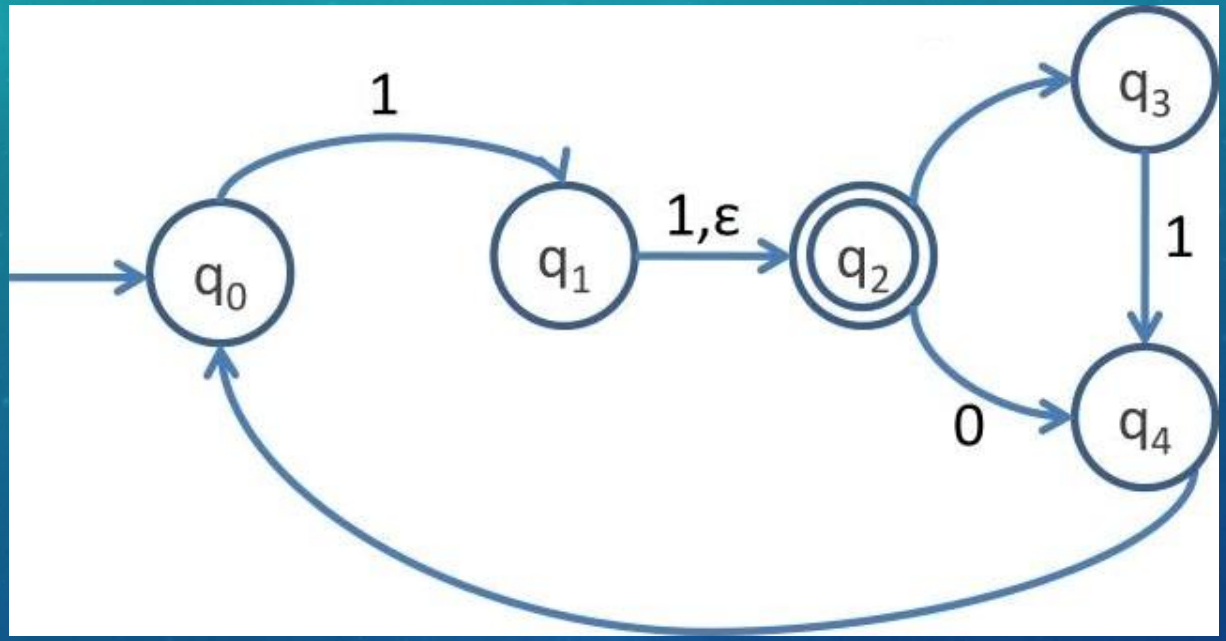




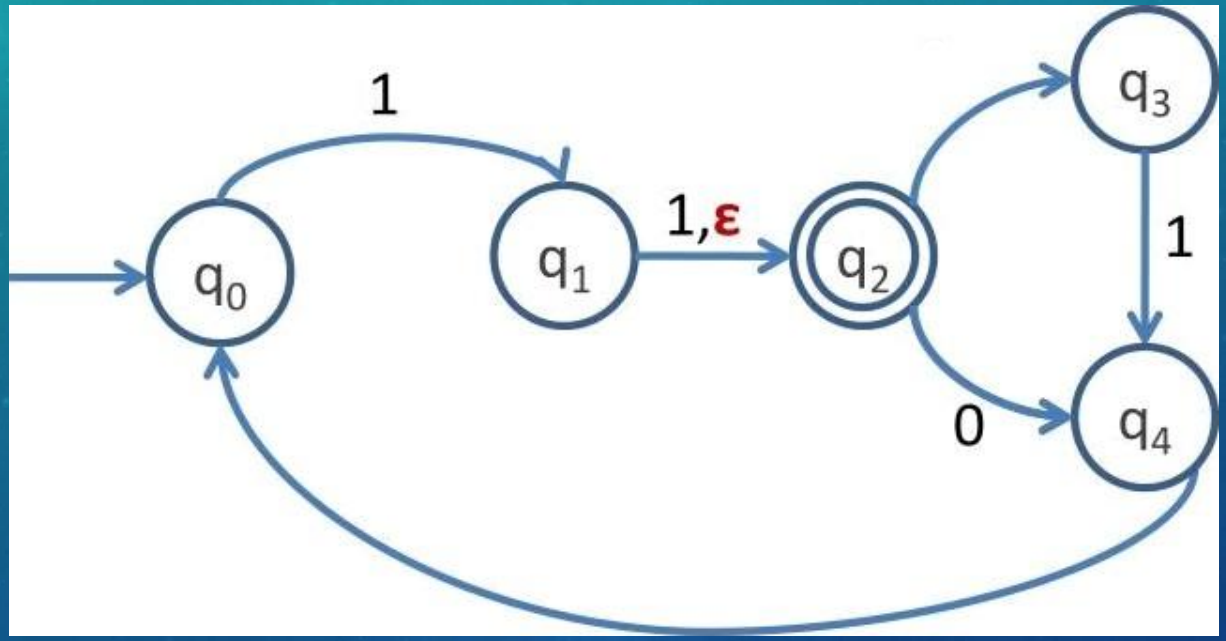
NFA, —NFA EQUIVALENCE

- If an c-move takes me from q_1 to q_2 and q_2 is an accepting state, then, when removing the c-move, I should convert q_1 to an accepting state.

EXAMPLE

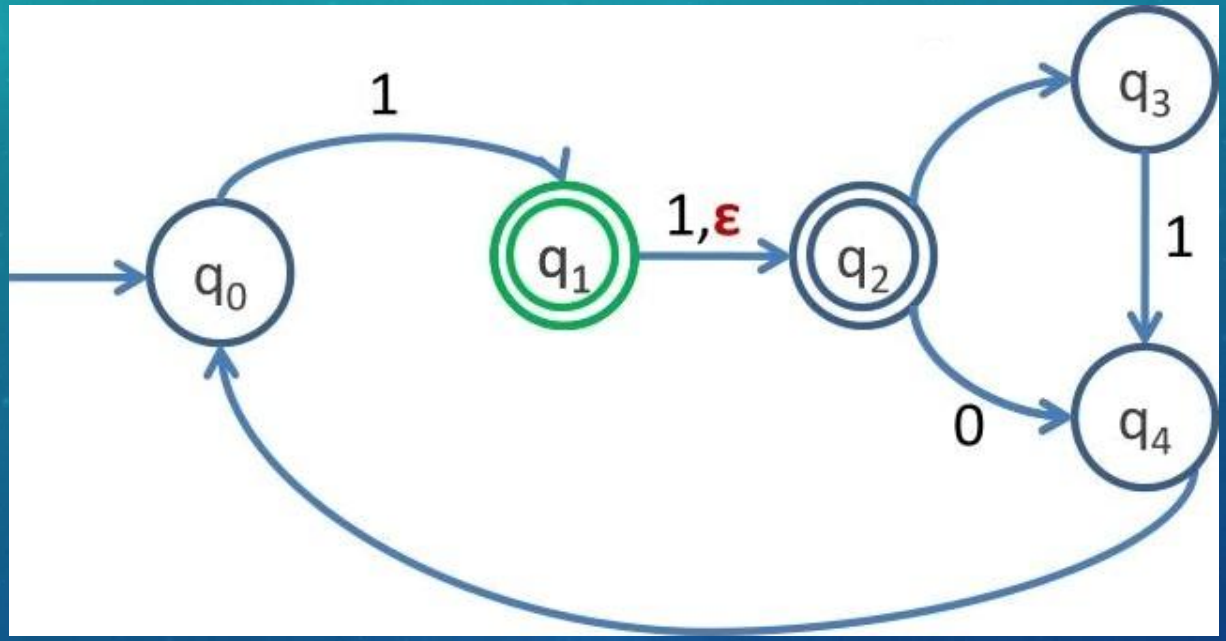


EXAMPLE

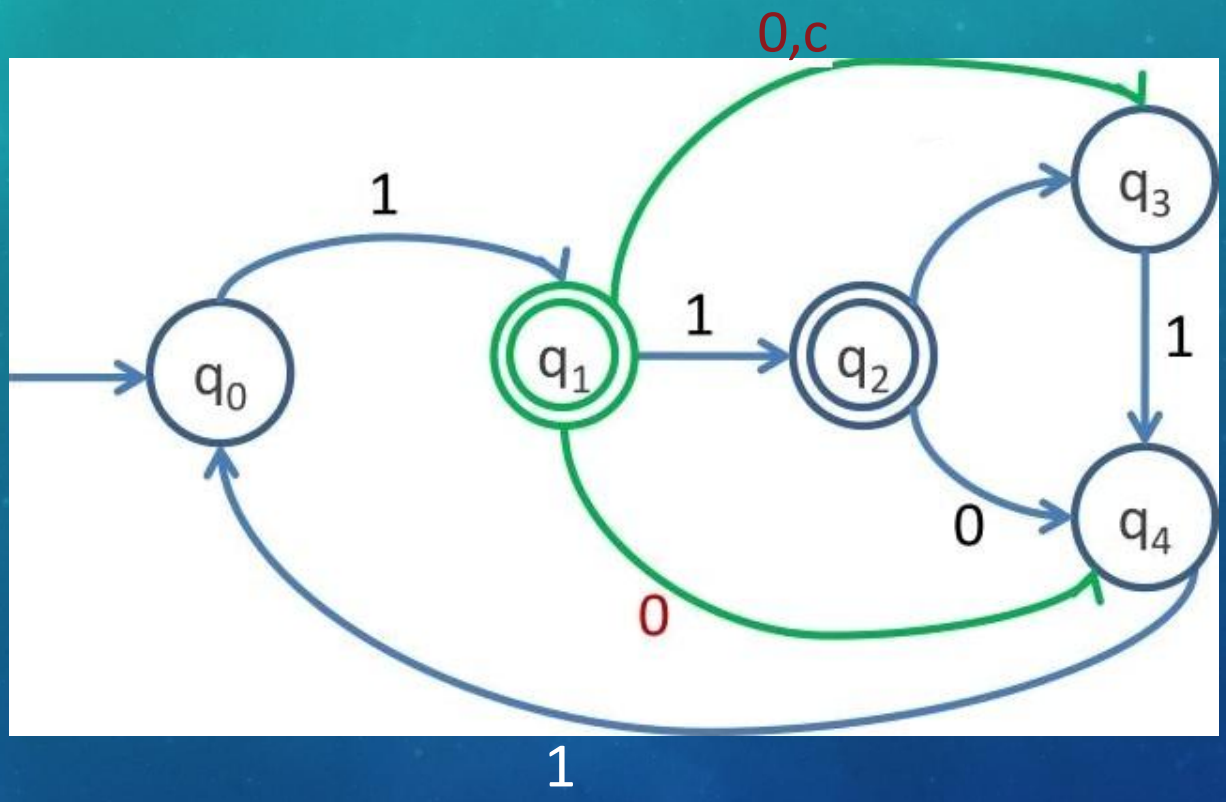


1

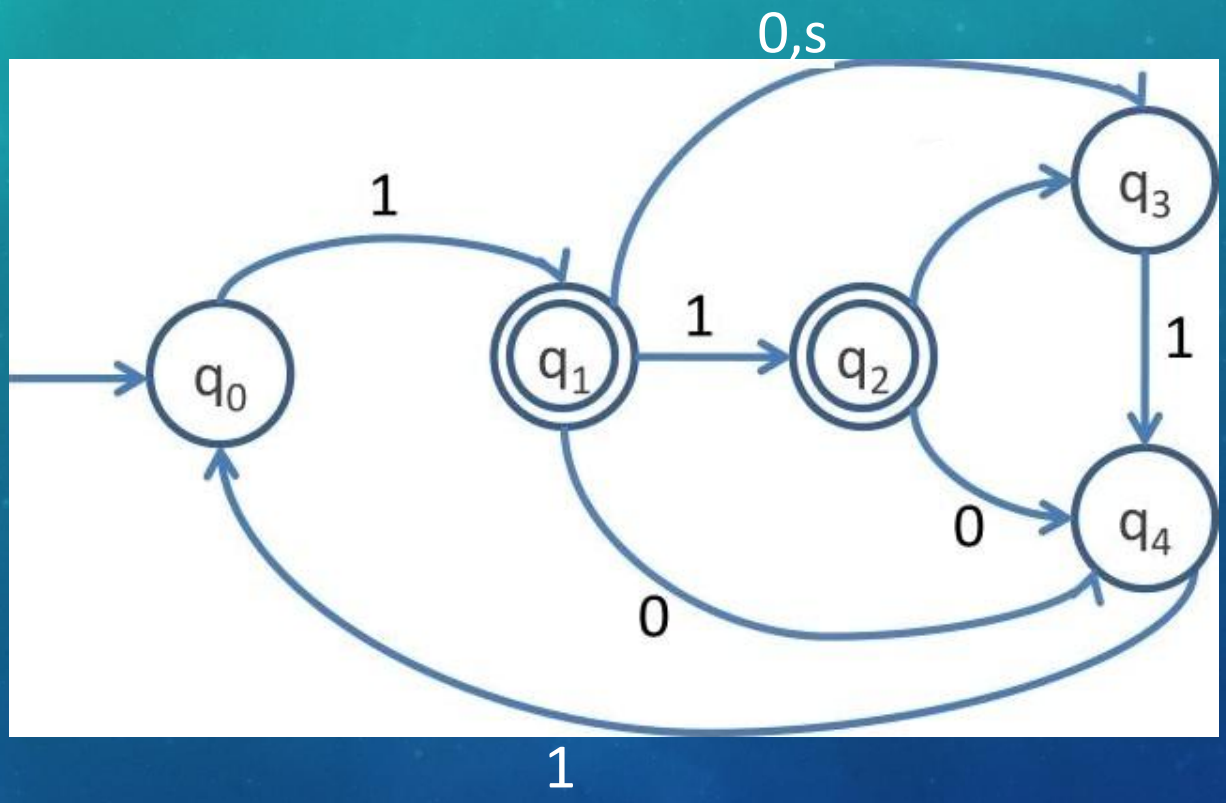
EXAMPLE



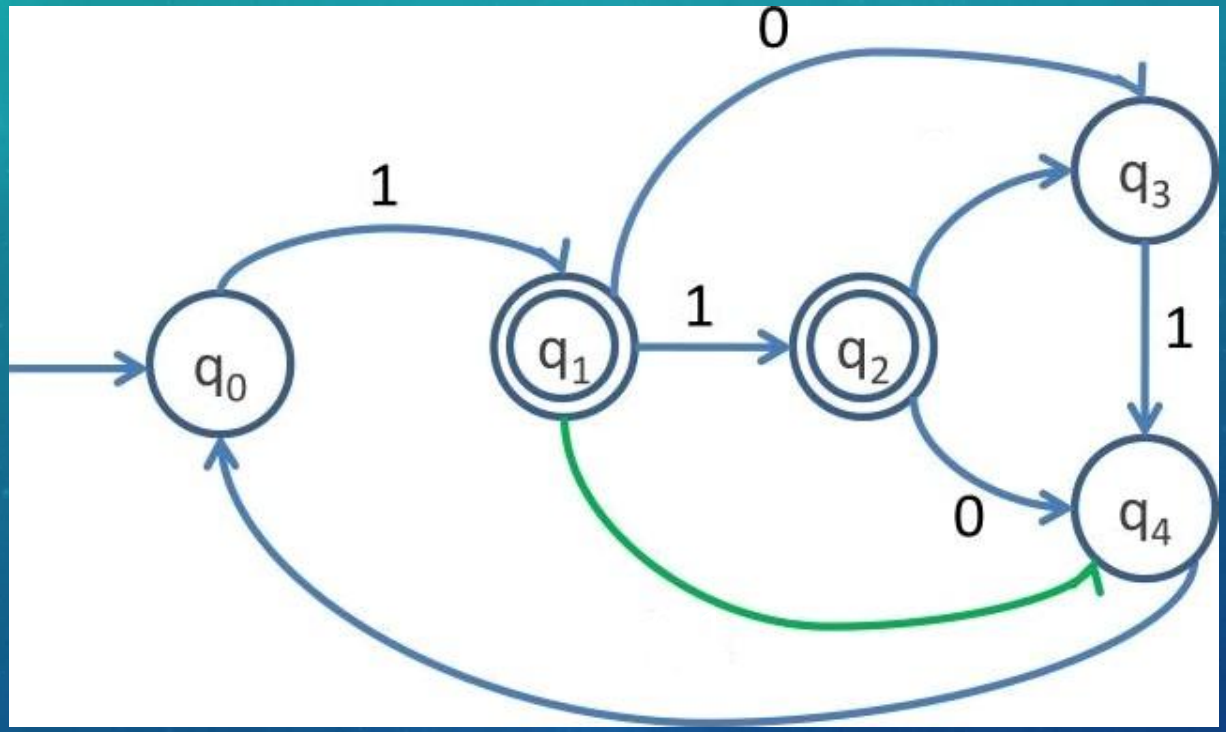
EXAMPLE



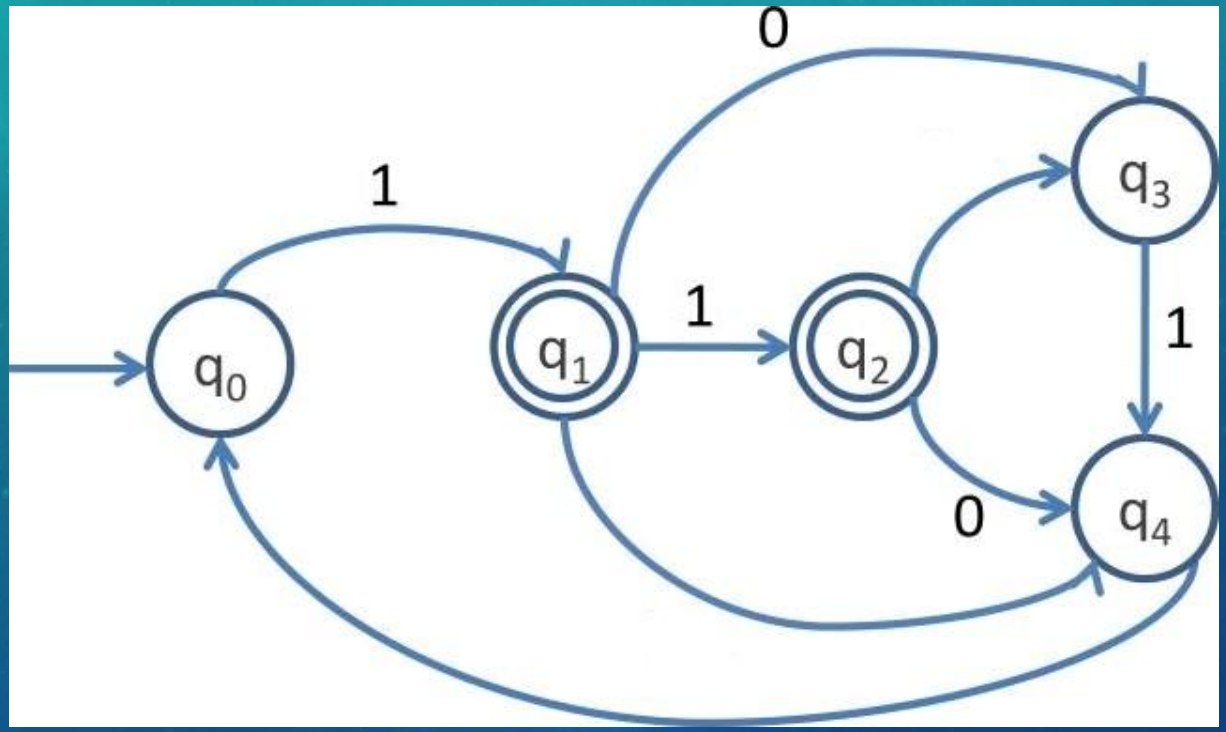
EXAMPLE



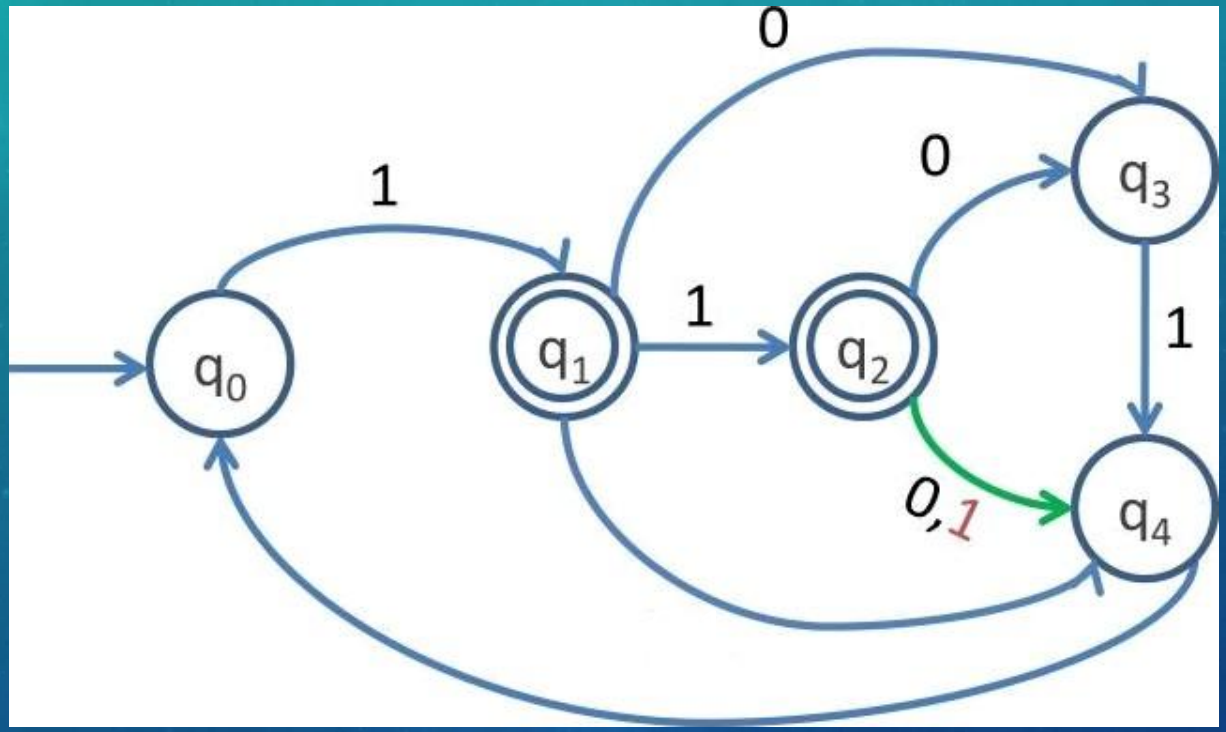
EXAMPLE



EXAMPLE

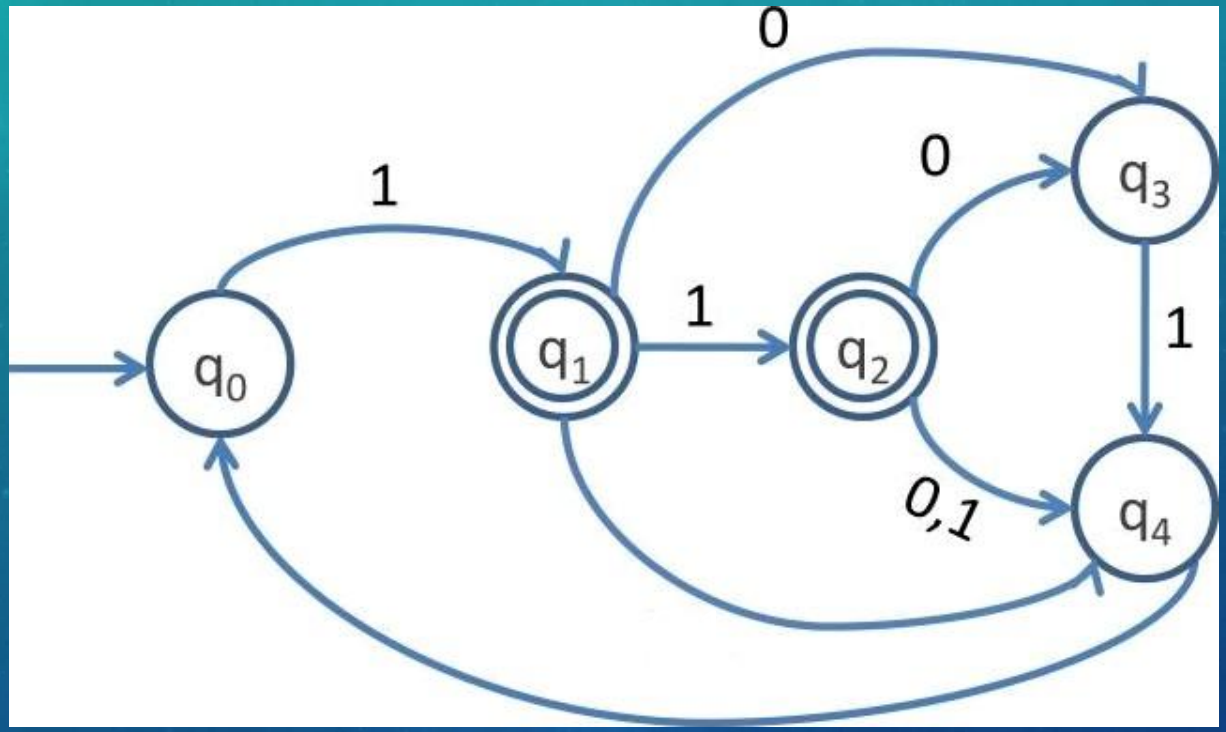


EXAMPLE



1

EXAMPLE



WEEK 14

REVISION AND RECAP

Decidability and Computability
Complexity Theory
Equivalence of NFA & DFA (Revisited)

WEEK 15

FEEDBACK AND FUTURE DIRECTIONS ABOUT FINAL TERM

- **Decidability and Computability**: Understanding what problems computers can and cannot solve.
- **Complexity Theory**: Measuring the efficiency of algorithms and identifying their limitations.
- **Equivalence of NFA & DFA**: Core principles of automata theory with practical applications.

Decidability and Computability

- **Key Concept**: Decidable problems have algorithms that always provide a solution (e.g., checking if a number is prime).
  Undecidable problems, like the Halting Problem, lack such universal solutions.
- **Real-World Impact**: These ideas guide the boundaries of software and hardware capabilities.

Complexity Theory

- **Key Concept**: Complexity theory helps us classify problems based on resources like time and memory.
  - *P vs. NP*: A central unsolved question—can every problem that's easy to verify also be solved efficiently?
- **Real-World Impact**: Optimizing algorithms for tasks like data analysis, cryptography, and scheduling.

Equivalence of NFA & DFA

- **Key Concept**: NFAs and DFAs are equally powerful in recognizing regular languages, but differ in complexity.
- **Real-World Impact**: Used in building compilers, regex engines, and pattern-matching systems.

WEEK 16

FINAL DOCUMENTATION/PRESENTATION

Showcase application of Theory of Computation concepts in a Documentation or Presentation.

WEEK 17

Final Examination revision and recap covering the entire course content