Automata Theory Tutorial

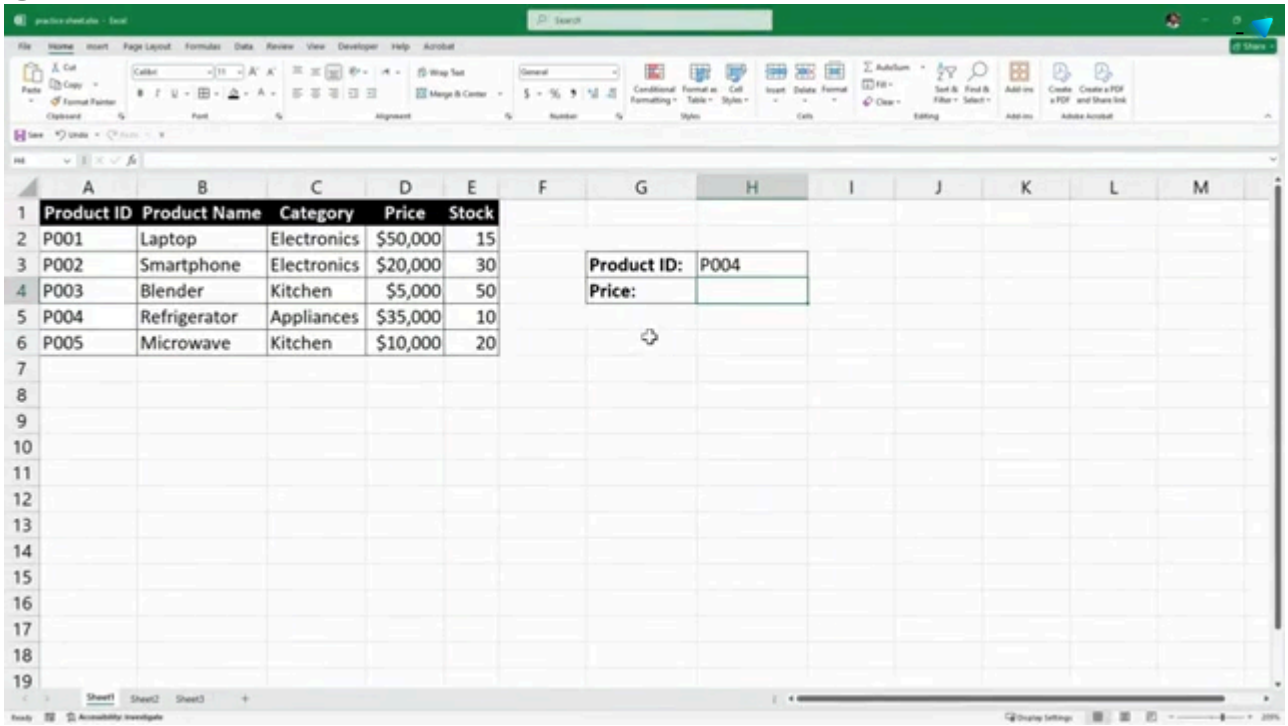
Automata Theory is a branch of computer science that deals with designing abstract self-propelled computing devices that follow a predetermined sequence of operations automatically.

An automaton with a finite number of states is called a **Finite Automaton**. This is a brief and concise tutorial that introduces the fundamental concepts of Finite Automata, Regular Languages, and Pushdown Automata before moving onto Turing machines and Decidability.

What is Automata Theory?

In mathematics and computer science, **automata theory** deals with abstract machines and computational problems. Automatons are self-propelled computing devices that follow predetermined operations automatically.

According to John von Neumann, "Automata theory is a logical theory that deals with the study of automata and information, emphasizing the need for a detailed, mathematical, and analytical theory to understand complex automata."



Product ID	Product Name	Category	Price	Stock
P001	Laptop	Electronics	$50,000	15
P002	Smartphone	Electronics	$20,000	30
P003	Blender	Kitchen	$5,000	50
P004	Refrigerator	Appliances	$35,000	10
P005	Microwave	Kitchen	$10,000	20

What is an Automaton?

An automaton is a finite-sized device with specified inputs and outputs, whose behavior is determined by inputs. In other words, an automaton refers to self-operating mechanical objects that transform information based on predetermined instructions or procedures.

An automaton can also refer to a class of electromechanical devices, both theoretical and real, that transform information after being set in motion.

Types of Automata

The automata could be classified into different types as follows –

- **Finite Automata** – A Finite automaton is an automaton with a finite number of states that changes its state based on the input string of symbols. When the desiring symbol is search, then the transition occurs.
- **Pushdown Automata** – A pushdown automaton (PDA) is a stack-based automaton used in the theory of computation. It is more capable than finite-state machines to solve little complex problem than FA.
- **Turing Machine** – A Turing machine is a theoretical device that uses a table of rules to manipulate symbols on a tape strip, consisting of an infinitely long tape divided into cells with 1's, 0's, or empty spaces.
- **Linear Bounded Automata** – A linear-bounded automaton (LBA) is a Turing machine that uses only the input tape space, unlike a Turing machine. Its length is a linear function of the input's length, and only a finite contiguous portion of the tape can be accessed by the read/write head.
- **Cellular Automata** – Cellular automata are deterministic systems that evolve in discrete time and space, typically a grid. They consist of locally updated cells that follow a global time scale and recursive rule, updating synchronously across the grid.

Who Should Learn Automata Theory?

This tutorial has been prepared for students pursuing a degree in any information technology or computer science related field. It attempts to help students grasp the essential concepts involved in automata theory.

Prerequisites to Learn Automata Theory

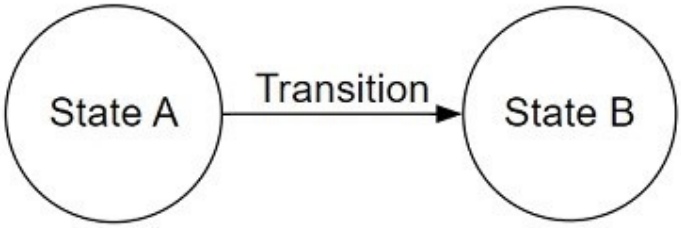
This tutorial has a good balance between theory and mathematical rigor. The readers are expected to have a basic understanding of discrete mathematical structures.

FAQs on Automata Theory

We have collected here a set of **Frequently Asked Questions** (FAQs) on **Automata Theory**, followed by their answers.

1. What is a Finite Automaton?

A **finite-state machine (FSM)** is a mathematical model of computation that can be in one of a finite number of states at any given time. It can change from one state to another in response to inputs, called a **transition**.



An FSM is defined by its states, initial state, and inputs. There are two types of FSM: **deterministic** and **non-deterministic**. For any non-deterministic FSM, an equivalent deterministic one can be constructed.

2. Difference between Deterministic and Non-deterministic Finite Automata

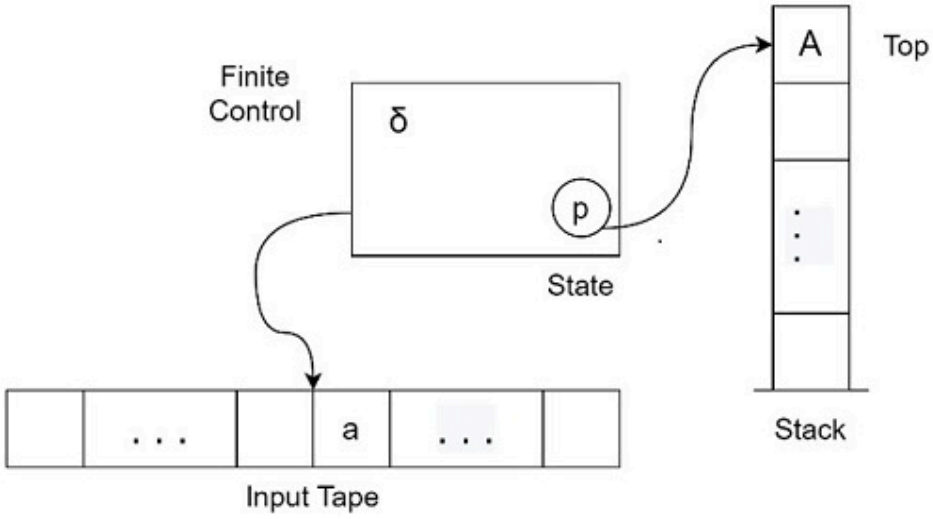
The following table highlights the major differences between **Deterministic** and **Non-deterministic Finite Automate** –

Aspect	DFA	NFA
State Transitions	Each input determines the resulting state uniquely	Some inputs may allow a choice of resulting states
Predictability	Final state is completely determined by input word	Several possible final states for a given input word
Empty Transitions	Can change state only after reading an input	May change state without reading any input
Initial State	One initial state	May have more than one initial state
Word Acceptance	Word is accepted if the final state is an acceptor state	Word is accepted if at least one possible final state is an acceptor state

State Occupancy	In one state at a time	Can be thought of as being in multiple states at once

3. What is a Pushdown Automaton?

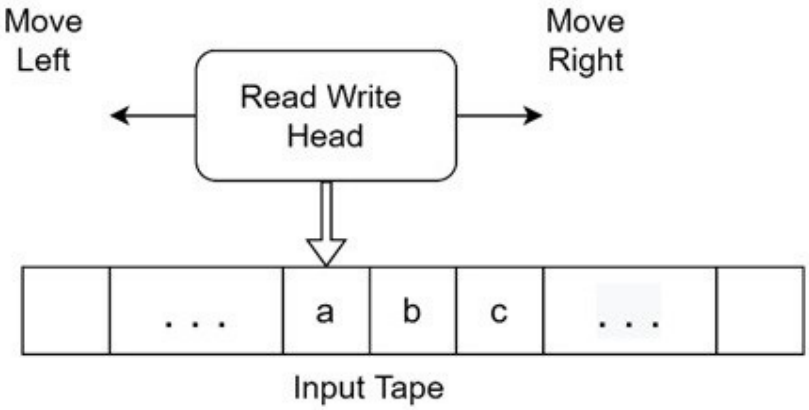
Pushdown automata are nondeterministic finite state machines with additional memory in the form of a stack. It is more powerful than a FSM but less than a Turing machine. They accept context-free languages.



For example, to ensure a valid code, a programmer can feed the code into a pushdown automaton programmed with transition functions that implement the context-free grammar for the language of balanced parentheses. If the code is valid and all parentheses are matched, the pushdown automaton will accept the code. If unbalanced parentheses are present, the automaton can return the code's invalidity.

4. What is a Turing Machine?

A **Turing machine** is an abstract computational model that performs computations by reading and writing to an infinite tape. It provides a powerful computational model for solving computer science problems and testing the limits of computation. Turing machines was invented by Alan Turing in 1936.



A Turing Machine consists of an infinite tape, a tape head, and a state transition table. They execute on an input string of bits, with the tape head in a certain state and the table governing its behavior.

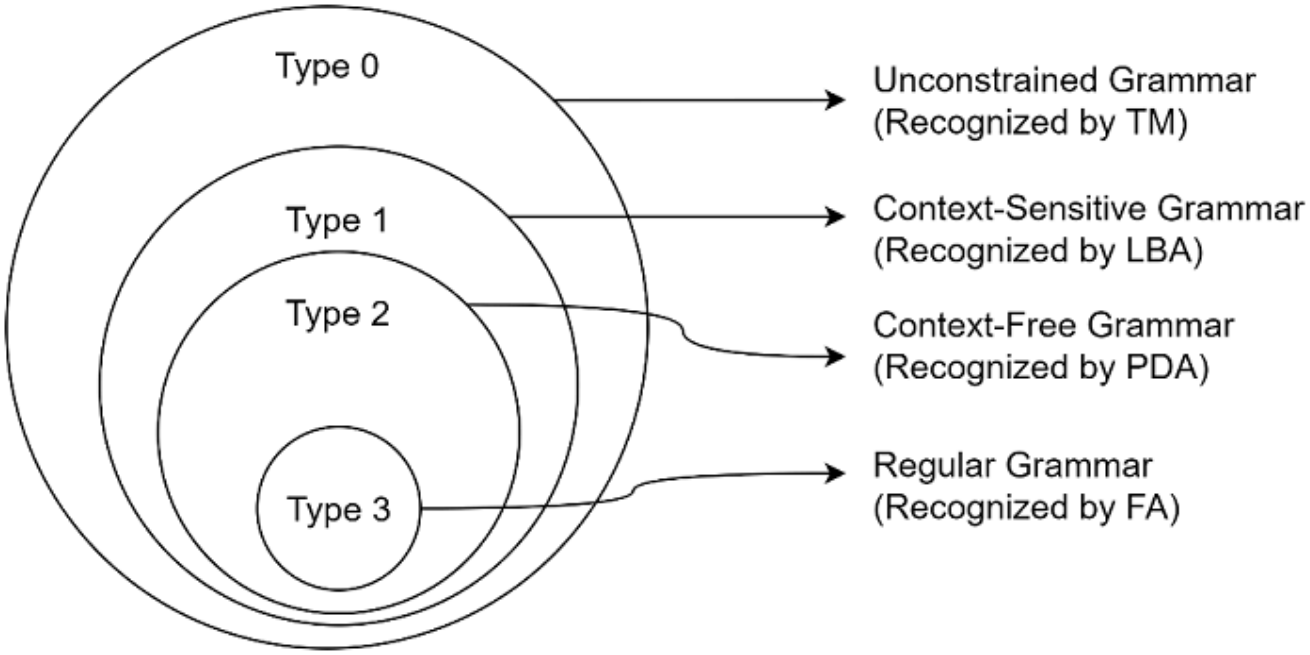
Turing machines can simulate the complexity of a program and its reactions to different data in memory.

5. What is the Chomsky Hierarchy?

The **Chomsky** hierarchy is a classification system in computer science and linguistics that consists of four levels –

- Type 0 (unrestricted)
- Type 1 (context-sensitive)
- Type 2 (context-free)
- Type 3 (regular)

Named after Noam Chomsky, it characterizes the expressive power of different types of formal languages and grammars. It is a fundamental concept in the study of formal languages and is used in the development of parsing algorithms and other tools for working with formal languages.



6. What are Context-Free Languages?

Context-free languages (CFLs) are generated by **context-free grammars**, which are identical to the set of languages accepted by pushdown automata. Regular languages are a subset of context-free languages, and inputted languages are accepted by computational models if they end in an accepting final state.

Formal language theory defines a language as a set of symbols constrained by specific rules, like the written English language. A valid sentence must follow specific grammar rules. A context-free language is a **language generated** by a context-free grammar, which can be generated by multiple context-free grammars, making it more general and inclusive.

$\{a^n b^n \mid n > 0\}$ is an example of a Context Free Grammar, which represents all strings with equal numbers of a's and b's. For example $a^4 b^4$ is **aaaabbbb**.

7. What are Context-Sensitive Languages?

Context-sensitive languages are more powerful than context-free grammars due to their ability to describe certain languages that cannot be described by context-free grammars. They are positioned between context-free and unrestricted grammars in the Chomsky hierarchy.

A Context-Sensitive Grammar (CSG) allows production rules to be surrounded by terminal and nonterminal symbols.

$\{a^n b^n c^n \mid n > 0\}$ is an example of **CFG**, which represents all strings with equal numbers of a's and b's and cs. For example $a^4 b^4 c^4$ is **aaaabbbbcccc**. To generate such language through CSG, it is necessary to keep track of the left and right contexts while generating.

8. What are Recursively Enumerable Languages?

Recursive Enumerable (RE) languages, also known as Type-0 languages, are generated by type-0 grammars, and can be accepted or recognized by a Turing machine.

RE languages can enter the final state for the language's strings and may or may not enter the rejecting state for non-language strings. TM can loop forever for non-language strings, making them Turing recognizable languages.

These strings may do one of the following –

- Halt and Accept
- Halt and Reject
- Loop Forever

Another subset of Recursive Enumerable is Recursive Languages which can be accepted by Total Turing Machine, and it will halt in either of the two cases: "Halt and accept" or "Halt and reject".

9. What is the Significance of Pumping Lemma?

Pumping lemma is a kind of Lemma (Generally minor, proven proposition which can be used as a stepping stone to a larger result) where we prove something through contradiction. In TOC pumping lemma are used to disprove a language is regular or context free (if we use **pumping lemma for CFG**).

Pumping lemma could be serves as a tool for proving that certain languages are not regular or not context-free. By applying the pumping lemma, we can show that some languages cannot be recognized by finite automata or pushdown automata. This is particularly useful in theoretical computer science and formal language theory to establish the limitations of these computational models.

The pumping lemma helps in classifying languages and understanding the capabilities and constraints of different types of automata.

10. What is a Grammar in the Context of Formal Languages?

A formal grammar is a set of production rules for strings in a formal language, defining which strings are valid according to the language's syntax, rather than their meaning or context. It focuses on the form of these strings in a formal language.

It is represented as **G(V, N, P, S)**. It generates all syntactically correct strings over the alphabet, primarily used in the syntactic analysis phase, particularly during compilation, during the compilation phase.

G = (V, N, P, S) is a grammar, where –

- **N** is a finite set of non-terminal symbols
- **V** is a finite set of terminal symbols
- **P** describes a set of production rules
- **S** is the start symbol

11. Difference Between a Language and a Grammar

A grammar is a set of production rules which are used to generate strings of a language. **Grammars** are nothing but a collection of symbols, production rules and a start symbol, represented through four-tuples **G = (V, N, P, S)**.

Language on the other hand the product of a grammar. If a grammar is not defined, we cannot create a language. In the following example it is shown the concept of **grammar and languages** used by that grammar.

\mathrm{G \: = \: (V,N,P,S) \: = \: (\lbrace S,A,B,C \rbrace , &

A possible language **L(G)** could be formed through this grammar, **L(G) = {abc}**

S → **ABC** from **A** → **a, B** → **b** and **C** → **c**, it will produce "**abc**" which is a language.

12. Difference Between a Decidable Problem and an Undecidable Problem

For **decidable** problems an algorithm can provide a definite answer, like "yes" or "no" for any given instance, while **undecidability** refers to problems where no algorithm can provide a definite answer for all possible instances.

Aspect	Decidable Problems	Undecidable Problems
Definition	Algorithm exists for definite answer	No algorithm for definite answer in all cases
Answer	Definite "yes" or "no"	Lack definite answer for all instances
Algorithm	Efficient solving algorithm exists	No algorithm for all instances
Time	Finite answer within reasonable time	May not terminate for all inputs
Proof	Proven by existence of algorithm	Proven through rigorous mathematical proofs

Impact	Allows efficient problem-solving	Poses fundamental challenges
Example	Language membership, sorting	Halting problem, Collatz Conjecture

13. What is the Halting Problem?

The halting problem is a decision problem in computability theory that determines whether a computer program will terminate or run forever.

For example, if we take a positive number "**x**" (x > 0) from the user and inside a while loop, check "x" is 0 or not, inside the loop we are not updating the value of "x". This will never halt.

Halting problem is a well-known problem that has been proven to be undecidable, meaning no program can solve it for general enough computer programs.

Turing machines, which are as strong as "usual computers," are often used in computation theory. In 1936, Alan Turing proved that the halting problem over Turing machines is undecidable using a Turing machine, meaning no Turing machine can correctly decide for all possible program/input pairs.

14. What are P and NP Classes in Computational Complexity?

P is a complexity class that includes decision problems that can be solved by a deterministic Turing machine using polynomial time. It is often considered "efficiently solveable" or "tractable" computational problems.

Intractable problems are theoretically solvable but cannot be solved in practice. P contains natural problems like calculating the greatest common divisor, and finding maximum matching, determining if a number is prime is also part of P.

NP, or Non-deterministic Polynomial Time, is a set of decision problems solvable in polynomial time on a non-deterministic Turing machine. These problems can be verified by a deterministic Turing machine in polynomial time. Some examples include Boolean satisfiability problem (SAT), the Hamiltonian path problem (special case of TSP), the Vertex cover problem, etc.

15. Difference Between NP-Complete and NP-Hard Problems

In computation theory, a problem **X** can be termed as NP-Hard if there is an existing NP-Complete problem **Y** that can be reduced to **X** within polynomial time.

The difficulty level of an NP-Hard problem is like that of an NP-Complete problem. However, NP-Hard problems do not necessarily have to fall under the NP Class. A problem can only be NP-Complete if it's part of both NP-Hard and NP Problems.

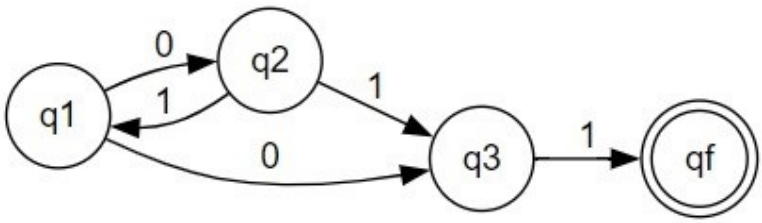
Aspect	NP-Hard	NP-Complete
Meaning	The solution to an NP-Hard Problem X requires the existence of an NP-Complete Problem Y, which can be reduced to the problem X in polynomial time.	A problem X is NP-Complete when there is an NP problem Y, which is reducible to X in a polynomial line.
NP Class	Doesn't need to be in NP class	Must be in NP class
Relationship	Subset of NP-Hard is NP-Complete	Must be both NP and NP-Hard
Examples	Circuit-satisfactory Vertex Cover	Hamiltonian cycle determination in a graph Boolean formula satisfiability problem.

16. What is a Transition Function in the Context of Automata?

In the context of automata theory, there are certain tables which includes states, input symbols, starting state, set of final states and transition functions.

(Q, Σ, q0, F, T), where **Q** is the set of states and **Σ** are the input symbols.

When a state **q** takes an input from the set **Σ**, it sends from **q** to another set **q'** or the **q** itself. The transition based on inputs are defined in the transition functions. The set of transition functions are mentioned in the set **T**.



In this state diagram, there are several transitions, some of them could be –

$$\delta(\mathrm{q1}, 0) \rightarrow \mathrm{q2}$$

$$\delta(\mathrm{q3}, 1) \rightarrow \mathrm{qf}$$

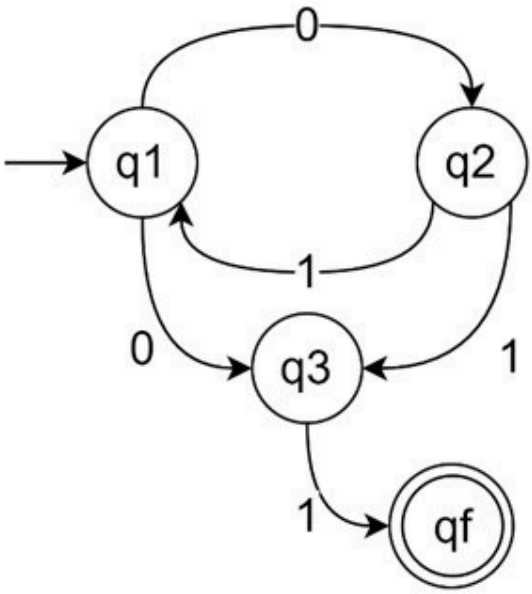
17. What is a State Diagram?

A state diagram in automata theory is a visual representation of a finite automaton's behavior.

It consisting of vertices (the set **Q**) representing states, input symbols (**Σ**) and output symbols (Z), if there is no output set, it could have a subset of **Q, F** the set of Final states or acceptance states, and a set of transitions **T**.

It is used to describe and analyze the system behavior by showing possible states and transitions between these states. The state diagram provides a clear and concise way to understand the possible states and transitions of a finite automaton.

Here is a sample state diagram −



Here we have four states:

- Q = {q1, q2, q3, qf}
- Final state F = {qf}
- Inputs Σ = {0, 1}
- Initial state q1

And,

\mathrm{T \: = \: \lbrace \delta(q1, 0) \: \rightarrow \: (

18. What is the Significance of States in Automata?

Automata theory is used to conceptualize a system which helps to design it in real life. It must have the inputs, outputs and transitions associated with inputs. The fundamental components that represent a finite number of conditions or situations a system are the states in state diagrams

States are crucial for analyzing and representing a system's behavior, tracking different conditions of objects. A finite set of states describes an automaton's behavior in

response to input symbols. This state-based approach is particularly useful for systems that can be abstracted into a finite number of distinct conditions.

19. What is a Deterministic Pushdown Automaton?

A **deterministic pushdown automaton** (DPDA or DPA) is a variation of the pushdown automaton in automata theory, accepting deterministic context-free languages.

Here the machine transitions are based on the current state, input symbol, and topmost symbol of the stack, with lower symbols having no immediate effect. Machine actions include pushing, popping, or replacing the stack top.

Formally we can say, for a PDA machine, **M = (Q, Σ, Γ, δ, q0, z, F)** is deterministic if for every **q** ∈ **Q, a** ∈ **Σ** ∪ **{λ}, b** ∈ **Γ**

- **δ(q, a, b)** has at most one element
- If **δ(q, λ, b)** is non-empty and **δ(q, c, b)** must be empty for every **c** ∈ **Σ**

20. What is a Non-deterministic Pushdown Automaton?

A **Non-deterministic Pushdown Automaton** (NPDA) is a machine that has all the features of an NFA and a stack. Its program uses its current state, current symbol under the read head, and the symbol at the top of the stack to make decisions. NPDAs are more powerful than deterministic PDAs.

The stack in NPDAs is distinct from the input tape's "language" alphabet. The stack is used in the start state of the control automaton, with only initial symbols on the stack.

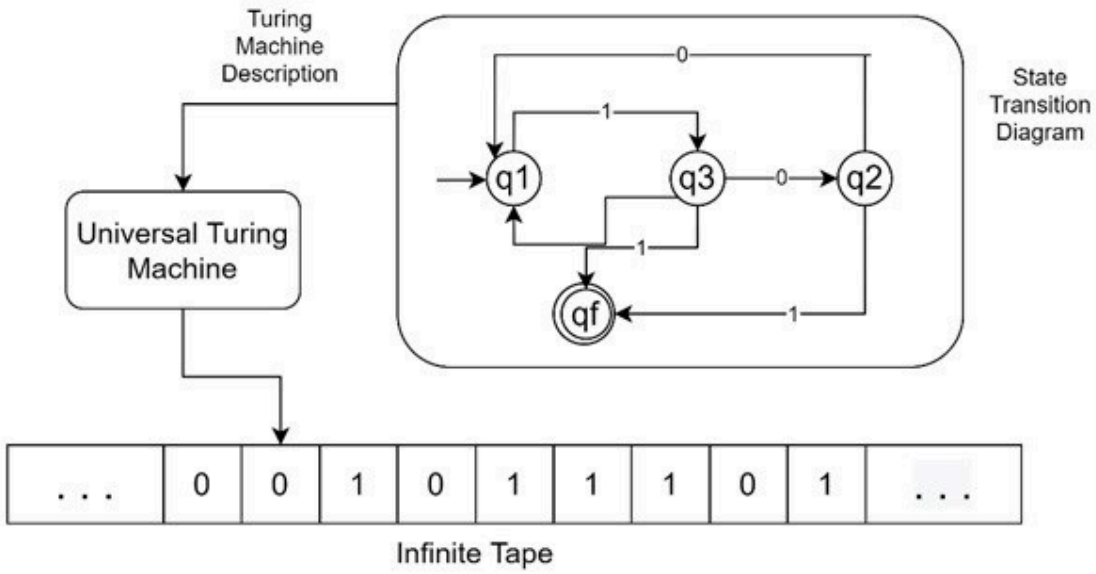
The state, input element, and top symbol in the stack determine the next step (transition) at each step. Transition steps include making a state change, reading a symbol from the input tape, shifting to the next symbol on the right, and modifying the stack.

21. What is a Universal Turing Machine?

A Turing Machine is a mathematical tool that is equivalent to a digital computer (Turing, 1930s). It consists of an input output relation that the machine computes, with the input given in binary form on the machine's tape. The output consists of the tape's contents when the machine halts. The contents of the tape are determined by a finite state machine (FSM) inside the Turing Machine

However, Turing Machines require constructing a different one for every new computation and input output relation. This led to the invention of the **Universal Turing Machine** (UTM), which takes in the description of a machine M and can simulate it on the rest of the input tape's contents.

Take a look at the following conceptual diagram of Universal Turning Machine –



Here, we are using the infinite tape, the UTM is working on them, but additionally it is taking the machine description or state transition from a separate block to the UTM module, in implementation, the tape itself holds the machine description at a separate zone.

22. Why Formal Languages are used in Computer Science?

A formal language in computer science is a set of strings over a finite set of symbols, called an alphabet, and structured strings created using this alphabet, based on defined grammar rules, constitute the formal language.

Formal languages are structured around three key concepts: Alphabet, String, and Grammar.

- **Alphabet** is a finite set of distinct symbols,
- **String** is a finite sequence of symbols selected from an alphabet. The order of symbols matters in a string, and empty strings have zero symbols.
- **Grammar** is a set of formal rules that govern the combination of symbols to compose strings in a formal language.

These rules are changed for different classes of formal languages as regular, context-free, context-sensitive, and recursively enumerable.

23. What is the Role of Automata in Parsing?

Parsing is a grammar-based process that uses production rules to derive a string and check its acceptability. It is a tool used to check if a string is syntactically correct or not.

A parser can be top-down or bottom-up, starting from the top with the start symbol and using a parse tree to derive the string.

Top-Down Parsing

- The PDA uses top-down parsing to match the start symbol of a grammar on its stack with the input.
- If it's a terminal, then compare it with the current input symbol,
- If it is non-terminal, then replace by production rules.
- The stack tracks the parsing context and expected symbols, and input is accepted if the stack empties and all input is consumed.

Bottom-Up Parsing

- The PDA starts with an empty stack and reads the input symbols.
- It reduces the sequences to non-terminals through shift-reduce actions.
- The stack contains partially constructed parse trees, input is accepted if the start symbol is present when all input is consumed.

24. What is a Linear-bounded Automaton?

A **linear bounded automaton** is a multi-track non-deterministic Turing machine with a bounded finite length of tape. The computation in linear bounded automata is restricted to the constant bounded area, with two special symbols serving as left and right end markers.

A deterministic linear bounded automaton is context-sensitive, and a linear bounded automaton with an empty **language is undecidable**.

Nondeterministic Linear Boundary Algorithms (LBAs) bear the same relationship to context-sensitive languages as pushdown automata do to context-free languages.

It is unknown whether deterministic versions of LBA have equivalent power to the nondeterministic version.

It is a 9 tuple automata: G = (Q, Σ, Γ, δ, q0, B, F, GL, GR)

- **Q** – the set of states
- **Σ** – Set of input alphabets
- **Γ** – Tape alphabets Σ ⊂ Γ

- **δ** – set of transition functions
- **q0** – Initial state
- **B** – Set of blank symbols, B ∈ Γ Σ
- **F** – Set of final states F ⊂ Q
- **GL** – Left terminal GL ∈ Σ
- **GR** – Right terminal GR ∈ Σ

25. What is the Significance of Myhill-Nerode Theorem?

The **Myhill-Nerode theorem** states that a language **L** is regular if **L~** (**L~** is relation on strings **x** and **y**, where no distinguishable extension for **x** and **y**) has a finite number of equivalence classes, and if this number is **N**, then there are **N** states in a minimal Deterministic Finite Automaton (DFA) recognizing L.

Apart from the definition, the Myhill-Nerode theorem is used to explain the concept that which languages simple machines, or "finite state automata," can recognize. It helps to understand which languages these simple machines can handle or how much simplification can be done on a machine to accept the languages.

26. What is the Equivalence of Automata and Grammars?

The equivalence indicates the relationship between different types of automata and the classes of grammars they can recognize. According to Chomskys classification there are four types of grammar.

Languages	Automata
Recursively Enumerable Languages: Equivalent to Type-0 grammars	Recognized by Turing Machines
Context-Sensitive Languages: Equivalent to Type-1 grammars	Recognized by Linear Bounded Automata (LBA)
Context-Free Languages: Equivalent to Type-2 grammars	Recognized by Pushdown Automata (PDA)
Regular Languages: Equivalent to Type-3 grammars	Recognized by Finite State Automata (FSA)

27. What is a Regular Expression?

The Type-3 Grammar in Chomskys classification is also termed as regular grammar. A Finite automaton can accept regular languages through simple expressions which is

known as **Regular Expressions**

Regular expressions are the most effective way to represent any language. These expressions define a string and match character combinations. The string searching algorithm uses this pattern to find operations on a string, making it easy to check the language.

A regular expression, denoted by **x***, indicates zero or more occurrences of **x**, while **x+** signifies one or more occurrences of **x**, generating a range of characters.

(a | b)* denotes all possible combination of **a** and **b** of any size. [Here the symbol "**|**" denotes **OR**]

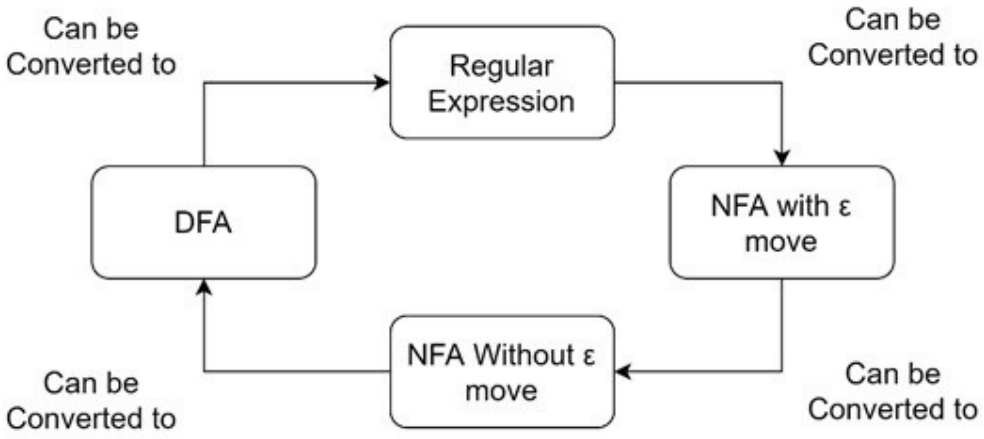
28. How do Regular Expressions Relate to Finite Automata?

Regular expressions are a language-describing language accepted by finite automata, providing the most effective representation of any language, with **Σ** denoting the input set as an alphabet. And it can be defined as follows –

- **Φ** represents the empty set,
- **ε** denotes the null string,
- For each '**a**' in **Σ**, is a regular expression and denotes the set '**a**'.

If "**r**" and "**s**" are regular expressions representing languages L1 and L2, then

- "**r + s**" is same as (**L1** ∪ **L2**)
- **rs** is same as **L1L2** concatenation
- **r*** are equivalent **L1*** closure respectively



The figure demonstrates the easy conversion of **RE to Non-deterministic Finite Automata (NFA)** with epsilon moves, then to Deterministic Finite Automata (DFA),

which can be easily converted to RE.

29. What is the Importance of Closure Properties in Automata Theory?

The **closure property in automata theory** and formal languages refers to a class of languages remaining within that class after certain operations are performed on its members.

In the context of regular languages, they share the same characteristics and limitations when performed through operations like union or concatenation, so we can say regular languages are closed under Union.

Regular languages are closed under union, intersection, concatenation, complement and Kleene closure, etc. In contrast, Deterministic CFLs are closed under complementation, intersection with regulars and inverse homomorphism.

30. What is a Non-deterministic Turing Machine?

A **Non-Deterministic Turing Machine** (TM) has a group of actions for every state and symbol, making transitions non-deterministic. The computation of a TM is a tree of configurations, starting from the start configuration.

A machine like the NTM can make multiple transitions in every step such as transitioning from input symbols to state Qi, Qj, Qk, and so on. The NTM "guesses" the best transition that eventually leads to an accepting state Qf, as no transition is well established in every step. This concept is like a maze, where a machine can always guess the correct choice, finding the quickest way out if there is one.

NTMs are valuable tools for computer scientists to understand computational limits, complexity theory, and efficient algorithm-solving classes. They are particularly useful in studying complexity classes like NP (nondeterministic polynomial time) and NP-complete problems, allowing researchers to reason about problems where the solution can be verified efficiently but may be hard to find initially. However, designing NTM is not practically possible like DTM.

PHP Tutorial

R Tutorial

HTML Tutorial

CSS Tutorial

JavaScript Tutorial

SQL Tutorial

TRENDING TECHNOLOGIES

Cloud Computing Tutorial

Amazon Web Services Tutorial

Microsoft Azure Tutorial

Git Tutorial

Ethical Hacking Tutorial

Docker Tutorial

Kubernetes Tutorial

DSA Tutorial

Spring Boot Tutorial

SDLC Tutorial

Unix Tutorial

CERTIFICATIONS

Business Analytics Certification

Java & Spring Boot Advanced Certification

Data Science Advanced Certification

Cloud Computing And DevOps

Advanced Certification In Business Analytics

Artificial Intelligence And Machine Learning

DevOps Certification

Game Development Certification

Front-End Developer Certification

AWS Certification Training

Python Programming Certification

COMPILERS & EDITORS

Online Java Compiler