1.a) What is Automata Theory? Describe the applications of Finite Automata.

Here is the answer in a **clear, exam-friendly** format:

---

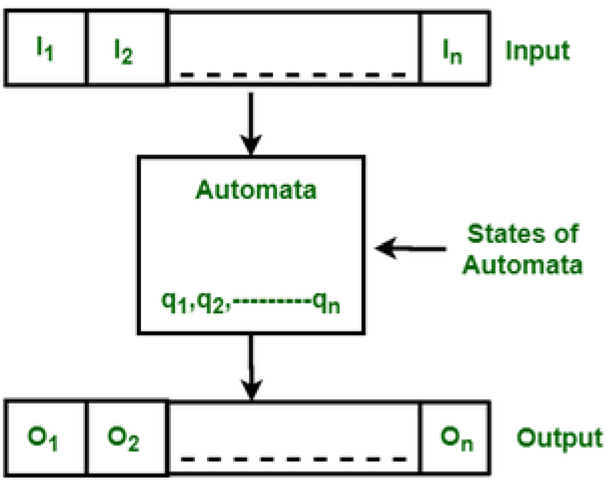
a) What is Automata Theory?

Automata Theory is a branch of theoretical computer science that studies **abstract machines (automata)** and the **problems they can solve**.
It provides mathematical models for understanding **computation, languages, and algorithms**.

It provides a formal framework for understanding the fundamental capabilities and limitations of computation by analyzing mathematical models of how machines process information, such as symbols and strings, according to a set of rules.

In simple words:

👉 **Automata Theory deals with machines that read inputs and decide whether the input is acceptable or not.**
These machines help us formally analyze how computers and programs work.



Applications of Finite Automata (FA)

Finite Automata are widely used because they are simple and efficient. Here are the key applications:

1. Lexical Analysis in Compilers

- FA is used to identify **tokens** such as keywords, identifiers, operators, numbers.
- Every lexical analyzer uses **Deterministic Finite Automata (DFA)**.

2. Pattern Matching / String Searching

- Text editors (e.g., *Find and Replace*), search engines use FA to match patterns quickly.
- Regular expression engines internally use automata.

3. Network Protocol Design

- Communication protocols are modeled using automata to ensure correct sending/receiving sequences.

4. Software for Text Processing

- Spell checkers, text validation tools, DNA sequence analysis use finite automata.

5. Hardware Circuits / Control Systems

- Digital circuits, traffic lights, and control units are modeled as **state machines**.
- Finite automata represent different operational states.

6. Input Validation

- FA is used to check if an input string is valid, such as:
    - Valid **email format**

- Valid **mobile number**
- Valid **password rules**

7. Robotics and AI State Machines

- Robots use automata to decide actions based on current states and inputs.

8. Natural Language Processing (NLP)

- Tokenization, morphological parsing, and finite-state transducers are used in NLP.

One-line simplified list (as you asked earlier):

1. **Lexical analysis** in compilers.
2. **Pattern matching** and text searching.
3. **Network protocol modeling.**
4. **Digital circuit and control system design.**
5. **Input validation** in software.
6. **Speech and NLP processing.**
7. **Robotics finite-state control.**

---

b) Explain the work of lexical analyzer with an on/off switch.

Here is the answer in a **clear, simple, exam-friendly way**, using the **ON/OFF switch analogy**.

---

b) Explain the work of a Lexical Analyzer using an ON/OFF switch analogy

A **lexical analyzer** (also called **lexer** or **scanner**) is the first phase of a compiler.
Its main job is to **read a program's source code character by character** and convert it into **tokens** such as:

- identifiers
- keywords
- numbers
- operators
- punctuation

⭐ Explanation using ON/OFF switch analogy

Think of a lexical analyzer as an **ON/OFF switch–based machine** (a finite automaton):

🔵 1. The switch starts in OFF state

- Initially, the lexical analyzer is in a **starting state (OFF)**.
- It has not recognized any token yet.

🔵 2. When a character comes, the switch turns ON

- When the first character is read, the machine **turns ON** and moves to a new state.

- If the character is a letter → it starts building an **identifier/keyword**.
- If the character is a digit → it starts building a **number**.
- If it is a symbol → it detects an **operator or punctuation**.

🔵 3. The switch remains ON while characters form a valid token

- The lexical analyzer continues reading characters **one by one**.
- As long as they belong to the same token, the machine stays ON and **extends the token**.

Example:
Input: `int`

- Read **i** → switch ON → identifier state
- Read **n** → continue ON
- Read **t** → continue ON

🔵 4. The switch turns OFF when the token ends

- When a character is found that **cannot** be part of the current token (like a space, operator, or newline),
  the lexical analyzer **turns OFF** and finalizes the token.

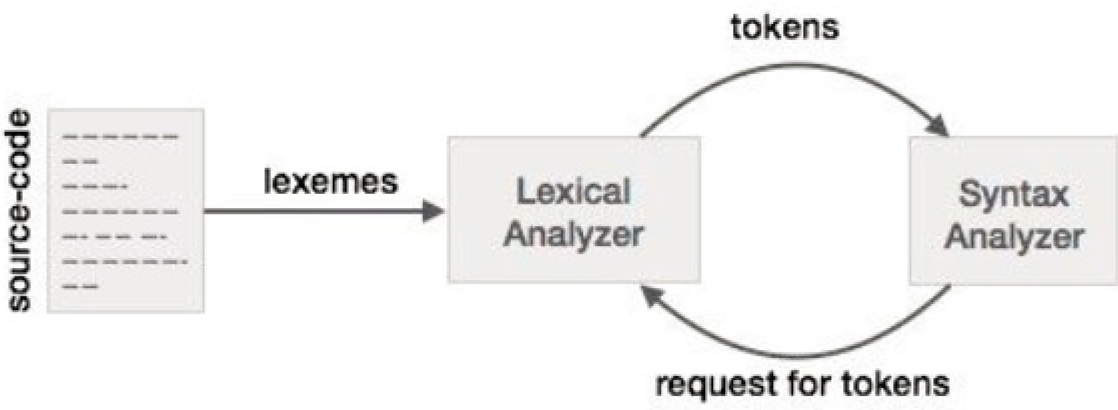
Example:

```
int x;
```

After reading `int`, the space turns the switch OFF, and the token **"int (keyword)"** is produced.

🔵 5. The switch turns ON again for the next token

- Now it starts reading the next token (`x`, then `;`) the same way.



⭐ Summary (Exam-friendly 5 lines)

- A lexical analyzer reads source code character by character.
- Using a finite automaton (like an ON/OFF switch), it switches **ON** when a valid token starts.
- It stays **ON** while characters form a valid token.
- It switches **OFF** when the token ends and outputs the token.
- Then it turns ON again to process the next token.

---

c) The Chomsky Hierarchy is a containment hierarchy of classes of formal languages. Describe with suitable figure(s).

The **Chomsky Hierarchy**, proposed by Noam Chomsky (1956), classifies formal languages into **four types** based on the **power of their grammars** and the **machines** that recognize them.

It is a **containment hierarchy**, meaning:

👉 **Type-3 ⊆ Type-2 ⊆ Type-1 ⊆ Type-0**

Higher types are more powerful and can generate more complex languages.

---

1. Type–0: Unrestricted Grammars

- Most powerful grammar.
- Productions have **no restrictions**.
- Recognized by **Turing Machines**.
- Can describe **any computable language**.

Example:

$\alpha \rightarrow \beta$ (α and β can be any combination of terminals & non-terminals)

---

2. Type–1: Context-Sensitive Grammars (CSG)

- Productions are of the form:
  $\alpha A \beta \rightarrow \alpha \gamma \beta$, where $|\gamma| \geq 1$ (length increases or stays same).
- Recognized by **Linear Bounded Automata (LBA)**.
- Used in natural language modeling and some programming constructs.

---

3. Type–2: Context-Free Grammars (CFG)

- Productions of the form:
  A → γ
- Recognized by **Pushdown Automata (PDA)**.
- Used in **programming languages, syntax analysis, expression parsing**.

---

4. Type–3: Regular Grammars

- Productions of the form:
  A → aB or **A → a**
- Recognized by **Finite Automata (FA)**.
- Used in **lexical analysis, text search, pattern matching**.

⭐ Chomsky Hierarchy – Containment Diagram (Very Important Figure)

Draw this in your exam:

```
+-------------------------------------------+
|       Type–0: Unrestricted Languages     |
|   (Recognized by Turing Machine)          |
|   +----------------------------------+   |
|   |  Type–1: Context-Sensitive        |   |
|   |  (Linear Bounded Automata)        |   |
|   |   +--------------------------+   |   |
|   |   |  Type–2: Context-Free     |   |   |
|   |   |  (Pushdown Automata)      |   |   |
|   |   |    +-----------------+   |   |   |
|   |   |    | Type–3: Regular  |   |   |
|   |   |    | (Finite Automata)|   |
|   |   |    +-----------------+   |
|   |   +--------------------------+   |
|   +----------------------------------+   |
+-------------------------------------------+
```

You can also draw a simple nested-circle figure:

Nested Circle Figure

```
Type–0 (Unrestricted)
   └── Type–1 (Context-Sensitive)
         └── Type–2 (Context-Free)
               └── Type–3 (Regular)
```

⭐ Short Exam Summary (5–6 lines)

- The Chomsky Hierarchy classifies formal languages into four types based on grammatical restrictions and computational power.
- **Type-0**: Unrestricted grammars (Turing Machines).
- **Type-1**: Context-Sensitive grammars (Linear Bounded Automata).
- **Type-2**: Context-Free grammars (Pushdown Automata).
- **Type-3**: Regular grammars (Finite Automata).
- It is a containment hierarchy:
  Regular ⊆ Context-Free ⊆ Context-Sensitive ⊆ Unrestricted.

Here is a **clear, simple, exam-friendly** explanation of **Theorems, Lemmas, and Corollaries**, exactly suitable for short/long questions.

---

d) Analyze the terms: Theorems, Lemmas, and Corollaries

In mathematics and theoretical computer science, results are organized using **theorems**, **lemmas**, and **corollaries** to make proofs structured and easy to understand.

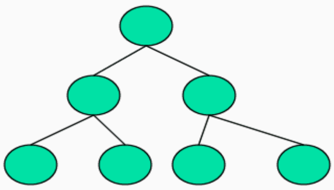
Well, they are basically just **facts**: some result that has been arrived at.

- A Theorem is a **major** result
- A Corollary is a theorem that **follows on** from another theorem
- A Lemma is a **small** results (less important than a theorem)

We typically refer to:

- A major result as a "**theorem**"
- An intermediate result that we show to prove a larger result as a "**lemma**"
- A result that follows from an already proven result as a "**corollary**"

An example:
Theorem: *The height of an n-node binary tree is at least floor(lg n)*
Lemma: *Level i of a perfect binary tree has 2^i nodes.*
Corollary: *A perfect binary tree of height h has $2^{h+1}-1$ nodes.*



1. Theorem

Definition:

A **theorem** is a major, important statement that has been **logically proven** using axioms, definitions, and previously established results.

Characteristics:

- Central and significant result.
- Often requires a detailed proof.
- Widely applicable and meaningful.

Example:

"The intersection of two regular languages is regular."

---

2. Lemma

Definition:

A **lemma** is a *supporting result*—a small, auxiliary statement proved mainly to help establish a **theorem**.

Characteristics:

- Not the main result, but simplifies the proof of a theorem.
- Breaks down a complex proof into smaller pieces.
- Often easier to prove than the theorem itself.

Example:

A lemma used inside the proof of the Pumping Lemma for regular languages.

---

3. Corollary

Definition:

A **corollary** is a result that **follows directly** from a previously proven theorem, usually with little or no additional proof.

Characteristics:

- Immediate consequence of a theorem.
- Supports, extends, or gives a quick application of a theorem.
- Simple and short.

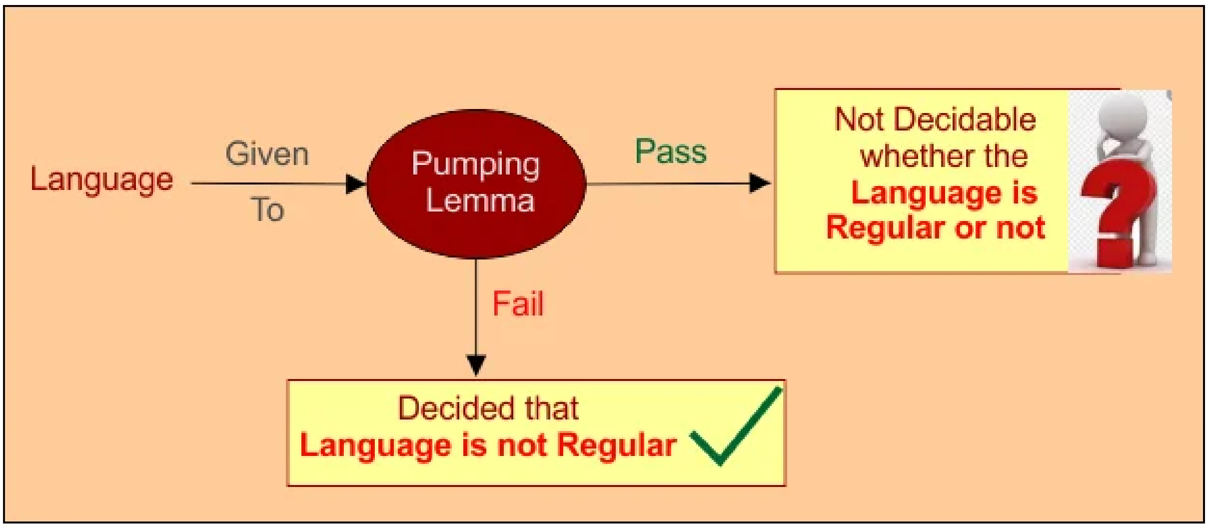
Example:

From the theorem “Every DFA has an equivalent NFA,” a corollary is:
“Every regular language can be accepted by an NFA.”

Summary (5 lines for exam)

- A **theorem** is a major proven statement and the main result.
- A **lemma** is a smaller supporting result used to help prove a theorem.
- A **corollary** is an immediate consequence derived from a theorem.
- Lemmas → help prove Theorems → give Corollaries.
- These terms make proofs structured, logical, and easier to understand.

2.a) Describe the basic functionalities of pumping lemma.



Pumping Lemma Test (PLT) - For Regular Language

The **Pumping Lemma** is a fundamental property of **regular languages** that helps us understand their structure. Its basic functionalities are:

1. To prove a language is *not* regular

- The main use of the pumping lemma is to **show that a language is not regular** by contradiction.
- If a language fails to satisfy the pumping lemma, then it **cannot** be regular.

2. To describe a repetitive structure in long strings

- It states that any sufficiently long string in a regular language can be **divided into three parts** (x, y, z) such that:
    - The middle part **y can be repeated ("pumped")** any number of times (0,1,2, …).
    - The resulting string will still be in the language.

3. To express the idea of finite memory of finite automata

- Pumping lemma captures the fact that a **finite automaton has limited states**.
- If a string is longer than the number of states, it must repeat states — creating a loop that can be pumped.

4. To provide a necessary condition for regularity

- If a language is regular, it **must satisfy** the pumping lemma.
- But satisfying the lemma *does not guarantee* regularity — it's a **necessary but not sufficient** condition.

For a regular language L, there exists a pumping length p such that any string s with $|s| \geq p$ can be split into

$$s = xyz$$

satisfying **three conditions**:

1. $|xy| \leq p$

The first two parts (**x** and **y**) are within the first **p** characters.

2. $|y| > 0$

The **y-part is not empty** (it must contain at least one character).

3. $xy^iz \in L$ **for all** $i \geq 0$

Pumping (repeating or removing) **y** any number of times must keep the new string **inside the language**.

Short 4–5 line exam answer (very easy)

- The pumping lemma provides a property that all **regular languages** must satisfy.
- It says long strings can be split into three parts x,y,zx, y, zx,y,z where **y can be repeated** any number of times.
- This shows the repetitive loop behavior of finite automata.
- It is mainly used to **prove that certain languages are not regular** by showing they violate this property.

b) Use the pumping lemma to prove that the language is not context free.

$$A = \{ 0^{2n} 1^{3n} 0^{n} \mid n \geq 0 \}$$

To prove that the language

$$A = \{0^{2n}1^{3n}0^n \mid n \geq 0\}$$

is **not context-free**, we will use the **Pumping Lemma for Context-Free Languages**.

Pumping Lemma for CFLs (Informal Statement)

If A is a context-free language, then there exists a constant p (the **pumping length**) such that any string $s \in A$ with $|s| \geq p$ can be split into five parts:

$$s = uvwxy$$

such that:

1. $|vwx| \leq p$
2. $|vx| \geq 1$ (i.e., v or x is not empty)
3. For all $i \geq 0$, the string $uv^iwx^iy \in A$

Proof by Contradiction

Assume:

The language A is context-free.

Then by the pumping lemma, there exists a pumping length p.

Step 1: Choose a String in A

Let's pick:

$$s = 0^{2p}1^{3p}0^p$$

Clearly, $s \in A$, with $n = p$. Also, $|s| = 2p + 3p + p = 6p \geq p$, so the pumping lemma applies.

Step 2: Split the String

The pumping lemma says:

$$s = uvwxy$$

with:

- $|vwx| \leq p$
- $|vx| \geq 1$
- $uv^i wx^i y \in A$ for all $i \geq 0$

Step 3: Analyze vwx

Since $|vwx| \leq p$, the substring vwx can only span **one** of the following blocks (can't span all of them as each block is ≥ p long):

1. The **first block of 0s** (i.e., the 0^{2p})
2. The **block of 1s** (i.e., the 1^{3p})
3. The **last block of 0s** (i.e., the 0^p)

We handle each case to show a contradiction.

Case 1: vwx is within the first block of 0s

- Pumping v and x changes the number of 0s in the first block.
- New string after pumping: uv^2wx^2y will have **more than** $2n$ **0s** in the first block, but the other blocks won't change proportionally.
- So the string won't be of the form $0^{2n}1^{3n}0^n$.
- **Contradiction**.

Case 2: vwx is within the 1s

- Pumping v and x changes the number of 1s.
- New string: number of 1s is no longer $3n$, but the first and last blocks remain unchanged.
- Not in A.
- **Contradiction**.

Case 3: vwx is within the last block of 0s

- Pumping adds or removes 0s from the last block only.
- The number of 0s in the last block will not match the required n, making the structure invalid.
- **Contradiction**.

c) Define push down automata with example.

A **Pushdown Automaton (PDA)** is a type of automaton that uses a stack as an additional memory structure. It is an extension of Finite Automata (FA) that allows it to recognize a broader class of languages — specifically, **context-free languages (CFLs)**.

A **Pushdown Automaton (PDA)** is a type of **automaton** that uses a **stack** in addition to its finite control.
It is more powerful than a **Finite Automaton (FA)** but less powerful than a **Turing Machine**.

PDA is mainly used to recognize **Context-Free Languages (CFLs)**.

Formal Definition:

A **PDA** is defined as a 7-tuple:M=(Q,Σ,Γ,δ,q0,Z0,F)

Where:

Symbol	Meaning
Q	Finite set of states
Σ	Input alphabet
Γ	Stack alphabet
δ	Transition function: δ(q, a, X) → (p, γ) where: – q = current state – a = current input symbol (or ε) – X = top of stack symbol – γ = string to replace X on stack
q_0	Start state
Z_0	Initial stack symbol
F	Set of accepting (final) states

Working Principle:

- PDA reads input from left to right.

- It can **push** or **pop** symbols on the stack.
- The **stack** provides **memory**, allowing PDA to recognize patterns like matching parentheses.
- PDA can accept input by:
    1. **Final state**, or
    2. **Empty stack.**

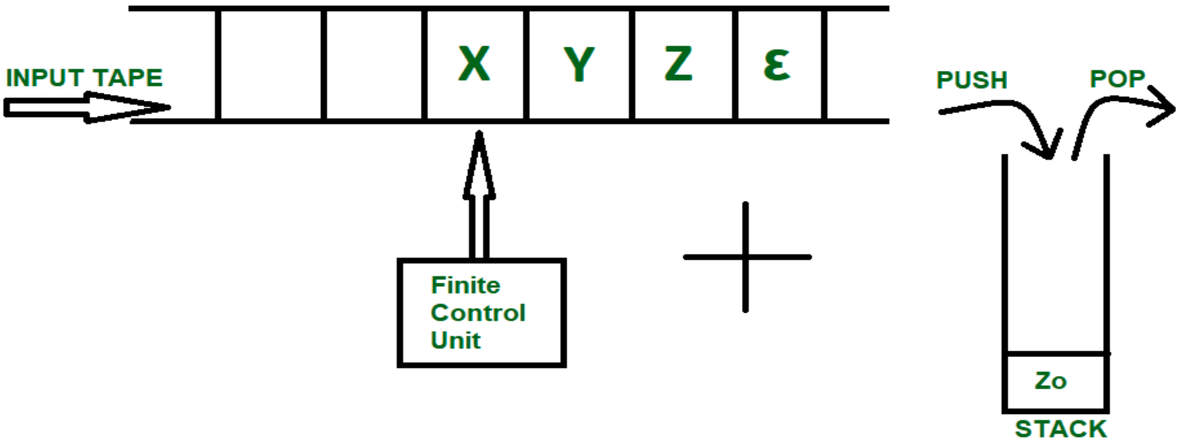


Diagram (conceptually):
Input Tape → a b b a
Stack → Z0 ↓
States → q0, q1, qf

Transition example:

δ(q0, a, Z0) = (q1, AZ0)
δ(q1, b, A) = (q1, ε)

Meaning:

- When reading a, push A on stack.
- When reading b, pop A from stack.

Example:

PDA for language:
[
L = { a^n b^n \ | \ n ≥ 0 }
]
Steps:

- For each a, push symbol (say X) on stack.
- For each b, pop one X.
- Accept if stack becomes empty at end.

The stack allows the PDA to remember an unlimited amount of information, making it suited for languages with nested structures like parentheses.

Example: PDA for language $L = \{a^n b^n \mid n \geq 1\}$

This language has equal number of a's and b's — not regular but context-free.

How the PDA works

1. For each **a** read, push **A** onto stack.
2. For each **b** read, pop **A** from stack.
3. Accept if stack becomes empty after all input is processed.

Transitions

- $\delta(q, a, Z) = (q, AZ)$
- $\delta(q, a, A) = (q, AA)$ → push for each 'a'
- $\delta(q, b, A) = (q, \epsilon)$ → pop for each 'b'
- Accept when stack becomes empty.

Simple Example (Step-by-step for “aaabbb”)

Input: **aaabbb**

Step	Action	Stack
a	push A	A
a	push A	AA
a	push A	AAA
b	pop A	AA
b	pop A	A
b	pop A	ε (empty)

Stack empty → **Accepted** ✔

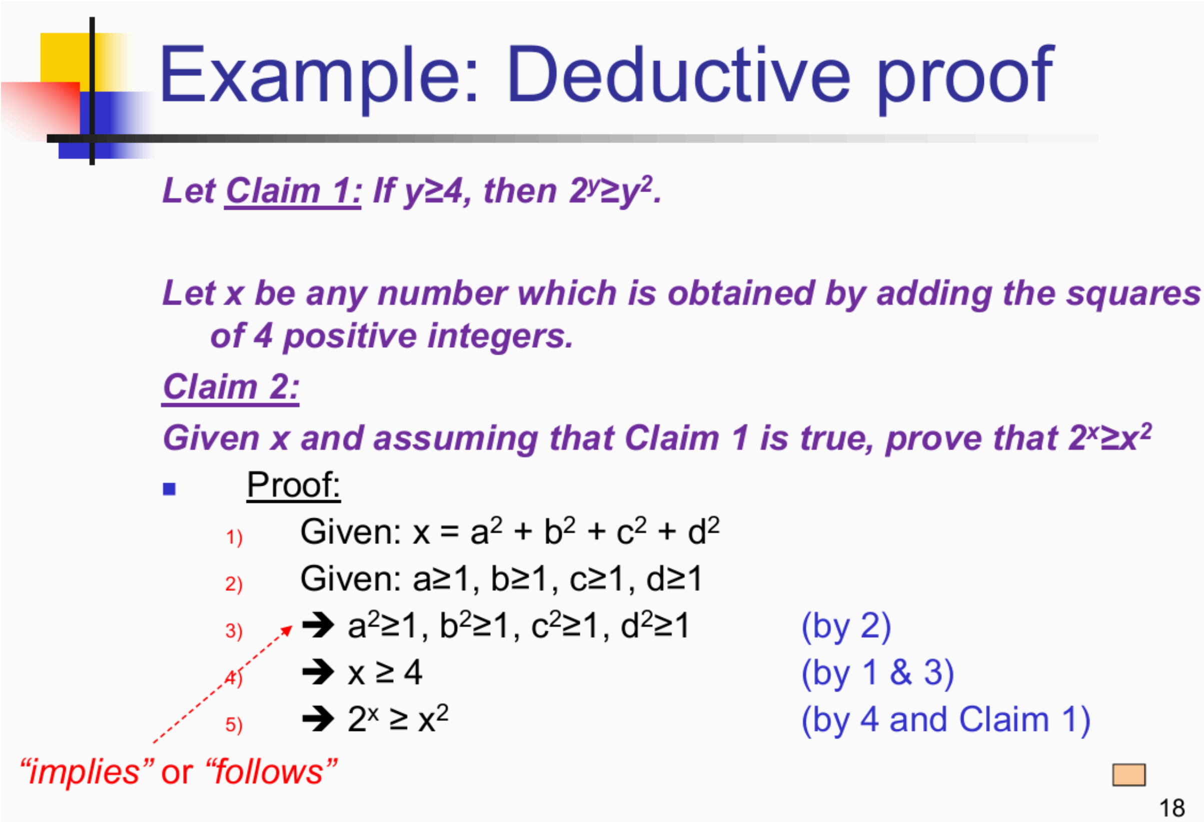
3.a) Describe Deductive proof. Prove that, if x is the sum of the squares of four positive integers, then

$$2^x \geq x^2.$$

Deductive proof is a method in which we start from known facts, definitions, axioms, or previously proven theorems, and then use logical reasoning to reach a conclusion.

Key features of deductive proof

- Moves from **general statements → specific conclusion**
- Each step follows by **logical necessity**
- If the premises are true, the conclusion must be true
- Used in mathematics to prove theorems with certainty



We must check whether

$$2^x = x^2$$

✔ Test with smallest possible values

Let

$$a = b = c = d = 1$$

Then

$$x = 1^2 + 1^2 + 1^2 + 1^2 = 4$$

Check the equation:

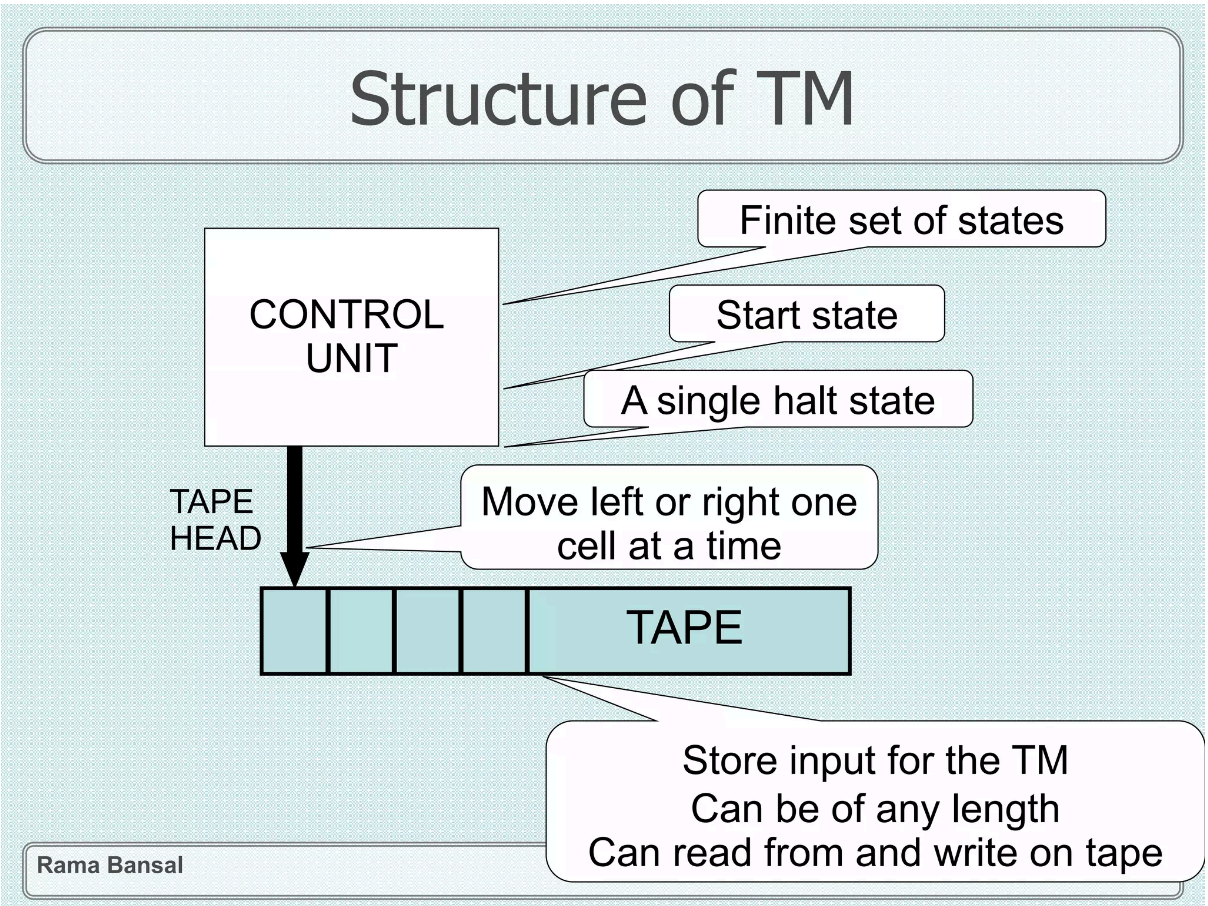
$$2^4 = 16$$

$$4^2 = 16$$

So **for this specific case**, the equation holds.

---

b) Briefly explain the configuration of a TM.



A **configuration of a Turing Machine** describes the **complete status** of the machine at any moment during its computation.
It tells us *everything needed* to continue the computation from that point.

A TM configuration consists of three main parts:

1. Current state of the TM

- The state in which the machine is currently operating (e.g., q0,q1,q2,…)

2. Tape contents

- All symbols written on the tape, including:
  - Input symbols
  - Blank symbols

- ○ Symbols that were overwritten during computation

3. Position of the tape head

- The exact location where the head is pointing or scanning a symbol.

Representation of a configuration

A configuration is usually written as:

α , q ,β

Where:

- α: symbols to the **left** of the head
- q: current state
- β: current symbol under the head and the rest of the tape to the **right**

c) Let x be a real number. Then prove that, [x] = [x] if and only if x is an integer.

c) Proof: $\lfloor x \rfloor = \lceil x \rceil$ iff x is an integer

We must prove the statement in **two directions**:

(⟹) If $\lfloor x \rfloor = \lceil x \rceil$, then x is an integer

Assume

$$\lfloor x \rfloor = \lceil x \rceil.$$

Let this common value be n.
So,

$$\lfloor x \rfloor = n \quad \text{and} \quad \lceil x \rceil = n.$$

By definition of floor:

$$n = \lfloor x \rfloor \leq x < n + 1.$$

By definition of ceiling:

$$n - 1 < x \leq n = \lceil x \rceil.$$

Combining both:

$$n \leq x \leq n.$$

So

$$x = n.$$

Since n is an integer, x must also be an **integer**.

Thus, if floor = ceiling, x is an integer. ✔

(⇐) If x is an integer, then $\lfloor x \rfloor = \lceil x \rceil$

Let $x = n$, where n is an integer.

- The floor of an integer is the integer itself:

$$\lfloor n \rfloor = n.$$

- The ceiling of an integer is also the integer:

$$\lceil n \rceil = n.$$

Therefore,

$$\lfloor x \rfloor = n = \lceil x \rceil.$$

Thus, if x is an integer, floor = ceiling. ✔

Final Conclusion

$$\boxed{\lfloor x \rfloor = \lceil x \rceil \iff x \in \mathbb{Z}}$$

The floor and ceiling of a real number are equal **if and only if** the number itself is an integer.

d) For all n ≥ 0, prove that;

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}.$$

✅ Proof by Mathematical Induction

We will prove that for all $n \geq 0$:

$$1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

Step 1: Base Case (n = 0)

Left-hand side (LHS):

$$\sum_{i=1}^{0} i^2 = 0$$

Right-hand side (RHS):

$$\frac{0(0+1)(2 \cdot 0 + 1)}{6} = 0$$

Thus,

$$\text{LHS} = \text{RHS}$$

✔ Base case is true.

Step 2: Induction Hypothesis

Assume the formula is true for some $n = k$.

That is, assume:

$$\sum_{i=1}^{k} i^2 = \frac{k(k+1)(2k+1)}{6}$$

This is our **induction hypothesis**.

Step 3: Induction Step

We must prove that it also holds for $n = k + 1$:

$$\sum_{i=1}^{k+1} i^2 = ?$$

Start with:

$$\sum_{i=1}^{k+1} i^2 = \left(\sum_{i=1}^{k} i^2\right) + (k+1)^2$$

Use induction hypothesis:

$$= \frac{k(k+1)(2k+1)}{6} + (k+1)^2$$

Factor out $(k + 1)$:

$$= (k+1)\left[\frac{k(2k+1)}{6} + (k+1)\right]$$

Bring to common denominator 6:

$$= (k+1)\left[\frac{k(2k+1) + 6(k+1)}{6}\right]$$

Expand numerator:

$$k(2k+1) + 6(k+1) = 2k^2 + k + 6k + 6 = 2k^2 + 7k + 6$$

Factor the quadratic:

$$2k^2 + 7k + 6 = (2k+3)(k+2)$$

Thus,

$$\sum_{i=1}^{k+1} i^2 = (k+1)\left[\frac{(2k+3)(k+2)}{6}\right]$$

Rewrite in standard form:

$$= \frac{(k+1)(k+2)(2k+3)}{6}$$

Replace $k+1$ with n:

$$= \frac{n(n+1)(2n+1)}{6}$$

✔ Formula holds for $n = k + 1$.

4.a

advantages of DFA (Deterministic Finite Automata) with examples:

Advantages of DFA

1. **Simplicity and Determinism**
   DFA has a simple and deterministic structure, meaning for each state and input symbol, there is exactly **one next state**. This makes DFA predictable and easy to implement in software and hardware.
   Example: A DFA that recognizes binary strings ending with `01` always knows exactly which state to go next for every input, without ambiguity.

2. **Fast Recognition**
   Since DFA reads each input symbol **exactly once** and moves through states deterministically, it can recognize strings in **linear time O(n)**, where n is the length of the input string.
   Example: Checking if a password contains a specific pattern like `abc` can be efficiently done with a DFA in one pass.

3. **Easy to Implement**
   DFA can be implemented using simple **transition tables** or arrays, which makes it practical for programming and designing hardware circuits.
   Example: Lexical analyzers in compilers use DFAs to identify keywords, operators, and identifiers.

4. **Closure Properties**
   DFA languages are closed under **union, intersection, and complement**. This allows combining DFAs to handle more complex

patterns.
Example: A DFA for strings ending with `0` and another DFA for strings starting with `1` can be combined to accept strings that satisfy both conditions.

5. **No Backtracking Required**
Unlike NFA (Nondeterministic Finite Automata), DFA does **not need to try multiple paths**; it always has a unique next move, making it faster and more memory-efficient for execution.

✅ **Summary Example:**
DFA to recognize strings over `{0,1}` ending with `01`:

- States: `q0` (start), `q1`, `q2` (final)
- Input: `0` or `1`
- Transition:
    - `q0` → `0` → `q1`, `q0` → `1` → `q0`
    - `q1` → `0` → `q1`, `q1` → `1` → `q2`
    - `q2` → `0` → `q1`, `q2` → `1` → `q0`

This DFA **quickly and deterministically** accepts strings ending with `01`, showing all the advantages above.

b) Construct a DFA for the following language:

Let, Σ = {0, 1}, L = {w|w is a binary string that has even number of 1s and even number of 0s}.

L = { w has an even number of 1s and even number of 0s} }

Alphabet: sigma = {0, 1})

Idea

To track both:

- **Even or odd number of 0s**, and
- **Even or odd number of 1s**

We need to *remember* the state of both — so we need **4 states**:

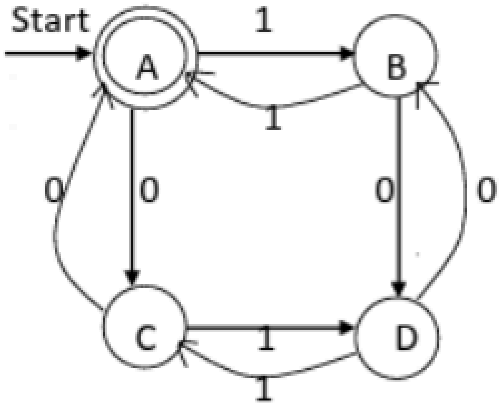
State	Meaning
q_0	Even 0s, Even 1s
q_1	Odd 0s, Even 1s
q_2	Even 0s, Odd 1s
q_3	Odd 0s, Odd 1s

State Transitions

Current State	Input = 0	Input = 1
q_0	$\rightarrow q_1$	$\rightarrow q_2$
q_1	$\rightarrow q_0$	$\rightarrow q_3$
q_2	$\rightarrow q_3$	$\rightarrow q_0$
q_3	$\rightarrow q_2$	$\rightarrow q_1$

DFA Diagram

Transition diagram:



Transition table:

δ	0	1
→ *A	C	B
B	D	A
C	A	D
D	C	B

- (q_0) is the **start** and also the **accepting** state (since both counts start at even)
- Only (q_0) is accepting, since it represents **even 0s and even 1s**

Final Answer: Summary

- **States:** ($Q = \{q_0, q_1, q_2, q_3\}$)
- **Start State:** (q_0)
- **Accepting State:** ($\{q_0\}$)

- **Alphabet:** ({0, 1})
- **Transition Function:** As shown in the table above

This DFA recognizes all binary strings that contain **an even number of 0s and an even number of 1s**.

b) Construct an NFA for the following: Strings where the first symbol is present somewhere later on at least once.[6]

To construct an **NFA** for the language:

> **Strings where the first symbol is present somewhere later on at least once**

This means:

- The **first character** of the input (either `0` or `1`) must **reappear later** in the string.
- Examples:
  - Accepted: `00`, `0110`, `1001`, `010110`
  - Rejected: `01`, `10` (because the first symbol does not repeat later)

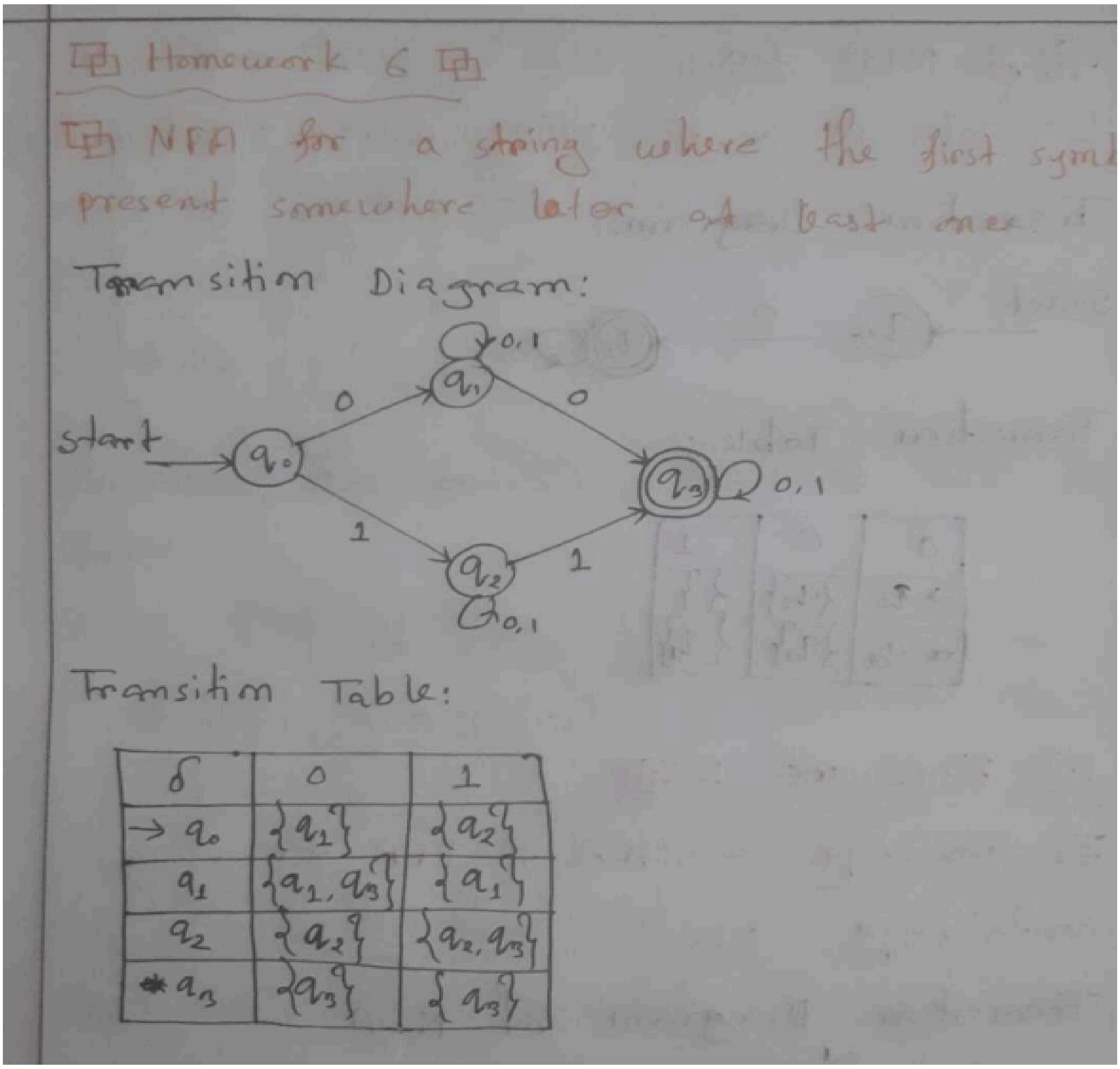
💡 **Idea:**

We can design the NFA using nondeterminism:

1. Read the first symbol (`0` or `1`) and remember it using states.
2. Then move through the rest of the string.
3. If we find the same symbol again, accept.

States:

- `q0`: Start state (before reading first character)
- `q1`: Remember first symbol was `0`
- `q2`: Remember first symbol was `1`
- `qf`: Accepting state (once the first symbol is seen again)
- `qd`: Dead state (optional — not always necessary in NFA)



Homework 6

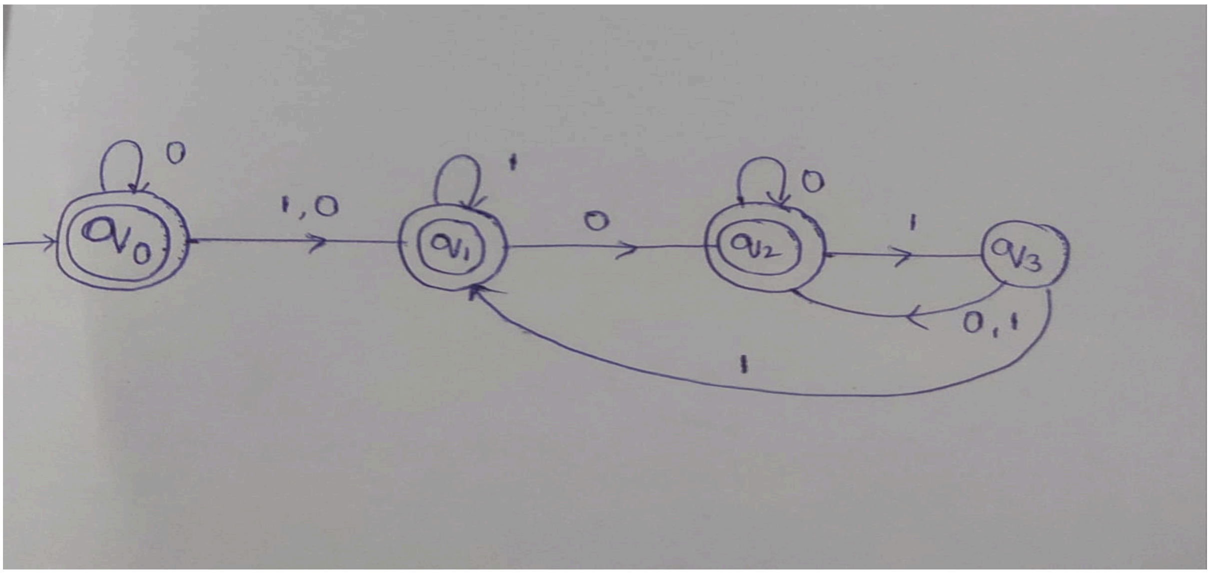
NFA for a string where the first symb[ol] present somewhere later at least once

Transition Diagram:

Transition Table:

δ	0	1
→ q_0	$\{q_1\}$	$\{q_2\}$
q_1	$\{q_1, q_3\}$	$\{q_1\}$
q_2	$\{q_2\}$	$\{q_2, q_3\}$
* q_3	$\{q_3\}$	$\{q_3\}$

5.a) Build an NFA for the following language: L = {w | w ends in 101}



$$L = \{w \mid w \text{ ends in } 101\}$$

Step 1: Understanding the language

- The strings can be anything, as long as the **last three symbols are** `101`.
- So, before the last `101`, the NFA can read **any combination of 0s and 1s.**

Step 2: Define states

We can define states to track the progress toward the ending `101`:

State	Meaning
q0	Start state, haven't seen anything for `101` yet
q1	Saw `1` (possible start of ending `101`)
q2	Saw `10` (middle of ending `101`)
q3	Saw `101` → final/accepting state

States

- q0 = Start state
- q1 = Saw `1` (possible start of ending `101`)
- q2 = Saw `10`
- q3 = Saw `101` (accepting state)

Transition Table

Current State	Input 0	Input 1
q0	{q0}	{q1}
q1	{q2}	{q1}
q2	{q0}	{q3}
q3	{q2}	{q1}

q3 is the **accepting state** because we have successfully read a string ending with `101`.

b) Advantages and Caveats of NFA (Nondeterministic Finite Automaton)

Advantages of NFA

1. **Simplicity of Design:**
   Easier to design than DFA for some complex languages because nondeterminism allows multiple choices.

2. **Fewer States:**
   Often requires **fewer states** than a DFA for the same language.

3. **Flexible Transitions:**
   Can use **ε-transitions** (moves without input), which makes modeling certain patterns easier.

4. **Good for Theoretical Analysis:**
   Useful in proofs and conversions (e.g., from regex to automata).

5. **Expressiveness:**
   Can represent the same languages as DFA (all regular languages) but more succinctly.

Caveats / Disadvantages of NFA

1. **Nondeterminism Not Directly Implementable:**
   Computers cannot directly execute nondeterministic choices; must simulate with DFA or backtracking.

2. **Conversion to DFA Can Cause State Explosion:**
   Converting NFA to DFA may result in exponentially more states (subset construction).

3. **Complexity in Simulation:**
   Checking acceptance requires **tracking multiple paths**, which can be less efficient.

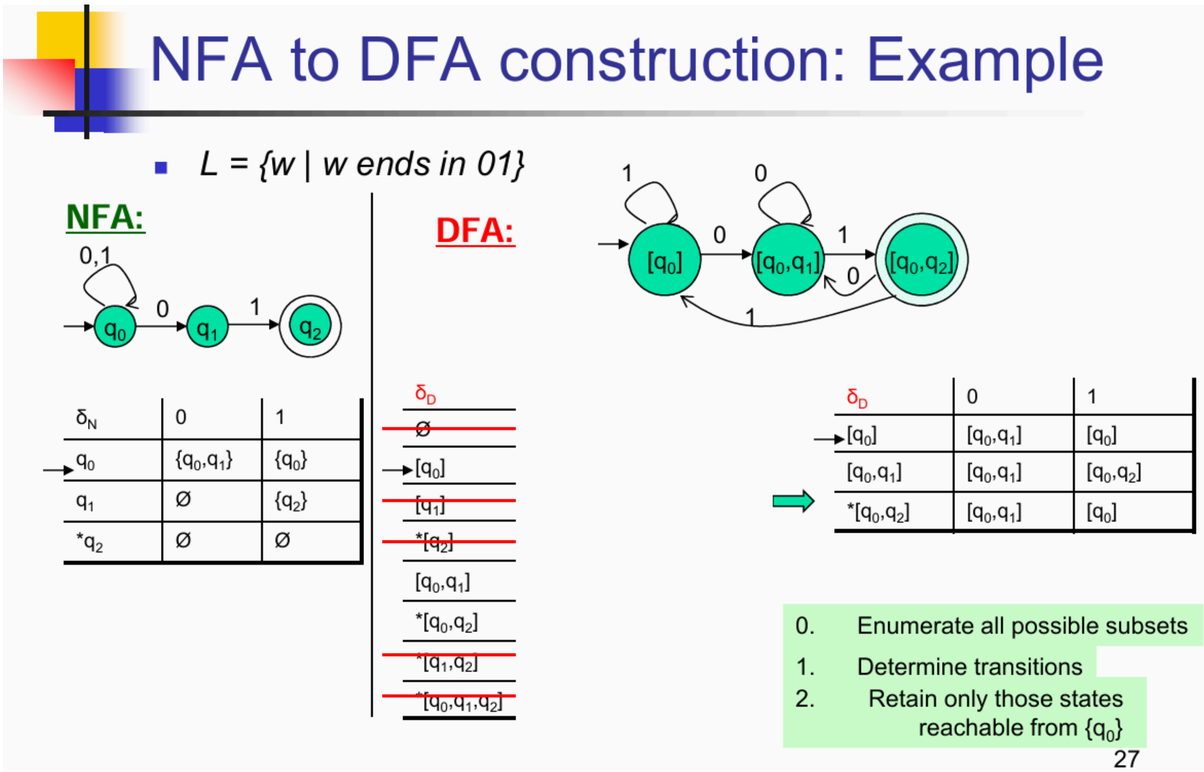
4. **Ambiguity in Transitions:**
   Multiple transitions for the same input can complicate analysis or implementation.

5. Limited Power: Even though NFA is more flexible, it still has some limitations and cannot handle very complex patterns or languages.

✅ **Summary:**

- **Advantages:** Easier design, fewer states, uses ε-moves, concise representation.
- **Caveats:** Harder to implement directly, possible exponential growth when converted to DFA, multiple paths to track.

c) Convert the NFA from Question 5(a), to **DFA**. L={w|w ends in 101},Σ={0,1}



Same like this

6.

a. necessities of explicit ε-transitions in finite automata:

1. **Simplify NFA design** – allows state changes without consuming input.

2. **Facilitate union of languages** – connect NFAs for L1 ∪ L2 easily.

3. **Facilitate concatenation** – connect NFAs for L1L2 without extra symbols.

4. **Handle optional symbols/strings** – model patterns like a? naturally.

5. **Enable repetition/closure** – implement Kleene star (L*) easily.

6. **Simplify modular construction** – build complex automata from smaller pieces.

7. **Assist DFA conversion** – ε-closure simplifies subset construction.

8. **Reduce number of explicit transitions** – fewer edges in NFA graph.

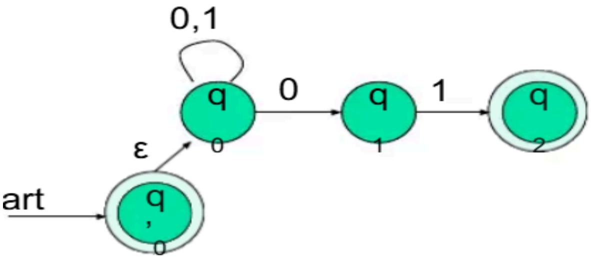
9. **Allow flexible backtracking** – can “try” multiple paths without input.

b) Build an e-NFA for the following language:

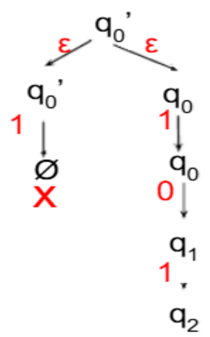
L= ={w|w is empty, or if non-empty will end in 11}

Example of an ε-NFA

L = {w | w is empty, or if non-empty will end in 01}



Simulate for w=101:



Δ_E	0	1	ε
→ $^*q'_0$	Ø	Ø	$\{q'_0, q_0\}$
q_0	$\{q_0, q_1\}$	$\{q_0\}$	$\{q_0\}$
q_1	Ø	$\{q_2\}$	$\{q_1\}$
$^*q_2$	Ø	Ø	$\{q_2\}$

ECLOSE(q'_0)

ECLOSE(q_0)

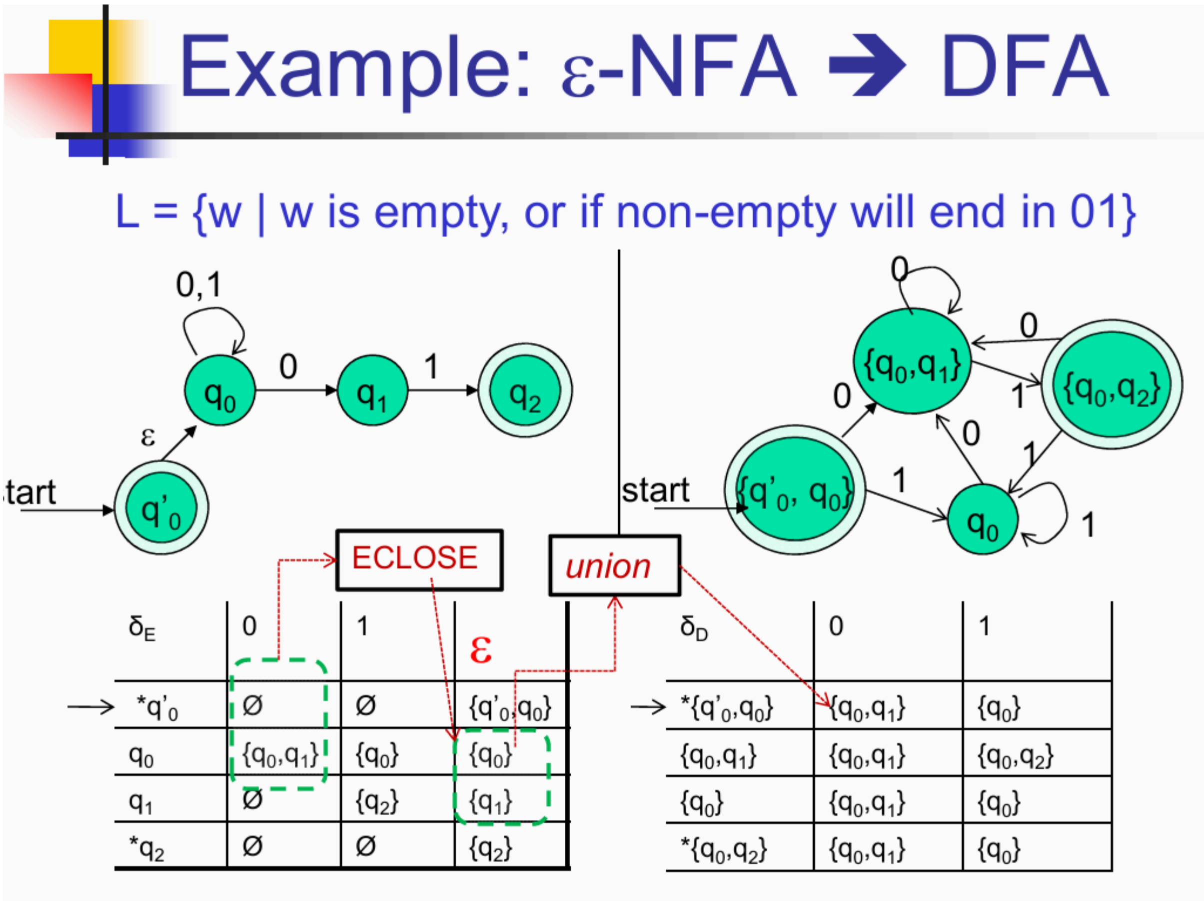
ε-closure of a state q, ***ECLOSE(q)***, is the set of all states (including itself) that can be *reached* from q by repeatedly making an arbitrary number of ε-transitions.

c) Convert ε-NFA to DFA based on Question 6(b)

Question 6(b):

Build an e-NFA for the following language:

L= ={w|w is empty, or if non-empty will end in 11}



Same like this

7.a) What is regular expression? Describe the operations of regular expressions.

Here's a clear explanation:

---

a) What is a Regular Expression (RE)?

A **Regular Expression (RE)** is a formal way to describe a **set of strings (language) over an alphabet** using symbols and operators.

- It is widely used in **pattern matching**, **lexical analysis**, and defining **regular languages**.
- Example: Over the alphabet (\Sigma = {0,1}), the RE `0*1` represents all strings with **zero or more 0s followed by a 1**: `1, 01, 001, 0001…`

Operations of Regular Expressions

1. **Union (+ or ∪)**
   - Represents **choice** between two patterns.
   - Example: `a + b` represents `{a, b}`
2. **Concatenation (·)**
   - Represents **sequence** of patterns.
   - Example: `ab` represents `{ "ab" }`
3. **Kleene Star (*)**
   - Represents **zero or more repetitions** of a pattern.
   - Example: `a*` represents `{ ε, a, aa, aaa… }`
4. **Kleene Plus (+)**
   - Represents **one or more repetitions** of a pattern.
   - Example: `a+` represents `{ a, aa, aaa… }`

5. **Optional (?)**
   - Represents **zero or one occurrence** of a pattern.
   - Example: `a?` represents `{ ε, a }`
6. **Parentheses ( )**
   - Used for **grouping** to control order of operations.
   - Example: `(a+b)c` represents `{ac, bc}`

Summary Example

- Alphabet: `{0,1}`
- RE: `(0+1)*11` → All strings over `{0,1}` that **end with `11`**.

b) Prove that if L=L(A) for some DFA A, then there is a regular expression R such that L=L(R).

Statement:

If $L = L(A)$ for some DFA A, then there exists a regular expression R such that $L = L(R)$.

Proof (Short Version):

1. Let DFA $A = (Q, \Sigma, \delta, q_0, F)$.
2. Convert A to a **Generalized NFA (GNFA)**:
   - Transitions labeled with **regular expressions**.
   - Add a **new start** and **new final state**.
3. **Eliminate intermediate states** one by one using:

$$R_{ij}^{new} = R_{ij} + R_{ir}(R_{rr})^* R_{rj}$$

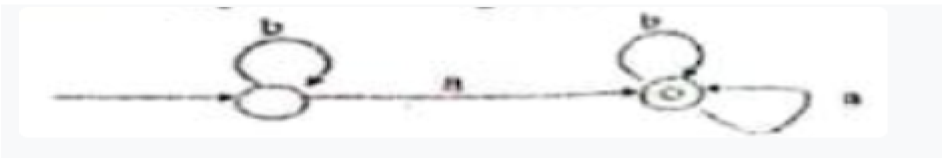
4. After all eliminations, the transition from start to final state is labeled with a **single regular expression** R.
5. Thus, $L(A) = L(R)$.

✅ **Conclusion:** Every DFA recognizes a **regular language**, so a corresponding **regular expression always exists**.

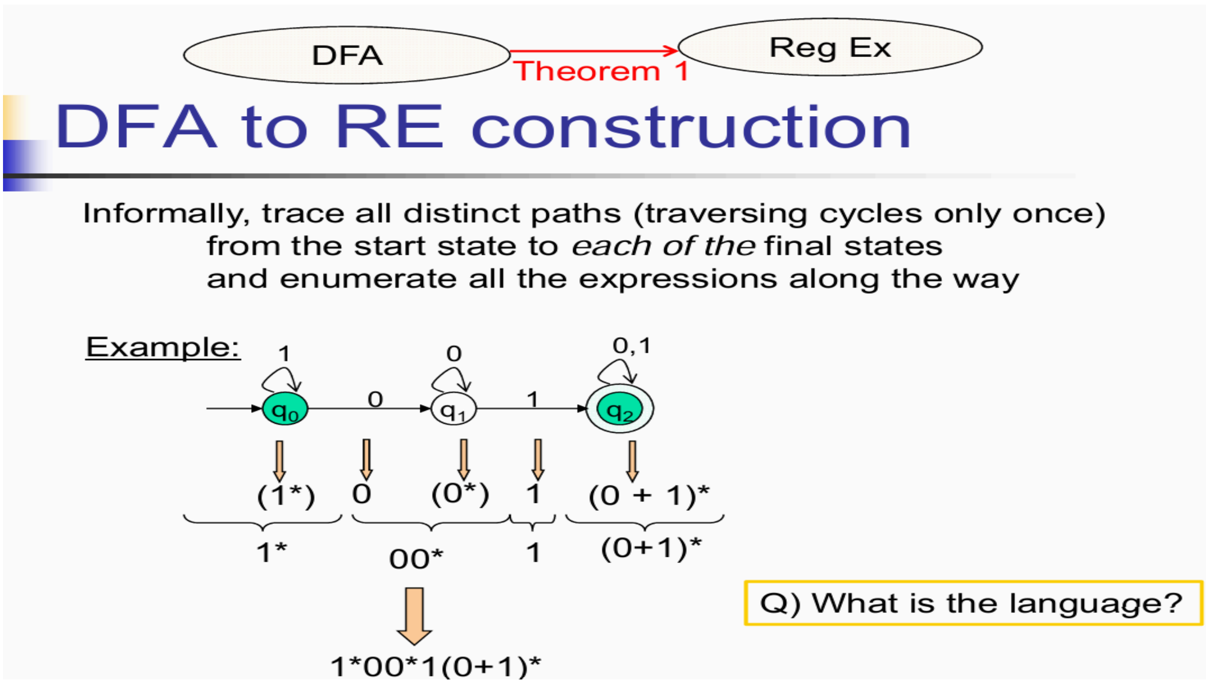
If $L = L(A)$ for some DFA, then there is a regular expression R such that $L = L(R)$ □

- We are going to construct regular expressions from a DFA by eliminating states.
- When we eliminate a state s, all the paths that went through s no longer exist in the automaton.
- If the language of the automaton is not to change, we must include, on an arc that goes directly from q to p, the labels of paths that went from some state q to state p, through s.
- The label of this arc can now involve strings, rather than single symbols (may be an infinite number of strings).
- We use a regular expression to represent all such strings.
- Thus, we consider automata that have regular expressions as labels.

c) Convert the following DFA to an equivalent regular Expression.



Theorem 1: Proofs in the book For every DFA A there exists a regular expression R such that L(R)=L(A)



d) Define- i) Associativity ii) Identity and iii) Distributive Law

i) Associativity:
A binary operation $*$ on a set is **associative** if the grouping of elements does not affect the result.
Mathematically:

$$(a * b) * c = a * (b * c) \quad \text{for all } a, b, c \text{ in the set.}$$

Example: Addition of numbers: $(2 + 3) + 4 = 2 + (3 + 4) = 9$.

ii) Identity:
An element e in a set is an **identity element** for a binary operation $*$ if combining it with any element of the set leaves that element unchanged.
Mathematically:

$$a * e = e * a = a \quad \text{for all } a \text{ in the set.}$$

Example: 0 is the identity for addition ($a + 0 = a$), and 1 is the identity for multiplication ($a \cdot 1 = a$).

iii) Distributive Law:
A binary operation $*$ is **distributive** over another operation $\oplus$ if:

$$a * (b \oplus c) = (a * b) \oplus (a * c) \quad \text{and} \quad (b \oplus c) * a = (b * a) \oplus (c * a)$$

Commutative: E+F = F+E

Associative: (E+F)+G = E+(F+G)

(EF)G = E(FG)

Distributive: E(F+G) = EF + EG

(F+G)E = FE+GE

a) Regarding conversion of DFA from NFA, a bad case for the Subset Construction is occurred. Analyze this using the Pigeonhole Principle.

Correctness of subset construction

Theorem: *If D is the DFA constructed from NFA N by subset construction, then L(D)=L(N)*

- Proof:
  - Show that $\hat{\delta}_D(\{q_0\},w) \equiv \hat{\delta}_N(q_0,w\}$, for all w
  - Using induction on w's length:
    - Let w = xa
    - $\hat{\delta}_D(\{q_0\},xa) \equiv \delta_D(\hat{\delta}_N(q_0,x\}, a) \equiv \hat{\delta}_N(q_0,w\}$

A bad case where #states(DFA)>>#states(NFA)

- L = {w | w is a binary string s.t., the k^{th} symbol from its end is a 1}
  - NFA has k+1 states
  - But an equivalent DFA needs to have at least 2^k states

(Pigeon hole principle)

- *m* holes and >*m* pigeons
  - => at least one hole has to contain two or more pigeons

When converting an NFA (Nondeterministic Finite Automaton) to a DFA (Deterministic Finite Automaton) using the **subset construction method**, the DFA can potentially have **up to 2n2^n2n states** if the NFA has nnn states.

A **bad case** happens when this exponential blow-up actually occurs, i.e., when **almost all subsets of NFA states are reachable** in the resulting DFA.

Bad Case in Subset Construction:

- Converting an NFA with n states to a DFA can produce up to 2^n DFA states.
- **Reason (Pigeonhole Principle):**
  - Each DFA state is a subset of NFA states.
  - If fewer than 2^n DFA states exist, some subsets must merge.
  - **Bad case:** all 2^n subsets are reachable → DFA has maximum states.

Example: NFA with 3 states → DFA may need $2^3 = 8$ states.

Illustrative Example:

- NFA with $n = 3$ states: $\{q_0, q_1, q_2\}$
- Possible subsets: $\emptyset, \{q_0\}, \{q_1\}, \{q_2\}, \{q_0, q_1\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\}$ → $2^3 = 8$ subsets.
- A cleverly designed NFA may reach **all 8 subsets**, so the DFA needs **all 8 states**, which is the worst-case blow-up.

b) Briefly describe, how we can eliminate ε- transitions?

Eliminating ε-Transitions (Short):

1. Find **ε-closure** of each state (states reachable via ε).
2. Redirect transitions: for input (a), connect states through ε-closures.
3. Mark a state as final if any state in its ε-closure is final.
4. **Remove all ε-transitions**.

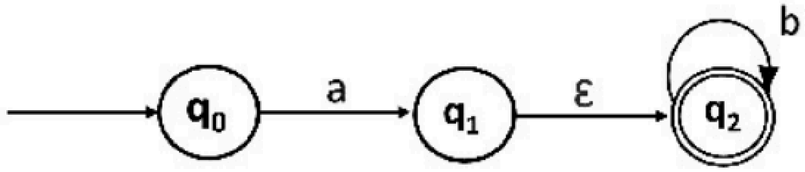
✅ Result: Equivalent NFA without ε-transitions.

NFA with ε can be converted to NFA without ε, and this NFA without ε can be converted to DFA. To do this, we will use a method, which can remove all the ε transition from given NFA. The method will be:

1. Find out all the ε transitions from each state from Q. That will be called as ε-closure{q1} where qi ∈ Q.
2. Then δ' transitions can be obtained. The δ' transitions mean a ε-closure on δ moves.
3. Repeat Step-2 for each input symbol and each state of given NFA.
4. Using the resultant states, the transition table for equivalent NFA without ε can be built.

Example:

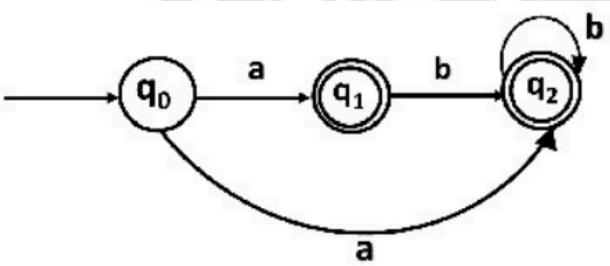
Convert the following NFA with ε to NFA without ε.



Solutions: We will first obtain ε-closures of q0, q1 and q2 as follows:

States	A	B
→q0	{q1, q2}	Φ
*q1	Φ	{q2}
*q2	Φ	{q2}

State q1 and q2 become the final state as ε-closure of q1 and q2 contain the final state q2. The NFA can be shown by the following transition diagram:



c) What are the uses of ε-Transitions?

Here’s a more **extended list of uses of ε-transitions** with short descriptions:

1. **Simplify NFA construction:**
   - Combine smaller NFAs into a bigger NFA without extra input symbols.
2. **Represent alternatives (union):**
   - For expressions like (A|B), ε-transitions connect the start state to multiple choices.
3. **Handle optional symbols:**
   - For (a?) or (a*), ε-transitions allow skipping input.
4. **Facilitate concatenation:**
   - Link the end of one NFA to the start of another NFA.
5. **Reduce the number of transitions:**
   - Avoid multiple direct transitions for the same input symbol.
6. **Intermediate/temporary transitions:**
   - Useful during design, then removed when converting to DFA.
7. **Model “do nothing” moves:**
   - Represents states that can change without consuming input.
8. **Ease NFA to DFA conversion:**
   - ε-closures help systematically construct the equivalent DFA.
9. **Simplify handling of complex regular expressions:**
   - Makes it easier to represent parentheses, repetition, or optional groups.

10. **Enable modular NFA design:**

    ○ Allows designing NFAs for sub-patterns and connecting them using ε-transitions.