Ch 1

Here are **common viva questions** from Unit 1 with **simple, direct answers** you can remember easily:

SET THEORY

Q1: What is a set?

A: A collection of distinct objects, like {1, 2, 3} or {a, b, c}.

Q2: What is a power set?

A: The set of all possible subsets of a given set.

Example: For {1, 2}, power set = {{}, {1}, {2}, {1, 2}}.

Q3: What is an empty set?

A: A set with no elements, written as {} or ∅.

Q4: What is the difference between ∅ and {∅}?

A: ∅ is an empty set. {∅} is a set containing one element — and that element is the empty set.

Q5: What is union and intersection?

A: Union (A ∪ B) = all elements in A or B.

Intersection (A ∩ B) = elements in both A and B.

LOGIC

Q6: What is a proposition?

A: A statement that is either true or false, like "2+2=4" (true) or "It is raining" (can be true/false).

Q7: What is a predicate?

A: A statement with a variable, like "$x > 5$". It’s not true or false until we give x a value.

Q8: What are quantifiers?

A: Words like "for all" ($\forall$) and "there exists" ($\exists$) that turn predicates into propositions.

AUTOMATA BASICS

Q9: What is an alphabet (Σ)?

A: A finite set of symbols, like $\Sigma = \{0, 1\}$ or $\Sigma = \{a, b, c\}$.

Q10: What is a string?

A: A finite sequence of symbols from an alphabet, like "010" from $\Sigma = \{0, 1\}$.

Q11: What is the empty string (ε)?

A: A string with zero symbols. Length = 0.

Q12: What is Kleene closure (Σ^*)?

A: The set of **all possible strings** (any length, including zero) from alphabet Σ.

Example: If $\Sigma = \{a\}$, then $\Sigma^* = \{\varepsilon, a, aa, aaa, \ldots\}$.

Q13: What is positive closure (Σ^+)?

A: Same as Kleene closure but **without the empty string ε**. $\Sigma^+ = \Sigma^* - \{\varepsilon\}$.

Q14: What is a language?

A: A set of strings chosen from Σ^* (i.e., $L \subseteq \Sigma^*$).

Example: All binary strings of length 2: L = {00, 01, 10, 11}.

Q15: What is concatenation of strings?

A: Joining two strings. "good" + "day" = "goodday".

Q16: What is a substring?

A: A string obtained by removing zero or more symbols from start and/or end.

Example: Substrings of "abc" = ε, "a", "b", "c", "ab", "bc", "abc".

Q17: Difference between empty string (ε) and empty language (∅)?

A: Empty string ε = a string with no symbols.

Empty language ∅ = a language with **no strings at all**.

COMPUTATION THEORY

Q18: What are the three areas of Theory of Computation?

A:

1. **Automata Theory** – How computation is done.

2. **Computability Theory** – What can be computed.

3. **Complexity Theory** – What can be computed efficiently.

Q19: What is an automaton?

A: An abstract model of a machine that processes input and produces output by following states.

Q20: Why study automata?

A: To design software, compilers (like lexical analyzers), text processors, and understand computation limits.

EXTRA QUICK REVIEW TIPS

- **Set** → Collection of objects.

- **Power Set** → All subsets.

- **Σ*** → All strings (with ε).

- **Σ^+** → All strings (no ε).

- **Language** → Set of strings.

- **Empty Language** → No strings at all.

Want me to create flashcards, a one-page cheat sheet, or practice more Q&A?

Ch 2

Here are **possible viva questions** from the Finite Automata chapter with **simple, clear answers**:

Basic Concepts (5-6 Questions)

1. What is a Finite Automaton?

Answer: A mathematical model with limited memory that reads input symbols and changes states based on fixed rules. It's like a simple robot with a tiny brain.

2. What are the 5 components of FA?

Answer: (Q, Σ, δ, q_0, F)

- Q: Set of states

- Σ: Input alphabet

- δ: Transition function

- q_0: Start state

- F: Set of final states

3. What's the difference between DFA and NFA?

Answer:

- **DFA:** One state → one input → exactly one next state (deterministic)

- **NFA:** One state → one input → zero, one, or multiple next states (non-deterministic)

DFA Specific (4-5 Questions)

4. Can a DFA have multiple final states?

Answer: Yes! F is a set, so it can have 0, 1, or multiple final states.

5. What is the extended transition function in DFA?

Answer: It tells which state you reach after reading an **entire string** (not just one symbol).

6. Design a DFA for strings ending with "01" over {0,1}

Answer: Need 3 states: q_0(start), q_1(saw 0), q_2(saw 01 - final). Transitions: q_0-0→q_1, q_0-1→q_0, q_1-0→q_1, q_1-1→q_2, q_2-0→q_1, q_2-1→q_0.

NFA Specific (4-5 Questions)

7. Why use NFA if DFA exists?

Answer: NFAs are often easier to design and more compact than equivalent DFAs.

8. How do you convert NFA to DFA?

Answer: Using **subset construction**: DFA states = subsets of NFA states. From set S on input a, go to union of all states reachable from any state in S.

9. What is ε-NFA?

Answer: An NFA that can change states without reading input using ε-transitions (free jumps).

ε-NFA (3-4 Questions)

10. What is ε-closure?

Answer: All states reachable from a given state using only ε-transitions (including itself).

11. How to remove ε-transitions?

Answer:

1. Find ε-closure for each state

2. For new transitions: from state q on input a, take ε-closure of all states reachable from ε-closure(q) on a

---

Conversion & Equivalence (3-4 Questions)

12. Are DFA, NFA, and ε-NFA equivalent?

Answer: Yes! All three accept exactly the same class of languages (regular languages).

13. What is the subset construction method?

Answer: The algorithm to convert NFA to DFA where each DFA state represents a set of NFA states.

Finite State Machines with Output (4-5 Questions)

14. Difference between Moore and Mealy machine?

Answer:

- **Moore:** Output depends only on current state. Output length = input length + 1

- **Mealy:** Output depends on current state AND input. Output length = input length

15. Which has more states: Moore or Mealy?

Answer: Generally, Moore machine has more states than equivalent Mealy machine.

16. Give real-world examples

Answer:

- **Moore:** Traffic light (output depends only on current light)

- **Mealy:** Vending machine (output depends on money inserted + button pressed)

Minimization (2-3 Questions)

17. Why minimize DFA?

Answer: To reduce number of states → less memory, faster processing.

18. What is the minimization method?

Answer: Partition states into groups of equivalent states and merge them.

Quick Tips for Viva:

1. **Draw diagrams** when explaining

2. **Use simple examples** like {0,1} alphabet

3. **Compare concepts** (DFA vs NFA, Moore vs Mealy)

4. **Know conversions** (NFA→DFA, ε-NFA→NFA)

5. **Practice designing** small DFAs for simple patterns

Total: ~25-30 possible questions covering the entire chapter! 🎯

Ch 3

Viva Questions on Regular Expressions with Answers

1. Basic Concepts

Q1: What is a regular expression?

A: A regular expression is a sequence of characters that forms a search pattern. It's an algebraic notation used to describe regular languages. It consists of symbols from an alphabet and operators (union, concatenation, Kleene star).

Q2: What is a regular language?

A: A language is regular if it can be described by a regular expression, recognized by a finite automaton (DFA/NFA), or generated by a regular grammar.

---

2. Operators and Properties

Q3: What are the basic operators in regular expressions?

A: Three fundamental operators:

1. **Union (| or +):** OR operation - `a|b` matches 'a' or 'b'

2. **Concatenation (.):** Followed by - `ab` means 'a' followed by 'b'

3. **Kleene Star (*):** Zero or more repetitions - `a*` matches "", "a", "aa", ...

Q4: What is the difference between * and + operators?

A:

- `*` means **zero or more** repetitions (includes empty string)

- `+` means **one or more** repetitions (excludes empty string)

- Example: `a*` = {"", "a", "aa", ...}, `a+` = {"a", "aa", "aaa", ...}

Q5: Is concatenation commutative?

A: No, concatenation is **not commutative**. `ab ≠ ba`. Union is commutative: `a|b = b|a`.

---

3. Equivalence with Finite Automata

Q6: Are regular expressions equivalent to finite automata?

A: Yes, for every regular expression, there exists an equivalent finite automaton (NFA with ε-moves), and vice versa. They both describe the same class of languages: regular languages.

Q7: How do you convert a regular expression to an NFA?

A: Using Thompson's construction:

- Single symbol 'a': Two-state automaton

- Union (R|S): New start with ε to both, combine final states

- Concatenation (R.S): Connect final of R to start of S

- Kleene Star (R*): New start/final with ε transitions

Q8: How do you identify final states in this conversion?

A: Final states are those where a complete pattern is matched:

- For `a`: State after 'a' is final

- For `ab`: State after 'b' is final

- For `a*`: Start state is also final (accepts empty string)

---

4. Closure Properties

Q9: What are closure properties of regular languages?

A: Regular languages are closed under:

1. **Union:** $L_1 \cup L_2$ is regular

2. **Intersection:** $L_1 \cap L_2$ is regular

3. **Concatenation:** $L_1.L_2$ is regular

4. **Kleene Star:** L^* is regular

5. **Complement:** L' is regular

6. **Reversal:** L^R is regular

Q10: Why are closure properties important?

A: They help:

- Prove languages are regular by construction

- Prove languages are NOT regular (by contradiction)

- Simplify language operations in compilers

---

5. Pumping Lemma

Q11: What is the pumping lemma for regular languages?

A: It states: For any regular language L, there exists a pumping length p such that any string s in L with $|s| \geq p$ can be divided into s = xyz where:

1. $|y| > 0$

2. $|xy| \leq p$

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

Q12: What is the pumping lemma used for?

A: To prove that a language is **NOT regular**. It cannot prove a language IS regular.

Q13: Show using pumping lemma that a^nb^n is not regular.

A:

1. Assume $L = \{a^nb^n\}$ is regular with pumping length p

2. Choose $s = a^pb^p$ ($|s| = 2p \geq p$)

3. By pumping lemma, s = xyz with $|xy| \leq p$, so y contains only a's

4. Pump y: $xy^2z = a^{p+k}b^p$ where $k = |y| > 0$

5. This has more a's than b's, not in L → Contradiction

6. Therefore, L is not regular

---

6. Applications and Examples

Q14: Give practical applications of regular expressions.

A:

1. **Validation:** Email, password format checking

2. **Search:** Find patterns in text (grep, Ctrl+F)

3. **Tokenization:** Breaking code into tokens in compilers

4. **Text processing:** Find/replace in editors

Q15: Write regular expressions for:

a) Strings ending with '00' over {0,1}

A: `(0+1)*00`

b) Strings containing '101' as substring

A: `(0+1)*101(0+1)*`

c) Strings with even number of 0's

A: `1*(01*01*)*`

d) Strings starting with 0 and ending with 1

A: `0(0+1)*1`

---

7. Advanced Questions

Q16: What is the difference between regular and context-free languages?

A:

- **Regular:** Can be described by regex/FA, limited memory

- **Context-free:** Need pushdown automata (stack memory)

- Example: $a^n b^n$ is context-free but not regular

Q17: Can a regular expression match parentheses balancing?

A: No, parentheses balancing (like `((()))`) is not regular. It requires counting/memory beyond finite automata capability.

Q18: What is ε-NFA? Why is it useful?

A: ε-NFA is an NFA with ε-transitions (empty string moves). Useful for:

- Simplifying regex to NFA conversion

- Easier union/concatenation constructions

- Can be converted to equivalent DFA

Q19: What is the Arden's theorem?

A: Arden's theorem helps solve equations of form X = AX + B to find regular expressions from automata. Solution: X = A*B if ε ∉ A.

Q20: What are the limitations of regular expressions?

A:

- Cannot count or match arbitrary repetitions

- Cannot handle nested structures (like HTML tags)

- Cannot match patterns requiring memory (like $a^n b^n$)

- Limited to regular languages only

---

8. Quick Short Questions

Q: Is every finite language regular?

A: Yes, any finite language is regular (can be described by union of its strings).

Q: Is ∅ (empty language) regular?

A: Yes, represented by regular expression ∅.

Q: Is ε (empty string language) regular?

A: Yes, represented by regular expression ε.

Q: Can regular expressions describe infinite languages?

A: Yes, using * operator: `a*` describes infinite language {"", "a", "aa", ...}.

Q: What is the identity element for concatenation?

A: ε (empty string) because R.ε = ε.R = R.

Viva Tips:

1. Start with definition when asked "What is...?"
2. Give examples with small alphabets {0,1} or {a,b}
3. Draw diagrams for automata questions
4. Mention both sides: what regex CAN and CANNOT do
5. Relate to practical applications when possible

This covers most common viva questions on regular expressions!

Ch 4

Unit 4: Context Free Grammar - Possible Viva Questions & Answers

Section 1: Basic Concepts

Q1: What is a Context-Free Grammar (CFG)?

A: A CFG is a formal grammar consisting of:

- **V:** Set of variables/non-terminals

- **T:** Set of terminals

- **P:** Production rules ($A \rightarrow \alpha$ where $A \in V$, $\alpha \in (V \cup T)^*$)

- **S:** Start symbol

It's called "context-free" because a variable can be replaced regardless of its surrounding context.

Q2: What is the difference between Regular Grammar and CFG?

A:

Regular Grammar	**Context-Free Grammar**
Right/Left linear only	More flexible productions
Can be recognized by FA	Requires PDA

| LHS has exactly 1 variable | LHS has exactly 1 variable |

| RHS limited patterns | RHS can be any combination |

Q3: What is a Context-Free Language (CFL)?

A: A language is CFL if there exists a CFG that generates it. Formally, L is CFL if ∃ CFG G such that L = L(G).

Section 2: Derivation & Parse Trees

Q4: What is the difference between leftmost and rightmost derivation?

A:

- **Leftmost derivation:** Always expand the leftmost variable first

- **Rightmost derivation:** Always expand the rightmost variable first

- Example: For S → S+S, string "a+b+c":

  - Leftmost: S → S+S → a+S → a+S+S → a+b+S → a+b+c

  - Rightmost: S → S+S → S+c → S+S+c → S+b+c → a+b+c

Q5: What is a parse tree?

A: A tree representation of a derivation where:

- Root = Start symbol S

- Internal nodes = Variables

- Leaves = Terminals

- In-order traversal gives the string

Q6: What is an ambiguous grammar? Give an example.

A: A grammar is ambiguous if a string has ≥2 different parse trees (or ≥2 leftmost/rightmost derivations).

Example:

```
S → S + S | S * S | a
```

String "a+a*a" has two parse trees:

- (a+a)*a

- a+(a*a)

Section 3: Normal Forms & Simplification

Q7: What is Chomsky Normal Form (CNF)?

A: A CFG is in CNF if all productions are of form:

1. A → BC (two variables)

2. A → a (one terminal)

3. S → ε is allowed only for start symbol

Q8: How do you convert a CFG to CNF?

A: Steps:

1. Add new start symbol if needed

2. Remove ε-productions (nullable symbols)

3. Remove unit productions (A → B)

4. Replace long productions (A → $X_1X_2...X_n$) with chain productions

5. Convert mixed productions (A → aB) to A → XB, X → a

Q9: What is Greibach Normal Form (GNF)?

A: All productions are of form:

A → aα where:

- a is a terminal

- α is a string of variables (can be empty)

Q10: What are useless symbols? How to remove them?

A: Two types:

1. **Non-generating symbols:** Can't derive any terminal string

2. **Non-reachable symbols:** Can't be reached from S

Removal process: Remove non-generating first, then non-reachable.

Section 4: Properties & Applications

Q11: What is the Pumping Lemma for CFLs?

A: If L is CFL, ∃ p (pumping length) such that ∀ s ∈ L with |s| ≥ p, s can be divided as s = uvxyz where:

1. |vxy| ≤ p

2. |vy| ≥ 1

3. ∀ i ≥ 0, uv^ixy^iz ∈ L

Q12: How to prove a language is not CFL using Pumping Lemma?

A: Steps:

1. Assume L is CFL, so pumping length p exists

2. Choose a string $s \in L$ with $|s| \geq p$

3. Show for all possible divisions uvxyz, pumping breaks the language rules

4. Contradiction → L is not CFL

Example: $L = \{a^nb^nc^n\}$ is not CFL.

Q13: What are closure properties of CFLs?

A:

Closed under:

- Union ($S \to S_1|S_2$)

- Concatenation ($S \to S_1S_2$)

- Kleene Star ($S \to S_1S|\varepsilon$)

- Substitution

Not closed under:

- Intersection (counterexample: $\{a^nb^nc^m\} \cap \{a^mb^nc^n\}$)

- Complement (would imply closed under intersection)

- Difference

Q14: What is Chomsky Hierarchy?

A:

Type	Grammar	Language	Automaton
Type 0	Unrestricted	Recursively Enumerable	Turing Machine

| Type 1 | Context-Sensitive | Context-Sensitive | Linear Bounded Automaton |

| Type 2 | Context-Free | Context-Free | Pushdown Automaton |

| Type 3 | Regular | Regular | Finite Automaton |

Section 5: Applications & Practical Questions

Q15: Where are CFGs used in real life?

A:

1. **Programming languages:** Syntax definition (BNF/EBNF)

2. **Compilers:** Parser generation (YACC, Bison)

3. **Natural Language Processing:** Sentence structure

4. **XML/HTML:** Document structure validation

5. **Pattern matching:** More powerful than regex

Q16: What is BNF (Backus-Naur Form)?

A: A notation for writing CFG rules:

- Non-terminals: `<variable>`

- ::= means "is defined as"

- | means "or"

Example: `<digit> ::= 0|1|2|3|4|5|6|7|8|9`

Q17: What is the difference between CFG and Context-Sensitive Grammar?

A:

| **CFG** | **Context-Sensitive Grammar** |

|---------|-----------------------------|

| A → α | αAβ → αγβ |

| No context needed | Replacement depends on context |

| Recognized by PDA | Recognized by LBA |

| Example: {a^nb^n} | Example: {$a^nb^nc^n$} |

Q18: Why is ambiguity bad in programming languages?

A: Ambiguity leads to:

- Multiple interpretations of the same code

- Unpredictable compiler behavior

- Example: `x = a + b * c` could be `(a+b)*c` or `a+(b*c)`

Q19: What is a sentential form?

A: Any string derived from the start symbol that may contain both terminals and non-terminals. If S →* α, then α is a sentential form.

Q20: How do you check if a grammar is ambiguous?

A:

1. Find a string with multiple parse trees

2. Or find multiple leftmost/rightmost derivations

3. Note: There's no algorithm to decide ambiguity for all grammars

---

Short Questions (One-line Answers)

Q21: Can a CFL be recognized by a Finite Automaton?

A: No, CFL requires Pushdown Automaton (PDA).

Q22: What is ε-production?

A: A production of form $A \to \varepsilon$ (produces empty string).

Q23: What is a unit production?

A: $A \to B$ where both are variables.

Q24: What is a nullable variable?

A: A variable that can derive ε.

Q25: What is the language of a grammar?

A: $L(G) = \{w \in T^* \mid S \to^* w\}$

Quick Tips for Viva:

1. Always start definitions with formal notation if possible

2. Give examples for every concept

3. Relate concepts to each other (e.g., CNF → simplification → parsing efficiency)

4. Mention applications - examiners love real-world connections

5. Practice drawing parse trees quickly

6. Be ready with counterexamples for non-closure properties

Ch 5

Excellent idea! Here's a comprehensive list of **possible viva questions** on Push Down Automata (Unit 5) with **clear, concise answers**:

Section 1: Basic Concepts

Q1: What is a PDA? How is it different from a Finite Automaton?

A: A Push Down Automaton (PDA) is a computational model with finite states, an input tape, and a **stack** (LIFO memory). Unlike Finite Automata which have only finite memory (states), PDAs have **infinite memory** through the stack, allowing them to recognize Context-Free Languages like `$\{a^n b^n \mid n \geq 0\}$`.

Q2: What are the main components of a PDA?

A: A PDA has:

1. **Finite State Control** (like NFA)

2. **Input Tape** (holds the string)

3. **Stack** (infinite LIFO memory)

4. **Transition Function δ** (defines moves)

Q3: What is the formal definition of a PDA?

A: PDA = (Q, Σ, Γ, δ, q_0, Z_0, F) where:

- Q = Finite set of states

- Σ = Input alphabet

- Γ = Stack alphabet

- δ = Transition function $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to Q \times \Gamma^*$

- q_0 = Initial state

- Z_0 = Initial stack symbol

- F = Set of final states

Section 2: PDA Operations & Acceptance

Q4: What are the basic stack operations in PDA?

A: Three operations:

1. **Push**: Add symbol(s) to stack top

2. **Pop**: Remove top symbol

3. **Step/No-op**: Leave stack unchanged

Q5: What is an Instantaneous Description (ID)?

A: An ID (q, w, γ) is a "snapshot" of PDA computation where:

- q = Current state

- w = Remaining input

- γ = Current stack contents (leftmost = top)

Q6: What are the two ways a PDA can accept a string?

A:

1. **Acceptance by Final State**: String accepted if PDA ends in any final state after reading entire input.

2. **Acceptance by Empty Stack**: String accepted if stack is empty after reading entire input.

Q7: Can you convert a PDA accepting by final state to one accepting by empty stack?

A: Yes! Add new start state `$q_s$` with new bottom marker `$X_0$`. Push original `$Z_0$`, run original PDA. From each final state, add ε-transitions to a new cleanup state `$q_e$` that pops everything until `$X_0$` is popped.

Section 3: Types of PDAs

Q8: What is the difference between DPDA and NPDA?

A:

- **DPDA** (Deterministic): For every (state, input, stack-top), there's **at most one** transition.

- **NPDA** (Non-deterministic): Can have **multiple possible transitions** for same (state, input, stack-top).

- **Power**: NPDAs are more powerful (recognize all CFLs); DPDAs recognize only deterministic CFLs.

Q9: Why is NPDA more powerful than DPDA?

A: Some languages like `$\{ww^R \mid w \in (a+b)^*\}$` require guessing the middle point, which needs non-determinism. DPDAs cannot handle such languages.

Section 4: PDA and CFG Relationship

Q10: How are PDAs and CFGs related?

A: They are **equivalent** in power:

- Every CFG can be converted to an equivalent PDA

- Every PDA can be converted to an equivalent CFG

- Both describe Context-Free Languages

Q11: How to convert a CFG to PDA?

A: For CFG G = (V, T, P, S), create PDA with:

- One state `q`

- Stack alphabet = $V \cup T$

- Initial stack symbol = S

- **Two types of transitions**:

  1. For each rule $A \to \alpha$: $\delta(q, \varepsilon, A) = (q, \alpha)$

  2. For each terminal a: $\delta(q, a, a) = (q, \varepsilon)$

Q12: How to convert PDA to CFG?

A: More complex. Create variables [qXp] meaning: "Starting in state q with X on stack, reach state p having popped X". Generate productions based on PDA transitions.

Section 5: Applications & Examples

Q13: Design a PDA for L = {a^nb^n | n ≥ 0}

A:

- States: q_0 (start), q_1 (popping), q_2 (final)

- Transitions:

 1. $\delta(q_0, a, Z_0) = (q_0, AZ_0)$ [Push A for each a]

 2. $\delta(q_0, a, A) = (q_0, AA)$ [Continue pushing]

 3. $\delta(q_0, b, A) = (q_1, \varepsilon)$ [Start popping for first b]

 4. $\delta(q_1, b, A) = (q_1, \varepsilon)$ [Continue popping]

 5. $\delta(q_0, \varepsilon, Z_0) = (q_2, Z_0)$ [Accept empty string]

 6. $\delta(q_1, \varepsilon, Z_0) = (q_2, Z_0)$ [Accept when done]

Q14: Design a PDA for balanced parentheses

A: Similar to a^nb^n but with '(' and ')':

- Push '(' when seen

- Pop '(' when ')' seen (only if '(' on top)

- Accept if stack has Z_0 at end

Q15: What languages cannot be recognized by PDA?

A: Languages requiring two independent comparisons like `{$a^nb^nc^n$ | n ≥ 0}` or `{ww | w ∈ (a+b)*}` need more than one stack (need Turing Machine).

Section 6: Tricky Questions

Q16: Can a PDA accept by both final state and empty stack simultaneously?

A: Yes, if designed that way, but in general, these are two different acceptance criteria. We can convert between them.

Q17: Why is the stack called "push down" automaton?

A: Because you can only access the **top** of the stack. You "push" symbols down onto it, and "pop" from the top. It's a LIFO (Last-In-First-Out) structure.

Q18: Can we have multiple stacks in a PDA?

A: Standard PDA has **one stack**. Adding a second stack makes it as powerful as a Turing Machine (2-PDA = TM).

Q19: What is the role of ε-transitions in PDA?

A: ε-transitions allow the PDA to:

1. Change state without reading input

2. Perform stack operations without consuming input

3. Enable non-deterministic choices

4. Implement "lookahead" behavior

Q20: How to check if a string is accepted by a given PDA?

A: Trace all possible computation paths from initial ID (q_0, w, Z_0). If any path leads to:

- Final state acceptance: (q_f, ε, γ) where q_f ∈ F

- Empty stack acceptance: (q, ε, ε)

Then the string is accepted.

Quick Tips for Viva:

1. **Always start with simple definitions**

2. **Use examples** (especially $a^n b^n$)

3. **Draw diagrams** when explaining

4. **Mention "stack"** repeatedly - it's the key difference from FA

5. **Compare with FA** to show why PDA is more powerful

Want me to explain any specific question in more detail? Or would you like a mock viva practice? 😊

Ch 6

Excellent question! Here's a comprehensive list of **viva questions** from the Turing Machine chapter with simple, clear answers.

UNIT 6: TURING MACHINE - VIVA QUESTIONS & ANSWERS

BASIC DEFINITIONS (6.1)

Q1. What is a Turing Machine?

A: A theoretical model of computation consisting of an infinite tape, a read-write head, and a finite control with transition rules. It can compute anything that is algorithmically computable.

Q2. What are the 7 components of a TM (7-tuple)?

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

- Q: Finite set of states

- Σ: Input alphabet

- Γ: Tape alphabet (includes Σ and extra symbols like B)

- δ: Transition function

- q_0: Initial state

- B: Blank symbol

- F: Set of final states

Q3. What is an Instantaneous Description (ID)?

A: A "snapshot" of a TM's computation showing: current state, tape contents, and head position. Written as α q β where q is the state, α is left tape, β is right tape.

Q4. How does a TM accept a string?

A: A TM accepts a string if it starts with that string on tape and eventually **halts in a final state** (ACCEPT state).

ROLES OF TM (6.2)

Q5. How is TM a language recognizer?

A: It reads input and decides whether to accept or reject it. Languages accepted by TM are called Recursively Enumerable languages.

Q6. How can TM compute functions? Example?

A: The input represents the function argument (e.g., number of 1's = number n), and the output on tape after halting represents the function value.

Example: f(x) = x+1. Input: 111 (for 3), Output: 1111 (for 4).

Q7. What is a TM with storage in state?

A: The state itself stores data as a tuple (q, data). Useful for remembering small amounts of information.

Q8. What is a TM enumerator?

A: A TM that systematically generates (prints) all strings of a language, one by one, usually separated by #.

VARIATIONS OF TM (6.3)

Q9. What's the difference between multi-track and multi-tape TM?

A:

- **Multi-track:** Single tape divided into parallel tracks (like multiple lines on same page)

- **Multi-tape:** Multiple separate tapes, each with its own head

Q10. Are multi-tape TM more powerful than single-tape TM?

A: No, they are equivalent. Any multi-tape TM can be simulated by a single-tape TM (using multiple tracks to represent multiple tapes).

Q11. What is Non-deterministic TM (NTM)?

A: TM where δ gives multiple possible moves for (state, symbol). It accepts if ANY path leads to acceptance. NTM = Deterministic TM in power.

Q12. What are restricted TM models?

A: Models with limitations but still TM-equivalent:

1. **Semi-infinite tape** (tape infinite only in one direction)

2. **Multistack machine** (2 stacks = TM)

3. **Counter machine** (2 counters = TM)

ADVANCED CONCEPTS (6.4)

Q13. What is Church-Turing Thesis?

A: The hypothesis that: "Anything computable by an algorithm is computable by a Turing Machine." It's not provable but widely accepted.

Q14. What is a Universal Turing Machine (UTM)?

A: A special TM that can simulate ANY other TM. Its input is: ⟨M⟩#w where ⟨M⟩ is encoded description of TM M, and w is input for M.

Q15. Why is UTM important?

A: It's the theoretical basis for stored-program computers (like modern computers). One fixed hardware (UTM) can run any program (encoded TM).

Q16. What is encoding of a TM?

A: Converting a TM's description (states, symbols, transitions) into a binary string ⟨M⟩ so it can be given as input to another TM (like UTM).

Q17. What languages does UTM accept?

A: The Universal Language \(L_u = \{ \langle M, w \rangle | M \text{ accepts } w \} \). This is RE but not recursive (undecidable).

Q18. What's the main difference between TM and FA/PDA?

A:

	FA/PDA	TM
Tape Movement	Left-to-right only	Left AND Right
Modify Input?	No (read-only)	Yes (read/write)
Memory	Limited (none or stack)	Infinite tape
Can halt early?	No	Yes

PRACTICAL/TECHNIQUE QUESTIONS

Q19. What is "marking" in TM?

A: Changing a symbol to a special marker (X, Y) to remember that cell has been processed. Essential for counting, matching, and keeping track.

Q20. How do you design a TM for palindrome checking?

A:

1. Mark first symbol (a→X or b→Y)

2. Go to end, check if last symbol matches

3. Mark last symbol

4. Return to next unmarked symbol

5. Repeat until all marked or mismatch found

Q21. What does "enumerating binary strings" mean?

A: Listing all binary strings systematically: ε, 0, 1, 00, 01, 10, 11, 000,... This creates one-to-one mapping with natural numbers.

Q22. Can a TM have infinite states?

A: No, by definition Q is finite. But tape is infinite.

Q23. Can TM solve any problem?

A: No! There are **uncomputable problems** like Halting Problem. TM defines the limit of computation.

Q24. What are final states in TM?

A: Two types: ACCEPT (halts and accepts) and REJECT (halts and rejects). Some TMs also have LOOP (never halts) for rejection.

TRICKY/INSIGHTFUL QUESTIONS

Q25. If TM has infinite tape, is it more powerful than a computer?

A: No, theoretically equivalent. Computer has finite memory but can request more (like infinite tape). TM is simpler but equally powerful theoretically.

Q26. Why are 2 stacks enough for TM power?

A: One stack = left part of tape, other stack = right part of tape. Moving head = pop from one, push to other.

Q27. Why are 2 counters enough for TM power?

A: Two counters can simulate a stack. Since 2 stacks = TM, therefore 2 counters = TM.

Q28. What happens if TM tries to move left from leftmost cell?

A: It stays in same cell (no left movement possible). Some definitions add special left-end marker.

Q29. Can TM have 0 final states?

A: Yes, but then it never accepts anything—only rejects or loops.

VIVA TIPS:

1. **Start simple:** Begin with "TM is a mathematical model of computation..."
2. **Use analogies:** "Tape is like infinite paper", "UTM is like a computer CPU"
3. **Give examples:** Always provide simple examples (like a^nb^n language)
4. **Connect concepts:** Show how encoding → UTM → Church-Turing thesis are linked
5. **Be honest:** Say "I don't know" if unsure, but try to relate to something you know

This covers virtually all viva questions from this chapter! Good luck! 🍀

Ch 7

Unit 7: Undecidability & Intractability - Possible Viva Questions & Answers

---

1. What is Computational Complexity?

Answer: It is the study of the resources (time and memory) required to solve computational problems. We focus on how resource needs **grow** as input size increases, not on exact numbers.

2. Differentiate between Time and Space Complexity.

Answer:

- **Time Complexity:** Measures the number of steps (runtime) an algorithm takes.

- **Space Complexity:** Measures the amount of memory (tape cells) an algorithm uses.

Both are expressed as functions of input size `n`.

3. What is a Turing Machine? Why is it used?

Answer: A Turing Machine is a simple abstract model of a computer with infinite memory (tape) and a read/write head. It is used because it is **universal**—any algorithm that can run on any real computer can be simulated on a TM, making it perfect for theoretical analysis.

4. What are Tractable and Intractable Problems?

Answer:

- **Tractable:** Problems that can be solved in **polynomial time** (e.g., \(O(n^2)\)). Considered "efficiently solvable."

- **Intractable:** Problems that require **exponential time** (e.g., \(O(2^n)\)) for the best-known algorithms. Considered "hard" or impractical for large inputs.

5. Define P and NP classes.

Answer:

- **Class P:** Set of **decision problems** solvable by a **deterministic** TM in polynomial time.

- **Class NP:** Set of decision problems where a proposed "yes" answer can be **verified** by a deterministic TM in polynomial time.

Key Relationship: \(P \subseteq NP\) (Every tractable problem can be verified quickly).

6. What is the P vs NP Problem?

Answer: It is the most famous open question in CS: Is **P = NP**?

- If **yes**, all problems easy to verify are also easy to solve.

- If **no** (which most believe), there exist hard problems that can be checked quickly but not solved quickly.

7. What is NP-Completeness?

Answer: A problem is **NP-Complete** if:

1. It is in **NP** (solutions can be verified quickly).

2. It is **NP-hard** (every problem in NP can be reduced to it in polynomial time).

Example: SAT, Traveling Salesman.

8. Explain the significance of Cook's Theorem.

Answer: **Cook-Levin Theorem** proved that **SAT is NP-Complete**. This was the **first** NP-Complete problem found, making it a benchmark. Now, to prove a new problem is NP-Complete, we just need to reduce SAT to it.

9. What is a Reduction? Why is it important?

Answer: A **reduction** transforms one problem into another so that solving the second solves the first.

Importance: It allows us to compare problem difficulty. If problem A reduces to problem B, then B is at least as hard as A.

10. What is the Halting Problem? Is it solvable?

Answer: The Halting Problem asks: "Given a program and its input, will it eventually stop or run forever?"

No, it is not solvable. It was proven **undecidable** by Alan Turing—no algorithm can correctly answer this for all possible programs.

11. Explain the proof of the Halting Problem's undecidability.

Answer: Proof by contradiction:

1. Assume a program `H(P, I)` exists that solves it.

2. Create a tricky program `K(P)` that does the **opposite** of what `H` predicts when given itself as input.

3. Ask: Does `K(K)` halt? This leads to a contradiction in both cases, proving `H` cannot exist.

12. What are Undecidable Problems? Give examples.

Answer: Problems for which **no algorithm can always give a correct yes/no answer** in finite time.

Examples:

- Halting Problem

- Post's Correspondence Problem (PCP)

- Whether two CFGs are equivalent

- Whether a CFG is ambiguous

13. What is SAT? Why is it so important?

Answer: **SAT (Boolean Satisfiability)** asks: "Given a Boolean formula, is there an assignment of TRUE/FALSE to variables that makes the whole formula TRUE?"

Importance: It is the **first proven NP-Complete problem** (Cook's Theorem). It serves as the foundation for proving other problems NP-Complete.

14. Is SAT in P or NP?

Answer: SAT is in **NP** (solutions can be verified quickly). Whether it is in **P** is unknown—if someone proves SAT is in P, then \(P = NP\).

15. What is Post's Correspondence Problem (PCP)?

Answer: Given two lists of strings, find a sequence of indices where concatenating strings from List 1 equals concatenating strings from List 2 in the same order. It is **undecidable**.

16. Differentiate between Decidable and Undecidable Problems.

Answer:

- **Decidable:** An algorithm exists that always halts with the correct answer.

- **Undecidable:** No such algorithm can exist for all inputs.

17. What is the difference between Undecidable and Intractable?

Answer:

- **Undecidable:** No algorithm exists at all (e.g., Halting Problem).

- **Intractable:** An algorithm exists but takes exponential time, making it impractical (e.g., SAT).

18. What are Complexity Classes? Give examples.

Answer: Groups of problems with similar resource requirements.

Examples: **P, NP, NP-Complete, NP-hard, EXPTIME**.

19. Can a problem be in both P and NP-Complete?

Answer: Only if **P = NP**. Currently, if a problem is NP-Complete, it is believed **not** to be in P.

20. How do you prove a problem is NP-Complete?

Answer: Two steps:

1. Show it is in **NP** (verify a solution quickly).

2. Show it is **NP-hard** by reducing a known NP-Complete problem (like SAT) to it in polynomial time.

Quick Review Tip: Remember—**P** is "easy to solve," **NP** is "easy to check," **NP-Complete** is "hardest in NP," and **Undecidable** is "impossible to solve algorithmically."