SAHYADRI
COLLEGE OF ENGINEERING & MANAGEMENT
An Autonomous Institution
MANGALURU

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

RUBRICS TO EVALUATE THE ASSIGNMENT ON ATC USING JFLAP

Academic Year : 2025-26

Max Marks : 20

Sl.No	Criteria	Excellent (5M)	Good (4M)	Average (3M)	Needs Improvement (2M)	Poor(1M)
1	Problem Understanding and Modelling	Clearly understands the language/problem description. Identifies correct alphabet, constraints, and structure.	Understands most requirements correctly. Minor misunderstandings or missing details, but overall modeling matches the problem.	Basic understanding with notable gaps. Some elements of the model are incorrect or incomplete.	Misunderstanding of key components of the language/problem.	No meaningful understanding..
2	Automaton Implementation in JFLAP	JFLAP automaton is fully correct and complete. Transitions, states, start/accept states are properly defined.	Mostly correct automaton with minor logical or structural issues.	Automaton works partially:	Significant errors in the JFLAP automaton.	JFLAP automaton is non-functional or unrelated to the required language.
3	Example Analysis Using JFLAP Traces	Provides multiple test strings (accepted + rejected).	Provides correct examples with adequate explanation.	Includes examples, but they are limited or lacking detailed trace explanation.	Very few examples	No examples, no simulation traces, or examples are incorrect/irrelevant.
4	Analysis and Interpretation of Results	Interprets results thoroughly.	Good interpretation with mostly correct reasoning.	Some interpretation is present, but lacks depth.	Interpretation is shallow or incorrect. Limited reasoning based on JFLAP output	No meaningful interpretation. Results are misinterpreted or missing.

Total Marks Awarded

USN	Student name	1	2	3	4	Total
4SF23CS246	Vivek Neeralagi					
4SF23CS030	Aravind P Sagar					

Signature

CHAITHRA S

1 Introduction

This report presents the design and analysis of several computational models in the theory of computation, including Finite Automata (FA), Pushdown Automata (PDA), and Turing Machines (TM). For each language, appropriate computational models are constructed and verified using JFLAP. The report is organized so that **each question has its methodology, results, and discussion grouped under Sections 3, 4, and 5 respectively**.

2 Problem Statements

1. Design a NPDA for $L = \{a^n b^n | n \geq 1\}$
2. Construct DFA for $L = \{ab^n a^m : n \geq 2, m \geq 3\}$
3. DFA for strings ending with '0011', $\Sigma = \{0, 1\}$
4. NPDA for $L = \{a^n b^m c^{n+m}\}$
5. Minimal DFA where 'a' is never followed by 'bb'

12. NFA for $\{ab, abc\}^*$
13. DFA for strings ending with 'abb'
14. DFA for strings ending with 'abba'
15. DFA/NFA for strings containing "the"
16. DFA/NFA for strings ending with "ing"

29. TM accepting palindromes over $\{a, b\}$
30. TM accepting $\{ww^R\}$
24. TM accepting $\{0^n 1^n\}$
25. TM accepting $\{0^n 1^n 2^n\}$
26. TM for strings containing substring 001

3 Methodology

This section details the construction approach for each question.

3.1 Methodology for Question 1

Design a non deterministic PDA for accepting the language $L = \{a^n b^n | n \geq 1\}$

1. Open JFLAP. 2. Click on **Pushdown Automaton**. 3. **Add States**:

   - Click the **State Creator** tool (circle icon).
   - Click on the canvas to create 4 states: `q0`, `q1`, `q2`, `q3`.
   - Right-click `q0` and select **Initial**.
   - Right-click `q3` and select **Final**.

4. **Define Transitions**:

   - Click the **Transition Creator** tool (arrow icon).
   - Create the following transitions (Input, Pop, Push):
   - `q0` -> `q1`: `(a, Z, aZ)` (Push 'a' on 'Z')
   - `q1` -> `q1`: `(a, a, aa)` (Push 'a' on 'a')
   - `q1` -> `q2`: `(b, a, )` (Pop 'a' on 'b')
   - `q2` -> `q2`: `(b, a, )` (Pop 'a' on 'b')
   - `q2` -> `q3`: `(, Z, Z)` (Accept if stack is empty of 'a's)

5. **Verify**:

   - Go to **Input** -> **Step with Closure** or **Fast Run**.
   - Test with `aabb` (Accept), `ab` (Accept), `a` (Reject), `b` (Reject).

6. **Save**:

   - Save the file as `1.jff` in the `set_1/1` folder.

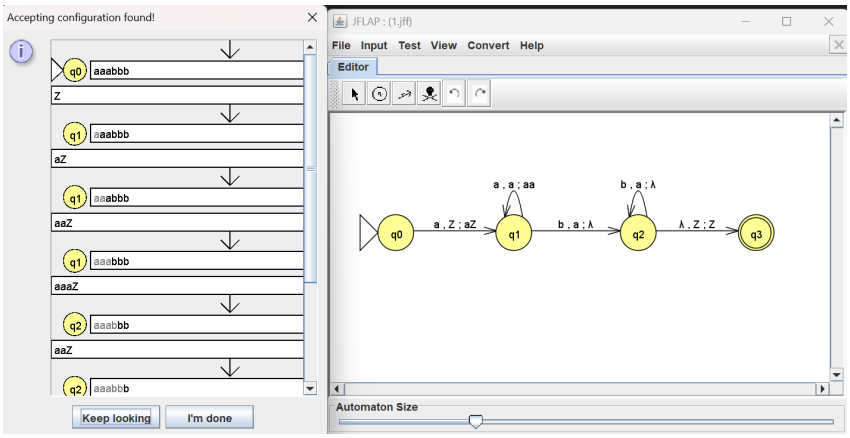

Figure 1: Automaton for Question 1

3.2 Methodology for Question 2

Construct a DFA for $L = \{ab^n a^m : n \geq 2, m \geq 3\}$

1. Open JFLAP. 2. Click on **Finite Automaton**. 3. **Add States**:

- Create states `q0` to `q6`.
- Set `q0` as **Initial**.
- Set `q6` as **Final**.

4. **Define Transitions**:

- `q0` -> `q1`: `a`
- `q1` -> `q2`: `b`
- `q2` -> `q3`: `b`
- `q3` -> `q3`: `b` (Loop for $n > 2$)
- `q3` -> `q4`: `a`
- `q4` -> `q5`: `a`
- `q5` -> `q6`: `a`
- `q6` -> `q6`: `a` (Loop for $m > 3$)

5. **Verify**:

- Test with `abbaaa` (Accept), `abbbaaa` (Accept), `abbaaaa` (Accept).
- Test with `abaaa` (Reject - only 1 b), `abbaa` (Reject - only 2 a's).

6. **Save**:

- Save as `2.jff` in `set_1/2`.

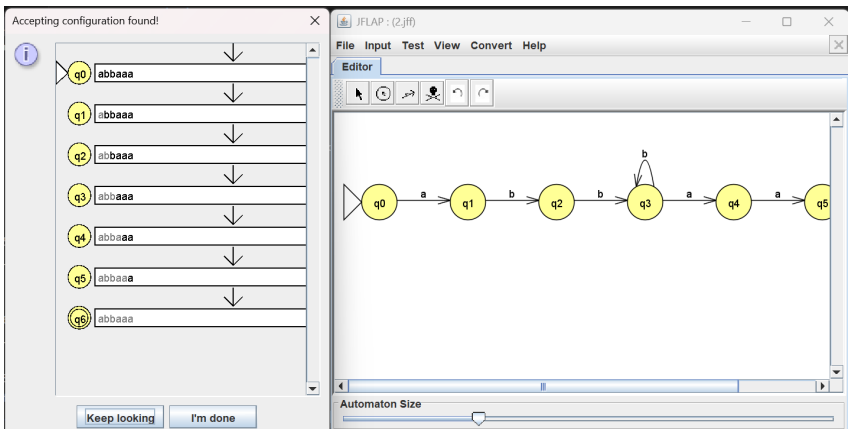

Figure 2: Automaton for Question 2

3.3 Methodology for Question 3

Draw a DFA for the language accepting strings ending with '0011' over input alphabets $\Sigma = \{0, 1\}$

1. Open JFLAP. 2. Click on **Finite Automaton**. 3. **Add States**:

- Create states `q0` to `q4`.
- Set `q0` as **Initial**.
- Set `q4` as **Final**.

4. **Define Transitions**:

- `q0` -> `q1`: `0` (First '0')
- `q0` -> `q0`: `1` (Reset)
- `q1` -> `q2`: `0` (Second '0')
- `q1` -> `q0`: `1` (Reset)
- `q2` -> `q3`: `1` (First '1')
- `q2` -> `q2`: `0` (Stay on '00')
- `q3` -> `q4`: `1` (Second '1' - Accept)
- `q3` -> `q1`: `0` (Back to '0')
- `q4` -> `q1`: `0` (Back to '0')
- `q4` -> `q0`: `1` (Reset)

5. **Verify**:

- Test with `0011` (Accept), `10011` (Accept), `00011` (Accept).
- Test with `001` (Reject), `011` (Reject), `00110` (Reject).

6. **Save**:

- Save as `3.jff` in `set_1/3`.

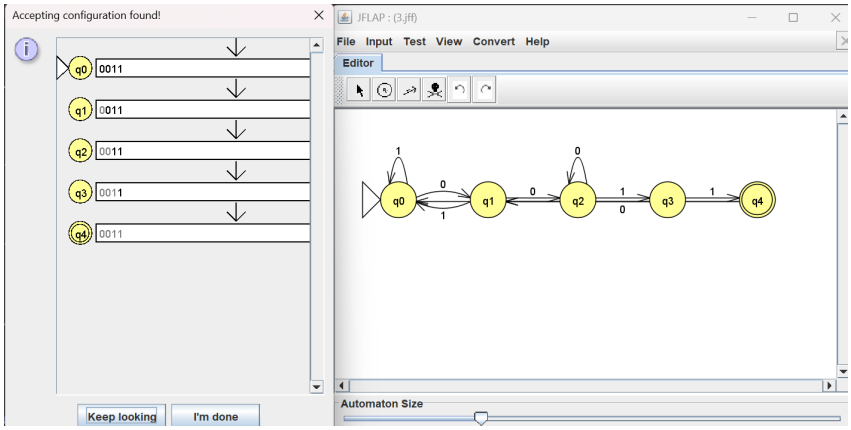

Figure 3: Automaton for Question 3

3.4 Methodology for Question 4

Construct npda's that accept the following languages on $\Sigma = \{a, b, c\}$. $L = \{a^n b^m c^{n+m} : n \geq 0, m \geq 0\}$

1. Open JFLAP. 2. Click on **Pushdown Automaton**. 3. **Add States**:

- Create states `q0` to `q4`.
- Set `q0` as **Initial**.
- Set `q4` as **Final**.

4. **Define Transitions**:

- `q0` -> `q1`: `(a, Z, aZ)` (Push first 'a')
- `q0` -> `q2`: `(b, Z, bZ)` (Push first 'b' if no 'a's)
- `q0` -> `q4`: `(, Z, Z)` (Accept empty string)
- `q1` -> `q1`: `(a, a, aa)` (Push 'a's)
- `q1` -> `q2`: `(b, a, ba)` (Push first 'b' on 'a')
- `q1` -> `q3`: `(c, a, )` (Pop 'a' for 'c' if no 'b's)
- `q2` -> `q2`: `(b, b, bb)` (Push 'b's)
- `q2` -> `q3`: `(c, b, )` (Pop 'b' for 'c')
- `q3` -> `q3`: `(c, b, )` (Pop 'b's)
- `q3` -> `q3`: `(c, a, )` (Pop 'a's after 'b's are gone)
- `q3` -> `q4`: `(, Z, Z)` (Accept if stack empty)

5. **Verify**:

- Test with `abc` (Accept), `aabbcc` (Accept), `ac` (Accept), `bc` (Accept).
- Test with `ab` (Reject), `abc` (Accept), `abcc` (Reject - too many c's), `aabc` (Reject - too few c's).

6. **Save**:

- Save as `4.jff` in `set_1/4`.

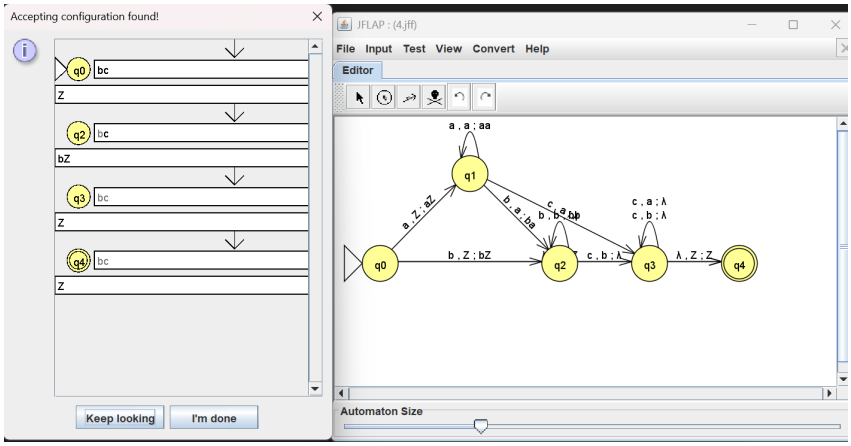

Figure 4: Automaton for Question 4

3.5 Methodology for Question 5

Construction of a minimal DFA accepting set of strings over $\{a, b\}$ in which every 'a' is never be followed by 'bb'.

1. Open JFLAP. 2. Click on **Finite Automaton**. 3. **Add States**:

- Create states `q0`, `q1`, `q2`, `q3`.
- Set `q0` as **Initial**.
- Set `q0`, `q1`, `q2` as **Final**.

4. **Define Transitions**:

- `q0` -> `q0`: `b` (Leading 'b's are fine)
- `q0` -> `q1`: `a` (Found an 'a', start checking)
- `q1` -> `q1`: `a` (Another 'a', reset check)
- `q1` -> `q2`: `b` (Found 'ab', warning)
- `q2` -> `q1`: `a` (Found 'aba', safe, reset check)

- `q2` -> `q3`: `b` (Found 'abb', Reject)
- `q3` -> `q3`: `a` (Trap)
- `q3` -> `q3`: `b` (Trap)

5. **Verify**:
   - Test with `ab` (Accept), `aba` (Accept), `bba` (Accept), `aab` (Accept).
   - Test with `abb` (Reject), `aabb` (Reject), `babb` (Reject).
6. **Save**:
   - Save as `5.jff` in `set_1/5`.

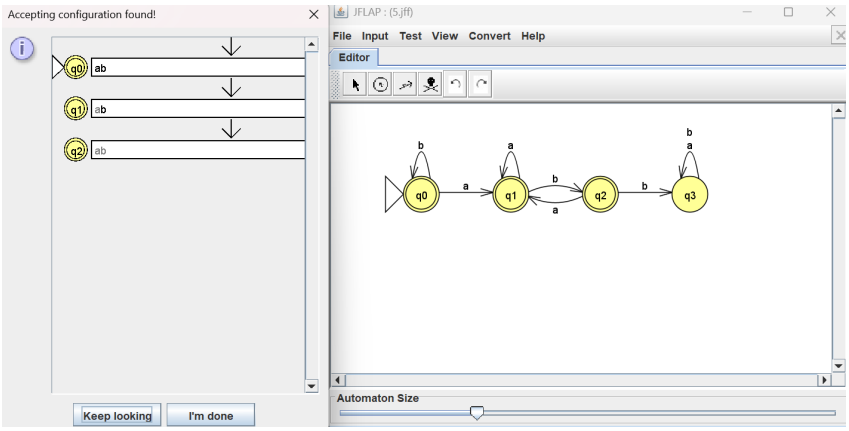

Figure 5: Automaton for Question 5

3.6 Methodology for Question 12

Construct an NFA that accepts the language $\{ab, abc\}^*$. This is the set of strings where ab and abc may be repeated. Example strings include abcab, ababcab, abcabcabc, and the empty string.

1. Open JFLAP. 2. Click on **Finite Automaton**. 3. **Add States**:
   - Create states `q0` to `q4`.
   - Set `q0` as **Initial** and **Final**.
4. **Define Transitions**:
   - `q0` -> `q1`: `a` (Start of 'ab' or 'abc')
   - `q1` -> `q0`: `b` (Completes 'ab', back to start)
   - `q1` -> `q2`: `b` (Part of 'abc')

- `q2` -> `q0`: `c` (Completes 'abc', back to start)

5. **Verify**:

- Test with `ab` (Accept), `abc` (Accept), `abab` (Accept), `abcab` (Accept), `ababc` (Accept).
- Test with `a` (Reject), `ac` (Reject), `abb` (Reject), `abca` (Reject).

6. **Save**:

- Save as `12.jff` in `set_2/12`.

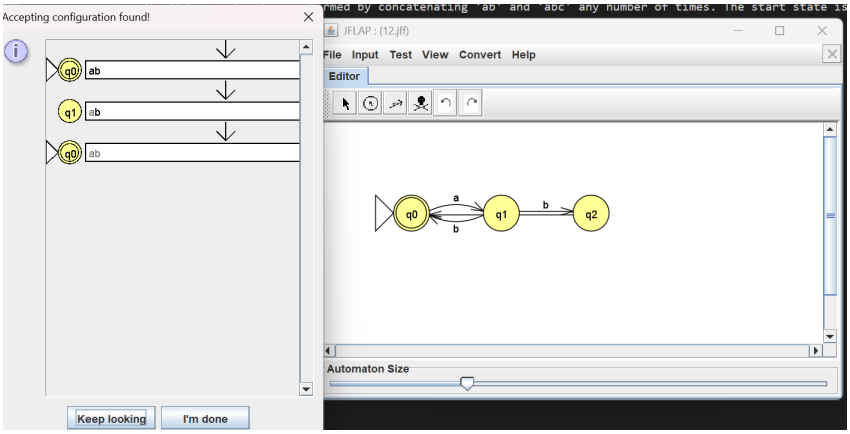

Figure 6: Automaton for Question 12

3.7 Methodology for Question 13

Draw a DFA for the language accepting strings ending with 'abb' over input alphabets $\Sigma = \{a, b\}$

1. Open JFLAP. 2. Click on **Finite Automaton**. 3. **Add States**:

- Create states `q0` to `q3`.
- Set `q0` as **Initial**.
- Set `q3` as **Final**.

4. **Define Transitions**:

- `q0` -> `q1`: `a` (First 'a')
- `q0` -> `q0`: `b` (Reset)
- `q1` -> `q2`: `b` (First 'b')
- `q1` -> `q1`: `a` (Stay on 'a')

- `q2` -> `q3`: `b` (Second 'b' - Accept)
- `q2` -> `q1`: `a` (Back to 'a')
- `q3` -> `q0`: `b` (Reset)
- `q3` -> `q1`: `a` (Back to 'a')

5. **Verify**:
   - Test with `abb` (Accept), `aabb` (Accept), `babb` (Accept).
   - Test with `ab` (Reject), `ba` (Reject), `abba` (Reject).
6. **Save**:
   - Save as `13.jff` in `set_2/13`.

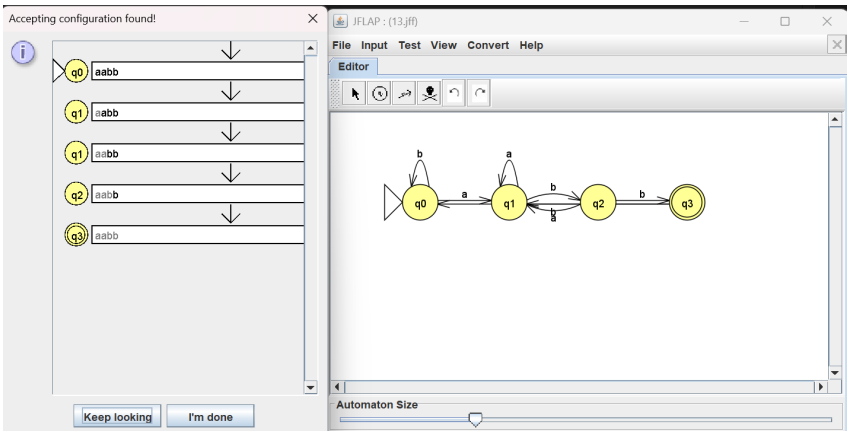

Figure 7: Automaton for Question 13

3.8 Methodology for Question 14

Draw a DFA for the language accepting strings ending with 'abba' over input alphabets $\Sigma = \{a, b\}$

1. Open JFLAP. 2. Click on **Finite Automaton**. 3. **Add States**:
   - Create states `q0` to `q4`.
   - Set `q0` as **Initial**.
   - Set `q4` as **Final**.
4. **Define Transitions**:
   - `q0` -> `q1`: `a` (First 'a')

- `q0` -> `q0`: `b` (Reset)
- `q1` -> `q2`: `b` (First 'b')
- `q1` -> `q1`: `a` (Stay on 'a')
- `q2` -> `q3`: `b` (Second 'b')
- `q2` -> `q1`: `a` (Back to 'a')
- `q3` -> `q4`: `a` (Second 'a' - Accept)
- `q3` -> `q0`: `b` (Reset)
- `q4` -> `q1`: `a` (Overlap 'a')
- `q4` -> `q2`: `b` (Overlap 'ab')

5. **Verify**:
   - Test with `abba` (Accept), `aabba` (Accept), `babba` (Accept).
   - Test with `abb` (Reject), `aba` (Reject), `abbab` (Reject).
6. **Save**:
   - Save as `14.jff` in `set_2/14`.

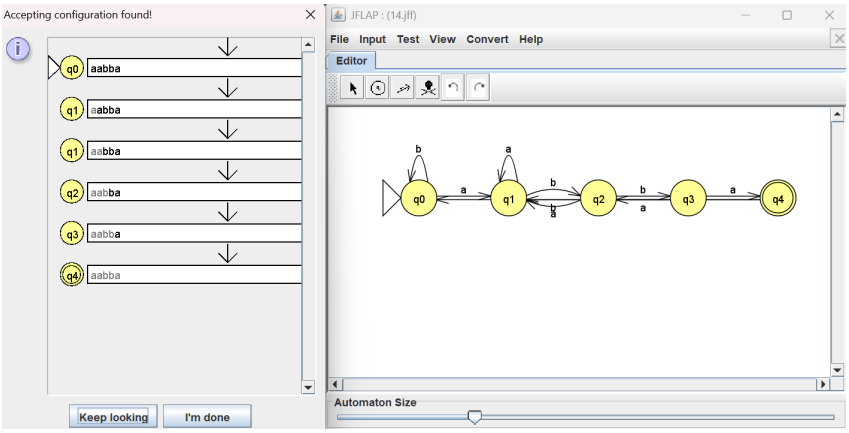

Figure 8: Automaton for Question 14

3.9 Methodology for Question 15

Draw a deterministic and non-deterministic finite automata which accept a string containing "the" anywhere in a string of a-z, e.g., "there" but not "those".

1. Open JFLAP. 2. Click on **Finite Automaton**. 3. **Add States**:

- Create states `q0` to `q3`.
- Set `q0` as **Initial**.
- Set `q3` as **Final**.

4. **Define Transitions**:
   - `q0` -> `q1`: `t`
   - `q0` -> `q0`: `[a-z] - t` (Any other letter)
   - `q1` -> `q2`: `h`
   - `q1` -> `q1`: `t` (Stay on 't')
   - `q1` -> `q0`: `[a-z] - t, h` (Reset)
   - `q2` -> `q3`: `e` (Found "the")
   - `q2` -> `q1`: `t` (Back to 't')
   - `q2` -> `q0`: `[a-z] - t, e` (Reset)
   - `q3` -> `q3`: `[a-z]` (Loop forever once found)
5. **Verify**:
   - Test with `there` (Accept), `breathe` (Accept), `the` (Accept).
   - Test with `those` (Reject), `teh` (Reject), `th` (Reject).
6. **Save**:
   - Save as `15.jff` in `set_2/15`.

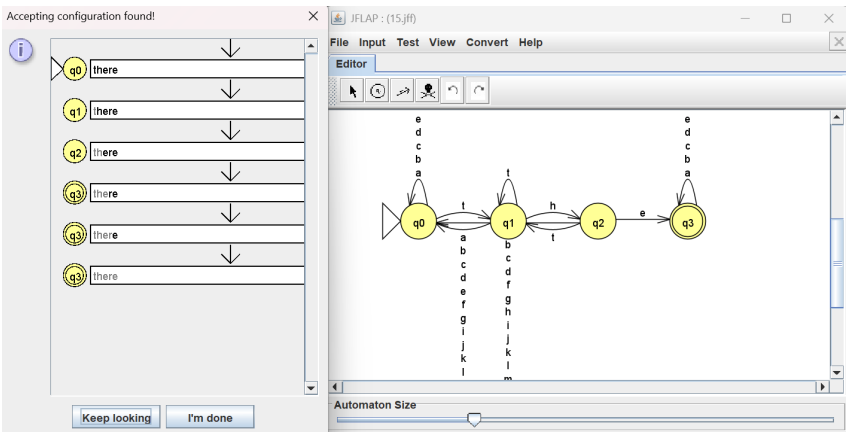

Figure 9: Automaton for Question 15

3.10 Methodology for Question 16

Draw a deterministic and non-deterministic finite automata which accept a string containing "ing" at the end of a string in a string of a-z, e.g., "anything" but not "anywhere".

1. Open JFLAP. 2. Click on **Finite Automaton**. 3. **Add States**:

- Create states `q0` to `q3`.
- Set `q0` as **Initial**.
- Set `q3` as **Final**.

4. **Define Transitions**:

- `q0` -> `q1`: `i`
- `q0` -> `q0`: `[a-z] - i` (Reset)
- `q1` -> `q2`: `n`
- `q1` -> `q1`: `i` (Stay on 'i')
- `q1` -> `q0`: `[a-z] - i, n` (Reset)
- `q2` -> `q3`: `g` (Found "ing" at end)
- `q2` -> `q1`: `i` (Back to 'i')
- `q2` -> `q0`: `[a-z] - i, g` (Reset)
- `q3` -> `q1`: `i` (Overlap 'i')
- `q3` -> `q0`: `[a-z] - i` (Reset, since it must be at the end)

5. **Verify**:

- Test with `anything` (Accept), `sing` (Accept), `going` (Accept).
- Test with `anywhere` (Reject), `in` (Reject), `singer` (Reject - 'ing' is not at end).

6. **Save**:

- Save as `16.jff` in `set_2/16`.

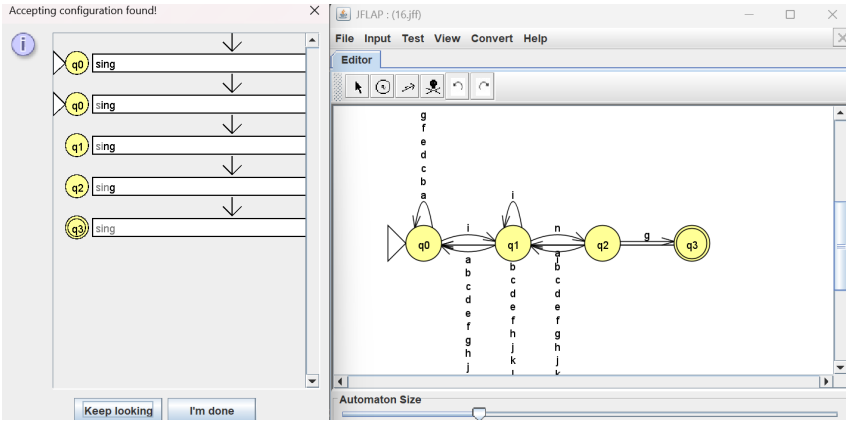

Figure 10: Automaton for Question 16

3.11 Methodology for Question 29

Design a Turing machine to accept a palindrome consisting of a's and b's of any length.

1. Open JFLAP. 2. Click on **Turing Machine** (Single Tape). 3. **Add States**:

- Create states `q0` to `q6`.
- Set `q0` as **Initial**.
- Set `q6` as **Final**.

4. **Define Transitions**:

- **Start**:
- `q0 -> q1: (a, ☐, R)` $(Read'a', markasblank, moveRight)$
- `q0 -> q4: (b, ☐, R)` $(Read'b', markasblank, moveRight)$
- `q0 -> q6: (☐, ☐, S)` $(Emptystringispalindrome, Accept)$
- `Move Right (after 'a'):`
- `q1 -> q1: (a, a, R)`
- `q1 -> q1: (b, b, R)`
- `q1 -> q2: (☐, ☐, L)` $(Foundend, moveLeft)$
- `Match End (for 'a'):`
- `q2 -> q3: (a, ☐, L)` $(Match'a', markblank, moveLeft)$
- `q2 -> q6: (☐, ☐, S)` $(Single'a'left, Accept)$
- `Return Left:`

- `q3 -> q3: (a, a, L)`
- `q3 -> q3: (b, b, L)`
- `q3 -> q0:` $(\square, \square, R)(Backtostart)$
- `Move Right (after 'b'):`
- `q4 -> q4: (a, a, R)`
- `q4 -> q4: (b, b, R)`
- `q4 -> q5:` $(\square, \square, L)(Foundend, moveLeft)$
- `Match End (for 'b'):`
- `q5 -> q3: (b,` $\square, L)(Match'b', markblank, moveLeft)$
- `q5 -> q6:` $(\square, \square, S)(Single'b'left, Accept)$

5. **Verify**:
   - Test with `aba` (Accept), `abba` (Accept), `a` (Accept), `b` (Accept).
   - Test with `ab` (Reject), `abb` (Reject).
6. **Save**:
   - Save as `29.jff` in `set_3/29`.

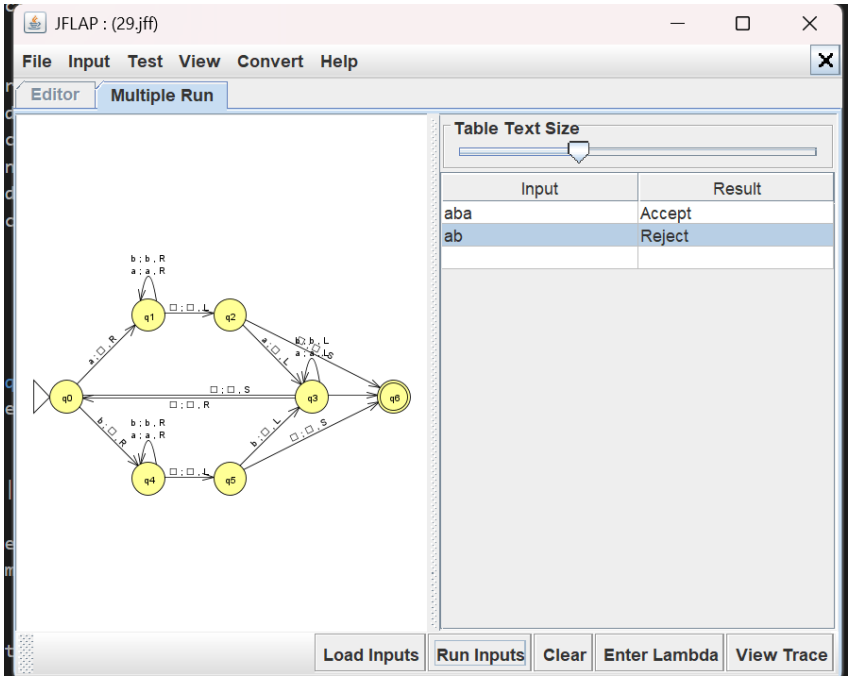

Figure 11: Automaton for Question 29

3.12 Methodology for Question 30

Construct a TM to accept the language $L = \{ww^R | w \in (a+b)^*\}$

1. Open JFLAP. 2. Click on **Turing Machine** (Single Tape). 3. **Add States**:

- Create states `q0` to `q6`.
- Set `q0` as **Initial**.
- Set `q6` as **Final**.

4. **Define Transitions**:

- This is essentially the same as the even-length palindrome logic, but strictly ww^R implies even length.
- **Start**:
- `q0` -> `q1`: `(a,` $\square, R$)
- `q0 -> q4:` `(b,` $\square, R$)
- `q0 -> q6:` ($\square, \square, S$)($Empty string is ww^R$ `where` $w = \epsilon$)
- `Move Right (after 'a'):`
- `q1 -> q1:` `(a, a, R)`
- `q1 -> q1:` `(b, b, R)`
- `q1 -> q2:` ($\square, \square, L$)
- `Match End (for 'a'):`
- `q2 -> q3:` `(a,` $\square, L$)
- `Return Left:`
- `q3 -> q3:` `(a, a, L)`
- `q3 -> q3:` `(b, b, L)`
- `q3 -> q0:` ($\square, \square, R$)
- `Move Right (after 'b'):`
- `q4 -> q4:` `(a, a, R)`
- `q4 -> q4:` `(b, b, R)`

- `q4 -> q5:` $(\square, \square, L)$
- `Match End (for 'b'):`
- `q5 -> q3:` $(b, \square, L)$

5. **Verify**:
   - Test with `abba` (Accept), `aaaa` (Accept), `bbaabb` (Accept).
   - Test with `aba` (Reject - odd length), `ab` (Reject), `a` (Reject).
6. **Save**:
   - Save as `30.jff` in `set_3/30`.

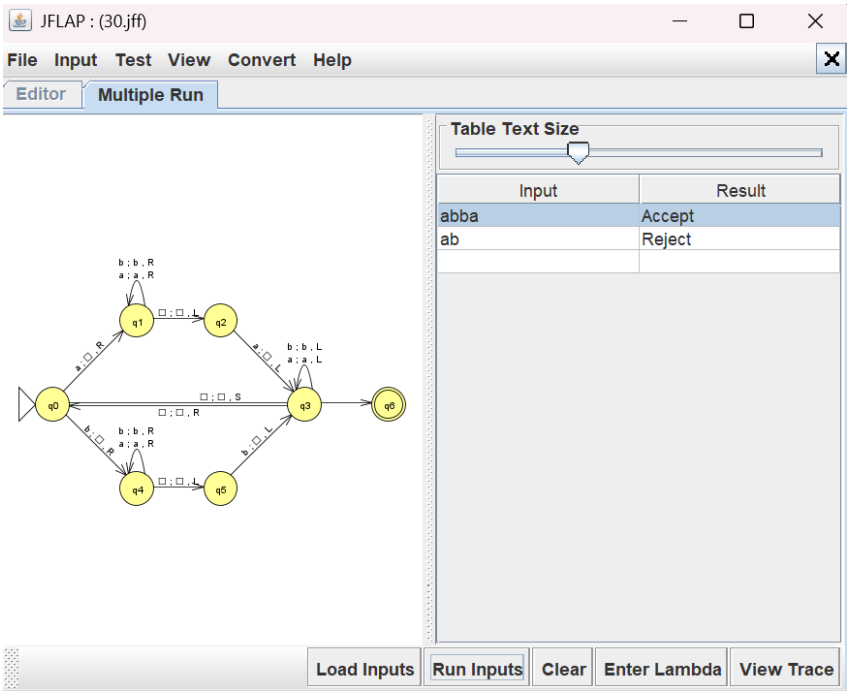

Figure 12: Automaton for Question 30

3.13 Methodology for Question 24

Design a Turing Machine to accept the language $L = \{0^n 1^n | n \geq 1\}$.

1. Open JFLAP. 2. Click on **Turing Machine** (Single Tape). 3. **Add States**:
   - Create states `q0` to `q4`.
   - Set `q0` as **Initial**.
   - Set `q4` as **Final**.
4. **Define Transitions**:

- **Start**:
- `q0` -> `q1`: `(0, X, R)` (Mark '0' with 'X', move Right)
- `q0` -> `q4`: `(Y, Y, R)` (If all 0s marked, check for completion)
- **Find Matching 1**:
- `q1` -> `q1`: `(0, 0, R)` (Skip 0s)
- `q1` -> `q1`: `(Y, Y, R)` (Skip Ys)
- `q1` -> `q2`: `(1, Y, L)` (Found matching '1', mark with 'Y', move Left)
- **Return to Start**:
- `q2` -> `q2`: `(0, 0, L)`
- `q2` -> `q2`: `(Y, Y, L)`
- `q2` -> `q0`: `(X, X, R)` (Back to start after X)
- **Final Check**:
- `q0` -> `q3`: `(Y, Y, R)` (All 0s handled, ensure no extra 1s)
- `q3` -> `q3`: `(Y, Y, R)`
- `q3` -> `q4`: $(\square, \square, S)(\textit{Accept})$

5. **Verify**:
   - Test with `01` (Accept), `0011` (Accept), `000111` (Accept).
   - Test with `001` (Reject), `011` (Reject), `0` (Reject), `1` (Reject).
6. **Save**:
   - Save as `24.jff` in `set_3/24`.

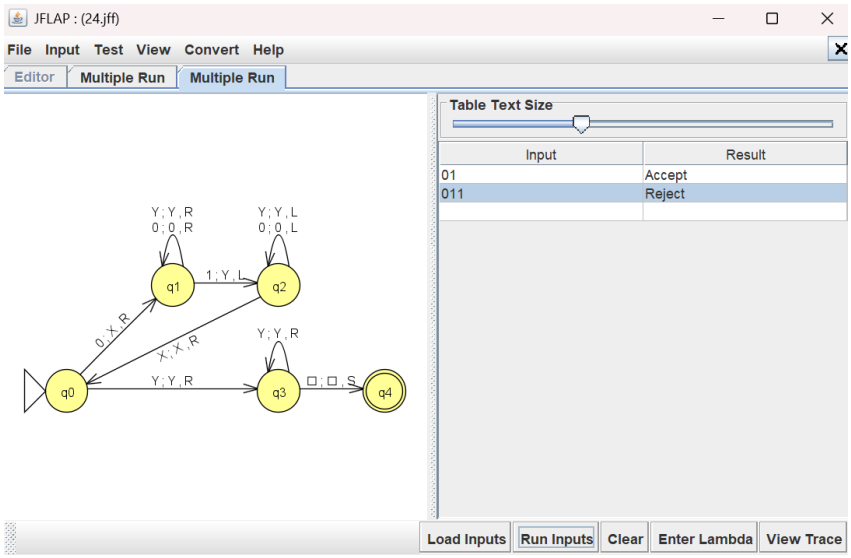

Figure 13: Automaton for Question 24

3.14 Methodology for Question 25

Design a Turing Machine to accept the language $L = \{0^n 1^n 2^n | n \geq 1\}$.

1. Open JFLAP. 2. Click on **Turing Machine** (Single Tape). 3. **Add States**:

- Create states `q0` to `q6`.
- Set `q0` as **Initial**.
- Set `q6` as **Final**.

4. **Define Transitions**:

- **Start**:
- `q0` -> `q1`: `(0, X, R)` (Mark '0', move Right)
- `q0` -> `q5`: `(Y, Y, R)` (If all 0s marked, check for completion)
- **Find Matching 1**:
- `q1` -> `q1`: `(0, 0, R)`
- `q1` -> `q1`: `(Y, Y, R)`
- `q1` -> `q2`: `(1, Y, R)` (Found '1', mark 'Y', move Right)
- **Find Matching 2**:
- `q2` -> `q2`: `(1, 1, R)`
- `q2` -> `q2`: `(Z, Z, R)`

- `q2` -> `q3`: `(2, Z, L)` (Found '2', mark 'Z', move Left)
- **Return to Start**:
- `q3` -> `q3`: `(0, 0, L)`
- `q3` -> `q3`: `(1, 1, L)`
- `q3` -> `q3`: `(2, 2, L)`
- `q3` -> `q3`: `(Y, Y, L)`
- `q3` -> `q3`: `(Z, Z, L)`
- `q3` -> `q0`: `(X, X, R)` (Back to start)
- **Final Check**:
- `q5` -> `q5`: `(Y, Y, R)`
- `q5` -> `q5`: `(Z, Z, R)`
- `q5` -> `q6`: $(\square, \square, S)(Accept)$

5. **Verify**:
   - Test with `012` (Accept), `001122` (Accept).
   - Test with `01` (Reject), `0122` (Reject), `0012` (Reject).
6. **Save**:
   - Save as `25.jff` in `set_3/25`.

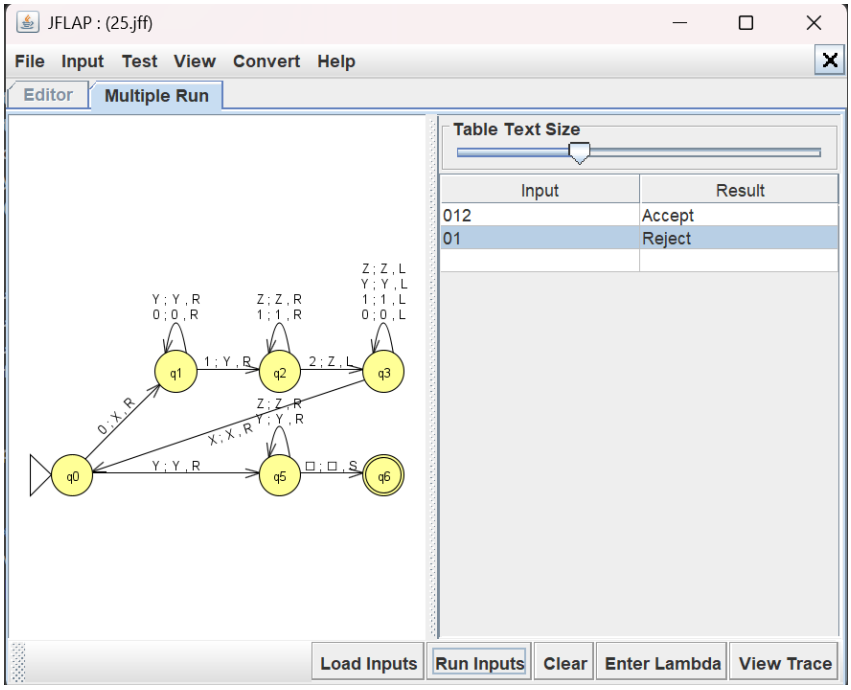

Figure 14: Automaton for Question 25

3.15 Methodology for Question 26

Construct a Turing Machine to accept the language $L = \{w|w \in (0+1)^*\}$ Containing the substring 001.

1. Open JFLAP. 2. Click on **Turing Machine** (Single Tape). 3. **Add States**:

- Create states `q0` to `q3`.
- Set `q0` as **Initial**.
- Set `q3` as **Final**.

4. **Define Transitions**:

- **Scan Right**:
- `q0` -> `q0`: `(1, 1, R)` (Skip 1s)
- `q0` -> `q1`: `(0, 0, R)` (Found first '0')
- `q1` -> `q1`: `(0, 0, R)` (Found second '0', stay in q1/q2 logic) - *Correction*: Better to have distinct states for '0', '00'.
- Let's refine:
- `q0` (Start):
- `1` -> `q0`, R

- `0` -> `q1`, R
- `q1` (Saw '0'):
- `1` -> `q0`, R (Reset)
- `0` -> `q2`, R (Saw '00')
- `q2` (Saw '00'):
- `0` -> `q2`, R (Stay in '00' state, e.g. '000')
- `1` -> `q3`, R (Saw '001', Accept)
- `q3` (Accept):
- `0` -> `q3`, R
- `1` -> `q3`, R
- $\square$– > `q3`, S

5. **Verify**:
   - Test with `001` (Accept), `1001` (Accept), `0001` (Accept).
   - Test with `00` (Reject), `01` (Reject), `111` (Reject).
6. **Save**:
   - Save as `26.jff` in `set_3/26`.

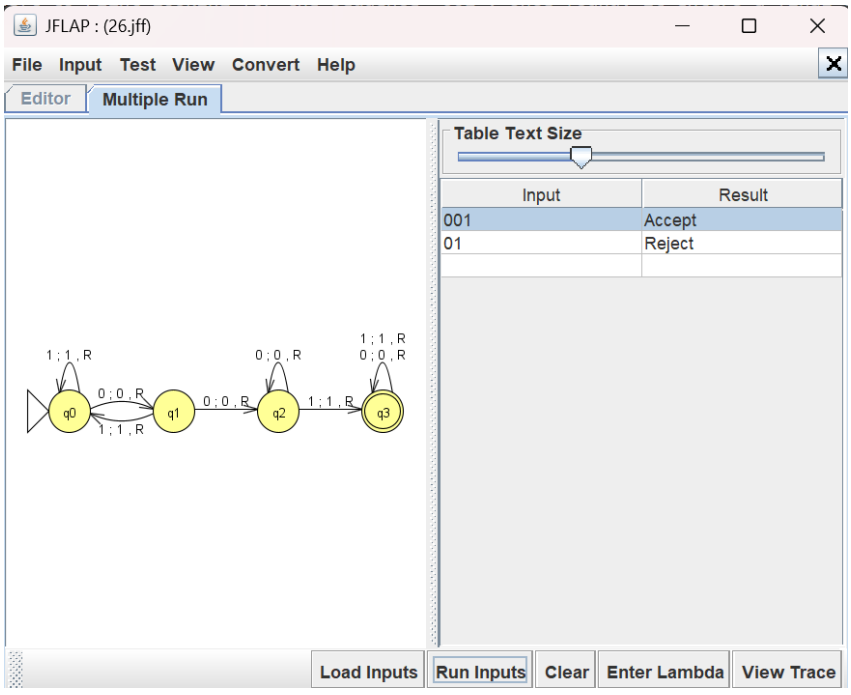

Figure 15: Automaton for Question 26

4 Results

This section presents acceptance tables and correctness verification for each question.

4.1 Results for Question 1

Design a non deterministic PDA for accepting the language $L = \{a^n b^n | n \geq 1\}$

Description

The PDA works by pushing 'a's onto the stack. When the first 'b' is encountered, it switches state and begins popping 'a's for each 'b'. If the stack is empty (only bottom marker Z remains) after processing all input, the string is accepted.

- **States**:
- `q0`: Start state. Transitions to `q1` on first 'a'.
- `q1`: Pushes 'a's.
- `q2`: Pops 'a's on 'b's.
- `q3`: Final state (Accept).

Transitions

- $\delta(q0, a, Z) = \{(q1, aZ)\}$
- $\delta(q1, a, a) = \{(q1, aa)\}$
- $\delta(q1, b, a) = \{(q2, \lambda)\}$
- $\delta(q2, b, a) = \{(q2, \lambda)\}$
- $\delta(q2, \lambda, Z) = \{(q3, Z)\}$

Test Strings

String	Result	Explanation
`ab`	**Accept**	1 'a', 1 'b'. Stack matches.
`aabb`	**Accept**	2 'a's, 2 'b's.
`aaabbb`	**Accept**	3 'a's, 3 'b's.
`a`	**Reject**	Missing 'b'.
`b`	**Reject**	Missing 'a'.
`aabbb`	**Reject**	More 'b's than 'a's.
`aaabb`	**Reject**	More 'a's than 'b's.

4.2 Results for Question 2

Construct a DFA for $L = \{ab^n a^m : n \geq 2, m \geq 3\}$

Description

The DFA accepts strings starting with a single 'a', followed by at least two 'b's, and ending with at least three 'a's.

- **States**:
- `q0`: Start. Expects 'a'.
- `q1`: Expects first 'b'.
- `q2`: Expects second 'b'.
- `q3`: Satisfied $n \geq 2$. Loops on 'b'. Expects first 'a' of suffix.
- `q4`: Expects second 'a'.
- `q5`: Expects third 'a'.
- `q6`: Final state. Satisfied $m \geq 3$. Loops on 'a'.

Transitions

- $\delta(q0, a) = q1$
- $\delta(q1, b) = q2$
- $\delta(q2, b) = q3$
- $\delta(q3, b) = q3$
- $\delta(q3, a) = q4$
- $\delta(q4, a) = q5$
- $\delta(q5, a) = q6$
- $\delta(q6, a) = q6$

Test Strings

String	Result	Explanation
`abbaaa`	**Accept**	Min requirement: 1 a, 2 b's, 3 a's.
`abbbaaa`	**Accept**	3 b's ($n = 3$), 3 a's.
`abbaaaa`	**Accept**	2 b's, 4 a's ($m = 4$).
`abaaa`	**Reject**	Only 1 b ($n = 1 < 2$).
`abbaa`	**Reject**	Only 2 a's ($m = 2 < 3$).
`bbaaa`	**Reject**	Starts with b.
`abbaaba`	**Reject**	'b' after 'a's.

4.3 Results for Question 3

Draw a DFA for the language accepting strings ending with '0011' over input alphabets $\Sigma = \{0, 1\}$

Description

The DFA tracks the progress towards the suffix '0011'. Any interruption resets the state to the appropriate prefix match.

- **States**:
- `q0`: Start. No prefix matched.
- `q1`: Matched '0'.
- `q2`: Matched '00'.
- `q3`: Matched '001'.
- `q4`: Matched '0011' (Final).

Transitions

State	Input 0	Input 1
q0	q1	q0
q1	q2	q0
q2	q2	q3
q3	q1	q4
q4	q1	q0

Test Strings

String	Result	Explanation
`0011`	**Accept**	Ends with 0011.
`110011`	**Accept**	Ends with 0011.
`00011`	**Accept**	Ends with 0011.
`001`	**Reject**	Incomplete suffix.
`00110`	**Reject**	Ends with 0, not 1.
`011`	**Reject**	Missing leading 0s.

4.4 Results for Question 4

Construct npda's that accept the following languages on $\Sigma = \{a, b, c\}$. $L = \{a^n b^m c^{n+m} : n \geq 0, m \geq 0\}$

Description

The PDA pushes 'a's and then 'b's onto the stack. For every 'c' encountered, it pops one symbol from the stack. Since 'b's are pushed after 'a's, they are popped first. The total number of 'c's must equal the total number of 'a's and 'b's combined.

- **States**:
- `q0`: Start. Handles empty string or transitions to pushing states.
- `q1`: Pushes 'a's.
- `q2`: Pushes 'b's.
- `q3`: Pops symbols ('b's then 'a's) for each 'c'.
- `q4`: Final state.

Transitions

- $\delta(q0, \lambda, Z) = \{(q4, Z)\}$ (Accept ϵ)
- $\delta(q0, a, Z) = \{(q1, aZ)\}$
- $\delta(q0, b, Z) = \{(q2, bZ)\}$
- $\delta(q1, a, a) = \{(q1, aa)\}$
- $\delta(q1, b, a) = \{(q2, ba)\}$
- $\delta(q1, c, a) = \{(q3, \lambda)\}$

- $\delta(q2, b, b) = \{(q2, bb)\}$
- $\delta(q2, c, b) = \{(q3, \lambda)\}$
- $\delta(q3, c, b) = \{(q3, \lambda)\}$
- $\delta(q3, c, a) = \{(q3, \lambda)\}$
- $\delta(q3, \lambda, Z) = \{(q4, Z)\}$

Test Strings

String	Result	Explanation
`abc`	**Accept**	1 a + 1 b = 2 c's.
`aabbcc`	**Accept**	2 a + 2 b = 4 c's.
`ac`	**Accept**	1 a + 0 b = 1 c.
`bc`	**Accept**	0 a + 1 b = 1 c.
`ab`	**Reject**	Missing c's.
`aabc`	**Reject**	2 a + 1 b != 1 c.
`abcc`	**Reject**	1 a + 1 b != 2 c's.

4.5 Results for Question 5

Construction of a minimal DFA accepting set of strings over $\{a, b\}$ in which every 'a' is never be followed by 'bb'.

Description

The DFA rejects any string containing the substring "abb". It accepts all other strings.

- **States**:
- `q0`: Start. Safe state.
- `q1`: Just saw 'a'.
- `q2`: Just saw 'ab'.
- `q3`: Saw 'abb' (Trap state).

Transitions

State	Input a	Input b
q0	q1	q0
q1	q1	q2
q2	q1	q3
q3	q3	q3

Test Strings

String	Result	Explanation
`ab`	**Accept**	'a' followed by single 'b'.
`aba`	**Accept**	'a' followed by 'b' then 'a'.
`bba`	**Accept**	No 'a' before 'bb'.
`aab`	**Accept**	'a' followed by 'a' then 'b'.
`abb`	**Reject**	Contains 'abb'.
`aabb`	**Reject**	Contains 'abb'.
`babb`	**Reject**	Contains 'abb'.

4.6 Results for Question 12

Construct an NFA that accepts the language $\{ab, abc\}^*$.

Description

The NFA accepts strings formed by concatenating 'ab' and 'abc' any number of times. The start state is also the final state to accept the empty string.

- **States**:
- `q0`: Start/Final state. Ready to process next 'ab' or 'abc'.
- `q1`: Saw 'a'. Could be start of 'ab' or 'abc'.
- `q2`: Saw 'ab' (as prefix of 'abc').

Transitions

- $\delta(q0, a) = \{q1\}$
- $\delta(q1, b) = \{q0, q2\}$ (Nondeterminism here: return to q0 for 'ab', or go to q2 for 'abc')
- $\delta(q2, c) = \{q0\}$

Test Strings

String	Result	Explanation
`ab`	**Accept**	Matches 'ab'.
`abc`	**Accept**	Matches 'abc'.
`abcab`	**Accept**	Matches 'abc' then 'ab'.
`ababc`	**Accept**	Matches 'ab' then 'abc'.
`a`	**Reject**	Incomplete.
`ac`	**Reject**	Invalid sequence.
`abb`	**Reject**	'ab' followed by 'b' is invalid.

4.7 Results for Question 13

Draw a DFA for the language accepting strings ending with 'abb' over input alphabets $\Sigma = \{a, b\}$

Description

The DFA tracks the suffix 'abb'.

- **States**:
- `q0`: Start. No prefix matched.
- `q1`: Matched 'a'.
- `q2`: Matched 'ab'.
- `q3`: Matched 'abb' (Final).

Transitions

State	Input a	Input b
q0	q1	q0
q1	q1	q2
q2	q1	q3
q3	q1	q0

Test Strings

String	Result	Explanation
`abb`	**Accept**	Ends with abb.
`aabb`	**Accept**	Ends with abb.
`babb`	**Accept**	Ends with abb.
`ab`	**Reject**	Incomplete suffix.
`abba`	**Reject**	Ends with a.
`abbb`	**Reject**	Ends with bbb.

4.8 Results for Question 14

Draw a DFA for the language accepting strings ending with 'abba' over input alphabets $\Sigma = \{a, b\}$

Description

The DFA tracks the suffix 'abba'.

- **States**:
- `q0`: Start. No prefix matched.
- `q1`: Matched 'a'.
- `q2`: Matched 'ab'.
- `q3`: Matched 'abb'.
- `q4`: Matched 'abba' (Final).

Transitions

State	Input a	Input b
q0	q1	q0
q1	q1	q2
q2	q1	q3
q3	q4	q0
q4	q1	q2

Test Strings

String	Result	Explanation
`abba`	**Accept**	Ends with abba.
`aabba`	**Accept**	Ends with abba.
`babba`	**Accept**	Ends with abba.
`abb`	**Reject**	Incomplete suffix.
`abbab`	**Reject**	Ends with b.
`abbb`	**Reject**	Ends with bbb.

4.9 Results for Question 15

Draw a deterministic and non-deterministic finite automata which accept a string containing “the” anywhere in a string of a-z.

Description

The DFA looks for the substring "the". Once found, it enters a final state and stays there.

- **States**:
- `q0`: Start. No part of "the" matched.
- `q1`: Matched ’t’.
- `q2`: Matched ’th’.
- `q3`: Matched "the" (Final).

Transitions

State	Input t	Input h	Input e	Other [a-z]
q0	q1	q0	q0	q0
q1	q1	q2	q0	q0
q2	q1	q0	q3	q0
q3	q3	q3	q3	q3

Test Strings

String	Result	Explanation
`there`	**Accept**	Contains "the".
`breathe`	**Accept**	Contains "the".
`the`	**Accept**	Is "the".
`those`	**Reject**	"tho", not "the".
`teh`	**Reject**	"teh", not "the".
`th`	**Reject**	Incomplete.

4.10 Results for Question 16

Draw a deterministic and non-deterministic finite automata which accept a string containing “ing” at the end of a string in a string of a-z.

Description

The DFA tracks the suffix "ing". It is similar to seeking a substring, but if any character follows "ing", it must reset or partially reset to ensure "ing" is the *very last* thing.

- **States**:
- `q0`: Start. No prefix matched.
- `q1`: Matched ’i’.
- `q2`: Matched ’in’.
- `q3`: Matched "ing" (Final).

Transitions

State	Input i	Input n	Input g	Other [a-z]
q0	q1	q0	q0	q0
q1	q1	q2	q0	q0
q2	q1	q0	q3	q0
q3	q1	q0	q0	q0

Test Strings

String	Result	Explanation
`anything`	**Accept**	Ends with "ing".
`sing`	**Accept**	Ends with "ing".
`going`	**Accept**	Ends with "ing".
`anywhere`	**Reject**	Ends with "ere".
`singer`	**Reject**	Ends with "er".
`in`	**Reject**	Incomplete.

4.11 Results for Question 29

Design a Turing machine to accept a palindrome consisting of a's and b's of any length.

Description

The TM matches the first character with the last character, erasing both. It repeats this process until the string is empty or has one character left.

- **States**:
- `q0`: Start. Reads first char.
- `q1`: Moves right to find end (after reading 'a').
- `q2`: Checks last char (expecting 'a').
- `q3`: Returns to start (moving left).
- `q4`: Moves right to find end (after reading 'b').
- `q5`: Checks last char (expecting 'b').
- `q6`: Final state (Accept).

Transitions

- $\delta(q0, a) = (q1, \square, R)$
- $\delta(q0, b) = (q4, \square, R)$
- $\delta(q0, \square) = (q6, \square, S)$
- ... (See JFLAP file for full set)

Test Strings

String	Result	Explanation
aba	**Accept**	Palindrome.
abba	**Accept**	Palindrome.
a	**Accept**	Single char is palindrome.
ab	**Reject**	Ends don't match.
abb	**Reject**	Ends don't match.

4.12 Results for Question 30

Construct a TM to accept the language $L = \{ww^R | w \in (a+b)^*\}$

Description

This language represents even-length palindromes. The TM matches the first and last characters, erasing them, and repeats. It rejects odd-length strings (unlike the general palindrome TM which accepts the last single character).

- **States**:
- q0: Start. Reads first char.
- q1: Moves right to find end (after reading 'a').
- q2: Checks last char (expecting 'a').
- q3: Returns to start.
- q4: Moves right to find end (after reading 'b').
- q5: Checks last char (expecting 'b').
- q6: Final state (Accept).

Transitions

- $\delta(q0, a) = (q1, \square, R)$
- $\delta(q0, b) = (q4, \square, R)$
- $\delta(q0, \square) = (q6, \square, S)$
- ... (See JFLAP file for full set)

Test Strings

String	Result	Explanation
`abba`	**Accept**	$w = ab, w^R = ba.$
`aaaa`	**Accept**	$w = aa, w^R = aa.$
`aba`	**Reject**	Odd length.
`ab`	**Reject**	Not a palindrome.

4.13 Results for Question 24

Design a Turing Machine to accept the language $L = \{0^n 1^n | n \geq 1\}$.

Description

The TM marks '0's with 'X' and corresponding '1's with 'Y'. It zig-zags between the beginning of the 0s and the beginning of the 1s.

- **States**:
- `q0`: Start. Marks next '0'.
- `q1`: Moves right to find first '1'.
- `q2`: Moves left to find last 'X'.
- `q3`: Checks if any '1's remain after all '0's are marked.
- `q4`: Final state (Accept).

Transitions

- $\delta(q0, 0) = (q1, X, R)$
- $\delta(q1, 1) = (q2, Y, L)$
- ... (See JFLAP file for full set)

Test Strings

String	Result	Explanation
`01`	**Accept**	1 zero, 1 one.
`0011`	**Accept**	2 zeros, 2 ones.
`001`	**Reject**	More zeros.
`011`	**Reject**	More ones.

4.14 Results for Question 25

Design a Turing Machine to accept the language $L = \{0^n 1^n 2^n | n \geq 1\}$.

Description

The TM marks '0' with 'X', finds the first '1' and marks it 'Y', then finds the first '2' and marks it 'Z'. It repeats this cycle.

- **States**:
- `q0`: Start. Marks next '0'.
- `q1`: Moves right to find first '1'.
- `q2`: Moves right to find first '2'.
- `q3`: Moves left to find last 'X'.
- `q5`: Checks if any '1's or '2's remain after all '0's are marked.
- `q6`: Final state (Accept).

Transitions

- $\delta(q0, 0) = (q1, X, R)$
- $\delta(q1, 1) = (q2, Y, R)$
- $\delta(q2, 2) = (q3, Z, L)$
- ... (See JFLAP file for full set)

Test Strings

String	Result	Explanation
`012`	**Accept**	1 of each.
`001122`	**Accept**	2 of each.
`01`	**Reject**	Missing 2.
`0122`	**Reject**	Extra 2.

4.15 Results for Question 26

Construct a Turing Machine to accept the language $L = \{w | w \in (0 + 1)^*\}$ Containing the substring 001.

Description

The TM scans the tape from left to right looking for the sequence '001'. Once found, it enters a final state and halts (or loops in final state).

- **States**:
- `q0`: Start. No part of '001' matched.
- `q1`: Matched '0'.
- `q2`: Matched '00'.
- `q3`: Matched '001' (Final).

Transitions

- $\delta(q0, 1) = (q0, 1, R)$
- $\delta(q0, 0) = (q1, 0, R)$
- $\delta(q1, 1) = (q0, 1, R)$
- $\delta(q1, 0) = (q2, 0, R)$
- $\delta(q2, 0) = (q2, 0, R)$
- $\delta(q2, 1) = (q3, 1, R)$
- $\delta(q3, 0/1) = (q3, 0/1, R)$

Test Strings

String	**Result**	**Explanation**
`001`	**Accept**	Contains 001.
`1001`	**Accept**	Contains 001.
`0001`	**Accept**	Contains 001.
`01`	**Reject**	No 001.
`00`	**Reject**	Incomplete.

5 Discussion

This section contains per-question interpretation of correctness and conceptual insights.

5.1 Discussion for Question 1

The NPDA utilizes the stack to maintain the count of 'a's. For every 'a' read, one is pushed onto the stack. For every 'b', one 'a' is popped. The acceptance condition relies on the stack being empty (or containing only the bottom marker) after processing the entire string, ensuring the number of 'a's exactly matches the number of 'b's. The non-determinism allows for the transition from pushing to popping mode.

5.2 Discussion for Question 2

This DFA enforces a strict sequence: a single 'a', followed by at least two 'b's, and then at least three 'a's. The states are arranged sequentially to count the mandatory occurrences. Loops on the intermediate states allow for $n > 2$ and $m > 3$, ensuring that extra 'b's or 'a's do not break the acceptance condition as long as the minimum counts are met.

5.3 Discussion for Question 3

The DFA maintains a state corresponding to the longest suffix of the input read so far that matches a prefix of '0011'. If a character breaks the sequence, the transition returns to the state representing the longest valid prefix preserved (e.g., receiving a '0' after '001' transitions back to the '00' state). This ensures that the machine always correctly identifies when the specific suffix '0011' concludes the string.

5.4 Discussion for Question 4

This NPDA demonstrates adding two counts on the stack. First, 'a's are pushed, then 'b's are pushed. When 'c's are encountered, the machine pops 'b's first, and once 'b's are exhausted, it begins popping 'a's. This correctly verifies that the total number of 'c's equals the sum of 'a's and 'b's ($n + m$). The transition to the popping state is non-deterministic or triggered by the first 'c'.

5.5 Discussion for Question 5

The DFA acts as a filter for the forbidden substring 'abb'. It tracks the sequence 'a', 'ab'. If 'b' follows 'ab', it enters a trap state (dead state) from which it never escapes. All other states are accepting, ensuring that only strings containing the sequence 'abb' are rejected, while all others are accepted.

5.6 Discussion for Question 12

The NFA handles the ambiguity where 'ab' can be a standalone unit or the prefix of 'abc'. The non-determinism at the end of 'ab' allows the machine to either loop back to the start (accepting 'ab') or continue to match 'c' (accepting 'abc'). This simplifies the design compared to a DFA, which would require lookahead-like states.

5.7 Discussion for Question 13

Similar to other suffix-matching DFAs, this machine tracks the progress towards 'abb'. The transition on 'a' from the 'ab' state goes back to the 'a' state, preserving the potential start of a new pattern. The final state represents having just completed 'abb'.

5.8 Discussion for Question 14

This DFA recognizes the suffix 'abba'. The final state is not a trap; receiving an 'a' in the final state transitions back to the state representing 'a' (start of pattern), and 'b' transitions to the state for 'ab' (overlap), allowing detection of overlapping occurrences or subsequent valid suffixes.

5.9 Discussion for Question 15

This automaton searches for the substring 'the'. Unlike suffix matching, once the final state is reached (meaning 'the' has been seen), the machine loops in that accepting state regardless of future input. This effectively "latches" onto the acceptance once the criteria is met.

5.10 Discussion for Question 16

This DFA ensures the string *ends* with 'ing'. If 'ing' is found, any subsequent non-matching character must reset the search, ensuring the pattern is strictly at the end of the string. For example, 'sing' is accepted, but 'singer' is rejected because the 'er' moves the machine out of the final state.

5.11 Discussion for Question 29

The Turing Machine operates by matching the first and last symbols. It marks the first symbol, moves to the end to verify the corresponding last symbol, marks it, and repeats. This recursive matching from outside in verifies the palindrome property. It accepts both even and odd length palindromes (where one center character remains).

5.12 Discussion for Question 30

This TM verifies even-length palindromes (ww^R). It follows the same logic as the general palindrome machine but strictly rejects if a single middle character remains. It requires the tape to be fully cleared (marked) in pairs, ensuring the total length is even.

5.13 Discussion for Question 24

The TM uses a zig-zag approach. It marks a '0' at the start, moves right to find the first unmarked '1', marks it, and returns to the start. This one-to-one mapping ensures equal counts of 0s and 1s. If it runs out of one symbol while the other remains, it rejects.

5.14 Discussion for Question 25

5.15 Results for Question 30

Construct a TM to accept the language $L = \{ww^R | w \in (a+b)^*\}$

Description

This language represents even-length palindromes. The TM matches the first and last characters, erasing them, and repeats. It rejects odd-length strings (unlike the general palindrome TM which accepts the last single character).

- **States**:
- `q0`: Start. Reads first char.
- `q1`: Moves right to find end (after reading 'a').
- `q2`: Checks last char (expecting 'a').
- `q3`: Returns to start.
- `q4`: Moves right to find end (after reading 'b').
- `q5`: Checks last char (expecting 'b').
- `q6`: Final state (Accept).

Transitions

- $\delta(q0, a) = (q1, \square, R)$
- $\delta(q0, b) = (q4, \square, R)$

- $\delta(q0, \square) = (q6, \square, S)$
- ... (See JFLAP file for full set)

Test Strings

String	Result	Explanation
`abba`	**Accept**	$w = ab, w^R = ba.$
`aaaa`	**Accept**	$w = aa, w^R = aa.$
`aba`	**Reject**	Odd length.
`ab`	**Reject**	Not a palindrome.

5.16 Results for Question 24

Design a Turing Machine to accept the language $L = \{0^n 1^n | n \geq 1\}$.

Description

The TM marks '0's with 'X' and corresponding '1's with 'Y'. It zig-zags between the beginning of the 0s and the beginning of the 1s.

- **States**:
- `q0`: Start. Marks next '0'.
- `q1`: Moves right to find first '1'.
- `q2`: Moves left to find last 'X'.
- `q3`: Checks if any '1's remain after all '0's are marked.
- `q4`: Final state (Accept).

Transitions

- $\delta(q0, 0) = (q1, X, R)$
- $\delta(q1, 1) = (q2, Y, L)$
- ... (See JFLAP file for full set)

Test Strings

String	Result	Explanation
01	**Accept**	1 zero, 1 one.
0011	**Accept**	2 zeros, 2 ones.
001	**Reject**	More zeros.
011	**Reject**	More ones.

5.17 Results for Question 25

Design a Turing Machine to accept the language $L = \{0^n 1^n 2^n | n \geq 1\}$.

Description

The TM marks '0' with 'X', finds the first '1' and marks it 'Y', then finds the first '2' and marks it 'Z'. It repeats this cycle.

- **States**:
- q0: Start. Marks next '0'.
- q1: Moves right to find first '1'.
- q2: Moves right to find first '2'.
- q3: Moves left to find last 'X'.
- q5: Checks if any '1's or '2's remain after all '0's are marked.
- q6: Final state (Accept).

Transitions

- $\delta(q0, 0) = (q1, X, R)$
- $\delta(q1, 1) = (q2, Y, R)$
- $\delta(q2, 2) = (q3, Z, L)$
- ... (See JFLAP file for full set)

Test Strings

String	Result	Explanation
012	**Accept**	1 of each.
001122	**Accept**	2 of each.
01	**Reject**	Missing 2.
0122	**Reject**	Extra 2.

5.18 Results for Question 26

Construct a Turing Machine to accept the language $L = \{w|w \in (0+1)^*\}$ Containing the substring 001.

Description

The TM scans the tape from left to right looking for the sequence '001'. Once found, it enters a final state and halts (or loops in final state).

- **States**:
- `q0`: Start. No part of '001' matched.
- `q1`: Matched '0'.
- `q2`: Matched '00'.
- `q3`: Matched '001' (Final).

Transitions

- $\delta(q0, 1) = (q0, 1, R)$
- $\delta(q0, 0) = (q1, 0, R)$
- $\delta(q1, 1) = (q0, 1, R)$
- $\delta(q1, 0) = (q2, 0, R)$
- $\delta(q2, 0) = (q2, 0, R)$
- $\delta(q2, 1) = (q3, 1, R)$
- $\delta(q3, 0/1) = (q3, 0/1, R)$

Test Strings

String	Result	Explanation
`001`	**Accept**	Contains 001.
`1001`	**Accept**	Contains 001.
`0001`	**Accept**	Contains 001.
`01`	**Reject**	No 001.
`00`	**Reject**	Incomplete.

6 Discussion

This section contains per-question interpretation of correctness and conceptual insights.

6.1 Discussion for Question 1

The NPDA utilizes the stack to maintain the count of 'a's. For every 'a' read, one is pushed onto the stack. For every 'b', one 'a' is popped. The acceptance condition relies on the stack being empty (or containing only the bottom marker) after processing the entire string, ensuring the number of 'a's exactly matches the number of 'b's. The non-determinism allows for the transition from pushing to popping mode.

6.2 Discussion for Question 2

This DFA enforces a strict sequence: a single 'a', followed by at least two 'b's, and then at least three 'a's. The states are arranged sequentially to count the mandatory occurrences. Loops on the intermediate states allow for $n > 2$ and $m > 3$, ensuring that extra 'b's or 'a's do not break the acceptance condition as long as the minimum counts are met.

6.3 Discussion for Question 3

The DFA maintains a state corresponding to the longest suffix of the input read so far that matches a prefix of '0011'. If a character breaks the sequence, the transition returns to the state representing the longest valid prefix preserved (e.g., receiving a '0' after '001' transitions back to the '00' state). This ensures that the machine always correctly identifies when the specific suffix '0011' concludes the string.

6.4 Discussion for Question 4

This NPDA demonstrates adding two counts on the stack. First, 'a's are pushed, then 'b's are pushed. When 'c's are encountered, the machine pops 'b's first, and once 'b's are exhausted, it begins popping 'a's. This correctly verifies that the total number of 'c's equals the sum of 'a's and 'b's ($n + m$). The transition to the popping state is non-deterministic or triggered by the first 'c'.

6.5 Discussion for Question 5

The DFA acts as a filter for the forbidden substring 'abb'. It tracks the sequence 'a', 'ab'. If 'b' follows 'ab', it enters a trap state (dead state) from which it never escapes. All other states are accepting, ensuring that only strings containing the sequence 'abb' are rejected, while all others are accepted.

6.6 Discussion for Question 12

The NFA handles the ambiguity where 'ab' can be a standalone unit or the prefix of 'abc'. The non-determinism at the end of 'ab' allows the machine to either loop back to the start (accepting 'ab') or continue to match 'c' (accepting 'abc'). This simplifies the design compared to a DFA, which would require lookahead-like states.

6.7 Discussion for Question 13

Similar to other suffix-matching DFAs, this machine tracks the progress towards 'abb'. The transition on 'a' from the 'ab' state goes back to the 'a' state, preserving the potential start of a new pattern. The final state represents having just completed 'abb'.

6.8 Discussion for Question 14

This DFA recognizes the suffix 'abba'. The final state is not a trap; receiving an 'a' in the final state transitions back to the state representing 'a' (start of pattern), and 'b' transitions to the state for 'ab' (overlap), allowing detection of overlapping occurrences or subsequent valid suffixes.

6.9 Discussion for Question 15

This automaton searches for the substring 'the'. Unlike suffix matching, once the final state is reached (meaning 'the' has been seen), the machine loops in that accepting state regardless of future input. This effectively "latches" onto the acceptance once the criteria is met.

6.10 Discussion for Question 16

This DFA ensures the string *ends* with 'ing'. If 'ing' is found, any subsequent non-matching character must reset the search, ensuring the pattern is strictly at the end of the string. For example, 'sing' is accepted, but 'singer' is rejected because the 'er' moves the machine out of the final state.

6.11 Discussion for Question 29

The Turing Machine operates by matching the first and last symbols. It marks the first symbol, moves to the end to verify the corresponding last symbol, marks it, and repeats. This recursive matching from outside in verifies the palindrome property. It accepts both even and odd length palindromes (where one center character remains).

6.12 Discussion for Question 30

This TM verifies even-length palindromes (ww^R). It follows the same logic as the general palindrome machine but strictly rejects if a single middle character remains. It requires the tape to be fully cleared (marked) in pairs, ensuring the total length is even.

6.13 Discussion for Question 24

The TM uses a zig-zag approach. It marks a '0' at the start, moves right to find the first unmarked '1', marks it, and returns to the start. This one-to-one mapping ensures equal counts of 0s and 1s. If it runs out of one symbol while the other remains, it rejects.

6.14 Discussion for Question 25

This TM extends the counting logic to three symbols. It marks a '0', searches for a '1' to mark, then searches for a '2' to mark, before returning to the start. This ensures n instances of each symbol appear in the correct order ($0^n1^n2^n$).

6.15 Discussion for Question 26

The TM scans the tape linearly for the pattern '001'. Upon finding the sequence '0', '0', '1' consecutively, it enters the final accepting state and halts. The machine does not need to consume the entire input; finding the substring once is sufficient for acceptance.

7 Comparative Analysis

The following table summarizes the key differences between the computational models explored in this assignment. It highlights the hierarchy of power, memory capabilities, and the class of languages each model can recognize.

Model	**Memory Mechanism**	**Language Class**	**Key Characteristic**
DFA / NFA	Finite States (No auxiliary memory)	Regular Languages	Limited to patterns with fixed memory requirements (e.g., modulo counting, fixed substrings).
PDA	Finite States + Infinite Stack (LIFO)	Context-Free Languages	Can count two related quantities or handle nested structures (e.g., balanced parentheses, $a^n b^n$).
TM	Finite States + Infinite Tape (Read/Write)	Recursively Enumerable Languages	Capable of arbitrary computation, including multiple comparisons and complex pattern recognition (e.g., $0^n 1^n 2^n$, ww^R).

8 Conclusion

This assignment provided a comprehensive practical application of automata theory, traversing the Chomsky hierarchy from Regular to Recursively Enumerable languages. Through the design and verification of Finite Automata, Pushdown Automata, and Turing Machines, we observed the increasing computational power required to solve more complex problems.

The transition from DFAs to PDAs highlighted the necessity of stack memory for matching counts, while the move to Turing Machines demonstrated the power of an infinite read/write tape for solving problems beyond the scope of context-free grammars. The use of JFLAP for simulation was instrumental in visualizing state transitions and verifying the correctness of each model against edge cases.

References

[1] Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2006). *Introduction to Automata Theory, Languages, and Computation* (3rd ed.). Pearson.

[2] Sipser, M. (2012). *Introduction to the Theory of Computation* (3rd ed.). Cengage Learning.

[3] Linz, P. (2016). *An Introduction to Formal Languages and Automata* (6th ed.). Jones & Bartlett Learning.