**Theory of computation Practical**

**Practical 1**

**Aim:**

**Design a Program for creating machine that accepts the string always ending with 101.**

**Code:**

class StateMachine:

def \_\_init\_\_(self):

self.state = 'q0'

def process\_input(self, input\_string):

for symbol in input\_string:

if self.state == 'q0' and symbol == '1':

self.state = 'q1'

elif self.state == 'q0' and symbol == '0':

self.state = 'q0'

elif self.state == 'q1' and symbol == '1':

self.state = 'q0'

elif self.state == 'q1' and symbol == '0':

self.state = 'q2'

elif self.state == 'q2' and symbol == '1':

self.state = 'q3'

elif self.state == 'q2' and symbol == '0':

self.state = 'q0'

elif self.state == 'q3' and symbol == '1':

self.state = 'q4'

elif self.state == 'q3' and symbol == '0':

self.state = 'q4'

elif self.state == 'q4' and symbol == '1':

self.state = 'q4'

elif self.state == 'q4' and symbol == '0':

self.state = 'q4'

else:

self.state = 'q0'

# Check the final state

return self.state == 'q3'

def main():

input\_string = input("Enter a string of 0s and 1s: ")

fsm = StateMachine()

result = fsm.process\_input(input\_string)

if result:

print("Accepted! The string always ends with '101'.")

else:

print("Rejected! The string does not end with '101'.")

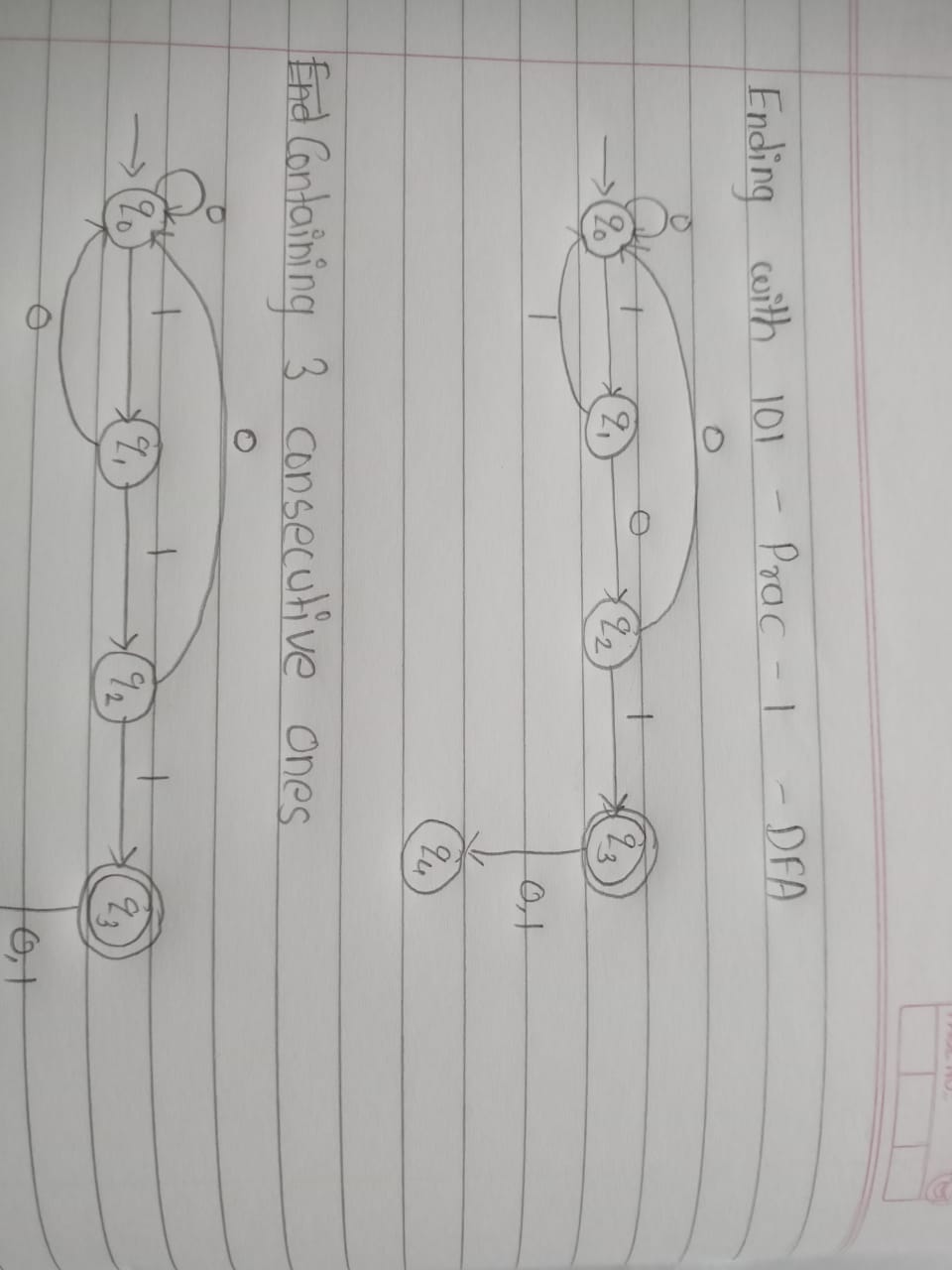
if \_\_name\_\_ == "\_\_main\_\_":

main()

**Output:**

Enter a string of 0s and 1s: 00101

Accepted! The string always ends with '101'.

**DFA:**

**Practical 2**

**Aim:**

**Design a Program for creating machine that accepts three consecutive one.**

**Code:**

class StateMachine:

def \_\_init\_\_(self):

self.state = 'q0'

self.consecutive\_ones = 0

def process\_input(self, input\_string):

for symbol in input\_string:

if self.state == 'q0' and symbol == '1':

self.consecutive\_ones = 1

self.state = 'q1'

elif self.state == 'q1' and symbol == '1':

self.consecutive\_ones += 1

if self.consecutive\_ones == 3:

self.state = 'q2'

else:

self.consecutive\_ones = 0

self.state = 'q0'

# Check the final state

return self.state == 'q2'

def main():

input\_string = input("Enter a string of 0s and 1s: ")

fsm = StateMachine()

result = fsm.process\_input(input\_string)

if result:

print("Accepted! The string contains three consecutive '1's.")

else:

print("Rejected! The string does not contain three consecutive '1's.")

if \_\_name\_\_ == "\_\_main\_\_":

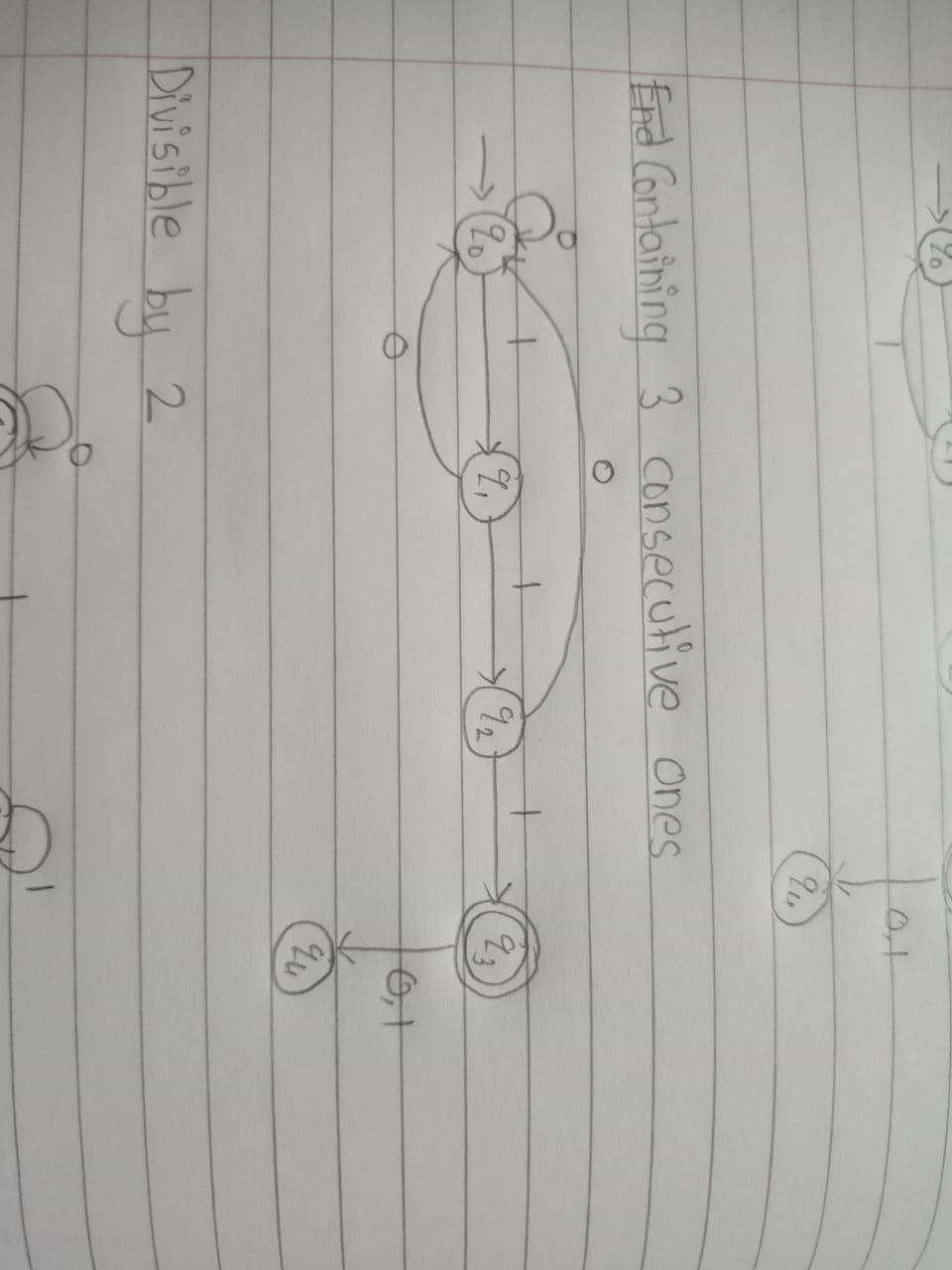
main()

**Output:**

Enter a string of 0s and 1s: 00111

Accepted! The string contains three consecutive '1's.

**DFA:**

****

**Practical – 3**

**Aim:**

**Design a program for accepting decimal number divisible by 2.**

**Code:**

class StateMachine:

def \_\_init\_\_(self):

self.state = 'q0'

def process\_input(self, input\_string):

for digit in input\_string:

if self.state == 'q0':

self.state = 'q1' if digit == '0' or digit == '2' or digit == '4' or digit == '6' or digit == '8' else 'q0'

elif self.state == 'q1':

self.state = 'q0' if digit.isdigit() else 'q0'

# Check the final state

return self.state == 'q1'

def main():

input\_string = input("Enter a decimal number: ")

fsm = StateMachine()

result = fsm.process\_input(input\_string)

if result:

print("Accepted! The decimal number is divisible by 2.")

else:

print("Rejected! The decimal number is not divisible by 2.")

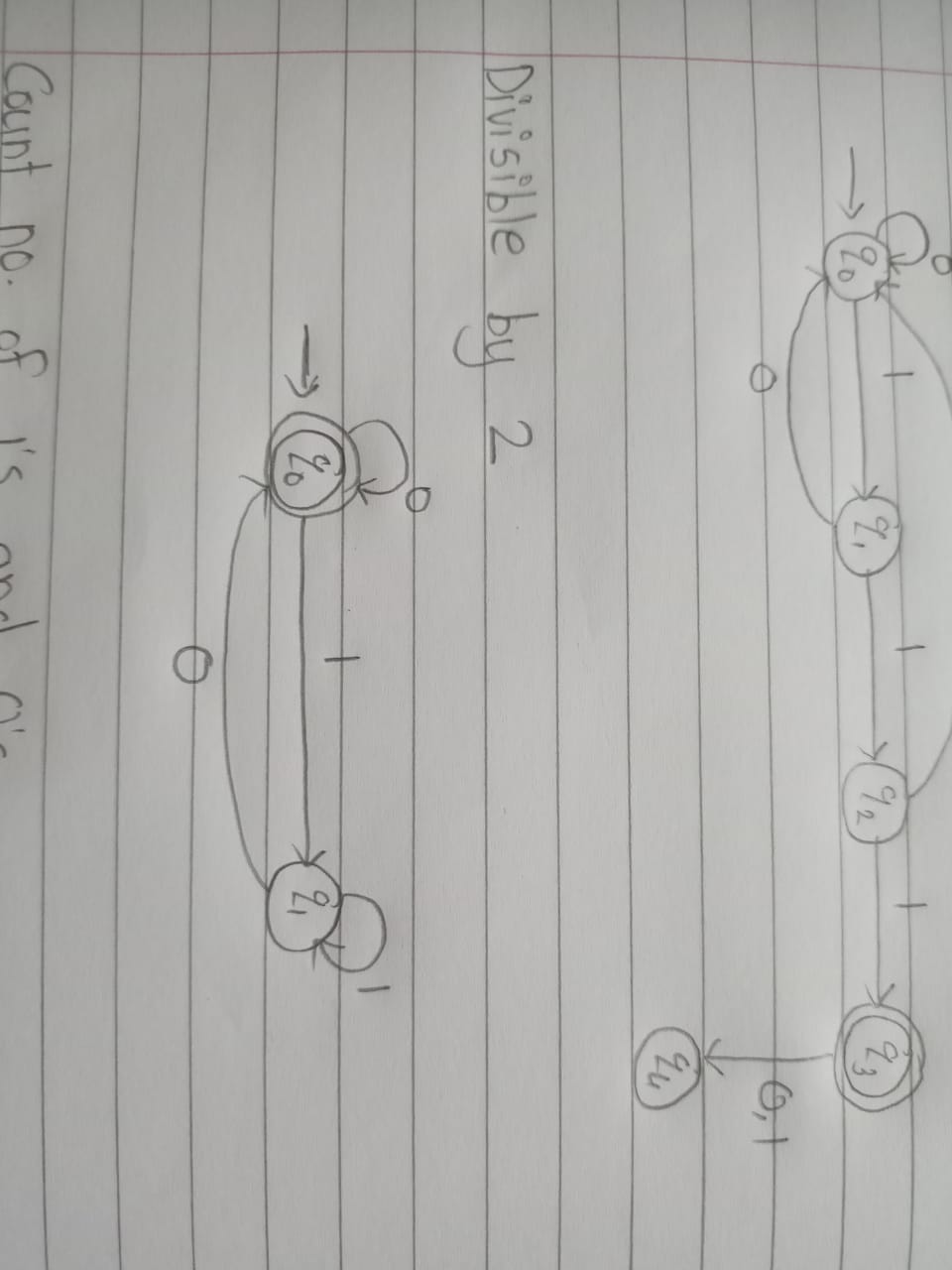
if \_\_name\_\_ == "\_\_main\_\_":

main()

**Output:**  
Enter a decimal number: 4.0

Accepted! The decimal number is divisible by 2.

**DFA:**

****

**Practical – 4**

**Aim:**

**Design a program for creating a machine which accepts string having equal no. of 1’s and 0’s.**

**Code:**

class StateMachine:

def \_\_init\_\_(self):

self.state = 'q0'

def process\_input(self, input\_string):

for symbol in input\_string:

if self.state == 'q0' and symbol == '0':

self.state = 'q1'

elif self.state == 'q0' and symbol == '1':

self.state = 'q2'

elif self.state == 'q1' and symbol == '0':

self.state = 'q0'

elif self.state == 'q1' and symbol == '1':

self.state = 'q3'

elif self.state == 'q2' and symbol == '0':

self.state = 'q3'

elif self.state == 'q2' and symbol == '1':

self.state = 'q0'

elif self.state == 'q3' and symbol == '0':

self.state = 'q2'

elif self.state == 'q3' and symbol == '1':

self.state = 'q1'

# Check the final state

return self.state == 'q0'

def main():

input\_string = input("Enter a string of 1s and 0s: ")

fsm = StateMachine()

result = fsm.process\_input(input\_string)

if result:

print("Accepted! The string has an equal number of 1s and 0s.")

else:

print("Rejected! The string does not have an equal number of 1s and 0s.")

if \_\_name\_\_ == "\_\_main\_\_":

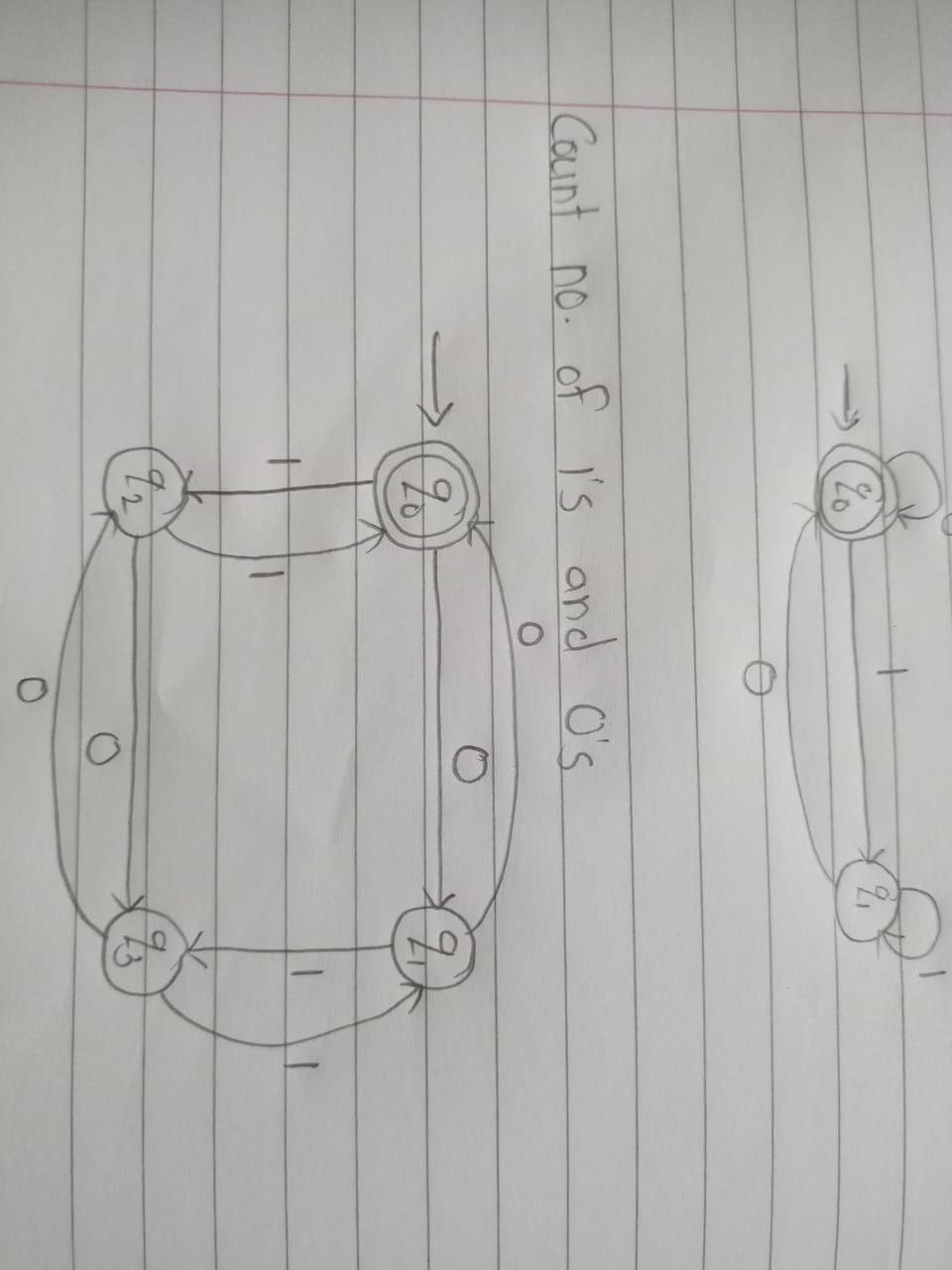
main()

**Output:**

Enter a string of 1s and 0s: 0011

Accepted! The string has an equal number of 1s and 0s.

**DFA:**

****

**Practical – 5**

**Aim:**

**Design a program for creating a machine which count number of 1’s and 0’s in a given string.**

**Code:**

class CountMachine:

def \_\_init\_\_(self):

self.state = "start"

self.count\_0 = 0

self.count\_1 = 0

def transition(self, char):

if self.state == "start":

if char == '0':

self.count\_0 += 1

elif char == '1':

self.count\_1 += 1

else:

raise ValueError("Invalid input character")

else:

raise ValueError("Invalid state")

def process\_input(self, input\_str):

for char in input\_str:

self.transition(char)

def get\_counts(self):

return self.count\_0, self.count\_1

def main():

input\_str = input("Enter a string containing 1's and 0's: ")

machine = CountMachine()

machine.process\_input(input\_str)

count\_0, count\_1 = machine.get\_counts()

print("Number of 0's:", count\_0)

print("Number of 1's:", count\_1)

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Output:**

Enter a string containing 1's and 0's: 1010

Number of 0's: 2

Number of 1's: 2

**Practical – 6**

**Aim:**

**Write a program for tokenization of given input.**

**Code:**  
def tokenize(input\_string):

tokens = []

current\_word = ""

state = "start"

for char in input\_string:

if state == "start":

if char != " ": # Start of a new word

current\_word += char

state = "in\_word"

elif state == "in\_word":

if char != " ": # Continue adding characters to the current word

current\_word += char

else: # End of the current word, append it to tokens

tokens.append(current\_word)

current\_word = ""

state = "start"

if current\_word: # Append the last word if it exists

tokens.append(current\_word)

return tokens

# Test the tokenizer

input\_string = "Tokenization of given input"

tokens = tokenize(input\_string)

print(tokens)

**Output:**  
['This', 'is', 'a', 'sample', 'input', 'string.']

**Practical – 7**

**Aim:**

**Write a program for generating regular expressions for regular grammar**

**Code:**  
class RegularGrammar:

def \_\_init\_\_(self):

self.productions = {}

def add\_production(self, non\_terminal, expression):

if non\_terminal in self.productions:

self.productions[non\_terminal].append(expression)

else:

self.productions[non\_terminal] = [expression]

def generate\_regex(self, non\_terminal):

if non\_terminal not in self.productions:

return None

regex = "|".join(self.productions[non\_terminal])

if len(self.productions[non\_terminal]) > 1:

regex = f"({regex})"

return regex

if \_\_name\_\_ == "\_\_main\_\_":

grammar = RegularGrammar()

# Example regular grammar rules

grammar.add\_production("S", "a")

grammar.add\_production("S", "bS")

grammar.add\_production("S", "")

# Generating regular expressions for each non-terminal

for non\_terminal in grammar.productions:

regex = grammar.generate\_regex(non\_terminal)

print(f"Regex for {non\_terminal}: {regex}")

**Output:**  
Regex for S: (a|bS|)

**Practical 8**

**Aim:**

**Write a program for generating derivation sequence / language for the given sequence of productions**

**Code:**

def generate\_derivation(grammar, start\_symbol, max\_depth):

# Define a recursive function to derive sequences

def derive(sentence, depth):

# If the maximum depth is reached, return the current sentence

if depth >= max\_depth:

return [sentence]

derivations = []

# Iterate over each symbol in the sentence

for i, symbol in enumerate(sentence):

# Check if the symbol has production rules defined

if symbol in grammar:

# Apply each production rule to the symbol

for production in grammar[symbol]:

# Create a new sentence by replacing the symbol with the production

new\_sentence = sentence[:i] + production + sentence[i+1:]

# Recursively derive sequences from the new sentence

derivations.extend(derive(new\_sentence, depth + 1))

return derivations

# Start the derivation process with the start symbol and depth 0

return derive(start\_symbol, 0)

# Test the function

grammar = {

'S': ['AB', 'BC'],

'A': ['a'],

'B': ['b'],

'C': ['c']

}

start\_symbol = 'S'

max\_depth = 3

derivation\_sequence = generate\_derivation(grammar, start\_symbol, max\_depth)

# Print the generated derivation sequence/language

print("Derivation sequence / Language:")

for derivation in derivation\_sequence:

print(derivation)

**Output:**

Derivation sequence / Language:

ab

ab

bc

bc

**Practical 9**

**Aim:**

**Design a Turing machine to recognize all strings consisting of an even number of 1's.**

**Code:**

states = {

'A': {

'0': ('A', '0', 'R'),

'1': ('B', '0', 'R'),

'\_': ('C', '\_', 'L')

},

'B': {

'0': ('B', '0', 'R'),

'1': ('A', '0', 'R'),

'\_': ('C', '\_', 'L')

},

'C': {}

}

initial\_state = "A"

final\_state = {"A"}

def turing\_machine(input\_str):

current\_state = initial\_state

tape = list(input\_str)

i\_head = 0

while True:

if tape[i\_head] not in states[current\_state]:

return False

new\_state, write\_value, move\_dir = states[current\_state][tape[i\_head]]

tape[i\_head] = write\_value

if move\_dir == 'R':

i\_head += 1

elif move\_dir == 'L':

i\_head -= 1

current\_state = new\_state

if current\_state in final\_state and i\_head >= len(tape):

return True

elif current\_state not in states or i\_head >= len(tape) or i\_head < 0:

return False

print(turing\_machine('011')) # True

print(turing\_machine('01111')) # True

print(turing\_machine('010101111')) # True

print(turing\_machine('01')) # False

print(turing\_machine('1101')) # False

print(turing\_machine('1101101')) # False

print(turing\_machine('110110111')) # False

**Output:**

True

True

True

False

False

False

False