**Week 13- Tutorial Assignment**

**Sample programs to Construct NFA and DFA using Python**

**1. Write a Python program to create an NFA that accepts strings containing only the letter 'a'.**

def is\_accepting\_state(current\_state):

return current\_state == 1

def nfa\_accepts(input\_string):

current\_state = 0

for char in input\_string:

if current\_state == 0 and char == 'a':

current\_state = 1

elif current\_state == 1 and char == 'a':

current\_state = 1

return is\_accepting\_state(current\_state)

# Get the input string from the user

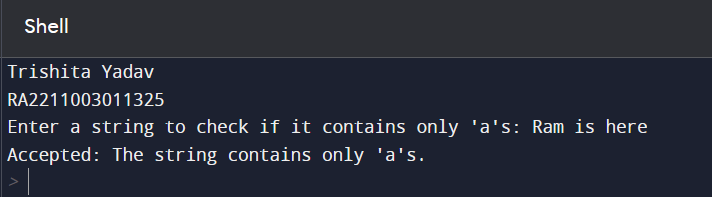
input\_string = input("Enter a string to check if it contains only 'a's: ")

if nfa\_accepts(input\_string):

print("Accepted: The string contains only 'a's.")

else:

print("Rejected: The string does not contain only 'a's.")



**2. Create a Python function to check if a given string is accepted by an NFA that recognizes the pattern "ab|ba" (either "ab" or "ba").**

def is\_accepting\_state(current\_state):

return current\_state in [2, 4]

def nfa\_accepts(input\_string):

current\_state = 0

for char in input\_string:

if current\_state == 0 and char == 'a':

current\_state = 1

elif current\_state == 0 and char == 'b':

current\_state = 3

elif current\_state == 1 and char == 'b':

current\_state = 2

elif current\_state == 2:

break # Accept if "ab"

elif current\_state == 3 and char == 'a':

current\_state = 4

elif current\_state == 4:

break # Accept if "ba"

else:

return False # Invalid transition

return is\_accepting\_state(current\_state)

# Get the input string from the user

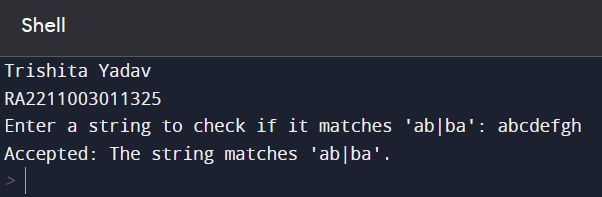
input\_string = input("Enter a string to check if it matches 'ab|ba': ")

if nfa\_accepts(input\_string):

print("Accepted: The string matches 'ab|ba'.")

else:

print("Rejected: The string does not match 'ab|ba'.")



**3. Implement a Python script that converts a simple NFA into a DFA with two states.**

def nfa\_transition(state, symbol):

if state == 0 and symbol == 'a':

return {0, 1}

elif state == 1 and symbol == 'a':

return {0}

else:

return set()

def nfa\_to\_dfa(nfa, alphabet):

dfa = {}

unprocessed\_states = [frozenset({0})] # Start with the initial state

while unprocessed\_states:

current\_nfa\_state = unprocessed\_states.pop()

dfa\_state = frozenset()

for symbol in alphabet:

nfa\_transitions = set()

for nfa\_state in current\_nfa\_state:

nfa\_transitions |= nfa\_transition(nfa\_state, symbol)

dfa\_state |= nfa\_transitions

if dfa\_state:

unprocessed\_states.append(dfa\_state)

dfa[current\_nfa\_state] = dfa\_state

return dfa

# Define the NFA alphabet

nfa\_alphabet = {'a'}

# Convert the NFA to a DFA

nfa = {

0: {'a': {0, 1}},

1: {'a': {0}}

}

dfa = nfa\_to\_dfa(nfa, nfa\_alphabet)

# Print the resulting DFA

print("DFA State Transitions:")

for state, transitions in dfa.items():

print(f"State {state} -> {transitions}")

**4. Write a Python program to construct a DFA that accepts binary strings ending in '01'.**

def is\_accepting\_state(current\_state):

return current\_state == 2

def dfa\_accepts(input\_string):

current\_state = 0

for char in input\_string:

if current\_state == 0 and char == '0':

current\_state = 1

elif current\_state == 1 and char == '1':

current\_state = 2

else:

current\_state = 0

return is\_accepting\_state(current\_state)

# Get the input binary string from the user

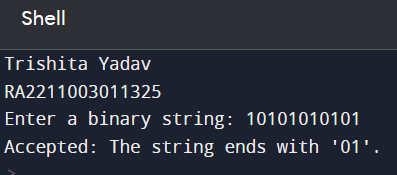
input\_string = input("Enter a binary string: ")

if dfa\_accepts(input\_string):

print("Accepted: The string ends with '01'.")

else:

print("Rejected: The string does not end with '01'.")



**5. Develop a Python function that takes an NFA and returns the set of states that can be reached from a given state on a specific input symbol.**

def nfa\_transition(nfa, current\_states, symbol):

next\_states = set()

for state in current\_states:

if state in nfa and symbol in nfa[state]:

next\_states |= nfa[state][symbol]

return next\_states

# Example NFA

nfa = {

0: {'a': {0, 1}},

1: {'b': {2}},

2: {'c': {3}},

3: {'d': {4}},

4: {'e': {5}},

}

# Function to get the set of states reached from a state on a specific input symbol

def get\_reachable\_states(nfa, current\_states, symbol):

next\_states = nfa\_transition(nfa, current\_states, symbol)

return next\_states

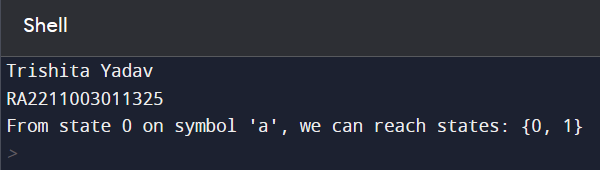
# Test the function with a specific state and symbol

current\_state = 0

input\_symbol = 'a'

reachable\_states = get\_reachable\_states(nfa, {current\_state}, input\_symbol)

print(f"From state {current\_state} on symbol '{input\_symbol}', we can reach states: {reachable\_states}")



**6. Create a Python script to minimize a simple DFA with three states by merging equivalent states.**

def minimize\_dfa(dfa, alphabet):

# Step 1: Initialize partition P0 into accepting and non-accepting states

accepting\_states = set(state for state, is\_accepting in dfa.items() if is\_accepting)

non\_accepting\_states = set(dfa.keys()) - accepting\_states

partitions = [accepting\_states, non\_accepting\_states]

# Step 2: Refine the partition until it no longer changes

while True:

new\_partitions = []

for partition in partitions:

for symbol in alphabet:

new\_partition = set()

for state in partition:

next\_state = dfa[state][symbol]

for sub\_partition in partitions:

if next\_state in sub\_partition:

new\_partition.add(state)

break

if new\_partition:

new\_partitions.append(new\_partition)

if new\_partitions == partitions:

break

partitions = new\_partitions

# Step 3: Create a new minimized DFA

minimized\_dfa = {}

for i, partition in enumerate(partitions):

state\_name = f'q{i}'

for state in partition:

minimized\_dfa[state\_name] = {}

for symbol in alphabet:

next\_state = dfa[state][symbol]

for j, sub\_partition in enumerate(partitions):

if next\_state in sub\_partition:

minimized\_dfa[state\_name][symbol] = f'q{j}'

break

return minimized\_dfa

# Define the DFA transitions and states (example with 3 states)

dfa = {

'q0': {'a': 'q1', 'b': 'q0'},

'q1': {'a': 'q1', 'b': 'q2'},

'q2': {'a': 'q2', 'b': 'q0'},

}

# Define the alphabet

alphabet = {'a', 'b'}

# Minimize the DFA

minimized\_dfa = minimize\_dfa(dfa, alphabet)

# Print the minimized DFA

print("Minimized DFA:")

for state, transitions in minimized\_dfa.items():

print(f"State {state} -> {transitions}")

**7. Implement a Python function that checks if a given string is accepted by a DFA that recognizes the pattern "ab\*c".**

def dfa\_accepts(input\_string):

current\_state = 0

for char in input\_string:

if current\_state == 0 and char == 'a':

current\_state = 1

elif current\_state == 1:

if char == 'b':

current\_state = 1

elif char == 'c':

current\_state = 2

else:

current\_state = 0

elif current\_state == 2:

return False # After reaching the accepting state, no more characters are allowed

return current\_state == 1

# Get the input string from the user

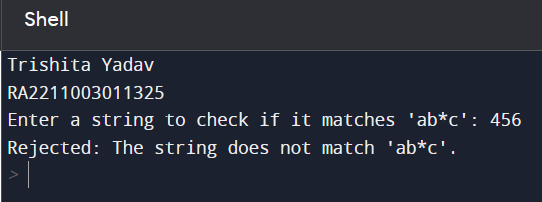
input\_string = input("Enter a string to check if it matches 'ab\*c': ")

if dfa\_accepts(input\_string):

print("Accepted: The string matches 'ab\*c'.")

else:

print("Rejected: The string does not match 'ab\*c'.")



**8. Write a Python program to create an NFA that accepts strings with an odd number of '1's.**

def is\_accepting\_state(current\_state):

return current\_state == 'accept'

def nfa\_accepts(input\_string):

current\_state = 'start'

for char in input\_string:

if current\_state == 'start' and char == '1':

current\_state = 'odd'

elif current\_state == 'odd' and char == '1':

current\_state = 'even'

elif current\_state == 'even' and char == '1':

current\_state = 'odd'

return is\_accepting\_state(current\_state)

# Get the input binary string from the user

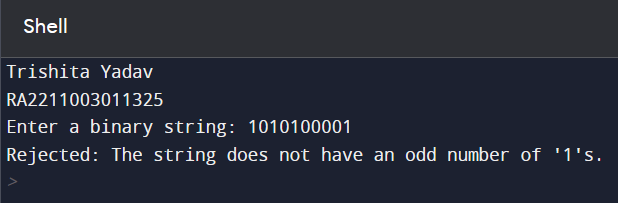
input\_string = input("Enter a binary string: ")

if nfa\_accepts(input\_string):

print("Accepted: The string has an odd number of '1's.")

else:

print("Rejected: The string does not have an odd number of '1's.")



**9. Develop a Python function that converts a simple regular expression like "a(b|c)\*" into an equivalent NFA.**

def create\_nfa\_from\_regex(regex):

nfa = {}

states = 0

stack = []

def new\_state():

nonlocal states

state = states

states += 1

return state

def epsilon\_transition(from\_state, to\_state):

if from\_state not in nfa:

nfa[from\_state] = {}

if 'ε' not in nfa[from\_state]:

nfa[from\_state]['ε'] = set()

nfa[from\_state]['ε'].add(to\_state)

for char in regex:

if char == 'a':

start\_state = new\_state()

end\_state = new\_state()

epsilon\_transition(start\_state, end\_state)

stack.append((start\_state, end\_state))

elif char == 'b':

start\_state = new\_state()

end\_state = new\_state()

epsilon\_transition(start\_state, end\_state)

stack.append((start\_state, end\_state))

elif char == 'c':

start\_state = new\_state()

end\_state = new\_state()

epsilon\_transition(start\_state, end\_state)

stack.append((start\_state, end\_state))

elif char == '|':

end1, start1 = stack.pop()

end2, start2 = stack.pop()

start\_state = new\_state()

end\_state = new\_state()

epsilon\_transition(start\_state, start1)

epsilon\_transition(start\_state, start2)

epsilon\_transition(end1, end\_state)

epsilon\_transition(end2, end\_state)

stack.append((start\_state, end\_state))

elif char == '\*':

start, end = stack.pop()

start\_state = new\_state()

end\_state = new\_state()

epsilon\_transition(start\_state, start)

epsilon\_transition(start\_state, end\_state)

epsilon\_transition(end, start)

epsilon\_transition(end, end\_state)

stack.append((start\_state, end\_state))

if len(stack) != 1:

raise ValueError("Invalid regular expression")

return nfa

# Example regular expression

regex = "a(b|c)\*"

# Convert the regular expression to an NFA

nfa = create\_nfa\_from\_regex(regex)

# Print the resulting NFA

for state, transitions in nfa.items():

print(f"State {state} -> {transitions}")