THEORY OF COMPUTATION PRACTICALS

**Name - Vidya Vipul**

**College Roll No. - 4026**

**Q1. Design a Finite Automata (FA) that accepts all strings over S={0, 1} having three consecutive 1's as**

**a substring. Write a program to simulate this FA.**

**CODE:**

#include<iostream>

using namespace std;

void State1(string w,int i);

void State2(string w,int i);

void State3(string w,int i);

void State4(string w,int i);

int main(){

string w;

cout << "enter string: ";

cin >> w;

State1(w, 0);

return 0;

}

void State1(string w, int i){

cout << "state 1" << endl;

if (i == w.length()){

cout << "string is rejected";

return;

}

else{

if (w[i] == '1')

State2(w, i + 1);

if (w[i] == '0')

State1(w, i + 1);

}

}

void State2(string w, int i){

cout << "state 2" << endl;

if (i == w.length()){

cout << "string is rejected";

return;

}

else{

if (w[i] == '1')

State3(w, i + 1);

if (w[i] == '0')

State2(w, i + 1);

}

}

void State3(string w, int i){

cout << "state 3" << endl;

if (i == w.length()){

cout << "string is accepted";

return;

}

else{

if (w[i] == '1')

State4(w, i + 1);

if (w[i] == '0')

State3(w, i + 1);

}

}

void State4(string w, int i){

cout << "state 4" << endl;

if (i == w.length()){

cout << "string is accepted";

return;

}

else{

if (w[i] == '1')

{cout<<"string is rejected";return;}

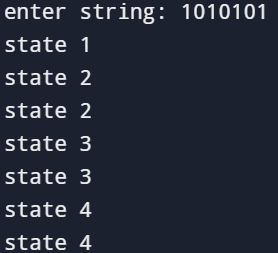
if (w[i] == '0')

State4(w, i + 1);

}

}

Output:



**Q2. Design a Finite Automata (FA) that accepts all strings over S={0, 1} having either exactly two 1's or**

**exactly three 1's, not more nor less. Write a program to simulate this FA.**

**CODE:**

#include<iostream>

using namespace std;

void state1(string w, int i); void state2(string w, int i); void state3(string w, int i); void state4(string w, int i);

void state1(string w,int i){ cout<<"state 1"<<endl;

if(i==w.length()){

cout<<"String is Rejected"; return;

}

else{

if(w[i]=='1'){

state2(w,i+1);

}

if (w[i]=='0'){

state1(w,i+1);

}

}

}

void state2(string w,int i) {

cout<<"state 2"<<endl;

if(i==w.length()){

cout<<"String is rejected"; return;

}

else{

if(w[i]=='1'){

state3(w,i+1);

}

if(w[i]=='0'){

state1(w,i+1);

}

}

}

void state3(string w,int i) {

cout<<"state 3"<<endl;

if(i==w.length()){

cout<<"String is rejected"; return;

}

else{

if(w[i]=='1'){

state4(w,i+1);

}

if(w[i]=='0'){

state1(w,i+1);

}

}

}

void state4(string w,int i) {

cout<<"state 4"<<endl;

if(i==w.length()){

cout<<"String is accepted"; return;

}

else{

if (w[i]=='1'){

state4(w,i+1);

}

if (w[i]=='0'){

state4(w,i+1);

}

}

}

int main(){

string w;

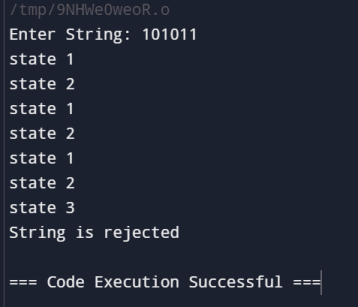
cout<<"Enter String: ";

cin>>w;

state1(w,0);

}

Output:’



**Q3. Design a Finite Automata (FA) that accepts all strings over S={0, 1} having either exactly two 1's or**

**exactly three 1's, not more nor less. Write a program to simulate this FA.**

**CODE:**9/4/24, 10:40 AM

state = 'qo'

first\_char =

second\_char =

for **i,** char in enumerate(input\_string):

**if** state == 'qo':

state = 'q1**'**

first\_char = char

elif state ==

'q1':

state **=** 'q2'

second\_char = char

elif state

==

'q2'**:**

'q3'

state =

elif state == 'q3':

if char == first\_char:

state =

'q4'

else:

return False

elif state

==

'q4':

**if** char == second\_char:

state =

'q5'

else:

return False

elif state == 'q5':

if char == second\_char:

state

else:

=

'q6'

Untitled0.ipynb - Colab

return False

return state == 'q6'

# Test the FA with some strings

test\_strings

==

**[**"aabb", "abab", "aaaa", "bbaa", "abba"]

results = {s: simulate\_fa(s) for s in test\_strings}

print(results)

{'aabb': False, 'abab': False, 'aaaa': False, 'bbaa': False, 'abba': False}

Start coding or generate with AI.

https://colab.research.google.com/drive/1lfSp\_GjYBiDujUEFTUsl\_ruBA392JOI-?authuser=0#scrollTo=fFI0N50eaXAL&printMode=true

1/1

111

**Q4. Design a Finite Automata (FA) that accepts language L2, over S= {a, b} where L2= a(a+b)\*b. Write**

**a program to simulate this FA.**

def simulate\_fa(input\_string):

current\_state = 'q0'

for char in input\_string:

if current\_state == 'q0' and char == 'a':

current\_state = 'q1'

elif current\_state == 'q1' and char == 'a':

current\_state = 'q1'

elif current\_state == 'q1' and char == 'b':

current\_state = 'q1'

elif current\_state == 'q1' and char == 'b':

current\_state = 'q2'

else:

return False

if current\_state == 'q2':

return True

else:

return False

test\_strings = ["aab", "abb", "aaabbb", "ab", "ba", "aaa", "bbb"] for string in test\_strings:

if simulate\_fa(string):

print(f"'{string}' is accepted.")

else:

print(f"'{string}' is rejected.")

'aab' is rejected.

'abb' is rejected.

'aaabbb' is rejected.

'ab' is rejected.

'ba' is rejected.

'aaa' is rejected.

'bbb' is rejected.

**Q5.Design a Finite Automata (FA) that accepts language EVEN-EVEN over S={a, b}. Write a program**

**to simulate this FA**

class DFA:

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

# Define states

self.states = {'q0', 'q1', 'q2', 'q3'}

# Define start state

self.start\_state = 'q0'

# Define accept state

self.accept\_states = {'q0'}

# Define transitions

self.transition = {

'q0': {'a': 'q1', 'b': 'q2'},

'q1': {'a': 'q0', 'b': 'q3'},

'q2': {'a': 'q3', 'b': 'q0'},

'q3': {'a': 'q2', 'b': 'q1'}

}

def accepts(self, input\_string):

current\_state = self.start\_state

for symbol in input\_string:

if symbol not in {'a', 'b'}:

return False # Reject if symbol is not in alphabet

current\_state = self.transition[current\_state][symbol]

return current\_state in self.accept\_states

# Create an instance of the DFA

dfa\_even\_even = DFA()

# Test strings

test\_strings = ['a', 'b', 'ab', 'aabb', 'abab', 'bbaa', 'aaa', 'bbb', 'ababab', '']

print("Testing EVEN-EVEN DFA:")

for string in test\_strings:

result = dfa\_even\_even.accepts(string)

print(f"{string}: {'Accepted' if result else 'Rejected'}")

Testing EVEN-EVEN DFA:

a: Rejected

b: Rejected

ab: Rejected

aabb: Accepted

abab: Accepted

bbaa: Accepted

aaa: Rejected

bbb: Rejected

ababab: Rejected

: Accepted

Start coding or generate with AI.

**Q6. Write a program to simulate an FA that accepts**

**a. Union of the languages L1 and L2**

**b. Intersection of the languages L1 and L2**

**c. Language L1 L2 (concatenation)**

def \_\_init\_\_(self, states, alphabet, transition, start\_state, accept\_states): self.states = states

self.alphabet = alphabet

self.transition = transition

self.start\_state = start\_state

self.accept\_states = accept\_states

def accepts(self, input\_string):

current\_state = self.start\_state

for symbol in input\_string:

if symbol not in self.alphabet:

return False

current\_state = self.transition[current\_state][symbol]

return current\_state in self.accept\_states

def \_\_repr\_\_(self):

return f"DFA(states={self.states}, alphabet={self.alphabet}, start\_state={self.start

def union(dfa1, dfa2):

new\_states = {(q1, q2) for q1 in dfa1.states for q2 in dfa2.states}

new\_start\_state = (dfa1.start\_state, dfa2.start\_state)

new\_accept\_states = {(q1, q2) for q1 in dfa1.accept\_states for q2 in dfa2.states} | \ {(q1, q2) for q1 in dfa1.states for q2 in dfa2.accept\_states}

new\_transition = {}

for (q1, q2) in new\_states:

new\_transition[(q1, q2)] = {}

for symbol in dfa1.alphabet | dfa2.alphabet:

next\_q1 = dfa1.transition.get(q1, {}).get(symbol, None)

next\_q2 = dfa2.transition.get(q2, {}).get(symbol, None)

new\_transition[(q1, q2)][symbol] = (next\_q1, next\_q2)

return DFA(new\_states, dfa1.alphabet | dfa2.alphabet, new\_transition, new\_start\_state, n

def intersection(dfa1, dfa2):

new\_states = {(q1, q2) for q1 in dfa1.states for q2 in dfa2.states}

new\_start\_state = (dfa1.start\_state, dfa2.start\_state)

new\_accept\_states = {(q1, q2) for q1 in dfa1.accept\_states for q2 in dfa2.accept\_states

new\_transition = {}

for (q1, q2) in new\_states:

new\_transition[(q1, q2)] = {}

for symbol in dfa1.alphabet | dfa2.alphabet:

next\_q1 = dfa1.transition.get(q1, {}).get(symbol, None)

next\_q2 = dfa2.transition.get(q2, {}).get(symbol, None)

if next\_q1 is not None and next\_q2 is not None:

new\_transition[(q1, q2)][symbol] = (next\_q1, next\_q2)

else:

new\_transition[(q1, q2)][symbol] = None

return DFA(new\_states, dfa1.alphabet | dfa2.alphabet, new\_transition, new\_start\_state, n

def concatenate(dfa1, dfa2):

new\_states = dfa1.states | dfa2.states

t t t t df 1 t t t t

new\_start\_state = dfa1.start\_state

new\_accept\_states = dfa2.accept\_states

new\_transition = {}

new\_transition.update(dfa1.transition)

new\_transition.update(dfa2.transition)

for state in dfa1.accept\_states:

new\_transition[state] = {symbol: dfa2.start\_state for symbol in dfa2.alphabet} return DFA(new\_states, dfa1.alphabet | dfa2.alphabet, new\_transition, new\_start\_state, n

# Example DFA for L1 and L2

dfa1 = DFA(

states={'q0', 'q1'},

alphabet={'a', 'b'},

transition={

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

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

},

start\_state='q0',

accept\_states={'q1'}

)

dfa2 = DFA(

states={'p0', 'p1'},

alphabet={'a', 'b'},

transition={

'p0': {'a': 'p0', 'b': 'p1'},

'p1': {'a': 'p1', 'b': 'p1'}

},

start\_state='p0',

accept\_states={'p1'}

)

# Create union, intersection, and concatenation DFAs

union\_dfa = union(dfa1, dfa2)

intersection\_dfa = intersection(dfa1, dfa2)

concat\_dfa = concatenate(dfa1, dfa2)

# Test input strings

test\_strings = ['a', 'b', 'aa', 'ab', 'ba', 'bb', 'aab', 'bba']

print("Union DFA Results:")

for string in test\_strings:

print(f"{string}: {union\_dfa.accepts(string)}")

print("\nIntersection DFA Results:")

for string in test\_strings:

print(f"{string}: {intersection\_dfa.accepts(string)}")

print("\nConcatenation DFA Results:")

for string in test\_strings:

print(f"{string}: {concat\_dfa.accepts(string)}")

Union DFA Results:

a: True

b: True

aa: True

ab: True

ba: True

bb: True

aab: True

bba: True

Intersection DFA Results: a: False

b: False

aa: False

ab: True

ba: True

bb: False

aab: True

bba: True

Concatenation DFA Results: a: False

b: False

aa: False

ab: False

ba: False

bb: False

aab: True

bba: False

**Q7. Design a PDA and write a program for simulating the machine which accepts the language**

**{anbn where n>0, S= {a, b}}.**

**CODE:**

**def simulate\_pda(input\_string):**

**"""**

**Simulates a PDA for the language {anbn | n > 0, S = {a, b}}.**

**Args:**

**input\_string: The string to be processed by the PDA.**

**Returns:**

**True if the input string is accepted by the PDA, False otherwise.**

**"""**

**stack = ['Z0'] # Initial stack symbol**

**state = 'q0' # Initial state**

**for char in input\_string:**

**if state == 'q0' and char == 'a':**

**stack.append('a')**

**state = 'q0'**

**elif state == 'q0' and char == 'b':**

**return False # b before a is not allowed**

**elif state == 'q0' and char == 'a':**

**stack.append('a')**

**elif state == 'q0' and char == 'b' and len(stack) > 1:**

**if stack[-1] == 'a':**

**stack.pop()**

**state = 'q1'**

**else:**

**return False**

**elif state == 'q1' and char == 'b' and len(stack) > 1 :**

**if stack[-1] == 'a':**

**stack.pop()**

**state = 'q1'**

**else:**

**return False**

**else:**

**return False**

**if state == 'q1' and len(stack) == 1 and stack[0] == 'Z0':**

**return True # Accepted if all a's are matched with b's**

**else:**

**return False # Not accepted**

**# Example usage**

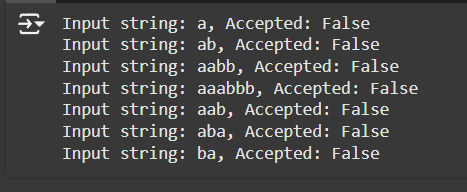
**strings = ["a", "ab", "aabb", "aaabbb", "aab", "aba", "ba"]**

**for string in strings:**

**result = simulate\_pda(string)**

**print(f"Input string: {string}, Accepted: {result}")**

**OUTPUT -**

****

**Q8.Design a PDA and write a program for simulating the machine which accepts the language {wXwr**

**| w**

**is any string over S={a, b} and wr**

**is reverse of that string and X is a special symbol }.**

**CODE:**

**def simulate\_pda(input\_string):**

**"""**

**Simulates a PDA for the language {wXwr | w is any string over S={a, b} and wr is reverse of that string and X is a special symbol }.**

**Args:**

**input\_string: The string to be processed by the PDA.**

**Returns:**

**True if the input string is accepted by the PDA, False otherwise.**

**"""**

**stack = ['Z0'] # Initial stack symbol**

**state = 'q0' # Initial state**

**input\_string\_list = list(input\_string)**

**#Check if X exists**

**try:**

**x\_index = input\_string\_list.index('X')**

**except ValueError:**

**return False**

**w = input\_string\_list[:x\_index]**

**wr = input\_string\_list[x\_index+1:]**

**wr.reverse()**

**if len(w) != len(wr):**

**return False**

**for i in range(len(w)):**

**if state == 'q0':**

**if w[i] == 'a' or w[i] == 'b':**

**stack.append(w[i])**

**state = 'q0'**

**else:**

**return False**

**if input\_string\_list[x\_index] != 'X':**

**return False**

**for i in range(len(wr)):**

**if state == 'q0':**

**if stack[-1] == wr[i]:**

**stack.pop()**

**state = 'q0'**

**else:**

**return False**

**if len(stack) == 1 and stack[0] == 'Z0':**

**return True**

**else:**

**return False**

**# Example usage**

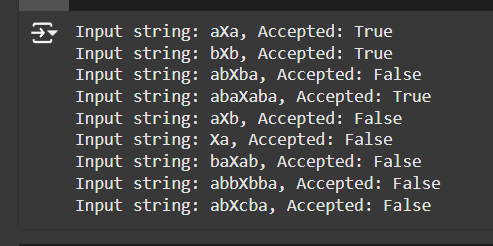
**strings = ["aXa", "bXb", "abXba", "abaXaba", "aXb", "Xa", "baXab", "abbXbba", "abXcba"]**

**for string in strings:**

**result = simulate\_pda(string)**

**print(f"Input string: {string}, Accepted: {result}")**

**OUTPUT -**

****