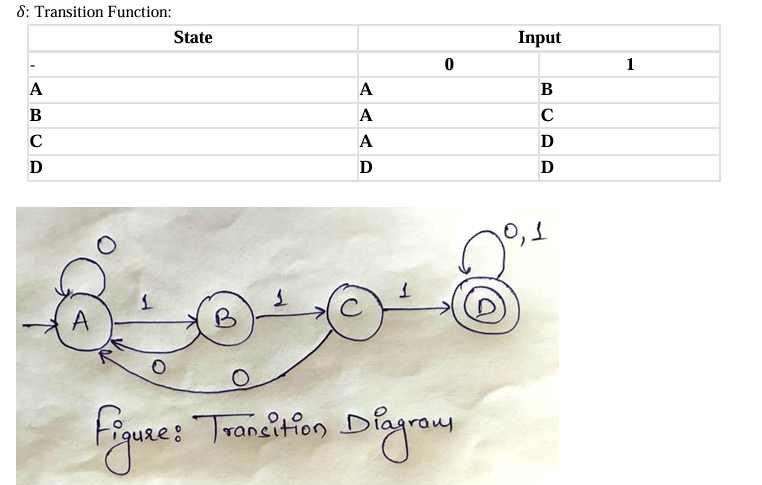
1. Design a Finite State Machine (FSM) that accepts all strings over input symbols {0, 1} having three consecutive 1's as a substring. A = {111, 0111, 1110, 0101011110101,...} means any string should be declared valid if it contains 111 as a substring.  
   Let M be the machine, M(Q, Σ, 𝛿, q0, F) where

Q: set of states: {A, B, C, D}   
Σ: set of input symbols: {0, 1}   
q0: initial state (A) State   
F: set of Final states: {D}   


| from enum import Enum  class State(Enum):  S0 = 0  S1 = 1  S2 = 2  S3 = 3  def accepts\_string(input\_str):  current\_state = State.S0    for c in input\_str:  if current\_state == State.S0:  if c == '1':  current\_state = State.S1  elif current\_state == State.S1:  if c == '1':  current\_state = State.S2  else:  current\_state = State.S0  elif current\_state == State.S2:  if c == '1':  current\_state = State.S3  else:  current\_state = State.S0  elif current\_state == State.S3:  pass # Stay in S3    return current\_state == State.S3  # Directly take input and check input\_str = input("Enter a binary string: ") if accepts\_string(input\_str):  print("Accepted (contains 111)") else:  print("Rejected (does not contain 111)") |
| --- |

OUTPUT:

1. Enter a binary string: 100101101101 Rejected (does not contain 111)

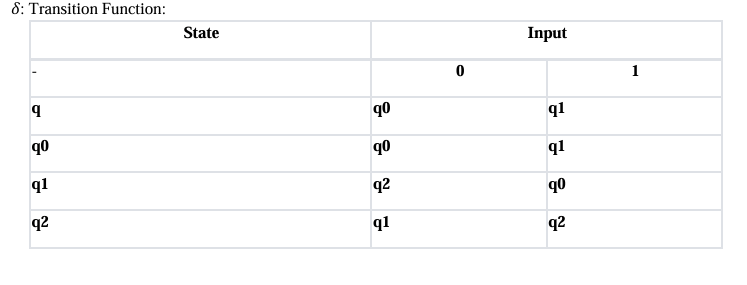
2. Enter a binary string: 1010111 Accepted (contains 111)

Complexity

* Time Complexity: O(n), where n is the length of the input string (since we process each character exactly once).
* Space Complexity: O(1), since we only store a fixed number of states and variables regardless of input size.

1. Design a Finite State Machine (FSM) that accepts all strings over input symbols {0, 1} which are divisible by 3

A = {0, 00, 000, 11, 011, 110, ...}means any binary string that when divided by three gives the remainder zero.   
Let M be the machine for above, hence it can be defined as M(Q, Σ, 𝛿, q0, F) where Q: set of states: {q, q0, q1, q2}   
Σ: set of input symbols: {0, 1}   
q0: initial state (q)   
F: set of Final states: {q0}

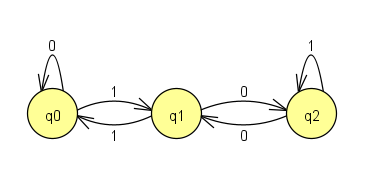


011 - 3

110 - 6

1001 - 9

Transition diagram:



| from enum import Enum  class State(Enum):  S0 = 0 # Remainder 0 (divisible by 3)  S1 = 1 # Remainder 1  S2 = 2 # Remainder 2  def is\_divisible\_by\_3(input\_str):  current\_state = State.S0    for c in input\_str:  if current\_state == State.S0:  if c == '0':  current\_state = State.S0  elif c == '1':  current\_state = State.S1  elif current\_state == State.S1:  if c == '0':  current\_state = State.S2  elif c == '1':  current\_state = State.S0  elif current\_state == State.S2:  if c == '0':  current\_state = State.S1  elif c == '1':  current\_state = State.S2    return current\_state == State.S0  # Test the function input\_str = input("Enter a binary string: ") if is\_divisible\_by\_3(input\_str):  print("Accepted (the number is divisible by 3)") else:  print("Rejected (the number is not divisible by 3)") |
| --- |

OUTPUT:

Enter a binary string:

1010 Rejected (the number is not divisible by 3)

Enter a binary string:

11 Accepted (the number is divisible by 3)

Complexity

Time Complexity: O(n), where n is the length of the input string. Each character is processed exactly once.

Space Complexity: O(1), since we only need a fixed amount of space for the current state.

1. Design a Finite State Machine (FSM) that accepts all decimal string which are divisible by 3

A = {0, 3, 6, 9, 03, 06, 09, 12, 012, ..} means any decimal number string that when divided by three gives the remainder zero.

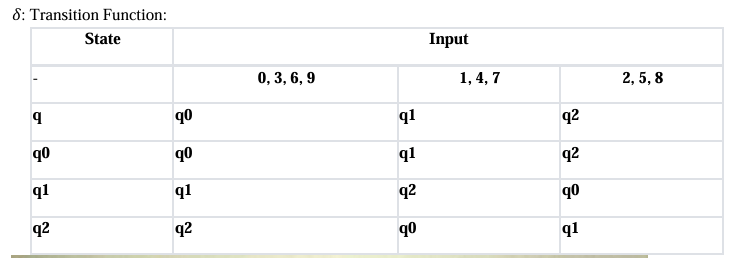
Let M be the machine, hence it can be defined as M(Q, Σ, 𝛿, q0, F) where

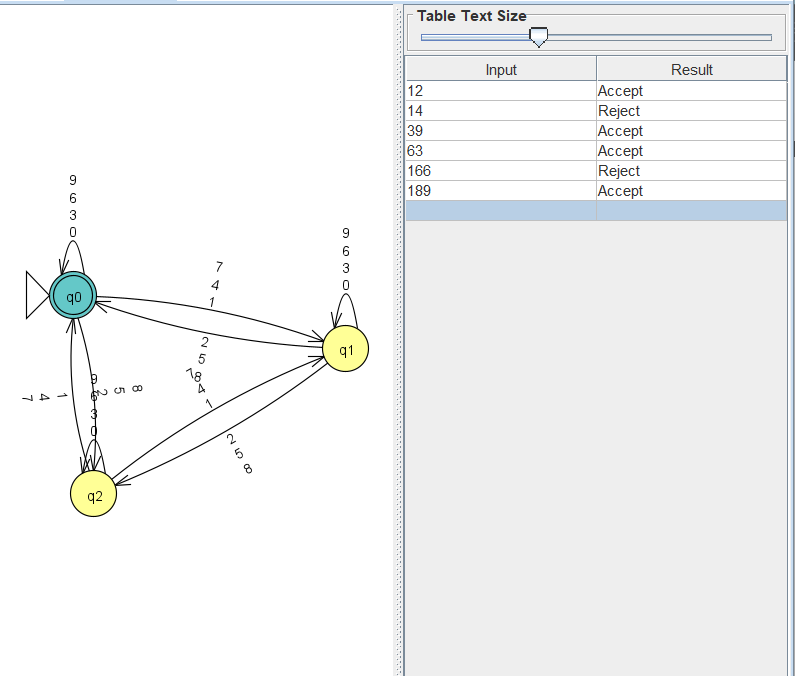
Q: set of states: {q, q0, q1, q2}

Σ: set of input symbols: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

q0: initial state (q)

F: set of Final states: {q0}





CODE:

| from enum import Enum  class State(Enum):  S0 = 0  S1 = 1  S2 = 2   def is\_divisible\_by\_3(input\_str):  current\_state = State.S0    for c in input\_str:  if not c.isdigit():  print("Invalid input: Non-decimal character encountered!")  return False    digit = int(c)     if current\_state == State.S0:  current\_state = State((0\*10+digit)%3)  elif current\_state == State.S1:  current\_state = State((1\*10+digit)%3)  elif current\_state == State.S2:  current\_state = State((2\*10+digit)%3)    return current\_state == State.S0   input\_str = input("Enter your input string: ") if is\_divisible\_by\_3(input\_str):  print("Accepted (is divisible by 3)") else:  print("Rejected (is not divisible by 3)") |
| --- |

OUTPUT:

Enter a decimal string: 25 Rejected (the number is not divisible by 3)

Enter a decimal string: 27 Accepted (the number is divisible by 3)

Time Complexity :

Time Complexity: O(n), where n is the length of the input string. Each character is processed exactly once.

Space Complexity: O(1), since we only need a fixed amount of space for the current state.

1. Design a PushDown Automata (PDA) that accepts all strings having equal number of 0's and 1's over input symbol {0, 1} for a language 0n1n where n >= 1.

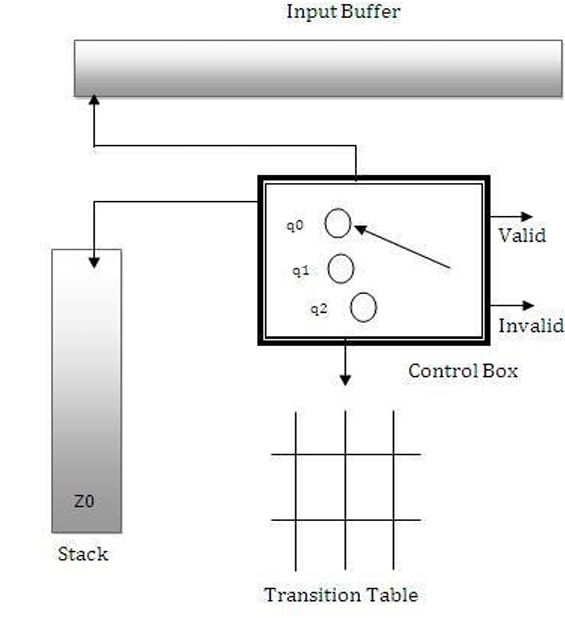
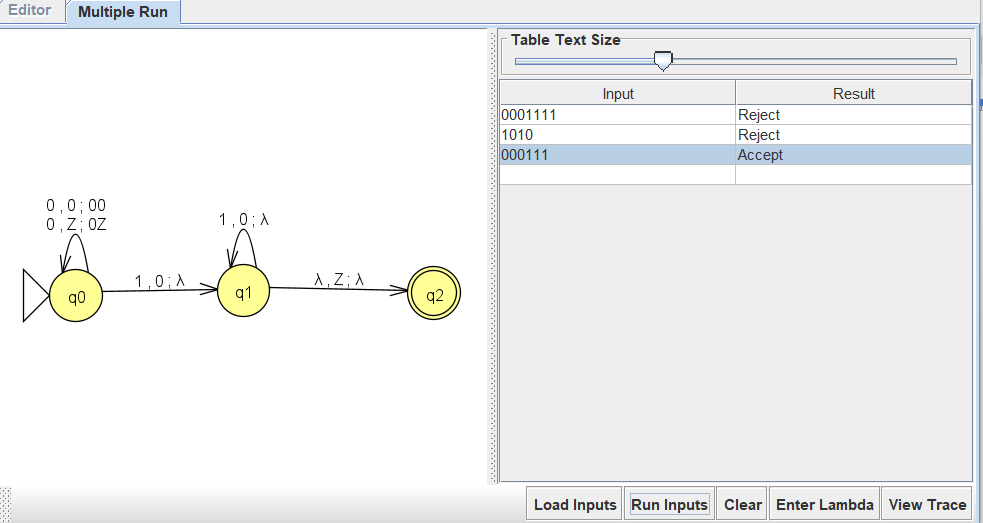
A = {01, 0011, 000111, ...} means all strings having n number of 0's followed by n numbers of 1's where n can be any number greater than equal tp and count of 0's must be equal to count of 1's.   
  
Block diagram of push down automata is shown in Figure 1. 

Figure 1: Block Diagram of PushDown Automata.   
  
Input string can be valid or invalid, valid if it follows the language 0^n 1^n where n >= 1 else invalid. PDA has to determine whether the input string is according to the language or not.   
Let M be the PDA machine for above AIM, hence it can be defined as M(Q, Σ, Г, δ, q0, Z0, F) where   
Q: set of states: {q0, q1, q2}  
Σ: set of input symbols: {0, 1}   
Г: Set of stack symbols: {A, Z0}   
q0: initial state (q0)   
Z0: initial stack symbol (Z0)   
F: set of Final states: { } [Note: Here, the set of final states is null as the decision of validity of string is based on whether it is empty or not.]  
δ: Transition Function: (Transition state diagram is shown in Figure 2.)

δ(q0, 0, Z0) → (q0, AZ0)   
δ(q0, 0, 0) → (q0, 00)   
δ(q0, 1, 0) → (q1, ε)   
δ(q1, 1, 0) → (q1, ε)   
δ(q1, ε, Z0) → (q2, ε)



CODE:

| def is\_valid\_string(input\_str):  stack = []  i = 0  n = len(input\_str)    # Push all leading '0's onto the stack  while i < n and input\_str[i] == '0':  stack.append('0')  i += 1    # Pop '0's for each '1'  while i < n and input\_str[i] == '1':  if not stack: # Stack is empty (no matching '0')  return False  stack.pop()  i += 1    # String is valid if we processed all characters and stack is empty  return i == n and not stack  # Example usage: input\_str = input("Enter a string of 0's and 1's: ") if is\_valid\_string(input\_str):  print("Accepted: The string has equal number of 0's and 1's in the form 0^n1^n.") else:  print("Rejected: The string does not have equal number of 0's and 1's in the form 0^n1^n.") |
| --- |

OUTPUT:

Enter a string of 0's and 1's:

0011 Accepted: The string has an equal number of 0's and 1's in the form 0^n1^n.

Enter a string of 0's and 1's: 0011101

Rejected: The string does not have an equal number of 0's and 1's in the form 0^n1^n.

Time Complexity:

Time Complexity: O(n), where n is the length of the input string. We process each character exactly once.   
Space Complexity: O(n), since in the worst case, we may need to store up to n 0s in the stack.

1. Design a Program to create a PDA machine that accepts the well-formed parenthesis.

A = {(),(()), ()(), (()), ((()(()))) ...} means all the parenthesis that are open must be closed or a combination of all legal parenthesis. Here, the opening parenthesis is '(' and closing parenthesis is ')'.  
Let M be the PDA machine for above AIM, hence it can be defined as M(Q, Σ, Г, δ, q0, Z0, F) where

Q: set of states: {q0, q1}

Σ: set of input symbols: {(, )}

Г: Set of stack symbols: {(, Z}

q0: initial state (q0)

Z0: initial stack symbol (Z)

F: set of Final states: { } [Note: Here, set of final states is null as decision of validity of string is based on stack whether it is empty or not. If empty means valid else invalid.]

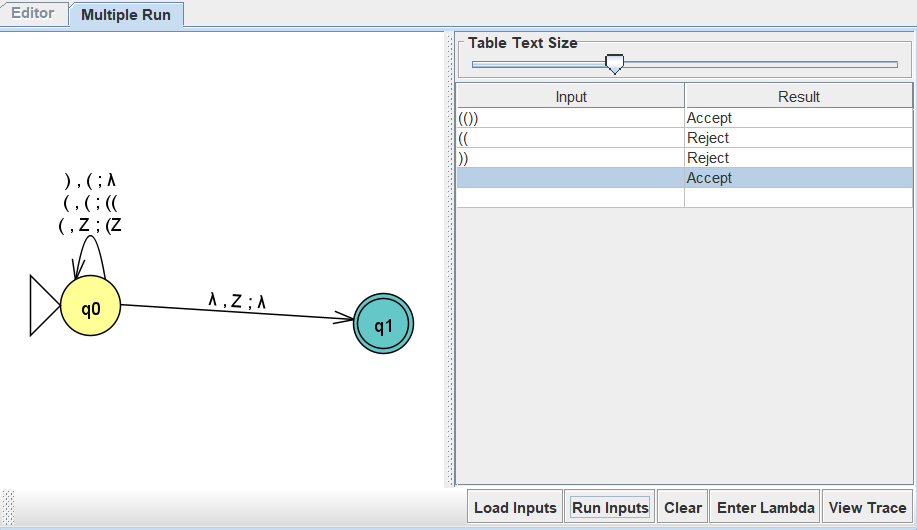
δ: Transition Function: (Transition state diagram is shown in Figure 2.)

δ(q0, (, Z) → (q0, (Z)

δ(q0, (, () → (q0, (()

δ(q0, ), () → (q0, ε)

δ(q0, ε, Z) → (q1, ε)



CODE:

| def is\_well\_formed\_parentheses(input\_str):  stack = []  for c in input\_str:  if c == '(':  stack.append(c)  elif c == ')':  if not stack: # If stack is empty  return False  stack.pop()  else:  return False # Invalid character  return not stack # True if stack is empty  # Example usage: input\_str = input("Enter a string of parentheses: ") if is\_well\_formed\_parentheses(input\_str):  print("Accepted: The parentheses are well-formed.") else:  print("Rejected: The parentheses are not well-formed.") |
| --- |

Output:

Enter a string of parentheses: (()()())

Accepted: The parentheses are well-formed.

Enter a string of parentheses: )()((

Rejected: The parentheses are not well-formed.

Time Complexity:

Time Complexity: O(n), where n is the length of the input string. We process each character exactly once.   
Space Complexity: O(n), since in the worst case, we may need to store up to n 0s in the stack.

1. Design a PDA to accept WCWR where w is any binary string and WR is the reverse of that string and C is a special symbol.

A = {0C0, 1C1, 011000110C011000110, 101011C110101, ..... } means string must have some binary string followed by special character 'C' followed reverse of binary string that appears before 'C'.

Let M be the PDA machine for above AIM, hence it can be defined as M(Q, Σ, Г, δ, q0, Z0, F) where

Q: set of states: {q0, q1, q2}

Σ: set of input symbols: {0, 1, C}

Г: Set of stack symbols: {A, B, Z}

q0: initial state (q0)

Z0: initial stack symbol (Z)

F: set of Final states: { } [Note: Here, set of final states is null as decision of validity of string is based on stack whether it is empty or not. If empty means valid else invalid.]

δ: Transition Function: (Transition state diagram is shown in Figure 2.)

δ(q0, 0, Z) → (q0, AZ)

δ(q0, 1, Z) → (q0, BZ)

δ(q0, 0, A) → (q0, AA)

δ(q0, 0, B) → (q0, AB)

δ(q0, 1, A) → (q0, BA)

δ(q0, 1, B) → (q0, BB)

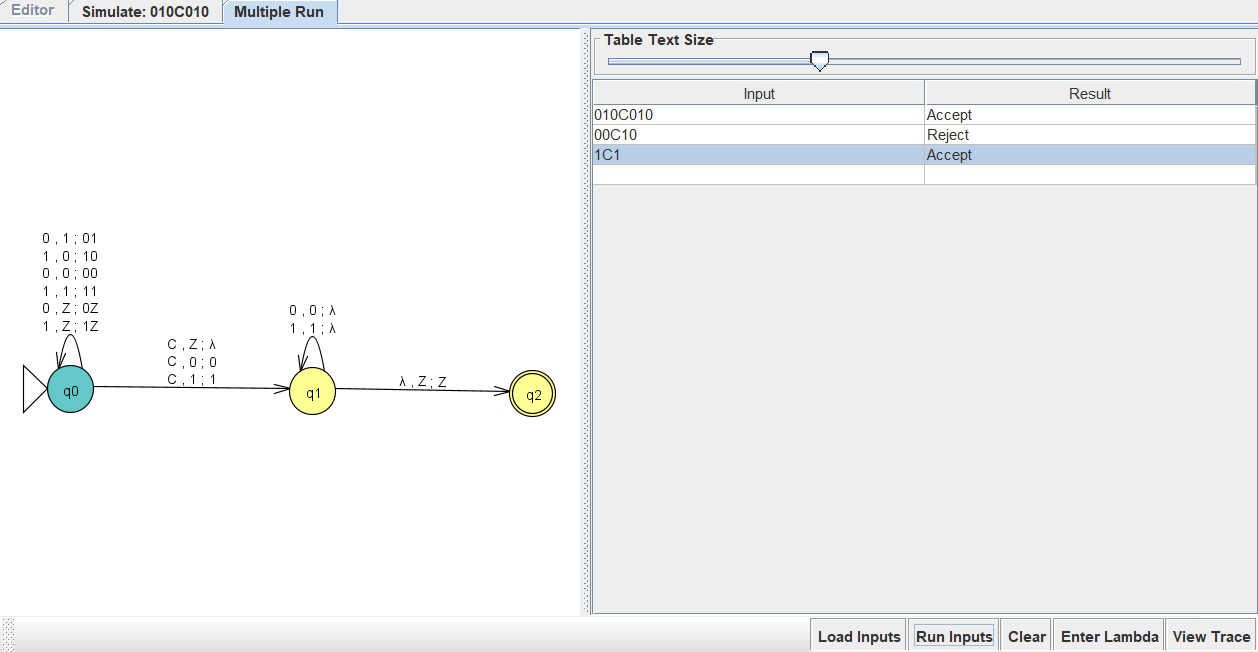
δ(q0, C, A) → (q1, A)

δ(q0, C, B) → (q1, B)

δ(q1, 0, A) → (q1, ε)

δ(q1, 1, B) → (q1, ε)

δ(q1, ε, Z) → (q2, ε)



| def is\_valid\_wCwR(input\_str):  stack = []  i = 0  n = len(input\_str)    # Step 1: Push the first part (w) onto stack until 'C'  while i < n and input\_str[i] != 'C':  if input\_str[i] not in ('0', '1'):  return False  stack.append(input\_str[i])  i += 1    # Check if 'C' was found  if i == n or input\_str[i] != 'C':  return False  i += 1 # Skip the 'C'    # Step 2: Verify second part matches reverse of first part (wR)  while i < n:  if input\_str[i] not in ('0', '1'):  return False  if not stack or stack.pop() != input\_str[i]:  return False  i += 1    # Step 3: Accept if stack is empty  return not stack  # Example usage: input\_str = input("Enter a string (in the form wCwR, where w is binary and C is a special symbol): ") if is\_valid\_wCwR(input\_str):  print("Accepted: The string is in the form wCwR.") else:  print("Rejected: The string is not in the form wCwR.") |
| --- |

OUTPUT:

Enter a string (in the form wCwR, where w is a binary string and C is a special symbol):

1C1 Accepted: The string is in the form wCwR.

Enter a string (in the form wCwR, where w is a binary string and C is a special symbol):

100C01 Rejected: The string is not in the form wCwR.

Time Complexity:

Time Complexity: O(n), where n is the length of the input string. We process each character exactly once.   
Space Complexity: O(n), since in the worst case, we may need to store up to n 0s in the stack.