INDEX

S. No	Program
1.	Write a program to implement “Breadth First Search”
2.	Write a program to implement “Depth First Search”.
3.	Write a program of implementation of the A* algorithm for Transversing problem.
4.	Write a program implementing A-Star Search Algorithm.
5.	Write a python code to create a TIC-TAC-TOE game where a player can play against a computer opponent that uses the MIN-MAX algorithm.

Q- 1 Python Program for BFS Algorithm Implementation.

Sol:-

```
  graph = {
 'A': ['B', 'C'],
 'B': ['D', 'E'],
 'C': ['F'],
 'D': [],
 'E': ['F'],
 'F': []
}

visited_nodes = []  # List to keep track of visited nodes.
queue = []          # Initialize a queue for BFS traversal.

def breadth_first_search(visited, graph, start_node):
    visited.append(start_node)
    queue.append(start_node)

    while queue:
        current_node = queue.pop(0)  # Dequeue the first node in the queue.
        print(current_node, end=" ")  # Print the current node.

        for neighbor in graph[current_node]:
            if neighbor not in visited:
                visited.append(neighbor)
                queue.append(neighbor)

# Driver Code
print("Breadth-First Search:")
breadth_first_search(visited_nodes, graph, 'A')  # Start BFS from node 'A'
```

OUTPUT

```
[Running] python -u "c:\Users\Prachi Priya\ai"
Breadth-First Search:
A B C D E F
[Done] exited with code=0 in 0.093 seconds
```

Q-2 Python Program for DFS Algorithm Implementation

Sol:-

```
# Define the graph as an adjacency list.
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

# Create a set to keep track of visited nodes.
visited_nodes = set()

# Define a Depth-First Search (DFS) function.
def depth_first_search(graph, start_node):
    # Check if the start_node is already visited.
    if start_node not in visited_nodes:
        print(start_node)  # Print the current node.
        visited_nodes.add(start_node)  # Mark the node as visited.

        # Recursively visit all neighbors of the current node.
        for neighbor in graph[start_node]:
            depth_first_search(graph, neighbor)

# Driver Code
print("Depth-First Search Result:")
depth_first_search(graph, 'A')  # Start DFS from node 'A'
```

<u>OUTPUT:</u>

```
[Running] python -u "c:\Users\Prachi Priya\ai"
Depth-First Search Result:
A B D E F C
[Done] exited with code=0 in 0.196 seconds
```

-

Q-3 Python implementation of the A* algorithm for solving search problem.
Sol:-

```
from simpleai.search import SearchProblem, astar

GOAL = 'HELLO WORLD'

class HelloProblem(SearchProblem):
    def actions(self, state):
        return list(' ABCDEFGHIJKLMNOPQRSTUVWXYZ') if len(state) < len(GOAL) else []

    def result(self, state, action):
        return state + action

    def is_goal(self, state):
        return state == GOAL

    def heuristic(self, state):
        return sum([1 if state[i] != GOAL[i] else 0
                    for i in range(len(state))]) + (len(GOAL) - len(state))

problem = HelloProblem(initial_state='')
result = astar(problem)

print(result.state)
print(result.path())
```

```
[Running] python -u "c:\Users\Prachi Priya\ai"
HELLO WORLD
[(None, ''), ('H', 'H'), ('E', 'HE'), ('L', 'HEL'), ('L', 'HELL'), ('O', 'HELLO'), (' ', 'HELLO '), ('W', 'HELLO W'), ('O', 'HELLO
WO'), ('R', 'HELLO WOR'), ('L', 'HELLO WORL'), ('D', 'HELLO WORLD')]

[Done] exited with code=0 in 0.214 seconds
```

Q-4 write a python code for A* algorithm demonstration

Sol:-

```
import heapq

# Define the A* search function with inputs graph, start node, goal node, and heuristics
def custom_astar_search(graph, start, goal, heuristic_fn):
    # Initialize the open list with the starting node and a cost of 0
    open_nodes = [(0, start)]
    # Initialize the closed list as an empty set (nodes already evaluated)
    closed_nodes = set()
    # Initialize a dictionary to store the cost to reach each node, initially set to infinity
    g_costs = {node: float('inf') for node in graph}
    # Set the cost to reach the starting node as 0
    g_costs[start] = 0
    # Initialize a dictionary to store the parent node of each node
    parents = {}

    # Main loop: continue until there are nodes in the open list
    while open_nodes:
        # Pop the node with the lowest f_cost (g_cost + heuristic) from the open list
        _, current_node = heapq.heappop(open_nodes)

        # Check if the current node is the goal node
        if current_node == goal:
            # Reconstruct and return the path from the goal to the start
            path = [current_node]
            while current_node != start:
                current_node = parents[current_node]
                path.append(current_node)
            path.reverse()
            return path

        # If the current node is not the goal, add it to the closed list
        if current_node not in closed_nodes:
            closed_nodes.add(current_node)

            # Explore neighbors of the current node and update their g_costs and f_costs
            for child, cost in graph[current_node]:
                # Calculate the tentative g_cost for the child node
                tentative_g_cost = g_costs[current_node] + cost
                # If this g_cost is lower than the previously recorded g_cost, update it
                if tentative_g_cost < g_costs[child]:
                    g_costs[child] = tentative_g_cost
                    # Calculate the f_cost for the child node (g_cost + heuristic)
                    f_cost = tentative_g_cost + heuristic_fn[child]
```

```
                # Add the child node to the open list with its f_cost
                heapq.heappush(open_nodes, (f_cost, child))
                # Record the parent of the child node
                parents[child] = current_node

    # If the open list becomes empty and the goal is not reached, return None (no path found)
    return None

# Define the graph structure (graph) and heuristic values for the nodes
graph_structure = {
    'Start': [('A', 1), ('B', 4)],
    'A': [('B', 2), ('C', 5), ('D', 12)],
    'B': [('C', 2)],
    'C': [('D', 3)],
    'D': [],
}

heuristic_values = {
    'Start': 7,
    'A': 6,
    'B': 2,
    'C': 1,
    'D': 0,
}

# Define the start and goal nodes
start_node = 'Start'
goal_node = 'D'

# Call the custom A* search function to find the path from the start to the goal
resulting_path = custom_astar_search(graph_structure, start_node, goal_node, heuristic_values)

# Print the result: either the path found or a message indicating no path found
if resulting_path:
    print("Path from", start_node, "to", goal_node, ":", ' -> '.join(resulting_path))
else:
    print("No path found from", start_node, "to", goal_node)
```

OUTPUT

```
[Running] python -u "c:\Users\Prachi Priya\ai"
Path from Start to D : Start -> A -> B -> C -> D

[Done] exited with code=0 in 0.22 seconds
```

Q-5 write a python code to create a TIC-TAC-TOE game where a player can play against a computer opponent that uses the minimax algorithm .

Sol:-

```
import heapq

# Define the A* search function with inputs graph, start node, goal node, and heuristics
def custom_astar_search(graph, start, goal, heuristic_fn):
    # Initialize the open list with the starting node and a cost of 0
    open_nodes = [(0, start)]
    # Initialize the closed list as an empty set (nodes already evaluated)
    closed_nodes = set()
    # Initialize a dictionary to store the cost to reach each node, initially set to infinity
    g_costs = {node: float('inf') for node in graph}
    # Set the cost to reach the starting node as 0
    g_costs[start] = 0
    # Initialize a dictionary to store the parent node of each node
    parents = {}

    # Main loop: continue until there are nodes in the open list
    while open_nodes:
        # Pop the node with the lowest f_cost (g_cost + heuristic) from the open list
        _, current_node = heapq.heappop(open_nodes)

        # Check if the current node is the goal node
        if current_node == goal:
            # Reconstruct and return the path from the goal to the start
            path = [current_node]
            while current_node != start:
                current_node = parents[current_node]
                path.append(current_node)
            path.reverse()
            return path

        # If the current node is not the goal, add it to the closed list
        if current_node not in closed_nodes:
            closed_nodes.add(current_node)

            # Explore neighbors of the current node and update their g_costs and f_costs
            for child, cost in graph[current_node]:
                # Calculate the tentative g_cost for the child node
                tentative_g_cost = g_costs[current_node] + cost
                # If this g_cost is lower than the previously recorded g_cost, update it
```

```
            if tentative_g_cost < g_costs[child]:
                g_costs[child] = tentative_g_cost
                # Calculate the f_cost for the child node (g_cost + heuristic)
                f_cost = tentative_g_cost + heuristic_fn[child]
                # Add the child node to the open list with its f_cost
                heapq.heappush(open_nodes, (f_cost, child))
                # Record the parent of the child node
                parents[child] = current_node

    # If the open list becomes empty and the goal is not reached, return None (no path found)
    return None

# Define the graph structure (graph) and heuristic values for the nodes
graph_structure = {
    'Start': [('A', 1), ('B', 4)],
    'A': [('B', 2), ('C', 5), ('D', 12)],
    'B': [('C', 2)],
    'C': [('D', 3)],
    'D': [],
}

heuristic_values = {
    'Start': 7,
    'A': 6,
    'B': 2,
    'C': 1,
    'D': 0,
}

# Define the start and goal nodes
start_node = 'Start'
goal_node = 'D'

# Call the custom A* search function to find the path from the start to the goal
resulting_path = custom_astar_search(graph_structure, start_node, goal_node,
heuristic_values)

# Print the result: either the path found or a message indicating no path found
if resulting_path:
    print("Path from", start_node, "to", goal_node, ":", ' -> '.join(resulting_path))
else:
    print("No path found from", start_node, "to", goal_node)
```

OUTPUT

```
 | |
-----
 | |
-----
 | |
-----
Enter row (0, 1, 2): 0
Enter column (0, 1, 2): 0
X| |
-----
 |O|
-----
 | |
-----
Enter row (0, 1, 2):
```

Q-5 WAP in python to implement Alpha-Beta pruning to find the optimal value of a node.
Sol:-

```
# Define the tree with modified node values
tree = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F', 'G'],
    'D': [10, 12],
    'E': [8, 6],
    'F': [4, 6],
    'G': [14, 11],
}

# Define the minimax algorithm with alpha-beta pruning
def minimax_alpha_beta(node, alpha, beta, is_maximizing):
    # Check if the node is a string, indicating a non-terminal node
    if isinstance(node, str):
        # Get the children of the current node
        children = tree[node]

        # If it's a maximizing node
        if is_maximizing:
            value = -float('inf')

            # Iterate through the children
            for child in children:
                # Recursively call the function for the child node (switch to minimizing)
                value = max(value, minimax_alpha_beta(child, alpha, beta, False))

                # Update alpha with the maximum value
                alpha = max(alpha, value)

                # Prune the tree if necessary (alpha >= beta)
                if alpha >= beta:
                    break

            return value
        else:  # It's a minimizing node
            value = float('inf')

            # Iterate through the children
            for child in children:
                # Recursively call the function for the child node (switch to maximizing)
```

```
                value = min(value, minimax_alpha_beta(child, alpha, beta, True))

                # Update beta with the minimum value
                beta = min(beta, value)

                # Prune the tree if necessary (alpha >= beta)
                if alpha >= beta:
                    break

            return value
    else:
        # If the node is not a string, it's a leaf node with a known value
        return node

# Call the minimax algorithm with initial values and store the result
optimal_value = minimax_alpha_beta('B', -float('inf'), float('inf'), True)

# Print the optimal value for the root node 'B'
print("Optimal Value for the Root Node (B):", optimal_value)
```

<u>OUTPUT:</u>

```
[Running] python -u "c:\Users\Prachi Priya\tempCodeRunnerFile.python"
Optimal Value for the Root Node (B): 10

[Done] exited with code=0 in 0.084 seconds
```

Q-8 WAP in python to draw membership function curve in fuzzy relations of the following function:

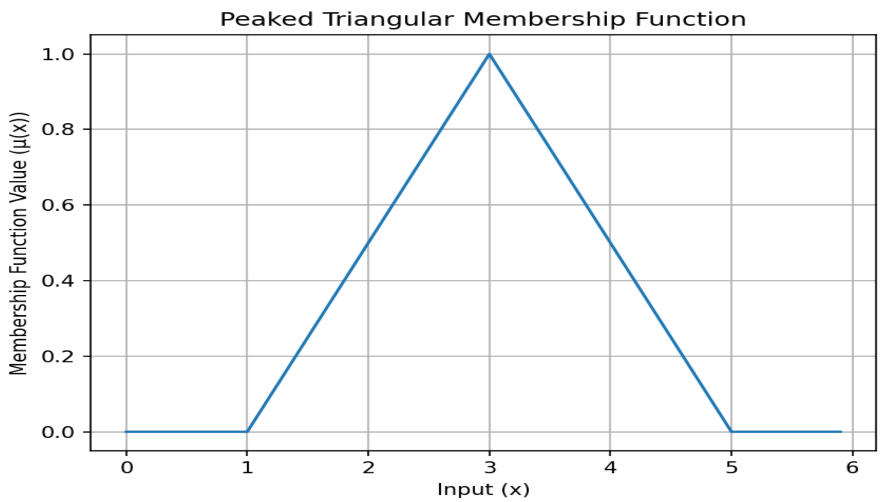
a) Triangular function
b) Gaussian function
c) Trapezoid function

Sol:-

a) For triangular function

```
  import matplotlib.pyplot as plt
import numpy as np

def triangular_membership_function(x, a, b, c):
    """Calculates the triangular membership function value for a given input x."""
    if x < a:
        return 0
    elif x <= b:
        return (x - a) / (b - a)
    elif x <= c:
        return 1
    else:
        return (c - x) / (c - b)

# Define parameters for the triangular membership function
a = 1
b = 3
c = 5

# Generate input values
x = np.arange(0, 6, 0.1)

# Calculate membership function values for each input value
y = [triangular_membership_function(value, a, b, c) for value in x]

# Plot the membership function
plt.plot(x, y)
plt.xlabel('Input (x)')
plt.ylabel('Membership Function Value (μ(x))')
plt.title('Triangular Membership Function')
plt.grid(True)
plt.show()
```

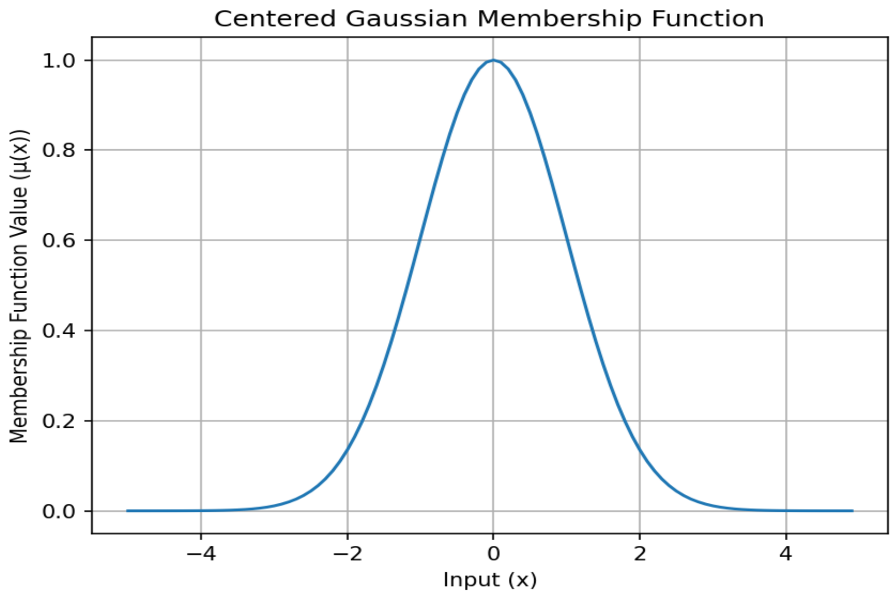
OUTPUT



b) Gaussian function

Sol:
```
import matplotlib.pyplot as plt
import numpy as np

def gaussian_membership_function(x, mean, sigma):
  """Calculates the Gaussian membership function value for a given input x."""
  return np.exp(-(x - mean)*2 / (2 * sigma*2))

# Define parameters for the  Gaussian membership function
mean = 0  # Center the Gaussian curve at x = 0
sigma = 1

# Generate input values
x = np.arange(-5, 5, 0.1)
# Calculate membership function values for each input value
y = [gaussian_membership_function(value, mean, sigma) for value in x]
# Plot the centered Gaussian curve
plt.plot(x, y)
plt.xlabel('Input (x)')
plt.ylabel('Membership Function Value (μ(x))')
plt.title('Centered Gaussian Membership Function')
plt.grid(True)
plt.show()
```

OUTPUT:



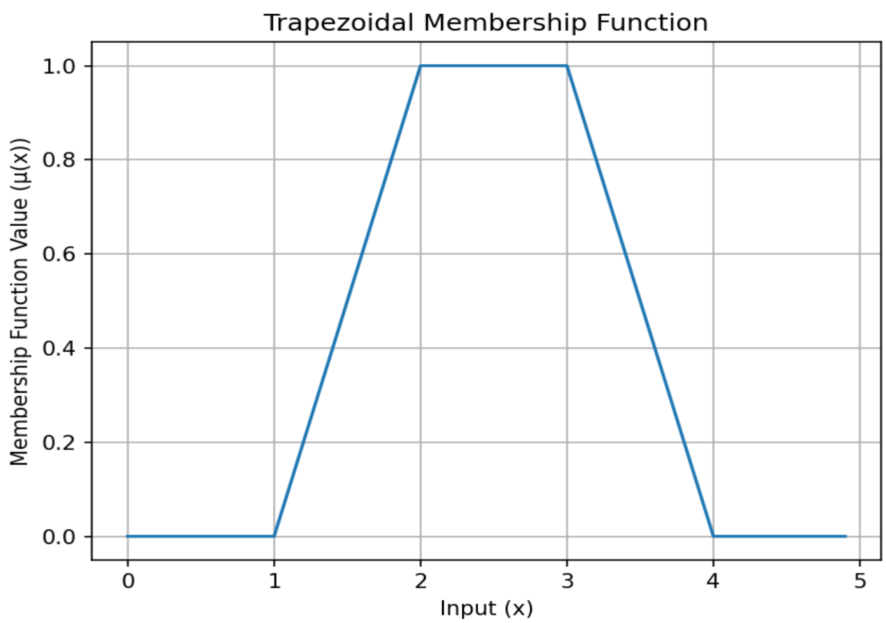
c) Trapezoid function

Sol:

```
import matplotlib.pyplot as plt
import numpy as np

def trapezoidal_membership_function(x, a, b, c, d):
    """Calculates the trapezoidal membership function value for a given input x."""
    if x < a:
        return 0
    elif x <= b:
        return (x - a) / (b - a)
    elif x <= c:
        return 1
    elif x <= d:
        return (d - x) / (d - c)
    else:
        return 0

# Define parameters for the trapezoidal membership function
a = 1
b = 2
c = 3
d = 4

# Generate input values
```

```
x = np.arange(0, 5, 0.1)

# Calculate membership function values for each input value
y = [trapezoidal_membership_function(value, a, b, c, d) for value in x]

# Plot the trapezoidal membership function
plt.plot(x, y)
plt.xlabel('Input (x)')
plt.ylabel('Membership Function Value (μ(x))')
plt.title('Trapezoidal Membership Function')
plt.grid(True)
plt.show()
```

OUTPUT:



Q-9 WAP in python using fuzzy logic for a tipping service based on the service and quality factors

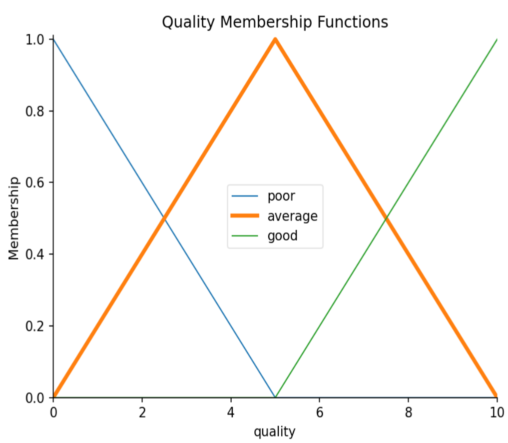
Sol:-

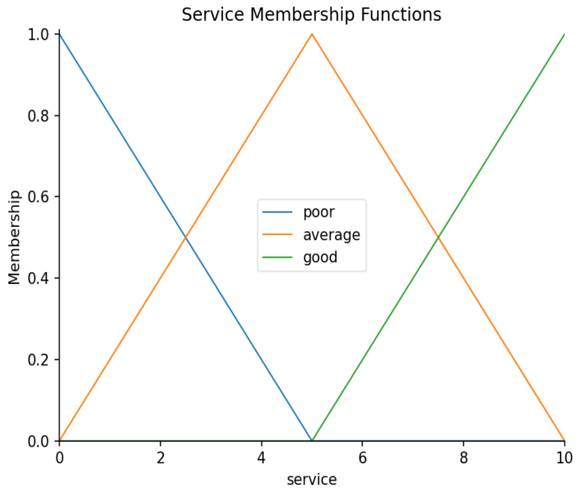
```
import skfuzzy as fuzz
import numpy as np
from skfuzzy import control as ctrl
import matplotlib.pyplot as plt

# Define the universe of discourse
quality = ctrl.Antecedent(np.arange(0, 11, 1), 'quality')
service = ctrl.Antecedent(np.arange(0, 11, 1), 'service')
tip = ctrl.Consequent(np.arange(0, 26,

1), 'tip')

# Auto-membership function population is possible with .automf(3, 5, or 7)
quality.automf(3)
service.automf(3)

# Custom membership functions can be built interactively with a familiar,

# Pythonic API
tip['low'] = fuzz.trimf(tip.universe, [0, 0, 13])
tip['medium'] = fuzz.trimf(tip.universe, [0, 13, 25])
tip['high'] = fuzz.trimf(tip.universe, [13, 25, 25])

# Plot quality membership functions
quality['average'].view()
plt.title('Quality Membership Functions')
plt.savefig('quality_membership_functions.png')
plt.show()

# Plot service membership functions
service.view()
plt.title('Service Membership Functions')
plt.savefig('service_membership_functions.png')
plt.show()

# Plot tip membership functions
tip.view()
plt.title('Tip Membership Functions')
plt.savefig('tip_membership_functions.png')
plt.show()

# Define fuzzy rules
rule1 = ctrl.Rule(quality['poor'] & service['poor'], tip['low'])
```

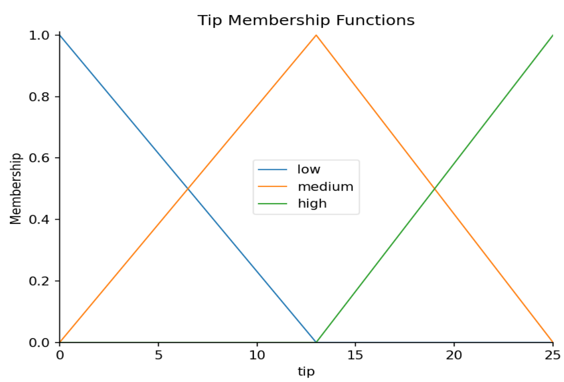
```
rule2 = ctrl.Rule(service['average'], tip['medium'])
rule3 = ctrl.Rule(service['good'] | quality['good'], tip['high'])

# Create a control system
tipping_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])
tipping = ctrl.ControlSystemSimulation(tipping_ctrl)

# Pass inputs to the ControlSystem using Antecedent labels with Pythonic API

# Note: if you like passing many inputs all at once, use .inputs(dict_of_data)
tipping.input['quality'] = 25
tipping.input['service'] = 15


# Crunch the numbers
tipping.compute()

# Print the output
print("Recommended tip amount:", tipping.output['tip'])

# Plot tip output
tip.view(sim=tipping)
plt.title('Tip Output')
plt.savefig('tip_output.png')
plt.show()
```

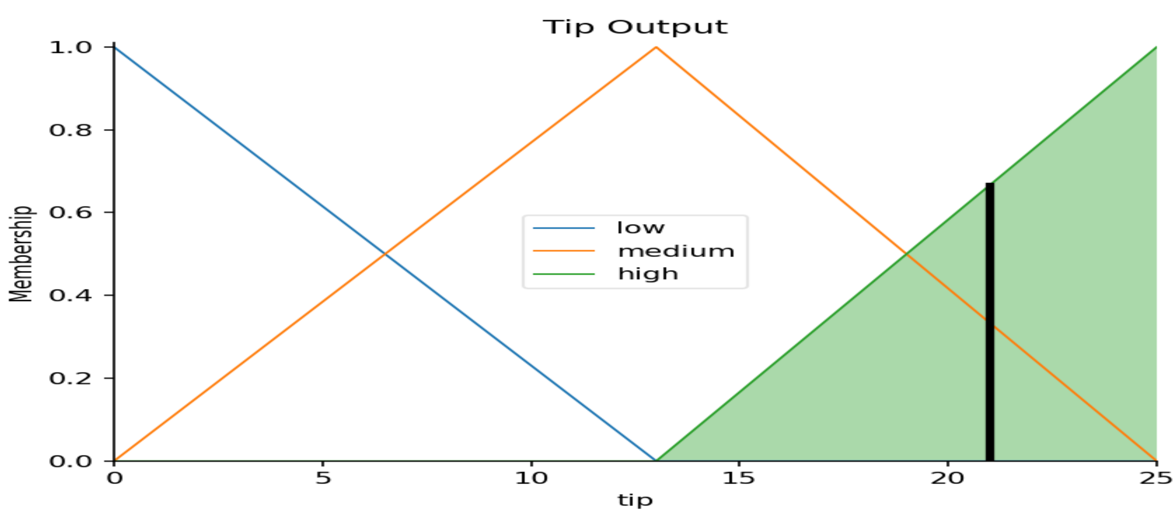
OUTPUT:

```
[Running] python -u "c:\Users\Prachi Priya\tempCodeRunnerFile.python"
Recommended tip amount: 21.0
```




Service Membership Functions
Membership
service
poor
average
good


Tip Membership Functions
Membership
tip
low
medium
high


Tip Output
Membership
tip
low
medium
high

Program 10: Python Program for Linear Regression.

xi (Week)	yi (Sales in thousands)
1	1.2
2	1.8
3	2.6
4	3.2
5	3.8

Predict the 7th & 12th Weeks Sale

Code:

```
import matplotlib.pyplot as plt


# Function to calculate the average of a list
def average(lst):
    total = sum(lst)
    avg = total / len(lst)
    return avg


# Function to perform linear regression and predict sales for given weeks
def predict_sale(x, y):
    # Calculate the average values for x and y
    x_bar = average(x)
    y_bar = average(y)

    # Calculate the necessary values for linear regression formula
    x_sq_bar = average([i**2 for i in x])
    xy_bar = average([x[i]*y[i] for i in range(len(x))])

    # Calculate the slope (al) and y-intercept (a0) for the linear regression line
    al = (xy_bar - (x_bar * y_bar)) / (x_sq_bar - (x_bar**2))
    a0 = y_bar - al * x_bar
```

```
    # Get user input for weeks to predict sales
    week_input = input("Enter the week(s) for which you want to predict the sales (separated by commas): ")
    weeks = [int(week) for week in week_input.split(',')]

    # Calculate predicted sales for the input weeks
    week_sales = [a0 + (al * week) for week in weeks]

    # Print the predicted sales for each input week
    for week, sales in zip(weeks, week_sales):
        print(f"Predicted sales for week {week}: {sales}")

    # Plotting the actual values, predicted values, and linear regression line
    plt.scatter(x, y, color='red', label='Actual Values')
    plt.scatter(weeks, week_sales, color='green', label='Predicted Values')
    plt.plot(x, [a0 + al * i for i in x], color='blue', label='Linear Regression')
    plt.plot([x[-1]] + weeks, [y[-1]] + week_sales, linestyle='dashed', color='blue', label='Extended Linear Regression')

    # Set plot settings
    plt.xticks(range(min(x), max(weeks) + 1))
    plt.xlabel('No. of Weeks')
    plt.ylabel('Sales (in thousands)')
    plt.legend()
    plt.show()

# Sample data
x = [1, 2, 3, 4, 5]  # Assuming weeks are represented as integers starting from 1
y = [1.2, 1.8, 2.6, 3.2, 3.8]

# Call the function to predict sales and plot the results
```

predict_sale(x, y)

Output:

```
Enter the week(s) for which you want to predict the sales (separated by commas): 7,12
Predicted sales for week 7: 5.159999999999998
Predicted sales for week 12: 8.459999999999997
```