1. A* search:

```
import heapq
# Define the graph representing the map of Romania
romania_map = {
    'Arad': {'Zerind': 75, 'Sibiu': 140, 'Timisoara': 118},
    'Zerind': {'Arad': 75, 'Oradea': 71},
    'Oradea': {'Zerind': 71, 'Sibiu': 151},
    'Sibiu': {'Arad': 140, 'Oradea': 151, 'Fagaras': 99, 'Rimnicu Vilcea': 80},
    'Timisoara': {'Arad': 118, 'Lugoj': 111},
    'Lugoj': {'Timisoara': 111, 'Mehadia': 70},
    'Mehadia': {'Lugoj': 70, 'Drobeta': 75},
    'Drobeta': {'Mehadia': 75, 'Craiova': 120},
    'Craiova': {'Drobeta': 120, 'Rimnicu Vilcea': 146, 'Pitesti': 138},
    'Rimnicu Vilcea': {'Sibiu': 80, 'Craiova': 146, 'Pitesti': 97},
    'Fagaras': {'Sibiu': 99, 'Bucharest': 211},
    'Pitesti': {'Rimnicu Vilcea': 97, 'Craiova': 138, 'Bucharest': 101},
    'Bucharest': {'Fagaras': 211, 'Pitesti': 101, 'Giurgiu': 90, 'Urziceni': 85},
    'Giurgiu': {'Bucharest': 90},
    'Urziceni': {'Bucharest': 85, 'Hirsova': 98, 'Vaslui': 142},
    'Hirsova': {'Urziceni': 98, 'Eforie': 86},
    'Eforie': {'Hirsova': 86},
    'Vaslui': {'Urziceni': 142, 'Iasi': 92},
    'Iasi': {'Vaslui': 92, 'Neamt': 87},
    'Neamt': {'Iasi': 87}
}

# Define the heuristic function (straight-line distance to Bucharest)
heuristic = {
    'Arad': 366,
    'Zerind': 374,
```

```
    'Oradea': 380,
    'Sibiu': 253,
    'Timisoara': 329,
    'Lugoj': 244,
    'Mehadia': 241,
    'Drobeta': 242,
    'Craiova': 160,
    'Rimnicu Vilcea': 193,
    'Fagaras': 178,
    'Pitesti': 98,
    'Bucharest': 0,
    'Giurgiu': 77,
    'Urziceni': 80,
    'Hirsova': 151,
    'Eforie': 161,
    'Vaslui': 199,
    'Iasi': 226,
    'Neamt': 234
}

def astar_search(graph, start, goal, heuristic):
    open_list = [(0, start)]
    closed_list = set()
    parent = {}
    g_score = {city: float('inf') for city in graph}
    g_score[start] = 0

    while open_list:
        current_node = heapq.heappop(open_list)[1]

        if current_node == goal:
```

```
            path = []
            while current_node in parent:
                path.append(current_node)
                current_node = parent[current_node]
            path.append(start)
            return path[::-1]

        closed_list.add(current_node)

        for neighbor, cost in graph[current_node].items():
            if neighbor in closed_list:
                continue

            tentative_g_score = g_score[current_node] + cost

            if tentative_g_score < g_score[neighbor]:
                parent[neighbor] = current_node
                g_score[neighbor] = tentative_g_score
                f_score = tentative_g_score + heuristic[neighbor]
                heapq.heappush(open_list, (f_score, neighbor))

    return None

if __name__ == "__main__":
    start_city = 'Mehadia'
    goal_city = 'Craiova'
    shortest_path = astar_search(romania_map, start_city, goal_city, heuristic)

    if shortest_path:
        print(f"Shortest path from {start_city} to {goal_city}: {shortest_path}")
```

```
        total_distance = sum(romania_map[shortest_path[i]][shortest_path[i+1]] for i in
range(len(shortest_path)-1))
        print(f"Total distance: {total_distance} km")
    else:
        print(f"No path found from {start_city} to {goal_city}.")
```

2. Vacuum cleaner:

```
import random

def display(room):
    for row in room:
        print(row)

room = [
    [1, 1, 1, 1],
    [1, 1, 1, 1],
    [1, 1, 1, 1],
    [1, 1, 1, 1],
]

print("All the rooms are dirty")
display(room)

x = 0
y = 0

while x < 4:
    while y < 4:
        room[x][y] = random.choice([0, 1])
        y += 1
    x += 1
    y = 0

print("Before cleaning the room I detect all of these random dirts")
display(room)

x = 0
y = 0
z = 0

while x < 4:
    while y < 4:
        if room[x][y] == 1:
            print("Vacuum in this location now,", x, y)
```

```
                room[x][y] = 0
                print("cleaned", x, y)
                z += 1
            y += 1
        x += 1
        y = 0

    pro = (100 - ((z / 16) * 100))
    print("Room is clean now, Thanks for using")
    display(room)
    print('performance=', pro, '%')
```

3. Alpha-beta pruning:

```
maximum, minimum = 1000, -1000


def fun_alphabeta(d, node, maxp, v, A, B):
    if d == 0:  # Assuming leaf nodes are at depth 0
        return v[node]


    if maxp:
        best = minimum
        for i in range(0, 2):
            value = fun_alphabeta(d - 1, node * 2 + i, False, v, A, B)
            best = max(best, value)
            A = max(A, best)
            if B <= A:
                break
        return best
    else:
        best = maximum
        for i in range(0, 2):
            value = fun_alphabeta(d - 1, node * 2 + i, True, v, A, B)
```

```
        best = min(best, value)

        B = min(B, best)

        if B <= A:

            break

    return best


scr = []

x = int(input("Enter total number of leaf nodes: "))

for i in range(x):

    y = int(input("Enter node value: "))

    scr.append(y)


d = int(input("Enter depth value: "))

node = int(input("Enter node value: "))


print("The optimal value is:", fun_alphabeta(d, node, True, scr, minimum, maximum))
```

4. Tic-tac-toe:

```
import os

turn = 'X'
win = False
spaces = 9

def draw(board):
    for i in range(6, -1, -3):
        print(' ' + board[i] + '|' + board[i+1] + '|' + board[i+2])

def takeinput(board, spaces, turn):
    pos = -1
    print(turn + "'s turn:")
    while pos == -1:
        try:
            print("Pick position 1-9:")
            pos = int(input())
```

```
            if pos < 1 or pos > 9:
                pos = -1
            elif board[pos - 1] != ' ':
                pos = -1
        except ValueError:
            print("Enter a valid position")
            pos = -1

    spaces -= 1
    board[pos - 1] = turn

    if turn == 'X':
        turn = 'O'
    else:
        turn = 'X'

    return board, spaces, turn

def checkwin(board):
    # Check rows
    for i in range(0, 9, 3):
        if board[i] != ' ' and board[i] == board[i+1] == board[i+2]:
            return board[i]

    # Check columns
    for i in range(3):
        if board[i] != ' ' and board[i] == board[i+3] == board[i+6]:
            return board[i]

    # Check diagonals
    if board[0] != ' ' and board[0] == board[4] == board[8]:
        return board[0]
    if board[2] != ' ' and board[2] == board[4] == board[6]:
        return board[2]

    return 0

board = [' '] * 9

while not win and spaces:
    draw(board)
    board, spaces, turn = takeinput(board, spaces, turn)
    win = checkwin(board)
    os.system('cls' if os.name == 'nt' else 'clear')  # 'cls' is for Windows, use 'clear' for
Linux/Mac

if not win and not spaces:
    print("Draw")
```

```
elif win:
    print(f'{win} wins')
```

5. Mcculloch Pitts:

```
def mcculloch_pitts(inputs, weights, threshold):
    """McCulloch-Pitts neuron model."""
    # Ensure the number of inputs matches the number of weights
    assert len(inputs) == len(weights), "Number of inputs must match number of weights"

    # Calculate the weighted sum of inputs
    weighted_sum = sum(x * w for x, w in zip(inputs, weights))

    # Apply the threshold function
    output = 1 if weighted_sum >= threshold else 0

    return output

def test_logic_gate(logic_gate):
    """Test a logic gate using McCulloch-Pitts neuron."""
    print(f"Testing {logic_gate} gate:")

    if logic_gate == "AND":
        # AND gate
        inputs = [(0, 0), (0, 1), (1, 0), (1, 1)]
        weights = (1, 1)
        threshold = 2

    elif logic_gate == "OR":
        # OR gate
        inputs = [(0, 0), (0, 1), (1, 0), (1, 1)]
        weights = (1, 1)
        threshold = 1

    elif logic_gate == "XOR":
        # XOR gate (requires a combination of AND, OR, and NOT gates)
        inputs = [(0, 0), (0, 1), (1, 0), (1, 1)]
        weights_and = (1, 1)
        weights_or = (1, 1)
        weights_not = (-1,)
        threshold = 1

        for input_pair in inputs:
            input1, input2 = input_pair
            # XOR is implemented using a combination of AND, OR, and NOT
            and_result = mcculloch_pitts(input_pair, weights_and, threshold)
```

```
            or_result = mcculloch_pitts(input_pair, weights_or, threshold)
            not_result = mcculloch_pitts((and_result,), weights_not, threshold)

            xor_result = mcculloch_pitts((or_result, not_result), weights_and, threshold)
            print(f"{input_pair}: {xor_result}")

        return

    elif logic_gate == "AND NOT":
        # AND NOT gate
        inputs = [(0, 0), (0, 1), (1, 0), (1, 1)]
        weights = (1, -1)
        threshold = 0

    else:
        print("Invalid logic gate.")
        return

    # Test the logic gate
    for input_pair in inputs:
        result = mcculloch_pitts(input_pair, weights, threshold)
        print(f"{input_pair}: {result}")

# Test different logic gates
test_logic_gate("AND")
test_logic_gate("OR")
test_logic_gate("XOR")
test_logic_gate("AND NOT")
```

6. Perceptron:

```
import numpy as np


class Perceptron:
    def __init__(self, input_size, learning_rate=0.1, threshold=0.0, max_iterations=1000):
        self.weights = np.random.rand(input_size)
        self.threshold = threshold
        self.learning_rate = learning_rate
        self.max_iterations = max_iterations


    def activate(self, net_input):
```

```
        return 1 if net_input >= self.threshold else 0

    def train(self, input_data, labels):
        iteration = 0

        while iteration < self.max_iterations:
            converged = True

            for i in range(len(input_data)):
                input_vector = np.array(input_data[i])
                label = labels[i]

                net_input = np.dot(input_vector, self.weights)
                predicted_output = self.activate(net_input)
                error = label - predicted_output

                if error != 0:
                    converged = False
                    self.weights += self.learning_rate * error * input_vector

            if converged:
                break

            iteration += 1

        return iteration

# Example usage
input_data = [
    [0, 0],
    [0, 1],
```

```
    [1, 0],
    [1, 1]
]

labels = [0, 0, 0, 1]

perceptron = Perceptron(input_size=2)
iterations = perceptron.train(input_data, labels)

print("Converged in {} iterations".format(iterations))
print("Final weights:", perceptron.weights)
```