PYTHON CODES

1) 8-Puzzle problem

```
import heapq
def heuristic(state, goal):
  return sum(abs(b % 3 - g % 3) + abs(b // 3 - g // 3) for b, g in zip(state, 
goal) if b)
def astar_search(start, goal):
  open_list, closed = [(0, start)], set()
  came_from, cost = {}, {tuple(start): 0}
  while open_list:
    _, current = heapq.heappop(open_list)
    if current == goal:
      break
    closed.add(tuple(current))
    for next_state in get_neighbors(current):
      if tuple(next_state) not in closed:
        new_cost = cost[tuple(current)] + 1
        if tuple(next_state) not in cost or new_cost < 
cost[tuple(next_state)]:
          cost[tuple(next_state)] = new_cost
          priority = new_cost + heuristic(next_state, goal)
          heapq.heappush(open_list, (priority, next_state))
          came_from[tuple(next_state)] = current
  return came_from, cost
def get_neighbors(state):
  neighbors, zero = [], state.index(0)
  moves = [(-3, 0), (3, 0), (-1, -1), (1, 1)]
  for move, col in moves:
    new_pos = zero + move
    if 0 <= new_pos < len(state) and (col == 0 or zero // 3 == new_pos // 
3):
      neighbor = state[:]
      neighbor[zero], neighbor[new_pos] = neighbor[new_pos], 
neighbor[zero]
      neighbors.append(neighbor)
  return neighbors
def reconstruct_path(came_from, start, goal):
```

```
    current, path = goal, [goal]
    while current != start:
        current = came_from[tuple(current)]
        path.append(current)
    return path[::-1]
start_state = [1, 0, 3, 4, 2, 5, 6, 7, 8]
goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]
came_from, cost = astar_search(start_state, goal_state)
path = reconstruct_path(came_from, start_state, goal_state)
print("Solution path:")
for state in path:
    print(state)
print("Cost:", cost[tuple(goal_state)])
```

2)**8-QUEEN PROBLEM:**

```
N = 8
def solveNQueens(board, col):
        if col == N:
                print(board)
                return True
        for i in range(N):
                if isSafe(board, i, col):
                        board[i][col] = 1
                        if solveNQueens(board, col + 1):
                                return True
                        board[i][col] = 0
        return False
def isSafe(board, row, col):
        for x in range(col):
                if board[row][x] == 1:
                        return False
        for x, y in zip(range(row, -1, -1), range(col, -1, -1)):
                if board[x][y] == 1:
                        return False
        for x, y in zip(range(row, N, 1), range(col, -1, -1)):
                if board[x][y] == 1:
                        return False
```

```
        return True
board = [[0 for x in range(N)] for y in range(N)]
if not solveNQueens(board, 0):
        print("No solution found")
```

3)A STAR ALGORITHM:

```
import heapq
def a_star(start, goal, graph, heuristic):
    open_set = [(heuristic[start], start)]
    g_score = {node: float('inf') for node in graph}
    g_score[start] = 0
    came_from = {}

    while open_set:
        _, current = heapq.heappop(open_set)
        if current == goal:
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            path.append(start)
            return path[::-1]

        for neighbor, cost in graph[current].items():
            tentative_g_score = g_score[current] + cost
            if tentative_g_score < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g_score
                heapq.heappush(open_set, (tentative_g_score +
heuristic[neighbor], neighbor))
    return None
if __name__ == "__main__":
    graph = {
        'A': {'B': 1, 'C': 4},
        'B': {'A': 1, 'C': 2, 'D': 5},
        'C': {'A': 4, 'B': 2, 'D': 1},
        'D': {'B': 5, 'C': 1}
    }
```

```
    heuristic = {
        'A': 7,
        'B': 6,
        'C': 2,
        'D': 0
    }

    start = 'A'
    goal = 'D'
    path = a_star(start, goal, graph, heuristic)

    if path:
        print("Path found:", path)
    else:
        print("No path found")
```

4)BFS:

```
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    visited.add(start)

    while queue:
        vertex = queue.popleft()
        print(vertex, end=" ")
        for neighbor in graph[vertex]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
if __name__ == "__main__":
    graph = {
        'A': ['B', 'C'],
        'B': ['A', 'D', 'E'],
        'C': ['A', 'F'],
        'D': ['B'],
```

```
        'E': ['B', 'F'],
        'F': ['C', 'E']
    }
    print("BFS Traversal starting from node 'A':")
    bfs(graph, 'A')
```

5)ALPHA BETA PRUNINBG:

```
print("Alpha-Beta Pruning Algorithm")
MAX, MIN = 1000, -1000

def minimax(depth, nodeIndex, maximizingPlayer, values, alpha, beta):
    if depth == 3:
        return values[nodeIndex]

    if maximizingPlayer:
        best = MIN
        for i in range(0, 2):
            val = minimax(depth + 1, nodeIndex * 2 + i, False, values, alpha,
beta)
            best = max(best, val)
            alpha = max(alpha, best)
            if beta <= alpha:
                break
        return best
    else:
        best = MAX
        for i in range(0, 2):
            val = minimax(depth + 1, nodeIndex * 2 + i, True, values, alpha,
beta)
            best = min(best, val)
            beta = min(beta, best)
            if beta <= alpha:
                break
        return best

if __name__ == "__main__":
    values = [3, 5, 6, 9, 1, 2, -1, 0]
    print(values)
```

```
    print("The optimal value is:", minimax(0, 0, True, values, MIN, MAX))
```

6)CRYPTO ARTHMETIC:

```
from itertools import permutations
def solve_cryptarithmetic(puzzle):
    parts = puzzle.split('+')
    left = parts[0].strip()
    right, result = parts[1].split('=')
    right = right.strip()
    result = result.strip()
    letters = set(left + right + result)
    for perm in permutations(range(10), len(letters)):
        mapping = dict(zip(letters, perm))
        if mapping[left[0]] == 0 or mapping[right[0]] == 0 or
mapping[result[0]] == 0:
            continue
        left_num = int(''.join(str(mapping[char]) for char in left))
        right_num = int(''.join(str(mapping[char]) for char in right))
        result_num = int(''.join(str(mapping[char]) for char in result))
        if left_num + right_num == result_num:
            return mapping
    return None  # No solution found
# Example usage:
puzzle = "BASE + BALL = GAMES"
solution = solve_cryptarithmetic(puzzle)
if solution:
    print("Solution found:")
    for letter, digit in solution.items():
        print(f"{letter}: {digit}")
else:
    print("No solution found.")
```

7)DFS:

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start, end=" ")
```

```
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
if __name__ == "__main__":
    graph = {
        1: [2, 3],
        2: [1, 4],
        3: [1, 5],
        4: [2],
        5: [3]
    }

    print("DFS Traversal starting from node 1:")
    dfs(graph, 1)
```

8)FEED FORWARD:

```
        import numpy as np

# Sigmoid activation function and its derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

# Initialize parameters
def initialize_parameters(input_dim, hidden_dim, output_dim):
    np.random.seed(42)  # For reproducibility
    W1 = np.random.randn(input_dim, hidden_dim)  # Weights for input
to hidden layer
    b1 = np.zeros((1, hidden_dim))  # Biases for hidden layer
    W2 = np.random.randn(hidden_dim, output_dim)  # Weights for
hidden to output layer
    b2 = np.zeros((1, output_dim))  # Biases for output layer
    return W1, b1, W2, b2

# Forward propagation
```

```
def forward_propagation(X, W1, b1, W2, b2):
    Z1 = np.dot(X, W1) + b1
    A1 = sigmoid(Z1)
    Z2 = np.dot(A1, W2) + b2
    A2 = sigmoid(Z2)
    return A2, A1

# Compute cost (mean squared error)
def compute_cost(A2, Y):
    m = Y.shape[0]
    cost = np.sum((A2 - Y) ** 2) / (2 * m)
    return cost

# Backward propagation
def backward_propagation(X, Y, A2, A1, W2):
    m = X.shape[0]
    dA2 = A2 - Y
    dZ2 = dA2 * sigmoid_derivative(A2)
    dW2 = np.dot(A1.T, dZ2) / m
    db2 = np.sum(dZ2, axis=0, keepdims=True) / m
    dA1 = np.dot(dZ2, W2.T)
    dZ1 = dA1 * sigmoid_derivative(A1)
    dW1 = np.dot(X.T, dZ1) / m
    db1 = np.sum(dZ1, axis=0, keepdims=True) / m
    return dW1, db1, dW2, db2

# Update parameters using gradient descent
def update_parameters(W1, b1, W2, b2, dW1, db1, dW2, db2,
learning_rate):
    W1 -= learning_rate * dW1
    b1 -= learning_rate * db1
    W2 -= learning_rate * dW2
    b2 -= learning_rate * db2
    return W1, b1, W2, b2

# Training the neural network
def train(X, Y, hidden_dim, epochs, learning_rate):
```

```
    input_dim = X.shape[1]
    output_dim = Y.shape[1]
    W1, b1, W2, b2 = initialize_parameters(input_dim, hidden_dim,
output_dim)

    for epoch in range(epochs):
        A2, A1 = forward_propagation(X, W1, b1, W2, b2)
        cost = compute_cost(A2, Y)
        dW1, db1, dW2, db2 = backward_propagation(X, Y, A2, A1, W2)
        W1, b1, W2, b2 = update_parameters(W1, b1, W2, b2, dW1, db1,
dW2, db2, learning_rate)

        if epoch % 100 == 0:
            print(f"Epoch {epoch}: Cost {cost}")

    return W1, b1, W2, b2

# Predict function
def predict(X, W1, b1, W2, b2):
    A2, _ = forward_propagation(X, W1, b1, W2, b2)
    return A2

# Sample data (for demonstration purposes)
X = np.array([
    [0.1, 0.2],
    [0.9, 0.8],
    [0.8, 0.9],
    [0.2, 0.1]
])
Y = np.array([
    [0.1],
    [0.9],
    [0.8],
    [0.2]
])

# Training parameters
```

```
hidden_dim = 5
epochs = 1000
learning_rate = 0.01

# Train the model
W1, b1, W2, b2 = train(X, Y, hidden_dim, epochs, learning_rate)

# Predict
predictions = predict(X, W1, b1, W2, b2)

print("Predicted Output after training:")
print(predictions)
```

9)MAP COLORING:

```
class CSP:
    def __init__(self, variables, domains):  # Corrected the method name
        self.variables = variables
        self.domains = domains

    def is_consistent(self, variable, assignment):
        return all(assignment[neighbor] != assignment[variable] for
neighbor in self.variables[variable] if neighbor in assignment)

    def backtracking_search(self, assignment={}):
        if len(assignment) == len(self.variables):
            return assignment

        unassigned = [var for var in self.variables if var not in assignment]
        first_unassigned = unassigned[0]

        for value in self.domains[first_unassigned]:
            assignment[first_unassigned] = value
            if self.is_consistent(first_unassigned, assignment):
                result = self.backtracking_search(assignment)
                if result is not None:
                    return result
            assignment.pop(first_unassigned)
```

```
        return None

def main():
    # Define the variables and domains for the Map Coloring problem
    variables = {
        'WA': ['NT', 'SA'],
        'NT': ['WA', 'SA', 'Q'],
        'SA': ['WA', 'NT', 'Q', 'NSW', 'V'],
        'Q': ['NT', 'SA', 'NSW'],
        'NSW': ['Q', 'SA', 'V'],
        'V': ['SA', 'NSW']
    }

    domains = {
        'WA': ['red', 'green', 'blue'],
        'NT': ['red', 'green', 'blue'],
        'SA': ['red', 'green', 'blue'],
        'Q': ['red', 'green', 'blue'],
        'NSW': ['red', 'green', 'blue'],
        'V': ['red', 'green', 'blue']
    }

    csp = CSP(variables, domains)
    solution = csp.backtracking_search()

    if solution is not None:
        print("Solution found:")
        for var, val in solution.items():
            print(f"{var}: {val}")
    else:
        print("No solution found.")

if __name__ == "__main__":  # Corrected the variable names
    main()
```

10)MIN MAX ALGORITHM:

```
 import math

def minimax (curDepth, nodeIndex,
maxTurn, scores,
targetDepth):

  # base case : targetDepth reached
  if (curDepth == targetDepth):
      return scores[nodeIndex]

  if (maxTurn):
      return max(minimax(curDepth + 1, nodeIndex * 2,
      False, scores, targetDepth),
      minimax(curDepth + 1, nodeIndex * 2 + 1,
      False, scores, targetDepth))

  else:
      return min(minimax(curDepth + 1, nodeIndex * 2,
      True, scores, targetDepth),
      minimax(curDepth + 1, nodeIndex * 2 + 1,
      True, scores, targetDepth))

  # Driver code
scores = [3, 5, 2, 9, 12, 5, 23, 23]
treeDepth = math.log(len(scores), 2)
print("The optimal value is : ", end = "")
print(minimax(0, 0, True, scores, treeDepth))
```

11)MISSINORIES-CALIBERS:

```
from collections import deque

def is_valid_state(m, c):
    return 0 <= m <= 3 and 0 <= c <= 3 and (m == 0 or m >= c) and ((3 - m)
== 0 or (3 - m) >= (3 - c))

def get_next_states(state):
    m_left, c_left, b_left, m_right, c_right, b_right = state
```

```
    next_states = []
    for i in range(3):
        for j in range(3):
            if 1 <= i + j <= 2:
                if b_left:
                    new_state = (m_left - i, c_left - j, 0, m_right + i, c_right + j, 1)
                else:
                    new_state = (m_left + i, c_left + j, 1, m_right - i, c_right - j, 0)
                if is_valid_state(new_state[0], new_state[1]) and
is_valid_state(new_state[3], new_state[4]):
                    next_states.append(new_state)
    return next_states

def bfs():
    start, goal = (3, 3, 1, 0, 0, 0), (0, 0, 0, 3, 3, 1)
    queue = deque([(start, [])])
    visited = {start}
    while queue:
        state, path = queue.popleft()
        if state == goal:
            return path + [goal]
        for next_state in get_next_states(state):
            if next_state not in visited:
                visited.add(next_state)
                queue.append((next_state, path + [state]))
    return None

def print_solution(solution):
    if solution:
        print("Solution found!")
        for i, state in enumerate(solution):
            print(f"Step {i + 1}: {state[:3]} || {state[3:]}")
    else:
        print("No solution found.")

if __name__ == "__main__":
    print_solution(bfs())
```

12)TIC TAC TOE:

```
def print_board(x_state, z_state):
   board = [str(i) if x_state[i] == z_state[i] == 0 else ('X' if x_state[i] else
'O') for i in range(9)]
   print(f" {board[0]} | {board[1]} | {board[2]} ")
   print("---|---|---")
   print(f" {board[3]} | {board[4]} | {board[5]} ")
   print("---|---|---")
   print(f" {board[6]} | {board[7]} | {board[8]} ")

def check_win(x_state, z_state):
   wins = [[0, 1, 2], [3, 4, 5], [6, 7, 8], [0, 3, 6], [1, 4, 7], [2, 5, 8], [0, 4, 8],
[2, 4, 6]]
   for win in wins:
      if sum(x_state[i] for i in win) == 3:
         print("X won the game")
         return True
      elif sum(z_state[i] for i in win) == 3:
         print("O won the game")
         return True
   return False

if __name__ == "__main__":
   total_turns = 9
   x_state = [0] * 9
   z_state = [0] * 9
   turn = 1  # 1 for X and 0 for O
   print("Welcome to TIC-TAC-TOE")
   while True:
      print_board(x_state, z_state)
      player = 'X' if turn == 1 else 'O'
      print(f"{player}'s Chance")
      value = int(input("Please enter a value (0-8): "))

      if not (0 <= value <= 8 and x_state[value] == z_state[value] == 0):
         print("Invalid move! Please choose an empty cell (0-8).")
         continue
```

```
    if turn == 1:
        x_state[value] = 1
    else:
        z_state[value] = 1

    total_turns -= 1

    if check_win(x_state, z_state) or total_turns == 0:
        print("GAME OVER")
        print_board(x_state, z_state)
        break
    turn = 1 – turn
```

13)TSP:

```
import itertools
import math
def calculate_distance(p1, p2):
    return math.hypot(p1[0] - p2[0], p1[1] - p2[1])
def total_distance(points, order):
    return sum(calculate_distance(points[order[i]], points[order[(i + 1) %
len(order)]]) for i in range(len(order)))
def tsp_bruteforce(points):
    return min(
        (total_distance(points, perm), perm)
        for perm in itertools.permutations(range(len(points)))
    )
if __name__ == "__main__":
    points = [(0, 0), (1, 5), (5, 2), (6, 6)]
    min_distance, optimal_order = tsp_bruteforce(points)
    print("Minimum Distance:", min_distance)
    print("Optimal Order:", optimal_order)
```

14)VACCUME CLEANER:

```
a=[[1,0,1,0],[1,1,1,1],[1,0,1,1],[1,0,1,1]]
print("Room With dust are represented as 1 and Room With NO Dust
represented as 0\nRoom Structure with and without Dirt\n",a)
```

```
print("AGENT is Cleaning")
for i in range(4):
    for j in range(4):
        if(a[i][j]==1):
            print("Agent Cleaned Location",i,j)
            a[i][j]=0
    print("Agent Cleaned Room",i+1)
print("Room After Cleaning \n",a)
```

15)WATER JUG:

```
from collections import deque

def solve_water_jug_problem(cap1, cap2, target):
    queue = deque([(0, 0, [])])
    visited = set([(0, 0)])

    while queue:
        jug1, jug2, path = queue.popleft()

        if jug1 == target or jug2 == target:
            path.append((jug1, jug2))
            for step in path:
                print(f"Jug1: {step[0]} liters, Jug2: {step[1]} liters")
            return

        for next_jug1, next_jug2 in [(cap1, jug2), (jug1, cap2), (0, jug2),
(jug1, 0),
                                     (jug1 - min(jug1, cap2 - jug2), jug2 + min(jug1, cap2
- jug2)),
                                     (jug1 + min(jug2, cap1 - jug1), jug2 - min(jug2, cap1
- jug1))]:
            if (next_jug1, next_jug2) not in visited and 0 <= next_jug1 <= cap1
and 0 <= next_jug2 <= cap2:
                visited.add((next_jug1, next_jug2))
                queue.append((next_jug1, next_jug2, path + [(jug1, jug2)]))

    print("No solution found.")
```

```
# Example usage
solve_water_jug_problem(4, 3, 2)
```

16)DECISION TREE:

```
import numpy as np


class TreeNode:
    def __init__(self, feature_index=None, threshold=None, left=None, right=None, value=None):
        self.feature_index = feature_index
        self.threshold = threshold
        self.left = left
        self.right = right
        self.value = value


class DecisionTreeClassifier:
    def __init__(self, max_depth=None):
        self.max_depth = max_depth
        self.tree = None


    def fit(self, X, y):
        self.tree = self._build_tree(X, y, depth=0)


    def _build_tree(self, X, y, depth):
        if len(y) == 0:
            return None
```

```
        if depth == self.max_depth or len(np.unique(y)) == 1:
            return TreeNode(value=np.bincount(y).argmax())

        best_split = self._find_best_split(X, y)
        if not best_split:
            return TreeNode(value=np.bincount(y).argmax())

        X_left, y_left, X_right, y_right = self._split_data(X, y,
best_split['feature_index'], best_split['threshold'])
        if len(y_left) == 0 or len(y_right) == 0:
            return TreeNode(value=np.bincount(y).argmax())

        left_subtree = self._build_tree(X_left, y_left, depth + 1)
        right_subtree = self._build_tree(X_right, y_right, depth + 1)
        return TreeNode(feature_index=best_split['feature_index'],
threshold=best_split['threshold'], left=left_subtree, right=right_subtree)

    def _find_best_split(self, X, y):
        best_split = {}
        best_gini = float('inf')

        for feature_index in range(X.shape[1]):
            thresholds = np.unique(X[:, feature_index])
            for threshold in thresholds:
                X_left, y_left, X_right, y_right = self._split_data(X, y, feature_index,
threshold)
                gini = self._gini_index(y_left, y_right)
```

```
                if gini < best_gini:
                    best_gini = gini
                    best_split = {'feature_index': feature_index, 'threshold': threshold}

        return best_split

    def _split_data(self, X, y, feature_index, threshold):
        left_mask = X[:, feature_index] <= threshold
        return X[left_mask], y[left_mask], X[~left_mask], y[~left_mask]

    def _gini_index(self, y_left, y_right):
        n_left, n_right = len(y_left), len(y_right)
        if n_left == 0 or n_right == 0:
            return 0
        gini_left = 1.0 - sum((np.sum(y_left == c) / n_left) ** 2 for c in np.unique(y_left))
        gini_right = 1.0 - sum((np.sum(y_right == c) / n_right) ** 2 for c in np.unique(y_right))
        return (n_left * gini_left + n_right * gini_right) / (n_left + n_right)

    def predict(self, X):
        return np.array([self._predict_sample(x, self.tree) for x in X])

    def _predict_sample(self, x, node):
        if node.value is not None:
            return node.value
        if x[node.feature_index] <= node.threshold:
```

```
            return self._predict_sample(x, node.left)
        else:
            return self._predict_sample(x, node.right)

def create_synthetic_data():
    np.random.seed(42)
    X = np.random.rand(100, 2)
    y = (X[:, 0] + X[:, 1] > 1).astype(int)
    return X, y

def train_test_split(X, y, test_size=0.2):
    indices = np.arange(X.shape[0])
    np.random.shuffle(indices)
    test_size = int(len(y) * test_size)
    train_indices, test_indices = indices[:-test_size], indices[-test_size:]
    return X[train_indices], X[test_indices], y[train_indices], y[test_indices]

if __name__ == "__main__":
    X, y = create_synthetic_data()
    X_train, X_test, y_train, y_test = train_test_split(X, y)

    model = DecisionTreeClassifier(max_depth=3)
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)

    accuracy = np.mean(y_pred == y_test)
```

```
    print(f"Accuracy: {accuracy}")
```

PROLOG

1)**SUM**

```
sum(0,0).
sum(N,Total):-
N>0,
N1 is N-1,
sum(N1,Result),
Total is N+Result.
```

OUT PUT:

```
%sum(3,Total).
Total = 6 .
```

2)**DOB**

```
person(seetha,"05-11-2005").
person(navya,"28-04-2004").
person(kavya,"05-09-2005").
get_dob(Name,DOB):-
person(Name,DOB).
get_name(DOB,Name):-
person(Name,DOB).
```

OUT PUT:

% get_name("05-11-2005", Name).

Name = seetha

%get_dob(seetha,DOB).

DOB = "05-11-2005".

3)**STUDENT-TEACHER**

studies(annya,ai).

studies(divya,ai).

studies(anjali,csbs).

studies(seetha,maths).

teaches(jk,ai).

teaches(jimin,csbs).

teaches(tae,maths).

teaches(jin,ai).

lecturer(Professor,Student):-

teaches(Professor,Course),

studies(Student,Course).

OUT PUT:

%

?- lecturer(jk,Student).

Student = annya .

%?- teaches(jimin,Student).

Student = csbs.

%?- teaches(jimin,Course).

Course = csbs.

%?- studies(annya,Course).

Course = ai.

%?- studies(seetha,Course).

Course = maths.

4)**PLANETS DB**

```
planet(jupiter,30000).
planet(earth,40000).
planet(satrun,7000).
planet(venus,8999).
planet(neptune,90000).

list_of(Planets,Distance):-
   findall(Planets,planet(Planets,_),Planets).
list_distance(Distance):-
   findall(Distance,planet(_,Distance),Planets).
list_distance(Distance):-
   findall(Distance,planet(_,Distance),Distance).
```

OUTPUT:

%planet(earth,Distance).

Distance = 40000.

%planet(Planet,8999).

Planet = venus.

5)**TOWERS OF HANOI:**

```
% Move a single disk from Source peg to Destination peg
move_disk(1, Source, Destination, _) :-
    write('Move disk 1 from '), write(Source), write(' to '), write(Destination), nl.

% Move N disks from Source peg to Destination peg using Auxiliary peg
move_disk(N, Source, Destination, Auxiliary) :-
    N > 1,
    N1 is N - 1,
    move_disk(N1, Source, Auxiliary, Destination),
    write('Move disk '), write(N), write(' from '), write(Source), write(' to '),
write(Destination), nl,
    move_disk(N1, Auxiliary, Destination, Source).

% Solve the Towers of Hanoi puzzle with N disks
towers_of_hanoi(N) :-
    move_disk(N, 'Source', 'Destination', 'Auxiliary').
```

OUTPUT:

```
towers_of_hanoi(3).
Move disk 1 from Source to Destination
Move disk 2 from Source to Auxiliary
Move disk 1 from Destination to Auxiliary
Move disk 3 from Source to Destination
Move disk 1 from Auxiliary to Source
Move disk 2 from Auxiliary to Destination
Move disk 1 from Source to Destination
True
```

6)**BIRD CAN FLY OR NOT:**

```
% Define birds and their ability to fly
bird(sparrow, fly).
bird(pigeon, fly).
bird(squirrel, cannotfly).
bird(tan, cannotfly).

% Predicate to check if a bird can fly
can_fly(Bird) :- bird(Bird, fly).

% Query to find all birds that can fly
fly_of(Birds) :- findall(Bird, can_fly(Bird), Birds).
```

OUTPUT:

```
can_fly(sparrow).
true.

?- fly_of(Birds).
Birds = [sparrow, pigeon].

?- bird(sparrow,Fly).
Fly = fly.

?- bird(sparrow,fly).
True
```

7)**FAMILY TREE:**

```
% Facts: parent relationships
parent(john, mary).
```

```prolog
parent(john, joe).
parent(susan, mary).
parent(susan, joe).
parent(mary, ann).
parent(mary, tom).
parent(david, ann).
parent(david, tom).

% Rules: defining relationships based on parent relationships

% X is a child of Y if Y is a parent of X
child(X, Y) :- parent(Y, X).

% X is a grandparent of Z if X is a parent of Y and Y is a parent of Z
grandparent(X, Z) :- parent(X, Y), parent(Y, Z).

% X is a grandchild of Z if Z is a grandparent of X
grandchild(X, Z) :- grandparent(Z, X).

% X is a sibling of Y if they share at least one parent
sibling(X, Y) :- parent(Z, X), parent(Z, Y), X \= Y.

% X is a cousin of Y if their parents are siblings
cousin(X, Y) :-
   parent(A, X),
   parent(B, Y),
```

```
    sibling(A, B).

% X is an aunt or uncle of Y if X is a sibling of Y's parent
aunt_or_uncle(X, Y) :-
    parent(Z, Y),
    sibling(X, Z).

% X is a niece or nephew of Y if Y is a sibling of X's parent
niece_or_nephew(X, Y) :-
    parent(Z, X),
    sibling(Z, Y).

% X is a descendant of Y if Y is a parent of X or Y is an ancestor of Z who is a parent of X
descendant(X, Y) :- parent(Y, X).
descendant(X, Y) :- parent(Y, Z), descendant(X, Z).

% X is an ancestor of Y if X is a parent of Y or X is an ancestor of Z who is a parent of Y
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

OUT PUT:

```
% Find all children of john
% ?- child(X, john).

% Find all siblings of mary
```

% ?- sibling(X, mary).

% Find all grandparents of ann

% ?- grandparent(X, ann).

% Find all cousins of tom

% ?- cousin(X, tom).

% Find all descendants of john

% ?- descendant(X, john).

% Find all ancestors of tom

% ?- ancestor(X, tom).

8) **DIETING SYSTEM BASED ON DIEASEASE:**

% Define diseases and their symptoms

dise(heartatack,"heart").

dise(fever,"hot").

dise(cold,"runn").

% Define dietary recommendations based on symptoms

diet(heart,"Avoid oil food").

diet(hot,"Avoid Cool food").

diet(runn,"Avoid cool food").

```
food(Dise,Diet):-
    dise(Dise,Fa),
    diet(Fa,Diet).
```

OUT PUT:

%Fa=symptom

%dise(fever,Fa).

Fa = "hot".

%?- diet(runn,Diet).

Diet = "Avoid cood food".

9)**MONKEY-BANANA:**

```
% Initial state
initial_state(on_ground).

% Actions and their effects
action(on_ground, push_box, box_under_banana).
action(box_under_banana, climb, on_box).
action(on_box, reach, banana_reached).

% Plan generation
plan(State, [], State).
plan(State1, [Action | Rest], Goal) :-
    action(State1, Action, State2),
    plan(State2, Rest, Goal).
```

OUTPUT:

```
% ?- initial_state(State), plan(State, Plan, banana_reached).
%State = on_ground,
%Plan = [push_box, climb, reach]
```

10)**FRUIT COLOR:**

```
% Define fruits and their colors
fruit_color(apple, red).
fruit_color(banana, yellow).
fruit_color(grape, purple).
fruit_color(orange, orange).
fruit_color(lemon, yellow).
fruit_color(cherry, red).

% Query to find fruits by color using backtracking
find_fruits_by_color(Color, Fruit) :-
   fruit_color(Fruit, Color).
```

OUT PUT:

```
% ?- find_fruits_by_color(yellow, Fruit).
% Fruit = banana ;
% Fruit = lemon.
%fruit_color(lemon,Color).
%Color = yellow.
%fruit_color(apple, red).
%true
```

11)**BFS:**

```
% Define the graph using edge facts
edge(a, b).
edge(a, c).
edge(b, d).
edge(b, e).
edge(c, f).
edge(d, g).
edge(e, g).
edge(f, g).

% BFS Algorithm
bfs(Start, Goal, Path) :-
   bfs_helper([[Start]], Goal, RevPath),
   reverse(RevPath, Path).

% Helper predicate for BFS
bfs_helper([[Goal|Rest] | _], Goal, [Goal|Rest]).
bfs_helper([[Node|Path] | Paths], Goal, Result) :-
   findall([NextNode, Node | Path],
         (edge(Node, NextNode), \+ member(NextNode, [Node | Path])),
         NewPaths),
   append(Paths, NewPaths, UpdatedPaths),
   bfs_helper(UpdatedPaths, Goal, Result).
```

OUT PUT:

% ?- bfs(a, g, Path).

% Path = [a, b, d, g] ;

% Path = [a, b, e, g] ;

% Path = [a, c, f, g] ;

12)**MEDICAL DIAGNOSIS:**

```
% Define diseases and their symptoms
dise(heart_attack, "heart").
dise(fever, "hot").
dise(cold, "runny_nose").
dise(diabetes, "sugar").
dise(allergy, "itchy").
dise(migraine, "headache").
dise(stomach_ache, "pain").
dise(arthritis, "joint_pain").

% Define dietary recommendations based on symptoms
diet("heart", "Avoid oily food").
diet("hot", "Avoid cool food").
diet("runny_nose", "Avoid cold food").
diet("sugar", "Avoid sugary food").
diet("itchy", "Avoid allergens").
diet("headache", "Stay hydrated and avoid caffeine").
diet("pain", "Avoid spicy food").
diet("joint_pain", "Avoid heavy lifting").

% Match disease to dietary recommendation
```

```
food(Dise, Diet) :-
   dise(Dise, Symptom),
   diet(Symptom, Diet).
```

OUT PUT:

```
% Query to get the dietary recommendation for 'arthritis'
% ?- food(arthritis, Diet).
% Diet = "Avoid heavy lifting".

% Query to get the symptom for 'diabetes'
% ?- dise(diabetes, Symptom).
% Symptom = "sugar".
% food(arthritis,"Avoid heavy lifting" ).
%true.
```

13)**FORWARD CHAINING:**

```
% Facts
likes(john, pizza).
likes(mary, sushi).

% Rule
eats_out(X) :- likes(X, _).

% Forward chaining process
forward_chaining :-
   % Collect all facts
   findall(X, likes(X, _), People),
```

```
    % Apply the rule to each person
    forall(member(Person, People), (eats_out(Person),
assert(fact(eats_out(Person))))).


% Query to list all facts
list_facts :-
    fact(Fact),
    write(Fact), nl,
    fail.
list_facts.
```

OUT PUT:

```
% forward_chaining, list_facts.
%eats_out(john)
%eats_out(mary)
%true.
```

14)**BACKWARD CHAINING:**

```
% Facts
likes(john, pizza).
likes(mary, sushi).


% Rule
eats_out(X) :- likes(X, _).


% Backward chaining process
% Check if the goal can be satisfied
backward_chaining(Goal) :-
    call(Goal), !. % Use call/1 to execute the goal
```

OUTPUT:

% Query to check if john eats out

% ?- backward_chaining(eats_out(john)).

% Query to check if mary eats out

% ?- backward_chaining(eats_out(mary)).