**UCS  
import heapq # Import heapq for priority queue (helps in picking the smallest cost first)**

**def uniform\_cost\_search(graph, start, goal):**

**priority\_queue = [(0, start)] # This is a list that acts as a priority queue (stores (cost, node))**

**visited = {} # Dictionary to store visited nodes with their cost**

**while priority\_queue: # Keep running until there are no more nodes to explore**

**cost, node = heapq.heappop(priority\_queue) # Get the node with the lowest cost**

**if node in visited: # If the node is already visited, skip it**

**continue**

**visited[node] = cost # Mark the node as visited with its cost**

**print(f"Visiting: {node} (Cost: {cost})") # Print the current node being visited**

**if node == goal: # If we reach the goal node, print the final cost and return**

**print(f"Reached {goal} with cost {cost}")**

**return cost # Return the final shortest cost**

**# Explore all the neighbors of the current node**

**for neighbor, weight in graph.get(node, []): # Get all neighbors and their costs**

**if neighbor not in visited: # If the neighbor is not visited**

**new\_cost = cost + weight # Calculate the new cost to reach this neighbor**

**heapq.heappush(priority\_queue, (new\_cost, neighbor)) # Add to priority queue**

**print(f" Queueing {neighbor} (Cost: {new\_cost})") # Print queued node and cost**

**print(f"{goal} is not reachable") # If the loop finishes and goal is not reached**

**return float('inf') # Return infinity (goal cannot be reached)**

**# Take user input to build the graph**

**graph = {} # Dictionary to store the graph**

**edges = int(input("Enter number of edges: ")) # Ask user how many edges in the graph**

**for \_ in range(edges): # Loop to take input for each edge**

**u, v, w = input("Enter edge (start, end, cost): ").split() # Take edge input**

**w = int(w) # Convert cost to integer**

**if u not in graph: # If the start node is not already in the graph, create an empty list**

**graph[u] = []**

**graph[u].append((v, w)) # Add the neighbor node with its cost**

**# Ask user for start and goal nodes**

**start = input("Enter start node: ")**

**goal = input("Enter goal node: ")**

**# Call the UCS function to find the shortest cost from start to goal**

**result = uniform\_cost\_search(graph, start, goal)**

**expected output**

**Enter number of edges: 10**

**Enter edge (start, end, cost): S p 1**

**Enter edge (start, end, cost): S d 3**

**Enter edge (start, end, cost): S e 9**

**Enter edge (start, end, cost): p q 16**

**Enter edge (start, end, cost): d c 11**

**Enter edge (start, end, cost): d e 5**

**Enter edge (start, end, cost): d b 4**

**Enter edge (start, end, cost): e h 13**

**Enter edge (start, end, cost): e r 7**

**Enter edge (start, end, cost): r f 8**

**Enter edge (start, end, cost): f G 2**

**Enter edge (start, end, cost): r G 3**

**Enter start node: S**

**Enter goal node: G  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
DFS Water JUG  
  
def water\_jug\_dfs(cap\_a, cap\_b, target):**

**stack = [(0, 0, [])] # (jug A, jug B, action sequence)**

**visited = set() # To keep track of visited states**

**while stack: # While there are states to explore**

**jug\_a, jug\_b, actions = stack.pop() # Get the last state (DFS uses stack)**

**if (jug\_a, jug\_b) in visited: # If already visited, skip**

**continue**

**visited.add((jug\_a, jug\_b)) # Mark as visited**

**actions.append((jug\_a, jug\_b)) # Add current state to action sequence**

**if jug\_a == target or jug\_b == target: # If we reach target, print sequence**

**print("Action Sequence:", actions)**

**return actions**

**# Possible actions**

**stack.append((cap\_a, jug\_b, actions[:])) # Fill Jug A**

**stack.append((jug\_a, cap\_b, actions[:])) # Fill Jug B**

**stack.append((0, jug\_b, actions[:])) # Empty Jug A**

**stack.append((jug\_a, 0, actions[:])) # Empty Jug B**

**# Pour A → B**

**pour = min(jug\_a, cap\_b - jug\_b)**

**stack.append((jug\_a - pour, jug\_b + pour, actions[:]))**

**# Pour B → A**

**pour = min(jug\_b, cap\_a - jug\_a)**

**stack.append((jug\_a + pour, jug\_b - pour, actions[:]))**

**print("No solution found")**

**return None**

**# Example Run**

**water\_jug\_dfs(4, 3, 2)  
  
  
  
OUTPUT  
Action Sequence: [(0, 0), (4, 0), (1, 3), (1, 0), (0, 1)]  
  
  
BFS Waterjug  
  
from collections import deque**

**def water\_jug\_bfs(cap\_a, cap\_b, target):**

**queue = deque([(0, 0, [])]) # BFS queue**

**visited = set() # Keep track of visited states**

**while queue:**

**jug\_a, jug\_b, actions = queue.popleft() # Get the first state**

**if (jug\_a, jug\_b) in visited: # Skip if already visited**

**continue**

**visited.add((jug\_a, jug\_b))**

**actions.append((jug\_a, jug\_b)) # Store action sequence**

**if jug\_a == target or jug\_b == target: # If goal reached, print sequence**

**print("Action Sequence:", actions)**

**return actions**

**# Possible actions**

**queue.append((cap\_a, jug\_b, actions[:])) # Fill Jug A**

**queue.append((jug\_a, cap\_b, actions[:])) # Fill Jug B**

**queue.append((0, jug\_b, actions[:])) # Empty Jug A**

**queue.append((jug\_a, 0, actions[:])) # Empty Jug B**

**# Pour A → B**

**pour = min(jug\_a, cap\_b - jug\_b)**

**queue.append((jug\_a - pour, jug\_b + pour, actions[:]))**

**# Pour B → A**

**pour = min(jug\_b, cap\_a - jug\_a)**

**queue.append((jug\_a + pour, jug\_b - pour, actions[:]))**

**print("No solution found")**

**return None**

**# Example Run**

**water\_jug\_bfs(4, 3, 2)  
  
  
output  
Action Sequence: [(0, 0), (4, 0), (1, 3), (1, 0), (0, 1)**

**UCS WATERJUG  
  
import heapq**

**def water\_jug\_ucs(cap\_a, cap\_b, target):**

**priority\_queue = [(0, 0, 0, [])] # (cost, jug A, jug B, action sequence)**

**visited = set()**

**while priority\_queue:**

**cost, jug\_a, jug\_b, actions = heapq.heappop(priority\_queue) # Get lowest cost state**

**if (jug\_a, jug\_b) in visited: # Skip if already visited**

**continue**

**visited.add((jug\_a, jug\_b))**

**actions.append((jug\_a, jug\_b)) # Store action sequence**

**if jug\_a == target or jug\_b == target: # If goal reached, print sequence**

**print("Action Sequence:", actions)**

**return actions**

**# Possible actions with cost**

**heapq.heappush(priority\_queue, (cost + 1, cap\_a, jug\_b, actions[:])) # Fill Jug A**

**heapq.heappush(priority\_queue, (cost + 1, jug\_a, cap\_b, actions[:])) # Fill Jug B**

**heapq.heappush(priority\_queue, (cost + 1, 0, jug\_b, actions[:])) # Empty Jug A**

**heapq.heappush(priority\_queue, (cost + 1, jug\_a, 0, actions[:])) # Empty Jug B**

**# Pour A → B**

**pour = min(jug\_a, cap\_b - jug\_b)**

**heapq.heappush(priority\_queue, (cost + 1, jug\_a - pour, jug\_b + pour, actions[:]))**

**# Pour B → A**

**pour = min(jug\_b, cap\_a - jug\_a)**

**heapq.heappush(priority\_queue, (cost + 1, jug\_a + pour, jug\_b - pour, actions[:]))**

**print("No solution found")**

**return None**

**# Example Run**

**water\_jug\_ucs(4, 3, 2)  
  
  
OUTPUT  
Action Sequence: [(0, 0), (4, 0), (1, 3), (1, 0), (0, 1)]  
  
BFS DFS UCS A\* FOR GRAPH SEARCH  
  
import heapq**

**from collections import deque**

**# Function to take user input for the graph**

**def get\_graph():**

**graph = {}**

**print("Enter graph (Format: Node: Neighbors, type STOP to finish):")**

**while True:**

**entry = input().strip()**

**if entry.upper() == "STOP":**

**break**

**node, neighbors = entry.split(":")**

**graph[node.strip()] = [n.strip() for n in neighbors.split(",") if n.strip()]**

**return graph**

**# Function to take user input for edge costs**

**def get\_costs():**

**costs = {}**

**print("Enter edge costs (Format: Node1,Node2: Cost, type STOP to finish):")**

**while True:**

**entry = input().strip()**

**if entry.upper() == "STOP":**

**break**

**edge, cost = entry.split(":")**

**node1, node2 = edge.split(",")**

**costs[(node1.strip(), node2.strip())] = int(cost.strip())**

**return costs**

**# Function to take user input for heuristics**

**def get\_heuristics():**

**heuristics = {}**

**print("Enter heuristic values (Format: Node: Value, type STOP to finish):")**

**while True:**

**entry = input().strip()**

**if entry.upper() == "STOP":**

**break**

**node, value = entry.split(":")**

**heuristics[node.strip()] = int(value.strip())**

**return heuristics**

**# Depth-First Search (DFS)**

**def dfs(graph, start, goal, visited=None, path=[]):**

**if visited is None:**

**visited = set()**

**visited.add(start)**

**path.append(start)**

**if start == goal:**

**return path**

**for neighbor in graph.get(start, []):**

**if neighbor not in visited:**

**result = dfs(graph, neighbor, goal, visited, path[:])**

**if result:**

**return result**

**return None**

**# Breadth-First Search (BFS)**

**def bfs(graph, start, goal):**

**queue = deque([(start, [])])**

**visited = set()**

**while queue:**

**node, path = queue.popleft()**

**path.append(node)**

**if node == goal:**

**return path**

**if node not in visited:**

**visited.add(node)**

**for neighbor in graph.get(node, []):**

**queue.append((neighbor, path[:]))**

**return None**

**# Uniform Cost Search (UCS)**

**def ucs(graph, costs, start, goal):**

**pq = [(0, start, [])] # (cost, node, path)**

**visited = set()**

**while pq:**

**cost, node, path = heapq.heappop(pq)**

**path.append(node)**

**if node == goal:**

**return path**

**if node not in visited:**

**visited.add(node)**

**for neighbor in graph.get(node, []):**

**new\_cost = cost + costs.get((node, neighbor), 1)**

**heapq.heappush(pq, (new\_cost, neighbor, path[:]))**

**return None**

**# A\* Search**

**def a\_star(graph, costs, heuristics, start, goal):**

**pq = [(heuristics.get(start, 0), 0, start, [])] # (f(n), g(n), node, path)**

**visited = set()**

**while pq:**

**\_, path\_cost, node, path = heapq.heappop(pq)**

**path.append(node)**

**if node == goal:**

**return path**

**if node not in visited:**

**visited.add(node)**

**for neighbor in graph.get(node, []):**

**new\_cost = path\_cost + costs.get((node, neighbor), 1)**

**heapq.heappush(pq, (new\_cost + heuristics.get(neighbor, 0), new\_cost, neighbor, path[:]))**

**return None**

**# Taking user inputs**

**graph = get\_graph()**

**costs = get\_costs()**

**heuristics = get\_heuristics()**

**start = input("Enter start node: ").strip()**

**goal = input("Enter goal node: ").strip()**

**# Running all search algorithms**

**dfs\_path = dfs(graph, start, goal)**

**bfs\_path = bfs(graph, start, goal)**

**ucs\_path = ucs(graph, costs, start, goal)**

**a\_star\_path = a\_star(graph, costs, heuristics, start, goal)**

**# Output results**

**print("\n📌 \*\*Paths Found by Each Algorithm\*\*")**

**print(f"DFS Path: {dfs\_path}")**

**print(f"BFS Path: {bfs\_path}")**

**print(f"UCS Path: {ucs\_path}")**

**print(f"A\* Path: {a\_star\_path}")  
  
OUTPUT  
  
Enter graph (Format: Node: Neighbors, type STOP to finish):**

**A: B,C**

**B: D,E**

**C: F**

**D:**

**E: F**

**F:**

**STOP**

**Enter edge costs (Format: Node1,Node2: Cost, type STOP to finish):**

**A,B: 1**

**A,C: 2**

**B,D: 3**

**B,E: 4**

**C,F: 5**

**E,F: 6**

**STOP**

**Enter heuristic values (Format: Node: Value, type STOP to finish):**

**A: 6**

**B: 4**

**C: 5**

**D: 3**

**E: 2**

**F: 0**

**STOP**

**Enter start node: A**

**Enter goal node: F  
  
  
  
📌 \*\*Paths Found by Each Algorithm\*\***

**DFS Path: ['A', 'B', 'D', 'E', 'F']**

**BFS Path: ['A', 'B', 'E', 'F']**

**UCS Path: ['A', 'C', 'F']**

**A\* Path: ['A', 'C', 'F']  
  
  
  
  
  
  
  
ROBO PATH PLANNING  
import heapq**

**# 📌 Grid (0 = free space, 1 = obstacle)**

**grid = [**

**[0, 0, 0, 1, 0],**

**[0, 1, 0, 1, 0],**

**[0, 1, 0, 0, 0],**

**[0, 0, 0, 1, 1],**

**[0, 1, 0, 0, 0]**

**]**

**# 📌 Get user input**

**start = tuple(map(int, input("Enter Start (row col): ").split()))**

**goal = tuple(map(int, input("Enter Goal (row col): ").split()))**

**# 📌 A\* Algorithm**

**def a\_star(grid, start, goal):**

**pq = [(0, start, [])] # (cost, position, path)**

**moves = [(0, 1), (1, 0), (0, -1), (-1, 0)] # Right, Down, Left, Up**

**visited = set()**

**while pq:**

**cost, current, path = heapq.heappop(pq)**

**if current == goal:**

**return path + [current]**

**if current in visited:**

**continue**

**visited.add(current)**

**for move in moves:**

**new\_pos = (current[0] + move[0], current[1] + move[1])**

**if 0 <= new\_pos[0] < len(grid) and 0 <= new\_pos[1] < len(grid[0]) and grid[new\_pos[0]][new\_pos[1]] == 0:**

**heapq.heappush(pq, (cost + 1 + abs(new\_pos[0] - goal[0]) + abs(new\_pos[1] - goal[1]), new\_pos, path + [current]))**

**# 📌 Run A\* and Print Path**

**path = a\_star(grid, start, goal)**

**print("\n📌 Path:", path if path else "No path found!")  
  
  
OUTPUT  
Enter Start (row col): 0 0**

**Enter Goal (row col): 4 4**

**📌 Path: [(0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (2, 3), (2, 4), (3, 4), (4, 4)  
  
  
  
PATH PLANNING 2  
  
# 📌 Get user input for start and goal positions**

**x1, y1 = input("Enter Start (row col): ").split()**

**x2, y2 = input("Enter Goal (row col): ").split()**

**# 📌 Convert input to integers**

**x1, y1, x2, y2 = int(x1), int(y1), int(x2), int(y2)**

**# 📌 Chebyshev Distance: max(|x1 - x2|, |y1 - y2|)**

**chebyshev\_distance = max(abs(x1 - x2), abs(y1 - y2))**

**# 📌 Manhattan Distance: |x1 - x2| + |y1 - y2|**

**manhattan\_distance = abs(x1 - x2) + abs(y1 - y2)**

**# 📌 Print results**

**print("\n📌 Chebyshev Distance:", chebyshev\_distance)**

**print("📌 Manhattan Distance:", manhattan\_distance)  
  
output  
Enter Start (row col): 1 2**

**Enter Goal (row col): 4 5**

**📌 Chebyshev Distance: 3**

**📌 Manhattan Distance: 6  
  
  
  
Enter Start (row col): 0 0**

**Enter Goal (row col): 4 4**

**📌 Chebyshev Distance: 4**

**📌 Manhattan Distance: 8  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
TRAVELLING SALES MAN PROBLEM  
  
import random # Import random module for shuffling the initial path**

**# Function to get user input for the number of cities and their distances**

**def get\_user\_input():**

**cities = int(input("Enter the number of cities: ")) # Take number of cities as input**

**dist = {} # Dictionary to store distances between cities**

**print("Enter distances between cities (Format: city1 city2 distance). Enter 'done' to finish:")**

**while True:**

**entry = input().strip() # Read user input**

**if entry.lower() == "done": # Stop when user types "done"**

**break**

**city1, city2, distance = map(int, entry.split()) # Convert input to integers**

**dist[(city1, city2)] = distance # Store distance**

**dist[(city2, city1)] = distance # Since it's an undirected graph, add reverse path too**

**return list(range(cities)), dist # Return city list and distance dictionary**

**# Function to calculate total distance of a given route**

**def cost(route, dist):**

**total\_cost = 0**

**for i in range(len(route) - 1): # Iterate through the route**

**total\_cost += dist.get((route[i], route[i+1]), float('inf')) # Get distance, default to infinity if missing**

**total\_cost += dist.get((route[-1], route[0]), float('inf')) # Add return-to-start distance**

**return total\_cost**

**# Function to perform a 2-opt swap by reversing part of the route**

**def two\_opt(route, i, j):**

**return route[:i] + route[i:j+1][::-1] + route[j+1:] # Reverse part of the route**

**# Hill Climbing Algorithm for TSP with 2-opt optimization**

**def tsp\_hill\_climbing(cities, dist):**

**random.shuffle(cities) # Start with a random tour**

**best\_cost = cost(cities, dist) # Compute cost of initial route**

**while True: # Keep improving until no better route is found**

**improved = False**

**for i in range(len(cities) - 1): # Iterate through cities**

**for j in range(i + 1, len(cities)): # Pick another city to swap**

**new\_route = two\_opt(cities, i, j) # Apply 2-opt swap**

**new\_cost = cost(new\_route, dist) # Compute new route cost**

**if new\_cost < best\_cost: # If new route is better**

**cities, best\_cost = new\_route, new\_cost # Update best route**

**improved = True # Mark improvement**

**if not improved: # Stop if no improvement**

**break**

**return cities, best\_cost # Return the best route and cost**

**# Get user input for cities and distances**

**cities, dist = get\_user\_input()**

**# Run Hill Climbing for TSP**

**best\_path, best\_distance = tsp\_hill\_climbing(cities, dist)**

**# Print the best-found route and total distance**

**print("\nBest Route:", best\_path)**

**print("Total Distance:", best\_distance)  
  
  
  
OUTPUT :  
  
Enter the number of cities: 4**

**Enter distances between cities (Format: city1 city2 distance). Enter 'done' to finish:**

**0 1 20**

**0 2 10**

**0 3 15**

**1 2 15**

**1 3 11**

**2 3 17**

**Done  
  
Best Route: [0, 2, 3, 1]**

**Total Distance: 41  
  
  
  
  
  
  
  
  
MODEL CHECKING PROPOSITIONAL LOGIC  
  
from itertools import product # Import product to generate all truth assignments**

**# Function to evaluate a propositional logic expression with a given truth assignment**

**def evaluate\_expression(expression, model):**

**for symbol, value in model.items():**

**expression = expression.replace(symbol, str(value)) # Replace symbols (A, B, C) with True/False values**

**# Replace logical operators with Python syntax**

**expression = expression.replace("¬", "not ") \**

**.replace("∧", " and ") \**

**.replace("∨", " or ") \**

**.replace("⇒", " <= ") \**

**.replace("⇔", " == ")**

**return eval(expression) # Evaluate the final expression**

**# Function to find a satisfying assignment (model) for the KB**

**def find\_model(KB):**

**symbols = sorted(set("".join(KB).replace("¬", "").replace("∧", "").replace("∨", "").replace("⇒", "").replace("⇔", "").replace(" ", ""))) # Extract unique symbols**

**for values in product([True, False], repeat=len(symbols)): # Generate all possible truth assignments**

**model = dict(zip(symbols, values)) # Create a model (dictionary of symbol assignments)**

**if all(evaluate\_expression(statement, model) for statement in KB): # Check if all KB statements are satisfied**

**return model # Return a satisfying model**

**return None # No model found**

**# 📌 Test Case 1**

**KB1 = [**

**"(A ∨ B)",**

**"(¬A ⇔ ¬B ∨ C)",**

**"(¬A ∨ ¬B ∨ C)"**

**]**

**# 📌 Test Case 2**

**KB2 = [**

**"(C ⇔ B ∨ D)",**

**"(A ⇒ ¬B ∧ ¬D)",**

**"(¬(B ∧ ¬C) ⇒ A)",**

**"(¬D ⇒ C)"**

**]**

**# 📌 Run the function for both test cases**

**print("\n📌 \*\*Test Case 1:\*\*")**

**model1 = find\_model(KB1)**

**if model1:**

**print("✅ Satisfying Model Found:", model1)**

**else:**

**print("❌ No satisfying model found!")**

**print("\n📌 \*\*Test Case 2:\*\*")**

**model2 = find\_model(KB2)**

**if model2:**

**print("✅ Satisfying Model Found:", model2)**

**else:**

**print("❌ No satisfying model found!")  
  
  
OUTPUT  
📌 \*\*Test Case 1:\*\***

**✅ Satisfying Model Found: {'A': False, 'B': True, 'C': True}**

**📌 \*\*Test Case 2:\*\***

**✅ Satisfying Model Found: {'A': True, 'B': False, 'C': True, 'D': False}  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
DPLL  
  
def dpll(clauses, assignments):**

**# If all clauses are satisfied**

**if not clauses:**

**return True, assignments**

**# If any clause is empty => unsatisfiable**

**if [] in clauses:**

**return False, {}**

**# Unit clause propagation**

**for clause in clauses:**

**if len(clause) == 1:**

**literal = clause[0]**

**return dpll(assign\_literal(clauses, literal), assignments + [literal])**

**# Choose first literal from first clause**

**literal = clauses[0][0]**

**# Try assigning it True**

**sat, result = dpll(assign\_literal(clauses, literal), assignments + [literal])**

**if sat:**

**return True, result**

**# Try assigning it False**

**sat, result = dpll(assign\_literal(clauses, negate(literal)), assignments + [negate(literal)])**

**return sat, result**

**def assign\_literal(clauses, literal):**

**new\_clauses = []**

**for clause in clauses:**

**if literal in clause:**

**continue # Clause is satisfied**

**new\_clause = [l for l in clause if l != negate(literal)]**

**new\_clauses.append(new\_clause)**

**return new\_clauses**

**def negate(literal):**

**return literal[1:] if literal.startswith('-') else '-' + literal**

**# Example input: (A ∨ ¬B) ∧ (B ∨ C) ∧ (¬A ∨ ¬C)**

**cnf = [['A', '-B'], ['B', 'C'], ['-A', '-C']]**

**satisfiable, model = dpll(cnf, [])**

**if satisfiable:**

**print("SATISFIABLE. One possible model:", model)**

**else:**

**print("UNSATISFIABLE.")  
  
  
  
OUTPUT  
  
SATISFIABLE. One possible model: ['A', 'B', '-C']  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
KB FORWARD AND BACKWARD CHAINING   
  
  
# Initial knowledge base**

**objects\_in\_room = {**

**"table": "center",**

**"chair": "left of table",**

**"bookshelf": "right of table",**

**"bed": "corner1",**

**"door": "wall1",**

**"window1": "wall2",**

**"window2": "wall3"**

**}**

**walls = {**

**"wall1": ["door"],**

**"wall2": ["window1"],**

**"wall3": ["window2"],**

**"wall4": []**

**}**

**corners = {**

**"corner1": ["bed"],**

**"corner2": [],**

**"corner3": [],**

**"corner4": []**

**}**

**# Forward chaining to infer reverse "left/right" facts**

**def infer\_reverse\_positions(kb):**

**inferred = {}**

**for obj, pos in kb.items():**

**if "left of" in pos:**

**ref = pos.split("left of")[1].strip()**

**inferred[ref] = f"right of {obj}"**

**elif "right of" in pos:**

**ref = pos.split("right of")[1].strip()**

**inferred[ref] = f"left of {obj}"**

**kb.update(inferred)**

**return kb**

**# Query functions**

**def furniture\_in\_room(kb):**

**return list(kb.keys())**

**def count\_windows(kb):**

**return len([obj for obj in kb if "window" in obj])**

**def location\_of(obj, kb):**

**return kb.get(obj, "Unknown")**

**def left\_of(item, kb):**

**for obj, pos in kb.items():**

**if pos == f"left of {item}":**

**return obj**

**return "None"**

**def right\_of(item, kb):**

**for obj, pos in kb.items():**

**if pos == f"right of {item}":**

**return obj**

**return "None"**

**def at\_wall(wall\_id):**

**return walls.get(wall\_id, [])**

**def at\_corner(corner\_id):**

**return corners.get(corner\_id, [])**

**# Apply forward chaining**

**full\_kb = infer\_reverse\_positions(objects\_in\_room.copy())**

**# Print the answers**

**print("Furniture in the room:", furniture\_in\_room(full\_kb))**

**print("Number of windows:", count\_windows(full\_kb))**

**print("Where is the table?:", location\_of("table", full\_kb))**

**print("What is to the left of the table?:", left\_of("table", full\_kb))**

**print("What is to the right of the table?:", right\_of("table", full\_kb))**

**print("What is at wall2?:", at\_wall("wall2"))**

**print("What is in corner1?:", at\_corner("corner1"))  
  
  
OUTPUT   
Furniture in the room: ['table', 'chair', 'bookshelf', 'bed', 'door', 'window1', 'window2']**

**Number of windows: 2**

**Where is the table?: center**

**What is to the left of the table?: chair**

**What is to the right of the table?: bookshelf**

**What is at wall2?: ['window1']**

**What is in corner1?: ['bed']**

**CLASSICAL PLANNING SIMPLE MONKEY AND BANANA  
  
# Initial state**

**monkey\_pos = "A"**

**box\_pos = "B"**

**bananas\_pos = "C"**

**monkey\_on\_box = False**

**monkey\_height = "Low"**

**box\_height = "Low"**

**bananas\_height = "High"**

**holding = None**

**# STEP 1: Move monkey from A to B**

**if monkey\_pos != box\_pos:**

**print(f"Action: move from {monkey\_pos} to {box\_pos}")**

**monkey\_pos = box\_pos**

**# STEP 2: Push box from B to C**

**if monkey\_pos == box\_pos:**

**print(f"Action: push box from {box\_pos} to {bananas\_pos}")**

**box\_pos = bananas\_pos**

**monkey\_pos = bananas\_pos**

**# STEP 3: Climb up box**

**if monkey\_pos == box\_pos:**

**print("Action: climb")**

**monkey\_on\_box = True**

**monkey\_height = "High"**

**# STEP 4: Grasp bananas**

**if monkey\_pos == bananas\_pos and monkey\_height == bananas\_height:**

**print("Action: grab")**

**holding = "bananas"**

**# STEP 5: Climb down**

**if monkey\_on\_box:**

**print("Action: down")**

**monkey\_on\_box = False**

**monkey\_height = "Low"**

**# STEP 6: Push box back to original position B**

**if monkey\_pos == box\_pos:**

**print(f"Action: push box from {box\_pos} to B")**

**box\_pos = "B"**

**monkey\_pos = "B"**

**OUTPUT  
  
Action: move from A to B**

**Action: push box from B to C**

**Action: climb**

**Action: grab**

**Action: down**

**Action: push box from C to B  
  
  
CLASSICAL PLANNING TOWER HENOI   
  
  
def hanoi(n, source, target, auxiliary, plan, depth=1):**

**if n == 1:**

**plan.append({**

**"step": len(plan) + 1,**

**"disk": 1,**

**"from": source,**

**"to": target,**

**"preconditions": f"Top disk on {source} is 1, {target} is empty or top disk > 1",**

**})**

**else:**

**hanoi(n-1, source, auxiliary, target, plan, depth+1)**

**plan.append({**

**"step": len(plan) + 1,**

**"disk": n,**

**"from": source,**

**"to": target,**

**"preconditions": f"Top disk on {source} is {n}, {target} is empty or top disk > {n}",**

**})**

**hanoi(n-1, auxiliary, target, source, plan, depth+1)**

**def print\_plan(plan, n):**

**print(f"\nTowers of Hanoi plan for {n} disks:\n")**

**for action in plan:**

**print(f"{action['step']}. Move disk {action['disk']} from {action['from']} to {action['to']}")**

**print(f" - Express State : Disk {action['disk']} on {action['from']}")**

**print(f" - Preconditions : {action['preconditions']}")**

**print(f" - Action : move({action['disk']}, {action['from']}, {action['to']})")**

**print(f" - Effects : Disk {action['disk']} is on {action['to']}, removed from {action['from']}\n")**

**if \_\_name\_\_ == "\_\_main\_\_":**

**n = int(input("Enter the value of n: "))**

**plan = []**

**hanoi(n, 'A', 'C', 'B', plan)**

**print\_plan(plan, n)**

**OUTPUT   
Towers of Hanoi plan for 3 disks:**

**1. Move disk 1 from A to C**

**- Express State : Disk 1 on A**

**- Preconditions : Top disk on A is 1, C is empty or top disk > 1**

**- Action : move(1, A, C)**

**- Effects : Disk 1 is on C, removed from A**

**2. Move disk 2 from A to B**

**- Express State : Disk 2 on A**

**- Preconditions : Top disk on A is 2, B is empty or top disk > 2**

**- Action : move(2, A, B)**

**- Effects : Disk 2 is on B, removed from A**

**3. Move disk 1 from C to B**

**MISSIONARIES AND CANNIBALS PROBLEM   
  
from collections import deque**

**# Initial and goal states**

**start\_state = (3, 3, 1) # (missionaries, cannibals, boat on left: 1=left, 0=right)**

**goal\_state = (0, 0, 0)**

**# Valid moves for the boat (<= 2 people total)**

**moves = [(1,0), (2,0), (0,1), (0,2), (1,1)]**

**def is\_valid(state):**

**m, c, \_ = state**

**m\_right = 3 - m**

**c\_right = 3 - c**

**# No one should be negative or more than 3**

**if m < 0 or c < 0 or m > 3 or c > 3:**

**return False**

**# Missionaries should not be outnumbered**

**if (m > 0 and m < c) or (m\_right > 0 and m\_right < c\_right):**

**return False**

**return True**

**def bfs():**

**queue = deque()**

**queue.append((start\_state, [], [])) # (state, path, description\_path)**

**visited = set()**

**while queue:**

**state, path, desc = queue.popleft()**

**if state == goal\_state:**

**return desc**

**if state in visited:**

**continue**

**visited.add(state)**

**m, c, boat = state**

**for move\_m, move\_c in moves:**

**if boat == 1:**

**new\_state = (m - move\_m, c - move\_c, 0)**

**dir\_arrow = "→"**

**else:**

**new\_state = (m + move\_m, c + move\_c, 1)**

**dir\_arrow = "←"**

**if is\_valid(new\_state):**

**step\_desc = f"{len(desc)+1}. Crossing {len(desc)+1}: {move\_m}M, {move\_c}C {dir\_arrow}"**

**left\_bank = f"{new\_state[0]}M, {new\_state[1]}C"**

**right\_bank = f"{3 - new\_state[0]}M, {3 - new\_state[1]}C"**

**state\_desc = f" Left bank: {left\_bank} | Right bank: {right\_bank}"**

**queue.append((new\_state, path + [state], desc + [step\_desc + '\n' + state\_desc]))**

**return None**

**# Main**

**solution = bfs()**

**if solution:**

**print("Solution found in", len(solution), "crossings:\n")**

**for line in solution:**

**print(line, "\n")**

**else:**

**print("No solution found.")  
  
  
OUTPUT**

**Solution found in 11 crossings:**

**1. Crossing 1: 0M, 2C →**

**Left bank: 3M, 1C | Right bank: 0M, 2C**

**2. Crossing 2: 0M, 1C ←**

**Left bank: 3M, 2C | Right bank: 0M, 1C**

**3. Crossing 3: 0M, 2C →**

**Left bank: 3M, 0C | Right bank: 0M, 3C**

**... and so on till all are safely moved.**