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
```

```python
        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
```

```python
# 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.
```