1.JUG wala

```
def water_jug_dfs(jug1_capacity, jug2_capacity, target):
  # Define the initial state and the goal state
  initial_state = (0, 0)
  goal_state = target

  stack = [initial_state]
  visited = set()
  path = []

  while stack:
    current_state = stack.pop()
    x, y = current_state

    if current_state == goal_state:
      path.append(current_state)
      return path

    if current_state in visited:
      continue

    visited.add(current_state)
    path.append(current_state)

    next_states = []
```

```
        next_states.append((jug1_capacity, y))



        next_states.append((x, jug2_capacity))

        next_states.append((0, y))

        next_states.append((x, 0))

        pour_to_jug2 = min(x, jug2_capacity - y)
        next_states.append((x - pour_to_jug2, y + pour_to_jug2))



        pour_to_jug1 = min(y, jug1_capacity - x)
        next_states.append((x + pour_to_jug1, y - pour_to_jug1))

        for state in next_states:
            if state not in visited:
                stack.append(state)
    return None

jug1_capacity = 4
jug2_capacity = 3
target = (2, 0)

solution_path = water_jug_dfs(jug1_capacity, jug2_capacity, target)
```

```
if solution_path:
    print("Solution path found:")
    for state in solution_path:
        print(state)
else:
    print("No solution found.")
```

2.MC

```
import heapq

# Define the initial, goal, and possible states
INITIAL_STATE = (3, 3, 1)
GOAL_STATE = (0, 0, 0)

class State:
    def __init__(self, missionaries, cannibals, boat):
        self.missionaries = missionaries
        self.cannibals = cannibals
        self.boat = boat
        self.parent = None

    def is_valid(self):
```

```
        if self.missionaries < 0 or self.cannibals < 0 or self.missionaries > 3 or self.cannibals > 3:
            return False
        if self.missionaries > 0 and self.missionaries < self.cannibals:
            return False
        if (3 - self.missionaries) > 0 and (3 - self.missionaries) < (3 - self.cannibals):
            return False
        return True

    def is_goal(self):
        return self.missionaries == GOAL_STATE[0] and self.cannibals == GOAL_STATE[1] and self.boat == GOAL_STATE[2]

    def __eq__(self, other):
        return self.missionaries == other.missionaries and self.cannibals == other.cannibals and self.boat == other.boat

    def __lt__(self, other):
        return (self.missionaries + self.cannibals) < (other.missionaries + other.cannibals)

    def __hash__(self):
        return hash((self.missionaries, self.cannibals, self.boat))

def best_first_search():
    initial_state = State(*INITIAL_STATE)
    frontier = []
    heapq.heappush(frontier, (0, initial_state))
```

```
    explored = set()

    while frontier:
        _, state = heapq.heappop(frontier)

        if state.is_goal():
            return state

        explored.add(state)

        successors = generate_successors(state)
        for successor in successors:
            if successor not in explored and successor not in [f[1] for f in frontier]:
                heapq.heappush(frontier, (successor.missionaries + successor.cannibals,
successor))

    return None

def generate_successors(state):
    successors = []
    moves = [(1, 0), (2, 0), (1, 1), (0, 1), (0, 2)]

    for move in moves:
        if state.boat == 1:
            new_state = State(state.missionaries - move[0], state.cannibals - move[1], 0)
        else:
```

```
            new_state = State(state.missionaries + move[0], state.cannibals + move[1], 1)

        if new_state.is_valid():
            new_state.parent = state
            successors.append(new_state)

    return successors

def print_solution(state):
    path = []
    while state:
        path.append(state)
        state = state.parent

    for s in reversed(path):
        print(f"({s.missionaries}, {s.cannibals}, {s.boat})")

def main():
    solution = best_first_search()
    if solution:
        print("Solution found!")
        print_solution(solution)
    else:
        print("No solution found.")

if __name__ == "__main__":
```

```python
    main()
```

3. A*

```python
import heapq


def heuristic(state, goal_state):
    return abs(state[0] - goal_state[0]) + abs(state[1] - goal_state[1])


def water_jug_a_star(jug1_capacity, jug2_capacity, target):

    initial_state = (0, 0)
    goal_state = target

    priority_queue = [(0, initial_state)]
    heapq.heapify(priority_queue)

    cost_so_far = {initial_state: 0}
    came_from = {initial_state: None}

    while priority_queue:
        _, current_state = heapq.heappop(priority_queue)
        x, y = current_state

        if current_state == goal_state:
```

```
    path = []
    while current_state is not None:
        path.append(current_state)
        current_state = came_from[current_state]
    path.reverse()
    return path

next_states = []

next_states.append((jug1_capacity, y))

next_states.append((x, jug2_capacity))

next_states.append((0, y))

next_states.append((x, 0))

pour_to_jug2 = min(x, jug2_capacity - y)
next_states.append((x - pour_to_jug2, y + pour_to_jug2))

pour_to_jug1 = min(y, jug1_capacity - x)
next_states.append((x + pour_to_jug1, y - pour_to_jug1))

for next_state in next_states:
    new_cost = cost_so_far[current_state] + 1
    if next_state not in cost_so_far or new_cost < cost_so_far[next_state]:
```

```
            cost_so_far[next_state] = new_cost
            priority = new_cost + heuristic(next_state, goal_state)
            heapq.heappush(priority_queue, (priority, next_state))
            came_from[next_state] = current_state

    return None

jug1_capacity = 4
jug2_capacity = 3
target = (2, 0)

solution_path = water_jug_a_star(jug1_capacity, jug2_capacity, target)

if solution_path:
    print("Solution path found:")
    for state in solution_path:
        print(state)
else:
    print("No solution found.")
```

4. AO*

```
import heapq

class AONode:
```

```python
    def __init__(self, state, heuristic=0):
        self.state = state
        self.heuristic = heuristic
        self.cost = float('inf')
        self.parent = None
        self.children = []
        self.solved = False
        self.best_child = None

    def add_child(self, child_node, cost=1):
        self.children.append((child_node, cost))

    def __lt__(self, other):
        return self.heuristic < other.heuristic

def heuristic(state, goal_state):
    return abs(state[0] - goal_state[0]) + abs(state[1] - goal_state[1])

def ao_star_search(initial_state, goal_state, jug1_capacity, jug2_capacity):
    # Create the root node
    root = AONode(initial_state, heuristic(initial_state, goal_state))

    # Priority queue for AO* (min-heap)
    priority_queue = [(root.heuristic, root)]
    heapq.heapify(priority_queue)
```

```
# While there are nodes to process
while priority_queue:
    _, current_node = heapq.heappop(priority_queue)

    # If we've reached the goal, reconstruct and return the path
    if current_node.state == goal_state:
        path = []
        while current_node is not None:
            path.append(current_node.state)
            current_node = current_node.parent
        path.reverse()
        return path

    # Generate all possible next states
    next_states = []

    # Fill Jug 1
    next_states.append((jug1_capacity, current_node.state[1]))

    # Fill Jug 2
    next_states.append((current_node.state[0], jug2_capacity))

    # Empty Jug 1
    next_states.append((0, current_node.state[1]))

    # Empty Jug 2
```

```
        next_states.append((current_node.state[0], 0))

        # Pour from Jug 1 to Jug 2
        pour_to_jug2 = min(current_node.state[0], jug2_capacity - current_node.state[1])
        next_states.append((current_node.state[0] - pour_to_jug2, current_node.state[1] +
pour_to_jug2))

        # Pour from Jug 2 to Jug 1
        pour_to_jug1 = min(current_node.state[1], jug1_capacity - current_node.state[0])
        next_states.append((current_node.state[0] + pour_to_jug1, current_node.state[1] -
pour_to_jug1))

        # Add child nodes to the current node
        for next_state in next_states:
            child_node = AONode(next_state, heuristic(next_state, goal_state))
            current_node.add_child(child_node)
            child_node.parent = current_node
            heapq.heappush(priority_queue, (child_node.heuristic, child_node))

        # Update the cost and best child of the current node
        for child, cost in current_node.children:
            if child.cost + cost < current_node.cost:
                current_node.cost = child.cost + cost
                current_node.best_child = child

        # Mark the current node as solved if all children are solved
```

```
        current_node.solved = all(child.solved for child, _ in current_node.children)

    # If no solution found, return None
    return None

# Define the capacities of the jugs and the target amount
jug1_capacity = 4
jug2_capacity = 3
target = (2, 0)

# Solve the problem using AO* search
solution_path = ao_star_search((0, 0), target, jug1_capacity, jug2_capacity)

# Print the solution path
if solution_path:
    print("Solution path found:")
    for state in solution_path:
        print(state)
else:
    print("No solution found.")
```

5.Queens

```
def is_safe(board, row, col):
    # Check this row on left side
```

```
    for i in range(col):
        if board[row][i] == 1:
            return False

    # Check upper diagonal on left side
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    # Check lower diagonal on left side
    for i, j in zip(range(row, len(board), 1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    return True

def solve_n_queens_util(board, col):
    # Base case: If all queens are placed
    if col >= len(board):
        return True

    # Consider this column and try placing this queen in all rows one by one
    for i in range(len(board)):
        if is_safe(board, i, col):
            # Place this queen in board[i][col]
            board[i][col] = 1
```

```
            # Recur to place rest of the queens
            if solve_n_queens_util(board, col + 1):
                return True

            # If placing queen in board[i][col] doesn't lead to a solution,
            # then backtrack and remove queen from board[i][col]
            board[i][col] = 0

    # If the queen cannot be placed in any row in this column, return False
    return False

def solve_n_queens(n):
    board = [[0] * n for _ in range(n)]
    if not solve_n_queens_util(board, 0):
        print("Solution does not exist")
        return False

    # Print the solution
    for row in board:
        print(" ".join("Q" if x == 1 else "." for x in row))
    return True

# Solve the 8-Queens problem
solve_n_queens(8)
```