Python - Problems

1. Travelling salesman problem

```
import numpy as np

class TravelingSalesman:
    def __init__(self, distance_matrix):
        self.distance_matrix = distance_matrix
        self.num_cities = len(distance_matrix)

    def nearest_neighbor(self, start=0):
        visited = [start]
        total_distance = 0
        current_city = start
        for _ in range(self.num_cities - 1):
            distances = self.distance_matrix[current_city]
            nearest_city = None
            min_distance = float('inf')
            for city in range(self.num_cities):
                if city not in visited and distances[city] < min_distance:
                    nearest_city = city
                    min_distance = distances[city]
            total_distance += min_distance
            visited.append(nearest_city)
            current_city = nearest_city
        total_distance += self.distance_matrix[current_city][start]
        visited.append(start)
        return visited, total_distance
distance_matrix = [
    [0, 29, 20, 21, 17],
    [29, 0, 15, 28, 12],
    [20, 15, 0, 18, 24],
    [21, 28, 18, 0, 31],
    [17, 12, 24, 31, 0]
]
tsp = TravelingSalesman(distance_matrix)
tour, distance = tsp.nearest_neighbor(start=0)
print("Tour:", tour)
print("Total Distance:", distance)
```

Output:

```
Tour: [0, 4, 1, 2, 3, 0]
Distance: 83
```

2. Chinese Postman problem

```python
def find_odd_degree_vertices(graph):
    degrees = {}
    for u, v in graph:
        degrees[u] = degrees.get(u, 0) + 1
        degrees[v] = degrees.get(v, 0) + 1
    odd_vertices = []
    for vertex, degree in degrees.items():
        if degree % 2 != 0:
            odd_vertices.append(vertex)
    return odd_vertices

def chinese_postman(graph):
    odd_vertices = find_odd_degree_vertices(graph)
    if len(odd_vertices) == 0 or len(odd_vertices) == 2:
        return True
    else:
        return False

graph1 = [(0, 1), (0, 2), (0, 3), (1, 2), (2, 3)]
graph2 = [(0, 1), (0, 2), (0, 3), (1, 2), (2, 3), (1, 3)]

for graph in [graph1, graph2]:
    is_eulerian = chinese_postman(graph)
    if is_eulerian:
        print("A solution exists for the graph:")
        print(graph)
    else:
        print("No solution exists for the graph:")
        print(graph)
```

Output:

```
A solution exists for the graph:
[(0, 1), (0, 2), (0, 3), (1, 2), (2, 3)]
No solution exists for the graph:
[(0, 1), (0, 2), (0, 3), (1, 2), (2, 3), (1, 3)]
```

3. Towers of Hanoi Problem

```
n = int(input("Enter the number of rings: "))

def tower_of_hanoi(n, source, destination, auxiliary):
    if n == 1:
        print(f"Move disk 1 from {source} to {destination}")
    else:
        tower_of_hanoi(n - 1, source, auxiliary, destination)
        print(f"Move disk {n} from {source} to {destination}")
        tower_of_hanoi(n - 1, auxiliary, destination, source)

tower_of_hanoi(n, 'A', 'C', 'B')
```

Output:

```
Enter the number of rings: 3
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
```

4. Missionaries and cannibals problem

```
class RiverCrossing:
    def __init__(self, initial_state=(3, 3, 1)):
        self.initial_state = initial_state
        self.visited = set()

    def is_safe(self, state):
        m, c, boat = state
        return not (m < c and m > 0) and not ((3 - m) < (3 - c) and (3 - m) > 0)
```

```
    def get_next_states(self, state):
        m, c, boat = state
        next_states = []
        if boat == 1:
            if m > 0:
                next_states.append((m - 1, c, 0))
            if m > 1:
                next_states.append((m - 2, c, 0))
            if c > 0:
                next_states.append((m, c - 1, 0))
            if c > 1:
                next_states.append((m, c - 2, 0))
            if m > 0 and c > 0:
                next_states.append((m - 1, c - 1, 0))
        else:
            if m < 3:
                next_states.append((m + 1, c, 1))
            if m < 2:
                next_states.append((m + 2, c, 1))
            if c < 3:
                next_states.append((m, c + 1, 1))
            if c < 2:
                next_states.append((m, c + 2, 1))
            if m < 3 and c < 3:
                next_states.append((m + 1, c + 1, 1))
        return [next_state for next_state in next_states if self.is_safe(next_state)]

    def solve(self):
        queue = [(self.initial_state, [])]
        while queue:
            state, path = queue.pop(0)
            if state in self.visited:
                continue
            self.visited.add(state)
            if state == (0, 0, 0):
                return path + [state]
            for next_state in self.get_next_states(state):
                queue.append((next_state, path + [state]))
        return None

river_crossing = RiverCrossing()
```

```python
solution = river_crossing.solve()
if solution:
    print("Solution found:")
    for step in solution:
        print(step)
else:
    print("No solution found.")
```

Output:

```
Solution found:
(3, 3, 1)
(3, 1, 0)
(3, 2, 1)
(3, 0, 0)
(3, 1, 1)
(1, 1, 0)
(2, 2, 1)
(0, 2, 0)
(0, 3, 1)
(0, 1, 0)
(1, 1, 1)
(0, 0, 0)
```

5. Eight Queens Problem

```python
class NQueens:
    def __init__(self, n=8):
        self.n = n
        self.board = [-1] * n

    def is_safe(self, row, col):
        for i in range(row):
            if self.board[i] == col or \
               abs(self.board[i] - col) == abs(i - row):
                return False
        return True

    def solve_n_queens_util(self, row):
        if row == self.n:
            return True
        for col in range(self.n):
            if self.is_safe(row, col):
                self.board[row] = col
                if self.solve_n_queens_util(row + 1):
                    return True
                self.board[row] = -1
        return False
```

```
    def solve(self):
        if self.solve_n_queens_util(0):
            self.print_board()
        else:
            print("No solution found.")

    def print_board(self):
        for row in range(self.n):
            line = ['Q' if self.board[row] == col else '.' for col in range(self.n)]
            print(' '.join(line))

n_queens = NQueens()
n_queens.solve()
```

Output:

```
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .
```

6. Monkey and Banana Problem

```
class Monkey:
    def __init__(self):
        self.on_box = False

    def push_box(self):
        print("Monkey pushes the box.")
        self.on_box = True

    def grab_bananas(self):
        if self.on_box:
            print("Monkey grabs the bananas!")
        else:
            print("Monkey can't reach the bananas.")

monkey = Monkey()
monkey.push_box()
monkey.grab_bananas()
```

Output:

```
Monkey pushes the box.
Monkey grabs the bananas!
```

7. The Konigsberg bridge problem

```python
class Graph:
  def __init__(self, graph_dict):
    self.graph_dict = graph_dict

  def is_eulerian(self):
    odd_degree_vertices = 0
    for vertex in self.graph_dict:
      if len(self.graph_dict[vertex]) % 2 != 0:
        odd_degree_vertices += 1
    if odd_degree_vertices == 0:
      return "Eulerian Circuit exists"
    elif odd_degree_vertices == 2:
      return "Eulerian Path exists"
    else:
      return "No Eulerian Path or Circuit exists"

graph = {
  'A': ['B', 'C'],
  'B': ['A', 'C', 'D'],
  'C': ['A', 'B', 'D'],
  'D': ['B', 'C']
}

g = Graph(graph)
result = g.is_eulerian()
print(result)
```

Output:

```
Eulerian Path exists
```