Challenge 1: Palindromic Paths in a Grid

Pseudocode:

```
function findPalindromicPaths(grid):
  paths = []

  // Recursive function to explore paths
  function explorePath(currentPath, row, col):
    if row == grid.length - 1 and col == grid.length - 1:
      # Check if current path is a palindrome
      if isPalindrome(currentPath):
        paths.append(currentPath)
      return

    # Explore valid moves (down, right)
    if row < grid.length - 1:
      explorePath(currentPath + grid[row + 1][col], row + 1, col)
    if col < grid.length - 1:
      explorePath(currentPath + grid[row][col + 1], row, col + 1)

  explorePath("", 0, 0)
  return paths

function isPalindrome(path):
  # Check if the path string reads the same forward and backward
  return path == path[::-1]
```

Challenge 2: Traveling Salesman Problem

Pseudocode:

```
// Assuming distance is represented by a matrix (distances) where
distances[i][j] represents distance between city i and j

function travelingSalesman(distances):
  n = len(distances)  // Number of cities

  // dp[i][mask] stores minimum distance to visit all cities in mask
starting from city i
  dp = [[float('inf')] * (1 << n) for _ in range(n)]

  # Starting from any city (0 in this case) with an initial mask (1
representing visited city 0)
  dp[0][1 << 0] = 0

  for mask in range(1, 1 << n):  // Iterate through all possible
visited city combinations
    for i in range(n):  // Iterate through each city as a starting
point
      if mask & (1 << i) != 0:  // Check if city i is already visited
in current mask
        for j in range(n):  // Iterate through other cities
```

```
          if i != j and (mask & (1 << j)) == 0:  // Avoid revisiting
and unvisited cities
            dp[i][mask] = min(dp[i][mask], distances[i][j] +
dp[j][mask ^ (1 << j)])  // Update min distance considering visiting
j from i

  // Minimum distance to visit all cities and return to starting city
0
  return min(dp[i][(1 << n) - 1] for i in range(n))
```

Challenge 3: LRU Cache Implementation

Pseudocode:

```
class Node:
  def __init__(self, key, value):
    self.key = key
    self.value = value
    self.prev = None
    self.next = None

class LRUCache:
  def __init__(self, capacity):
    self.capacity = capacity
    self.cache = {}  # Map keys to Doubly Linked List nodes
    self.head = None
    self.tail = None

  def set(self, key, value):
    if key in self.cache:
      # Update value and move node to head (most recently used)
      self.moveToHead(self.cache[key])
    else:
      # Create new node and add to head
      newNode = Node(key, value)
      self.addToHead(newNode)

      # Evict least recently used if exceeding capacity
      if len(self.cache) > self.capacity:
        self.removeLeastRecentlyUsed()

    self.cache[key] = newNode
```