Exercise 4

Explanation:

The code implements a solution to determine if a knight on a chessboard of size R x C can visit all squares exactly once, starting from a given position (posx, posy). It uses backtracking to explore all possible knight moves, defined by the moves list, and recursively attempts to visit all squares. The is_valid_move function ensures that moves stay within the board and avoid revisiting squares. The move function marks squares as visited with a move count, recursively explores valid moves, and backtracks if a path fails. If all squares are visited (move_count == R * C), the function returns True along with the board showing the knight's path. If no solution exists, it returns False.

Code:

```
def max_knight_moves(R, C, posy, posx):
  # Create a board to keep track of visited squares
  board = [[0 for _ in range(C)] for _ in range(R)]

  # Possible moves of a knight
  moves = [(2, 1), (2, -1), (-2, 1), (-2, -1), (1, 2), (1, -2), (-1, 2), (-1, -2)]

  # Function to check if the move is valid
  def is_valid_move(x, y):
    return 0 <= x < R and 0 <= y < C and board[x][y] == 0

  # Function to perform backtracking
  def move(x, y, move_count):
    if move_count == R * C: # If all squares are visited
      return True

    for dx, dy in moves: # Try all possible moves
      new_x = x + dx
      new_y = y + dy

      if is_valid_move(new_x, new_y): # If the move is valid
        board[new_x][new_y] = move_count + 1 # Mark the square as visited with the move count
        if move(new_x, new_y, move_count + 1): # Recursion to continue the path
          return True
        board[new_x][new_y] = 0 # Backtrack: unmark the square, as smth went wrong

    return False # If no valid moves are found, return False

  board[posx][posy] = 1 # Mark the starting position as visited
  if move(posx, posy, 1): # Start backtracking from the starting position
    return True, board # Return True and the board with the path
  else:
    return False, board
```

```python
# Example usage
R = 7
C = 3
posx = 0 #the board is 0-indexed
posy = 2

""" Example indexes board:
(0,0),(0,1),(0,2)
(1,0),(1,1),(1,2)
(2,0),(2,1),(2,2)
(3,0),(3,1),(3,2)
(4,0),(4,1),(4,2)
(5,0),(5,1),(5,2)
(6,0),(6,1),(6,2)
"""

print("Is it possible to visit all squares?")

result, board = max_knight_moves(R, C, posy, posx)

if result:
    print("Yes, it is possible.")

    print("Path:")
    # Print the board with the path in a nice format
    for row in board:
        print(" ".join(f"{cell:2}" for cell in row)) # Print each cell with a width of 2

else:
    print("No, it is not possible.")
```

Test case:

```
R = 7
C = 3
posx = 0 #the board is 0-indexed
posy = 2

Is it possible to visit all squares?
Yes, it is possible.
Path:
3  6   1
8  21  4
5  2   7
20 9   18
17 12 15
14 19 10
11 16 13
```

Exercise 6

Explanation:

The algorithm works as follows, first of all we check if the string provided is already the character that we are searching for, if not, we iterate through the whole string and transforming a pair of characters into one, using backtracking we are able to, if unable to find a correct solution, continue the execution, testing for other combinations of characters of the string provided, let that be the original one or some derived from it.

Code:

```
def backtrack_substitution(text,characterToFind, M):
  # Base case: if the text is reduced to a single character
  if len(text) == 1 and text[0] == characterToFind:
    return True

  # Recursive function to perform backtracking
  def backtrack(text):
    # If the text is reduced to a single character
    if len(text) == 1 and text[0] == characterToFind:
      return True

    # Iterate through the text to find pairs of characters to substitute
    for i in range(len(text)-1):
      char1 = text[i]
      char2 = text[i+1]

      # Find the indices of the characters in the substitution table
      index1 = M[0].index(char1)
      index2 = M[0].index(char2)
      # Get the substitution character from the table
      substitution_char = M[index1][index2]
      # Create a new text with the substitution
      new_text = text[:i] + substitution_char + text[i+2:]

      # Recursively call backtrack with the new text
      if backtrack(new_text):
        return True

    # If no valid substitution is found, return False
    return False
  # Start backtracking from the original text
  result = backtrack(text)
  return result

# Example usage
M=[
  ['_','a','b','c','d'],
  ['a','b','b','a','d'],
  ['b','c','a','d','a'],
  ['c','b','a','c','c'],
```

```
    ['d','d','c','d','b']
  ]

text = 'abbababa'
characterToFind = 'd'
result= backtrack_substitution(text, characterToFind, M)
if result:
    print(f"Yes, it is possible to reduce '{text}' to '{characterToFind}'")
else:
    print(f"No, it is not possible to reduce '{text}' to '{characterToFind}'")
```

Test case:

text = 'abbababa'
characterToFind = 'd'

Yes, it is possible to reduce 'abbababa' to 'd'