8.4 Recursion

oddCount

In [1]:

```
def oddCount(L):
    if L == []:
        return 0
    else:
        rest = oddCount(L[1:])
        if L[0] % 2 != 0:
            return 1 + rest
        else:
            return rest

# @testFunction
def testOddCount():
    assert(oddCount([]) == 0)
    assert(oddCount([1]) == 1)
    assert(oddCount([2]) == 0)
    assert(oddCount([1,2,3,4,5,4,3]) == 4)
    assert(oddCount([1,2,3,4,5,4,3,2]) == 4)
    assert(oddCount([2,4,6,8,10,12,14]) == 0)
    assert(oddCount([1,1,1,1,1]) == 5)

def main():
    testOddCount()

main()
```

oddSum

In [2]:

```
def oddSum(L):
    if L == []:
        return 0
    else:
        rest = oddSum(L[1:])
        if L[0] % 2 != 0:
            return L[0] + rest
        else:
            return rest

# @testFunction
def testOddSum():
    assert(oddSum([]) == 0)
    assert(oddSum([1]) == 1)
    assert(oddSum([2]) == 0)
    assert(oddSum([1,2,3,4,5,4,3]) == 12)   # 1+3+5+3
    assert(oddSum([1,2,3,4,5,4,3,2]) == 12) # 1+3+5+3
    assert(oddSum([2,4,6,8,10,12,14]) == 0)
    assert(oddSum([1,1,1,1,1]) == 5) # 1+1+1+1+1

def main():
    testOddSum()

main()
```

oddValues

In [3]:

```
def oddValues(L):
    if L == []:
        return []
    else:
        rest = oddValues(L[1:])
        if L[0] % 2 == 1:
            return [L[0]] + rest
        else:
            return rest

# @testFunction
def testOddValues():
    assert(oddValues([]) == [])
    assert(oddValues([1]) == [1])
    assert(oddValues([2]) == [])
    assert(oddValues([1,2,3,4,5,4,3]) == [1,3,5,3])
    assert(oddValues([1,2,3,4,5,4,3,2]) == [1,3,5,3])
    assert(oddValues([2,4,6,8,10,12,14]) == [])
    assert(oddValues([1,1,1,1,1]) == [1,1,1,1,1])

def main():
    testOddValues()

main()
```

oddMax

In [4]:

```
def oddMax(L):
    if L == []:
        return None
    else:
        maxOdd = oddMax(L[1:])
        if L[0] % 2 == 1:
            if maxOdd == None:
                maxOdd = L[0]
            elif L[0] > maxOdd:
                maxOdd = L[0]
        return maxOdd

# @testFunction
def testOddMax():
    assert(oddMax([]) == None)
    assert(oddMax([1]) == 1)
    assert(oddMax([2]) == None)
    assert(oddMax([1,2,3,4,5,4,3]) == 5)
    assert(oddMax([1,2,3,4,5,4,3,2]) == 5)
    assert(oddMax([2,4,6,8,10,12,14]) == None)
    assert(oddMax([1,1,1,1,1]) == 1)

def main():
    testOddMax()

main()
```

interleave

In [5]:

```
def interleave(L, M):
    if L == []:
        return M
    elif M == []:
```

```
        return L
    else:
        return [L[0]] + [M[0]] + interleave(L[1:], M[1:])

# @testFunction
def testInterleave():
    assert(interleave([1,2], [3,4]) == [1,3,2,4])
    assert(interleave([1,2], [3,4,5,6]) == [1,3,2,4,5,6])
    assert(interleave([1,2,5,6], [3,4]) == [1,3,2,4,5,6])
    assert(interleave([],[3,4]) == [3,4])
    assert(interleave([1,2],[]) == [1,2])
    assert(interleave([1,2,3,4],[5,6,7]) == [1,5,2,6,3,7,4])

def main():
    testInterleave()

main()
```

flatten

In [6]:

```
def flatten(L):
    if L == []:
        return []
    else:
        rest = flatten(L[1:])
        if isinstance(L[0], list):
            return flatten(L[0]) + rest
        else:
            return [L[0]] + rest

# @testFunction
def testFlatten():
    assert(flatten([1,[2]]) == [1,2])
    assert(flatten([1,2,[3,[4,5],6],7]) == [1,2,3,4,5,6,7])
    assert(flatten(['wow', [2,[[]]], [True]]) == ['wow', 2, True])
    assert(flatten([]) == [])
    assert(flatten([[]]) == [])
    assert(flatten([3]) == [3])
    assert(flatten([[[3]]]) == [3])

def main():
    testFlatten()

main()
```

getCourse

In [7]:

```
# from cmu_cs3_utils import testFunction

def getCourse(courseCatalog, course):
    prefix = courseCatalog[0]
    subCatalog = courseCatalog[1:]
    if course in courseCatalog:
        return prefix + '.' + course
    else:
        for list in subCatalog:
            courseOrder = getCourse(list, course)
            if courseOrder != None:
                return prefix + '.' + courseOrder

# @testFunction
```

```python
def testGetCourse():
    courseCatalog = ['CMU',
                        ['CIT',
                            [ 'ECE', '18-100', '18-202', '18-213' ],
                            [ 'BME', '42-101', '42-201' ],
                        ],
                        ['SCS',
                            [ 'CS',
                              ['Intro', '15-110', '15-112' ],
                              '15-122', '15-150', '15-213'
                            ],
                        ],
                        '99-307', '99-308'
                    ]
    assert(getCourse(courseCatalog, '18-100') == 'CMU.CIT.ECE.18-100')
    assert(getCourse(courseCatalog, '15-112') == 'CMU.SCS.CS.Intro.15-112')
    assert(getCourse(courseCatalog, '15-213') == 'CMU.SCS.CS.15-213')
    assert(getCourse(courseCatalog, '99-307') == 'CMU.99-307')
    assert(getCourse(courseCatalog, '15-251') == None)

    courseCatalog = ['SA',
                        ['CB', '10-100', '10-200',
                            ['DC', '20-300' ]
                        ]
                    ]
    assert(getCourse(courseCatalog, '10-100') == 'SA.CB.10-100')
    assert(getCourse(courseCatalog, '10-200') == 'SA.CB.10-200')
    assert(getCourse(courseCatalog, '20-300') == 'SA.CB.DC.20-300')
    assert(getCourse(courseCatalog, '30-400') == None)

def main():
    testGetCourse()

main()
```

powerSet

In [8]:

```python
# from cmu_cs3_utils import testFunction

def powerset(L):
    if L == []:
        return [[]]

    else:
        first = L[0]
        rest = L[1:]
        result = []
        for value in powerset(rest):
            result.append(value)
            result.append(sorted(value + [first]))
        return sorted(result)

# @testFunction
def testIsPowerSet():
    # Make sure that your output list is sorted.
    L = [1, 2, 3]
    output = [[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]]
    assert(powerset(L) == sorted(output))

    L = [5, 8]
    output = [[], [5], [8], [5, 8]]
    assert(powerset(L) == sorted(output))
```

```
    L = [-1]
    output = [[], [-1]]
    assert(powerset(L) == sorted(output))

    L = []
    assert(powerset(L) == [[]])

def main():
    testIsPowerSet()

main()
```

addOffsets

In [9]:

```
# from cmu_cs3_utils import testFunction

def addOffsets(offsets, L):
    if L == []:
        return []
    else:
        newOffsets = offsets[1:] + [offsets[0]]
        rest = addOffsets(newOffsets, L[1:])
        return [L[0]+offsets[0]] + rest

# @testFunction
def testAddOffsets():
    assert(addOffsets([1,2],[10,20,30,40,50]) == [11,22,31,42,51])
    assert(addOffsets([1,2,3],[10,20,30,40,50,60,70]) == [11,22,33,41,52,63,71
    assert(addOffsets([1,2,3],[10,20]) == [11,22])
    assert(addOffsets([1],[10,20]) == [11,21])
    assert(addOffsets([1],[10]) == [11])
    assert(addOffsets([1],[ ]) == [ ])

def main():
    testAddOffsets()

main()
```

hasRepeatedDigit

In [10]:

```
def hasRepeatedDigit(n):
    n = abs(n)
    if n < 10:
        return False
    else:
        rest = hasRepeatedDigit(n//10)
        return n%10 == (n // 10) % 10 or rest


# @testFunction
def testHasRepeatedDigit():
    assert(hasRepeatedDigit(1221) == True)
    assert(hasRepeatedDigit(1212) == False)
    assert(hasRepeatedDigit(-1221) == True)
    assert(hasRepeatedDigit(-1212) == False)
    assert(hasRepeatedDigit(1234) == False)
    assert(hasRepeatedDigit(0) == False)
    assert(hasRepeatedDigit(1123) == True)
    assert(hasRepeatedDigit(1233) == True)

def main():
```

```
    testHasRepeatedDigit()

main()
```

isAlmostPalindrome

In [11]:

```
def isAlmostPalindrome(s):
    result =  isAlmostPalindromeHelper(s, 0)
    return True if result == 1 else False

def isAlmostPalindromeHelper(s, result):
    if len(s) <= 1:
        return result
    else:
        if s[0] != s[-1]:
            result += 1
        return isAlmostPalindromeHelper(s[1:-1], result)

# @testFunction
def testIsAlmostPalindrome():
    assert(isAlmostPalindrome('stars') == True) # r -> t
    assert(isAlmostPalindrome('abca') == True) # c -> b
    assert(isAlmostPalindrome('abc') == True)  # c -> a
    assert(isAlmostPalindrome('abcb') == False)
    assert(isAlmostPalindrome('kayak') == False) # already a palindrome!
    assert(isAlmostPalindrome('a') == False) # already a palindrome!

def main():
    testIsAlmostPalindrome()

main()
```

hasSublistSum

Note: you may not use for or while loops in this exercise. Instead, use recursion.

Write the function hasSublistSum(L, s) that takes a list of integers L and an integer s, and returns True if there exist elements in L that sum to s. Otherwise, the function returns False.

For example, hasSublistSum([1, 6, 2, 8], 3) returns True because 1 and 2 sum to 3. On the other hand, hasSublistSum([7, -1, 4], -5) returns False because no combination of elements sums to -5. In addition, the function should always return true when s = 0 since the empty list is a sublist of any list, and its sum is 0.

In [12]:

```
# from cmu_cs3_utils import testFunction

def hasSublistSum(L, s):
    if s == 0:
        return True
    elif L == []:
        return False
    else:
        first = L[0]
        rest = L[1:]
        # print(first,rest, s)
        if first == s or hasSublistSum(rest, s-first):
            return True
        else:
            return hasSublistSum(rest, s)
```

```
# @testFunction
def testHasSublistSum():
    L = [1, 6, 1, 1, 8]
    assert(hasSublistSum(L, 3) == True)

    L = [3, -1, 7]
    assert(hasSublistSum(L, -1) == True)

    L = [8, 1, 1]
    assert(hasSublistSum(L, 2) == True)

    L = [7, -1, 4]
    assert(hasSublistSum(L, -5) == False)

    L = []
    assert(hasSublistSum(L, 0) == True)


def main():
    testHasSublistSum()

main()
```

digitCountMapInRange

Note: you may not use for or while loops here. Instead, use recursion.

With this in mind, write the recursive function digitCountMapInRange(lo, hi) that takes integers lo and hi, and returns a map of the counts of all the digits that occur in the numbers between lo and hi inclusively.

For example, for digitCountMapInRange(9, 12), we consider the numbers in the range from 9 to 12. That is: 9, 10, 11, 12. These numbers include one 0, four 1's, one 2, and one 9, so: assert(digitCountMapInRange(9, 11) == { 0:1, 1:4, 2:1, 9:1 }).

Be sure to handle negative numbers and 0 correctly. For example, for digitCountMapInRange(-1, 3), we consider the numbers in the range from -1 to 3. That is: -1, 0, 1, 2, 3. These numbers include one 0, two 1's, one 2, and one 3, so: assert(digitCountMapInRange(-1, 3) == { 0:1, 1:2, 2:1, 3:1 })

In [13]:

```
# from cmu_cs3_utils import testFunction

def digitCountMapInRange(lo, hi):
    return digitCountMapInRangeHelper(lo, hi, dict())

def digitCountMapInRangeHelper(lo, hi, result):
    if lo > hi:
        return result
    else:
        lo_new = abs(lo)
        result[lo_new%10] = result.get(lo_new%10, 0) + 1
        if lo_new//10 > 0:
            result[(lo_new//10)%10] = result.get((lo_new//10)%10, 0) + 1
        res = digitCountMapInRangeHelper(lo+1, hi, result)
        return dict(sorted(res.items()))

# @testFunction
def testDigitCountMapInRange():
    assert(digitCountMapInRange(9, 12) == { 0:1, 1:4, 2:1, 9:1 })
    assert(digitCountMapInRange(8, 9) == { 8:1, 9:1 })
```

```
    assert(digitCountMapInRange(-1, 3) == { 0:1, 1:2, 2:1, 3:1 })
    assert(digitCountMapInRange(20, 22) == { 0:1, 1: 1, 2:4 })
    assert(digitCountMapInRange(12, 9) == dict())

def main():
    testDigitCountMapInRange()

main()
```

8.7 Fractals

In [14]:

```
def onAppStart(app):
    app.level = 0

def drawFractal(level, cx, cy, r):
    if level == 0:
        drawCircle(cx, cy, r, fill='black')
    else:
        drawCircle(cx, cy, r*3/4, fill='black')
        drawFractal(level-1, cx-r/2, cy-r*2/3, r/3)
        drawFractal(level-1, cx+r/2, cy-r*2/3, r/3)


def onKeyPress(app, key):
    if (key in ['up', 'right']) and (app.level < 4):
        app.level += 1
    elif (key in ['down', 'left']) and (app.level > 0):
        app.level -= 1

def redrawAll(app):
    drawFractal(app.level, 200, 250, 150)
    drawLabel(f'Level {app.level} Fractal',
              app.width/2, 30, size=16, bold=True)
    drawLabel('Use arrows to change level',
              app.width/2, 50, size=12, bold=True)

def main():
    runApp()
```

8.8 Backtracking

solveKingsTour

Note: you must use backtracking to solve this problem. As long as you use backtracking with recursion properly, you may also use for or while loops here.

Background: in chess, a king can move to any of the immediately neighboring squares (or "cells", as we often call values on a 2d board). For a cell at (row, col) that is not next to an edge or corner, this will include 8 neighbors, as shown here: (row-1, col-1) (row-1, col) (row-1, col+1) (row , col-1) (row , col) (row , col+1) (row+1, col-1) (row+1, col) (row+1, col+1)

Next, given any rows-by-cols rectangular board (and not just an 8x8 board), a king's tour is a numbering of the cells such that: Every cell contains a positive integer. All the integers are between 1 and rows*cols (inclusive). Starting from 1, you can reach each subsequent number by making a legal king move. For example, consider this 2d list: [[12, 11, 10, 9], [3, 1, 8, 6], [2, 4, 5, 7]]

This board is a legal king's tour.

With this in mind, write the function solveKingsTour(initialBoard) that takes a 2d list of integers which are a partially-completed king's tour, so that the first N values are already placed on the board for some value of N, and the unused cells all contain 0's. Your function should use backtracking to solve this board (without mutating the initialBoard), and return the solution, or None if there is no solution. Also, if the initialBoard is entirely empty (full of 0's), you should start your king's tour by placing a 1 in the top-left corner of the board.

Because there are many possible solutions, we have included the helper function solvesKingsTour(initialBoard, solutionBoard) that our test function calls to verify that your solution in fact works.

In [15]:

```
# from cmu_cs3_utils import testFunction
import copy

def solveKingsTour(initialBoard):
    board = copy.deepcopy(initialBoard)
    row, col = findBiggestValue(board)
    if row == 0 and col == 0:
        board[0][0] = 1
        row, col = 0, 0
    return solve(board, row, col)

def findBiggestValue(board):
    rows,cols = len(board), len(board[0])
    bestValue = 0
    bestRow, bestCol = 0, 0
    for row in range(rows):
        for col in range(cols):
            if board[row][col] > bestValue:
                bestValue = board[row][col]
                bestRow, bestCol = row, col
    return bestRow, bestCol

def solve(board, row, col):
    rows,cols = len(board), len(board[0])
    if board[row][col] == rows*cols:
        return board
    else:
        prevValue = board[row][col]
        nextValue = prevValue + 1
        for nextRow in (row-1, row, row+1):
            for nextCol in (col-1, col, col+1):
                if (((nextRow,nextCol) != (row, col)) and
                    (0 <= nextRow < rows) and (0 <= nextCol < cols) and
                    (board[nextRow][nextCol] == 0)):
                       board[nextRow][nextCol] = nextValue
                       solution = solve(board, nextRow, nextCol)
                       if solution != None:
                           return board
                       board[nextRow][nextCol] = 0
    return None



# This helper function is used by the test function below
def solvesKingsTour(initialBoard, solutionBoard):
    rows,cols = len(initialBoard), len(initialBoard[0])
    # first verify the non-0 values in the initialBoard are in the solutionBoa
```

```
    for row in range(rows):
        for col in range(cols):
            if initialBoard[row][col] != 0:
                if initialBoard[row][col] != solutionBoard[row][col]:
                    return False
    # next, verify that the solutionBoard is actually a kings tour:
    # 1. Load a dictionary mapping each move to its location on the board:
    moveLocations = dict()
    for row in range(rows):
        for col in range(cols):
            move = solutionBoard[row][col]
            if ((move < 1) or (move > rows*cols)):
                return False
            moveLocations[move] = (row, col)
    # 2. Make sure we have the right number of moves
    if len(moveLocations) != rows*cols:
        return False
    # 3. Make sure each move is a legal move:
    for move in range(1, rows*cols):
        row0,col0 = moveLocations[move]
        row1,col1 = moveLocations[move+1]
        drow = abs(row0 - row1)
        dcol = abs(col0 - col1)
        if (drow > 1) or (dcol > 1):
            return False
    # Everything passed, so we have a solution!
    return True

# @testFunction
def testSolveKingsTour():
    initialBoard = [[0, 0, 0],
                    [0, 0, 0],
                    [0, 0, 0],
                    [0, 0, 0]]
    initialBoardCopy = copy.deepcopy(initialBoard)
    solutionBoard = solveKingsTour(initialBoard)
    assert(solvesKingsTour(initialBoard, solutionBoard))
    assert(initialBoard == initialBoardCopy) # check initialBoard is unmutated

    initialBoard = [[0, 0, 1],
                    [0, 2, 0],
                    [0, 0, 0]]
    initialBoardCopy = copy.deepcopy(initialBoard)
    solutionBoard = solveKingsTour(initialBoard)
    assert(solvesKingsTour(initialBoard, solutionBoard))
    assert(initialBoard == initialBoardCopy) # check initialBoard is unmutated

def main():
    testSolveKingsTour()

main()
```

solveMiniSudoku

Note: you must use backtracking to solve this problem. As long as you use backtracking with recursion properly, you may also use for or while loops here.

Background: we have previously written a function to verify that a Sudoku board is a valid solution. Here, we will limit ourselves to a 4x4 Sudoku board, which we will solve using backtracking.

A 4x4 Sudoku board is solved if: Each row contains the numbers 1-4. Each column contains the numbers 1-4. Each of the 4 blocks, or quadrants, of the board contain the numbers 1-4. A 4x4 Sudoku board is legal, but unsolved, if some of the cells are 0's, and those 0's can be replaced with integers to make a solved board. Rephrased, a 4x4 Sudoku board is legal, but unsolved, if every row, column, and block contains numbers from 0 to 4, and the only number that occurs more than once is 0.

For example, this is a legal but unsolved Sudoku board: [[3, 0, 0, 0], [0, 1, 0, 3], [4, 0, 1, 0], [0, 0, 0, 0]]

We can solve this board by replacing the 0's with the appropriate numbers, to get this solved board: [[3, 4, 2, 1], [2, 1, 4, 3], [4, 3, 1, 2], [1, 2, 3, 4]]

With this in mind, write the function solveMiniSudoku(board) that takes a 2d list of integers which are a partially-completed 4x4 Sudoku board. Your function should use backtracking to solve this board (without mutating the initial board), and return the solution, or None if there is no solution.

In [16]:
```
# from cmu_cs3_utils import testFunction
import copy

def solveMiniSudoku(board):
    solutionBoard = copy.deepcopy(board)
    return solve(solutionBoard)

def solve(board):
    if isFull(board):
        return board
    else:
        rows, cols = len(board), len(board[0])
        row, col = findEmptyPosition(board)
        # print(row, col)
        for number in (1, 2, 3, 4):
            board[row][col] = number
            # print(board)
            if isLegalMove(board):
                # print('islegal')
                solution = solve(board)
                if solution != None:
                    return board
                else:
                    board[row][col] = 0
                    # print(board)
            else:
                # print('notlegal')
                board[row][col] = 0
                # print(board)
        # print('None')
        return None

def findEmptyPosition(board):
    rows, cols = len(board), len(board[0])
    for row in range(rows):
        for col in range(cols):
            if board[row][col] == 0:
                return row, col

def isFull(board):
    rows, cols = len(board), len(board[0])
```

```
    for row in range(rows):
        for col in range(cols):
            if board[row][col] == 0:
                return False
    return True

def isLegalMove(board):
    return legalRow(board) and legalCol(board) and legalQuadrant(board)

def legalRow(board):
    for rowList in board:
        for val in rowList:
            if val != 0:
                if rowList.count(val) > 1:
                    return False
    return True

def legalCol(board):
    rows, cols = len(board), len(board[0])
    for col in range(cols):
        colList = [board[row][col] for row in range(rows)]
        for val in colList:
            if val != 0:
                if colList.count(val) > 1:
                    return False
    return True

def legalQuadrant(board):
    rows, cols = len(board), len(board[0])
    list1 = []
    list2 = []
    list3 = []
    list4 = []
    for row in (0, 1):
        for col in (0, 1):
            list1.append(board[row][col])
    for row in (0, 1):
        for col in (2, 3):
            list2.append(board[row][col])
    for row in (2, 3):
        for col in (0, 1):
            list3.append(board[row][col])
    for row in (2, 3):
        for col in (2, 3):
            list4.append(board[row][col])
    for list in list1, list2, list3, list4:
        for val in list:
            if val != 0:
                if list.count(val) > 1:
                    return False
    return True

# @testFunction
def testSolveMiniSudoku():
    board1  = [[3, 0, 0, 0],
               [0, 1, 0, 3],
               [4, 0, 1, 0],
               [0, 0, 0, 0]]
    solved1 = [[3, 4, 2, 1],
               [2, 1, 4, 3],
               [4, 3, 1, 2],
               [1, 2, 3, 4]]
    board1Copy = copy.deepcopy(board1)
    assert(solveMiniSudoku(board1) == solved1)
```

```
    assert(board1 == board1Copy) # verify we do not mutate the original board

    board2  = [[4, 0, 0, 2],
               [0, 2, 0, 0],
               [0, 0, 3, 0],
               [0, 0, 0, 1]]
    solved2 = [[4, 3, 1, 2],
               [1, 2, 4, 3],
               [2, 1, 3, 4],
               [3, 4, 2, 1]]
    board2Copy = copy.deepcopy(board2)
    assert(solveMiniSudoku(board2) == solved2)
    assert(board2 == board2Copy) # verify we do not mutate the original board

    board3  = [[0, 3, 0, 0],
               [0, 0, 0, 3],
               [1, 0, 0, 0],
               [0, 0, 4, 0]]
    solved3 = [[2, 3, 1, 4],
               [4, 1, 2, 3],
               [1, 4, 3, 2],
               [3, 2, 4, 1]]
    board3Copy = copy.deepcopy(board3)
    assert(solveMiniSudoku(board3) == solved3)
    assert(board3 == board3Copy) # verify we do not mutate the original board

    board4 = [[4, 3, 0, 0], # This board has no solution
               [1, 0, 3, 0],
               [0, 0, 2, 0],
               [4, 1, 0, 1]]
    board4Copy = copy.deepcopy(board4)
    assert(solveMiniSudoku(board4) == None)
    assert(board4 == board4Copy) # verify we do not mutate the original board

def main():
    testSolveMiniSudoku()

main()
```

isKingsTour

Note: you may not use for or while loops in this exercise. Instead, use recursion.

Background: In chess, a king can move one square at a time in any of the 8 directions (up, down, left, right, up-left, up-right, down-left, down-right).

A "king's tour" is a 2D list of integers representing a valid sequence of moves by the king. More specifically, a king's tour with dimensions MxN should contain the integers from 1 to M*N, where each integer (other than 1) can be reached from the previous integer via a legal king's move.

For example, this is a king's tour:

[[3, 5, 1], [4, 2, 6]]

Notice that we can make a legal king's move to get from: 1 to 2 (down-left), 2 to 3 (up-left), 3 to 4 (down), 4 to 5 (up-right), and 5 to 6 (down-right). By contrast, the following 2D list is not a legal king's tour because there is no legal king's move to get from 5 to 6:

[[4, 3, 1], [5, 2, 6]]

With this in mind, use recursion to write the function isKingsTour(board) that takes a 2d list of integers and returns True if it is a legal king's tour and False otherwise.

Hint: You may want to first write a recursive helper function that returns the row, col location of the 1 on the board (or None if there is no 1). Then, check that the current value (starting at 1) has a neighbor one larger than it, and continue checking until you've hit all the values.

To see some more examples of valid and invalid kings tours, you can revisit the 8.10 solveKingsTour guided exercise. Note that the guided exercise asks you to generate a king's tour, while this exercise only asks you to check if a king's tour is legal.

In [17]:

```
# from cmu_cs3_utils import testFunction

def findPosition(board, row, col, currValue):
    rows, cols = len(board), len(board[0])
    if board[row][col] == currValue:
        return row, col
    else:
        if col < cols-1:
            return findPosition(board, row, col+1, currValue)
        elif row < rows-1:
            return findPosition(board, row+1, 0, currValue)
        return None

def isKingsTour(board):
    start = findPosition(board, 0, 0, 1)
    if start == None:
        return False
    else:
        startRow, startCol = start
        return isKingsTourHelper(board, startRow, startCol, 1)

def isKingsTourHelper(board, row, col, currValue):
    rows, cols = len(board), len(board[0])
    nextValue = currValue + 1
    if board[row][col] == rows*cols:
        return True
    else:
        nextPosition = findPosition(board, 0, 0, nextValue)
        if nextPosition != None:
            newRow, newCol = nextPosition
            if isLegalMove(row, col, newRow, newCol):
                return isKingsTourHelper(board, newRow, newCol, nextValue)
        return False

def isLegalMove(row, col, newRow, newCol):
    return row-1 <= newRow <= row+1 and col-1 <= newCol <= col+1

# @testFunction
def testIsKingsTour():
    board1 = [[3, 5, 1],
              [4, 2, 6]]
    assert(isKingsTour(board1) == True)

    board2 = [[1]]
    assert(isKingsTour(board2) == True)

    board3 = [[2]] # not a valid king's tour because it does not start with 1
    assert(isKingsTour(board3) == False)

    board4 = [[4, 3, 1],
```

```
                 [5, 2, 6]]
    assert(isKingsTour(board4) == False)

def main():
    testIsKingsTour()

main()
```