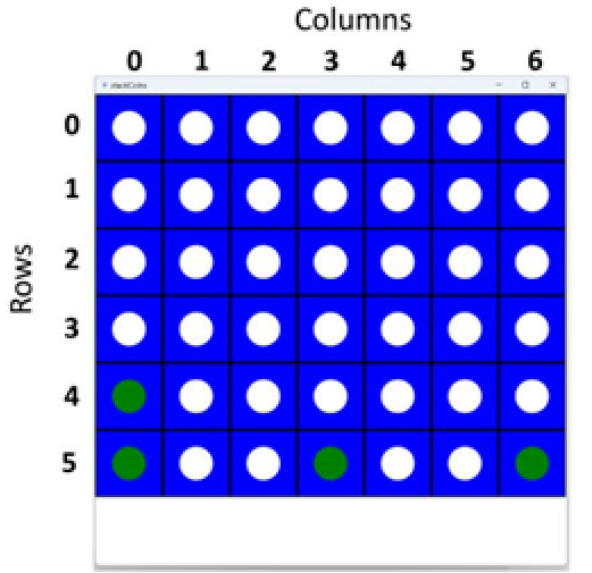
Review



(a) `drawStackCoins(app, "50,40,53,56")`

(b) (15 points) **Free Response:** Animations

Now, Let's animate the StackCoins board game.

In this part assume the function `drawStackCoins(app, coinCells)` already exists and works well. Do not write it again, instead just call it when you need it.

The game should have the following features:

1. The game starts with an empty board (all cells have white circles); See figure (b) from the previous problem.
2. A coin is inserted in a column by pressing at any place on that column.
3. A coin should be stacked on the selected column. Meaning it should be inserted on the next empty cell in that column. For example: the next empty cell in column 3 in Figure 2 is cell (`"23"`) since cell (`"33"`) is filled. *Remember to update the string coinCells to keep track of inserted coins.*
4. If the user presses on a column that is already full, a message should be displayed on the empty space at the bottom of the canvas saying: *Invalid Column !!!*; See figure 2 below.
5. The game can be reset by pressing the r key.

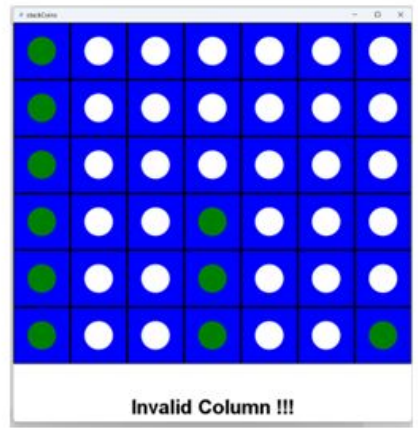


Figure 2: The view after trying to insert a coin in a full column (column 0)

Notes:

- **Design your helper functions wisely and the problem will be easier.**
- `.split()` may be useful here but you may only loop over the result, and may not index/slice the result or use list functions or list methods.
- Do not hardcode for a 400x400 canvas. However, you may assume the canvas is square and at least 100x100 pixels.
- You will be penalized if your code results in an MVC violation.
- Make reasonable choices for anything not specified above.
- To solve this, you need to write onAppStart, onKeyPress, onMousePress, and redrawAll.
- You do not need to include any imports or the main function.

5. (15 points) **Free Response:**

Write the function `isValidRGBStr(s)` that takes in a string `s` and returns `True` if `s` is a valid RBG string and `False` if it is not.

A valid RGB string is of the form `rgb(x,y,z)` or `RGB(x,y,z)` and satisfies the following constraints:

- `x`, `y`, `z` represent integers between 0 and 255.
- There are no white spaces within the string.

For example:

```
# valid RGB strings
assert(isValidRGBStr('rgb(0,0,0)') == True)
assert(isValidRGBStr('RGB(255,128,4)') == True)
assert(isValidRGBStr('RGB(255,255,255)') == True)

#invalid cases
# ups, spaces inside
assert(isValidRGBStr('RGB(255,255,   255)') == False)
# wrong capitalization: Rgb is not valid
assert(isValidRGBStr('Rgb(255,255,255)') == False)
# It must be of the form rgb(x,y,z) or RGB(x,y,z)
assert(isValidRGBStr('(255,128,4)') ==False)
 # red component equal to 15112 is not valid. It must be between 0 and 255
assert(isValidRGBStr("rgb(15112,2,255)") == False)
# nonsense RGB string, It must be of the form rgb(x,y,z) or RGB(x,y,z)
assert(isValidRGBStr('color blue') == False)
```

Hint: The following built-in `string` methods may be useful: `isdigit()` that checks if the characters are numerical, and `split`.

3. (7 points) **Free Response: Smallest Integer Inside Square Brackets**

Write the function `smallestIntInsideBrackets(s)` that finds the smallest integer among all numbers appearing **inside square brackets** `[ ]` in the given string `s`.

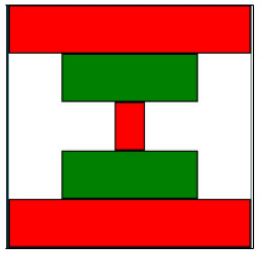
If there are no valid integers inside square brackets, return `None`. A valid integer consists only of digits and may have an optional leading $-$ (negative sign). Decimal numbers (floats) are not considered valid integers.

Here are some test cases:

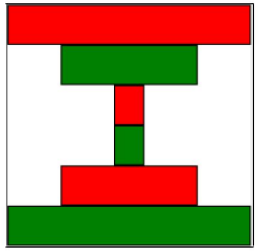
```
assert smallestIntInsideBrackets("15[112] Spring [25]") == 25
assert smallestIntInsideBrackets("15[112] Spring [25] -42") == 25
assert smallestIntInsideBrackets("15[112] Spring [25] [-42]") == -42
assert smallestIntInsideBrackets("NoBracketsHere") == None
assert smallestIntInsideBrackets("Floats are not Ints [15.112]") == None
assert smallestIntInsideBrackets("Be [care]ful with [non]digits [42]") == 42
```

drawBlockyHourGlass (quiz 5)

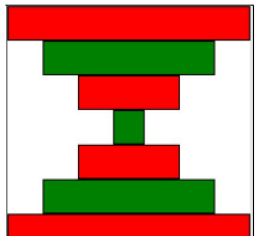
`drawBlockyHourGlass(app, 5)` produces the graphics below:



`drawBlockyHourGlass(app, 6)` produces the graphics below:



`drawBlockyHourGlass(app, 7)` produces the graphics below:



Here are the specifications:

- There are exactly n blocks.
- The blocks have the same height.
- The width of the blocks decreases proportionally towards the center and then increases back symmetrically.
- The smallest block(s) always have a width of 50 pixels.
- The largest blocks span the entire window width.
- The colors of the blocks alternate between red and green, with the topmost block always being red.

Examples:

Exam

Exercise

- At the end of this set of operations, which pairs of lists will still be aliased?

```
1 a = [ 1, 2, "x", "y" ]
2 b = a
3 c = [ 1, 2, "x", "y" ]
4 d = c
5 a.pop(2)
6 b = b + [ "wow" ]
7 c[0] = 42
8 d.insert(3, "yey")
```

1. **a** and **b**
2. **c** and **d**
3. **All** lists are aliases of the **same** list
4. **None** of them are aliased

CMUP-S24 Midterm

```
def ct4(L):
    M = L
    N = [v//10 for v in L]
    print(N)
    M.append(N.pop(0))
    L = sorted(L)
    print(L)
    print(M)
    print(N)

L = [15, 5]
ct4(L)
print(L) # do not miss this!
```

Lists: Taxonomy of problems

- **Cat 1**: given a list L, process L and compute some value out of its elements. Example: `alternatingSum`, `hasDuplicates`
- **Cat 2:** Given some input, return a **NEW** list with some properties. Example: return the *first n 7ish* numbers, `findDuplicates`
- **Cat 3:** Given a list L, write a ***non-destructive function*** to solve some task using L **without modifying** the original L.
  - Example: `nondestructiveRemoveNonInts`
- **Cat 4:** Given a list L, write a ***destructive function*** to solve some task that **mutates** L**.**
  - Example: `destructiveRemoveNonInts`
- **Cat 5:** Solve a task involving **2d lists.**
  - Example: is`MagicSquare`

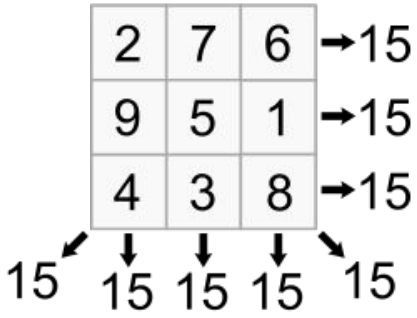
Example problems with two lists

`isSwappyListpair:` Two lists L1 and L2 are Swappy List Pairs if exactly one swap of two elements within L1 transforms it into L2. Write a function `isSwappyListPair(L1, L2)` that returns `True` if L1 and L2 are Swappy List Pairs, meaning L1 can be made identical to L2 by swapping exactly two of its elements. Otherwise, return `False`.

```
assert isSwappyListPair([1,2,3,42], [3,2,1,42])  # Swaps index 0 and 3
assert isSwappyListPair(['equal', 'lists'], ['equal', 'lists']) == False  # Identical lists
assert isSwappyListPair(["equal"], ["equal"]) == False  # Identical strings
assert isSwappyListPair([42, 1, 112], [122, 2, 2025]) == False  # Too many mismatches
assert isSwappyListPair([0,1,5,1,1,2], [5, 1, 2, 1, 1, 0]) == False  # Too many swaps required
assert isSwappyListPair([42, 3], [42, 2]) == False  # One mismatch, cannot be fixed with a swap
assert isSwappyListPair([15], [15,112]) == False  # Length mismatch
```

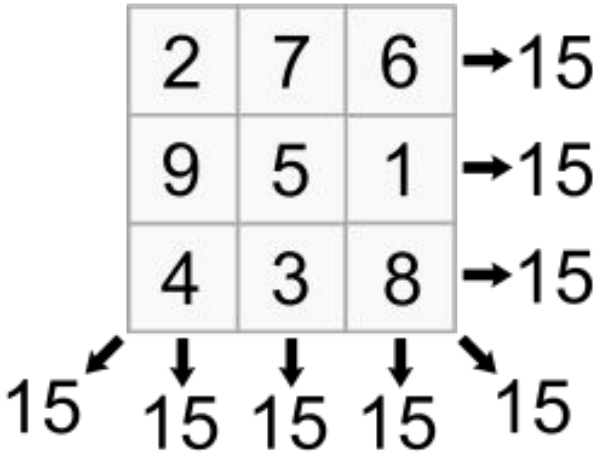
isMagicSquare

- Write the function `isMagicSquare(L)` that takes an arbitrary list L and returns `True` if it is a magic square and False otherwise, where a magic square has these properties:
  - The list is 2d, non-empty, **square**, and contains **only integers**, where no integer occurs more than once in the square.
  - Each row, each column, and each of the 2 diagonals each sum to the same total. Note that we do not require that the integers are strictly in the range from 1 to n for some n. We only require that the integers are unique and that the sums are identical.



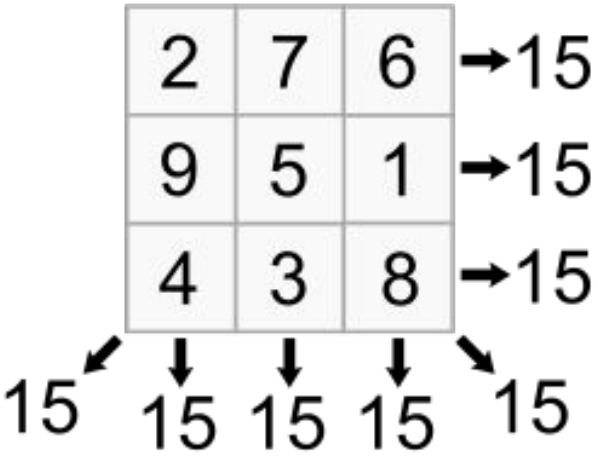
`isMagicSquare(L)`

- Check that L is a list
- Check that L is a non-empty *square* 2d list
  - Check that all elements are lists
  - Check that all elements have `len(L)` elements
- Check that all elements in every row of L are integers
- Check that all elements in every row are unique
- Check that the sum of rows, the sum of columns, and the sum of diagonals are equal



`isMagicSquare(L)`

- Check that L is a list
- Check that L is a non-empty *square* 2d list:
  - Check that all elements are lists
  - Check that all elements have `len(L)` elements
- Check that all elements in every row of L are integers
- Check that all elements in every row are unique
- Check that the sum of rows, the sum of columns, and the sum of diagonals are equal

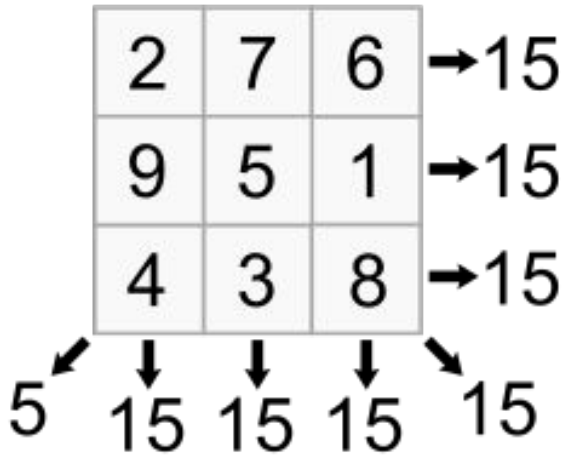


isValidSquare(L)

isValidSquare(L)

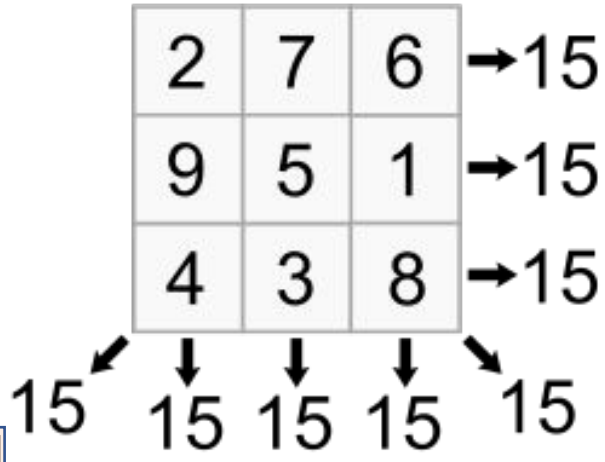
- Check that L is a list
- Check that L is a non-empty *square* 2d list:
  - Check that all elements are lists
  - Check that all elements have `len(L)` elements
- Check that all elements in every row of L are integers
- Check that all elements in every row are unique

```
1  def isValidSquare(L):
2      if not isinstance(L, list):
3          return False
4      if len(L) == 0:
5          return False
6      for row in L:
7          if not isinstance(row, list):
8              return False
9      nrows = len(L)
10     for row in L:
11         if len(row) != nrows:
12             return False
13     for row in L:
14         for val in row:
15             if not isinstance(val, int):
16                 return False
17     valuesSeen = []
18     for row in L:
19         for val in row:
20             if val in valuesSeen:
21                 return False
22             valuesSeen.append(val)
23     return True
```



isMagicSquare(L)

- Check that L is a list
- Check that L is a non-empty *square* 2d list:
  - Check that all elements are lists
  - Check that all elements have `len(L)` elements
- Check that all elements in every row of L are integers
- Check that all elements in every row are unique
- Check that the sum of rows, the sum of columns, and the sum of diagonals are equal



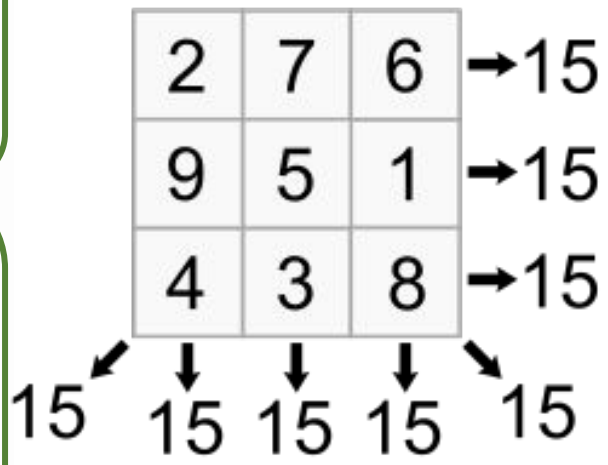
isMagic(L)

isMagic(L)

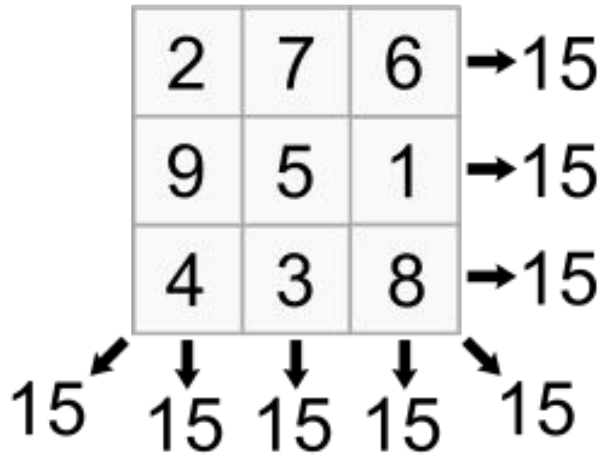
- Check the sum of rows
- Check the sum of columns
- Check the sum of diagonals

```
def isMagic(L):
    thesum = sum(L[0])  # sum the first row
    # check rows
    for row in L:
        if sum(row) != thesum:
            return False
    # check cols
    N = len(L)
    for jcol in range(N):
        col = []
        for irow in range(N):
            col.append(L[irow][jcol])
        if sum(col) != thesum:
            return False
    #diagonals
    # upper
    diag1 = []
    for i in range(N):
        diag1.append(L[i][i])
    if sum(diag1) != thesum:
        return False

    # upper
    diag2 = []
    for i in range(N):
        diag2.append(L[i][N-i-1])
    if sum(diag2) != thesum:
        return False
    return True
```



isMagicSquare(L)



Two helper functions

Main task

```python
def isValidSquare(L):
    if not isinstance(L, list):
        return False
    if len(L) == 0:
        return False
    for row in L:
        if not isinstance(row, list):
            return False
    nrows = len(L)
    for row in L:
        if len(row) != nrows:
            return False
    for row in L:
        for val in row:
            if not isinstance(val, int):
                return False
    valuesSeen = []
    for row in L:
        for val in row:
            if val in valuesSeen:
                return False
            valuesSeen.append(val)
    return True

def isMagic(L):
    thesum = sum(L[0])  # sum the first row
    # check rows
    for row in L:
        if sum(row) != thesum:
            return False
    # check cols
    N = len(L)
    for jcol in range(N):
        col = []
        for irow in range(N):
            col.append(L[irow][jcol])
        if sum(col) != thesum:
            return False
    #diagonals
    # upper
    diag1 = []
    for i in range(N):
        diag1.append(L[i][i])
    if sum(diag1) != thesum:
        return False

    # upper
    diag2 = []
    for i in range(N):
        diag2.append(L[i][N-i-1])
    if sum(diag2) != thesum:
        return False
    return True

def isMagicSquare(L):
    if not isValidSquare(L):
        return False
    return isMagic(L)
```