Daniel Schroeder
W Keith Striby
CS-325 Fall 2017
Group Assignment 2

The recursive approach:

```
OutsideLoop(){
      // MemoryMatrix = make an [n+1]x[n+1] matrix of -1
      // make the last row and column of value 0 to the memory matrix
      for 0 < i < n {
            Matrix[i][n] = 0
            Matrix[n][i] = 0
      }
      //call the inner recursion up the diagonal
      for 0 < i < n {
            innerRecursion(i, i, n, MemoryMatrix)
      }
}

InnerRecursion(){
      //check if we have memory of current square, ignore last row and column
      if MemoryMatrix[x][y] == -1
            for 0 < i < n
                  MemoryMatrix[i][y]=A[i][y]+max(InnerRecursion(Right),
                                                    InnerRecursion(Below))

                  if MemoryMatrix[i][y] > Maximum
                        Store MemoryMatrix[i][y] as new Maximum
      // we have it in memory so just return
      return MemoryMatrix[x][y]
}
```

This inner recursion moves "UP" the current column given by the parameters x and y. For each new square of the game matrix it reaches, it sets its value to the maximum of the squares to the right or below. When it looks for the max of these two squares, it recursively calls InnerRecursion(Right) and InnerRecursion(Below) which will compute that square's value, or get it from the memory matrix if it has already been memorized.

A key part of this algorithm is adding the buffer row and column with values of 0. This allows us to check the right and bottom squares in our recursion without having to check for index boundaries.

Iterative Dynamic Method:

```
Pseudo Code:
Iterative Approach(n):
      //Fill Memory Matrix with all -1
      VMile = [[-1 for x in range(n+1)] for y in range(n+1)]
      maxscore = 0

      # fill far right column with 0's so that we can check VMile[i][j+1]

      for i in range(n, -1, -1):
        VMile[i][n] = 0

      // go "up" each column assigning the value of each column to the square
      // value plus the max of the right and below squares

      for j in range(n - 1, -1, -1):
            // make lowest square 0 so we can check VMile[i+1][j]
            VMile[n][j] = 0

            for i in range(n - 1, -1, -1):

                  //assign value of square value plus max of right or bottom
                  VMile[i][j] = readInput(i, j) + max(VMile[i+1][j],
                                                      VMile[i][j+1])

                  //if current square is bigger than max, set max
                  maxscore = max(maxscore, VMile[i][j])

      return maxscore
```

The iterative approach is a double for-loop that begins at the nth x nth square in the matrix and looks at the squares to the right and below (which are buffered values of zero, so the max length of the bottom right corner of the board is equal to itself since it has no possible moves. Then the algorithm moves "up" in the y-direction, looking at the square above the nth x nth square. It will then take the max of the square to the right (0 buffer column) or below (the previously computed value) and add it to the value in the input matrix. Similarly, if this value is originally larger than the nth x nth square, say an example like:

X	20
X	-2

Then the max would be the zero that is buffered on the right side and the value would be its original value since the player can move off the board at that position. When the inside for-loop reaches an index of 0 (the top of the matrix), the outer for loop decrements and it begins working at the bottom square of the columns to the left. Now all the values we check to the right have already been solved and all the values to the bottom will be solved in the previous increment through the for-loop.

Running Time Analysis:

```
def greatestPath(n):
    VMile = [[-1 for x in range(n)]for y in range(n)]    } O(N^2) initialize array
    maxscore =0

    VMile.append([])
    # fill far right column with 0's
    for i in range(n, -1, -1):
        VMile[i].append(0)      } O(N) Make right row 0's
        VMile[n].append(0)

    # add last element n+1
    VMile[n].append(0) } O(C) Add last buffer square
    for j in range(n - 1, -1, -1):
        VMile[n][j] =0
        for i in range(n - 1, -1, -1):
            VMile[i][j] = readInput(i, j) +max(VMile[i+1][j], VMile[i][j+1])
            maxscore = max(maxscore, VMile[i][j])
                                                   } O(N^2) Loop through every square in matrix

    return maxscore
```

$$T(n) = O(n^2) + O(n^2) + O(n) + O(C)$$

$$= 2\,O(n^2) + O(n) + O(C)$$

$$= O(n^2)$$