Ken Ford
CSCI 3104
07/07/17

Homework 5

1) a)
A = [1, 5, 2, 3, 8]
LIS(A, 4, 3) = [1, 2], i = [0, 2, 3], len = 2
LIS(A, 5, ∞) = [1, 2, 3, 8], i = [0, 2, 3, 4], len = 4

b)
LIS(A, 0, M)
The base case of LIS would be 0 in this case, as we cannot check index A[0-1] as mentioned in the definition of j. As such, we treat the array as empty.

c)
LIS(A, j, M) for any 1 <= j <= n

The recurrence for this problem occurs when we find a value that we can add to the LIS in the comparison between A[i] and A[j]. If we determine that it does not belong to the LIS when A[i] < A[j], we do nothing to it. Otherwise we add it and add 1 to the length of the LIS.

d)
To memoize the above recurrence, we would save the solutions in a separate array for everything prior to the checked element in the array. If there exists a longer subsequence, we can just add 1 to the element in the current index. To find the max length of the LIS, we would just need to check for the maximum value in the array. This way we can reference solutions that were calculated earlier.

e) See .py file

2) a)

```
def findMaxScore(A):
    Initialize new n x n matrix, B
    B[0][0] = A[0][0]
    for i in range(0,row length):
        B[0][i] = A[0][i] + A[0][I + 1]
    for i in range(0,column length):
        B[i][0] = A[i][0] + A[I + 1][0]
    for i in range(1,row length):
        for j in range(1, column length)
            B[i][j] = max(initial node + right neighbor, initial node + below neighbor)

    Initialize new n length matrices for last column, row as lastColumn and lastRow
    lastColumnMax = max(lastColumn)
    lastRowMax = max(lastRow)
    maxScore = max(lastColumnMax, lastRowMax)
    return maxScore
```

b)

The algorithm is able to successfully find the maximum score possible in the table by through the table at a downward left angle. In doing so, they add and compare the possible moves to the right and downwards that we can take through the table. In these comparisons, we make the greedy choice of moving our comparison node to the maximum sum of the neighboring values. As such, we will always have the maximum possible value for an optimal move saved in our second array. By the time we reach the last row and column, we have the values for the best case scenarios for moving to each of the indices found on these two edges of the table. By taking the max of the last row and column, we find the maximum scores that percolated to the edges of the table. As such, the max of these two values will show us the largest possible score in the table. We know that it is the maximum as we never deal with any smaller outcomes in the comparison of the neighboring values.

c)

Time Complexity:

Initialization: O(2n)

B index calculation: O(n^2)

Max function calls x 3: O(3n)

Total time complexity: O(n^2 + 5n) →Asymptotically→ O(n^2)

Space Complexity:

Perform calculation for n x n matrix, so O(n^2)

3)
```
def findBlackHole(A):
    Initialize array for suspects
    for I in range(0, len(A)):
        candidate = A[i][i]
        if candidate's left neighbor == 1 (or off table):
            if candidate's right neighbor == 1 (or off table):
                if candidate's top neighbor == 0 (or off table):
                    if candidate's bottom neighbor == 0 (or off table):
                        suspects.append(candidate)
    for I in range(0, len(suspects)):
        if A[suspects[0]][suspects[1]] == 0:
            (if 0 exists in row, suspects[0] by definition cannot be black hole)
            delete suspects[0]
        else :
            (if 0 exists in column, suspects[1] by definition cannot be black hole)
            delete suspects[1]
    if there is still something in suspects array:
        (this for loop needs to skip the diagonal value)
        for I in range(0, i – 1, I + 1, len(A)):
            if A[i][suspects[0]] == 0:
                if A[suspects[0]][i] == 1:
                    return True, black hole found
    return false, no black hole found
```

Run time:

Finding suspects: O(n)

Minimize suspect list: O(n)

Final check for black hole: O(n)

Final run time: O(3n)→Asymptotically→O(n)

Vertices	0	1	2	3	4
0	0	-	-	-	0
1	-	0	-	-	0
2	-	-	0	-	0
3	-	-	-	0	0
4	1	1	1	1	0

* A black hole is defined as an element in an adjacency matrix that has n – 1 1's and one 0 in its in row and all 0's in its out column.

4) See .py file