cs3821 Assignment 2

Jennifer z5207930

April 2020

1 Question 2

1.1 Sample Questions

a) 3
b) 4
c) 4

1.2 Pseudo-Code

```
# Counts the number of free boxes across/down
def free_tiles(data, row, col, dir):
    nFree = 0
    if (dir == "across"):

        # Look through every column @row to find an empty slot
        for i in range(col, len(data)):
            if (data[i] > row): nFree += 1
            else: break
        return nFree

    if (dir == "down") : return int(data[col] - col)

# Get the next coord to look at
def getNextCoord(data, prev_coord, prev_direction):

    if (prev_coord == None):
        for i in range(len(data)):
            if (data[i] != 0): return (i, 0)

        # There are no unshaded regions at all.
        return None
```

```
    (p_row, p_col) = prev_coord
    if (prev_direction == "across"):

        # If there is still unshaded regions down
        if (data[p_col] >= p_row + 1): return (p_row + 1, p_col)

        # Look through starting at the next column
        for i in range (p_col + 1, len(data)):
            if (data[i] > p_row + 1): return (p_row + 1, i)

        # No more unshaded regions
        return None

    if (prev_direction == "down"):

        # Look through every column @ prev_row to find an empty slot
        for i in range(p_col, len(data)):
            if (data[i] > p_row): return (p_row, i)

        # No more unshaded regions
        return None


# Flipped the direction of the box for easier implementation
def minBlocks(data, m):
    minBlocks = 0

    # Get left-most, top-most empty co-ord
    prev_coord = None
    prev_direction = None
    coord = getNextCoord(data, prev_coord, prev_direction)
    while (coord != None):
        coord = getNextCoord(data, prev_coord, prev_direction)
        (row, col) = coord
        nTiles_across = free_tiles(data, row, col, "across")
        nTiles_down = free_tiles(data, row, col, "down")

        # Fill whichever direction has more tiles
        if (nTiles_across > nTiles_down): prev_direction = "across"
        else: prev_direction = "down"

        last_checked = (row, col)
        minBlocks += 1

    if (minBlocks > nCols): return nCols
    return minBlocks
```

```
# Given a set of data, find the number of blocks required
def __main__(data):

    # Look for parts where there is a whole column of shaded blocks
    # Partition them into smaller regions to find minBlocks required in each region
    head = 0
    blocks_required = 0
    for i in range(len(data)):
        if (data[i] == 0):

            # There were two "walls" in a row
            # Shift head forward
            if (i == head):
                head = i + 1
                continue

            blocks_required += minBlocks(data[head:(i - 1)])
            head = i + 1
            continue
    return blocks_required
```

1.3 Worst Case Complexity

The worst case complexity is $O(n^2)$. The algorithm has several helper functions.

The function *free tiles* looks for how many free "boxes" there are across or down from a given virtual "coordinate." This function is worst case $O(n)$ and best case $O(1)$.

The function *get next coord* gets the next virtual "coordinate" that is not shaded and we should look at. This function is worst case $O(n)$ and best case $O(1)$.
The function *minBlocks* gets the minimum amount of $1*n$ or $n*1$ boxes to shade in order to fill in our rectangle. This function assumes that none of the values in the given *data* is 0 - meaning there is no "wall" blocking us from filling in the entire rectangle. A while loop is run until there are no more coordinates to look at, maxing out at the number of columns which exist within the rectangle. Hence, at worst case, this is an $O(n^2)$ function and best case $O(n)$.

Our main function partitions the $n * m$ rectangle by separating whenever there is a "wall" by looping through the data once, hence a $O(n)$ function.

The algorithm works in a way where every search is linear except the one within minBlocks which is very close to $O(n)$ as we are guaranteed to have a smaller search space after each round. However, this causes the algorithm to have an overall worst case complexity of $O(n^2)$