COMP 257

ADVANCED ALGORITHMS

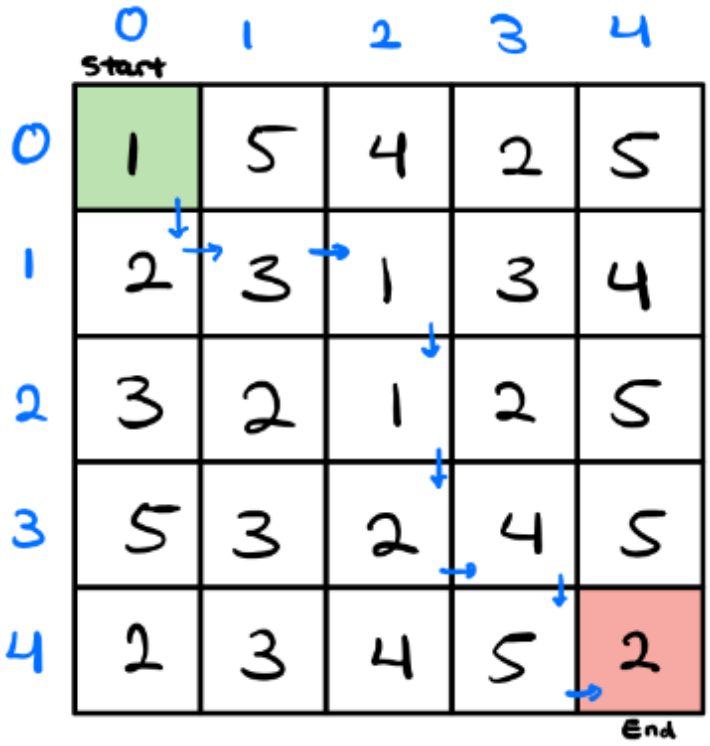
Hw 8 Project Proposal

Path Planning Algorithm for Weighted Grid

Problem Description

Given a grid of size m x n, with weighted values in each space, return value of the minimum cost path. The path must begin on the top right grid space (0,0) and end at the bottom left grid space (m,n). The path is only allowed to go in two directions: right or down.

Ex:



- input: grid[m][n] =

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

- m = 5, n = 5
- output: 21

Brute Force Approach

Big O Complexity: O(m*n)

Not sure about this complexity; realistically this should have the worst runtime, but I'm not sure how to calculate that based on my approach. The approach to this was a bit wonky as it required a stack. The idea is that the algorithm should find all possible paths then calculate cost of each path and choose the cheapest cost of all possible paths. My best solution was to traverse down the first column, summing all cells until there are no more rows left, then traverse and sum all cells to the right. Then on the next iteration continue summing in the same manner but traverse down the first column until n-1, go right one, down one, and right all the way. Then do this again but instead of right one, down one, go right 2, down 1 and then all the way right. Continue this pattern until all paths have been taken. However, this is a pretty complex strategy that requires recursion and I could not think of a way to code this in which all paths could be found without recursion. So I found a way to use a stack to hold values instead. I'm not sure if this solution would work, I still have to think it through a little more and probably work out some kinks. If you have any suggestions for this, please let me know. Or if you think this is a bad example for this project's purpose, let me know and I can attempt another proposal.

```
def (grid[][], m, n):
    costs = [][]
    sum = grid[0][0]
    stack = append coordinate (0,0)

    while stack is not empty
        currentCoord = last element in stack (in form x,y)
        remove last element in stack

        if currentCoord = (m,n)
            sum = sum + grid[currentCoord at x][currentCoord at y]
        else
            if currentCoord at x < m
                increment x value of currentCoord by 1
                append currentCoord to stack
            if currentCoord at y < n
                increment y value of currentCoord by 1
                append currentCoord to stack

    return sum
```

Greedy Algorithm Approach

Big O Complexity: O(m+n-2) = O(m+n)

This complexity is derived from the longest possible path length which can be found in many ways. The easiest way is by taking the size of the rows and the size of the columns and subtracting 2 because the first and last cells only have a path on one side (the first does not have

one coming in, the last does not have one coming out. Since 2 is a constant, we can simplify the runtime to O(n+m). This runtime is the most ideal of the 3, however, the solution is not always guaranteed to be correct. This will be explained in further detail when I am able to provide examples from running the actual algorithm.

```
def (grid[][], m, n):

    sum = grid[0][0]
    row = 0
    col = 0

    while not at end of path
        down = grid[col][row+1]
        right = grid[col+1][row]
        if down <= right
            sum = sum + down
            row++
        else
            sum = sum + right
            col++

    return sum
```

Dynamic Programming Approach

Big O Complexity: O(n+m+nm) = O(nm)

This algorithm's runtime is simplified to n*m because it is the largest term in n+m+n*m. n comes from the first for loop, n comes from the second for loop, and n*m comes from the nested for loop towards the end.

```
def (grid[][], m, n):
    dp[0][0] = grid[0][0]

    for row in range of n
        dp[0][row] = dp[0][row-1] + grid[0][row]

    for col in range of m
        dp[col][0] = dp[col-1][0] + grid[col][0]

    for row in range of n
        for col in range of m
            dp[col][row] = grid[col][row] + min( dp[col-1][row], dp[col][row-1] )

    return dp[m][n]
```