Conor Farrell

4/17/23

Homework 2

Unique Paths:

class Solution:

def uniquePaths(self, m: int, n: int) -> int:

# dynamic programming iteration solution, similar to mathematical also using math to calculate the number of paths

# This method has the total number of unique paths at the bottom right corner instead of top left but still gives the same answer

# initalize array with all 1s

dp = [[1 for \_ in range(n)] for \_ in range(m)]

for i in range(1, m):

for j in range(1, n):

#print(dp)

dp[i][j] = dp[i - 1][j] + dp[i][j - 1]

#print(dp)

return dp[m-1][n-1]

def uniquePathsRecursive(self, m: int, n: int) -> int:

#Recursive Solution

result = {}

# uses a dictionary/hash table to store the paths that it adds.

# similar to the iterative approach that it goes from top left to bottom right, the last key should have the most paths.

def uniquePathsHelper(row, col):

if (row == 0 or col == 0):

return 1

elif (row, col) not in result: # Avoid duplicate results

result[(row, col)] = uniquePathsHelper(row - 1, col) + uniquePathsHelper(row, col - 1)

#print(result)

return result[(row, col)]

return uniquePathsHelper(m - 1, n - 1)

res = Solution()

print(res.uniquePaths(3,7))

print(res.uniquePathsRecursive(3,7))

# Output:

# 28

# 28

Shortest path:

#Viterbi algorithm

import random

class Node:

top\_to\_top = 0

top\_to\_bottom = 0

bottom\_to\_top = 0

bottom\_to\_bottom = 0

def \_\_init\_\_(self, ttt, ttb, btt, btb) -> None:

self.top\_to\_top = ttt

self.top\_to\_bottom = ttb

self.bottom\_to\_top = btt

self.bottom\_to\_bottom = btb

class Viterbi:

points = [] # List[Node]

def \_\_init\_\_(self,size):

self.points = Viterbi.generate\_random\_array(self,size)

def generate\_random\_array(self,arr\_size):

arr = [] # generate array size of 100 with distances being from 1-10

for \_ in range(arr\_size):

arr.append(Node(random.randint(1,10),random.randint(1,10),random.randint(1,10),random.randint(1,10)))

return arr

def print\_points(self,points):

for node in points:

print("\*---", node.top\_to\_top,"---\*")

# print(" \\ /")

print(" \\ /")

print(" \\ /")

print(" \\ ",node.bottom\_to\_top)

print(" \\/")

print(" /\\ ")

print(" / ",node.top\_to\_bottom)

print(" / \\")

print(" / \\")

#print(" / \\")

print("\*---", node.bottom\_to\_bottom,"---\*")

def shortest\_path(self, points):

directions = [] # keep track of directions

total\_distance = 0

location = ""

# location is needed so we know which nodes to compare

for node in points:

if(location == 'top'):

if(node.top\_to\_top < node.top\_to\_bottom):

directions.append(node.top\_to\_top)

total\_distance += node.top\_to\_top

else:

directions.append(node.top\_to\_bottom)

total\_distance == node.top\_to\_bottom

location = 'bottom'

elif(location == 'bottom'):

if(node.bottom\_to\_bottom < node.bottom\_to\_top):

directions.append(node.bottom\_to\_bottom)

total\_distance += node.bottom\_to\_bottom

else:

directions.append(node.bottom\_to\_top)

total\_distance == node.bottom\_to\_top

location = 'top'

else:

if(node.top\_to\_top == min(node.top\_to\_top,node.top\_to\_bottom, node.bottom\_to\_top, node.bottom\_to\_bottom)):

location = 'top'

directions.append(node.top\_to\_top)

total\_distance += node.top\_to\_top

elif(node.top\_to\_bottom == min(node.top\_to\_top,node.top\_to\_bottom, node.bottom\_to\_top, node.bottom\_to\_bottom)):

location = 'top'

directions.append(node.top\_to\_bottom)

total\_distance += node.top\_to\_bottom

if(node.bottom\_to\_top == min(node.top\_to\_top,node.top\_to\_bottom, node.bottom\_to\_top, node.bottom\_to\_bottom)):

location = 'bottom'

directions.append(node.bottom\_to\_top)

total\_distance += node.bottom\_to\_top

if(node.bottom\_to\_bottom == min(node.top\_to\_top,node.top\_to\_bottom, node.bottom\_to\_top, node.bottom\_to\_bottom)):

location = 'bottom'

directions.append(node.bottom\_to\_bottom)

total\_distance += node.bottom\_to\_bottom

print(directions)

print(total\_distance)

graph = Viterbi(100)

#graph.print\_points(graph.points)

graph.shortest\_path(graph.points)

#graph.print\_points\_cool(graph.points)