Contents

ALGORITHMS

TANMAY AGARWAL

1 NQueens Problem

In this problem we first create a driver code and then in that we define a board which is initialised to all zeros of the desired size. This board is basically our chess board and we implement our methods on this. We then call a utility function and pass this board in it as well as the zero column

n the utility function we define the base case, which is that if the col number is ==n-1 we will return True which just means that all the queens have been placed After this we iterate for all the rows : if issafe the given board with row and col, then : we set $board[row][col] = 1$ and then recursively call the utility function again providing the board and the $col + 1$. If that is true we return true, else we immediately backtrack by setting $board = 0$ and return false. The reason we backtrack is the fact that then either of the recursive utility functions returns false, it means that if we place a queen in that row and col we can be attacked by other queen

Now the issafe function checks for three aspects : right(row=same, col iterated), lower diagonal(using zip), upper diagonal(using zip). Remember we check the left upper diagonals and not the right , because we check whether a previously placed queen can cause troubles and that is done using the left diagonals and the left rows

NOTE : In the gold table problem we check from the right in the gold table and in this we use the left diagonals and that both the problems follow a different approach

2 MazeSolve

In this Problem again we create a driver function in which we initialise a solution arrest with all 0's and also our driver function takes a maze as input. Now this driver code initialises another utility function which takes in maze,0,0 and sol. 0,0 are the starting row and column

Now the base case of the utility function is that if the co-ordinates are equal to the last row and last column, then we set that row's and col's solution to be 1 and return True Now we say if $issafe(maze, x, y)$: we check the base case for this loop which is that whether or not the solution array is already 1 for this particular coordinate. If yes, return False cause this is not the solution and it failed some previous loops. When the base is false, we initialise the $sol[co-ordintes] = 1$ and recursively call the utility function for $x+1, x-1, y+1, y-1$ which return True if the cases are true If any of it is false we backtrack setting $sol[co-ordintes] = 0$ and return false

The issue function checks that the co-ordinates are within the boundary ($0 < x < n$) and that the original maze at that co-ordinates contains a 1. If all of it is satisfied it returns True or else False

2.1 Difference between NQueens and MazeSolver

The first and major difference is the issafe functions. The next is that the base case of the utility functions is different and that in Nqueens, the utility function has a loop, which loops over all rows whereas the mazesolver has no loops as it is not needed Now in the issafe in utility, nqueens doesn't have a base case whereas mazesolver does

3 Gold Mine Problem

This problem is very simple and does not have any backtracking in it. In this problem we create only one function which takes in the original gold table and the MxN dimensions.

it initialises a goldtable with zeros. Now it makes 3 moves: right, rightUp and rightDown on the goldtable(not the of gold table). It then adds the max of these 3 moves with the value in the og gold table and stores it in the goldtable. This process is iterated over all rows and cols but col starts from last and ends at first

The res will be in the first col

4 Linked Lists

4.1 Remove Duplicates Algo

In this first before starting any loops we initialise a temporary head named head to be equal to (self.head). We do this so that we need to want to disturb the original head pointer as we need the og head pointer to stay in its place.

Now we start a loop saying if head.next != None : we initialise a pointer named pre on head and another pointer, temp on the next of head. Now we start another loop saying while temp.next != None : We check if the temp data == head data. If yes: we say that pre.next=temp.next. Then we increment temp and set pre=temp.

If not we just say that pre=temp and temp=temp.next and finally in the ending of this loop we also increment our head pointer

4.2 Odd Even Algo

In this first we need to find the length of the list because our final action depends upon whether the list is even in length or odd in length. Now we store the length of the list in a variable n. then we initialise a temporary head to be self.head(head=self.head) and we assign a temp variable to be head.next.

Then we immediately assign a prev variable to the temp(prev=temp). This prev pointer will point to the second node(which is the fist even node) of the list and hence we need to come back to it for connecting the remaining nodes after completing the odd connections.(Note : the remaining connections will be all even connections now). Now we start a loop saying while temp.next != None: we connect head.next to temp.next and then increment head by doing a head=temp and then increment temp(temp=temp.next). After this loop our temp will point to the last node and head to the last but one node

if the list is even : head will be pointing to last odd node in the end and temp to last even. Hence to complete the list we need to connect the final odd node to our starting even node which is the prev pointer. Hence we say if list is even : head.next=pre and temp.next=None(as temp will be pointing to the last even node)

else if the list is odd, temp is the last odd node. hence temp.next= pre and head.next = None

5 Unique Algorithms

In this section I will cover a few unique and different algorithms.

5.1 Max Houses listing - Google Kickstart

5.1.1 Problem Details

In this problem we are given an array and a budget. The array contains listing of the houses. Now we need to output how many total number of houses can be bought based on our current budget.

5.1.2 solution

we initialise an empty queue in the beginning. Then we start our i index from the last to go all the way to the first(Can be done from first to last too). We check whether the current value at i exceeds our budget or not. If yes, we skip this value of i and move on to the next one. If no, we declare an initial variable which is nothing but the initial value of the house we bought. Now we also start count=1 becasue we have bought the house that i currently receides on. Now we start j from 0 all the way through to the end. If i==j, we just skip that value of j. Now if the value of $initial + arr[j] < budget$, new initial value will be $initial + arr[j]$ and count will be incremented(as we have now procured this house). we continue this loop. Now when we come out of the j loop we append the count value to q. This will be done for all the count values in i loop. This is done so that we know how many houses can be bought when we start from different i values. In the end we take the maximum element from q and display it. If q is empty that means that we cannot buy any house, and we return 0

5.2 Blocks

5.2.1 Problem Details

In this problem we are given a list of dictionaries and an array which contains the places of interests to us. Each new dictionary represents a block for a place which have gyms, schools etc etc. Now we need tp write a code to find out which block is best situated for us to buy a house in given our interests.

5.2.2 Solution

We first define a function which takes the array of dictionaries and the interests as input. Now we initialise a queue with 0's to the length of the array. each index in our queue is an analogy to a block. Now we define count to use as an index into the queue. Initially looping over the array gives us dictionaries. Hence 'i' is a dictionary. Now for every key and value in that dictionaries, We check whether they match our interests, and if they do we also check whether that particular value is True. If Yes, we increment the value at that block's index in the queue(which is accessed using count). Finally count is incremented, Specifying that we are moving to the next block and finally the INDEX of max element of queue is returned. Index specifies the block number and hence index is returned. (We add 1 to the index so that its more human readable)

5.3 Window Sliding Algorithm

In this type of problem we basically initialise an analogue to a window which is a queue. As we will see in one of the problems, initialising a queue is not important. The size of this queue, or the window, will be explicitly specified by the user. Now one major difference is whatever main loop we run, the running of the loop condition will be n-k times where n is the length of the original array and k is the length of the window we are sliding over it. To move a window we have to remove the first element and add the i+k'th element. (i is the variable being iterated over n-k times)

5.3.1 Sum of a given subarray(Window)

In this problem we need to find the maximum subarray of the required size. The size will be explicitly specified by the user and the subarray is is treated as a window. Now instead of initialising a queue(window max) to the first k elements of the array, we initialise a variable which will be equal to the sum of the first k elements of the main array. Now we start a loop which will run n-k times and over each iteration we update the $windowmax$ by subtracting the first element of the array, arr[i] and adding the $i + k'th$ element. This is nothing but a double edged queue and our answer will be the maximum of all the iterations

5.3.2 Max Element in an array using Window sliding

The problem description is pretty self contained. Now we first initialise a queue with the first k elements of the array. Now we loop over n-k+1 times. (If we don't add +1 we will be skipping the last element). Now we start another loop

within our main loop which runs over the length of k and we find the max element. Then after we come out of the second loop we remove the first element using .pop(0) and add the i+k'th element. Before adding we check whether or not we are exceeding 'n' or not.

5.4 String starting with

In This algorithm we are given an array of strings and we need to find which of the strings in the array match the other given separate string. To to this we loop over the array using 'i' which gives us access to individual strings and then we loop over the length of the other string using j. We say that if i[j]==sti[j](If the two match) then increment a count. Else, We break because we want the words to match in order(The first letter of the string in the array must match the first letter of our explicit string). Then inside the second loop itself we check, if count==len(sti), which means all starting elements match, we append the array string in a queue and finally return the queue

6 Dynamic Programming

In This section I will be going through dynamic programming related algorithms, and explain how the tables are being declared(Their range) and also the basic logic behind each problem. Else the problems themselves are pretty self contained.

6.1 Least Common Subsequence

In this problem we define a function which takes two strings namely : X and Y. Now we find the lengths of X and Y respectively. Let the length of X be m and length of Y be n. We create a dp array of zero's like : lcs=[[0 for j in range(m+1)]0 for i in range(n+1)] **Notice that the outside loops will have the number of rows**. i.e : our array has n+1 rows and m+1 cols. Don't forget this.

Now the first question is why are we initialising to the lengths of m+1 and n+1 and not just m and n only. This is becuase for our dynamic programming solution to work, we need the dp array to have an extra row and column which will be initialised to all zeros. This extra row and column is the 0th row and the 0th column. Hence we start our for loops now.

The for loops are : First will be the row loop : for i in range(n+1) and next will be the cols loop : for j in range(m+1) Now we say that if i or j is 0, lcs[i][j]=0. This just shows that we are initialising the entire 0th row and 0th col to 0. This

step can be skipped(but it is not skipped in the code) by changing our for loop values to (1,n+1) and (1,m+1), because while creating the array itself we have initialised everything to zero.

The next step is pretty trivial. If the strings match, we fill the table with diagonal element +1 or if they don't we take the maximum of the upper and the left element.

Note that strings are compared with $x[i-1] == y[j-1]$. We have to keep i-1 and j-1 as, when we start comparing our i and j will be starting from 1(As the i and j's zero value is gone in assigning zeros in the table) but we want the strings to be compared from the start which is their zeroth index. Hence i-1 and j-1.

Also note that when we use i to index into X and j to index into y, make sure that i and j are the indexes looping over their indexed string lengths. Means as length of X is m, and i is used to index into x(x[i-1]), i must be in the m+1 for loop and same for Y.

6.2 Eggs Dropping

In this problem we are given number of floors : f, and the number of eggs : n. We use the same approach as LCS. We first define a dp array with 0's for f+1 rows and n+1 cols. Remember from above that the number of rows loops comes outside. Now we skip the part we did not in LCS. Instead of starting our loops from 0, we start from 1 as the 0'th row and cols have already been initialised with zero at the time of declaration.

The condition to update any of the cell is .. nothing. We just iteratively say that dp[i][j]dp[i-1][j-1]+dp[i-1][j]+1. The dp[i-1][j-1] stands for egg break than go down. The second dp[i-1][j] signifies that no egg break hence, go down and +1 is for the current floor count.

If $dp[i][j] >= f$, we return i

6.3 Minimum Cost Path

We are given a 3xs matrix named Cost. We need to output the minimum cost it will take to traverse the grid from the start point(0,0) to the given points(x,y) This problem is slightly different from The above two. In this problem also we define a dp array. But here we do not need an extra row and a column to be initialised to zero. Hence the size of array declaration is m and n only(and not m+1 and n+1 like previous two)(Here m and n are pre-defined sizes and global variables which specify number of columns and row). Now here what we do is the first element of the dp array will be the same as the first element of the cost

array(Starting Point). Then we explicitly first will out the first row, and then the first column and then the rest of the array.

So first we declare only one for loop going over the 0th row only. Hence for i in range(m) : dp[0][i]=dp[0][i-1]+cost[0][i] That is the 0th row is filled with the cost+ the previous dp col value.

Same for the 0th col. For i in range(m): dp[i][0]=dp[i-1][0]+cost[i][0]

Now as the 0th row and col are filled we start our nested for loop with 1,m and 1,n and fill the dp with the minimum of dp's left,right and diagonal adding the current cost to it.

We return dp[x][y]

6.4 Gold Mine

Even the gold mine problem is a dynamic programming problem which is similar to the previous one and not the first two, but it has already been discussed with great rigour in section 3. Please refer to that

6.5 Regular Expression Matching (Leetcode Hard)

Given an input string (s) and a pattern (p), implement regular expression matching with support for '.' and '*' where:

$\cdot$ Matches any single character

$*$ Matches zero or more of the preceding element.

The matching should cover the entire input string (not partial).

Now this problem lives up to its reputation of being a hard leet code problem. To solve this we use the similar array declaration of the first two problem(Which means we declare one extra row and column), To make things easier, let the columns represent the pattern, and rows be the string. Which means if the length of pattern in x and string is y, then cols=x+1 and rows=y+1. But in this problem instead of initialising to 0's we initialise to False, as the answer is boolean

NOTE : This is an emphasis of what has been said in the first problem itself. The ith element of the string or the pattern is represented as the i+1'th element in the dp array as it has m+1 and n+1, rows and cols. Hence in a loop(the loop begins from one obviously) the first string element will be i-1 and the previous string element will be i-2 and so on

After declaring the array we set the dp[0][0] to be True. Then we only iterate over the cols of the 0th row to check for * in the pattern string. If there is * then the dp array value at that cell will be equal to the second previous one. This means : for i in range(1,m): if pattern[i-1]=='*' : dp[0][i]=dp[0][i-2]

Next we start our nested loop from again..1. Now if the pattern is . or the string matches with the pattern element, dp of the current cell will be equal to the dp of the diagonal element

if the pattern is *, initially we assign the secind previous value of dp to the current value of dp. Then we check if the previous value of pattern(This will be j-2 index) is equal to . or the previous of pattern matches the current of string, then we assign the top value of dp to current value of dp.

In the end we return dp[m][n]

7 Matrix Based Problems

In this section we will cover a few of my favourite matrix based problems. But first we will talk in dept about how to apprach a specific matrix based problem

7.1 Approach Strategy

Firstly, we will have to find the dimensions of the given matrix. The number of rows is obtained by len(matrix) and the number of columns by len(matrix[0]). After obtaining these values we check whether the matrix is empty or not. If it is, we return a 0.

Next we start traversing the matrox by our usual double for loops. The traversing loops is problem specific. In some problem we need to pre-process the matrix before we start any kind of traversals.

Now coming to the crux of the matter that is our dfs function. A Depth first search, as the name suggest goes "Depth" First. Which means in the four directions of moving in a matrix (up, down, left, right) it goes up first completely until it reaches a terminating condition. Now the "terminating condition" is very important in our dfs function. Without it, the recursion will not terminate. The terminating condition is usually if the index is out of bounds followed by a problem dependent check. Now we need to be careful about what the function returns

To give a blueprint of a dfs function, it namely will consist of four parts :

1. Termination Condition

2. The recusive part (Moving along the directions)

3. The return part

7.1.1 Terminating Condition

This is a check which is used to terminate the recursion if it is violated. a general check includes the bound check which is usually : if $i < 0$ or $i \geq len(grid)$ or $j < 0$ or $j \geq len(grid[0])$ followed by a problem specific check

7.1.2 The Recursive Part

This usually consists of four function call : dfs(Up), dfs(down), dfs(Left), dfs(Right). But we condense it using a for loop : Let neighbours = (0,1),(0,-1),(1,0),(-1,0) and we iterate over these neighbours which gives us the four moves themselves.

We also have to change the matrix element to something else to specify that we have already visited a cell

NOTE : If we use the neighbours approach we cannot go and check the calls, that is : we cannot do up **and** down **and** .. For that we have to explicitly assign the functions four times and store their return values in variables, and we do this in a few problems too

7.1.3 Return Part

This is the heart of our recursive function. Generally has return declared in two places : One return if the check is True and another at the end of all the recursive functions. If we don't want to return anything and just modify things in place, return can be left blank.

For counting certain variables, if you want to do an individual count : specify the count inside the dfs function as a parameter and every time we recurse, we increment the count.

7.2 Number Of Islands Problem

In this problem we are given a grid of one and zero's. We need to find how many islands exist in the matrix. An island is depeicted by a series of ones. We start

this problem using our approach strategy, meaning we specify the base case of the empty matrix and then start our normal double for loops inside which we check : if the current element on the grid is equal to one or not. If it is, the we call our DFS function.

Now the task of the dfs function is to calculate the number of ones which are together. So at max it can return only a one. If it returns one it means there is a group of islands. So first we start with out bound check could with : If grid[i][j] != 1, then it will return a zero. If the check is false, the first thing we do is change this cell value to -1 or literally anything to specify we have visited this node. Now we call the recursive calls using the niegahbours approach and in the end return a 1. Which means at most it returns one only

7.3 Surrounding X's

In this problem we have X's and O's. We ignore the O's on the perimeter. Hence we explicitly traverse through the perimeter and change the O's and the O's it connects to, to something else. After we are done traversing the perimeter, we start our double loops and change the somethings values back to O and all the remaining O's to X's. Remember whenever we find an O along the perimeter we call the dfs function to find all the other O's it connects to as they all won't change

Now the dfs function is pretty trivial, we don't return anything as we just want to modify the matrix in place. We keep a preliminary check coupled with if grid[i][j] != "O" and after the check, we change this grid value and call the recursive functions using the neighbours approach. And that is it

7.4 Word Search

In this we are given a grid of words. We do our initial things and start our double for loops. After initialsing the double for loops, we check if the the current grid cell matches the first letter of the word **and** we call our Helper function

In this helper function we first check if the count is equal to the length of the sting. If it is, We return true. Now we define our check, in this check we include if grid[i][j] != word[count] we return False. We use the count itself to index in the word. Now we start our recursion, after changing the current grid cell to something else, and in each recursion call we increment the count value and in the end we return the value of every call "OR'ed"

8 Quick Foot Notes

In the special reverse function problem we assign an initialise j outside of all the loops because for swapping we do not need the index to start from the first position always.

But for problems like finding duplicates and triplets and rest, we initialise the j index inside the i loop because we need the j to start from 0 every time the i loop is incremented

Also we use while loops at many places because we would like to check a condition and then increment. For loops only check the condition for termination. While loops are used to increment after checking a few conditions. Inside a while loop, many if and else blocks can be placed to check multiple conditions and increment accordingly. In short use for loops when there is no condition required to check before incrementing(generally avoid for loops for array based problems)

From a 2D list to extract number of rows and cols : Rows : len(list) and cols : len(list[0])