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 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)