Q1 House robbing :-

Sol

Pseudo code :-

function maxpossible rob (array) :=

arraysize = size of array

maxrob = list [i'th element of it shows max amount he can rob without robbing two houses adjacently].

maxrob [1] = array [1]

maxrob [2] = array [2].

for I in 3 to size of array

maxrob[I] = max(maxrob [I-1], maxrob[I-2] + array [I])

return maxrob [size of array]

Proof of Correctness :-

at any index I, he has two possible options, does not rob the current house, instead, rob the previous house. ii) Rob the current house and house 2 houses back as well. He will choose the strategy which gives him maximum possible amount. We have implemented the same in our algo and hence algo is correct.

Running time complexity :-

Running time complexity of our algo is O(n).

Scanned with CamScanner

Q3

Sol.

Pseudo code:-

```
dp = [ ].
dp[0] = Infinite [If we are at 0, we can never take it to 1]
dp[1] = 0 [If we are at 1, we are already there].
dp[2] = 1 [If we are at 2, we can take it to 1 in one step, n/2]
dp[3] = 1 [If we are at 3, we can take it to 1 in one step n = n/3].
for I in 4 to n
    dp[I] = dp[I-1] + 1
    If (I%2 = 0]
        dp[I] = min [dp[I], 1 + dp[I/2]].
    If (I%3 = 0].
        dp[I] = min (dp[I], 1 + dp[I/3]].
print dp[n].
```

Proof of correctness:-

For any number I, we have three possible ways. 1) first take it I-1, then take it to 1

Scanned with CamScanner

ii) If it is divisible by 2, first take it /2, then to 1.

iii) If it is divisible by 3, first take to /3 then to 1.

we will choose path which take least no steps.

we have implemented the same in our algo and hence our algo is correct.

time complexity :-

time complexity of our algo is O(n).

Q5

Sol.

take input n (no of shops)
take input cost (costs of all three vegetables in n shops).
for i in 2 to n.
cost [i][0] += min [cost [i-1][1], cost [i-1][2])
cost [i][1] += minimum of ((cost[i-1][0], cost[i-1][2])
cost [i][2] += minimum of (cost [i-1][0], cost [i-1][1]).
print (minimum value of cost (n-1)).

Scanned with CamScanner

proof of correctness :-

let, we are planning to buy i'th vegetable from j'th shop. Then we will have to buy vegetables from previous shop (j-1) that are not i'th vegetable. so we will buy vegetable which costs us least.

i.e. cost[j][i] += min(cost[j-1][1], cost[j-1][2]).

we have implemented same in our algo and hence our algo is correct

time complexity :-

time complexity of our algo is O(n).

Q4

Sol. function maxprice(price).

n = size of array price

for i in 2 to n-1

for j in 1 to i-1

price(i) = max(price(i), price(j)+price(i-j)).

n = take input integer.

price = take input array of integers

ans = -1

maxprice(price).

for i in 1 to n-1:

ans = maximum of (ans, price(i)+price(n-i)).

print ans.

Scanned with CamScanner

proof of correctness :-

we have to sell the goldbar in its pieces. So, we need to determine the maximum value of these pieces. The maximum of these pieces could either be the value given itself or we can increase it by further dividing it into subpieces. we have done the same in our algo and hence our algo is correct.

time complexity :-

time complexity of our algo is $(O(n^2))$.

Q8

Sol

pseudo code :-

```
function computeLPSarray(string) :-
    n = size of string
    lps = array of size n, initialise all elements by 0.
    lps[0] = 0.
    i = 1
    length = 0.
    while i < n :-
        if string[i] = string[length] :-
            length += 1
            lps[i] = length
            i += 1.
        else
            if length ≠ 0
                length = lps[length - 1]
            else
                lps[i] = 0 ; i += 1.
```

Scanned with CamScanner

return lps.

function solution(String) :-

revstr = reverse of original string.

Concat = String + "$" + revstr

lps = computeLPS Array (concat).

return (size of String) - (last element of lps array)

String = take input String

print (solution (String)).

Proof of Correctness :-

Suppose we are given a String. and we have to make it palindrome by appending min number of char(s). If we somehow calculate the max size of suffix substring that is palindrome, then our answer will be (size of String) - (size of maximum length suffix substring that is palindrome). We have done the same in our algo and hence this is correct.

time Complexity :-

time Complexity of the algo is O(n).

Scanned with CamScanner

Q2 Award problem

Sol

```
function findMinAmount (arr) :-
    n = size of arr
    ans = 0.
    ansarr = []  (empty array).
    incLeft = [].
    incLeft [1] = 1.
    for i in 2 to n:
        if arr[i] > arr[i-1]. :
            incLeft [i] = incLeft [i-1] + 1.
        else :
            incLeft [i].
    incRight = nX[i]  (array of n element, all equal to 1).
    i = n - 1
    while (i > 0) :-
        if (arr[i] > arr(i+1))
            incRight (i) = incRight (i+1) + 1
        i -= 1
    ans = 0
    for i in 1 to n.
        ans += max (incLef(i) + incRight (i)).
    return ans

n = .  take input integer (size of array).
array = take input array of integers
print ( findMinAmount (array)).
```

Scanned with CamScanner

Proof of Correctness :-

for any integer n, ~~the one~~ let p be the number it is continously greater then from left and let q be the number it is continously greater then from right. So, the minimum no. of awards we have to give to integer n is $\max(p, q) + 1$

for example

(1, 3, 5, 4, 3, 2, 1).

So consider 5, it is greater then continously 2 elements from its left and continously 4 elements from its right. So we have to give it at least $\max(2, 4) + 1 = 5$ awards

time complexity :-

time complexity of the problem is $O(n)$.

Scanned with CamScanner

Q6 Pseudo Code

Sol

```
function finalMaxAmount(r, years, investement, f1, f2, initial_amount
                        scheme, amount) :-

    for I in ~~range~~ 1 to investement :-
        amount(1).append(initial_amount * r[1][1]).
        scheme[1].append(-1).

    for J in 2 to years :-
        amount.[J].append(-1).
        scheme[J].append(-1).
        for I_prev in ~~range~~ 1 to investement
            if (I_prev == I)
                temp = (amount(J-1)(1) - f1) x (r(i)(J))
                if (temp > amount[J][I]
                    scheme[J][I] = I_prev
                    amount[J][I] = temp.
            else
                temp = (amount[J-1](I_prev) - f2) * (r[i](J))
                if (temp > amount(J)(i))
                    scheme(J)(i) = I_prev
                    amount(J)(i) = temp.

    return maximum of (amount(years-1))

years = take input integer
investement = take input integer
r = take input double dimensional array
```

Scanned with CamScanner

f_1 = take input integer

f_2 = take input integer

scheme = empty array.

initial_amount = take input integer

amount = [] (empty array).

ans = finalmaxAmount(r, years, investement, f_1, f_2, initialamount)

final_invest_company = index of max value of amount(years).

SchemeAns = [] (empty array).

Y = years

while (final_invest_company ≠ -1)

~~if~~ SchemeAns.append(final_invest_company).

final_invest_company = scheme(Y)(finalinvestcom)

Y -= 1.

print(year, ans)

print(scheme).

Proof of correctness :- for any particular year, for any investement, we can compute its maximum value, ~~by~~ by assuming it previously was investing in ith scheme (i from 0 to no of investements) and then taking maximum value of it. In this way, we can compute the maximum value of these investements (in last year and maximum of which will give us the

Scanned with CamScanner

answer. We are doing the same in our algo and hence our algo is correct.

time complexity :-

time complexity of our algo is ~~to~~ $n \times n = O(n^2)$.

where n = no. of investements.

Q7

Sol. Pseudo Code :-

```
function lcs(string1, string2, i, j, ans) :=
    if (string1[i] = string2[j])
        ans+ = string1[i]
    ~~if (string1~~
    if (i = size of string 1 or j = size of string 2)
        return ans
    else
        return lcs(string1, string2, i+1, j+1, ans)
ans1 = "" [empty string]
ans2 = "" [empty string].
if [i ≠ size of string1]
    ans1 = lcs(string1, string2, i+1, j, ans1)
if (j ≠ size of string2]
    ans2 = lcs(string1, string2, i, j+1, ans2)
if ((size of ans) < (size of ans1))
        ans = ans1
if ((size of ans) < (size of ans2)
        ans = ans2
return ans
```

Scanned with CamScanner

String 1 = take input string

String 2 = take input string

ans = empty string

ans = lcs (string1, string2, 0, 0, ans).

print (ans).

proof of correctness :-

we can compute longest common substring recursively. let's assume i'th char of S1 and j'th char of S2 matches. Then we can add this char to till computed lcs and recursively compute remaining portion of lcs. If i'th char of S1 and j'th char of S2 does not match, we store till computed lcs somewhere and compute lcs of (S1(i+1 - - -) and S2(j - - -)) and (S1(i - - -) and S2(j+1 - - -) and maximum size of any of these will be our answer. We are doing the same in our algo and hence our algo is correct.

time complexity :-

time complexity of the algo is O(n * m)

where

n = size of string 1

m = size of string 2

Scanned with CamScanner