**QUESTION 1**

// QUESTION 1

**RECURSIVE**

Function countStructuresRecursive(n)

sum = 0

i = 0

while sum < n

i++

sum += i

End while

totalStructures = 0

for i = i to n

totalStructures += helperRecurse(n - i, i)

End for

Return totalStructures

End Function

Function helperRecurse(n, prevStage)

if n == 2 and n >= prevStage

Return 0

End if

if n == 3 and n < prevStage

Return 2

End if

if n < 3

Return 1

End if

totalStructures = 0

for i = min(prevStage - 1, n) to 2

totalStructures += helperRecurse(n - i, i)

End for

Return totalStructures

End Function

**TOP DOWN**

Function countStructuresTD(n)

Initialize memoization table with -1

totalStructures = 0

for i = 2 to n

totalStructures += helperRecurseTD(n - i, i)

End for

Output memoization table

Return totalStructures

End Function

Function helperRecurseTD(n, prevStage)

if n == 2 and n >= prevStage

Return 0

End if

if n == 3 and n < prevStage

Return 2

End if

if n < 3

Return 1

End if

if memo[n][prevStage] is not -1

Return memo[n][prevStage]

End if

totalStructures = 0

for i = min(prevStage - 1, n) to 2

totalStructures += helperRecurseTD(n - i, i)

End for

memo[n][prevStage] = totalStructures

Return totalStructures

End Function

**BOTTOM UP**

FUNCTION countStructuresBU(n)

for i = 0 TO N LOOP 1 = O(N)

if i > 0

memo[0][i] = 1;

if i > 1

memo[1][i] = 1;

if i > 2

memo[2][i] = 1;

if i > 3

memo[3][i] = 2;

else

memo[3][i] = 1;

END if

END if

END if

END if

END for

for i = 4 TO N LOOP 2 = O(N)

for j = 4 TO N LOOP 3 = O(N)

for k = 3 TO min(j - 1, i) LOOP 4 = O(N)

memo[i][j] += memo[i - k][k];

END for

END for

END for

return memo[n][n];

**Bottom UP Complexity**

**LOOP 1 + (LOOP 2 \* LOOP 3 \* LOOP 4) = O(N) + O(N) \* O(N) \* O(N) = O(N ^ 3)**

**QUESTION 2**

**RECURSIVE**

function strategicValue(i, j):

cumulativeSum = new Array(j + 1)

for k from j down to i:

cumulativeSum[k] = depot[k] + (k + 1 <= j ? cumulativeSum[k + 1] : 0)

sum = 0

for k from i + 1 to j:

sum += depot[k - 1] \* cumulativeSum[k]

return sum

Function minStrategicValue(n, m, curr, start, end)

If m == 0

Return strategicValue(curr, end)

Else if curr == n

Return INT16\_MAX

Else

attack\_here = strategicValue(start, curr) + minStrategicValue(n, m - 1, curr + 1, curr + 1, end)

skip\_attack = minStrategicValue(n, m, curr + 1, start, end)

If attack\_here == INT\_MIN

attack\_here = INT\_MAX

End if

Return min(attack\_here, skip\_attack)

End if

End Function

**TOPDOWN**

Function minStrategicValueTD(n, m, curr, start, end)

If m == 0

Return strategicValue(curr, end)

Else if curr == n

Return INT16\_MAX

Else if memo[m][curr][start] != -1

Return memo[m][curr][start]

Else

attack\_here = strategicValue(start, curr) + minStrategicValueTD(n, m - 1, curr + 1, curr + 1, end)

skip\_attack = minStrategicValueTD(n, m, curr + 1, start, end)

If attack\_here == INT\_MIN

attack\_here = INT\_MAX

End if

memo[m][curr][start] = min(attack\_here, skip\_attack)

Return memo[m][curr][start]

End if

End Function

**BOTTOM UP**

// Base case initialization

for curr from 0 to n: **LOOP 1 = O(N^2)**

for start from 0 to n:

dp[curr][0][start] = strategicValue(curr, end)

for attack from 0 to m: **LOOP 2 = O(N\*M)**

for start from 0 to n:

dp[n][attack][n] = 0

// Calculate cumulative sums

for i from 0 to n-1: **LOOP 3 = O(N^2)**

for k from i to n-1:

cumulativeSum[i][k] = depot[k] + (k > i ? cumulativeSum[i][k - 1] : 0)

// Preprocessing for dp2

for start from 0 to n-1: **LOOP 4 = O(N^2)**

for curr from start+1 to n-1:

dp2[curr][start] = dp2[curr - 1][start] + depot[curr] \* cumulativeSum[start][curr - 1]

// Fill the DP table

for curr from n-1 down to 0: **LOOP 5 = O(M\*N^2)**

for remainingAttacks from 1 to m:

for start from 0 to curr:

attack\_here = dp2[curr][start] + dp[curr + 1][remainingAttacks - 1][curr + 1]

skip\_attack = dp[curr + 1][remainingAttacks][start]

dp[curr][remainingAttacks][start] = min(attack\_here, skip\_attack)

// Return result

return dp[0][m][0]

**L1 + L2 + l3 + L4 + L5 = O(M\*N^2) + O(N^2) + O(N^2) + O(N^2) + O(N\*M) = O(M\*N^2)**

**QUESTION 3**

// QUESTION 3

dividearrayMNPN(arr) :

hash = create empty unordered set

n = size of arr

for i from 0 to n - 1: **LOOP 1 = O(N)**

hash.insert(arr[i])

mnpn = 0

while mnpn is found in hash : **LOOP 2 = O(N)wc**

mnpn++

output "mnpn: " + mnpn

count = 0

index = -1

unique = create empty unordered set

for i from 0 to n - 1: **LOOP 3 = O(N)**

if arr[i] < mnpn and arr[i] is not in unique :

increment count

if count is equal to mnpn :

set index to(i + 1)

break

insert arr[i] into unique

count = 0

unique1 = create empty unordered set

for i from index to n - 1: **LOOP 4 = O(N)**

if arr[i] < mnpn and arr[i] is not in unique1 :

increment count

insert arr[i] into unique1

if count is equal to mnpn :

output "2"

output "[" + 1 + ", " + index + "]"

output "[" + (index + 1) + ", " + n + "]"

else:

output "-1"

All the loops are independent of each other and so the final complexity will be: O(n)

**QUESTION 4**

// QUESTION 4

maxTeamPower(M, N, grid) :

if M == 1 :

maxPower = 0

for i from 0 to N - 1 : **LOOP 1 = O(N)**

if grid[0][i] > maxPower:

maxPower = grid[0][i]

return maxPower

dp = allocate memory for 2D array of size M x N

for i from 0 to M - 1: **LOOP 2 = O(N)**

dp[i][0] = grid[i][0]

mmax = 0

prevmax = 0

for i from 0 to M - 1: **LOOP 3 = O(N)**

if dp[i][0] > mmax:

prevmax = mmax

mmax = dp[i][0]

else if dp[i][0] > prevmax:

prevmax = dp[i][0]

int maxPower = 0;

int prevmaxPower = 0;

int lastcolmax = 0;

int lastcolprevmax = 0;

for j from 1 to N - 1: **LOOP 4&5 = O(N\*M)**

for i from 0 to M - 1 :

if dp[i][j - 1] == mmax :

if lastcolmax != dp[i][j - 1] and lastcolmax + grid[i][j] > prevmax + grid[i][j]:

dp[i][j] = lastcolmax + grid[i][j]

else if lastcolmax == dp[i][j - 1] and lastcolprevmax + grid[i][j] > prevmax + grid[i][j]:

dp[i][j] = lastcolprevmax + grid[i][j]

else :

dp[i][j] = prevmax + grid[i][j]

else:

if lastcolmax != dp[i][j - 1] and lastcolmax + grid[i][j] > mmax + grid[i][j]:

dp[i][j] = lastcolmax + grid[i][j]

else :

dp[i][j] = mmax + grid[i][j]

maxPower = 0

prevmaxPower = 0

for i from 0 to M - 1: **LOOP 4&6 = O(N\*M)**

if dp[i][j] > maxPower:

prevmaxPower = maxPower

maxPower = dp[i][j]

else if dp[i][j] > prevmaxPower:

prevmaxPower = dp[i][j]

if maxPower > mmax:

lastcolmax = mmax

mmax = maxPower

if prevmaxPower > prevmax:

lastcolprevmax = prevmax

prevmax = prevmaxPower

maxPpower = 0

for i from 0 to M - 1: **LOOP 7 = O(M)**

maxPpower = max(maxPpower, dp[i][N - 1]);

deallocate memory for dp

return maxPpower

END

Total Complexity:

O(n) + O(n) + O(n) + O(n\*m) + O(n\*m) + O(n) = O(N\*M)