Design and Analysis of Algoriľhms

Q1

Pseudo Code:

Function countCombinations(n, prevHeight, memo):

if n equals 0: return 1

if n is less than 0 or prevHeight is 0: return 0

if memo[n][prevHeight] is not -1: return memo[n][prevHeight]

combinations = 0

for height from 1 to min(n, prevHeight):

combinations += countCombinations(n - height, height - 1, memo) memo[n][prevHeight] = combinations

return combinations

Function call(n):

Initialize a 2D memoization array memo[n+1][n] with all values set to -1 result = countCombinations(n, n - 1, memo)

Free memory used by memo return result

Function main():

Read integer n from user input

Print the result of call(n) which gives the number of combinations for N = n

6sympľoľic Timg Complg…iľy 6nalysis

* The time complexity primarily depends on the number of subproblems solved and the depth of recursion needed for each subproblem.
* Each function call to countCombinations computes combinations for a given n and prevHeight. The recursion explores each possible height from 1 to min(n, prevHeight). Therefore, in the worst case (ignoring memoization), the recursive calls can be numerous.
* Memoization significantly optimizes this by ensuring that each combination of n and prevHeight is computed exactly once. After memoization, the complexity becomes proportional to the size of the memoization table, which is roughly O(n^2).
* Given n as the number of blocks and prevHeight which can also be a maximum of n, each state (n, prevHeight) is computed once with a loop running up to n. This results in a

worst-case scenario of 𝑂(𝑛^2).

Q2

Psgudo Codg:

CONSTANT MX = 1001

DECLARE vals[MX] // Array to hold values

DECLARE dp[MX][MX] // DP table for memoization

FUNCTION calcInt(s, e)

sum ← 0

FOR i FROM s TO e - 1 FOR j FROM i + 1 TO e

sum ← sum + vals[i] \* vals[j] RETURN sum

FUNCTION minInt(s, e, sp)

IF s >= e THEN RETURN 0

IF sp = 0 THEN

RETURN calcInt(s, e)

IF dp[s][sp] ≠ -1 THEN RETURN dp[s][sp]

minVal ← calcInt(s, e)

FOR p FROM s TO e - 1

intSplit ← minInt(s, p, 0) + minInt(p + 1, e, sp - 1) minVal ← MIN(minVal, intSplit)

dp[s][sp] ← minVal RETURN minVal

MAIN

DECLARE n, ms // n for size, ms for max splits WHILE READ n, ms AND (n ≠ 0 OR ms ≠ 0) DO

FOR i FROM 0 TO n - 1

READ vals[i]

MEMSET dp TO -1 // Reset the DP table

PRINT "Min interaction cost: " + minInt(0, n - 1, ms) END WHILE

* Spacg of Sub-problgms and Opľimaliľy Subsľrucľurg

Sub-problems Definition:

* Parameterization: Sub-problems are defined by three parameters (s, e, sp), where:
* s (start): The starting index of the subarray under consideration.
* e (end): The ending index of the subarray.
* sp (splits): The number of allowed splits within the subarray to minimize the interaction cost.

How the Overall Goal is Reflected:

* The main goal is to minimize the interaction cost between elements of the array. It may be split into array parts, so each part of the array does not interact with another part of the array. In other words, it tries to calculate min interaction cost between the elements of the array with indices s to e with up to sp splits.

Justification of Optimality Substructure:

* The solution to a sub-problem (s, e, sp) can be optimally built from the solution of smaller sub-problems. Specifically, the optimal solution for (s, e, sp) can be found by considering all possible positions to place one split and then combining optimal solutions of the resulting two sub-problems. This property, as mentioned, justifies the use of dynamic programming, as optimal solutions to smaller subproblems can be reused in the construction of solutions for larger subproblems.
* Pgcursivg Iormula and Basg Casgs

Recursive Formula:

𝑑𝑝[𝑠][𝑠𝑝] = 𝑚𝑖𝑛𝑖 = 𝑠𝑒 − 1(𝑐𝑎𝑙𝑐𝐼𝑛𝑡(𝑠, 𝑖) + 𝑚𝑖𝑛𝐼𝑛𝑡(𝑖 + 1, 𝑒, 𝑠𝑝 − 1))

Where:

* calcInt(s, i) calculates the interaction cost of the segment from s to i.
* minInt(i+1, e, sp-1) computes the minimum interaction cost from i+1 to e with sp-1 splits.

Base Cases:

* When s>=e, the interaction cost is 0 because the segment has no valid pairs.
* When sp = 0, the interaction cost is calculated directly for the segment [s, e] without any splits.
* Boľľom-up Implgmgnľaľion Ordgr
* Initialization: Start by calculating calcInt(s, e) for all subarray segments where e >= s. This sets up the base case for sp = 0.
* Iterative Calculation: For each sp from 1 up to the maximum allowed splits, fill the dp table by using previously computed values for smaller splits and segments.
* Order of Solving Sub-problems: Begin with the smallest subarrays and fewest splits, gradually increasing the size of the subarrays and number of splits. For each sp, calculate values for all subarray sizes.

K. 6sympľoľic Punning Timg

Running Time:

* The computation for each sub-problem involves iterating over potential split points, which is 𝑂(𝑒−𝑠).
* O(e−s) for each sub-problem. Given that this needs to be done for each combination of s, e, and sp, the overall running time is 2 , where n is the array size and m is the maximum number of splits.

Q3

Psgudo Codg:

FUNCTION findMin(arr, size, present)

FOR i FROM 0 TO size - 1

present[arr[i]] += 1

min ← INT\_MAX

FOR i FROM 0 TO size - 1

IF present[i] == 0 min ← i BREAK

RETURN min

FUNCTION chkSegment(present, index)

FOR i FROM index - 1 DOWN TO 0

IF present[i] < 2 RETURN -1

RETURN 1

MAIN

READ size INITIALIZE arr[size]

INITIALIZE present[size] TO 0 READ elements INTO arr

index ← findMin(arr, size, present) IF chkSegment(present, index) == -1

PRINT "Impossible: -1" ELSE

startInd ← 1

endInd ← 0

FOR i FROM 0 TO size - 1 AND j FROM 0 TO index IF arr[i] == j

j++ endInd ← i

endInd++

PRINT "Segments: 2"

PRINT "Segment 1: " + startInd + " - " + endInd PRINT "Segment 2: " + (endInd + 1) + " - " + size

DELETE arr DELETE present

6sympľoľic Timg Complg…iľy 6nalysis

* findMin Function:
* The loop to count occurrences runs in O(n), where n is the size of the array.
* The second loop that finds the minimum missing value also runs in O(n).
* Total: O(n).
* chkSegment Function:
* The loop checks each element up to index-1 for a minimum presence of twice, which in the worst case can be O(n).
* Main Function:
* Initialization and input reading are both O(n).
* The findMin and chkSegment functions are called once each, adding up to 2\*O(n)

≈ O(n).

* The loop that determines the segments runs in O(n).

Q4

Psgudo Codg:

FUNCTION max\_power(hunters, dp, numRows, numColumns)

FOR i FROM 0 TO numRows - 1: dp[i][0] = hunters[i][0]

FOR j FROM 1 TO numColumns - 1: highest = INT\_MIN secondHighest = INT\_MIN

FOR i FROM 0 TO numRows - 1: IF dp[i][j - 1] > highest:

secondHighest = highest highest = dp[i][j - 1]

ELSE IF dp[i][j - 1] > secondHighest: secondHighest = dp[i][j - 1]

FOR k FROM 0 TO numRows - 1: IF dp[k][j - 1] == highest:

dp[k][j] = MAX(dp[k][j - 1], hunters[k][j] + secondHighest) ELSE:

dp[k][j] = MAX(dp[k][j - 1], hunters[k][j] + highest) maxPower = INT\_MIN

FOR i FROM 0 TO numRows - 1:

IF dp[i][numColumns - 1] > maxPower: maxPower = dp[i][numColumns - 1]

RETURN maxPower

MAIN:

READ numRows, numColumns

IF numColumns <= 0 OR numRows <= 0: PRINT "Invalid Input"

RETURN

IF numRows == 1:

ALLOCATE hunters[numRows][numColumns] PRINT "Enter values for hunters array:"

FOR i FROM 0 TO numRows - 1:

FOR j FROM 0 TO numColumns - 1: READ hunters[i][j]

maxPower = INT\_MIN

FOR i FROM 0 TO numColumns - 1: IF hunters[0][i] > maxPower:

maxPower = hunters[0][i]

PRINT "Maximum power level:", maxPower

FOR i FROM 0 TO numRows - 1: FREE hunters[i]

FREE hunters

ELSE:

ALLOCATE hunters[numRows][numColumns] ALLOCATE dp[numRows][numColumns]

PRINT "Enter values for hunters array:" FOR i FROM 0 TO numRows - 1:

FOR j FROM 0 TO numColumns - 1: READ hunters[i][j]

PRINT "Maximum power level:", max\_power(hunters, dp, numRows, numColumns)

// Free memory

FOR i FROM 0 TO numRows - 1: FREE hunters[i]

FREE dp[i] FREE hunters FREE dp

6sympľoľic Timg Complg…iľy 6nalysis

* Time Complexity: O(numRows \* numColumns) for the initialization and O(numRows

\* numColumns) for the core dynamic programming loop, leading to an overall time complexity of O(numRows \* numColumns). The nested loops through each row for each column (to find highest and second highest, and then to calculate values) essentially make the complexity linear in terms of the total number of elements in the matrix.