CS700
Algorithms and Complexity
Assignment 3

ANAND M K
242CS008

ABHIJITH C
242CS003

Department of Computer Science and Engineering
NITK Surathkal

Problem 1: Dance partners

Problem Statement: You are pairing couples for a very conservative formal ball. There are n men and m women, and you know the height and gender of each person there. Each dancing couple must be a man and a woman, and the man must be at least as tall as, but no more than 3 inches taller than, his partner. You wish to maximize the number of dancing couples given this constraint.

Pseudocode

```
  Function calculate_max_pairs(men[], num_men, women[], num_women):
  Sort men[] using merge_sort
  Sort women[] using merge_sort


// Initialize pointers and matches count

  i = 0, j = 0, matches = 0
  Allocate memory for matched_pairs[]



// Iterate through men and women to find valid pairs

  While i < num_men and j < num_women:
  If men[i] < women[j]:
  i++  // Move to the next man
  Else If men[i] > women[j] + 3:
  j++  // Move to the next woman
  Else:
  matched_pairs[matches] = (men[i], women[j])
  matches++
  i++, j++  // Move to the next man and woman


// Return the total number of matches

  Return matches
```

Time Complexity Analysis

1. Sorting the Arrays

The ‘merge_sort‘ algorithm is used to sort both the `men[]` and `women[]` arrays. The time complexity for sorting each array is:

$$O(n \log n) \quad \text{where } n \text{ is the size of the array.}$$

Sorting both arrays together takes:

$$O(\text{num\_men} \cdot \log(\text{num\_men})) + O(\text{num\_women} \cdot \log(\text{num\_women})).$$

2. Matching the Pairs

The while loop iterates through both arrays to find valid pairs. Since each element is processed once, the time complexity is:

$$O(\text{num\_men} + \text{num\_women}).$$

3. Overall Time Complexity

The overall time complexity is dominated by the sorting step:

$$O(\text{num\_men} \cdot \log(\text{num\_men}) + \text{num\_women} \cdot \log(\text{num\_women})).$$

For cases where $\text{num\_men} \approx \text{num\_women} = n$, this simplifies to:

$$O(n \log n).$$

Problem 2: Homework grade maximization

Problem Statement: In a class, there are n assignments. You have H hours to spend on all assignments, and you cannot divide an hour between assignments, but must spend each hour entirely on a single assignment. The I'th hour you spend on assignment J will improve your grade on assignment J by B[I, J], where for each J, B[1, J] $\geq$ B[2, J] $\geq$ ... $\geq$ B[H, J]$\geq$ 0. In other words, if you spend h hours on assignment J, your grade will be $\sum$ i=1 B[I, J] and time spent on each project has diminishing returns, the next hour being worth less than the previous one. You want to divide your H hours between the assignments to maximize your total grade on all the assignments. Give an efficient algorithm for this problem.

Pseudocode

```
Function maximizeGrades(n, H, B, B_sizes, totalGrade, allocation):
Initialize a priority queue pq
Initialize hoursSpent array for each assignment

// Initialize the priority queue with the first grade of each assignment
For each assignment i from 0 to n-1:
If grades available for assignment i:
Push the first grade to the priority queue

totalGrade = 0
allocatedHours = 0

// Allocate hours while there are hours left and the priority queue is not empty
While H > 0 and pq.size > 0:
current = pop(pq)  // Get the highest grade
assignmentIdx = current.assignmentIdx
totalGrade += current.grade
allocation[allocatedHours++] = assignmentIdx
H -= 1
hoursSpent[assignmentIdx] += 1

// Push the next grade for the current assignment if available
If hoursSpent[assignmentIdx] < B_sizes[assignmentIdx]:
next = B[assignmentIdx][hoursSpent[assignmentIdx]]
Push(pq, next)

Free memory for the priority queue
```

Time Complexity Analysis

The time complexity is analyzed based on the heap operations and the while loop.

1. Sorting and Heap Initialization:

- The heap is initialized by pushing the first grade from each assignment into the priority queue. - Each push operation takes $O(\log n)$, where n is the number of assignments. - The total time complexity for initializing the heap is:

$$O(n \log n)$$

2. Heap Operations in the While Loop:

- The while loop runs for H iterations (one for each available hour). For each iteration: - A pop operation to extract the highest grade, which takes $O(\log n)$. - A push operation to insert the next grade for the assignment, which also takes $O(\log n)$. - The total time complexity for the while loop is:

$$O(H \log n)$$

3. Overall Time Complexity:

The overall time complexity is dominated by the heap operations:

$$O(n \log n + H \log n)$$

Where n is the number of assignments and H is the total number of hours.

If H is large relative to n, the complexity will be dominated by $O(H \log n)$. If n is large, the complexity will be dominated by $O(n \log n)$.

Problem 3: Activity-Selection Problem

Problem Statement: Given a set S = 1, 2, ..., n of n proposed activities, with a start time si and a finish time fi for each activity i, select a maximum-size set of mutually compatible activities.

Pseudocode

```
Function activitySelection(activities[], n):
Sort activities[] by finish time in increasing order

Print "Selected Activities:"

lastSelectedIndex = 0  // The first activity is always selected
Print "Activity " + activities[lastSelectedIndex].id + " (Time: "
+ activities[lastSelectedIndex].start + " to " +
 activities[lastSelectedIndex].finish + ")"

For i = 1 to n-1:
If activities[i].start >= activities[lastSelectedIndex].finish:
Print "Activity " + activities[i].id + " (Time: "
+ activities[i].start + " to " + activities[i].finish + ")"
lastSelectedIndex = i  // Update the last selected activity index
End Function
```

Time Complexity Analysis

The time complexity is analyzed based on the heap operations and the while loop.

1. Sorting and Heap Initialization:

- The heap is initialized by pushing the first grade from each assignment into the priority queue.
- Each push operation takes $O(\log n)$, where n is the number of assignments.
- The total time complexity for initializing the heap is:

$$O(n \log n)$$

2. Heap Operations in the While Loop:

- The while loop runs for H iterations (one for each available hour). For each iteration:
  - A pop operation to extract the highest grade, which takes $O(\log n)$.
  - A push operation to insert the next grade for the assignment, which also takes $O(\log n)$.
- The total time complexity for the while loop is:

$$O(H \log n)$$

3. Overall Time Complexity:

The overall time complexity is dominated by the heap operations:

$$O(n \log n + H \log n)$$

Where n is the number of assignments and H is the total number of hours.

If H is large relative to n, the complexity will be dominated by $O(H \log n)$. If n is large, the complexity will be dominated by $O(n \log n)$.

Problem 4

Problem Statement: Alice wants to throw a party and is deciding whom to call. She has n people to choose from, and she has made up a list of which pairs of these people know each other. She wants to pick as many people as possible, subject to two constraints: at the party, each person should have at least five other people whom they know and five other people whom they don't know. Give an efficient algorithm that takes as input the list of n people and the list of pairs who know each other and outputs the best choice of party invitees. Give the running time in terms of n.

Pseudocode

```
Function initializeGraph(Graph *graph, int n):
graph->n = n
For i = 0 to n - 1 do:
graph->nodes[i].id = i
graph->nodes[i].neighborCount = 0

Function addEdge(Graph *graph, int person1, int person2):
graph->nodes[person1].neighbors[graph->nodes[person1].neighborCount++] = person2
graph->nodes[person2].neighbors[graph->nodes[person2].neighborCount++] = person1

Function filterNodes(Graph *graph, bool invitees[MAX_PEOPLE]):
changed = true
While changed do:
changed = false
For i = 0 to graph->n - 1 do:
If invitees[i] is false then continue
neighbors = countRelationships(graph->nodes[i], graph, invitees, true)
nonNeighbors = countRelationships(graph->nodes[i], graph, invitees, false)
If neighbors < 5 or nonNeighbors < 5 then:
invitees[i] = false
changed = true

Function countRelationships(Node *person, Graph *graph,
 bool invitees[MAX_PEOPLE], bool countKnows):
count = 0
For i = 0 to graph->n - 1 do:
If i == person->id or invitees[i] is false then continue
knows = false
For j = 0 to person->neighborCount - 1 do:
If person->neighbors[j] == i then:
knows = true
break
If (countKnows and knows) or (not countKnows and not knows) then:
count++
Return count
```

Time Complexity Analysis

Function: `initializeGraph`

The `initializeGraph` function initializes a graph with n nodes. It iterates through all nodes and sets their initial values.

$$\text{Time Complexity: } O(n)$$

Where n is the number of nodes in the graph.

Function: `addEdge`

The `addEdge` function adds an edge between two nodes. Each addition involves updating two lists, and since the neighbor lists are bounded by a constant factor $MAX\_CONNECTIONS$, this operation is constant time.

$$\text{Time Complexity: } O(1) \text{ per edge}$$

If there are m edges, the overall complexity is $O(m)$, but in the worst case (complete graph), this is $O(n^2)$.

Function: `filterNodes`

The `filterNodes` function iterates over all nodes and, for each node, calls the `countRelationships` function twice (for neighbors and non-neighbors). The outer loop runs n times, and the `countRelationships` function runs in $O(n)$ time.

$$\text{Time Complexity: } O(n^2)$$

Where n is the number of nodes in the graph.

Function: `countRelationships`

The `countRelationships` function iterates over all nodes and checks if they are neighbors or non-neighbors of the given person. It runs a nested loop where, in the worst case, both loops iterate n times.

$$\text{Time Complexity: } O(n^2)$$

Where n is the number of nodes in the graph.

Overall Time Complexity

The dominant time complexity in the program is $O(n^2)$ due to the `filterNodes` function, which calls `countRelationships` twice per node. Therefore, the overall time complexity is:

$$\text{Overall Time Complexity: } O(n^2)$$

Question 5

Problem Statement:

A contiguous subsequence of a list S is a subsequence made up of consecutive elements of S. For instance, if S is 5, 15, -30, 10, -5, 40, 10, then 15, -30, 10 is a contiguous subsequence but 5, 15, 40 is not. Give a linear-time algorithm for the following task: Input: A list of numbers, a1 , a2 , . . . , an . Output: The contiguous subsequence of maximum sum (a subsequence of length zero has sum zero). For the preceding example, the answer would be 10, -5, 40, 10, with a sum of 55. (Hint: For each j $\in$ 1, 2, . . . , n, consider contiguous subsequences ending exactly at position j.)

Pseudocode

```
Function findMaxSumSubsequence(array[], size):
maxSubseq.startIndex = 0
maxSubseq.endIndex = -1  // Indicates no subsequence yet
maxSubseq.maxSum = 0

currentSubarraySum = 0
maxSumSoFar = 0
potentialStartIndex = 0

For currentIndex = 0 to size - 1 do:
currentSubarraySum += array[currentIndex]  // Add the current element
to the subarray sum

If currentSubarraySum < 0 then:
currentSubarraySum = 0  // Reset the current subarray sum
potentialStartIndex = currentIndex + 1  // Start a new subarray from the next
element
Else if maxSumSoFar < currentSubarraySum then:
maxSumSoFar = currentSubarraySum  // Update the maximum sum found so far
maxSubseq.startIndex = potentialStartIndex  // Update the start index of the
subsequence
maxSubseq.endIndex = currentIndex  // Update the end index of the subsequence
maxSubseq.maxSum = maxSumSoFar  // Update the maximum sum

Return maxSubseq  // Return the result containing the maximum sum subsequence
```

Time Complexity Analysis

Function: `findMaxSumSubsequence`

The `findMaxSumSubsequence` function iterates through the entire array once. During each iteration, the algorithm updates the sum of the current subarray

and checks if a new maximum subsequence is found. The algorithm only performs constant-time operations for each element, resulting in a linear time complexity.

$$\text{Time Complexity: } O(n)$$

Where n is the size of the input array.

Space Complexity

The space complexity is determined by the space required for the input array and the result structure. - The input array takes $O(n)$ space. - The result structure, which stores the start and end indices of the subsequence and the maximum sum, takes constant space $O(1)$. Thus, the overall space complexity is:

$$\text{Space Complexity: } O(n)$$

Question 6

Problem Statement: You are going on a long trip. You start on the road at mile post 0. Along the way there are n hotels, at mile posts a1 ¡ a2 ¡ · · · ¡ an , where each ai is measured from the starting point. The only places you are allowed to stop are at these hotels, but you can choose which of the hotels you stop at. You must stop at the final hotel (at distance an), which is your destination.Algorithms You'd ideally like to travel 200 miles a day, but this may not be possible (depending on the spacing of the hotels). If you travel x miles during a day, the penalty for that day is (200 - x)2 . You want to plan your trip so as to minimize the total penalty that is, the sum, over all travel days, of the daily penalties. Give an efficient algorithm that determines the optimal sequence of hotels at which to stop.

Pseudocode

```
Function findOptimalStops(hotelCount, hotelDistances):
allStops[0] = 0  // Starting point at mile post 0
For i = 0 to hotelCount - 1 do:
allStops[i + 1] = hotelDistances[i]
hotelCount += 1  // Include the starting point in total count

Initialize minPenalty[hotelCount] to INT_MAX  // To store minimum
penalty for each stop
Initialize previousStop[hotelCount] to -1  // To store the previous
stop (for path reconstruction)
minPenalty[0] = 0  // No penalty at starting point

For currentStop = 1 to hotelCount - 1 do:
For earlierStop = 0 to currentStop - 1 do:
distanceTravelled = allStops[currentStop] - allStops[earlierStop]
penaltyForDay = (200 - distanceTravelled) * (200 - distanceTravelled)
totalPenalty = minPenalty[earlierStop] + penaltyForDay

If totalPenalty < minPenalty[currentStop] then:
minPenalty[currentStop] = totalPenalty
previousStop[currentStop] = earlierStop  // Store the previous hotel index

Print "Minimum Total Penalty: ", minPenalty[hotelCount - 1]

Initialize optimalStops[] to store the path
currentIndex = hotelCount - 1
stopCount = 0
While currentIndex != -1 do:
optimalStops[stopCount++] = currentIndex
currentIndex = previousStop[currentIndex]

Reverse the optimalStops path

Print "Optimal Sequence of Hotels (mile posts):"
For i = 1 to stopCount - 1 do:
Print allStops[optimalStops[i]]
```

Time Complexity Analysis

Function: `findOptimalStops`

The function `findOptimalStops` uses a dynamic programming approach to calculate the minimum penalty for each stop and determine the optimal sequence of hotel stops. It uses two nested loops: - The outer loop iterates through each stop (‘currentStop‘), and for each stop, the inner loop checks all

previous stops (‘earlierStop‘) to calculate the penalty. - Therefore, the time complexity of the nested loops is $O(n^2)$, where n is the number of stops (hotels plus the starting point).

Thus, the overall time complexity is:

$$\text{Time Complexity: } O(n^2)$$

Where n is the number of hotel stops (including the starting point).

Space Complexity

The space complexity is determined by the arrays used to store the distances, penalties, previous stops, and the optimal sequence: - The ‘allStops[]‘, ‘minPenalty[]‘, ‘previousStop[]‘, and ‘optimalStops[]‘ arrays each require $O(n)$ space. Thus, the overall space complexity is:

$$\text{Space Complexity: } O(n)$$

Where n is the number of stops (hotels plus the starting point).

Question 7

Problem Statement: A subsequence is palindromic if it is the same whether read left to right or right to left. For instance, the sequence A, C, G, T, G, T, C, A, A, A, A, T, C, G has many palindromic subsequences, including A, C, G, C, A and A, A, A, A (on the other hand, the subsequence A, C, T is not palindromic). Devise an algorithm that takes a sequence x[1 . . . n] and returns the (length of the) longest palindromic subsequence. Its running time should be O(n2). The Longest Common Subsequence (LCS) problem determines the length of the longest subsequence that is common to two sequences. A subsequence is a sequence derived from another sequence by deleting some or no elements without changing the order of the remaining elements.

Pseudocode

```
Function lcs(v1[], v2[], n):
Create a 2D DP table dp[n+1][n+1]
For i = 0 to n do:
For j = 0 to n do:
If i == 0 or j == 0:
dp[i][j] = 0
Else if v1[i-1] == v2[j-1]:
dp[i][j] = dp[i-1][j-1] + 1
Else:
dp[i][j] = max(dp[i-1][j], dp[i][j-1])
Return dp[n][n] // The length of the longest common subsequence

Function longestPalindromicSubsequence(v[], n):
Create an array reversed_v of size n
For i = 0 to n-1 do:
reversed_v[i] = v[n-i-1] // Reverse the string
Return lcs(v, reversed_v, n) // Find LCS between original string and
reversed string

Main Function:
Define an array of test cases test_cases[] with strings
For each test_case in test_cases do:
Call longestPalindromicSubsequence with test_case and print result
```

Time Complexity Analysis

Function: `lcs`

The function `lcs` calculates the length of the longest common subsequence (LCS) between two strings. The time complexity is determined by the two nested loops that iterate over the two strings.

- There are two loops, each of size $n + 1$ (since the DP table is $(n + 1) \times (n+1)$). - Each operation inside the loop (comparison and table update) takes constant time $O(1)$.

Thus, the time complexity of the `lcs` function is:

$$\text{Time Complexity of } lcs : O(n^2)$$

where n is the length of the input strings.

Function: `longestPalindromicSubsequence`

The function `longestPalindromicSubsequence` calls the `lcs` function with the original string and its reverse. The time complexity of reversing the string is $O(n)$, and the LCS computation takes $O(n^2)$.

Thus, the overall time complexity of the `longestPalindromicSubsequence` function is:

$$\text{Time Complexity of } longestPalindromicSubsequence : O(n^2)$$

where n is the length of the input string.

Main Function

In the `main` function: - We loop through all test cases. Let the number of test cases be t, and for each test case, the input string has a length of n. - For each test case, the function `longestPalindromicSubsequence` is called, which has a time complexity of $O(n^2)$.

Thus, the total time complexity for all test cases is:

$$\text{Total Time Complexity: } O(t \cdot n^2)$$

where t is the number of test cases and n is the length of the longest string in the test cases.

Space Complexity

The space complexity is dominated by the space required to store the DP table, which is $O(n^2)$ for the `lcs` function, and the space required to store the reversed string, which is $O(n)$.

Thus, the overall space complexity is:

$$\text{Space Complexity: } O(n^2)$$

where n is the length of the input string.

Question 8

Given two strings x = x 1 x2 · · · xn and y = y1 y2 · · · ym, we wish to find the length of their longest common substring, that is, the largest k for which there are indices i and j with x i xi+1 · · · xi+k-1 = yj yj+1 · · · yj+k-1. Show how to do this in time O(mn).

Pseudocode

```
Function longestCommonSubstring(X, Y):
n = length of X
m = length of Y
max_length = 0
Create a 2D dp array of size (n+1) x (m+1)
Initialize dp array with zeros

For i = 1 to n do:
For j = 1 to m do:
If X[i-1] == Y[j-1]:
dp[i][j] = dp[i-1][j-1] + 1
If dp[i][j] > max_length:
max_length = dp[i][j]
Else:
dp[i][j] = 0

Return max_length

Main Function:
Define test cases array with pairs of strings
For each test_case in test_cases do:
Call longestCommonSubstring with test_case[0] and test_case[1]
Print the length of the longest common substring
```

Time Complexity Analysis

Function: `longestCommonSubstring`

- The function `longestCommonSubstring` uses a 2D dynamic programming (DP) table ‘dp‘ to store the length of the longest common substring for each pair of substrings of X and Y. - The size of the DP table is $(n+1) \times (m+1)$, where n is the length of string X and m is the length of string Y. - We iterate over each cell of the DP table using two nested loops. Each loop runs for n and m iterations respectively.

Thus, the time complexity of the `longestCommonSubstring` function is:

$$\text{Time Complexity of } longestCommonSubstring : O(n \times m)$$

where n is the length of string X and m is the length of string Y.

Main Function

In the `main` function: - We have a fixed number of test cases. Let the number of test cases be t, and for each test case, the input strings X and Y have lengths n and m. - For each test case, we call the `longestCommonSubstring` function, which has a time complexity of $O(n \times m)$.

Thus, the total time complexity for all test cases is:

$$\text{Total Time Complexity: } O(t \cdot n \cdot m)$$

where t is the number of test cases, and n and m are the lengths of the respective strings for each test case.

Space Complexity

The space complexity is determined by the 2D DP table used in the `longestCommonSub` function, which requires $O(n \times m)$ space.

Thus, the overall space complexity is:

$$\text{Space Complexity: } O(n \times m)$$

where n is the length of string X and m is the length of string Y.

Question 9

Problem: Given an unlimited supply of coins of denominations x 1 x2 · · · xn , we wish to make change for a value v; that is, we wish to find a set of coins whose total value is v. This might not be possible: for instance, if the denominations are 5 and 10 then we can make change for 15 but not for 12. Give an O(nv) dynamic-programming algorithm for the following problem. Input: x1 x2 · · · xn ; v. Question: Is it possible to make change for v using coins of denominations x1 x2 · · · xn ?

Pseudocode

```
Function isTargetSumPossible(denominations, numDenominations, targetAmount):
Create a DP array isPossible of size (targetAmount + 1)
Initialize all elements of isPossible to false
isPossible[0] = true   // Base case: It is always possible to form 0

For i = 0 to numDenominations - 1 do:
For amount = denominations[i] to targetAmount do:
If isPossible[amount - denominations[i]] is true:
isPossible[amount] = true

Return isPossible[targetAmount]  // Return whether targetAmount can be formed

Main Function:
Define test cases array with different coin denominations and target amounts
For each test_case in test_cases do:
Call isTargetSumPossible with test_case.denominations,
 test_case.numDenominations, and test_case.targetAmount
Print the result (Yes/No) for whether the targetAmount can be formed
```

Time Complexity Analysis

Function: `isTargetSumPossible`

- The function `isTargetSumPossible` uses a dynamic programming (DP) array `isPossible` to track whether a certain amount can be formed using the given denominations. - The size of the DP array is targetAmount + 1, where targetAmount is the amount that needs to be formed. - We iterate over each coin denomination using the outer loop, which runs for numDenominations iterations. - For each denomination, the inner loop iterates through all amounts from the denomination value up to the target amount.

Thus, the time complexity of the `isTargetSumPossible` function is:

Time Complexity of `isTargetSumPossible` : $O(\text{numDenominations} \times \text{targetAmount})$

where numDenominations is the number of available denominations and targetAmount is the target amount that needs to be formed.

Main Function

In the `main` function: - We have a fixed number of test cases. Let the number of test cases be t, and for each test case, the input consists of coin denominations and the target amount. - For each test case, we call the `isTargetSumPossible` function, which has a time complexity of $O(\text{numDenominations} \times \text{targetAmount})$.

Thus, the total time complexity for all test cases is:

$$\text{Total Time Complexity: } O(t \cdot \text{numDenominations} \cdot \text{targetAmount})$$

where t is the number of test cases, numDenominations is the number of coin denominations, and targetAmount is the target amount for each test case.

Space Complexity

The space complexity is determined by the DP array `isPossible`, which requires $O(\text{targetAmount})$ space.

Thus, the overall space complexity is:

$$\text{Space Complexity: } O(\text{targetAmount})$$

where targetAmount is the amount that needs to be formed.

Question 10

Problem: Consider the following variation on the change-making problem (Exercise 11.): you are given denominations x1 x2 · · · xn , and you want to make change for a value v, but you are allowed to use each denomination at most once. For instance, if the denominations are 1, 5, 10, 20, then you can make change for $16 = 1 + 15$ and for $31 = 1 + 10 + 20$ but not for 40 (because you can't use 20 twice). Input: Positive integers x1 x2 · · · xn ; another integer v. Output: Can you make change for v, using each denomination xi at most once? Show how to solve this problem in time O(nv).

Pseudocode

```
Function canMakeChangeOnce(coinDenominations, numCoins, targetAmount):
Create a DP table dp[numCoins + 1][targetAmount + 1]
Initialize all elements of dp to false

dp[0][0] = true   // Base case: sum of 0 is always possible with 0 coins

For i = 1 to numCoins do:
currentCoin = coinDenominations[i - 1]

For j = 0 to targetAmount do:
dp[i][j] = dp[i - 1][j]  // Exclude the current coin (carry previous result)

If j >= currentCoin and dp[i - 1][j - currentCoin] is true:
dp[i][j] = true  // Include the current coin

Return dp[numCoins][targetAmount]  // Return whether targetAmount can be formed

Main Function:
Define test cases array with coin denominations and targetAmount
For each test_case in test_cases do:
Call canMakeChangeOnce with test_case.coinDenominations, test_case.numCoins,
 and test_case.targetAmount
Print the result (Yes/No) for whether the targetAmount can be formed using the
coins
```

Time Complexity Analysis

Function: `canMakeChangeOnce`

- The function `canMakeChangeOnce` uses dynamic programming (DP) to determine whether it is possible to make change for the given target amount using

the provided coin denominations. - The DP table `dp[i][j]` stores whether it is possible to form amount j using the first i coins. The table has dimensions $(\text{numCoins}+1) \times (\text{targetAmount}+1)$, where numCoins is the number of coins and targetAmount is the target amount. - We fill the table by iterating over the coins and amounts in two nested loops: one over the number of coins and one over the target amount. - Each cell in the DP table is filled in constant time.

Thus, the time complexity of the `canMakeChangeOnce` function is:

$$\text{Time Complexity of } \texttt{canMakeChangeOnce} : O(\text{numCoins} \times \text{targetAmount})$$

where numCoins is the number of coin denominations and targetAmount is the target amount.

Main Function

In the `main` function: - We have a fixed number of test cases. Let the number of test cases be t, and for each test case, the input consists of coin denominations and the target amount. - For each test case, we call the `canMakeChangeOnce` function, which has a time complexity of $O(\text{numCoins} \times \text{targetAmount})$.

Thus, the total time complexity for all test cases is:

$$\text{Total Time Complexity: } O(t \cdot \text{numCoins} \cdot \text{targetAmount})$$

where t is the number of test cases, numCoins is the number of coin denominations, and targetAmount is the target amount for each test case.

Space Complexity

The space complexity is determined by the DP table `dp`, which has dimensions $(\text{numCoins} + 1) \times (\text{targetAmount} + 1)$. Therefore, the space complexity is $O(\text{numCoins} \times \text{targetAmount})$.

Thus, the overall space complexity is:

$$\text{Space Complexity: } O(\text{numCoins} \times \text{targetAmount})$$

where numCoins is the number of coin denominations and targetAmount is the target amount.

Question 11

Problem: Given a convex polygon P = (v0, . . . , vn-1), and a weight function w on triangles, find a triangulation minimizing the total weight.

Pseudocode

```
Function calculateMinScore(vertexValues, start, end, memo):
If start + 1 == end:
Return 0    // Base case: No triangle can be formed with two vertices

If memo[start][end] != -1:
Return memo[start][end]    // Return cached result (memoization)

minScore = INT_MAX  // Initialize minScore to maximum value

For split = start + 1 to end - 1:
score = vertexValues[start] * vertexValues[end] * vertexValues[split]
+ calculateMinScore(vertexValues, start, split, memo)
+ calculateMinScore(vertexValues, split, end, memo)
If score < minScore:
minScore = score  // Update minScore

memo[start][end] = minScore  // Store result in memo table
Return minScore

Function findMinTriangulationScore(vertexValues, vertexCount):
Allocate memo table of size vertexCount x vertexCount
For i = 0 to vertexCount - 1:
For j = 0 to vertexCount - 1:
memo[i][j] = -1    // Initialize memo table with -1

result = calculateMinScore(vertexValues, 0, vertexCount - 1, memo)
Return result

Main Function:
Define vertexValues array and vertexCount
Call findMinTriangulationScore with vertexValues and vertexCount
Print the result (minimum triangulation score)
```

Time Complexity Analysis

Function: `calculateMinScore`

- The `calculateMinScore` function is a recursive function that computes the minimum triangulation score for a polygon, given the vertex values. - It uses

memoization to avoid redundant calculations. The recursive calls divide the problem into smaller subproblems by splitting the polygon at different points, and then combining the results. - The number of recursive calls depends on the number of vertices. In the worst case, for each pair of vertices $(start, end)$, the function tries all possible splits between them, resulting in a recursive tree structure.

Thus, the time complexity of the `calculateMinScore` function is dominated by the number of subproblems it computes. For a polygon with n vertices, the number of subproblems is $O(n^2)$, and for each subproblem, we iterate over all possible splits, which adds an additional factor of $O(n)$.

Thus, the time complexity of the `calculateMinScore` function is:

$$\text{Time Complexity of } \texttt{calculateMinScore} : O(n^3)$$

where n is the number of vertices.

Function: `findMinTriangulationScore`

In the `findMinTriangulationScore` function: - The function first allocates a memoization table, which takes $O(n^2)$ time and space. - It then calls the `calculateMinScore` function to calculate the minimum triangulation score, which takes $O(n^3)$ time as discussed above.

Thus, the total time complexity of the `findMinTriangulationScore` function is:

$$\text{Total Time Complexity: } O(n^3)$$

where n is the number of vertices.

Space Complexity

The space complexity is determined by: 1. The memoization table, which requires $O(n^2)$ space, where n is the number of vertices. 2. The recursion stack, which can go as deep as $O(n)$ in the worst case.

Thus, the overall space complexity is:

$$\text{Space Complexity: } O(n^2)$$

where n is the number of vertices.

Question 12

Problem: In the art gallery gaurding problem we are given a line L that represents a long hallway in an art gallery. We are also given a set X= x1 x2 · · · xn-1 of real numbers that specify the positions of the paintings in this hallway. Suppose that a single gaurd can protect all the paintings within distance at most 1 of his or her position(on both sides). Design an algorithm for finding a placement of gaurds that uses the minimum number of gaurds to gaurd all the paintings with position in X .

Pseudocode

```
Function compare(a, b):
diff = a - b
If diff < 0: return -1
If diff > 0: return 1
return 0

Function findMinimumGuards(paintingPositions, paintingCount,
guardPositions, guardCount):
Create a sorted copy of paintingPositions
Sort the sortedPositions array

Allocate guards array to store guard positions
Initialize guardIndex to 0

Set i = 0  // Traverse through sorted painting positions
While i < paintingCount:
leftmostPainting = sortedPositions[i]
currentGuardPosition = leftmostPainting + 1
Place guard at currentGuardPosition
Increment guardIndex

While i < paintingCount and sortedPositions[i] <= currentGuardPosition + 1:
Increment i // Skip all paintings covered by the guard

Set guardPositions to the guards array
Set guardCount to guardIndex

Free sortedPositions array

Main Function:
Define paintingPositions array and its count
Initialize guardPositions as NULL and guardCount as 0

Call findMinimumGuards(paintingPositions, paintingCount,
&guardPositions, &guardCount)

Print the number of guards and their positions

Free allocated memory for guard positions
```

Time Complexity Analysis

Function: `compare`

The `compare` function is used for sorting the painting positions in ascending order. It compares two painting positions and returns the result based on

their difference.

- The time complexity of this function is constant:

$$\text{Time Complexity of } \texttt{compare} : O(1)$$

Function: `qsort`

The sorting operation is done using the `qsort` function, which uses the quicksort algorithm under the hood. Quicksort has an average case time complexity of $O(n \log n)$ and a worst-case time complexity of $O(n^2)$.

Thus, the time complexity of the sorting step is:

$$\text{Time Complexity of sorting: } O(n \log n)$$

where n is the number of paintings.

Function: `findMinimumGuards`

In the `findMinimumGuards` function: - The sorting operation takes $O(n \log n)$ time. - The while loop traverses through the sorted painting positions array and places guards. In the worst case, each painting is considered once, so the time complexity of this loop is $O(n)$. - The nested while loop checks if a painting is covered by the current guard, which also runs in $O(n)$ time overall.

Thus, the total time complexity of the `findMinimumGuards` function is:

$$\text{Total Time Complexity: } O(n \log n)$$

where n is the number of paintings.

Space Complexity

The space complexity is determined by: 1. The array `sortedPositions`, which stores the sorted painting positions. This takes $O(n)$ space. 2. The array `guards`, which stores the guard positions. This also takes $O(n)$ space. 3. The input array `paintingPositions`, which takes $O(n)$ space.

Thus, the overall space complexity is:

$$\text{Space Complexity: } O(n)$$

where n is the number of paintings.