Theory Assignment-2: ADA Winter-2024

Vikranth Udandarao (2022570) Ansh Varshney (2022083)

1 Preprocessing and Assumptions

In this problem, no specific preprocessing steps are required. We assume or have been given the following:

1. The array consists of n integers with 0-based indexing.
2. Mr. Fox can either say RING or DING.
3. Mr. Fox is forbidden to say the same word more than three times a row.
4. All the account are settled in the end after Mr. Fox visits each booth in order and "Hot Box" is called, without the need for him to physically carry chickens, i.e. he visits each index exactly once.
5. Size of array is greater than equal to 1.
6. The array given to us is nums and it can have any value (positive, negative or zero).

2 Algorithm Description

The algorithm uses dynamic programming to find the maximum sum that can be obtained by choosing a subset of elements from the given array nums. It maintains a three-dimensional dynamic programming table dp, where dp[i][rc][dc] represents the maximum sum achievable up to index i considering rc consecutive *ring* and dc consecutive *ding* numbers for each index.

The algorithm uses recursion (memoization) to implement this logic. It starts from index $n-1$ and then reaches to the index 0 (base case) recursively. In each recursive call, it checks that rc and dc do not exceed 3. For each index, the algorithm considers all possible cases (ring, ding) and then returns the maximum possible value that can be obtained till that index. Additionally, it stores this value in the dp array so that it does not have to recalculate it if needed in the recursion ahead.

3 Subproblem Definition

The subproblem for index ind is defined as finding the maximum sum achievable starting from index 0 to index ind, such that neither ring nor ding is called more than 3 times consecutively till ind. The dp[ind][rc][dc] will basically store the maximum sum that can be obtained till index ind with ring called rc consecutive times and ding called dc consecutive times just before this index. The function CHECK(ind) returns the maximum sum that can be obtained from index 0 to ind in nums.

4 Recurrence of the Subproblem

The recurrence relation of the subproblem in this dynamic programming algorithm can be expressed as follows: Let dp[i][rc][dc] denote the maximum sum achievable up to index i considering rc consecutive RING before the index and dc consecutive DING before the index. Then, the recurrence relation is (when rc and dc both are less than 3):

$$\text{dp}[i][rc][dc] = \max \left\{ \begin{array}{ll} \text{dp}[i-1][rc+1][0] + \text{nums}[i], & \text{if RING} \\ \text{dp}[i-1][0][dc+1] - \text{nums}[i], & \text{if DING} \end{array} \right\}$$

For the case when rc equals 3: dp[i][rc][dc]= dp[i-1][0][1]-nums[ind]

For the case when dc equals 3: dp[i][rc][dc]= dp[i-1][1][0]+nums[ind]

This recurrence relation represents the maximum sum achievable at index i with different configurations of consecutive RINGs and DINGs till the index, while considering the reward or penalty associated with the current booth (indexed by i).

5 Specific Subproblem to Solve the Actual Problem

The specific subproblem to solve the actual problem is dp[$n-1$][0][0], where n is the size of the nums array. It represents the maximum sum achievable considering all elements of nums with no more than three consecutive ring or ding operations throughout.

6 Pseudocode

Algorithm 1 Function to Find Maximum Number of Chickens

```
function CHECK(nums, ind, rc, dc, dp)
    if ind == 0 then
        if rc == 3 then
            return −nums[ind]                         ▷ Base case if three consecutive rings
        else if dc == 3 then
            return nums[ind]                          ▷ Base case if three consecutive dings
        else if nums[ind] > 0 then
            return nums[ind]                          ▷ Positive reward
        else if nums[ind] < 0 then
            return −nums[ind]                         ▷ Negative penalty
        end if
    end if
    if dp[ind][rc][dc] ≠ INT_MIN then                 ▷ Check and return if already computed
        return dp[ind][rc][dc]
    end if
    ring ← INT_MIN                                    ▷ Initialize variables
    ding ← INT_MIN
    if rc == 3 then
        ding ← check(nums, ind - 1, 0, 1, dp) - nums[ind]           ▷ Compute ding
    else if dc == 3 then
        ring ← check(nums, ind - 1, 1, 0, dp) + nums[ind]           ▷ Compute ring
    else
        ring ← check(nums, ind - 1, rc + 1, 0, dp) + nums[ind]      ▷ Compute ring
        ding ← check(nums, ind - 1, 0, dc + 1, dp) - nums[ind]      ▷ Compute ding
    end if
    dp[ind][rc][dc] ← max(ring, ding)                 ▷ Store maximum value
    return max(ring, ding)                            ▷ Return maximum value
end function

function MAIN
    n ← size of nums
    a ← 2D array of integers with dimensions (4, 4) initialized with INT_MIN
    dp ← 3D array of integers with dimensions (n, 4, 4) initialized with a
    print CHECK(NUMS, N - 1, 0, 0, DP)
end function
```

7 Complexity Analysis

7.1 Time Complexity

The time complexity is $O(n)$, where n is the size of the input array. If memoization was not employed, the recursive relation had time complexity of order O(2^n). This could be shown using a recurrence tree. However, since we are storing repeated values in the dp array, we will not be calculating redundant values and this will reduce the complexity to O(n). Note that we cannot use recurrence relation to prove time complexity in memoization.

7.2 Space Complexity

The algorithm uses a three-dimensional dynamic programming table of size $O(n \times 4 \times 4)$, where n is the size of the input array. Hence, the space complexity is $O(n)$.

8 Explanation of the Running Time Complexity

The space complexity is O(n) as shown above. The time complexity can be proven as O(n) by the following argument:
The algorithm processes each booth in the array once, and at each booth, it computes the maximum chickens earned considering the current booth and the previously visited booths.
First, the algorithm initializes the memoization table dp which is done in O(1) time. Then, the algorithm performs constant time work to compute the maximum chickens earned for each booth, based on the previous booths' results stored in the memoization table. This process involves a constant number of lookups in the memoization table and comparisons to update the maximum value. Since there are n booths, time complexity for this operation is O(n). Lastly, the algorithm retrieves the maximum value from the memoization table in O(1) time.
So, the total time complexity will be O(n) + O(1) + O(1) which is O(n) only.

9 Proof of Correctness

Proof of Correctness by Induction:

Base Step (n = 1): The algorithm correctly handles the base case where there's only one booth, returning the reward if positive or the negative penalty if negative.

Inductive Hypothesis: Assume the algorithm correctly computes the maximum chickens earned for an array of length k, where $k \leq n$.

Inductive Step: We need to show the algorithm correctly computes the maximum chickens earned for an array of length $k + 1$. For the $(k + 1)$-th booth, if he said RING three times consecutively before, he must say DING at this booth. Otherwise, he can say RING, earning $M[k] + A[k+1]$ chickens. If he said DING three times consecutively before, he must say RING at this booth. Otherwise, he can say DING, earning $M[k] - A[k + 1]$ chickens. The maximum value out of both of these will be chosen, ensuring the correct computation of the maximum chickens earned for an array of length $k + 1$. Thus, by induction, the algorithm correctly computes the largest number of chickens Mr. Fox can earn for an array of length n.

SAMPLE TESTCASE: $n = 10$, nums $= [2, 4, 5, 6, -3, -7, 1, -4, 5, -7]$

OUTPUT: 40