Theory Assignment-2: ADA Winter-2024

Shamik Sinha (2022468)   Vansh Yadav (2022559)

Preprocessing

In the provided algorithm, prior to executing the algorithm, the DP array is initialized with zeros. This preparatory step ensures that the DP array begins with a clean state, devoid of any pre-existing values. By starting with a uniform state of zero values, the algorithm establishes a base condition from which subsequent computations can accurately determine the maximum number of chickens Mr. Fox can earn while navigating the obstacle course.

Algorithm Description

Assumptions Made

1. The arrays are composed of integers i.e. they are integer arrays or at the very least have numeric values (floating point or integer).
2. The array indexing follows a 0-based convention. An 'n' sized array has starting index 0 and last index at 'n'-1.

Input

The input consists of two lines:

- The first line contains an integer , denoting the size of the array.
- The second line contains n integers $A[1], A[2], \ldots, A[n]$ separated by spaces, representing the elements of the array. Each integer can be positive, negative, or zero.

Output

A single numeric value (Integer or Float), which is largest number of chickens that Mr. Fox earns by running the obstacle course.

Subproblem Definition

The subproblem in this context involves determining the maximum number of chickens Mr. Fox can earn up to a certain booth in the obstacle course, considering the choices of saying ”RING” or ”DING” at each booth while adhering to the constraints mentioned in the problem statement. Formally, we define a subproblem as finding the maximum number of chickens earned up to booth k with i consecutive ”RING” or ”DING” choices made before booth k, denoted as $dp[k][i][j]$, where:

- k represents the booth number,
- i represents the count of consecutive ”RING” or ”DING” choices before booth k (limited to a maximum of 3),
- j indicates the last choice made (0 for ”RING” , 1 for ”DING” and -1 is to denote that its the first recursive call).

Recurrence of the Subproblem

$$dp[ind][count][flag] = \max \begin{cases} \text{arr[ind]} + \text{dp[ind + 1][1][0]} & \text{if arr[ind]} < 0 \text{ and flag} = -1 \\ -\text{arr[ind]} + \text{dp[ind + 1][1][1]} & \text{if arr[ind]} < 0 \text{ and flag} = -1 \end{cases} .$$

$$dp[ind][count][flag] = \max \begin{cases} \text{arr[ind]} + \text{dp[ind + 1][count + 1][0]} & \text{if arr[ind]} < 0 \text{ and flag} = 0 \text{ and count} < 3 \\ |\text{arr}[ind]| + \text{dp}[ind + 1][1][1] & \text{if arr[ind]} < 0 \text{ and flag} = 0 \text{ and count} < 3 \end{cases} .$$

$$dp[ind][count][flag] = \begin{cases} -\text{arr[ind]} + \text{dp[ind + 1][1][1]} & \text{if arr[ind]} < 0 \text{ and flag} = 0 \text{ and count} = 3 \end{cases} .$$

$$dp[ind][count][flag] = \max \begin{cases} \text{arr[ind]} + \text{dp[ind + 1][1][0]} & \text{if arr[ind]} < 0 \text{ and flag} = 1 \\ -\text{arr[ind]} + \text{dp[ind + 1][count + 1][1]} & \text{if arr[ind]} < 0 \text{ and flag} = 1 \end{cases} .$$

$$dp[ind][count][flag] = \max \begin{cases} \text{arr[ind]} + \text{dp[ind + 1][1][0]} & \text{if arr[ind]} > 0 \text{ and flag} = -1 \\ -\text{arr[ind]} + \text{dp[ind + 1][1][1]} & \text{if arr[ind]} > 0 \text{ and flag} = -1 \end{cases} .$$

$$dp[ind][count][flag] = \max \begin{cases} \text{arr[ind]} + \text{dp[ind + 1][count + 1][0]} & \text{if arr[ind]} > 0 \text{ and flag} = 0 \text{ and count} < 3 \\ -\text{arr}[ind] + \text{dp}[ind + 1][1][1] & \text{if arr[ind]} < 0 \text{ and flag} = 0 \text{ and count} < 3 \end{cases} .$$

$$dp[ind][count][flag] = \begin{cases} -\text{arr[ind]} + \text{dp[ind + 1][1][1]} & \text{if arr[ind]} > 0 \text{ and flag} = 0 \text{ and count} = 3 \end{cases} .$$

$$dp[ind][count][flag] = \max \begin{cases} \text{arr[ind]} + \text{dp[ind + 1][1][0]} & \text{if arr[ind]} > 0 \text{ and flag} = 1 \\ -\text{arr[ind]} + \text{dp[ind + 1][count + 1][1]} & \text{if arr[ind]} > 0 \text{ and flag} = 1 \end{cases} .$$

$$T(n) = \begin{cases} 0 & \text{if } n = 0 \\ T(n-1) + O(1) & \text{otherwise} \end{cases}$$

This recurrence relation has a linear time complexity, since it reduces the problem size by one in each step and does a constant amount of work. This matches the overall time complexity of your code, which is O(n), where n is the size of the input array.

Specific Subproblem(s) for the Actual Problem

The specific subproblem that solves the actual problem involves determining the maximum number of chickens Mr. Fox can earn up to the last booth while adhering to the constraints mentioned in the problem statement.

Formally, this is represented by the subproblem of finding the maximum number of chickens earned up to booth n with i consecutive "RING" or "DING" choices made before booth n, denoted as $dp[n][i][j]$, where:

- n is the total number of booths,
- i represents the count of consecutive "RING" or "DING" choices before booth n (limited to a maximum of 3),
- j indicates the last choice made (0 for "RING" and 1 for "DING").

The solution to this specific subproblem, stored in $dp[n][i][j]$, provides the maximum number of chickens Mr. Fox can earn after navigating all booths, satisfying the conditions mentioned in the problem statement.

The final answer, representing the maximum number of chickens that Mr. Fox can earn while traversing the obstacle course, is stored in dp[0][0][0].

This value corresponds to the maximum number of chickens earned by Mr. Fox when he starts from the first booth, has made no consecutive occurrences of the same action and his previous action was arbitrary.

Thus, to obtain the final result of the algorithm, one should output dp[0][0][0].

Algorithm Description

The provided code efficiently solves the problem of maximizing Mr. Fox's chicken earnings while navigating an obstacle course with rewards and penalties. It uses dynamic programming to iterate through each booth, considering the options of saying "RING" or "DING" while adhering to the constraints. By updating a 3D array to store the maximum earnings at each booth, the algorithm computes the optimal solution. Finally, it returns the maximum number of chickens earned after navigating all booths.

1. **Initialization**
   - We start by initializing a 3D array dp to store the maximum number of chickens Mr. Fox can earn up to each booth. The dimensions of this array are $(n+1) \times 4 \times 2$, where n is the number of booths. The three dimensions represent:
     - k: the booth number,
     - i: the count of consecutive "RING" or "DING" said before the k-th booth (limited to a maximum of 3),
     - j: the last choice made (0 for "RING" and 1 for "DING").

2. **Base Case Initialization**
   - We initialize the values for the first booth ($k = 1$). For each possible count of consecutive choices i and each possible last choice j, we compute the initial values for $dp[1][i][j]$. These initial values represent the rewards or penalties associated with the first booth, depending on whether Mr. Fox chooses "RING" or "DING".

3. **Dynamic Programming**
   - Next, we iterate through the booths starting from the second booth ($k = 2$). For each booth k, we consider all possible counts of consecutive choices i and last choices j.
   - For each combination of i and j, we update $dp[k][i][j]$ based on the choices made at the previous booth ($k - 1$) and the reward/penalty associated with the current booth ($A[k]$).
   - We update $dp[k][i][j]$ by considering the two possible choices Mr. Fox has at the current booth: "RING" or "DING". We choose the option that maximizes the number of chickens earned up to the current booth, while adhering to the constraints mentioned in the problem statement.

4. **Maximum Chickens Earned**
   - After iterating through all booths, we compute the maximum number of chickens earned by Mr. Fox. This value is the maximum among $dp[n][0][0]$, $dp[n][1][0]$, $dp[n][2][0]$, and $dp[n][3][0]$, where n is the total number of booths. This represents the maximum number of chickens earned after navigating all booths.

5. **Return the Result**
   - Finally, we return the maximum number of chickens earned as the solution to the problem.

Time Complexity Analysis

1. **Input Reading**: Reading the input array `arr` takes linear time, $O(n)$, where n is the size of the array.
2. **Dynamic Programming Loop**:
   - The outer loop iterates over each index in the array `arr`, so it runs n times.
   - Inside this loop, there are three nested loops, each of which iterates at most 4 times (`count` from 0 to 3 and `flag` from 0 to 1).
   - Inside the innermost loop, there are constant time operations.

   Therefore, the time complexity within the dynamic programming loop is $O(n)$.

Overall, the time complexity of the code is $O(n)$.

Space Complexity Analysis

1. **Input Storage**: An array `arr` of size n is used to store the input elements, resulting in $O(n)$ space complexity.
2. **Dynamic Programming Table**:
   - The dynamic programming table `dp` is a 3D vector of size $(n+1) \times 4 \times 2$, which results in $O(n)$ space complexity.
3. **Other Variables**:
   - Other variables like `ind`, `count`, `flag`, and `result` are integers, which occupy constant space.

Summary

- **Time Complexity**: $O(n)$
- **Space Complexity**: $O(n)$

The code utilizes dynamic programming to optimize the solution to the problem, and its time and space complexities are both linear in terms of the input size.

Pseudocode

Below is the pseudocode which represents the logic of the C++ code we had written for the problem, in a more abstract and readable form. The function takes one array and its size (n). The function returns the largest number of chickens that Mr. Fox earns by running the obstacle course.

Algorithm 1 An Algorithm to find the largest number of chickens that Mr. Fox earns by running the obstacle course n.

```
function SOLVE(arr, n):
    $dp \leftarrow$ a 3D array of size $n+1 \times 4 \times 2$ filled with zeros
    for $ind$ from $n-1$ to 0 do:
        for $count$ from 0 to 3 do:
            for $flag$ from 0 to 1 do:
                $result \leftarrow dp[ind][count][flag]$
                if $arr[ind] < 0$ then:
                    if $flag == -1$ then:
                        $a1 \leftarrow arr[ind] + dp[ind+1][1][0]$
                        $a2 \leftarrow |arr[ind]| + dp[ind+1][1][1]$
                        $result \leftarrow \max(a1, a2)$
                    else if $flag == 0$ then:
                        if $count == 3$ then:
                            $result \leftarrow |arr[ind]| + dp[ind+1][1][1]$
                        else:
                            $a1 \leftarrow arr[ind] + dp[ind+1][count+1][0]$
                            $a2 \leftarrow |arr[ind]| + dp[ind+1][1][1]$
                            $result \leftarrow \max(a1, a2)$
                        end if
                    else if $flag == 1$ then:
                        if $count == 3$ then:
                            $result \leftarrow arr[ind] + dp[ind+1][1][0]$
                        else:
                            $a1 \leftarrow arr[ind] + dp[ind+1][1][0]$
                            $a2 \leftarrow |arr[ind]| + dp[ind+1][count+1][1]$
                            $result \leftarrow \max(a1, a2)$
                        end if
                    end if
                else:
                    if $flag == -1$ then:
                        $a1 \leftarrow arr[ind] + dp[ind+1][1][0]$
                        $a2 \leftarrow -1 \times arr[ind] + dp[ind+1][1][1]$
                        $result \leftarrow \max(a1, a2)$
                    else if $flag == 0$ then:
                        if $count == 3$ then:
                            $result \leftarrow -1 \times arr[ind] + dp[ind+1][1][1]$
                        else:
                            $a1 \leftarrow arr[ind] + dp[ind+1][count+1][0]$
                            $a2 \leftarrow -1 \times arr[ind] + dp[ind+1][1][1]$
                            $result \leftarrow \max(a1, a2)$
                        end if
                    else if $flag == 1$ then:
                        if $count == 3$ then:
                            $result \leftarrow arr[ind] + dp[ind+1][1][0]$
                        else:
                            $a1 \leftarrow arr[ind] + dp[ind+1][1][0]$
                            $a2 \leftarrow -1 \times arr[ind] + dp[ind+1][count+1][1]$
                            $result \leftarrow \max(a1, a2)$
                        end if
                    end if
                end if
            end for
        end for
    end for
    return $dp[0][0][0]$
end function
```

Proof of Correctness

Optimal Substructure:

We aim to maximize the number of chickens Mr. Fox can earn while navigating the obstacle course. This can be achieved by making optimal choices at each booth, considering the rewards and penalties associated with the choices.

For any booth k, the maximum number of chickens earned up to that booth depends on the maximum number of chickens earned at the previous booth, considering all possible choices made at the previous booth.

Overlapping Subproblems:

The overlapping subproblems arise from the fact that the same subproblem can occur at multiple booths. As we traverse through the booths, the choices made at previous booths affect the choices available at the current booth. Therefore, we can use dynamic programming to store and reuse the solutions to these overlapping subproblems efficiently.

Proof by Induction:

- **Base Case:** The base case occurs when Mr. Fox reaches the first booth ($k = 1$). Here, the maximum number of chickens earned is simply the reward or penalty associated with the first booth, depending on whether Mr. Fox chooses "RING" or "DING" ($dp[1][i][j]$). This base case is trivially correct.
- **Inductive Step:** For each subsequent booth k, we update $dp[k][i][j]$ based on the choices made at previous booths. By considering all possible choices at each booth, we select the one that maximizes the number of chickens earned up to the current booth while adhering to the constraints. Since we use optimal solutions to subproblems, we construct an optimal solution for the entire obstacle course.
- **Termination:** The algorithm terminates once Mr. Fox visits every booth, and the maximum number of chickens earned is stored in $dp[n][i][j]$, where n is the total number of booths.

By demonstrating both optimal substructure and overlapping subproblems, we establish the correctness of the algorithm, ensuring that it computes the largest number of chickens Mr. Fox can earn by navigating the obstacle course.