Theory Assignment-2: ADA Winter-2024

Aditya Sharma (2022038)   Ayan Kumar Singh (2022122)

1 Algorithm description

- The algorithm uses dynamic programming to solve the given problem. It employs top-down approach with memoization to avoid redundant computations and improve efficiency.
- Using recursive approach to find all possible combinations of "RING" and "DING" that the farmer can say at each index in array A.
- At each index i, we check this state (RINGstreak, DINGstreak, index). If we have been in this state before, then we we know what maxScore is achievable from this state. This is the overlapping problem that can be cached and thus, we don't need to recompute this again.
- We use memoization approach and are using 3D array to cached. dp[len + 1][4][4] with all values "unset" initially. As for each n indexes we can have 3 states of RINGstreak and DINGstreak respectively.

2 Subproblem definition

- At each index we have state of (index, RINGstreak,DINGstreak) and we have two choices:
  1 Either, we say "RING"
  2 or we say "DING"
- We explore both of these choices for each index when possible.
- In cases when we have exhausted our streak of "RING" or "DING", In that case we return whatever will be the next possible legal next step.
- In each recursive call we move one index of the array.
- In the base the we reach the end of the array. That happens when index becomes out of range of len, the function returns 0.

3 The specific subproblem(s) that solves the final problem (or gives the final answer)

- Exploration of both the choices of "RING" and "DING" at each index paired with passing of current RINGstreak and DINGstreak enables us to explore all possible paths and choose the optimal path to maximize no. of chickens at the end. The recursive approach paired up with memoization reduces redundant calculations where we may be starting with same conditions of (RINGstreak,DINGstreak,index) and we can access what maxScore was achievale with these starting conditions.

4 Recurrence of the subproblem

$$dp[index][ring][ding] = \begin{cases} -max\{-array[i] + dp[i+1][0][ding+1]\}, & \text{when } ring = 3 \\ array[i] + dp[i+1][ring+1][0], & \text{when } ding = 3 \\ max\{array[i] + dp[i+1][ring+1][0], -array[i] + dp[i+1][0][ding+1]\}, & \text{otherwise} \end{cases}$$

5 Complexity Analysis

Our algorithm uses dynamic programming using the memoization approach, for implementing the memoization approach we have created a 3-D Array of dimensions $n+1$, 4, 4. Creating such an array would require 4x4x$n+1$ auxiliary space thus the space complexity of our algorithm is:

$$O(n)$$

The number of sub-problems to our algorithm is determined by the size of our memoization table which is a 3D-Array of dimensions $n + 1$, 4, 4, where each entry stores a unique value. This in turn implies that the number of unique sub-problems are

$$O(n \times 4 \times 4) = O(n)$$

Also each sub-problem requires basic mathematical operations for computation, if it was not in the memoization table and if it was there, retrieving it takes constant time. Therefore time complexity per sub-problem is constant.

Hence, the overall time complexity of our algorithm really comes down to the size of the memoization table and the time taken to solve each sub-problem and since the size of the table was $O(n)$ and the calculation for sub-problem takes $O(1)$ time the complexity of our algorithm becomes $\mathbf{O(n)}$.

6 Pseudocode

Algorithm 1 Recursive Function with Memoization

```
 1: function SOLVE(array, ring, ding, index, dp)
 2:     len ← size of array
 3:     if index ≥ len then
 4:         return 0
 5:     else if ring == 3 then
 6:         return −array[index] + solve(array, 0, 1, index + 1, dp)
 7:     else if ding == 3 then
 8:         return array[index] + solve(array, 1, 0, index + 1, dp)
 9:     else if dp[index][ring][ding] ≠ unset then
10:         return dp[index][ring][ding]
11:     else
12:         val1 ← array[index] + solve(array, ring + 1, 0, index + 1, dp)
13:         val2 ← −array[index] + solve(array, 0, ding + 1, index + 1, dp)
14:         dp[index][ring][ding] ← max(val1, val2)
15:         return dp[index][ring][ding]
16:     end if
17: end function
```

7 Main Fucntion

- This is the main function that we use for SOLVE function.

```cpp
int main(){
    vector<int> array = {1,-2,3,-4,5,6,7,8};
    int len = array.size();
    vector<vector<vector<int>>> dp(len + 1, vector<vector<int>>(len + 1, vector<int>(
        len + 1, -1)));
    cout << solve(array, 0, 0, 0, dp) << endl;
    return 0;
}
```