Theory Assignment-2: ADA Winter-2024

Nikhil Kumar (2022322) Nikhil (2022321)

11 February 2024

1 Subproblem Definition

Finding maximum numbers of chickens Mr. Fox can earn in a given sequence of values which represent the chickens he encountered with restriction on consecutive actions.

2 Recurrence of the subproblem

Let's denote the function as `solve(a, rcount, dcount, score, pointer)`, where:

- `a` is the sequence of values representing the chickens Mr. Fox encounters.
- `rcount` is the count of consecutive ring actions.
- `dcount` is the count of consecutive ding actions.
- `score` is the current score obtained.
- `pointer` is the current index in the sequence `a`.

The recurrence relation is as follows:

$$\texttt{solve(a,rcount,dcount,score,pointer)} = \max \begin{pmatrix} \texttt{solve(a, 0, dcount + 1, score + sc, pointer + 1),} \\ \texttt{solve(a,rcount + 1, 0,score+a[pointer], pointer+ 1)} \end{pmatrix}$$

This relation states that at each step, Mr. Fox has two options: either perform a ding action or a ring action. Depending on the count of consecutive dings and rings, the score is updated accordingly. The function returns the maximum score achievable.

3 The specific subproblem(s) that solves the actual problem

1. **Base Case:**

$$\text{pointer} \geq \text{a.size()}$$

When `pointer` reaches the end of the vector `a`, the function returns `score`, indicating the end of the game.

2. **Ding Subproblem:**

$$(\text{dcount} < 3)$$

If `dcount` (the consecutive count of "dings") is less than 3, the function recursively calls itself, simulating the scenario where Mr. Fox chooses to "ding" (taking negative points) at the current position. It updates `dcount` and adds the appropriate score to the total `score`. If `dcount` reaches 3, it switches to the "ring" strategy.

3. **Ring Subproblem:** If `rcount` (the consecutive count of "rings") is less than 3, the function recursively calls itself, simulating the scenario where Mr. Fox chooses to "ring" (taking positive points) at the current position. It updates `rcount` and adds the appropriate score to the total `score`. If `rcount` reaches 3, it switches to the "ding" strategy.

4. **Decision Making:** After considering both "ding" and "ring" scenarios, the function selects the maximum score between the two strategies.

4 Algorithm Description

1. We define two variables, `ding` and `ring`, which store the result of saying "DING" and "RING" at the kth index of the array, respectively, and the previous optimal result already added to this.

2. We record the consecutive appearances of "RING" and "DING" using the `rcount` and `lcount`, respectively. If we say "RING," then we update `rcount = rcount+1` and `lcount` as 0 and vice versa.

3. We first of all check whether we have computed the result of this particular function call earlier using the 2d array as the memorization table, if we already know the answer then we directly return the answer else we proceed further.

4. Now, at first, we try to say "DING" at the kth index.
   i) For this, we first check whether we have used "DING" more than three times consecutively. This is achieved by comparing the value of `lcount` variable. The condition: `lcount < 3` ensures that we have not used "DING" more than three times.
   ii) Now if we have not used "DING" more than three times, then we check whether the value at the kth index is positive or negative. If the value is negative, then we take the modulus of that value and assign this to a temporary variable, `sc`. This basically means that Mr. Fox has won a reward this time by saying "DING." And if the value is positive, then we make it negative by multiplying it with -1 and assign this to the temporary variable `sc`, meaning that Mr. Fox has to pay a penalty of $-A[k]$ chickens this time.
   iii) Now we make a recursive call for the next element in the array and store the result of this call in the `ding` variable.
   iv) During the recursive call, we add the value of the temporary variable `sc` to the score, and we proceed for the next function call.
   v) Now, if our initial condition of `lcount < 3` fails, then this means that we have already said "DING" three times consecutively, and now we only have one option to say "RING," so we simply make a recursive call with the score as the old score plus the value at the kth index.

5. Now, we try to say "RING" at the kth index.
   i) For this, we first check whether we have used "RING" more than three times consecutively. This is achieved by comparing the value of `rcount` variable. The condition: `rcount < 3` ensures that we have not used "RING" more than three times.
   ii) Now if we have not used "RING" more than three times, then if the value is negative or positive, we assign this to a temporary variable, `sc`. This basically means that Mr. Fox has won a reward this time by saying "RING" if the value at the kth index was positive. If it was negative, then he paid a penalty of $A[k]$, and here we didn't have to take the modulus of that value, as in this case, the value is already trivial – if it is positive, then it is normally added; if negative, it is again added to the score only, which reduces the `sc` in this case.
   iii) Now we make a recursive call for the next element in the array and store the result of this call in the `ring` variable.
   iv) During the recursive call, we add the value of the temporary variable `sc` to the score. Then, we proceed to the next function call.
   v) Now, if our initial condition of `rcount < 3` fails, then this means that we have already said "RING" three times consecutively, and now we only have one option to say "DING," so we simply make a recursive call with updating the score accordingly.

6. Now, after the recursive calls return and we have a value in `ring` and `ding` variables, which represents saying "ring" and "ding" at that particular kth element, we return our final answer as the maximum of both of these variables as our optimal solution for that particular subproblem. And at the same time we store this answer in the 2d array for memorization.

5 Complexity Analysis

1. **Memoization Table Initialization**:
   - The memoization table is initialized once at the beginning of the program execution.
   - An 2d array is used to store all possible combinations of values for `rcount` and `dcount` and `pointer`.
   - Therefore, the time complexity for initialization of memoization table initialization is $O(N)$. As for every value in input array A we have to iterate the inner loop 2 times, one for DING and one for RING.

2. **Recursive Function Calls**:
   - The `solve` function is called recursively to compute the maximum number of chickens earned by Mr. Fox.
   - For each recursive call, there are a maximum of 16 unique subproblems (since both `rcount` and `dcount` can range from 0 to 3).
   - The function computes the maximum score for each subproblem and stores it in the memoization table.
   - The total number of recursive calls depends on the size of the input array n and the number of unique subproblems.
   - Since each subproblem is solved only once due to memoization, the total time spent on recursive calls is $O(n)$.

3. **Overall Time Complexity**:
   - The dominant factor in the time complexity is the number of unique subproblems, which is quadratic in terms of the input size n. Specifically, there are at most $16n$ unique subproblems.
   - Therefore, the overall time complexity of the optimized code is $O(n^2)$.

In summary, the optimized code achieves a time complexity of $O(n^2)$ by utilizing dynamic programming with memoization. It efficiently solves the problem by avoiding redundant computations through memoization and effectively reduces the time complexity from exponential to quadratic.

6 Pseudocode

The Memorization table is declared with a global scope. The dp array is initialized with INT_MIN as all the elements.

Algorithm 1 Mr. Fox Solver

```
function SOLVE(int *a, rcount: int, dcount: int, score: int, pointer: int, size: int)
    if pointer ≥ size then
        return score
    end if
    if dp[pointer][0] ≠ INT_MIN or dp[pointer][1] ≠ INT_MIN then
        if dp[pointer][0] ≠ INT_MIN then
            return dp[pointer][0]
        else
            return dp[pointer][1]
        end if
    end if
    ding ← 0
    ring ← 0
    if dcount < 3 then
        sc ← 0
        if a[pointer] < 0 then
            sc ← abs(a[pointer])
        else
            sc ← a[pointer] * -1
        end if
        ding ← SOLVE(a, 0, dcount + 1, score + sc, pointer + 1)
    else
        ring ← SOLVE(a, rcount + 1, 0, score + a[pointer], pointer + 1)
    end if
    if rcount < 3 then
        sc ← 0
        ring ← SOLVE(a, rcount + 1, 0, score + a[pointer], pointer + 1)
    else
        sc ← 0
        if a[pointer] < 0 then
            sc ← abs(a[pointer])
        else
            sc ← a[pointer] * -1
        end if
        ding ← SOLVE(a, 0, dcount + 1, score + sc, pointer + 1)
    end if
    score ← max(ding, ring)
    dp[pointer][0] = score
    dp[pointer][1] = score
    return score
end function
```