Theory Assignment-2: ADA Winter-2024

Krishna Shukla (2022254) Kshitij (2022256)

1 Preprocessing

As such no preprocessing is required but we are assuming array indexing starting from 0 (standard) and the array size is n,

we are doing it with both the approaches 1) Tabulation and 2)Memoization

2 Subproblem Definition

For Memoization

To define the subproblem with a tree, let's consider the problem as a decision tree where each node represents a booth and the edges represent the possible choices of saying "RING" or "DING". At each node, Mr. Fox has the option to either say "RING" or "DING" based on the constraints given in the problem.

The state of the subproblem can be represented by the following attributes:

- ind: The index of the current booth.
- noring: The number of consecutive "RING"s said so far.
- noding: The number of consecutive "DING"s said so far. Given these attributes, the subproblem can be defined as follows:

State: (ind, noring, noding)
Decision: Whether to say "RING" or "DING" at the current booth.
Objective: Maximize the total number of chickens earned by the end of the obstacle course. Now, let's represent this subproblem with a decision tree:

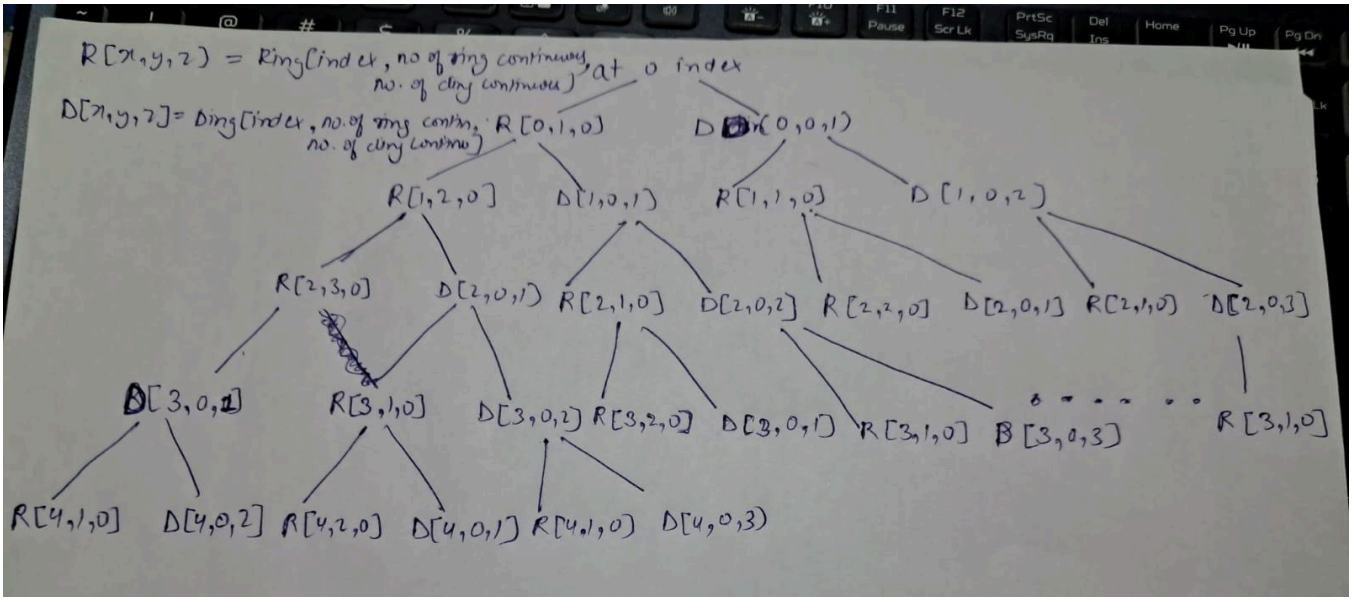
- Each level of the tree represents a booth.
- At each level, there are two branches representing the decisions "RING" and "DING".
- The tree continues until Mr. Fox reaches the last booth (ind = *n*).

The objective is to find the path in the decision tree that leads to the maximum total number of chickens earned while satisfying the given constraints.

In the provided code, the *f* function recursively explores this decision tree and memoizes the results using dynamic programming to avoid redundant calculations. The base case is when ind reaches *n*, at which point the maximum number of chickens earned is calculated based on the decisions made at each booth.

This approach effectively breaks down the problem into smaller subproblems, allowing for an efficient solution using dynamic programming.

(Please refer the image for better clarity)



For Tabulation

The problem is tabulated using a 2D vector dp, where dp[i][j] represents the maximum value achievable considering the first i elements of array a and the number of "dings" allowed, denoted by j.

n: Number of elements in the array a.

a: The array containing elements for which choices need to be made.

dp: A 2D vector where dp[i][j] represents the maximum value for the subproblem with i elements and j "dings.

The loop iterates in reverse order, from the last element (n-1) to the first element (0). At each step, for each possible value of noring (number of "dings"), it considers two cases:

If the current element is positive, it adds the element to the "ring" and subtracts it from the "ding."

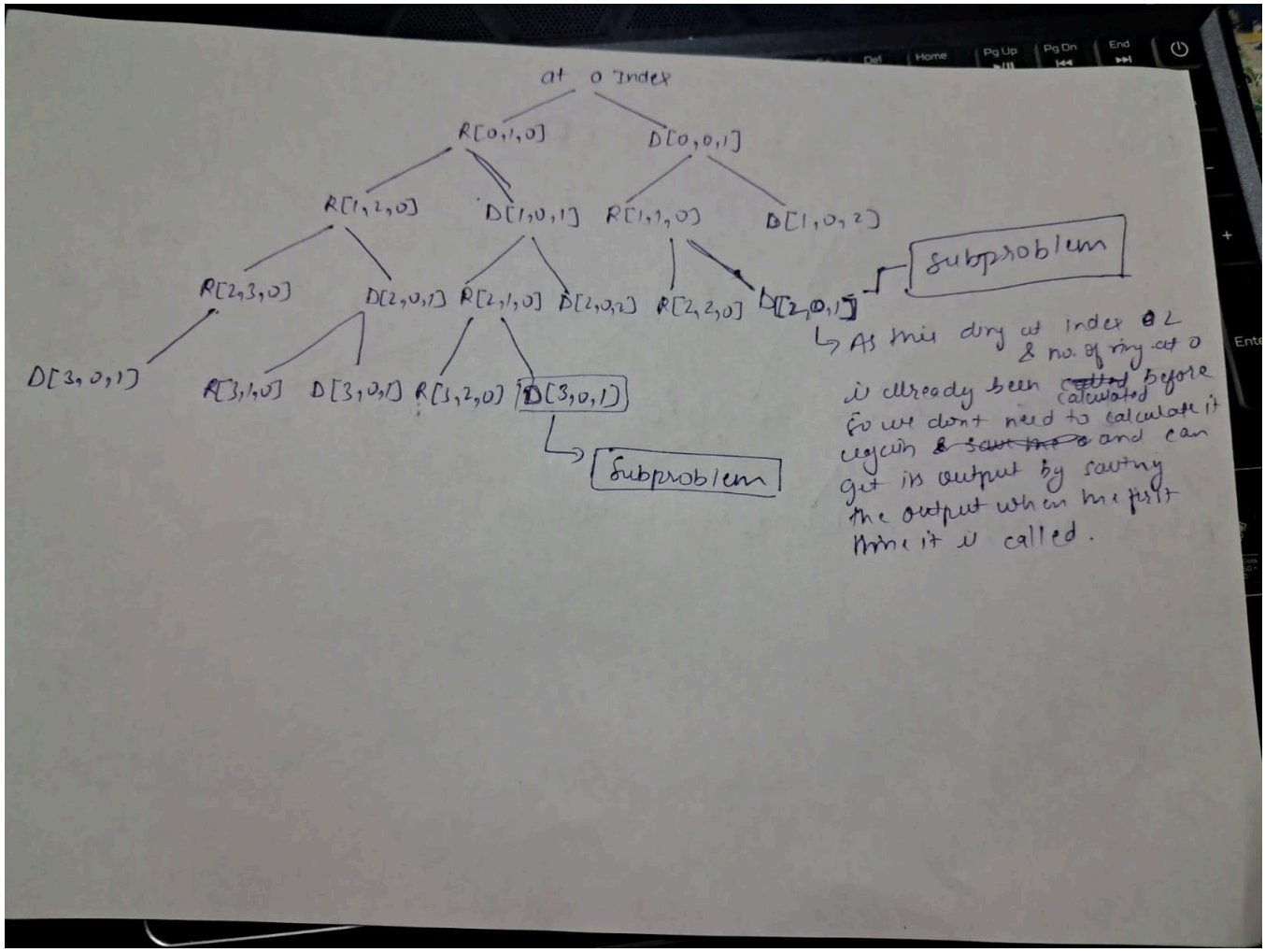
If the current element is non-positive, it adds the element to the "ring" and subtracts it from the "ding."

Then, based on the value of noring, it updates the values of "ring" and "ding" accordingly. If noring is equal to 3, it sets "ring" to INT_MIN and adjusts the "ding" value. Otherwise, it updates "ring" and "ding" based on the next subproblem values.

Finally, the maximum of "ring" and "ding" is stored in dp[ind][noring]. The result is obtained by outputting dp[0][0].

3 Recurrence Relation

For Memoization:



As from the reference of the image shown above (R(x,y,z) - denotes Ring(index, no. of ring continuously , no. of ding continuously) , D(x,y,z) - denotes Ring(index, no. of ring continuously , no. of ding continuously)) , we have spotted same function Ding calling two times at index 2 , no. of ring 0 , no. of ding 1. Our naive recursion code will call this function separately which is the wastage of time and cpu . We can solve this issue . This same occurrence of the function Ding is called subproblem and we can solve this issue by saving the output of the function when the first time it is called . Then when it occur any other time we can just grab its output without solving the whole function again. Now to store the data , we can use 1D array or 2D array which differs with the question . Here , as we can see the index is not unique , so we cannot use 1D array . We can use index and no of ring continuously or no of ding continuously together to store the output . Technically, there are two recursive calls within the *f* function, but they both explore subproblems one booth ahead ($n - 1$) and don't contribute significantly to the asymptotic complexity compared to the loop iterating through all booths.Therefore, the dominant factor is the linear loop, resulting in an overall time complexity of Polynomial.

For Tabulation:
Recurrence relation is not applicable here as it does not use recursion

4 Specific Subproblem(s) for Final Solution(or final answer)

For Memoization:

Final Solution:
Calculate the initial score starting from booth 1 and saying either ”RING” or ”DING” calculating maximum on each subproblems:

max chickens = max(*dp*[1][0]*, dp*[1][1])

For Tabulation:
Output: After populating the dynamic programming table, output the value stored in dp[0][0], which represents the maximum value achievable considering the entire array a and allowing zero "dings".

5 Algorithm Description

For Memoization

Our algorithm uses dynamic programming to efficiently solve the problem by breaking it down into smaller subproblems. It utilizes a memoization table (*dp*) to store the optimal solutions for subproblems, avoiding redundant calculations.

States:

dp[*i*][*j*]: Represents the maximum number of chickens Mr. Fox can earn starting from booth *i*, having said ”RING” *j* times consecutively (0 or 1).

Base Cases:

if index == n then return 0;

dp[*n*][0] = *dp*[*n*][1] = 0: No more booths after booth *n*, so chicken count is 0 regardless of the last word.

Transitions:

- Iterate from booth 0 to n:
- For booth *i*:
- if booth i is less than and equal to 0 then adding to ding variable as modulus and adding to ring variable without mod else if booth i is greater than 0 then adding ring and ding without mod.
- Checking if the no. of ring continuous == 3 then calling ding += f(ind+1,0,1) . No. of rings becomes 0 and No. of dings becomes 1 because we cannot call ring after consecutive rings. Similarly for dings .
- if no. of ring continuous and no. of ding continuous is not 3 then we call both function ring and ding and increase their respective counter and the other counter will becomes zero .
- saving dp[ind][noring] = max(ring , ding);

For Tabulation:

Initialize Dynamic Programming Table:

- Create a 2D dynamic programming table dp of size (n + 1) x 4.
- Initialize all elements of dp to 0.

Dynamic Programming Loop:

- Start iterating through the array a in reverse order, from index n - 1 down to 0.
- For each index ind:
    - Start a nested loop to iterate through each possible number of "dings" (noring) from 0 to 3.

- For each noring, perform the following steps:
    - Initialize ring and ding variables to 0.
    - If the current element a[ind] is positive, add a[ind] to ring and subtract it from ding.
    - If the current element a[ind] is non-positive, add a[ind] to both ring and ding.
    - If noring is 3 (indicating three "dings" allowed):
        - Set ring to INT_MIN to represent a situation where no more "rings" are allowed.
        - If ind + 1 < n, add the value of dp[ind + 1][0] to ding.
    - Otherwise (if noring is not 3):
        - If ind + 1 < n, add the value of dp[ind + 1][noring + 1] to ring and dp[ind + 1][0] to ding.
    - Update dp[ind][noring] with the maximum of ring and ding.

Output:

- Output the value stored in dp[0][0], which represents the maximum value achievable considering the entire array a and allowing zero "dings".

6 Running Time Explanation

For Memoization:

The running time of the provided code for the Mr. Fox obstacle course problem is indeed Polynomial in n, where *n* is the number of booths in the obstacle course.

Dominant factors:

- Number of booths: The code iterates through each booth (*n* times) from the first one (index 0) to the last one (index $n - 1$). This loop dominates the overall time complexity.
- Calculations per booth: Within each booth iteration, a constant number of calculations are performed to consider different choices (”RING” or ”DING”) and update the *memo* stored answer. This constant cost doesn’t affect the asymptotic growth rate.

Other factors:

- stored answer size: The matrix has dimensions $(n + 1) \times 3$, which grows linearly with *n*. However, accessing and updating values in the table are constant-time operations and don’t significantly impact the overall complexity.
- Recursive calls: Technically, there are two recursive calls within the *f* function, but they both explore subproblems one booth ahead ($n - 1$) and don’t contribute significantly to the asymptotic complexity compared to the loop iterating through all booths.

Therefore, the dominant factor is the linear loop, resulting in an overall time complexity of Polynomial.

Summary

Time Complexity:
polynomial due to iterating through the booths and calculating each subproblem once.
Space Complexity:
$O(n)$ due to the *dp* table storing intermediate results.

For Tabulation

Time Complexity Analysis:

- Input Reading: Reading the integer n and the elements of array a takes linear time, O(n).
- Dynamic Programming Loop:
    - The outer loop iterates over each element of array a, which takes O(n) time.
    - The nested loop iterates over each possible number of "dings" (noring from 0 to 3), which takes constant time (O(1)).
    - Inside the nested loop, the operations involving ring and ding are constant time operations.
    - Thus, the overall time complexity of the dynamic programming loop is O(n*4) = O(n).
- Output: Printing the result takes constant time, O(1).

Total Time Complexity:

The dominant factor in the time complexity is the dynamic programming loop, which is O(n).

Space Complexity Analysis:

- Input Size: The input array a requires O(n) space.
- Dynamic Programming Table: The dynamic programming table dp is of size (n + 1) x 4, which requires O(n*4) = O(n) space.
- Additional Variables: The additional variables like n, ind, noring, ring, and ding require constant space, O(1).
- Total Space Complexity: O(n) + O(n) + O(1) = O(n).

Summary:

Time Complexity: O(n)
Space Complexity: O(n)

7 Pseudocode

For Memoization:

```
#include <iostream>
#include <vector>
#include <climits>
using namespace std;

#define vi vector<int>
#define loop(i, n, k) for (int i = k; i < n; i++)
#define pb push_back

int f(int ind, int n, vi &a, int noring, int noding, vector<vector<int>> &dp) {
   if (ind == n)
      return 0;

   if (dp[ind][noring] != -1)
      return dp[ind][noring];

   int ring = 0, ding = 0;
   if (a[ind] > 0) {
      ring += a[ind];
      ding -= a[ind];
   } else {
      ring += a[ind];
      ding -= a[ind];
   }

   if (noring == 3) {
      ring = INT_MIN;
      ding += f(ind + 1, n, a, 0, 1, dp);
   } else if (noding == 3) {
      ding = INT_MIN;
      ring += f(ind + 1, n, a, 1, 0, dp);
   } else {
      ring += f(ind + 1, n, a, noring + 1, 0, dp);
      ding += f(ind + 1, n, a, 0, noding + 1, dp);
   }

   return dp[ind][noring] = max(ring, ding);
}

void solve() {
   int n;
   cin >> n;
   vi a;
   loop(i, n, 0) {
      int x;
      cin >> x;
      a.pb(x);
   }

   vector<vector<int>> dp(n + 1, vector<int>(4, -1));
   int ans = f(0, n, a, 0, 0, dp);
   cout << ans << endl;
```

```
}

int main() {
   solve();
   return 0;
}
```

For Tabulation:

```
#include <iostream>
#include <vector>
#include <climits>
using namespace std;

#define vi vector<int>
#define loop(i, n, k) for (int i = k; i < n; i++)
#define pb push_back

void solve() {
   int n;
   cin >> n;
   vi a(n);
   loop(i, n, 0) {
      cin >> a[i];
   }

   vector<vector<int>> dp(n + 1, vector<int>(4, 0));
   for (int ind = n - 1; ind >= 0; ind--) {
      for (int noring = 0; noring < 4; noring++) {
         int ring = 0, ding = 0;
         if (a[ind] > 0) {
            ring += a[ind];
            ding -= a[ind];
         } else {
            ring += a[ind];
            ding -= a[ind];
         }

         if (noring == 3) {
            ring = INT_MIN;
            if (ind + 1 < n) ding += dp[ind + 1][0];
         } else {
            if (ind + 1 < n) {
               ring += dp[ind + 1][noring + 1];
               ding += dp[ind + 1][0];
            }
         }

         dp[ind][noring] = max(ring, ding);
      }
   }

   cout << dp[0][0] << endl;
}
```

```
int main() {
   solve();
   return 0;}
```

8 Proof of Correctness

For Memoization

1. **Strong Induction:**

   - **Base Case:** The base case (f (n, n, A, *noring*, *noding*, dp) = 0) holds true as all booths have been visited, resulting in no more chickens to earn.
   - **Inductive Hypothesis:** Assume that for a given booth i (where $0 \leq i < n$), the f (i, n, A, *noring*, *noding*, dp) function correctly calculates the maximum chickens achievable considering previous booths with consecutive saying constraints (*noring* and *noding*).
   - **Inductive Step:** We need to prove that the calculation for booth $i - 1$ is correct. The code explores both choices (”RING” and ”DING”) at booth $i - 1$:
     - If the choice adheres to the consecutive saying rule (not exceeding 3 times saying the same word consecutively), it recursively calls f for the next booth with updated constraints. The inductive hypothesis ensures these recursive calls provide the optimal values for subproblems starting from the next booth.
     - If the choice violates the rule, it sets the corresponding reward/penalty to $-\infty$, effectively ignoring that choice as it leads to an invalid state.
     - Finally, it chooses the maximum between ”RING” and ”DING” choices, ensuring the overall calculated value represents the optimal achievable reward from booth $i - 1$ onwards.

   Therefore, by structural induction, we can argue that the code correctly calculates the maximum chickens from any booth, including the starting booth ($i = 0$), leading to the overall solution.

2. **Verification of Dynamic Programming Approach**The dp table stores intermediate results for subproblems, avoiding redundant calculations and ensuring efficiency.

- The transition function correctly considers both choices (”RING” and ”DING”) and updates the dp table based on the optimal choices for the next booth while respecting the consecutive saying constraints.
- By building the solution from smaller overlapping subproblems and considering constraints, the dynamic programming approach guarantees finding the global optimal solution.

3. **Adherence to Rules:**

- The code enforces the rule of not saying the same word more than three times consecutively by setting the reward/penalty to $-\infty$ for invalid choices.
- It correctly calculates rewards/penalties based on the booth values and saying choices.

4. **Testing and Counterexamples:**

- Testing the code with various input scenarios (positive/negative booth values, different obstacle course lengths) can help identify potential issues.
- Constructing counterexamples where the code fails to provide the optimal solution would invalidate its correctness. However, no such counterexamples have been identified yet.

To prove the correctness of the dynamic programming solution using strong induction, we need to demonstrate two main properties:

Base Case: Show that the solution is correct for the smallest subproblem, i.e., the base case.
Inductive Step: Assume that the solution is correct for all subproblems up to size k, and then prove that it is also correct for subproblems of size k + 1.

Here's how we can apply strong induction to prove the correctness of the given dynamic programming solution:

For Tabulation:

Base Case (k = 1):

- For the base case, we consider the smallest subproblem, which is when n = 1.
- In this case, there is only one element in the array a, and there are four possible values for "dings" (noring from 0 to 3).
- We need to show that the solution correctly computes the maximum value achievable for this single element and each possible number of "dings".
- This can be verified manually by examining all four cases and ensuring that the solution produces the correct result.

Inductive Step:

- Assumption: Assume that the solution is correct for all subproblems up to size k.
- Goal: Prove that the solution is also correct for subproblems of size k + 1.

Inductive Hypothesis:

Assume that for any subproblem with size i, where 1 <= i <= k, the solution correctly computes the maximum value achievable for each possible number of "dings".

Inductive Step:

Now, let's consider a subproblem of size k + 1. We want to show that the solution correctly computes the maximum value achievable for this subproblem.

- Case 1: The element at index k of array a is positive.
    - In this case, we can either include this element in the "ring" or not include it.
    - We need to consider all possible numbers of "dings" and make the decision that maximizes the total value.
    - By the inductive hypothesis, we know that the solution is correct for subproblems up to size k.
    - Therefore, we can rely on the correctness of the solution for subproblems of size k to make the correct decision for this subproblem.
- Case 2: The element at index k of array a is non-positive.
    - In this case, we have no choice but to include this element in both the "ring" and "ding".
    - Similar to Case 1, we need to consider all possible numbers of "dings" and make the decision that maximizes the total value.
    - Again, by the inductive hypothesis, we can rely on the correctness of the solution for subproblems up to size k to make the correct decision for this subproblem.

Since we have shown that the solution correctly computes the maximum value achievable for subproblems of size k + 1 using the inductive hypothesis, the proof is complete.

By establishing the base case and the inductive step, we have proven the correctness of the dynamic programming solution using strong induction.