Theory Assignment-2: ADA Winter-2024

Palak Bhardwaj(2022344) Yashovardhan Singhal(2022591)

February 11, 2024

1 Pre-Processing

In the provided algorithm, there isn't explicit preprocessing involved before executing the main part of the algorithm.
However, we initalised a DP Table to store the values of calculated subproblems in order to implement memorization, dp with dimensions `(n+1) x 4 x 4` is initialized with all values set to -1 before invoking the recursive function chickened.
Input Format: An array A[1, 2, ..., n] representing the numbers written at the door of each booth and The size n of the array.

2 Algorithm Description

Here's a step-by-step breakdown of the algorithm implemented:

1. **Function Definition**: Define a function `chickend` that takes five parameters: `index` (current booth index), `r` (number of consecutive 'RING's), `d` (number of consecutive 'DING's), `arr` (vector representing rewards/penalties at each booth), and `dp` (3D vector for memoization).
2. **Base Case**: If `index` equals the size of the `arr` vector, return 0, indicating that Mr. Fox has reached the end of the obstacle course.
3. **Memoization Check**:
   - Check if the memoization value for the current parameters (`index`, `r`, `d`) is already calculated. If yes, return the memoized value.
4. **Recursive Cases**:
   - If `d` equals 3 (maximum consecutive 'DING's reached), recursively call `chickend` with `index+1`, `r=1`, `d=0`, `arr`, and `dp`, and add the current booth's reward/penalty (`arr[index]`).
   - If `r` equals 3 (maximum consecutive 'RING's reached), recursively call `chickend` with `index+1`, `r=0`, `d=1`, `arr`, and `dp`, and subtract the current booth's reward/penalty (`arr[index]`).
   - Otherwise, recursively call `chickend` twice:
     - Increment `r` by 1 and call with `d=0`.
     - Increment `d` by 1 and call with `r=0`.
     - Return the maximum value obtained by adding or subtracting the current booth's reward/penalty.

This algorithm recursively explores different paths through the obstacle course, considering the maximum number of chickens Mr. Fox can earn based on the sequence of 'RING's and 'DING's he utters. The memoization array `dp` is used to store and re-use previously computed results, optimizing the algorithm by avoiding redundant calculations.

3 Subproblem

Using a Top-Down Approach: The subproblem addressed by the algorithm is to determine the maximum number of chickens that Mr. Fox can earn while navigating the obstacle course. This involves considering the current booth index, the consecutive occurrences of 'RING's (denoted by r), the consecutive occurrences of

'DING's (denoted by d), the array representing rewards/penalties at each booth, and the previously computed solutions for subproblems stored in the dp array.

The subproblem is recursively defined as follows: Let

$$chickend(index, r, d, arr, dp)$$

represent the maximum number of chickens that Mr. Fox can earn by starting from the $index$-th booth, with r consecutive 'RING's and d consecutive 'DING's so far, utilizing the rewards/penalties array arr, and leveraging the previously computed solutions stored in the dp array.

4 Recurrence Relation

The recurrence relation for the `chickend` function can be expressed as follows:

$$\text{chickend}(index, r, d, \text{arr}) = \begin{cases} 0, & \text{if index} = \text{arr.size()} \\ \max \begin{pmatrix} \text{chickend}(index+1, r+1, 0, \text{arr}) + \text{arr}[index], \\ \text{chickend}(index+1, 0, d+1, \text{arr}) - \text{arr}[index] \end{pmatrix}, & \text{otherwise} \end{cases}$$

Base Case:

$$\text{if } (index == n)\{ \text{ return } 0; \}$$

5 Final Subproblem

The final subproblem in the dynamic programming array dp would be represented as

$$dp[n][r][d]$$

where: - n is the size of the arr vector, - r represents the number of Consecutive Dings where - d represents the number of Consecutive Rings

6 Pseudo-Code

```
function CHICKEND(index, r, d, arr, dp)
    if index = arr.size() then
        return 0
    else
        if dp[index][r][d] ≠ −1 then
            return dp[index][r][d]
        end if
        if d = 3 then
            return Chickend(index + 1, 1, 0, arr, dp) + arr[index]
        end if
        if r = 3 then
            return Chickend(index + 1, 0, 1, arr, dp) − arr[index]
        end if
        dp[index][r][d] ← max(
            Chickend(index + 1, r + 1, 0, arr, dp) + arr[index],
            Chickend(index + 1, 0, d + 1, arr, dp) − arr[index])
        return dp[index][r][d]
    end if
end function
```

7 Space Complexity Analysis

The space complexity of the `chickend` function can be divided into two components:

1. **Function Stack Space:**

- Due to the recursive nature of the `chickend` function, space on the call stack is allocated for each recursive call.
- The maximum depth of recursion is determined by the size of the input vector `arr`.
- Therefore, the space complexity for the function stack is $O(n)$, where n represents the size of the `arr` vector.

2. **Dynamic Programming Memoization:**

- The function utilizes a memoization table `dp`, which is a 3D vector with dimensions `arr.size()` $\times 4 \times 4$.
- The space complexity for this memoization table is proportional to the size of the input vector `arr`, resulting in $O(n)$ space requirements.

Hence, the total space complexity of the `chickend` function, accounting for both the function stack space and the dynamic programming memorization is O(n) + O(n) = O(n), indicating that the total space complexity of the chickend function is **O(n).**

8 Time Complexity

1. **Subproblem Identification**:
   - Define a function $T(i, r, d)$ representing the time complexity of solving a subproblem with parameters (i, r, d).
   - i denotes the index, r denotes the count of consecutive elements selected from the beginning, and d denotes the count of consecutive elements not selected from the beginning.
   - With r and d bounded by 4, the total number of possible combinations of (i, r, d) is $n \times 4 \times 4$, where n is the size of the input array arr.
2. **Memoization**:
   - Utilize memoization to store the solutions of previously computed subproblems.
   - Retrieving a solution from the memoization table takes constant time ($O(1)$).
3. **Recurrence Relation**:
   - Define a recurrence relation for $T(i, r, d)$ based on the algorithm's recursive calls.
   - If dp$[i][r][d]$ already contains a value, then $T(i, r, d) = O(1)$.
   - Otherwise, the algorithm makes recursive calls to subproblems, leading to further exploration.
4. **Overall Time Complexity**:
   - Consider the number of distinct subproblems that need to be solved.
   - The maximum number of distinct subproblems is $n \times 4 \times 4$.
   - Each subproblem is solved only once due to memoization.
   - Therefore, the overall time complexity can be derived by summing up the time complexities of all distinct subproblems.
5. **Simplify**:
   - Since each subproblem is solved once, the time complexity of solving all distinct subproblems is $O(n)$.
   - Hence, the overall time complexity of the algorithm using memoization is $O(n)$.

9 Assumptions

The analysis of the algorithm's time complexity is based on the following assumptions:

1. The input array arr is of size n.
2. The parameters r and d are bounded by 4, i.e., $0 \leq r, d \leq 3$.
3. The memoization table dp is used to store and retrieve solutions of previously computed subproblems in constant time.
4. Recursive calls to subproblems are made based on the recurrence relation, with each subproblem being solved only once due to memoization.

10 Proof of Correctness

Since the algorithm uses dp, there isn't an explicit requirement for a proof of correctness but here is a general proof,

10.0.1 Base Case

When the index equals the size of the input array (index = arr.size()), the algorithm correctly returns 0. This is because there are no more booths to traverse.

10.0.2 Recurrence Relation

Base Case ($n = 1$): For the base case, consider the scenario where there is only one booth ($n = 1$). In this case, the algorithm computes the maximum reward/penalty for this single booth based on the constraints of consecutive 'RING's and 'DING's. Since there are no preceding booths, the algorithm correctly handles this case by returning the reward/penalty of the single booth.

Inductive Step ($n > 1$): Assume that the algorithm correctly computes the maximum number of chickens for subproblems of size $n - 1$. We will prove that it also correctly computes the maximum number of chickens for subproblems of size n.

Let's consider a subproblem of size n. The algorithm considers all possible cases for the current booth:

- If the maximum consecutive 'DING's (d) has been reached, the algorithm correctly moves to the next booth with $r = 1$ and $d = 0$, ensuring that Mr. Fox starts counting consecutive 'RING's again from the next booth.
- If the maximum consecutive 'RING's (r) has been reached, the algorithm correctly moves to the next booth with $r = 0$ and $d = 1$, ensuring that Mr. Fox starts counting consecutive 'DING's from the next booth.
- Otherwise, the algorithm explores both possibilities:
  - Incrementing r by 1 and moving to the next booth with $d = 0$.
  - Incrementing d by 1 and moving to the next booth with $r = 0$.

  The algorithm then selects the maximum reward/penalty from these two possibilities, ensuring that the maximum number of chickens is computed accurately for the current booth.

By establishing the base case and demonstrating that the algorithm correctly handles subproblems of size n, assuming it works for subproblems of size $n - 1$, we can conclude that the algorithm computes the maximum number of chickens accurately for all subproblems of size n.

Hence, the algorithm is correct.

11 Resources

Jeff Erickson (Book)
DP series by Striver Link 1
DP series by Striver Link 2