Theory Assignment-2: ADA Winter-2024

Devansh Grover (2022151)   Garvit Kochar (2022185)

1 Subproblem Definition

Given the array que[1, 2, . . . , n] (not necessarily sorted) representing the numbers at the doors of the booths, a sub-problem is to determine the maximum number of chickens that Mr. Fox can earn by visiting booths from i to n, given that he has said RING cntR times in a row and DING cntD times in a row.
The subproblem definition is based on the state parameters ***i*, cntR, and cntD**. So, the definition is, given an index ***i***, the count of consecutive **RINGS** denoted by **cntR**, and the count of consecutive **DINGS** denoted by **cntD**, determine the maximum score attainable for the subsequence of the given array starting from index **i** while ensuring that the counts of consecutive RINGS and DINGS do not exceed **3**. Therefore, the subproblems involve computing the maximum score using RINGS and DINGS until we reach the base cases.

Current index *i*: This represents our current position in the array. As we make decisions, we move forward in the array, so ***i*** increases. When ***i* equals the size** of the array, we've reached the end and the **base case** of the recursion is triggered.

cntR: This keeps track of how many times we've consecutively decided to add the current integer to our result. It's incremented when we make a RING decision and resets to 0 when we make a DING decision. It can't exceed 3, which is a constraint of the problem.

cntD: This keeps track of how many times we've consecutively decided to subtract the current integer from our result. It's incremented when we make a DING decision and resets to 0 when we make a RING decision. It also can't exceed 3, which is another constraint of the problem.

Given these three variables, the subproblem is to determine the maximum possible sum we can obtain from the subsequence of the array starting from index i, while ensuring that we don't make more than three consecutive RING or DING decisions.
This subproblem is solved recursively, with the results stored in a **memoization table** for reducing the time complexity. The solution to the original problem is then constructed from the solutions to these subproblems.

2 Reccurence of the subproblem

For the given input array que[] with size $'size'$, let dp[i][cntR][cntD] be the maximum result we can obtain at index i with cntR consecutive RINGS and cntD consecutive DINGS. Then, the recurrence relation is:

$$\text{dp[i][cntR][cntD]} = \begin{cases} 0 & \text{if } i == size \\ \max \begin{cases} \text{dp[i+1][cntR+1][0] + que[i]} \\ \text{dp[i+1][0][cntD+1] - que[i]} \end{cases} & \begin{matrix} \text{if cntR} < 3 \\ \text{if cntD} < 3 \end{matrix} \end{cases}$$

Therefore, mathematically it can be represented as:

$$f(i, size, r, d) = \begin{cases} 0 & \text{if } i == size \\ \max \begin{cases} f(i+1, size, r+1, 0) + \text{que}[i] \\ f(i+1, size, 0, d+1) - \text{que}[i] \end{cases} & \begin{matrix} \text{if r} < 3 \\ \text{if d} < 3 \end{matrix} \end{cases}$$

3 Specific Subproblem(s) that solve the actual Problem

Subproblem 1: The maximum number of chickens that Mr. Fox can earn from booth i onwards if he has already said RING cntR times in a row and DING cntD times in a row.
Subproblem 2: The maximum number of chickens that Mr. Fox can earn from booth $i + 1$ onwards if he says RING at booth i (and thus has said RING cntR + 1 times in a row and DING 0 times in a row).
Subproblem 3: The maximum number of chickens that Mr. Fox can earn from booth $i + 1$ onwards if he says DING at booth i (and thus has said RING 0 times in a row and DING cntD + 1 times in a row).

The base case for the recursion is when $i == size$, which means Mr. Fox has visited all the booths. In this case, the function returns 0 because there are no more chickens to be earned. If the solution to a subproblem has already been computed (i.e., $dp[i][cntR][cntD]! = -1$), the function returns the stored solution. Otherwise, it computes the solution by considering the two possibilities (saying RING or DING at booth i) and taking the maximum of the two. The solution to the subproblem is then stored in $dp[i][cntR][cntD]$ for future reference.

The solution to the original problem is the maximum number of chickens that Mr. Fox can earn from booth 1 onwards, which is given by dp[0][0][0]. The 'dp' array is used to store the solutions to the subproblems to avoid redundant computation.

This algorithm ensures that Mr. Fox earns the maximum number of chickens while adhering to the rules of the game. It runs in polynomial time because each subproblem is solved only once and the number of subproblems is polynomial in n (the number of booths).

4 Algorithm Description

Pre-processing: Create the array dp with dimensions $[size + 1][4][4]$. This will act as our memoization table. Set all the values in this table to -1. This may be done in main using the memset function.

Function ringDing: This is a recursive function that takes the input array , its size, the counts of the last decisions (cntR and cntD), the current index i, and the memoization table dp. It returns the maximum possible result from index i onwards.

Base Case: If $i == size$, it means we've reached the end of the array, so the function returns 0.

Memoization: If dp[i][cntR][cntD] is not -1, it means we've already computed the result for this state, so we return it to avoid redundant computation using dp array.

Recursive Case: If cntR is less than 3, we consider the decision of adding the current integer to the result and make a recursive call with cntR + 1 and cntD reset to 0. If cntD is less than 3, we consider the decision of subtracting the current integer from the result and make a recursive call with cntD + 1 and cntR reset to 0. We take the maximum of these two possibilities and store it in dp[i][cntR][cntD].

The ringDing function is called recursively for each state (i, cntR, cntD). Each state represents a subproblem, which is to find the maximum possible result from index i onwards, given the counts of the last decisions cntR and cntD. At each state, we have two options: make a RING decision (add the current integer to the result) or a DING decision (subtract it, i.e, if the integer is negative, add its magnitude and if it is positive, subtract its magnitude). However, we can't make more than three consecutive RING or DING decisions, which is enforced by the conditions cntR < 3 and cntD < 3. If we make a RING decision, we transition to the state (i + 1, cntR + 1, 0). If we make a DING decision, we transition to the state (i + 1, 0, cntD + 1). We make a recursive call to the ringDing function for each of these states. The result of the current state is the maximum of the results of the two possible transitions. This is calculated as max*(que[i] + ringDing(que, size, cntR + 1, 0, i + 1, dp),*

-que[i] + ringDing(que, size, 0, cntD + 1, i + 1, dp)). **Before making the recursive calls, we check if we've already computed the result for the current state in the DP table. If dp[*i*][cntR][cntD] is not -1, it means we've already solved this subproblem, so we return the stored result instead of making redundant recursive calls.** This is where the DP table is used to save recursive calls and reduce time complexity. After calculating the result of the current state, we store it in the DP table as dp[i][cntR][cntD] = res. This way, if we encounter the same state again in a future recursive call, we can simply return the stored result, avoiding redundant computation.

The final result of the problem is the result of the initial state (0, 0, 0).

Explanation of the input parameters:
1. **int que[]:** This is the input array.
2. **int size:** This is the size of the que[] array.
3. **int cntR:** This acts as a counter for the number of consecutive RINGS.
4. **int cntD:** This acts as a counter fot the number of consectutive DINGS.
5. **int *i*:** This represents the current index of the que[] array.
6. **int dp[][4][4]:** This is a 3D array used for dynamic programming.

   i. The first dimension, represented by [], will be **initialized by size+1 in main**. It corresponds to the variable i, which represents the current index of the array que[] that we're considering.
   ii. The second and third dimensions are of size 4, corresponding to the variables **cntR** and **cntD** respectively. These represent the number of consecutive **RINGS** and **DINGS**.
   iii. The array **dp needs to be initialized to -1 in main** before running the algorithm, which represents an uncomputed state. This may be done using the memset function in main.

Algorithm 1 Pseudocode

```
function RINGDING(que[], size, cntR, cntD, i, dp[][4][4])
1.      if i == size:
2.          return 0
3.      if dp[i][cntR][cntD] is not -1:
4.          return dp[i][cntR][cntD]
5.      res ← INT_MIN
6.      if cntR < 3:
7.          res ← max(res, que[i] + solve(que, size, cntR + 1, 0, i + 1, dp))
8.      if cntD < 3:
9.          res ← max(res, -que[i] + solve(que, size, 0, cntD + 1, i + 1, dp))
10.     dp[i][cntR][cntD] ← res
11.     return res
end function
```

Note: We are using zero based indexing. Therefore, i can be initialized to 0 if we want to start from the beginning of the input array. This is also why cntR < 3 and cntD < 3 is used instead of cntR < 4 and cntD < 4.

5 Running time of the Algorithm

The running time of this algorithm is $O(size \cdot 4^2)$, where $size$ is the size of the array 'que' and 4^2 is the number of values cntR and cntD can take. Since constants can be dropped in asymptotic analysis, the time complexity of this algorithm is $O(n)$.

Explanation:
The function ringDing is called recursively with different states.
The states are determined by the variables i, cntR and cntD.
Let $size = n$. The variable i can take up to n different values (from 0 to size-1), and both cntR and cntD can take up to 4 different values (from 0 to 3).

We notice that the algorithm does the following:
1. Recursive Calls: We start from the first element of the que array and make recursive calls to the solve function for each subsequent element. This continues until we reach the last element of the que[].
2. Base Case: Once we reach the last element, the base case of the recursion is triggered ($when\ i == size$), and the function returns 0. This represents the maximum value we can get from an empty subset of the que array.
3. Memoization: As we return from the recursive calls, we calculate the maximum value for each state (i, cntR, cntD) and store it in the *dp* array. If we encounter the same state again in a future recursive call, we can simply return the stored value instead of making redundant computations. This is where the *dp* array is used.

The dynamic programming table dp has dimensions [size+1][4][4], where *size* is the length of the que[], and 4 and 4 represent the number of possible values of cntR and cntD respectively. Each cell in this table represents a unique state of the problem, defined by the current index in the que[] and the counts of the last decisions. During the execution of the program, each state is visited only once, thanks to memoization. When a state is visited, its result is computed and stored in the corresponding cell of the dp array. If the same state is encountered again in a future recursive call, the stored result is returned immediately, avoiding redundant computation. So, even though the function ringDing is called recursively for each element in the que[], the actual computation for each state is done only once. Therefore, the total number of computations is proportional to the number of states, which is size n * 4 * 4 = 4n. In essence, we're building up the solution to the problem by solving smaller subproblems and storing their solutions for future use. Each state along the path is computed only once. The backtracking from the last element to the first is just returning the computed results, not doing any new computations.

Therefore, the total number of unique states is $n \cdot 4 \cdot 4$.
Since $4 \cdot 4$ is a constant, it can be ignored in the asymptotic analysis. Also, since each state is computed only once due to memoization, the time complexity is proportional to the number of states.
Hence, the time complexity is $O(n)$.

$[\ \pi_{distributor_id}\ (\sigma$