Theory Assignment-2: ADA Winter-2024

Aarzoo (2022008)   Anushka Srivastava (2022086)

1 Subproblem Definition

The subproblem definition in the problem is:

r : ring
d : ding

$r_1 \quad r_2 \quad r_3 \quad \dots \quad r_{i-3} \quad r_{i-2} \quad r_{i-1} \quad r_i \quad r_{i+1} \quad \dots \quad r_n$
$d_1 \quad d_2 \quad d_3 \quad \dots \quad d_{i-3} \quad d_{i-2} \quad d_{i-1} \quad d_i \quad d_{i+1} \quad \dots \quad d_n$

ring(i): The largest number of chickens that Mr. Fox earns by running the obstacle course from $A(1, \dots, i)$ if at the $\mathbf{i^{th}}$ index Mr. Fox chooses to **Ring** given the constraints that Mr. Fox is forbidden to say the same word more than three times a row.
ding(i): The largest number of chickens that Mr. Fox earns by running the obstacle course from $A(1, \dots, i)$ if at the $\mathbf{i^{th}}$ index Mr. Fox chooses to **Ding** given the constraints that Mr. Fox is forbidden to say the same word more than three times a row.

2 Recurrence of the sub-problem

$$Ring[i] = \{A[i-1] + \max(Ding[i-1], Ring[i-2], Ding[i-2], Ring[i-3], Ding[i-3])\} \tag{1}$$
$$Ding[i] = \{-A[i-1] + \max(Ring[i-1], Ring[i-2], Ding[i-2], Ring[i-3], Ding[i-3])\} \tag{2}$$

- **Case 1:** $Ring[i]$ = A[i-1] + $\max(Ding[i-1], Ring[i-2], Ding[i-2], Ring[i-3], Ding[i-3])$
  The recurrence relation considers the 7 possible cases (DDDR, RRDR, DDRR, RDRR, RDDR, DRRR, DRDR) excluding the case of 4 consecutive Rings, i.e., (RRRR), which determines the maximum number of chickens at the i^{th} index where at i^{th} position Mr. Fox chooses to Ring.
  - $Ding[i-1]$ represents consecutive Dings till $i-1$ index from $i-3$ index (DDDR).
  - $Ring[i-2]$ represents consecutive Rings till $i-2$ index from $i-3$ index (RRDR).
  - $Ding[i-2]$ represents consecutive Dings till $i-2$ index from $i-3$ index (DDRR).
  - $Ring[i-3]$ represents consecutive Rings till $i-3$ index from $i-3$ index (RDRR, RDDR).
  - $Ding[i-3]$ represents consecutive Dings till $i-3$ index from $i-3$ index (DRRR, DRDR).
- **Case 2:** $Ding[i]$ = -A[i-1] + $\max(Ring[i-1], Ring[i-2], Ding[i-2], Ring[i-3], Ding[i-3])$
  The recurrence relation considers the 7 possible cases (RRRD, DDRD, RRDD, DRDD, DRRD, RDDD, RDRD) excluding the case of 4 consecutive Rings, i.e., (RRRR), which determines the maximum number of chickens at the i^{th} index where at i^{th} position Mr. Fox chooses to Ring.
  - $Ring[i-1]$ represents consecutive Rings till $i-1$ index from $i-3$ index (RRRD).
  - $Ring[i-2]$ represents consecutive Rings till $i-2$ index from $i-3$ index (RRDD).
  - $Ding[i-2]$ represents consecutive Dings till $i-2$ index from $i-3$ index (DDRD).
  - $Ring[i-3]$ represents consecutive Rings till $i-3$ index from $i-3$ index (RDRD, RDDD).
  - $Ding[i-3]$ represents consecutive Dings till $i-3$ index from $i-3$ index (DRRD, DRDD).

3 Specific subproblem that solves the actual problem

The algorithm is based on the principle of filling up the tables, i.e., tabulation.

The subproblem that solves the final problem is: $\max(\mathbf{Ring[n]}, \mathbf{Ding[n]})$ where $Ring[n]$ represents the case when Mr. Fox chooses to Ring at the n^{th} index to achieve the largest number of chickens, and $Ding[n]$ represents the case when Mr. Fox chooses to Ding at the n^{th} index to achieve the largest number of chickens.

4 Algorithm Description

The algorithm is based on 1-based indexing for *ring* and *ding*.

Base Cases: The base cases are defined until the array size 2 because of the given constraint that there cannot be more than 3 consecutive Rings or Dings.

- If the length of the array A is less than 1, return -1.
- If the length of the array A is 1, return $\max(A[0], -A[0])$.
- If the length of the array A is 2, return $\max(A[0] + A[1], A[0] - A[1], -A[0] + A[1], -A[0] - A[1])$.

Dynamic arrays initialization: There are 2 dp arrays: *ring* and *ding*, each initialized with the size $n + 1$ (where n represents the size of array A). The dp arrays are initialized with indexes $0, 1, 2$ and 3 based on the recurrence relation. For index 0, the *ring* and *ding* arrays are initialized to 0 at index 0 to follow 1-based indexing. The corresponding indices are initialized for indexes 1 and 2, similar to the base case. For index 3, the $ring[3]$ and $ding[3]$ are initialized with the maximum value possible when either Ring or Ding is spoken at index 3.

Filling up the tables: The $forloop$ runs from 4 to $n + 1$ considering the 7 possible cases for Ring and Ding and excluding the case for 4 consecutive rings and dings. In the for loop, we are building up on our dp base cases such that we get the maximum value possible at each index.

- $prev\_prev\_ring = ring[i-2] - A[i-2]$: represents the case of $\_R\_\_$ where the dashes can take any value from R and D.
- $prev\_prev\_ding = ding[i-2] + A[i-2]$: represents the case of $\_D\_\_$ where the dashes can take any value from R and D.
- $prev\_prev\_prev\_ring = \max(ring[i-3] - A[i-3] + A[i-2], ring[i-3] - A[i-3] - A[i-2])$: represents the case of $R\_\_\_$ where dashes can take any value from R and D.
- $prev\_prev\_prev\_ding = \max(ding[i-3] + A[i-3] + A[i-2], ding[i-3] + A[i-3] - A[i-2])$: represents the case of $D\_\_\_$ where dashes can take any value from R and D.

For the case $\_\_D\_$, we set $prev\_ding$ as $ding[i-1]$. We take the max of these values and add the value of $A[i]$ if Ring is spoken at the i^{th} index. This value is then stored at $ring[i]$.
For the case $\_\_R\_$, we set $prev\_ring$ as $ring[i-1]$. We take the max of these values and add the value of $A[i]$ if Ding is spoken at the i^{th} index. This value is then stored at $ding[i]$.
Hence, at i^{th} position of both of these arrays, we have the optimized solution of the case if either Ring or Ding is spoken at that position.
Finally, we return the maximum of $ring[n]$ and $ding[n]$, which is the maximum number of chickens that can be earned if we speak Ring or Ding at the final index.

Final solution: Returned by the $\max(ring[n], ding[n])$.

5 Pseudocode

Illustrated below

Algorithm 1 Bottom up Approach using Tabulation

```
function MaxChickensEarned(n, A)
    if n < 1 then
        return -1
    end if
    if n == 1 then
        return max(A[0], −A[0])
    end if
    if n == 2 then
        return max(A[0] + A[1], A[0] − A[1], −A[0] + A[1], −A[0] − A[1])
    end if
    ring ← array of length n + 1 initialized with 0s
    ding ← array of length n + 1 initialized with 0s
    ring[0] ← 0
    ding[0] ← 0
    ring[1] ← A[0]
    ding[1] ← −A[0]
    ring[2] ← max(ring[1] + A[1], ding[1] + A[1])
    ding[2] ← max(ring[1] − A[1], ding[1] − A[1])
    ring[3] ← max(ring[2] + A[2], ring[1] − A[1] + A[2], ding[1] + A[1] + A[2], ding[2] + A[2])
    ding[3] ← max(ring[2] − A[2], ring[1] − A[1] − A[2], ding[1] + A[1] − A[2], ding[2] − A[2])
    for i ← 4 to n + 1 do
        prev_prev_ring ← ring[i − 2] − A[i − 2]
        prev_prev_ding ← ding[i − 2] + A[i − 2]
        prev_prev_prev_ring ← max(ring[i − 3] − A[i − 3] + A[i − 2], ring[i − 3] − A[i − 3] − A[i − 2])
        prev_prev_prev_ding ← max(ding[i − 3] + A[i − 3] + A[i − 2], ding[i − 3] + A[i − 3] − A[i − 2])
        prev_ding ← ding[i − 1]
        ring[i] ← max(prev_ding, prev_prev_ring, prev_prev_ding, prev_prev_prev_ring, prev_prev_prev_ding) + A[i − 1]
        prev_ring ← ring[i − 1]
        ding[i] ← max(prev_ring, prev_prev_ring, prev_prev_ding, prev_prev_prev_ring, prev_prev_prev_ding) − A[i − 1]
    end for
    return max(ring[n], ding[n])
end function
```

6 Explanation of Running Time of the Algorithm

The algorithm is based on tabulation. The base cases, i.e., if the array A size is either 1, 2, or 3, then it involves constant time operations followed by a for loop ranging from 4 to $n+1$ to determine the values of $ring[i]$ and $ding[i]$ from 4 to $n+1$, which takes $O(n)$ time.

The polynomial in the worst case, which determines the time complexity, is $f(n) = n + c$. Then the time complexity is $O(n)$.

The space complexity of the algorithm is also $O(2 * n)$, which is simply $O(n)$, since it takes 2 arrays $ring$ and $ding$, each of size $n+1$, to store the values in a bottom-up manner.