**Problem description**

A computer game has a shooter and a moving target. The shooter can hit any of n > 1 hiding spot located along a straight line in which the target can hide. The shooter can never see the target; all he knows is that the target moves to an adjacent hiding spot between every two consecutive shots.

Design a Dynamic Programing algorithm that guarantees hitting the target

**Detailed assumptions**

**Target Movement**

* The target moves between adjacent spots in a straight line.
* The target's movement can happen either before or after the shooter takes a shot. (This can be clarified further if there's a specific rule about the timing)
* There's no limit on the number of times the target can move.

**Shooter**

* The shooter can take one shot at a time, aiming at any of the spots.
* The shot is always successful if aimed at the spot where the target currently is.

**Other Assumptions**

* The number of spots (n) is greater than or equal to 1 (there must be at least one spot).

**Detailed solution including the pseudo-code and the steps of solution**

**Base Case**

If there's only one spot (n == 1), the target is guaranteed to be there, so we return 0 (no shot needed).

**DP Table Initialization**

Create a DP table dp of size n + 1 to store the minimum shots needed to hit the target from each spot (i).

Optionally initialize all elements to 0 for better readability (although unnecessary for functionality).

For edge cases (starting at the first or last spot i == 1 or i == n), set dp[i] to 1, considering the target might have moved to the opposite end in one shot.

Initialize other spots (i) with the worst-case scenario (n), assuming the target is as far as possible.

**Fill DP Table**

Iterate through the DP table from n - 1 (second-last spot) down to 1 (first spot). This avoids using the current spot itself in calculations during the first iteration.

For each spot i, consider both adjacent spots (i - 1 and i + 1). Skip spots outside the valid range (1 to n).

Calculate the adjustedShot by adding 1 (shot from the current spot) to the nextShot (minimum shots needed from the adjacent spot), considering the target might have moved there.

Update dp[i] with the minimum value between its current value and the calculated adjustedShot. This ensures we store the least number of shots needed from each spot.

**Return Minimum Shots**

After filling the DP table, dp[currentSpot] holds the minimum number of shots needed to hit the target from the starting position (currentSpot). Return this value.

**Pseudo code**

function hittingAMovingTarget2(n, currentSpot)

// Base case: Only one spot left, target is guaranteed to be there

if n == 1

return 0 // No shot needed

// Initialize DP table for minimum shots needed from each spot

dp[n + 1] = 0 // Initialize with 0s for better readability (optional)

// Improved initialization (consider target movement and edge cases)

for i = 1 to n

if i == 1 or i == n

dp[i] = 1 // Edge cases (starting at first or last spot) - target might be on the other end in one shot

else

dp[i] = n // Initialize with worst-case scenario (n shots) for other spots

end for

// Fill the DP table (iterative approach)

for i = n - 1 down to 1

// Minimum shots needed from current spot (consider target movement)

for j = i - 1 to i + 1 // Check both adjacent spots

if j < 1 or j > n // Skip spots outside the range

continue

// Minimum shots needed from adjacent spot (considering target movement)

nextShot = dp[j]

// One shot needed from this spot + minimum shots from adjacent spot

adjustedShot = 1 + nextShot

// Update dp[i] if reaching the target from adjacent spot takes fewer shots

dp[i] = min(dp[i], adjustedShot)

end for

end for

// Return the minimum number of shots needed from the current spot

return dp[currentSpot]

end function

**Complexity analysis for the algorithm**

**Initialization**

Initializing the DP table of size n + 1 takes O(n) time.

**Filling the DP Table**

The outer loop iterates from n - 1 down to 1, which takes O(n) iterations. Within each iteration, the inner loop checks both adjacent spots, leading to a maximum of two comparisons per iteration. However, in the worst case, both comparisons might involve calculations (checking if j is outside the range). Assuming constant time for these calculations, the inner loop contributes O(1) time per iteration.

**Combining the outer and inner loops**, the time complexity for filling the DP table becomes O(n) \* O(1) = O(n).

**Overall Time Complexity**

Adding the time complexities of initialization and filling the DP table, we get **O(n) + O(n) = O(2n),** which can be simplified to **O(n).**

**Sample output of the solution**

A black background with white text

Description automatically generated

A black background with white text

Description automatically generated

**A comparison between your algorithm and at least one other technique**

|  |  |  |
| --- | --- | --- |
|  | **Dynamic Programming** | **Greedy** |
| **Time Complexity** | O(n2) | O(n) |
| **Space Complexity** | O(n) | O(1) |
| **Readability** | Complex and Difficult | Easy |

**Conclusion**

**Dynamic programming have a big impact on this problem and is not the most efficient solution to it**