**Combinatorics** 

# **Problem Description**

Medium Math Dynamic Programming

have to take exactly k steps to do this. The goal is to find how many different ways you can do this. A way is different from another if the sequence of steps is different. Since you are moving on an infinite number line, negative positions are possible. The position reached after k steps must be exactly endPos. Since the number of ways could be very large, you need to return the result modulo 10^9 + 7 to keep the number manageable. Intuition

Given the need to move a specific number of steps and end at a specific position, this is a classic problem that can be addressed

First, consider the absolute difference between the startPos and endPos: this is the minimum number of moves required to get

To calculate the number of ways, you can use recursion. With each recursive call, you decrement k because you're making a

You start at a startPos on an infinite number line and want to reach an endPos. You can take steps to the left or right, and you

#### using **Dynamic Programming** or memoization to avoid redundant calculations.

from start to end without any extra steps. From there, any additional steps must eventually be counteracted by an equal number of steps in the opposite direction.

step. If k reaches 0, you check if you've arrived at endPos: if so, that's one valid way; if not, you return 0 as it's not a valid sequence of steps. The solution uses a depth-first search (DFS) approach where each 'node' is a position after a certain number of steps. The DFS

explores all possible sequences of left and right steps until k steps are performed. Memoization (@cache decorator) is key here, as it stores the result of previous computations. This is crucial since there are many overlapping subproblems, especially as k grows larger.

The dfs(i, j) function checks whether you can reach the relative position i (startPos - endPos) in j steps. It evaluates two scenarios at each step: moving right and moving left. The modulus operation is used on the sum of possibilities to keep the numbers within the constraints of the modulo. Since movement on the number line is symmetrical, taking an absolute of the i when moving left is a significant optimization.

Movements X steps to the right and Y steps to the left result in the same position as Y steps to the right and X steps to the left

if x and y are flipped. So you can always consider moves to the right and moves toward zero, enabling the dfs function to

Solution Approach The solution employs depth-first search (DFS) with memoization to systematically explore all possible step sequences that

**Recursion**: The dfs(i, j) function calls itself to explore each possibility. It increments or decrements the current position,

memoize more effectively as it encounters fewer unique (i, j) pairs.

amount to exactly k steps and arrive at endPos. The following patterns and algorithms are used:

should not happen in this approach), return 0 because reaching endPos is impossible.

respectively, for a right or left step, and decreases the remaining steps j by one. Base Cases: The function has two critical base cases: ∘ If j == 0 (no more steps left), check if the current position i is 0 (which means startPos == endPos). If so, return 1, representing one valid way to reach the end position; otherwise return 0, as it's not possible to reach endPos without steps left.

∘ If i > j (the current position is farther from 0 than the number of steps remaining) or j < 0 (somehow the steps went negative, which

## Memoization: The @cache decorator is used in Python to store the results of dfs calls, which prevents redundant

calculations of the same (i, j) state. Since many positions will be visited multiple times with the same number of remaining steps, storing these results significantly speeds up the computation.

(10\*\*\*9 + 7). This is done to avoid integer overflow and is a common practice in problems dealing with large numbers.

Modulo Arithmetic: The modulo operation % mod ensures that intermediate sums do not exceed the problem's specified limit

Symmetry and Absolute Value: When a step is taken to the left, the function calls itself with abs(i - 1) instead of -(i -

1). This is because taking a step to the left from position i is equivalent to taking a step to the right from position -i due to

the line's symmetry. Using the absolute value maximizes the effectiveness of memoization because it reduces the number of unique states.

functools module, which internally creates a mapping of arguments to the dfs function to their results.

endPos in k steps. It respects the symmetry of the problem and allows for an efficient computation of the result.

The code does not explicitly create a data structure for memoization; it relies on the cache mechanism provided by the Python

Finally, the result of the dfs function call for abs(startPos - endPos), k gives us the number of ways to reach from startPos to

**Example Walkthrough** Let's go through a small example to illustrate the solution approach. Suppose you start at startPos = 1 and want to reach endPos = 3. You have k = 3 steps to do it. How many different ways can

First, check the absolute difference between startPos and endPos, which is |3 - 1| = 2. This is the minimum number of

To account for the exact k steps, you can take 1 additional step to the left and then move back to the right, resulting in 3 total

Let's use the dfs(i, j) recursion with memoization to explore the possibilities: The initial call is dfs(abs(1-3), 3) which simplifies to dfs(2, 3).

The dfs function will now calculate this in two ways: consider a step to the left and consider a step to the right.

■ Step to the right: dfs(0+1, 0) simplifies to dfs(1, 0), which returns 0 because i is not equal to 0.

However, from the initial dfs(1, 2), if we make one more DFS call with one step to the left (dfs(abs(1-1), 1)), the result is

dfs(0, 1). This state has already been evaluated when it was encountered via the dfs(1, 2) from the first step to the right.

The use of memoization is crucial in this example because it avoids the re-computation of states that have already been visited,

When considering a step to the right (dfs(2+1, 2)), the state becomes dfs(3, 2).

• Step to the right: dfs(1+1, 1) simplifies to dfs(2, 1), which returns 0 because i > j.

 $\circ$  Here, since i = 3 is greater than j = 2, we return 0 because you can't reach position 0 in just 2 steps.

■ Step to the left: dfs(abs(0-1), 0) simplifies to dfs(1, 0), which also returns 0.

### When considering a step to the left (dfs(abs(2-1), 2)), the state becomes dfs(1, 2).

Solution Implementation

MOD = 10 \*\* 9 + 7

return 0

if steps remaining == 0:

**Python** 

steps (right  $\rightarrow$  left  $\rightarrow$  right).

you do this?

Hence, it will not compute again but fetch the result from the memoization cache, which should be zero since we learned no way exists to end at 0 starting from 1 with 1 step.

# Define the modulo constant to avoid large numbers

return 1 if current pos == 0 else 0

return dfs(abs(start\_pos - end\_pos), num\_steps)

// A memoization table to avoid redundant calculations

public int numberOfWays(int startPos, int endPos, int k) {

// Call the helper method to compute the result

if (posDiff > stepsRemain || stepsRemain < 0) {</pre>

return dfs(Math.abs(startPos - endPos), k, offset);

private int dfs(int posDiff, int stepsRemain, int offset) {

return memo[posDiff][stepsRemain + offset];

memo[posDiff][stepsRemain + offset] = (int)ans;

return memo[posDiff][stepsRemain + offset];

int numberOfWays(int startPos, int endPos, int k) {

const int MOD = 1e9 + 7; // Define the modulo constant.

// Create a memoization table initialized to -1 for DP.

std::vector<std::vector<int>> dp(k + 1, std::vector<int>(k + 1, -1));

dfs(Math.abs(currentPos - 1), stepsRemaining - 1); // Move backward

memoTable[currentPos][stepsRemaining] %= mod; // Apply modulo operation

// Kick off the dfs with the absolute distance to cover and the total steps.

def numberOfWays(self, start pos: int, end pos: int, num\_steps: int) -> int:

@lru cache(maxsize=None) # Cache the results of the function calls

# Use modulo operation to keep numbers in a reasonable range

# The absolute difference gives the minimum number of steps required

if current pos > steps\_remaining or steps\_remaining < 0:</pre>

# If no steps remaining, check if we are at the start

return (dfs(current pos + 1, steps remaining - 1) +

# We can't reach the target if we are too far or steps are negative

# Calculate the number of ways by going one step forward or backward

dfs(abs(current\_pos - 1), steps\_remaining - 1)) % MOD

# Call the dfs function with the absolute difference and the number of steps

from functools import lru\_cache # Python 3 caching decorator for optimization

return memoTable[currentPos][stepsRemaining];

# Define the modulo constant to avoid large numbers

return 1 if current pos == 0 else 0

return dfs(abs(start\_pos - end\_pos), num\_steps)

def dfs(current pos: int, steps remaining: int) -> int:

return dfs(Math.abs(startPos - endPos), k);

// Initialize the memoization table with null values

private static final int MOD = 1000000007;

memo = new Integer[k + 1][2 \* k + 1];

// Define a constant for the mod value as required by the problem

int offset = k; // Offset is used to handle negative indices

// A helper method to find the number of ways using depth-first search

// Calculate the result by considering a step in either direction

// Save the result to the memoization table before returning

def dfs(current pos: int, steps remaining: int) -> int:

@lru cache(maxsize=None) # Cache the results of the function calls

# Use modulo operation to keep numbers in a reasonable range

# The absolute difference gives the minimum number of steps required

if current pos > steps\_remaining or steps\_remaining < 0:</pre>

# If no steps remaining, check if we are at the start

return (dfs(current pos + 1, steps remaining - 1) +

# We can't reach the target if we are too far or steps are negative

# Calculate the number of ways by going one step forward or backward

dfs(abs(current\_pos - 1), steps\_remaining - 1)) % MOD

# Call the dfs function with the absolute difference and the number of steps

// Calculate the difference in positions as the first dimension for memo

■ Step to the left: dfs(abs(1-1), 1) simplifies to dfs(0, 1).

At this point we have dfs(0, 1). Considering further steps:

steps to reach from start to end without any extra steps.

Now, for dfs(1, 2), again we consider steps left and right:

such as dfs(1, 2) being re-computed after dfs(1, 0). By caching the results, we speed up the computation and also maintain the count of possible ways within the required modulo.

To summarize, from  $\frac{1}{2}$  to  $\frac{1}{2}$  to  $\frac{1}{2}$  to  $\frac{1}{2}$  with  $\frac{1}{2}$  steps, there are no valid ways to achieve the end position.

from functools import lru\_cache # Python 3 caching decorator for optimization class Solution: def numberOfWays(self, start pos: int, end pos: int, num\_steps: int) -> int:

solution instance = Solution() print(solution\_instance.numberOfWays(1, 2, 3)) # Example call to the method Java

// If position difference is greater than the remaining steps, or steps are negative, return 0 as it's not possible

long ans = dfs(posDiff + 1, stepsRemain - 1, offset) + <math>dfs(Math.abs(posDiff - 1), stepsRemain - 1, offset);

```
// If there are no remaining steps, check if the current position difference is zero
if (stepsRemain == 0) {
    return posDiff == 0 ? 1 : 0;
// Check the memo table to avoid redundant calculations
if (memo[posDiff][stepsRemain + offset] != null) {
```

ans %= MOD;

C++

public:

#include <vector>

#include <cstring>

class Solution {

#include <functional>

return 0;

class Solution {

private Integer[][] memo;

```
// Define the function 'dfs' for Depth First Search with memoization.
        std::function<int(int, int)> dfs = [&](int distance, int stepsRemaining) -> int {
            if (distance > stepsRemaining || stepsRemaining < 0) {</pre>
                return 0; // Base case: If distance is more than steps or steps are negative, return 0.
            if (stepsRemaining == 0) {
                return distance == 0 ? 1 : 0; // Base case: If no steps left, return 1 if distance is 0, else 0.
            if (dp[distance][stepsRemaining] != -1) {
                return dp[distance][stepsRemaining]; // If value is memoized, return it.
            // Calculate the number of ways by moving to the right or to the left and apply modulo.
            dp[distance][stepsRemaining] = (dfs(distance + 1, stepsRemaining - 1) +
                                            dfs(std::abs(distance - 1), stepsRemaining - 1)) % MOD;
            return dp[distance][stepsRemaining]; // Return the number of ways for current state.
        // Call 'dfs' with the absolute distance from startPos to endPos and k steps available.
        return dfs(std::abs(startPos - endPos), k);
};
TypeScript
// The function calculates the number of ways to reach the end position from the start position
// in exactly k steps.
function numberOfWays(startPos: number, endPos: number, k: number): number {
    const mod = 10 ** 9 + 7; // Define the modulo for large numbers to avoid overflow
    // Initialize a memoization table to store intermediate results with default value -1
    const memoTable = new Array(k + 1).fill(0).map(() => new Array(k + 1).fill(-1));
    // Depth-first search function to calculate the number of paths recursively
    const dfs = (currentPos: number, stepsRemaining: number): number => {
        // Base case: If the current position is greater than the remaining steps
        // or remaining steps are less than 0 then return 0 since it's not possible
        if (currentPos > stepsRemaining || stepsRemaining < 0) {</pre>
            return 0;
        // Base case: If no steps are remaining, check if we are at starting position
        if (stepsRemaining === 0) {
            return currentPos === 0 ? 1 : 0;
        // If result is already calculated for the given state, return the cached result
        if (memoTable[currentPos][stepsRemaining] !== -1) {
            return memoTable[currentPos][stepsRemaining];
        // Recursive case: sum the number of ways from moving forward and backward
        // Note: When moving backward, if currentPos is already 0, it will stay at 0 since
        // we can't move to a negative position on the number line
        memoTable[currentPos][stepsRemaining] =
            dfs(currentPos + 1, stepsRemaining - 1) + // Move forward
```

```
solution instance = Solution()
print(solution_instance.numberOfWays(1, 2, 3)) # Example call to the method
Time and Space Complexity
```

the DFS function generates. There are at most k + 1 levels to explore because each level corresponds to a step, and we do not go beyond k steps. At each

The time complexity of this algorithm largely depends on the parameters startPos, endPos, and k, and how many unique states

The given Python code defines a Solution class with a method numberOfWays that calculates the number of ways to reach the

endPos from the startPos in exactly k steps. The solution uses a depth-first search (DFS) approach with memoization.

# two calls may result in the same next state (e.g., dfs(1, j-1) and dfs(-1, j-1) both lead to dfs(1, j-1)), effectively reducing

**Time Complexity** 

**}**;

class Solution:

MOD = 10 \*\* 9 + 7

return 0

if steps remaining == 0:

the number of unique states. Therefore, for each step, there can be up to 2 \* (j - 1) + 1 unique positions to consider because we could either move left or right from each position or stay at the same position.

Considering memoization, which ensures that each unique state is only computed once, the time complexity can be estimated by

the number of unique (i, j) pairs for which the dfs function is called. Hence, the total number of unique calls to dfs is 0(k^2),

step, the number of positions that we could be at increases, however due to the absolute value used in the second call to dfs,

leading to a time complexity of  $T(n) = O(k^2)$ . **Space Complexity** The space complexity is determined by the size of the cache (to memoize the DFS calls) and the maximum depth of the recursion stack (which occurs when the DFS explores from j = k down to j = 0).

The cache potentially stores all unique states that the DFS visits, which we've established above is  $0(k^2)$ .

recursive chains would quickly find cached results and not recurse to their maximum possible depth. However, for the sake of computation, we consider the worst-case scenario without memoization's benefits, which is 0(k).

The maximum depth of the recursive call stack occurs when the function continually recurses without finding any cached results,

which would mean a maximum depth of k. This is a very rare case, though, because memoization ensures that even deep

Therefore, the space complexity is determined by the larger of the cache's size or the recursion stack's depth. In this case, the cache's size is larger; hence the space complexity is  $S(n) = O(k^2)$ .