<https://leetcode.com/problems/number-of-ways-to-stay-in-the-same-place-after-some-steps/description/>

You have a pointer at index 0 in an array of size arrLen. At each step, you can move 1 position to the left, 1 position to the right in the array, or stay in the same place (The pointer should not be placed outside the array at any time).

Given two integers steps and arrLen, return the number of ways such that your pointer is still at index 0 after **exactly** steps steps. Since the answer may be too large, return it **modulo** 109 + 7.

**Example 1:**

**Input:** steps = 3, arrLen = 2

**Output:** 4

**Explanation:** There are 4 differents ways to stay at index 0 after 3 steps.

Right, Left, Stay

Stay, Right, Left

Right, Stay, Left

Stay, Stay, Stay

**Example 2:**

**Input:** steps = 2, arrLen = 4

**Output:** 2

**Explanation:** There are 2 differents ways to stay at index 0 after 2 steps

Right, Left

Stay, Stay

**Example 3:**

**Input:** steps = 4, arrLen = 2

**Output:** 8

**Constraints:**

1 <= steps <= 500

1 <= arrLen <= 10^6

**Attempt 1: 2025-06-19**

**Solution 1: Native DFS (10 min, TLE 12/33)**

class Solution {

    int MOD = (int)(1e9 + 7);

    public int numWays(int steps, int arrLen) {

        return helper(steps, arrLen, 0);

    }

    private int helper(int steps, int arrLen, int index) {

        if(steps == 0) {

            return index == 0 ? 1 : 0;

        }

        int count = 0;

        // Stay

        count = (count + helper(steps - 1, arrLen, index)) % MOD;

        // Move right

        if(index + 1 < arrLen) {

            count = (count + helper(steps - 1, arrLen, index + 1)) % MOD;

        }

        // Move left

        if(index - 1 >= 0) {

            count = (count + helper(steps - 1, arrLen, index - 1)) % MOD;

        }

        return count;

    }

}

Time Complexity: O(3^steps)

Space Complexity: O(steps)

**Solution 2: Memoization (10 min, MLE 27/33)**

class Solution {

    int MOD = (int)(1e9 + 7);

    public int numWays(int steps, int arrLen) {

        Integer[][] memo = new Integer[steps + 1][arrLen];

        return helper(steps, arrLen, 0, memo);

    }

    private int helper(int steps, int arrLen, int index, Integer[][] memo) {

        if(steps == 0) {

            return index == 0 ? 1 : 0;

        }

        if(memo[steps][index] != null) {

            return memo[steps][index];

        }

        int count = 0;

        // Stay

        count = (count + helper(steps - 1, arrLen, index, memo)) % MOD;

        // Move right

        if(index + 1 < arrLen) {

            count = (count + helper(steps - 1, arrLen, index + 1, memo)) % MOD;

        }

        // Move left

        if(index - 1 >= 0) {

            count = (count + helper(steps - 1, arrLen, index - 1, memo)) % MOD;

        }

        return memo[steps][index] = count;

    }

}

Time Complexity: O(steps \* arrLen)

Space Complexity: O(steps \* arrLen)

**Solution 3: Optimized Memoization (10 min)**

class Solution {

int MOD = (int)(1e9 + 7);

public int numWays(int steps, int arrLen) {

// Maximum Reachable Position: In steps steps, you can't move

// further than steps positions away from the start.

int maxPos = Math.min(steps / 2 + 1, arrLen);

Integer[][] memo = new Integer[steps + 1][maxPos];

return helper(steps, arrLen, 0, memo, maxPos);

}

private int helper(int steps, int arrLen, int index, Integer[][] memo, int maxPos) {

if(steps == 0) {

return index == 0 ? 1 : 0;

}

// Additional check to avoid exceed maximum reachable position

if(index >= maxPos || index < 0) {

return 0;

}

if(memo[steps][index] != null) {

return memo[steps][index];

}

int count = 0;

// Stay

count = (count + helper(steps - 1, arrLen, index, memo, maxPos)) % MOD;

// Move right

count = (count + helper(steps - 1, arrLen, index + 1, memo, maxPos)) % MOD;

// Move left

count = (count + helper(steps - 1, arrLen, index - 1, memo, maxPos)) % MOD;

return memo[steps][index] = count;

}

}

Time Complexity: O(steps \* maxPos)

Space Complexity: O(steps \* maxPos)

**Solution 4: 2D DP (10 min)**

class Solution {

    public int numWays(int steps, int arrLen) {

        int MOD = (int)(1e9 + 7);

        // To avoid Memory Limit Exceed

        // Maximum Reachable Position: In steps steps, you can't move

        // further than steps positions away from the start.

        int maxPos = Math.min(steps / 2 + 1, arrLen);

        int[][] dp = new int[steps + 1][maxPos];

        // Base case: When 0 steps stay at index 0

        dp[0][0] = 1;

        for(int remainStep = 1; remainStep <= steps; remainStep++) {

            for(int index = 0; index < maxPos; index++) {

                // Stay

                dp[remainStep][index] = dp[remainStep - 1][index];

                // Move right (from left position)

                if(index > 0) {

                    dp[remainStep][index] = (dp[remainStep][index] + dp[remainStep - 1][index - 1]) % MOD;

                }

                // Move left (from right position)

                if(index < maxPos - 1) {

                    dp[remainStep][index] = (dp[remainStep][index] + dp[remainStep - 1][index + 1]) % MOD;

                }

            }

        }

        return dp[steps][0];

    }

}

Time Complexity: O(steps \* maxPos)

Space Complexity: O(steps \* maxPos)

**Refer to Deepseek**

**Pure DFS Approach**

Here's a pure DFS solution without memoization. Note that this will be very inefficient for larger inputs (it will likely time out on LeetCode for steps > 20), but it demonstrates the basic recursive approach.

**Recursive Definition**: At each step, we can either:

Stay in the same position

Move left (if possible)

Move right (if possible)

**Base Case**: When we've taken all steps (steps == 0), we check if we're at position 0.

**No Memoization**: We'll explore all possible paths without caching results.

**Solution Code**

class Solution {

private final int MOD = 1000000007;

public int numWays(int steps, int arrLen) {

return dfs(0, steps, arrLen);

}

private int dfs(int pos, int remainingSteps, int arrLen) {

// Base case: if no steps left, check if we're at position 0

if (remainingSteps == 0) {

return pos == 0 ? 1 : 0;

}

// Initialize count for this state

int count = 0;

// Option 1: Stay in current position

count = (count + dfs(pos, remainingSteps - 1, arrLen)) % MOD;

// Option 2: Move left (if possible)

if (pos > 0) {

count = (count + dfs(pos - 1, remainingSteps - 1, arrLen)) % MOD;

}

// Option 3: Move right (if possible)

if (pos < arrLen - 1) {

count = (count + dfs(pos + 1, remainingSteps - 1, arrLen)) % MOD;

}

return count;

}

}

**Explanation**

**Base Case**: When there are no remaining steps (remainingSteps == 0), we return 1 if we're at position 0 (successful path), otherwise 0.

**Recursive Cases**:

**Stay**: We recurse with the same position and one fewer step.

**Move Left**: If possible (position > 0), we recurse with position decreased by 1.

**Move Right**: If possible (position < arrLen - 1), we recurse with position increased by 1.

**Modulo Operation**: We apply modulo 10^9 + 7 at each step to prevent integer overflow (though with pure DFS, we likely won't hit large numbers before timing out).

**Important Notes**

**Performance Warning**: This pure DFS solution has a time complexity of O(3^steps) because at each step we have up to 3 choices. This grows extremely quickly:

For steps = 10 → ~59,000 operations

For steps = 20 → ~3.5 billion operations

For steps = 500 → impossibly large

**Practical Use**: This solution is only useful for understanding the recursive nature of the problem. For actual LeetCode submission, you should use the dynamic programming approach or at least the memoized DFS version.

**Testing**: If you want to test this locally, keep steps small (≤ 15) to avoid long execution times or stack overflows.

**Time Complexity**

The pure DFS solution has an **exponential time complexity**:

**Branching Factor**: At each step, there are up to 3 choices (stay, left, right)

**Recursion Depth**: The recursion goes steps levels deep

**Total Nodes**: This creates a recursion tree with up to 3<sup>steps</sup> nodes

**Formally**: O(3^steps)

Example growth:

steps=10 → ~59,000 operations

steps=20 → ~3.5 billion operations

steps=30 → ~205 trillion operations

**Space Complexity**

The space complexity comes from two factors:

**Recursion Stack**: The maximum depth is steps levels

**No Additional Data Structures**: We're not using any significant extra storage

**Formally**: O(steps)

This is because:

Each recursive call consumes stack space

The maximum stack depth equals the number of steps

No memoization means no extra storage for caching

**Why This Is Problematic**

For LeetCode constraints (steps ≤ 500):

3^500 is an astronomically large number (~10^238)

Even for steps=20, it's already computationally infeasible

The solution will time out for any non-trivial input

**Comparison with DP Solution**

The dynamic programming solution is much more efficient:

Time: O(steps × min(steps, arrLen))

Space: O(min(steps, arrLen)) with optimization

For steps=500, arrLen=500:

DP: ~250,000 operations

Pure DFS: ~10^238 operations (completely impractical)

**Conclusion**

While the pure DFS solution correctly represents the problem's recursive nature, its exponential complexity makes it unusable for practical purposes. The DP solution is necessary for actual implementation.

**DFS with Memoization Approach**

While the dynamic programming approach is more efficient for this problem, I'll provide a native DFS (Depth-First Search) solution with memoization for educational purposes. Note that this approach may not pass all test cases due to time constraints for larger inputs, but it demonstrates the recursive nature of the problem.

**Recursive Definition**: At each step, we can either:

Stay in the same position

Move left (if possible)

Move right (if possible)

**Base Case**: When we've taken all steps (steps == 0), we check if we're at position 0.

**Memoization**: We'll cache results of (current position, remaining steps) to avoid redundant calculations.

**Solution Code**

class Solution {

private final int MOD = 1000000007;

private Map<String, Integer> memo = new HashMap<>();

public int numWays(int steps, int arrLen) {

return dfs(0, steps, arrLen);

}

private int dfs(int pos, int remainingSteps, int arrLen) {

// Base case: if no steps left, check if we're at position 0

if (remainingSteps == 0) {

return pos == 0 ? 1 : 0;

}

// Create a memo key

String key = pos + "," + remainingSteps;

// Check if we've already computed this state

if (memo.containsKey(key)) {

return memo.get(key);

}

// Initialize count for this state

int count = 0;

// Option 1: Stay in current position

count = (count + dfs(pos, remainingSteps - 1, arrLen)) % MOD;

// Option 2: Move left (if possible)

if (pos > 0) {

count = (count + dfs(pos - 1, remainingSteps - 1, arrLen)) % MOD;

}

// Option 3: Move right (if possible)

if (pos < arrLen - 1) {

count = (count + dfs(pos + 1, remainingSteps - 1, arrLen)) % MOD;

}

// Store result in memo before returning

memo.put(key, count);

return count;

}

}

**Explanation**

**Base Case**: When there are no remaining steps (remainingSteps == 0), we return 1 if we're at position 0 (successful path), otherwise 0.

**Memoization Check**: Before computing, we check if we've already calculated the result for the current position and remaining steps combination.

**Recursive Cases**:

**Stay**: We recurse with the same position and one fewer step.

**Move Left**: If possible, we recurse with position decreased by 1.

**Move Right**: If possible, we recurse with position increased by 1.

**Modulo Operation**: We apply modulo 10^9 + 7 at each step to prevent integer overflow.

**Memoization Storage**: We store computed results to avoid redundant calculations in future recursive calls.

**Note on Performance**

This DFS approach with memoization is correct but may not be efficient enough for very large inputs (like steps = 500). The DP solution is preferred for such cases. However, this DFS solution demonstrates the fundamental recursive nature of the problem and can be useful for understanding the problem's structure.

**I have below memoization solution run into memory limit exceed issue when input as below, any update can help ?**

class Solution {

int MOD = (int)(1e9 + 7);

public int numWays(int steps, int arrLen) {

Integer[][] memo = new Integer[steps + 1][arrLen];

return helper(steps, arrLen, 0, memo);

}

private int helper(int steps, int arrLen, int index, Integer[][] memo) {

if(steps == 0) {

return index == 0 ? 1 : 0;

}

if(memo[steps][index] != null) {

return memo[steps][index];

}

int count = 0;

// Stay

count = (count + helper(steps - 1, arrLen, index, memo)) % MOD;

// Move right

if(index + 1 < arrLen) {

count = (count + helper(steps - 1, arrLen, index + 1, memo)) % MOD;

}

// Move left

if(index - 1 >= 0) {

count = (count + helper(steps - 1, arrLen, index - 1, memo)) % MOD;

}

return memo[steps][index] = count;

}

}

**Optimizing the Memoization Solution to Avoid Memory Limits**

The memory limit issue occurs because your current solution creates a memo array of size (steps + 1) × arrLen, which for steps=430 and arrLen=148488 would require about 255MB of memory (430 × 148488 × 4 bytes per integer). Here's how to optimize it:

**Key Observations for Optimization**

**Maximum Reachable Position**: In steps steps, you can't move further than steps positions away from the start.

**Symmetric Movement**: The problem is symmetric around position 0.

**Limited Space Needed**: We only need to track positions within [0, min(steps, arrLen-1)].

**Optimized Memoization Solution**

class Solution {

int MOD = (int)(1e9 + 7);

public int numWays(int steps, int arrLen) {

// Maximum Reachable Position: In steps steps, you can't move

// further than steps positions away from the start.

int maxPos = Math.min(steps / 2 + 1, arrLen);

Integer[][] memo = new Integer[steps + 1][maxPos];

return helper(steps, arrLen, 0, memo, maxPos);

}

private int helper(int steps, int arrLen, int index, Integer[][] memo, int maxPos) {

if(steps == 0) {

return index == 0 ? 1 : 0;

}

// Additional check to avoid exceed maximum reachable position

if(index >= maxPos || index < 0) {

return 0;

}

if(memo[steps][index] != null) {

return memo[steps][index];

}

int count = 0;

// Stay

count = (count + helper(steps - 1, arrLen, index, memo, maxPos)) % MOD;

// Move right

count = (count + helper(steps - 1, arrLen, index + 1, memo, maxPos)) % MOD;

// Move left

count = (count + helper(steps - 1, arrLen, index - 1, memo, maxPos)) % MOD;

return memo[steps][index] = count;

}

}

**Key Improvements**

**Reduced Position Range**: maxPos = Math.min(steps / 2 + 1, arrLen) limits the positions we need to track

**Iterative DP**: Uses O(maxPos) space instead of O(steps × maxPos)

**Early Termination**: Automatically returns 0 for positions beyond reachable range

For your input (steps=430, arrLen=148488):

Original: memo[431][148488] → ~255MB

Optimized: memo[431][216] → ~0.37MB (216 = 430/2 + 1)

DP version: uses just 2 arrays of size 216 → ~1.7KB

This should comfortably run within memory limits while maintaining the same correctness.

**2D DP Solution Mirroring Native DFS Solution**

Here's a 2D dynamic programming solution that exactly mirrors the structure of your pure DFS approach, maintaining the same logical flow while being more efficient:

class Solution {

int MOD = (int)(1e9 + 7);

public int numWays(int steps, int arrLen) {

// Calculate maximum position we can reach

int maxPos = Math.min(steps, arrLen - 1);

// dp[remainingSteps][currentPosition]

int[][] dp = new int[steps + 1][maxPos + 1];

// Base case: when steps = 0, only position 0 has value 1

dp[0][0] = 1;

// Fill the DP table in bottom-up manner

for (int remainingSteps = 1; remainingSteps <= steps; remainingSteps++) {

for (int pos = 0; pos <= maxPos; pos++) {

// Option 1: Stay in current position

dp[remainingSteps][pos] = dp[remainingSteps - 1][pos];

// Option 2: Move right (from left position)

if (pos > 0) {

dp[remainingSteps][pos] = (dp[remainingSteps][pos] + dp[remainingSteps - 1][pos - 1]) % MOD;

}

// Option 3: Move left (from right position)

if (pos < maxPos) {

dp[remainingSteps][pos] = (dp[remainingSteps][pos] + dp[remainingSteps - 1][pos + 1]) % MOD;

}

}

}

return dp[steps][0];

}

}

**Key Features That Mirror DFS:**

**Same State Representation**:

dp[remainingSteps][pos] exactly matches the DFS parameters

Base case (steps == 0) is identical

**Identical Transition Logic**:

Three options (stay, left, right) are handled the same way

Same boundary checks for left/right moves

**Same Order of Operations**:

Stay operation first

Then right move (if possible)

Then left move (if possible)

**Same Modulo Application**:

Applied at each addition just like in DFS

**Space Optimization (Optional):**

If you want to reduce space usage while keeping the 2D structure visible during debugging:

class Solution {

int MOD = (int)(1e9 + 7);

public int numWays(int steps, int arrLen) {

int maxPos = Math.min(steps, arrLen - 1);

// Use two 1D arrays instead of full 2D

int[] prev = new int[maxPos + 1];

int[] curr = new int[maxPos + 1];

prev[0] = 1;

for (int s = 1; s <= steps; s++) {

for (int pos = 0; pos <= maxPos; pos++) {

curr[pos] = prev[pos];

if (pos > 0) curr[pos] = (curr[pos] + prev[pos - 1]) % MOD;

if (pos < maxPos) curr[pos] = (curr[pos] + prev[pos + 1]) % MOD;

}

// Swap arrays for next iteration

int[] temp = prev;

prev = curr;

curr = temp;

// Clear current array for next use

Arrays.fill(curr, 0);

}

return prev[0];

}

}

This maintains the same logical structure while using O(n) space instead of O(n²).

**Refer to**

[L62.Unique Paths (Ref.L63,L64,L174,L2087,L2304,L2400)](note://E33F0B728ABE4CCF8B7E554A219CF1EA)

[L70.Climbing Stairs](note://WEBce2fed61238e31617ae6d19bbb7f7a01)

[L2400.Number of Ways to Reach a Position After Exactly k Steps (Ref.L62,L70)](note://WEB0d086d1f5849b8970f5b9498bd68e08d)