<https://leetcode.com/problems/number-of-ways-to-reach-a-position-after-exactly-k-steps/description/>

You are given two **positive** integers startPos and endPos. Initially, you are standing at position startPos on an **infinite** number line. With one step, you can move either one position to the left, or one position to the right.

Given a positive integer k, return *the number of****different****ways to reach the position*endPos*starting from*startPos*, such that you perform****exactly***k*steps*. Since the answer may be very large, return it **modulo** 109 + 7.

Two ways are considered different if the order of the steps made is not exactly the same.

**Note** that the number line includes negative integers.

**Example 1:**

**Input:** startPos = 1, endPos = 2, k = 3

**Output:** 3

**Explanation:** We can reach position 2 from 1 in exactly 3 steps in three ways:

- 1 -> 2 -> 3 -> 2.

- 1 -> 2 -> 1 -> 2.

- 1 -> 0 -> 1 -> 2.

It can be proven that no other way is possible, so we return 3.

**Example 2:**

**Input:** startPos = 2, endPos = 5, k = 10

**Output:** 0

**Explanation:** It is impossible to reach position 5 from position 2 in exactly 10 steps.

**Constraints:**

1 <= startPos, endPos, k <= 1000

**Attempt 1: 2025-06-08**

**Solution 1: Native DFS (10 min, TLE 2/35)**

class Solution {

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

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

return helper(startPos, endPos, k, 0);

}

private int helper(int curPos, int endPos, int k, int steps) {

if(steps > k) {

return 0;

}

if(curPos == endPos && steps == k) {

return 1;

}

int moveLeft = helper(curPos - 1, endPos, k, steps + 1) % MOD;

int moveRight = helper(curPos + 1, endPos, k, steps + 1) % MOD;

return (moveLeft + moveRight) % MOD;

}

}

Time Complexity: O(2^k)

Space Complexity: O(k)

**Solution 2: Memoization (10 min, TLE 34/35)**

class Solution {

    int MOD = (int)(1e9 + 7);

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

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

        return helper(startPos, endPos, k, 0, memo);

    }

    private int helper(int curPos, int endPos, int k, int steps, Map<String, Integer> memo) {

        if(steps > k) {

            return 0;

        }

        if(curPos == endPos && steps == k) {

            return 1;

        }

        String key = curPos + "\_" + endPos + "\_" + steps;

        if(memo.containsKey(key)) {

            return memo.get(key);

        }

        int moveLeft = helper(curPos - 1, endPos, k, steps + 1, memo) % MOD;

        int moveRight = helper(curPos + 1, endPos, k, steps + 1, memo) % MOD;

        int result = (moveLeft + moveRight) % MOD;

        memo.put(key, result);

        return result;

    }

}

Time Complexity: O(k^2)

Space Complexity: O(k^2)

**Solution 3: Memoization with pruning (10 min, 597 ms pass 35 test cases)**

class Solution {

    int MOD = (int)(1e9 + 7);

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

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

        return helper(startPos, endPos, k, 0, memo);

    }

    private int helper(int curPos, int endPos, int k, int steps, Map<String, Integer> memo) {

        if(steps > k) {

            return 0;

        }

        // Early pruning: Impossible cases

        // The remaining steps (k - steps) are less than the distance to endPos.

        if(Math.abs(curPos - endPos) > k - steps) {

            return 0;

        }

        if(curPos == endPos && steps == k) {

            return 1;

        }

        String key = curPos + "\_" + endPos + "\_" + steps;

        if(memo.containsKey(key)) {

            return memo.get(key);

        }

        int moveLeft = helper(curPos - 1, endPos, k, steps + 1, memo) % MOD;

        int moveRight = helper(curPos + 1, endPos, k, steps + 1, memo) % MOD;

        int result = (moveLeft + moveRight) % MOD;

        memo.put(key, result);

        return result;

    }

}

Time Complexity: O(k^2)

Space Complexity: O(k^2)

**Solution 4: Memoization with more pruning (30 min, 411 ms pass 35 test cases)**

class Solution {

    int MOD = (int)(1e9 + 7);

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

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

        return helper(startPos, endPos, k, 0, memo);

    }

    private int helper(int curPos, int endPos, int k, int steps, Map<String, Integer> memo) {

        if(steps > k) {

            return 0;

        }

        // Early pruning: Impossible cases

        // 1.The remaining steps (k - steps) are less than the distance to endPos.

        // 2.The parity (odd / even) of the remaining steps does not match the distance

        // (since each step changes position by +1 / -1), as extra steps beyond the

        // minimum distance must come in canceling pairs (+1 / -1), which requires an

        // even number of extra steps.

        int distance = Math.abs(curPos - endPos);

        int remainingSteps = k - steps;

        if(distance > remainingSteps || (remainingSteps - distance) % 2 != 0) {

            return 0;

        }

        if(curPos == endPos && steps == k) {

            return 1;

        }

        String key = curPos + "\_" + endPos + "\_" + steps;

        if(memo.containsKey(key)) {

            return memo.get(key);

        }

        int moveLeft = helper(curPos - 1, endPos, k, steps + 1, memo) % MOD;

        int moveRight = helper(curPos + 1, endPos, k, steps + 1, memo) % MOD;

        int result = (moveLeft + moveRight) % MOD;

        memo.put(key, result);

        return result;

    }

}

Time Complexity: O(k^2)

Space Complexity: O(k^2)

**Solution 5: Memoization with more pruning and short key (10 min, 336 ms pass 35 test cases)**

class Solution {

    int MOD = (int)(1e9 + 7);

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

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

        return helper(startPos, endPos, k, memo);

    }

    private int helper(int curPos, int endPos, int remainingSteps, Map<String, Integer> memo) {

        if(remainingSteps < 0) {

            return 0;

        }

        // Early pruning: Impossible cases

        // 1.The remaining steps (k - steps) are less than the distance to endPos.

        // 2.The parity (odd / even) of the remaining steps does not match the distance

        // (since each step changes position by +1 / -1), as extra steps beyond the

        // minimum distance must come in canceling pairs (+1 / -1), which requires an

        // even number of extra steps.

        int distance = Math.abs(curPos - endPos);

        if(distance > remainingSteps || (remainingSteps - distance) % 2 != 0) {

            return 0;

        }

        if(curPos == endPos && remainingSteps == 0) {

            return 1;

        }

        String key = curPos + "\_" + + remainingSteps;

        if(memo.containsKey(key)) {

            return memo.get(key);

        }

        int moveLeft = helper(curPos - 1, endPos, remainingSteps - 1, memo) % MOD;

        int moveRight = helper(curPos + 1, endPos, remainingSteps - 1, memo) % MOD;

        int result = (moveLeft + moveRight) % MOD;

        memo.put(key, result);

        return result;

    }

}

Time Complexity: O(k^2)

Space Complexity: O(k^2)

**Solution 6: DP with more pruning and short key (10 min, 55 ms pass 35 test cases)**

class Solution {

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

        int MOD = (int)(1e9 + 7);

        // Covers positions from -1000 to +2000

        int[][] dp = new int[3001][k + 1];

        // Adjust positions to be non-negative (since we can't have negative array indices)

        startPos += 1000;

        endPos += 1000;

        // Base case: When remainingSteps = 0, pos == endPos is 1

        for(int pos = 0; pos <= 3000; pos++) {

            if(pos == endPos) {

                dp[pos][0] = 1;

            }

        }

        // Fill DP table with bottom up

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

            for(int pos = 0; pos <= 3000; pos++) {

                int distance = Math.abs(pos - endPos);

                // Prune impossible states (exactly matching DFS logic)

                if(distance > remainingSteps || (remainingSteps - distance) % 2 != 0) {

                    //dp[pos][remainingSteps] = 0;

                    continue;

                }

                // Calculate moves (exactly matching DFS recursive calls)

                int moveLeft = 0;

                int moveRight = 0;

                if(pos > 0) {

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

                }

                if(pos < 3000) {

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

                }

                dp[pos][remainingSteps] = (moveLeft + moveRight) % MOD;

            }

        }

        return dp[startPos][k];

    }

}

Time Complexity: O(k^2)

Space Complexity: O(k^2)

**Refer to Deepseek**

**Why the memoized solution still time out ?**

The memoized solution still times out for the input startPos = 1, endPos = 10, k = 3 because of **unnecessary recursive calls** that could be **pruned early** based on the problem's constraints. Here's why:

**Problem with the Current Approach:**

**No Early Pruning**:

The code continues exploring paths even when it's **impossible** to reach endPos with the remaining steps.

For example, if curPos = 5 and steps = 2 (with k = 3 and endPos = 10), the remaining steps (1) cannot cover the distance (|10 - 5| = 5).

Without pruning, the recursion keeps going until steps > k, wasting time.

**String Key Overhead**:

Using curPos + "\_" + endPos + "\_" + steps as a key is inefficient (string operations are slow).

Since endPos is constant, it doesn't need to be part of the key.

**Unnecessary Recursion**:

The recursion explores all possible paths, even those that clearly cannot reach endPos in k steps.

**Why It Times Out for startPos=1, endPos=10, k=3?**

**Distance Check**: The distance between 1 and 10 is 9, but k=3.

**Fact**: To reach 10 from 1 in 3 steps, you’d need (k + distance) / 2 = (3 + 9)/2 = 6 right moves, but k=3 is too small.

**This is impossible**, but the current solution still explores all paths until steps > k.

**Optimized Fix: Add Early Pruning**

We should **stop recursion early** if:

The remaining steps (k - steps) are **less than** the distance to endPos.

The **parity** (odd/even) of the remaining steps does not match the distance (since each step changes position by ±1).

**Optimized Solution (With Early Pruning)**

class Solution {

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

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

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

return helper(startPos, endPos, k, 0, memo);

}

private int helper(int curPos, int endPos, int k, int steps, Map<String, Integer> memo) {

int distance = Math.abs(endPos - curPos);

int remainingSteps = k - steps;

// Early pruning: Impossible cases

if (distance > remainingSteps || (remainingSteps - distance) % 2 != 0) {

return 0;

}

if (steps == k) {

return curPos == endPos ? 1 : 0;

}

String key = curPos + "\_" + steps; // endPos is constant, no need in key

if (memo.containsKey(key)) {

return memo.get(key);

}

int moveLeft = helper(curPos - 1, endPos, k, steps + 1, memo) % MOD;

int moveRight = helper(curPos + 1, endPos, k, steps + 1, memo) % MOD;

int result = (moveLeft + moveRight) % MOD;

memo.put(key, result);

return result;

}

}

**Key Improvements:**

**Early Pruning**:

If distance > remainingSteps, return 0 immediately (can't reach).

If (remainingSteps - distance) is odd, return 0 (parity mismatch).

**Optimized Memoization Key**:

Removed endPos from the key (since it's constant).

Faster key generation (curPos + "\_" + steps).

**Faster Termination**:

The recursion now stops early for impossible cases, reducing unnecessary computations.

**Result**

**Before**: The original solution would explore all 2³ = 8 paths (even though most are impossible).

**After**: The optimized solution detects impossibility early and returns 0 immediately.

This should now **pass all test cases**, including startPos=1, endPos=10, k=3, because it **prunes invalid branches early**. 🚀

**Explain more on "The parity (odd/even) of the remaining steps does not match the distance (since each step changes position by ±1)."**

**Understanding Parity in Movement Problems**

In problems where you can move **±1** per step (like this LeetCode problem), **parity** (whether a number is odd or even) plays a crucial role in determining **reachability**. Here’s why:

**Key Insight:**

Each step changes your position by **+1 (right)** or **-1 (left)**.

This means the **total displacement** after k steps must match the distance between startPos and endPos.

**Parity Rule:**

Let distance = |endPos - startPos|.

Let remainingSteps = k - stepsTaken.

To reach endPos in exactly k steps:

**Distance ≤ Remaining Steps** (otherwise, you can’t cover the distance).

**(remainingSteps - distance) must be even** (explained below).

**Why Must (remainingSteps - distance) Be Even?**

**Every Step Changes Position by ±1**

If you need to cover distance = D, you must take at least D steps in the correct direction.

But if remainingSteps > D, the extra steps must **cancel out** (since you must end exactly at endPos).

**Extra Steps Must Cancel Each Other**

Suppose:

You need to cover distance = 5 (e.g., from 0 to 5).

You have remainingSteps = 7.

**Minimum steps needed = 5** (all right moves: +1 +1 +1 +1 +1).

**Extra steps = 2** (7 - 5 = 2).

These extra steps must **cancel out** (e.g., +1 -1 or -1 +1).

This requires an **even number of extra steps** (since each "cancel pair" takes 2 steps).

**If (remainingSteps - distance) is Odd?**

You’d have an **unmatched step** that can’t be canceled.

Example:

distance = 5, remainingSteps = 6 → 6 - 5 = 1 (odd).

You’d need to take 5 right steps and 1 left step → final position = 5 - 1 = 4 (not 5).

Thus, it’s impossible.

**Applied to the Given Problem (startPos=1, endPos=10, k=3)**

distance = |10 - 1| = 9.

remainingSteps = k = 3.

Check:

Distance ≤ Remaining Steps?9 ≤ 3 → **False** → Impossible.

Parity Check (even if distance ≤ remainingSteps):remainingSteps - distance = 3 - 9 = -6 (even, but already impossible due to distance).

**Thus, the answer is 0 (no valid paths).**

**Summary**

Parity Check Formula:(remainingSteps - distance) % 2 == 0 must hold.

Why?
Extra steps beyond the minimum distance must come in canceling pairs (+1 -1), which requires an **even** number of extra steps.

If parity fails → immediately return 0 (prune the branch).

This optimization **drastically reduces unnecessary recursive calls**, making the solution efficient. 🚀

**I see some solution initialize 2D DP array as dp[3001][k + 1], can you write up based on this initialization and please exactly replicate the logic based on below DFS solution ? such as the iteration on variable direction should match the DFS**

**Bottom-Up DP Solution with Fixed Array Size**

Here's a bottom-up DP solution that exactly replicates the logic of your DFS solution, using a fixed DP array size of [3001][k + 1] as you mentioned. This approach avoids position offset calculations and matches the DFS logic precisely.

class Solution {

private static final int MOD = (int)(1e9 + 7);

private static final int MAX\_POS = 3000; // Covers positions from -1000 to +2000

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

// Adjust positions to be non-negative (since we can't have negative array indices)

startPos += 1000;

endPos += 1000;

// DP table: dp[pos][remainingSteps]

int[][] dp = new int[MAX\_POS + 1][k + 1];

// Initialize base case: when remainingSteps = 0

for (int pos = 0; pos <= MAX\_POS; pos++) {

dp[pos][0] = (pos == endPos) ? 1 : 0;

}

// Fill the DP table in bottom-up manner

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

for (int pos = 0; pos <= MAX\_POS; pos++) {

int distance = Math.abs(pos - endPos);

// Prune impossible states (exactly matching DFS logic)

if (distance > remainingSteps || (remainingSteps - distance) % 2 != 0) {

dp[pos][remainingSteps] = 0;

continue;

}

// Calculate moves (exactly matching DFS recursive calls)

int moveLeft = (pos > 0) ? dp[pos - 1][remainingSteps - 1] : 0;

int moveRight = (pos < MAX\_POS) ? dp[pos + 1][remainingSteps - 1] : 0;

dp[pos][remainingSteps] = (moveLeft + moveRight) % MOD;

}

}

return dp[startPos][k];

}

}

**Key Features Matching Your DFS Solution:**

**Same Pruning Logic**:

Checks distance > remainingSteps first

Then checks parity with (remainingSteps - distance) % 2 != 0

Exactly matches your DFS pruning conditions

**Same Movement Logic**:

Calculates moveLeft as dp[pos-1][remainingSteps-1]

Calculates moveRight as dp[pos+1][remainingSteps-1]

Exactly replicates your recursive calls

**Same Base Case**:

Initializes dp[pos][0] = 1 only when pos == endPos

All other positions with 0 steps remain 0

**Same State Representation**:

Uses position and remaining steps as the two dimensions

Processes steps from 0 to k (bottom-up)

**Optimization Notes:**

**Position Adjustment**:

We add 1000 to make all positions non-negative

This allows using simple array indices without complex offset calculations

**Fixed Array Size**:

Uses [3001][k+1] array size to cover positions from -1000 to +2000

This is safe since problem constraints guarantee positions stay within reasonable bounds

**Boundary Checks**:

Checks pos > 0 before accessing pos-1

Checks pos < MAX\_POS before accessing pos+1

Prevents array index out of bounds

This solution maintains all the logic of your DFS approach while converting it to an efficient bottom-up DP implementation that won't time out on large inputs.

**Refer to**

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

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

[L1269.Number of Ways to Stay in the Same Place After Some Steps (Ref.L62,L70)](note://WEB4baede0c25f4adecb2aba1a04a052560)