<https://leetcode.com/problems/count-number-of-ways-to-place-houses/description/>

There is a street with n \* 2 **plots**, where there are n plots on each side of the street. The plots on each side are numbered from 1 to n. On each plot, a house can be placed.

Return *the number of ways houses can be placed such that no two houses are adjacent to each other on the same side of the street*. Since the answer may be very large, return it **modulo** 10^9 + 7.

Note that if a house is placed on the ith plot on one side of the street, a house can also be placed on the ith plot on the other side of the street.

**Example 1:**

**Input:** n = 1

**Output:** 4

**Explanation:**

Possible arrangements:

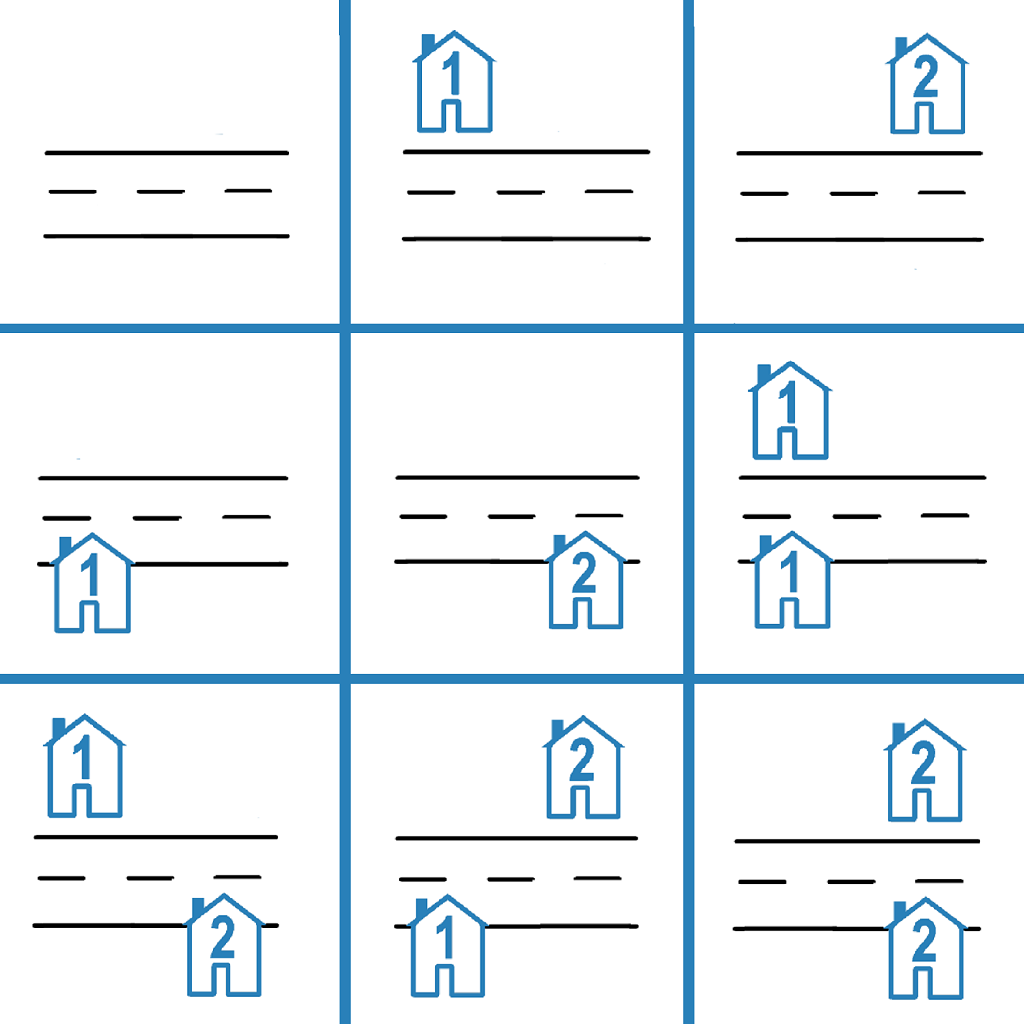
1. All plots are empty.

2. A house is placed on one side of the street.

3. A house is placed on the other side of the street.

4. Two houses are placed, one on each side of the street.

**Example 2:**



**Input:** n = 2

**Output:** 9

**Explanation:** The 9 possible arrangements are shown in the diagram above.

**Constraints:**

1 <= n <= 10^4

**Attempt 1: 2025-06-01**

**Solution 1: Native DFS (60 min, TLE 5/150)**

class Solution {

    int MOD = (int)(1e9 + 7);

    int oneSideCount = 0;

    public int countHousePlacements(int n) {

        // Calculate ways for one side using DFS

        helper(n, 0, false);

        // Total ways = (ways for one side)^2

        return (int) ((oneSideCount \* oneSideCount) % MOD);

    }

    private void helper(int n, int index, boolean prevPlaced) {

        if(index == n) {

            oneSideCount = (oneSideCount + 1) % MOD;

            return;

        }

        // Option 1: Not place house at current plot

        helper(n, index + 1, false);

        // Option 2: Place house at current plot (only if previous plot not placed)

        if(!prevPlaced) {

            helper(n, index + 1, true);

        }

    }

}

Time Complexity: O(2^n) — Explores all possible configurations (slow for large n).

Space Complexity: O(n) — Recursion stack depth.

**Solution 2: Memoization (30 min, 37 ms pass 150 test cases)**

class Solution {

    int MOD = (int)(1e9 + 7);

    Integer[][] memo;

    public int countHousePlacements(int n) {

        // prevPlaced: 0 (false), 1 (true)

        memo = new Integer[n + 1][2];

        // Calculate ways for one side using DFS, start with prevPlaced = false(0)

// and must use long to store the intermediate result even helper method

// return type as integer only, change to long because oneSideCount \* oneSideCount

// will exceed maximum integer if only keep as integer type for oneSideCount

        long oneSideCount = helper(n, 0, 0, memo);

        // Total ways = (ways for one side)^2

// Perform multiplication in long space to avoid overflow integer before MOD

        return (int) ((oneSideCount \* oneSideCount) % MOD);

    }

    // Because we use memo, the return type change from void to int

    private int helper(int n, int index, int prevPlaced, Integer[][] memo) {

        if(index == n) {

            return 1;

        }

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

            return memo[index][prevPlaced];

        }

        // Option 1: Not place house at current plot

        int notPlaced = helper(n, index + 1, 0, memo);

        int placed = 0;

        // Option 2: Place house at current plot (only if previous plot not placed)

        if(prevPlaced == 0) {

            placed = helper(n, index + 1, 1, memo);

        }

        return memo[index][prevPlaced] = ((notPlaced + placed) % MOD);

    }

}

Time Complexity: O(n) (each subproblem is computed once).

Space Complexity: O(n) for memoization table.

**Note: If statement "long oneSideCount = helper(n, 0, 0, memo);" write as "int oneSideCount = helper(n, 0, 0, memo);", the problem will overflow integer when input = 1000.**

The issue with your solution is **integer overflow** during the multiplication step. Here's why it fails for n = 1000 and how to fix it:

**Root Cause**

**Problem**:

When n = 1000, oneSideCount can be very large (up to ~1e209).

**Multiplying oneSideCount \* oneSideCount exceeds Integer.MAX\_VALUE (2^31 - 1), causing overflow.**

The negative output (-576117205) is a classic sign of integer overflow.

**Why It Happens**:

**Even though you use % MOD inside the helper, the final multiplication (oneSideCount \* oneSideCount) % MOD is performed using int arithmetic, which overflows before the modulo is applied.**

**Fix**

Convert oneSideCount to long before multiplication to prevent overflow:

**Key Changes**

**Return Type**:

Store oneSideCount as long to handle large numbers before squaring.

**Modulo Operation**:

Apply % MOD **after** the multiplication to ensure no overflow:

(oneSideCount \* oneSideCount) % MOD

**Helper Method**:

Removed redundant memo parameter (it's already a class field).

**Why This Works**

By using long for intermediate results, we avoid overflow during multiplication.

The modulo is applied only after all calculations are done in a wider data type (long).

**General Rule**

Always use long for intermediate results when:

Dealing with large numbers (especially factorials or exponentials).

Performing multiplications before modulo operations.

This ensures correctness for large inputs like n = 1000. The expected output (500478595) will now be returned correctly. 🚀

**Solution 3: 2D DP (30 min, 24 ms pass 150 test cases)**

class Solution {

    public int countHousePlacements(int n) {

        int MOD = (int)(1e9 + 7);

        // Empty street

        if(n == 0) {

            return 1;

        }

        long[][] dp = new long[n + 1][2];

        dp[1][0] = 1; // Base case: 1 plot, no house

        dp[1][1] = 1; // Base case: 1 plot, placed one house

        for (int i = 2; i <= n; i++) {

            dp[i][0] = (dp[i - 1][0] + dp[i - 1][1]) % MOD; // Skip current plot

            dp[i][1] = dp[i - 1][0]; // Place house (prev must be empty)

        }

        long oneSideCount = (dp[n][0] + dp[n][1]) % MOD;

        return (int) ((oneSideCount \* oneSideCount) % MOD);

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Solution 4: Space Optimized 2D DP (10 min, 4 ms pass 150 test cases)**

class Solution {

    public int countHousePlacements(int n) {

        int MOD = (int)(1e9 + 7);

        // Empty street

        if(n == 0) {

            return 1;

        }

        int prevNotPlaceHouse = 1; // Base case: 1 plot, no house

        int prevPlaceHouse = 1; // Base case: 1 plot, placed one house

        for (int i = 2; i <= n; i++) {

            int curNotPlaceHouse = (prevNotPlaceHouse + prevPlaceHouse) % MOD; // Skip current plot

            int curPlaceHouse = prevNotPlaceHouse; // Place house (prev must be empty)

            prevNotPlaceHouse = curNotPlaceHouse;

            prevPlaceHouse = curPlaceHouse;

        }

        long oneSideCount = (prevNotPlaceHouse + prevPlaceHouse) % MOD;

        return (int) ((oneSideCount \* oneSideCount) % MOD);

    }

}

Time Complexity: O(n)

Space Complexity: O(1)

**Solution 5: 1D DP (60 min, 5 ms pass 150 test cases)**

class Solution {

    public int countHousePlacements(int n) {

        int MOD = (int)(1e9 + 7);

        // dp[i] represents number of ways to arrange houses on i plots on one side

        // and must define as Long since dp[i] \* dp[i] may exceed maximum integer

        // before % MOD

        long[] dp = new long[n + 1];

        dp[0] = 1; // empty street since no plot

        dp[1] = 2; // either place or don't place

        for(int i = 2; i <= n; i++) {

            dp[i] = (dp[i - 1] + dp[i - 2]) % MOD;

        }

        return (int) ((dp[n] \* dp[n]) % MOD);

    }

}

Time Complexity: O(n)

Space Complexity: O(1)

**Solution 6: Space Optimized 1D DP (10 min, 4 ms pass 150 test cases)**

class Solution {

    public int countHousePlacements(int n) {

        int MOD = (int)(1e9 + 7);

        long a = 1; // empty street since no plot

        long b = 2; // either place or don't place

        for(int i = 2; i <= n; i++) {

            long c = (a + b) % MOD;

            a = b;

            b = c;

        }

        return (int) ((b \* b) % MOD);

    }

}

Time Complexity: O(n)

Space Complexity: O(1)

**Refer to Deepseek**

**Problem Description**

There is a street with n plots on each side. We need to count the number of ways to place houses on these plots such that:

No two houses are adjacent on the same side

Houses on opposite sides don't necessarily have any restrictions

Since the answer can be large, return it modulo 10^9 + 7

**Approach**

This problem can be solved using dynamic programming:

**Observation**: **The two sides of the street are independent except for the adjacency constraint on each side.**

**DP State**: For one side of the street, the number of ways to place houses is equivalent to the "House Robber" problem where we can't place houses adjacent to each other.

**Combination**: Since both sides are independent, the total ways are the square of the ways for one side.

**Solution Code**

class Solution {

private static final int MOD = 1\_000\_000\_007;

public int countHousePlacements(int n) {

if (n == 1) return 4;

// dp[i] represents number of ways to arrange houses on i plots on one side

long[] dp = new long[n + 1];

dp[0] = 1; // empty

dp[1] = 2; // either place or don't place

for (int i = 2; i <= n; i++) {

dp[i] = (dp[i - 1] + dp[i - 2]) % MOD;

}

// Total ways = (ways for one side)^2

return (int) ((dp[n] \* dp[n]) % MOD);

}

}

**Explanation**

**Base Cases**:

For 1 plot: 2 options (place or don't place) on each side → total 2×2 = 4 ways

**DP Transition**:

For plot i, we have two choices:

Don't place a house: same as dp[i-1]

Place a house: can only do this if we didn't place at i-1 → dp[i-2]

So dp[i] = dp[i-1] + dp[i-2] (Fibonacci sequence)

**Result Calculation**:

Since both sides are independent, total ways = (ways for one side) × (ways for other side) = dp[n]²

**Complexity Analysis**

**Time Complexity**: O(n) - Single pass to compute DP array

**Space Complexity**: O(n) - Can be optimized to O(1) by storing only last two values

**Alternative Space-Optimized Solution**

class Solution {

private static final int MOD = 1\_000\_000\_007;

public int countHousePlacements(int n) {

if (n == 1) return 4;

long prev = 2, curr = 3; // For n=2

for (int i = 3; i <= n; i++) {

long next = (prev + curr) % MOD;

prev = curr;

curr = next;

}

return (int) ((curr \* curr) % MOD);

}

}

This version uses O(1) space by only keeping track of the last two values in the sequence.

**I need an intuitive pure DFS solution also**

Here's an intuitive **pure DFS solution** for LeetCode 2320 (Count Number of Ways to Place Houses) that explores all valid house placements without using DP. While this approach is less efficient (O(2^n) time), it helps build intuition for the problem.

**Approach**

**Problem Breakdown**:

**Each plot has two choices: place a house (1) or leave it empty (0).**

**No two 1s can be adjacent on the same side.**

**Both sides of the street are independent, so total ways = (ways for one side)².**

**DFS Strategy**:

**For one side of the street, recursively explore all valid placements where no two houses are adjacent.**

Count all valid configurations for one side, then square the result for both sides.

**Solution Code (Pure DFS)**

class Solution {

private static final int MOD = 1\_000\_000\_007;

private int count = 0;

public int countHousePlacements(int n) {

// Calculate ways for one side using DFS

dfs(n, 0, false);

long waysOneSide = count;

// Total ways = (ways for one side)^2

return (int) ((waysOneSide \* waysOneSide) % MOD);

}

private void dfs(int n, int index, boolean prevPlaced) {

if (index == n) {

count = (count + 1) % MOD;

return;

}

// Option 1: Do not place a house at current plot

dfs(n, index + 1, false);

// Option 2: Place a house (only if previous plot was not occupied)

if (!prevPlaced) {

dfs(n, index + 1, true);

}

}

}

**Explanation**

**DFS Function**:

dfs(n, index, prevPlaced) explores all valid house placements starting from index.

**Base Case**: When index == n, increment the count (valid placement found).

**Recursive Cases**:

**Skip Current Plot**: Always allowed (dfs(n, index + 1, false)).

**Place House**: Only allowed if the previous plot was empty (!prevPlaced).

**Result Calculation**:

After counting all valid configurations for one side (waysOneSide), the total ways for both sides is waysOneSide² % MOD.

**Key Points**

**Time Complexity**: O(2^n) — Explores all possible configurations (slow for large n).

**Space Complexity**: O(n) — Recursion stack depth.

**Why Use MOD?** Prevents integer overflow for large n.

**Example Walkthrough (n = 2)**

**One Side Placements**:

[0, 0] (no houses)

[0, 1] (house at plot 2)

[1, 0] (house at plot 1)

Total: 3 valid ways (count = 3).

**Both Sides**:

Total ways = 3² = 9.

**Note**

This DFS solution is **not optimal** for large n (e.g., n > 20 will time out).

Use the **DP solution** (O(n) time) for larger inputs, as shown in the initial answer.

This pure DFS approach helps understand the problem's recursive nature before optimizing with DP! 🚀

**One question, for below statement, is the 3rd parameter set as "true" because of the current plot place a house (at 'index' place a house), when recursion move on for the next plot as 'index + 1', its previous plot as 'index' consider as occupied ?**

// Option 2: Place a house (only if previous plot was not occupied)

if (!prevPlaced) {

dfs(n, index + 1, true);

}

Yes, exactly! Let me break down why we set the 3rd parameter to true in this recursive call and how it enforces the adjacency constraint:

**Understanding the prevPlaced Parameter**

**Purpose**:

prevPlaced tracks whether we placed a house at the **immediately preceding plot** (index - 1).

This ensures no two houses are adjacent on the same side.

**In the Recursive Call**:

if (!prevPlaced) {

dfs(n, index + 1, true); // 'true' indicates we're placing a house at current 'index'

}

When we place a house at the current index:

The next plot (index + 1) cannot have a house.

Thus, we pass true to indicate that the current plot (index) is occupied, which will prevent the next recursive call from placing a house at index + 1.

**Walkthrough for n = 3**

Let's trace how this works for a street with 3 plots per side:

**1. Initial Call**

dfs(3, 0, false)(Start at plot 0, no house placed before it)

**2. Recursive Exploration**

**Option 1**: Don't place at plot 0
→ dfs(3, 1, false)

Can place or skip at plot 1.

**Option 2**: Place at plot 0 (allowed since prevPlaced = false)
→ dfs(3, 1, true)

Next call (plot 1) will **skip** placement because prevPlaced = true.

**3. Valid Configurations**

The DFS explores all valid sequences like:

[0, 0, 0]

[0, 0, 1]

[0, 1, 0]

[1, 0, 0]

[1, 0, 1]*(Note: [0, 1, 1] and [1, 1, 0] are invalid due to adjacency and are pruned by the !prevPlaced check.)*

**Why true is Correct**

**Current Action**: We're placing a house at index.

**Next Plot (index + 1)**:

Must know that index is occupied to enforce adjacency.

Passing true ensures the next call checks if (!prevPlaced) and skips placement if needed.

**Key Insight**

prevPlaced = true acts like a "lock" preventing house placement at the next plot.

This guarantees no two 1s are adjacent in the generated sequences.

**Example: Invalid Placement Blocked**

For n = 2, if we attempt:

Place at plot 0 ([1, ?])

→ Next call: dfs(2, 1, true)

→ if (!prevPlaced) fails → Skips placing at plot 1.

→ Only [1, 0] is allowed, not [1, 1].

This ensures all configurations are valid! 🏠➡️🚫🏠

**But how this DFS solution evolute into the DP solution ?**

Great question! Let's bridge the gap between the **DFS solution** and the **DP solution** step-by-step to see how we optimize the recursive approach into an efficient dynamic programming solution.

**From DFS to DP: Intuition**

The DFS solution has **overlapping subproblems** (it recomputes the same states repeatedly) and **optimal substructure** (the solution can be built from smaller subproblems). These are the two key properties that allow us to optimize it using DP.

**DFS Solution Recap**

void dfs(int n, int index, boolean prevPlaced) {

if (index == n) {

count++;

return;

}

// Option 1: Skip current plot

dfs(n, index + 1, false);

// Option 2: Place house (if previous is empty)

if (!prevPlaced) {

dfs(n, index + 1, true);

}

}

**Problem**: For large n, this leads to an exponential number of calls (O(2^n)).

**Key Insight**: The result of dfs(n, index, prevPlaced) depends only on:

index (current plot)

prevPlaced (whether the previous plot was occupied)

**Step 1: Memoization (Top-Down DP)**

We can cache the results of subproblems to avoid recomputation.

**Memoization Approach**

**State**: (index, prevPlaced)

**Memo Table**: dp[index][prevPlaced] stores the number of ways to arrange houses from index to n-1, given prevPlaced.

class Solution {

private static final int MOD = 1\_000\_000\_007;

private Integer[][] memo;

public int countHousePlacements(int n) {

memo = new Integer[n + 1][2]; // prevPlaced: 0 (false), 1 (true)

long waysOneSide = dfs(n, 0, 0); // Start with prevPlaced = false (0)

return (int) ((waysOneSide \* waysOneSide) % MOD);

}

private int dfs(int n, int index, int prevPlaced) {

if (index == n) return 1;

if (memo[index][prevPlaced] != null) return memo[index][prevPlaced];

int ways = dfs(n, index + 1, 0); // Skip current plot

if (prevPlaced == 0) {

ways = (ways + dfs(n, index + 1, 1)) % MOD; // Place house

}

memo[index][prevPlaced] = ways;

return ways;

}

}

**Time Complexity**: O(n) (each subproblem is computed once).

**Space Complexity**: O(n) for memoization table.

**Step 2: Bottom-Up DP**

We can eliminate recursion by iteratively filling the DP table.

**DP State Definition**

dp[i][0]: Ways to arrange houses up to plot i when i is **not** occupied.

dp[i][1]: Ways to arrange houses up to plot i when i is **occupied**.

**Transition**

If plot i is **not occupied** (dp[i][0]):

Previous plot i-1 could be either occupied or not.

dp[i][0] = dp[i-1][0] + dp[i-1][1]

If plot i is **occupied** (dp[i][1]):

Previous plot i-1 **must not** be occupied.

dp[i][1] = dp[i-1][0]

**Base Case**

n = 0

return 1 (empty street, not able to place house, but still a edge case, count as 1)

n = 1

dp[1][0] = 1 (only one plot, not place a house)

dp[1][1] = 1 (only one plot, place a house)

class Solution {

private static final int MOD = 1\_000\_000\_007;

public int countHousePlacements(int n) {

if(n == 0) {

return 1;

}

long[][] dp = new long[n + 1][2];

dp[1][0] = 1; // Base case: 0 plot, no house

dp[1][1] = 1; // Base case: 0 plot, place one house

for (int i = 2; i <= n; i++) {

dp[i][0] = (dp[i - 1][0] + dp[i - 1][1]) % MOD; // Skip current plot

dp[i][1] = dp[i - 1][0]; // Place house (prev must be empty)

}

long waysOneSide = (dp[n][0] + dp[n][1]) % MOD;

return (int) ((waysOneSide \* waysOneSide) % MOD);

}

}

**Time Complexity**: O(n).

**Space Complexity**: O(n) (can be optimized to O(1)).

**Step 3: Space Optimization (O(1) Space)**

Since dp[i] depends only on dp[i-1], we can use two variables instead of a full table.

**Optimized DP**

class Solution {

private static final int MOD = 1\_000\_000\_007;

public int countHousePlacements(int n) {

long prevEmpty = 1; // dp[i-1][0]

long prevOccupied = 1; // dp[i-1][1]

for (int i = 1; i <= n; i++) {

long currEmpty = (prevEmpty + prevOccupied) % MOD;

long currOccupied = prevEmpty; // Prev must be empty

// Update for next iteration

prevEmpty = currEmpty;

prevOccupied = currOccupied;

}

long waysOneSide = (prevEmpty + prevOccupied) % MOD;

return (int) ((waysOneSide \* waysOneSide) % MOD);

}

}

**Space Complexity**: O(1).

**Key Observations**

**DFS → Memoization**:

Cache results of (index, prevPlaced) to avoid recomputation.

**Memoization → Bottom-Up DP**:

Convert recursive calls into iterative table filling.

dp[i][0] and dp[i][1] directly correspond to DFS states.

**DP Optimization**:

Reduce space usage by tracking only the previous state.

**Why DP Works**

The problem has:

**Overlapping subproblems**: The same (index, prevPlaced) is recomputed many times in DFS.

**Optimal substructure**: The solution for n plots builds on solutions for n-1 plots.

DP efficiently captures these properties by storing and reusing intermediate results.

**Final Answer**

The DP solution evolves naturally from the DFS approach by:

Identifying overlapping subproblems.

Storing their results in a table (memoization).

Iteratively filling the table (bottom-up DP).

Optimizing space by observing state dependencies.

This progression transforms an exponential-time DFS into an efficient O(n) or O(1)-space DP solution! 🚀

**Though above explain is good, but I didn't see the explain about how the DFS evolute into the Fibonacci like DP solution ?**

You're absolutely right to ask for this clarification! Let me explicitly connect the DFS approach to the **simplified DP solution** (the one using dp[i] = dp[i-1] + dp[i-2]) that you've referenced. This is a beautiful optimization that emerges from the problem's structure.

**From DFS to the Simplified DP Solution**

**1. DFS State Analysis**

In the DFS solution, we tracked:

index: Current plot position

prevPlaced: Whether the previous plot had a house

This gave us two choices at each step:

**Don't place a house at current plot**:

Moves to index + 1 with prevPlaced = false

**Place a house at current plot (if allowed)**:

Moves to index + 1 with prevPlaced = true

**2. Key Realization: Fibonacci Pattern**

The constraints lead to a **Fibonacci-like recurrence**:

If we **don't place** a house at plot i, the number of ways is equal to the total ways for i-1 plots (since we're free to choose for the remaining plots).

If we **place** a house at plot i, we **must skip** plot i-1, so the number of ways is equal to the total ways for i-2 plots.

Thus, the recurrence relation is:

dp[i] = dp[i-1] + dp[i-2]

**3. Why This Matches the DP Solution**

The simplified DP solution directly implements this recurrence:

dp[i] = (dp[i - 1] + dp[i - 2]) % MOD;

dp[i-1]: Counts all valid configurations where plot i is **empty**.

dp[i-2]: Counts all valid configurations where plot i is **occupied** (forcing plot i-1 to be empty).

**4. Base Cases**

dp[0] = 1: Empty street (1 way: no houses).

dp[1] = 2: Either place or don't place a house on a single plot.

**5. Full Correspondence with DFS**

|  |  |  |
| --- | --- | --- |
| DFS Branch | DP Term | Explanation |
| dfs(n, index + 1, false) | dp[i-1] | Skip current plot → ways = all valid ways for  i-1  plots. |
| dfs(n, index + 1, true) | dp[i-2] | Place at current plot → must skip  i-1  → ways = all valid ways for  i-2  . |

**Example: n = 3**

**DFS Paths**:

Skip all: [0, 0, 0]

Skip first, place last: [0, 0, 1]

Place first, skip middle: [1, 0, 0]

Skip first, place middle, skip last: [0, 1, 0]

Place first, skip middle, place last: [1, 0, 1]*(Total: 5 ways, matching dp[3] = dp[2] + dp[1] = 3 + 2 = 5)*

**DP Calculation**:

dp[2] = 3 (from [0,0], [0,1], [1,0])

dp[1] = 2 (from [0], [1])

dp[3] = dp[2] + dp[1] = 5

**Why This Optimization Works**

The problem reduces to counting **binary strings without consecutive 1's**, which is exactly the Fibonacci sequence.

The DP solution captures this by combining:

**Non-placement** (dp[i-1]): Adds a 0 to all valid strings of length i-1.

**Placement** (dp[i-2]): Adds a 01 to all valid strings of length i-2.

**Final Answer**

The simplified DP solution (dp[i] = dp[i-1] + dp[i-2]) emerges by recognizing that:

The DFS's branching directly mirrors the Fibonacci recurrence.

The prevPlaced state in DFS is implicitly handled by the dp[i-2] term (ensuring no two consecutive houses).

The base cases (dp[0] = 1, dp[1] = 2) align with the DFS's terminal conditions.

This is why the elegant O(n) DP solution works! 🎉

**Refer to**

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

[L198.House Robber](note://WEB3a5c4401c63ae93266b0b603cb20076c)