# 2320. Count Number of Ways to Place Houses

Medium Dynamic Programming Leetcode Link

## Problem Description

Imagine a street that has `n` plots on each side, making a total of `n * 2` plots. The task is to determine the number of different ways to build houses on these plots with the constraint that no two houses can be adjacent to each other on the same side of the street. Interestingly, if a house is built on one side of a plot, it's still allowed to build another directly across from it on the other side of the street. The final answer to the number of ways to arrange houses should be provided modulo `10^9 + 7` to keep the numbers manageable, due to potentially large results for big values of `n`.

## Intuition

The solution uses dynamic programming to solve the problem efficiently. To understand the intuition behind this approach, let's consider the subproblem of deciding whether to place a house on a plot. When you're considering a new plot, you have two choices: either to place a house on it or not. If you choose to place a house, the next plot cannot have a house because of the no-adjacent-houses rule. Conversely, if you choose not to place a house, you are free to place a house on the next plot.

We can define two states expressed as arrays `f` and `g`, where `f[i]` represents the number of ways to place houses up to plot `i` with a house on the `i`th plot, and `g[i]` represents the number of ways to place houses up to plot `i` without a house on the `i`th plot. These states are dependent on the previous states:

- `f[i]` relies on `g[i-1]` since a house on plot `i` means the `i-1`th plot cannot have a house.
- `g[i]` combines `f[i-1]` and `g[i-1]` because plot `i` being empty allows for either a house or no house on the `i-1`th plot.

The solution iterates through each plot, updating these states according to the rules established, and takes the modulo `10**9 + 7` to ensure the answer remains in the required bounds. With the values for the final plot calculated, the total number of house placements on one side of the street is the sum of `f[-1]` and `g[-1]`.

Since the problem allows for independent house placement on either side of the street, the final answer is the square of this sum, again modulo `10**9 + 7`, to account for all combinations on both sides of the street.

## Solution Approach

The solution implements a form of dynamic programming, which is an optimization over plain recursion where subproblems are solved once and stored for future use, thus avoiding the recomputation of the same subproblems. The dynamic programming approach used here is tabulation, where we iteratively calculate and store the solution to each subproblem in a bottom-up fashion. In this case, the subproblems are the number of ways to place houses up to a certain plot without violating the "no adjacent houses" rule.

The code uses two lists, `f` and `g`, as mentioned before:

- `f` is initialized with `n` elements, all set to `1`, representing that there is initially one way to arrange a house on the `i`th plot. `f[0]` is always `1` because placing a house on the first plot obviously has no restrictions from previous plots.
- `g` is also initialized with `n` elements, all set to `1`, because at the beginning there is also one way to have a plot without a house on it.

The core logic of the problem is encapsulated in the for-loop:

- `f[i]` is updated to `g[i - 1]`. This captures the idea that if there is a house on the `i`th plot, then there could not have been one on the `i-1`th plot, and the number of ways to reach this state is exactly the number of ways to reach plot `i-1` without a house.
- `g[i]` is updated to `(f[i - 1] + g[i - 1]) % mod`. Here, if the `i`th plot does not have a house, we are free to choose whether we had a house on the `i-1`th plot or not, and the total number of ways to arrange houses up to the `i`th plot is the sum of both possibilities.

After the loop ends, you have the total count of placements for a side street stored in `f[n - 1]` and `g[n - 1]`. Since we can combine any arrangement on one side of the street with any arrangement on the other side, the total number of arrangements is the square of the sum of `f[n - 1]` and `g[n - 1]`. This value is calculated as `v * v % mod`, where `v` is `f[-1] + g[-1]`, and `mod` is `10**9 + 7`.

This approach efficiently calculates the solution in `O(n)` time, as it only involves a single pass through the plots, and in `O(n)` space, due to the storage required for the `f` and `g` arrays.

### Example Walkthrough

Let's illustrate the solution approach with a street that has `n = 3` plots on each side. Here's how we would use dynamic programming to find the count of all possible ways to build houses according to the given rules:

1. We start by initializing the arrays `f` and `g` with 3 elements each, set to `1`, since there's initially one way to have a house or not have a house on every plot. Hence, `f = [1, 1, 1]` and `g = [1, 1, 1]`.
2. Next, we iterate through the plots one by one and use the given relations to update `f[i]` and `g[i]`.
3. When `i = 1` (second plot):
   - We update `f[1]` to `g[0]`, which is `1`, because if we put a house on the second plot, we can't have had a house on the first plot.
   - We update `g[1]` to `(f[0] + g[0]) % mod`, which is `(1 + 1) % mod`, resulting in `2`, because if we don't put a house on the second plot, we could have either had a house or not had a house on the first plot.
4. When `i = 2` (third plot):
   - `f[2]` becomes `g[1]`, which is `2`, signifying that if we build on the third plot, we must not have built on the second.
   - `g[2]` becomes `(f[1] + g[1]) % mod`, which is `(1 + 2) % mod`, resulting in `3`. This accounts for the scenarios where the second plot was either empty or had a house, affecting the possibilities for the third plot.
5. Now `f = [1, 1, 2]` and `g = [1, 2, 3]`.
6. For the final count on one side of the street, we add the last elements of `f` and `g`, resulting in `2 + 3 = 5`, meaning there are 5 unique ways to place houses along one side of a 3-plot street.
7. For both sides of the street, we take this result and square it, giving us the final answer: `5 * 5 = 25`. But we must remember to take modulo `10^9 + 7`.

The calculations above simply illustrate the process. For different values of `n`, we would run the exact same process and compute the final count accordingly. Every step carefully abides by the rule of not allowing adjacent houses on the same side, while assuming both sides are independent, thus squaring the final sum before taking the modulo. This example demonstrates the solution approach for a small example where `n` is `3`. The same principles apply for any value of `n`.

## Python Solution

```python
class Solution:
    def countHousePlacements(self, n: int) -> int:
        # Define the modulo constant to handle large numbers.
        MOD = 10**9 + 7

        # f represents the count of ways to place houses such that
        # the last plot is empty.
        f = [1] * n

        # g represents the count of ways to place houses such that
        # the last plot is occupied.
        g = [1] * n

        # Iterate over the plots starting from the second one.
        for i in range(1, n):
            # If the last plot is empty, then it must come after an occupied plot.
            f[i] = g[i - 1]

            # If the last plot is occupied, it can come after either
            # an empty plot or another occupied plot.
            g[i] = (f[i - 1] + g[i - 1]) % MOD

        # v is the total number of ways to place houses on the last plot.
        v = f[-1] + g[-1]

        # Since we can independently choose how to place houses on each side of the street,
        # we square the number of ways to find the total combinations.
        # We return the result modulo MOD to handle large numbers.
        return (v * v) % MOD

```

## Java Solution

```java
class Solution {
    public int countHousePlacements(int n) {
        final int MODULO = 1_000_000_007; // Define a constant for the modulo value
        int[] place = new int[n]; // place[i] represents the number of ways to place houses up to position i without a house at posit
        int[] noPlace = new int[n]; // noPlace[i] represents the number of ways to place houses up to position i with a house at posi

        // Base cases
        place[0] = 1; // Only one way to not place a house at the first position
        noPlace[0] = 1; // Only one way to place a house at the first position

        // Fill in the arrays using dynamic programming approach
        for (int i = 1; i < n; ++i) {
            place[i] = noPlace[i - 1]; // We can only place a house if the previous position has no house
            noPlace[i] = (place[i - 1] + noPlace[i - 1]) % MODULO; // We can place a house if the previous position is either empty c
        }

        // Calculate the total number of ways for all positions
        // After calculating for n-1 positions, we have to consider options for placement at the nth place hence (place[n - 1] + noPl
        // Since we are considering placements on two sides of the street, we need to square the value for one side to get the total
        long totalWays = (place[n - 1] + noPlace[n - 1]) % MODULO;

        // Return the total number of ways with modulo, ensuring we don't exceed integer limits
        return (int) (totalWays * totalWays % MODULO);
    }
}

```

## C++ Solution

```cpp
class Solution {
public:
    int countHousePlacements(int n) {
        // Initialize a constant MOD to use for modulo operation to avoid overflow
        const int MOD = 1e9 + 7;

        // Create two arrays to hold the number of ways to place houses
        // f represents number of ways when the last plot is empty
        // g represents number of ways when the last plot has a house
        // Initialize the first element as 1 since there's 1 way to place 0 houses
        int f[n + 1], g[n + 1];
        f[0] = g[0] = 1;

        // Calculate the number of ways to place houses dynamically
        for (int i = 1; i < n; ++i) {
            f[i] = g[i - 1]; // If the current plot is empty, previous plot can either be empty or have a house
            g[i] = (f[i - 1] + g[i - 1]) % MOD; // If the current plot has a house, previous plot must be empty
        }

        // Calculate the final number of ways, considering both plots can be either empty or have a house
        // Long is used here to avoid integer overflow due to squaring the value
        long result = (f[n - 1] + g[n - 1]) % MOD;
        result = (result * result) % MOD; // We square it because we calculate for both sides of the street

        // Return the final result modulus MOD
        return static_cast<int>(result);
    }
};

```

## Typescript Solution

```typescript
// Function to count the number of ways to place houses in plots
function countHousePlacements(n: number): number {
    // Initialize two arrays to store intermediate values
    const emptyPlotCounts = new Array(n);
    const housePlotCounts = new Array(n);

    // Base case for the DP, 1 way to place on 0th index for both situations
    // 'n' is a BigInt literal (ES2020+ feature)
    emptyPlotCounts[0] = housePlotCounts[0] = 1n;

    // Define the modulo constant for the final result to prevent overflow
    const mod = BigInt(10 ** 9 + 7);

    // Populate both arrays using dynamic programming approach
    for (let i = 1; i < n; ++i) {
        // Count for the current plot being empty depends on the previous plot having a house
        emptyPlotCounts[i] = housePlotCounts[i - 1];
        // Count for the current plot having a house depends on the previous plot being either empty or having a house
        housePlotCounts[i] = (emptyPlotCounts[i - 1] + housePlotCounts[i - 1]) % mod;
    }

    // Calculate final value from both arrays, representing both placing or not placing a house on the last plot
    const totalCount = emptyPlotCounts[n - 1] + housePlotCounts[n - 1];

    // Return the total number of ways squared modulo mod
    // The square comes from calculating the same for both sides of the street
    // We must cast BigInt result back to a number before returning
    return Number(totalCount ** 2n % mod);
}

```

## Time and Space Complexity

The given code snippet is designed to count the number of ways to place houses on a street with `n` plots, where houses cannot be placed on adjacent plots. This is solved using dynamic programming with two arrays `f` and `g` representing the number of ways to arrange houses with certain conditions.

**Time Complexity:**

To analyze the time complexity, we look at the operations inside the main loop:

- The loop runs from `1` to `n-1`, which indicates `O(n)` iterations.
- Inside the loop, the operations are constant-time; `f[i] = g[i - 1]` and `g[i] = (f[i - 1] + g[i - 1]) % mod` each take `O(1)`.
- Calculation of `v` and final return statement also takes `O(1)` time.

Therefore, the overall time complexity of the code is `O(n)`.

**Space Complexity:**

For space complexity:

- We are using two arrays `f` and `g`, each of size `n`, resulting in `O(n)` space.
- The variables `mod` and `v` use constant space, `O(1)`.

Hence, the space complexity of the code is `O(n)` due to the two arrays.