2189. Number of Ways to Build House of Cards

Medium Math Dynamic Programming Leetcode Link

Problem Description

The challenge presents a scenario where you have `n` number of playing cards, and you are asked to build as many distinct "houses of cards" as possible. A "house of cards" here refers to a structure that consists of multiple rows built according to specific rules:

- Each house is made of rows of connected triangles and horizontal cards.
- A triangle is formed by leaning two cards against each other.
- You must place a horizontal card between every pair of adjacent triangles in a row.
- Triangles in rows above the first row must sit on a horizontal card from the row below.
- Triangles must occupy the leftmost possible position in every row.

The aim is to determine the number of ways you can use all `n` cards to create different houses of cards where "different" means that there is at least one row where the two houses have a differing number of cards.

Intuition

Constructing such a house starts with the foundational row, which sets the precedent for the rest of the structure. Starting with the base case where no triangles can be formed with the remaining cards or exactly enough cards present to form one more row of triangles, we work our way up.

A key to solving this problem is to realize that there's a recursive structure in the way we build the house of cards. Every row of triangles relies on the previous construction, and we can consider the problem at each step as a smaller version of the original problem. We can approach this using dynamic programming or recursion to break down the larger problem into these smaller, manageable components.

Moreover, every additional row of triangles requires precisely three more cards than the previous row (two for the triangles and one horizontal card) plus an additional card for the first triangle of the new row, leading to a total of `3 * k + 2` cards for the `k`th row since it has `k+1` triangles. If at any point the required number of cards exceeds the remaining cards (`n`), we cannot build any more houses with the current structure. However, if it matches, it means one distinct house can be built. If we have excess cards, we have the option to either start a new row or build upon the higher row.

The solution utilizes memoization (`@cache`), a technique in dynamic programming to remember the results of function calls with certain parameters (`n`, `k`) and avoid redundant calculations. This significantly speeds up the execution because many sub-problems are repeated in the recursive structure of the problem.

The `dfs (depth-first search)` function explores all possible ways to form a house of cards with the remaining `n` cards starting with `k` triangles in the current row. This recursive function is the core of the solution and checks if adding another row is possible or whether we should consider adding to a higher row. The first call starts with zero triangles, representing the inception of building from scratch. The `dfs` function returns the number of distinct houses that can be built with the given `n` cards by either extending the current row or moving to the next.

Solution Approach

The code provided implements a depth-first search (DFS) algorithm augmented with memoization, which is often used in dynamic programming to efficiently explore the entire solution space by avoiding unnecessary recalculations.

The core of the implementation is the `dfs` function. It takes two parameters: `n`, which is the number of cards still available, and `k`, which indicates the number of triangles in the current row that we're attempting to build. The goal of the `dfs` function is to return the number of distinct house configurations that can be created from the given state.

Let's dissect the `dfs` function:

1. We calculate the number of cards needed to add another row of triangles by using `x = 3 * k + 2`, where `k` is the current row's triangle count. This formula comes from the fact that each additional triangle in a row requires 3 more cards than the previous triangle (2 cards for the triangle itself and 1 card for the horizontal separator).
2. We have two base conditions:
   - If `x` (the cards required for the next row) is greater than `n` (the cards remaining), we cannot build any more rows, so we return 0.
   - If `x` is exactly equal to `n`, we can build one more row perfectly, and hence return 1, as we use up all the cards in a final complete row.
3. The recursive case involves two calls to `dfs`:
   - `dfs(n - x, k + 1)`: This tries to place a new row on top of the current row, hence we subtract the cards needed from the remaining `n` and increase the triangle count `k` by 1.
   - `dfs(n, k + 1)`: This attempts to skip placing a new row and instead sees if we could place an additional triangle in the next row up.
4. Finally, we sum the results of the two recursive calls, which gives the total number of distinct configurations possible from the current state.

Additionally, the `@cache` decorator from the `functools` package is used to memoize the results of `dfs`. This means that the first time `dfs` is called with a particular `n` and `k`, the result is stored. Any subsequent calls to `dfs` with the same `n` and `k` will return the stored result instead of running the function again. This optimization is crucial because without it, the function would recompute the same intermediate results many times, leading to an exponential time complexity.

The search begins by calling `dfs(n, 0)`, which represents the initial state where we have all `n` cards available and haven't used any to form a row yet.

The algorithm used here is a classic example of bottom-up dynamic programming. By solving small subproblems (like building a smaller house of cards) and combining those solutions to solve larger ones, the complexity of the problem is drastically reduced.

When the `dfs` function is called from the instance of the `Solution` class in `houseOfCards`, it starts the process of exploring all possible configurations and returns the total count. This count represents the number of distinct houses of cards that can be constructed using exactly `n` cards.

Example Walkthrough

Let's consider a small example where `n = 7` cards are available to build houses of cards. We will walk through the solution approach with this number.

1. Initially, we have all 7 cards (`n = 7`) and no triangles formed yet (`k = 0`). We start our depth-first search by invoking `dfs(7, 0)`.
2. The first step in our `dfs` function is to identify if we can build a new row. With `k = 0`, we calculate `x = 3 * k + 2 = 2`. We now have two cases to consider:
   - If we use 2 cards to form the first row (`k = 1`), with one triangle. This leaves us with `n - x = 7 - 2 = 5` cards. We then proceed to make a recursive call `dfs(5, 1)`.
   - If we decide not to form a row with the current `k`, we could see if we could start with a higher row. Since `k = 0` is the first row and we can't go higher at this point, we would only consider the first option here.

Now, let's delve into the recursive call `dfs(5, 1)`:

3. With `n = 5` cards left and `k = 1` triangle in the current row, again we calculate the cards needed for an additional row: `x = 3 * k + 2 = 5`. We are again confronted with two new possibilities:
   - We can use all 5 remaining cards to add another row (`k = 2`), resulting in no cards left (5 - 5 = 0). This leads to a base case where `x` is equal to `n`, so we can return 1 because we've used all cards (`dfs(0, 2)`).
   - Alternatively, we consider the possibility of a higher row without adding a new row at the current level, which is `dfs(5, 2)`.

Since our example doesn't have enough cards to explore more options, let's review the cases we have:

- With 7 cards, we can create one distinct house configuration:
  - 2 cards for the first row (1 triangle), and all 5 remaining cards to build the second row (2 triangles).

Thus, by applying the steps from the solution approach, `dfs(7, 0)` would return `1`, meaning there's only one distinct way to build a house of cards with 7 cards.

Implementing the memoization technique with the `@cache` decorator isn't crucial for this particular example, as our `n` is small and doesn't involve many recursive calls. However, as `n` grows, the number of recursive calls increases exponentially, and memoization becomes essential to store intermediate results and prevent redundant calculations, thus optimizing the solution for larger inputs.

Python Solution

```python
from functools import lru_cache

class Solution:
    def house_of_cards(self, n: int) -> int:
        # Define a decorated helper function with memoization
        # to avoid redundant computations.
        @lru_cache(maxsize=None)
        def dfs(remaining_cards: int, level: int) -> int:
            # Calculate the number of cards required to build
            # the next level of the house.
            required_cards = 3 * level + 2

            # Check if the required cards exceed the remaining cards.
            if required_cards > remaining_cards:
                return 0  # No more levels can be built, so return 0.

            # Check if the remaining cards exactly match the required cards.
            if required_cards == remaining_cards:
                return 1  # Found a solution where all cards are used, return 1.

            # Calculate two scenarios:
            # 1. Building the next level and checking the remaining cards
            # 2. Skipping the current level and trying the next one
            return (dfs(remaining_cards - required_cards, level + 1) +
                    dfs(remaining_cards, level + 1))

        # Start the recursive process with all cards available and starting
        # from the 0th level.
        return dfs(n, 0)

# Example usage:
# solution = Solution()
# number_of_ways = solution.house_of_cards(20)
# print(number_of_ways)

```

Java Solution

```java
class Solution {
    private Integer[][] memoization; // This will store previously computed results for dynamic programming.

    public int houseOfCards(int n) {
        memoization = new Integer[n + 1][n / 3 + 1]; // Initialize the memoization array.
        return countConfigurations(n, 0);
    }

    // Helper method to count the number of configurations recursively using dynamic programming.
    private int countConfigurations(int remainingCards, int currentLevel) {
        // Calculate the number of cards required to form the current level pyramid.
        int cardsNeededForLevel = 3 * currentLevel + 2;

        // Check if the number of cards required exceeds the number of remaining cards.
        if (cardsNeededForLevel > remainingCards) {
            return 0;
        }

        // If the number of required cards is exactly equal to the remaining cards, a valid configuration is found.
        if (cardsNeededForLevel == remainingCards) {
            return 1;
        }

        // Check the memoization array to see if this sub-problem has already been calculated.
        if (memoization[remainingCards][currentLevel] != null) {
            return memoization[remainingCards][currentLevel];
        }

        // Calculate the result in two scenarios:
        // 1. Including the current level in the configuration:
        //    Subtract the cards used for current level and recurse for next level.
        // 2. Excluding the current level; simply consider the next level with the same number of cards.
        // The total count is the sum of counts from both scenarios.
        int includingCurrentLevel = countConfigurations(remainingCards - cardsNeededForLevel, currentLevel + 1);
        int excludingCurrentLevel = countConfigurations(remainingCards, currentLevel + 1);

        // Store the computed result in the memoization array and return it.
        memoization[remainingCards][currentLevel] = includingCurrentLevel + excludingCurrentLevel;
        return memoization[remainingCards][currentLevel];
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <functional>
#include <cstring>

class Solution {
public:
    int houseOfCards(int cardsNumber) {
        // Creating a memoization table filled initially with -1 to denote uncomputed states
        std::vector<std::vector<int>> memo(cardsNumber + 1, std::vector<int>(cardsNumber / 3 + 1, -1));

        // Define a recursive depth-first search function using a lambda expression
        std::function<int(int, int)> depthFirstSearch = [&](int remainingCards, int currentPyramidHeight) -> int {
            // Calculate the number of cards needed to build the current layer of the pyramid
            int cardsNeededForCurrentLayer = 3 * currentPyramidHeight + 2;

            // If we don't have enough cards to build this layer, return 0 indicating no additional pyramids can be built
            if (cardsNeededForCurrentLayer > remainingCards) {
                return 0;
            }

            // If the exact number of remaining cards equals the number needed for the current layer, return 1
            // because we can build exactly one more pyramid
            if (cardsNeededForCurrentLayer == remainingCards) {
                return 1;
            }

            // If we have already computed the result for this state, return the cached result
            if (memo[remainingCards][currentPyramidHeight] != -1) {
                return memo[remainingCards][currentPyramidHeight];
            }

            // Recursive case: count the total pyramids that can be built by
            // either including or excluding the current layer of the pyramid
            return memo[remainingCards][currentPyramidHeight] =
                depthFirstSearch(remainingCards - cardsNeededForCurrentLayer, currentPyramidHeight + 1) +
                depthFirstSearch(remainingCards, currentPyramidHeight + 1);
        };

        // Call the depthFirstSearch function starting with all cards available and a pyramid height of 0
        return depthFirstSearch(cardsNumber, 0);
    }
};

```

Typescript Solution

```typescript
// Array `memo` to hold memoization results for dynamic programming, initialized with -1.
let memo: number[][] = Array(n + 1)
    .fill(0)
    .map(() => Array(Math.floor(n / 3) + 1).fill(-1));

// Helper function `depthFirstSearch` which calculates the number of possible ways
// to construct house of cards with `remainingCards` cards left and current base width `baseWidth`.
const depthFirstSearch = (remainingCards: number, baseWidth: number): number => {
    // Calculate the number of cards needed to increase the base width of the house by `baseWidth`.
    const cardsRequired = baseWidth * 3 + 2;

    // Base case: If the number of required cards exceed remainingCards, no houses can be built, return 0.
    if (cardsRequired > remainingCards) {
        return 0;
    }

    // Base case: If the number of required cards matches the remainingCards, exactly one house can be built, return 1.
    if (cardsRequired === remainingCards) {
        return 1;
    }

    // Check if the value has been memoized; if not, calculate it using recursion.
    if (memo[remainingCards][baseWidth] === -1) {
        // The current `memo[remainingCards][baseWidth]` is calculated by two recursive calls:
        // 1. Adding the cards required to the current layer (increase the base width),
        // 2. Going to the next layer without increasing the current base width.
        memo[remainingCards][baseWidth] = depthFirstSearch(remainingCards - cardsRequired, baseWidth + 1)
                                        + depthFirstSearch(remainingCards, baseWidth + 1);
    }

    // Return the memoized value.
    return memo[remainingCards][baseWidth];
};

// Function `houseOfCards` is the main entry function called with `n` cards available.
// It calculates the number of distinct "houses of cards" that can be built with `n` cards.
function houseOfCards(n: number): number {
    // Initialize the global memoization array according to the number of cards `n`.
    memo = Array(n + 1)
        .fill(0)
        .map(() => Array(Math.floor(n / 3) + 1).fill(-1));

    // Start the depth-first search recursion from base width 0.
    return depthFirstSearch(n, 0);
}

```

Time and Space Complexity

The provided Python code defines a function `houseOfCards` which calculates the number of different "houses" of cards that can be built with `n` cards. The recursive function `dfs` uses memoization (here implicitly via the `@cache` decorator from Python's `functools` module) to avoid redundant calculations for the same `(n, k)` pairs.

Time Complexity

The time complexity of this recursive function is O(`n^2`) in the worst case. This can be deduced by considering the following:

- Each level of the recursion represents a new `k` (the number of triangles at the current level), which goes from `0` up to `n/3` (since the base case is 3 cards for the smallest triangle).
- At each level of recursion, the function might be called twice – once with a decreased `n` (`n - x`) and once with the same `n` but an increased `k` (`k + 1`).
- However, not all combinations of `n` and `k` are valid, as `n` needs to be at least `3*k + 2` to form a triangle.
- The memoization effectively limits the number of unique recursive calls to O(`n^2`). With each additional layer, the number of potential triangles increments by one, but the number of cards required increases linearly with `k`. Hence, the number of states that need to be computed doesn't grow exponentially.

Therefore, the time complexity is O(`n^2`).

Space Complexity

The space complexity is O(`n^2`) as well. This is because of the following reasons:

- The memoization stores results for every unique `(n, k)` pair. Since `k` can range up to `n/3`, we would expect a maximum of `n/3 * n/3` states to be stored, given that not all `n` values will be computed for each `k`.
- Each call to `dfs` adds a new frame to the call stack. However, with memoization, the depth of the recursion is significantly reduced since the results of the common subproblems are reused.

Therefore, the space complexity of the algorithm, considering both the memoization cache and the recursion call stack, is O(`n^2`).