1259. Handshakes That Don't Cross

Hard Math Dynamic Programming

Leetcode Link

Problem Description

In the given problem, we have `numPeople`, a number representing an even number of people standing in a circle. The task is to calculate the number of different ways these people can perform handshakes. The condition is that each person shakes hands with exactly one other person, resulting in `numPeople / 2` handshakes in total. The twist here is that handshakes are not allowed to cross over each other. If we illustrate people as points on the circumference of the circle and handshakes as chords connecting these points, we must find the number of ways to draw these chords so that none intersect. Because the number of potential handshakes can be extremely large, the answer should be returned modulo `10^9 + 7`.

Intuition

The problem can be approached by using the concept of Catalan Numbers which are used in combinatorial mathematics. The nth Catalan number counts the number of ways to form non-intersecting handshakes among 'n' pairs of people standing in a circle, which is exactly the scenario described by the problem.

The intuition for the solution is to use dynamic programming due to the overlapping subproblems observed. The process starts with the basic understanding that any handshake splits the circle into two independent subproblems.

This is because once any two people shake hands, those inside the handshake cannot affect those outside and vice versa. Therefore, we can recursively calculate the number of ways for the left and right groups formed by each handshake and multiply them. Summing these products for each possible initial handshake (choosing pairs in turns) will give the total number of ways for `numPeople`.

The use of memoization (via Python's `@cache` decorator) optimizes the solution by storing results of the subproblems so that they don't need to be recalculated multiple times.

Solution Approach

The reference solution uses a recursive approach with dynamic programming to efficiently calculate the number of non-crossing handshake combinations in a circle of `numPeople`. Here's a step-by-step breakdown of the algorithm:

1. **Dynamic Programming and Cache**: The `@cache` decorator is used for memoization, which stores the results of the recursive function `dfs` to avoid recalculating them. This is essential for improving the efficiency as `dfs` will be called with the same parameters multiple times.
2. **Recursive Function `dfs`**: This function takes a single argument `i`, representing the number of people left to be paired for handshakes. The base case is when `i` is less than or equal to 1 (i.e., when there are no people or just one person left), the function returns 1 because there is only one way (or none, respectively) to organize the handshakes.
3. **Iterating Over Possible Pairings**:
   - Inside the recursive function, a loop iterates over every possible pairing of two people. The loop variable `l` represents the number of people in the left subproblem, which is incremented by 2 each time to ensure pairs are formed.
   - For each pair formed by people `l` and `l+1`, a subproblem to the left with `l` people and a subproblem to the right with `i - l - 2` people are created.
4. **Calculating Subproblems**:
   - For each pairing, the number of ways to organize handshakes is recursively calculated for both the left and the right subproblems. This corresponds to the number of ways people within each subproblem can shake hands without crossings.
5. **Combining Results and Applying Modulo**:
   - The results from the left and right subproblems are multiplied to get the total number of combinations that include that specific pairing. This is because the subproblems are independent, and the combination of their handshakes leads to a valid overall arrangement.
   - The running total `ans` is updated by adding this product. A modulo operation is applied to `ans` to ensure that the number stays within bounds specified by the problem (`10^9 + 7`).
6. **Returning the Result**:
   - Finally, the recursive function returns the total sum `ans` modulo `10^9 + 7`.

The main function `numberOfWays` calls `dfs` with `numPeople` as the argument to initiate the algorithm and returns the result, which is the total count of handshake configurations modulo `10^9 + 7`.

The solution effectively uses the divide-and-conquer strategy to reduce the complex problem into simpler, smaller instances. The memoization ensures that the overall time complexity is kept in check, with each subproblem being solved only once.

Example Walkthrough

Let's walk through a small example using the solution approach outlined above. Suppose `numPeople` is 4, which means we have 4 people standing in a circle.

First, we visualize these 4 people as points on a circle. Here are the steps we would follow:

1. **Dynamic Programming and Cache Initialization**: Before we dive into the recursion, the `@cache` decorator prepares the function to store any result computed for a specific number of people, which avoids redundant calculations.
2. **Calling the Recursive Function**: We call our recursive function `dfs(4)` since we want to calculate the number of ways 4 people can shake hands without crossing handshakes.
3. **Recursive Case `dfs(i)`**: The function takes `i = 4` as the number of people left to be paired.
   - Base cases (`i < 2`) return 1 but `i` is 4, so we proceed with recursion.
4. **Iterating Over Possible Pairings**:
   - We consider dividing the problem into subproblems. We start with person 0 and we try to make a pair with every other person, forming two subproblems per pair.
   - We have two possible pairings to consider: (0,1) and (0,2). Note that (0,3) would create a single subproblem without sub-divisions, which is implicitly included in the other pairings.
5. **Calculating Subproblems for Pair (0,1)**:
   - Choosing the pair (0,1) creates two subproblems: left with 0 people and right with 2 people (`dfs(2)`).
   - `dfs(2)` is a smaller instance of the problem where the base case applies: there is exactly 1 way for 2 people to shake hands without any crossings.
6. **Calculating Subproblems for Pair (0,2)**:
   - Choosing the pair (0,2) creates two subproblems: left with 2 people (`dfs(2)`) and right with 0 people.
   - As above, `dfs(2)` returns 1 since there's only 1 way to pair 2 people without crossings.
7. **Combining Results and Applying Modulo**:
   - For pair (0,1), the total for the subproblems is `1 (left) * 1 (right) = 1`.
   - For pair (0,2), the total for the subproblems is `1 (left) * 1 (right) = 1`.
   - We sum these up, which gives us `1 + 1 = 2`, the total number of non-intersecting handshake combinations for 4 people.
8. **Returning the Result**:
   - As there are no more pairs to evaluate, the function would return `2`.

When the main function calls `dfs(4)`, it would ultimately return the result of `2` as the count of handshake configurations for 4 people, modulo `10^9 + 7` (though in this small example, the modulo operation is not necessary).

This illustrates the efficiency of the dynamic programming approach; even though the number of potential configurations can grow rapidly with more people, each subproblem is only solved once and reused as needed, yielding a much faster overall solution.

Python Solution

```python
from functools import lru_cache  # Import the caching decorator

class Solution:
    def numberOfWays(self, numPeople: int) -> int:
        # Define a modulo constant for the results.
        MOD = 10**9 + 7

        # Use the lru_cache to memoize the previous results.
        @lru_cache(maxsize=None)
        def count_ways(people_left: int) -> int:
            # If no person or only one person is left, only one way is possible (base case).
            if people_left < 2:
                return 1

            # Initialize answer for this subproblem.
            answer = 0

            # Loop through the people by pairs, with a step of 2 because each person must be paired.
            for left_end in range(0, people_left, 2):
                # The right_end is calculated based on how many are to the left and subtracting 2 for
                # the partners in the current pair.
                right_end = people_left - left_end - 2

                # Calculate the ways by taking the product of ways for the left and right subproblems.
                answer += count_ways(left_end) * count_ways(right_end)
                # Apply the modulo at each step to keep the number within the integer limit.
                answer %= MOD

            # Return the computed answer for this subproblem.
            return answer

        # Start the recursion with the total number of people.
        return count_ways(numPeople)

# You can execute the code like this:
# solution = Solution()
# result = solution.numberOfWays(numPeople)
# where `numPeople` is the number of people you want to find the number of ways for.

```

Java Solution

```java
class Solution {
    private int[] cache; // Cache the results to avoid recomputation
    private final int MOD = (int)1e9 + 7; // Modulo constant for large numbers

    public int numberOfWays(int numPeople) {
        cache = new int[numPeople + 1]; // Initialize cache with the number of people
        return dfs(numPeople); // Use DFS to calculate the number of ways
    }

    // Helper method to compute the number of ways recursively
    private int dfs(int n) {
        // Base case: no people or a single pair
        if (n < 2) {
            return 1;
        }
        // Return cached value if already computed
        if (cache[n] != 0) {
            return cache[n];
        }
        // Iterate over possible splits with every second person
        for (int left = 0; left < n; left += 2) {
            int right = n - left - 2; // Number of people on the right side
            // Combine ways of left and right side under modulo
            cache[n] = (int)((cache[n] + (long)dfs(left) * dfs(right) % MOD) % MOD);
        }
        return cache[n]; // Return the total number of ways for n people
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Method to calculate the number of ways to form handshakes without crossing hands
    // "numPeople" is the number of people forming handshakes
    int numberOfWays(int numPeople) {
        const int MOD = 1e9 + 7; // Modulo to prevent integer overflow
        vector<int> dp(numPeople + 1, 0); // Create a dynamic programming table initialized to 0

        // Lambda function to calculate the number of ways using recursion and dynamic programming
        function<int(int)> countWays = [&](int n) {
            // Base case: for 0 or 2 people there's only one way to perform handshake (0: no one, 2: one handshake possible)
            if (n < 2) {
                return 1;
            }
            // If already computed, return the stored result to avoid re-computation
            if (dp[n] != 0) {
                return dp[n];
            }
            // Iterate over all possible even partitions of the n-people group
            for (int left = 0; left < n; left += 2) {
                int right = n - left - 2;
                // Recursively compute combinations for left and right partitions, then combine for total
                dp[n] = (dp[n] + (1LL * countWays(left) * countWays(right)) % MOD) % MOD;
            }
            // Return the result for n people
            return dp[n];
        };

        // Start the recursion with the total number of people
        return countWays(numPeople);
    }
};

```

Typescript Solution

```typescript
// Function to calculate the number of non-crossing handshakes for a given number of people
function numberOfWays(numPeople: number): number {
    const modulo = 10 ** 9 + 7; // Define module for large number arithmetic to prevent overflow
    const dp: number[] = Array(numPeople + 1).fill(0); // Initialize dynamic programming table with zeros

    // Define a helper function using DFS to find the number of ways to complete non-crossing handshakes
    const findWays = (peopleLeft: number): number => {
        if (peopleLeft < 2) { // Base case: if nobody or only one person is left, there is only one way
            return 1;
        }

        // Check if we have already computed the value for the current number of people
        if (dp[peopleLeft] !== 0) {
            return dp[peopleLeft];
        }

        // Iterate through the problem reducing it into subproblems
        for (let left = 0; left < peopleLeft; left += 2) {
            const right = peopleLeft - left - 2;
            // Calculate the number of ways by multiplying the subproblems and apply modulo
            dp[peopleLeft] += Number((BigInt(findWays(left)) * BigInt(findWays(right))) % BigInt(modulo));
            dp[peopleLeft] %= modulo; // Apply modulo to keep the number within the integer range
        }

        return dp[peopleLeft];
    };

    // Invoke the helper function with numPeople to find the result
    return findWays(numPeople);
}

```

Time and Space Complexity

The provided code solves the problem by using dynamic programming with memoization. Here is the analysis of the time complexity and space complexity:

Time Complexity

The `dfs` function essentially calculates the number of ways to pair `numPeople` people such that they form `numPeople / 2` pairs. Each person can choose a partner from the remaining `numPeople - 1` people, and then the problem is reduced to solving for `numPeople - 2`. However, due to the memoization (`@cache`), each subproblem is solved only once. The total number of subproblems corresponds to the number of even numbers from `0` to `numPeople - 2`, which is approximately `numPeople/2`.

For every call to `dfs(i)`, we iterate over all even values less than `i`, doing constant work each time, and each recursive call is to a problem size that is strictly smaller than the current problem, leading to a recursion tree of height `numPeople/2`.

Considering all of these factors, the time complexity is `O(n^2)` where `n` is `numPeople/2`, because there will be `n` layers of recursion and each layer requires summing up `n` subproblem results. Therefore, when `n` is `numPeople`, the time complexity is `O((n/2)^2)` which simplifies to `O(n^2)`.

Space Complexity

The space complexity is dominated by the memory used for storing the intermediate results in the cache. Since there are `n/2` layers of recursion, with each layer requiring constant space plus the call stack, the space complexity is `O(n)`, where `n` is `numPeople/2`.

Additionally, each recursive call uses a stack frame, but since we only make a new call after finishing with the previous call, the depth of the call stack will never exceed `n/2`. Therefore, taking the call stack into account does not change the overall space complexity, which remains `O(n)`.

Combining these observations, the overall space complexity of the algorithm is `O(n)`, where `n` is `numPeople/2`. Simplifying, we refer to it simply as `O(n)`.

In the above analysis, `n` refers to the `numPeople` variable.