935. Knight Dialer

Medium Dynamic Programming

Leetcode Link

Problem Description

In this problem, you are presented with a chess knight placed on a standard 3×4 numeric keypad, where the knight can only stand on the numbered keys (1-9 and 0). The rules for the knight's movement are the same as in the chess game – it moves in an "L" shape, which means it can move two squares in one direction and then one square in a perpendicular direction. The task is to calculate the number of distinct phone numbers of a given length `n` that can be dialed with the knight starting from any number (i.e., making `n-1` knight jumps). Since the number of distinct phone numbers could be very large, the answer should be returned modulo `10^9 + 7`.

Intuition

To solve this problem, we use dynamic programming to keep track of the number of ways the knight can reach a certain key on the keypad after a certain number of moves. We initialize an array `f` where each index represents a digit on the keypad and the value at each index represents the number of ways to reach that digit. Initially, the knight can be on any digit, so each value in `f` is 1.

For each subsequent move (up to `n-1` moves), we create a temporary array `t` to calculate the new number of ways to reach each digit from all other reachable digits, based on the unique movement pattern of the knight. Each entry `t[i]` is updated by adding the counts of the digits from which the knight can move to reach `i`, based on the possible knight moves. For instance, to reach digit 0, the knight could have come from digits 4 or 6, so `t[0]` is updated to the sum of `f[4]` and `f[6]`.

We repeat this process `n-1` times and then sum up the counts of all the keys in the final array `t` to get the total number of distinct phone numbers of length `n`. The sum is then taken modulo `10^9 + 7` to get the result within the required range.

Solution Approach

The solution uses dynamic programming to store the number of ways we can dial a number ending with each digit. Specifically, we know that each digit can be reached from one or more other digits. By repeatedly applying this knowledge, we can calculate the number of combinations leading to each digit step by step.

Here's an explanation of the flow of the algorithm:

- We begin with a list `f` of ten elements, each initialized to 1. This array represents the count of ways to reach every digit from 0 to 9 with zero knight moves (the starting position).
- For each move from the second (index 1) to the `n`th, we prepare a temporary array `t`, also with ten elements initialized to 0. This array will hold the number of ways to reach each digit after the current move.
- We iterate over all digits in `f` and update `t` based on the possible knight moves. For example, `t[0]` is the sum of the ways to arrive at `4` and `6` from `f`, because a knight move can get you from `4` or `6` to `0`.
- The connections based on the keypad layout and knight moves are hardcoded in this logic:

```
1 t[0] = f[4] + f[6]
2 t[1] = f[6] + f[8]
3 t[2] = f[7] + f[9]
4 t[3] = f[4] + f[8]
5 t[4] = f[0] + f[3] + f[9]
6 t[6] = f[0] + f[1] + f[7]
7 t[7] = f[2] + f[6]
8 t[8] = f[1] + f[3]
9 t[9] = f[2] + f[4]
```

  Notice that `t[5]` is missing because the key `5` is not reachable by the knight from any other key.

- After updating `t`, we assign its values to `f`, and then we reuse `f` in the next iteration for subsequent moves.
- After all iterations (n-1 moves), we sum up the numbers in `f`, which by now contains the count of ways to finish dialing a number with a final move to each digit.
- We take the sum modulo `10e+9 + 7` to ensure the final result fits within the integer range specified in the problem.

This algorithm employs dynamic programming to efficiently compute the possible number sequences, keeping only the counts of sequences ending in each digit rather than maintaining a list of all sequences. This dramatically reduces the space and time complexity of the solution, allowing it to work for large values of `n`.

Example Walkthrough

Let's say we want to calculate the number of distinct phone numbers that can be dialed with a knight's movement on a 3×4 numeric keypad and our phone numbers should be of length `n = 3`.

1. Initialize the array `f` to represent our starting point where all 10 digits (0-9) are possible starting positions with `f=[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]`.
2. As per our example, we'll perform `n-1 = 2` moves.
3. For the first move, we use a temporary array `t` to calculate possible moves:
   - To move to `0`, a knight could have come from `4` or `6`, so `t[0] = f[4] + f[6] = 1 + 1 = 2`.
   - To move to `1`, it could come from `6` or `8`, so `t[1] = f[6] + f[8] = 1 + 1 = 2`, and so on for the rest of the digits, ensuring `t[5]` remains `0`.
   - We end up with `t = [2, 2, 2, 2, 3, 0, 3, 2, 2, 2]`.
4. Then we copy `t` back to `f` and proceed to the next move with `f` now being `[2, 2, 2, 2, 3, 0, 3, 2, 2, 2]`.
5. Now, for the second (and final) move, we use the same process to update `t` using the new `f` values. Each element of `t` will represent the count of distinct phone numbers of length 3 ending with the respective digit.
6. After these moves and computing the last `t` values given the knight movement restrictions, we could end up with `t` looking something like `[5, 4, 4, 4, 8, 0, 8, 4, 4, 4]` (the exact values would depend on the connections between the numbers based on `t`).
7. Finally, we sum all the values (excluding `t[5]` which is `0`) to find the total number of distinct 3-digit phone numbers we can dial. Assume our sum equals `5`.
8. Since the problem instructs us to return the result modulo `10^9 + 7`, we compute `5 % (10e+9 + 7)` to find the result within the required range.

In this illustrative example, the temporary array `t` helps keep track of the possible knight moves after each step, and the dynamic programming aspect efficiently reuses and updates `f` to represent the number of ways to reach each digit. The summing up values in `f` after `n-1` moves yield the total number of distinct phone numbers that can be dialed, capturing the essence of the approach described in the provided content.

Python Solution

```python
class Solution:
    def knightDialer(self, n: int) -> int:
        # Base case: if there's only one move, the knight can be on any of the 10 digits
        if n == 1:
            return 10

        # Initialization of the count array representing the knight's current position
        counts = [1] * 10

        # Loop to calculate the number of distinct phone numbers of length n
        for _ in range(n - 1):
            # Temporary array to store new counts after the knight moves
            temp_counts = [0] * 10

            # The knight moves to calculate the counts for each position
            temp_counts[0] = counts[4] + counts[6]
            temp_counts[1] = counts[6] + counts[8]
            temp_counts[2] = counts[7] + counts[9]
            temp_counts[3] = counts[4] + counts[8]
            temp_counts[4] = counts[0] + counts[3] + counts[9]
            # No 5 in the move set since the knight cannot move from digit 5 to any other digit
            temp_counts[6] = counts[0] + counts[1] + counts[7]
            temp_counts[7] = counts[2] + counts[6]
            temp_counts[8] = counts[1] + counts[3]
            temp_counts[9] = counts[2] + counts[4]

            # Update counts for the next iteration
            counts = temp_counts

        # Return the sum of all the counts modulo 10^9 + 7 at the end of all moves
        # This is done to handle large numbers, as required by the problem statement
        return sum(counts) % (10**9 + 7)
```

Java Solution

```java
class Solution {
    private static final int MOD = (int) 1e9 + 7;

    public int knightDialer(int n) {
        // If there's only one move, the knight can be on any of the 10 digits
        if (n == 1) {
            return 10;
        }

        // Initialize an array to track the number of unique phone numbers
        int[] counts = new int[10];
        Arrays.fill(counts, 1);

        // Loop decrementing n to find unique numbers for each step
        while (--n > 0) {
            // Temporary array to hold the next state's number of unique phone numbers
            int[] temp = new int[10];
            // The knight on 0 can move to 4 or 6
            temp[0] = counts[4] + counts[6];
            // The knight on 1 can move to 6 or 8, etc.
            temp[1] = counts[6] + counts[8];
            temp[2] = counts[7] + counts[9];
            temp[3] = counts[4] + counts[8];
            temp[4] = counts[0] + counts[3] + counts[9];
            // Skipping index 5 as it's unreachable
            temp[6] = counts[0] + counts[1] + counts[7];
            temp[7] = counts[2] + counts[6];
            temp[8] = counts[1] + counts[3];
            temp[9] = counts[2] + counts[4];
            // Updating the counts by taking modulo to prevent overflow
            for (int i = 0; i < 10; ++i) {
                counts[i] = temp[i] % MOD;
            }
        }

        // Sum up all the unique phone numbers
        long totalNumbers = 0;
        for (long value : counts) {
            totalNumbers = (totalNumbers + value) % MOD;
        }

        // Return the total count as an integer
        return (int) totalNumbers;
    }
}
```

C++ Solution

```cpp
using ll = long long;

class Solution {
public:
    int knightDialer(int n) {
        // Base case: If there's only one move, all the numbers [0-9] can be dialed once.
        if (n == 1) return 10;

        // Modulo used for avoiding integer overflow issues.
        const int MOD = 1e9 + 7;

        // Initialize a vector to keep track of the count of unique numbers
        // that can be dialed from each digit on the keypad.
        vector<ll> counts(10, 1LL); // Start with 1 move possible from each digit.

        // Iterate over the number of hops minus 1 (since we've already initialized
        // the first hop).
        while (--n) {
            vector<ll> tempCounts(10); // Temporary vector to compute the next state.

            // Following lines update the count of unique numbers that can be dialed
            // from each digit, based on the knight's move rules on the keypad.
            tempCounts[0] = counts[4] + counts[6];
            tempCounts[1] = counts[6] + counts[8];
            tempCounts[2] = counts[7] + counts[9];
            tempCounts[3] = counts[4] + counts[8];
            tempCounts[4] = counts[0] + counts[3] + counts[9];
            // There is no key 5 in the hop pattern, so it's skipped.
            tempCounts[6] = counts[0] + counts[1] + counts[7];
            tempCounts[7] = counts[2] + counts[6];
            tempCounts[8] = counts[1] + counts[3];
            tempCounts[9] = counts[2] + counts[4];

            // Update the counts vector with the new values, applying the modulo operation.
            for (int i = 0; i < 10; ++i) counts[i] = tempCounts[i] % MOD;
        }

        // Calculate the final answer as the sum of the counts modulo MOD.
        ll ans = accumulate(counts.begin(), counts.end(), 0LL) % MOD;

        return static_cast<int>(ans); // Cast the long long result to int before returning.
    }
};
```

Typescript Solution

```typescript
// Define type alias for long long equivalent in TypeScript
type ll = number;

// Define the constant MOD to prevent integer overflow
const MOD: number = 1e9 + 7;

// Initializes a vector to keep track of the count of unique numbers that can be dialed
// from each digit on the keypad, with 1 move possible from each digit.
let counts: ll[] = new Array(10).fill(1);

// Function to compute the number of distinct phone numbers of length `n`
// that can be dialed using a knight's moves on a standard 10-key keypad
function knightDialer(n: number): number {
    // Base case: With only one move, all numbers [0-9] can be dialed once.
    if (n === 1) return 10;

    // Iterate over the number of hops minus 1 (since we've already initialized
    // the first hop).
    while (--n) {
        let tempCounts: ll[] = new Array(10).fill(0); // Temporary array for the next state.

        // Update the count of unique numbers that can be dialed from each digit,
        // based on the knight's move rules on the keypad.
        tempCounts[0] = (counts[4] + counts[6]) % MOD;
        tempCounts[1] = (counts[6] + counts[8]) % MOD;
        tempCounts[2] = (counts[7] + counts[9]) % MOD;
        tempCounts[3] = (counts[4] + counts[8]) % MOD;
        tempCounts[4] = (counts[0] + counts[3] + counts[9]) % MOD;
        // Digit 5 is not reachable by a knight's move, hence it's excluded.
        tempCounts[6] = (counts[0] + counts[1] + counts[7]) % MOD;
        tempCounts[7] = (counts[2] + counts[6]) % MOD;
        tempCounts[8] = (counts[1] + counts[3]) % MOD;
        tempCounts[9] = (counts[2] + counts[4]) % MOD;

        // Copy the newly computed values back into the original counts array
        counts = [...tempCounts];
    }

    // Calculate the final answer by summing the values in counts and applying the MOD.
    let ans: ll = counts.reduce((acc, val) => (acc + val) % MOD, 0);

    return ans; // Return the final count as an integer
}

// Example usage
// const numberOfWays = knightDialer(2);
// console.log(numberOfWays); // Output the number of unique numbers that can be dialed
```

Time and Space Complexity

The given Python code defines a function `knightDialer` that calculates the number of distinct phone numbers of a certain length that can be dialed using the movements of a knight on a phone keypad.

Time Complexity

The time complexity of the `knightDialer` function is $O(n)$, where `n` is the length of the phone number to dial. This is because the function contains a single loop that runs `n - 1` times, and within each loop iteration, the operations are constant time assignments. There is also a sum operation after the loop, which takes $O(10)$, i.e., constant time, because there are always 10 digits.

The time complexity is calculated as follows:

- Loop runs `n - 1` times.
- Each loop operation is a constant time operation due to predetermined transitions between keypad digits.

There are no nested loops or recursive calls that would increase the complexity, so the overall time complexity remains linear with the input `n`.

Space Complexity

The space complexity of the `knightDialer` function is $O(1)$, since the storage requirement does not scale with the input size `n`. The arrays `f` and `t` used in the function always have a fixed size of 10, which corresponds to the 10 digits in the phone keypad.

No additional space that grows with `n` is allocated during the computation, thus the space complexity is constant.

The space complexity is calculated as follows:

- Two fixed-size arrays `f` and `t` with size 10 each.
- Constant space for variables and loop iteration.

As a result, the amount of memory used does not depend on the input `n`, leading to a constant space complexity.