2827. Number of Beautiful Integers in the Range

Hard Math Dynamic Programming Leetcode Link

Problem Description

The problem requires us to find out how many integers within a given range `[low, high]` can be considered "beautiful". An integer is "beautiful" if it meets two criteria:

1. The count of even digits in the number is equal to the count of odd digits.
2. The number must be divisible by `k`.

We're tasked with returning the count of such beautiful integers.

Intuition

To solve this problem, the solution employs a technique known as "Digit Dynamic Programming" (Digit DP). The idea behind Digit DP is to build numbers digit by digit, keeping track of various conditions that must be met. It allows us to calculate the number of valid numbers below a certain threshold.

The solution uses a depth-first search (DFS) function with memoization to incrementally construct possible integers digit by digit, and at each step, checks if the conditions of beauty are satisfied.

1. The `dfs` function is recursively called for each position `pos` in the number while tracking:
   - `mod`: the current remainder when the number constructed so far is divided by `k`.
   - `diff`: a measure to determine if the count of even digits is equal to the count of odd digits.
   - `lead`: a flag to denote if we're still at leading zeroes.
   - `limit`: a flag to indicate if we should be bounded by the number formed by the integer's digits up to this point or be free to consider all digits from 0 to 9.
2. The `diff` parameter is maintained as an offset of 10 and adjusted by adding 1 for odd digits and subtracting 1 for even digits with the goal of reaching a `diff` of exactly 10 to ensure there's an equal count of odd and even digits.
3. The recursive calls construct the number by exploring all the possibilities for each digit's place, respecting the current constraints (like if we are limited by the maximum range or if we are considering leading zeroes).
4. The function returns the count of valid consecutive integer sequences that fit the defined beauty requirements up to the `high` limit and separately up to the `low - 1` limit.
5. The actual count of beautiful integers within the `[low, high]` range is determined by subtracting the count obtained for `low - 1` from the count obtained for `high`.

The solution captures the subtleties of the problem by using smart state transitions that ensure no possibility is left unexamined while at the same time preventing wasteful repetitions through memoization. The combination of DFS for enumeration and memoization for efficiency is a hallmark of the Digit DP approach.

Solution Approach

The solution uses the following algorithms, data structures, and patterns:

1. **Depth-First Search (DFS)**: DFS allows us to explore all possible integer combinations by traversing through each digit from most significant to least significant.
2. **Memoization**: The `@cache` decorator in Python is used for memoization, effectively storing the results of the DFS function calls with a particular set of parameters to avoid recalculations.
3. **Dynamic Programming (DP)**: By using memoization and the DFS pattern, the solution utilizes dynamic programming to build upon previously computed states.

The implementation specifics are as follows:

- The key function in the solution is `dfs(pos, mod, diff, lead, limit)`, which is used to recursively explore all possible numbers digit by digit.
- `pos` keeps track of the current digit index starting from the leftmost digit.
- `mod` represents the current remainder when the partially constructed number is divided by `k`.
- `diff` is managed such that its final value should be 10, representing an equal number of odd and even digits (initialized to 10, odd digits add 1, even digits subtract 1).
- `lead` is a boolean flag indicating whether the current series of recursions are still in the leading zeros part of the number.
- `limit` is a flag to ensure we don't exceed the upper bound of the high-end of the range during the DFS.
- The DFS function is implemented to stop when `pos` exceeds the length of the string representation of the current boundary (`high` or `low - 1`).
- Iterating over all digits 0-9 (0-up):
  - If the current digit is a leading zero, the recursive call adjusts only `pos` and `limit`. It continues leading zero considerations by keeping `lead` as true.
  - If a nonzero digit is placed, the function updates `mod`, `diff`, and sets `lead` to false, as we are now creating a nonzero number.
- The answer for the upper bound `high` is obtained by converting `high` to a string and passing it through the `dfs` function. The DFS function is then reset by clearing its cache.
- Another call to `dfs` is made using `low - 1` to count the valid numbers up to the lower limit, ensuring that we don't count numbers outside the given range.
- Finally, the difference between these two values gives us the count of beautiful integers within the `[low, high]` range.

The algorithm effectively breaks down a complicated counting problem into manageable states by using DFS and DP, while memoization guarantees that the time complexity remains controlled by caching the results of states that have already been computed.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we want to find the number of beautiful integers within the range `[20, 23]` where `k = 2`, meaning each beautiful number must be divisible by 2.

1. **Initialization**: We start off by initializing `diff` to 10. We'll use the `dfs` function to explore possible numbers starting at position 0 (the leftmost digit).
2. **Exploring Number 20**: For the first number, 20:
   - The leftmost digit is 2, which is even. So, we decrease `diff` by 1 (`diff = 9`), signifying one more even than odd digits so far.
   - The next digit is 0, which is even, and since it's a leading zero, we only change our position and maintain `lead` as true.
   - The number 20 is divisible by 2, which satisfies the second condition (`k = 2`).
3. **Moving to the Next Number**: We cannot increment beyond the number 23 as it is our `high` limit, so we look at the number 21, in which we add 1 to `diff` for the odd last digit, making it `diff = 10` again. The number 21, however, is not divisible by 2 and hence isn't beautiful.
4. **Continuing With Numbers 22 and 23**: We continue this process for 22 and 23:
   - For 22, we adjust `diff` to 8, and since it is divisible by 2, it meets the conditions.
   - For 23, `diff` is back to 9, but since 23 isn't divisible by 2, it isn't considered.
5. **Counting Beautiful Numbers**: Out of the numbers 20 through 23, the ones that satisfy all the conditions are 20 and 22. Hence, there are 2 beautiful integers.
6. **Adjusting for Lower Limit**: We also need to subtract the number of valid sequences up to one less than the lower limit (`low - 1`), which in this case would be `[20, 19]`. We perform the same operation for this range but expect to find 0 beautiful integers since it's below our range starting point.
7. **That yields our final answer**: After subtracting the count from `low - 1`, we still have 2 beautiful integers as our count.

This example demonstrates how the `dfs` function would explore all possible numbers for the given range, adjusting `mod` and `diff` accordingly to identify valid integers. Memoization ensures that if we were to calculate the same state again, we retrieve the count from the cache instead of recomputing. The final answer is obtained by the arithmetic difference between the upper bound (`dfs(high)`) and the lower bound adjusted by one (`dfs(low - 1)`).

Python Solution

```python
class Solution:
    def numberOfBeautifulIntegers(self, low: int, high: int, k: int) -> int:
        from functools import lru_cache

        @lru_cache(None)
        def dfs(position: int, modulo: int, distinct_count: int, is_leading: int, is_limited: int) -> int:
            # If we've constructed a number of the same length,
            # check if it's divisible by k and has 10 distinct digits
            if position == len(num_str):
                return modulo == 0 and distinct_count == 10

            upper_limit = int(num_str[position]) if is_limited else 9
            ans = 0

            # Try all possible digits for current position
            for digit in range(upper_limit + 1):
                # Leading zeros are treated differently since they don't affect the distinct digit count
                if digit == 0 and is_leading:
                    ans += dfs(position + 1, modulo, distinct_count, 1, is_limited and digit == upper_limit)
                else:
                    # Adjust the distinct digit count depending on the parity of the digit
                    next_distinct = distinct_count + (1 if digit % 2 == 1 else -1)
                    ans += dfs(position + 1, (modulo * 10 + digit) % k, next_distinct, 0, is_limited and digit == upper_limit)

            return ans

        # Find the count of beautiful numbers up to 'high'
        num_str = str(high)
        count_high = dfs(0, 0, 10, 1, 1)
        dfs.cache_clear()  # Clear the cache to reuse the function

        # Find the count of beautiful numbers below 'low' (since we're excluding the lower boundary)
        num_str = str(low - 1)
        count_low = dfs(0, 0, 10, 1, 1)

        # The difference will give the count for the inclusive range [low, high]
        return count_high - count_low

# Explanation:
# The above class Solution contains a method named 'numberOfBeautifulIntegers' which calculates the
# count of beautiful integers within the closed interval [low, high] that are also divisible by 'k'.

# A 'beautiful integer' is defined as an integer that contains exactly 10 distinct digits and their
# parity (even or odd) alternates.

# The internal 'dfs' (depth first search) function is a recursive function that explores all the valid
# combinations of digits from the current 'position' up to the length of the number in string form 'num_str'.

# The 'modulo' parameter represents the current value modulo 'k', 'distinct_count' keeps track of the count
# of distinct digits encountered so far, 'is_leading' is a flag to check if we're still at leading zeroes,
# and 'is_limited' indicates if we have a digit limit based on the target number.

```

Java Solution

```java
class Solution {
    private String numberString;  // The string representation of the number we are working with
    private int k;                // The given k value
    private Integer[][][] cache = new Integer[11][21][21];  // Cache to store intermediate results for dynamic programming

    // Main method to find the number of 'beautiful' integers between low and high inclusive
    public int numberOfBeautifulIntegers(int low, int high, int k) {
        this.k = k;  // Set the global k value
        numberString = String.valueOf(high);  // Convert the upper limit to string
        int countUpToHigh = dfs(0, 0, 10, true, true);  // Count beautiful numbers up to high
        numberString = String.valueOf(low - 1);  // Convert the lower limit to string after subtracting one
        cache = new Integer[11][21][21];  // Reset the cache for the new range
        int countUpToLowMinusOne = dfs(0, 0, 10, true, true);  // Count beautiful numbers up to low-1
        return countUpToHigh - countUpToLowMinusOne;  // Return the difference as the result
    }

    // Helper method to perform depth-first search and count beautiful numbers
    private int dfs(int pos, int mod, int diff, boolean isLeadingZero, boolean isLimit) {
        // Termination condition: if we have reached the end of the number string
        if (pos >= numberString.length()) {
            // If at the end the mod is 0 and diff is 10 (no number has been used),
            // we have found a valid number, otherwise return 0
            return mod == 0 && diff == 10 ? 1 : 0;
        }

        // Check our cache to save time if the result is already computed
        if (!isLeadingZero && !isLimit && cache[pos][mod][diff] != null) {
            return cache[pos][mod][diff];
        }

        int ans = 0;    // Initialize the answer for the current position to 0
        int upperBound = isLimit ? numberString.charAt(pos) - '0' : 9; // Set the maximum digit we can place here

        // Iterate through all possible digits we can place
        for (int digit = 0; digit <= upperBound; ++digit) {
            if (digit == 0 && isLeadingZero) {
                // If the current digit is 0 and we are still in the leading zeroes part
                ans += dfs(pos + 1, mod, diff, true, isLimit && digit == upperBound);
            } else {
                // Decide the next diff value based on the current digit
                int nextDiff = diff + (digit % 2 == 1 ? 1 : -1);
                // Add the count of beautiful numbers starting with the current digit
                ans += dfs(pos + 1, (mod * 10 + digit) % k, nextDiff, false, isLimit && digit == upperBound);
            }
        }

        // If we are not in leading zero or limit, update our cache
        if (!isLeadingZero && !isLimit) {
            cache[pos][mod][diff] = ans;
        }

        return ans;  // Return the cumulative count of beautiful numbers
    }
}

```

C++ Solution

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

class Solution {
public:
    int numberOfBeautifulIntegers(int low, int high, int k) {
        // Initialize the memoization table for dynamic programming.
        int memo[11][21][21];
        memset(memo, -1, sizeof(memo));
        // Convert the high limit number into a string.
        std::string high_str = std::to_string(high);

        // Declare the depth first search (dfs) function that we'll use for our digit DP.
        std::function<int(int, int, int, bool, bool)> dfs = [&](int position, int remainder, int even_odd_diff, bool leading_zeros,
            // Base case - if we've looked at all digits in high_str.
            if (position >= high_str.size()) {
                // Check if the number has equal odds and evens and if it's divisible by k.
                return (remainder == 0 && even_odd_diff == 10) ? 1 : 0;
            }
            // Check if we can use memoized data (We memoize on non-leading zeros and when we are not at the
            // numerical limit).
            if (!leading_zeros && !limit && memo[position][remainder][even_odd_diff] != -1) {
                return memo[position][remainder][even_odd_diff];
            }
            int count = 0;
            // Determine the limit for the current digit.
            int upper_bound = limit ? high_str[position] - '0' : 9;
            for (int digit = 0; digit <= upper_bound; ++digit) {
                if (digit == 0 && leading_zeros) {
                    // Go one level deeper while preserving the leading zeros.
                    count += dfs(position + 1, remainder, even_odd_diff, true, limit && digit == upper_bound);
                } else {
                    // Calculate the next even/odd differential.
                    int next_diff = even_odd_diff + (digit % 2 == 1 ? 1 : -1);
                    // Go one level deeper taking current digit into account.
                    count += dfs(position + 1, (remainder * 10 + digit) % k, next_diff, false, limit && digit == upper_bound);
                }
            }
            // Store the count for the current state into the memoization table.
            if (!leading_zeros && !limit) {
                memo[position][remainder][even_odd_diff] = count;
            }
            // Return running count of beautiful integers.
            return count;
        };

        // First, find the count of beautiful integers up to the high limit.
        int count_high = dfs(0, 0, 10, true, true);
        // Reset the memoization table for the next call.
        memset(memo, -1, sizeof(memo));
        // Convert the low limit number into a string.
        std::string low_str = std::to_string(low - 1);
        // Next, find the count of beautiful integers up to the low limit minus one.
        int count_low = dfs(0, 0, 10, true, true);
        // The final result is the difference between the two counts.
        return count_high - count_low;
    }
};

```

Typescript Solution

```typescript
function numberOfBeautifulIntegers(low: number, high: number, k: number): number {
    let highAsString = String(high);
    let memo: number[][][] = Array(11)
        .fill(0)
        .map(() =>
            Array(21)
                .fill(0)
                .map(() => Array(21).fill(-1)),
        );

    // Depth-first search function to find the number of beautiful integers
    const depthFirstSearch = (position: number, remainder: number, digitDifference: number, leadingZero: boolean, isLimit: boolean)
        // Base case: If the current position is at the end of the string length,
        // check if the number is divisible by k and the digit difference is 10 (beautiful).
        if (position === highAsString.length) {
            return remainder === 0 && digitDifference === 10 ? 1 : 0;
        }
        // If there's no leading zero, not at limit, and value is already computed in memo array, return the value.
        if (!leadingZero && !isLimit && memo[position][remainder][digitDifference] !== -1) {
            return memo[position][remainder][digitDifference];
        }
        let count = 0;
        const upperBound = isLimit ? Number(highAsString[position]) : 9;

        // Explore all possible next digits from 0 to upper bound
        for (let digit = 0; digit <= upperBound; ++digit) {
            if (digit === 0 && leadingZero) {
                // If the digit is 0 and it's a leading zero, we keep the leadingZero flag true.
                count += depthFirstSearch(position + 1, remainder, digitDifference, true, isLimit && digit === upperBound);
            } else {
                // Update the digit difference and the remainder when adding the current digit
                const nextDigitDifference = digitDifference + (digit % 2 === 0 ? -1 : 1);
                count += depthFirstSearch(position + 1, (remainder * 10 + digit) % k, nextDigitDifference, false, isLimit && digit
            }
        }

        // Memoize the result if there's no leading zero and not at the digit limit
        if (!leadingZero && !isLimit) {
            memo[position][remainder][digitDifference] = count;
        }
        return count;
    };

    // First, calculate the count of beautiful numbers less than or equal to high
    const countHigh = depthFirstSearch(0, 0, 10, true, true);
    // Update highAsString to the value of 'low - 1'
    highAsString = String(low - 1);
    // Reset memo array for the new range calculation
    memo = Array(11)
        .fill(0)
        .map(() =>
            Array(21)
                .fill(0)
                .map(() => Array(21).fill(-1)),
        );
    // Calculate the count of beautiful numbers less than 'low'
    const countLow = depthFirstSearch(0, 0, 10, true, true);
    // Return the difference to get the count of beautiful integers in the range [low, high]
    return countHigh - countLow;
}

```

Time and Space Complexity

The time complexity of the DFS function primarily depends on the number of possible states. The state is defined by the parameters `(pos, mod, diff, lead, limit)`. Since `pos` can take values from `0` to `L` where `L` is the number of digits in `high`, `mod` can range from `0` to `k - 1`, `diff` can theoretically range from `-L` to `L`, `lead` can be either `0` or `1`, and `limit` can also be either `0` or `1`, their multiplications define the number of states. However, note that `diff` values are actually going from `0` to `9` when counting unique differences, since `diff == 10` implies a valid count of unique differences. Hence, we have `2` options for `lead` and `limit` each, `L` options for `pos`, `k` options for `mod`, and `10` options for `diff`.

The time complexity can be roughly estimated as `O(10 * k * L * 2 * 2)`.

The space complexity is affected by the depth of the recursion and the memoization used. The recursion depth is `O(L)` since that is the maximum depth of the DFS. For memoization, we store a unique result for each of the possible states, giving us a space complexity similar to time complexity, which is `O(10 * k * L)` because we don't need to consider the space for precomputing `limit` and `lead`, which are just passed along in the recursive calls without consuming additional space.

Thus, the overall space complexity is `O(10 * k * L)`.