2698. Find the Punishment Number of an Integer

Medium Math Backtracking

Leetcode Link

Problem Description

The task here is to find the so-called "punishment number" for a given positive integer `n`. A "punishment number" is essentially the sum of the squares of particular integers ranging from 1 to `n`. An integer `i` (within this range) contributes to this sum if and only if the square of `i` (`i * i`) can be split into one or more contiguous substrings, and when these substrings are converted to integers and added together, the sum equals `i`.

For example, if `n` is 81, we look at each number from 1 to 81, square them, and see if the square can be partitioned into substrings that add up to the original number. In this case, if `n` is 81, *81 itself does not contribute because 81 is 6561 which cannot be split into substrings that add to 81. But the number 9 does contribute because 9*9 is 81, and 8+1 is 9.

Intuition

The solution involves iterating through each number `i` from 1 to `n`, calculating `i * i`, and checking if this square can meet the condition described above.

The intuition behind the implementation is that we can use recursion to check all possible substrings. For every position in the string representation of `i * i`, we recursively check all ways we could split the remaining string. If we reach the end of the string and the sum of the substrings equals `i`, then `i * i` is valid and should be included in our "punishment number" sum.

The `check` function in the code takes a string `s`, which represents the square of a number, an index `i` which indicates where to start partitioning, and an integer `t` that we are trying to match using the sum of substrings. It tries all possible ways to partition the string from index `i` onwards and returns true if any of those partitions sum up to `t`.

The main solution function `punishmentNumber` goes from 1 to `n`, checking each number `i` by calling the `check` function. If `check` returns true for a given `i`, it means we can sum up the substrings of `i * i` to get `i`, hence, we add `i * i` to our cumulative `ans` which is our punishment number.

This approach efficiently checks all possible partition points in the sequence and uses memorization by the nature of recursion to avoid rechecking the same combination of splits, leading to an optimal solution.

Solution Approach

To solve the "punishment number" problem, the implemented solution leverages recursion and backtracking as its core algorithmic concepts.

The `check` Helper Function

- This is a recursive function designed to check if a given squared number can be split into substrings that sum up to the original number before squaring.
- It takes the string `s` which represents the square of `i`, an index `i` which represents the current position in `s` that we're examining, and an integer `x` which is the value of the original number that we're aiming to reconstruct from the substrings.
- It iterates through the string starting from the current index `i` and accumulates a value `y`. If at any point the accumulated `y` is greater than `x`, there's no need to continue along that path.
- If `y` equals `x` and we are at the end of the string, then we know this is a valid partitioning. If not, we call `check` recursively with a new starting position just past the current substring we're considering. This recursive call will return `True` if the subsequent partitioning works out, or keep trying until all possibilities have been exhausted.

The `punishmentNumber` Main Function

- It iterates from 1 to `n`. For each iteration:
  - Calculates `i * i` to get the square of the current number.
  - It turns the square into a string and calls the `check` function with that string, the number `i` for starting index, and the original number `i` we are trying to partition into.
- If the `check` function returns `True`, which indicates that the square of `i` can be split into substrings adding up to `i`, then `i * i` gets added to the cumulative sum `ans`.
- After iteration completes, `ans` holds the sum of all valid squares and is returned as the final answer.

Algorithm and Data Structures

- The main data structure used here is string manipulation, as we're constantly slicing and parsing substrings.
- The algorithm uses recursion extensively to check all possible partitions, which inherently also uses a form of memoization due to overlapping subproblems within the recursive calls.
- The overall approach is a backtracking problem, where we try out different substring splits and backtrack if we realize that the current path does not lead us to a solution.

This solution is efficient because it explores all possible partitions while avoiding unnecessary work, which is reduced by not exploring splits that would lead to values greater than the target sum `x`. As `check` progresses, it "cuts off" branches that cannot possibly result in a successful split, thereby streamlining the recursive search process.

Example Walkthrough

Let's walk through an example to illustrate how the implemented solution works. Imagine we are tasked with finding the punishment number for `n = 10`. We want to check each number from 1 to 10 to see if its square can be split into substrings that sum up to itself.

Step 1: We start with `i = 1`.

- We calculate its square: `1 * 1 = 1`. There are no substrings to consider, so `1` is a valid partition by itself (1 = 1).
- The `check` function would receive `"1", 0, 1)` and return `True`, so we add `1 * 1 = 1` to the cumulative sum `ans`.

Step 2: Next, `i = 2`.

- Square it: `2 * 2 = 4`. Similar to `1`, `4` does not have substrings to split, so it doesn't meet the condition.
- The `check` function would receive `("4", 0, 2)` and return `False`.

And we continue this process...

Step 3: For `i = 3`.

- `3 * 3 = 9`. There's no way to split `9` into substrings that add up to `3`.
- The `check` function would receive `("9", 0, 3)` and return `False`.

Step 4: For `i = 9` (Skipping the steps for `i = 4` to `i = 8` for brevity).

- `9 * 9 = 81`. Here we can split `81` as `8` and `1`, where `8 + 1 = 9`.
- The `check` function would receive `("81", 0, 9)` and it would check the substring `"8"` (recursively calling `check` with `"1", 1, 1`) which returns `True`.
- Therefore, we add `9 * 9 = 81` to the cumulative sum `ans`.

Continued Iteration: We would continue iterating over each number up to and including `n`, performing similar calculations.

Final Step:

- After iterating from `1` through `n` (which is `10` in our example), we end up with `ans` holding the sum of valid squares, which, based on the provided numbers, are just `1` and `81`.
- The punishment number for `n = 10` would therefore be `1 + 81 = 82`.

This example demonstrates how the solution uses recursion to check possible splits for the square of each number and how the valid squares contribute to the punishment number.

Python Solution

```python
class Solution:
    def punishmentNumber(self, n: int) -> int:
        # Helper function to check if a square number can be split into two numbers that add up to the original number.
        def can_split_to_original(num_str, start_index, target_sum) -> bool:
            # Base case checks
            # If the starting index is beyond the string's length and target sum is zero, this is valid.
            if start_index >= num_length:
                return target_sum == 0

            current_num = 0
            # Consider different splits by incrementally including more digits.
            for j in range(start_index, num_length):
                current_num = current_num * 10 + int(num_str[j])
                # If the current number exceeds the target, there's no point in continuing.
                if current_num > target_sum:
                    break
                # Recursively check if the remaining string can fulfill the condition.
                if can_split_to_original(num_str, j + 1, target_sum - current_num):
                    return True
            return False

        total_sum = 0
        # Iterate over the range 1 through n, inclusive.
        for i in range(1, n + 1):
            square_num = i * i  # Calculate the square of the number.
            # If the squared number can be split as desired, add to total sum.
            if can_split_to_original(str(square_num), 0, i):
                total_sum += square_num

        # Return the total sum of all such square numbers.
        return total_sum
```

Java Solution

```java
class Solution {
    // Method to calculate the sum of special numbers up to n
    public int punishmentNumber(int n) {
        int sum = 0; // Initialize sum of special numbers
        // Iterate from 1 to n, inclusive
        for (int i = 1; i <= n; ++i) {
            int square = i * i; // Calculate the square of the current number
            // Check if the square can be expressed as a sum of numbers leading up to i
            if (isSpecial(square, 0, i)) {
                sum += square; // Add the square to the sum if it meets the condition
            }
        }
        return sum; // Return the total sum of special numbers
    }

    // Helper method to check if a string representation of a number can be split
    // into numbers adding up to a specific value
    private boolean isSpecial(String numStr, int startIndex, int remaining) {
        int length = numStr.length();
        // Base case: if we've reached the end of the string
        if (startIndex == length) {
            // Check if we've exactly reached the target sum
            return remaining == 0;
        }
        int currentNumber = 0;
        // Iterate through the substring starting at startIndex
        for (int j = startIndex; j < length; ++j) {
            // Update the current number by appending the next digit
            currentNumber = currentNumber * 10 + (numStr.charAt(j) - '0');
            // If the currentNumber exceeds the remaining sum needed, break (optimization)
            if (currentNumber > remaining) {
                break;
            }
            // Recursive call to check the rest of the string with the updated remaining value
            if (isSpecial(numStr, j + 1, remaining - currentNumber)) {
                return true; // If successful, return true
            }
        }
        return false; // If no combination adds up to remaining, return false
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to calculate the sum of special numbers
    int punishmentNumber(int n) {
        int sum = 0; // Initialize sum of the special numbers
        // Loop through all numbers from 1 to n
        for (int currentNum = 1; currentNum <= n; ++currentNum) {
            int square = currentNum * currentNum; // Calculate the square of the current number
            string squareStr = std::to_string(square); // Convert the square to a string
            // Check if the square satisfies the condition
            if (isSpecialNumber(square, squareStr, 0, currentNum)) {
                sum += square; // Add the square to the sum if the condition is satisfied
            }
        }
        return sum; // Return the sum of special numbers
    }

    // Helper function to check if the given string represents a special number
    bool isSpecialNumber(int originalNum, string numStr, int startIndex, int remainingSum) {
        int totalLength = numStr.size(); // Size of the number string
        // If we've reached beyond the end of the string, check if sum is 0 (completely decomposed)
        if (startIndex == totalLength) {
            return remainingSum == 0;
        }
        int partialSum = 0; // Initialize the partial sum of digits from 'index'
        // Iterate through the string starting from 'index'
        for (int j = startIndex; j < totalLength; ++j) {
            partialSum = partialSum * 10 + (numStr[j] - '0'); // Build the number by appending the digit
            // If the partial sum exceeds the remaining value of num, break (no further checks needed)
            if (partialSum > remainingSum) {
                break;
            }
            // Recurse to check if the remaining part of the string can make up the number num - partialSum
            if (isSpecialNumber(originalNum, j + 1, remainingSum - partialSum)) {
                return true; // If the condition holds, return true
            }
        }
        // If no valid decomposition found, return false
        return false;
    }
};
```

Typescript Solution

```typescript
// Define the punishmentNumber function which calculates the sum of all the special numbers upto n.
// Special numbers are those whose square can be split into two or more integers that sum to the original number.
function punishmentNumber(n: number): number {
    // Define helper function check to recursively determine if the square of a number
    // can be split into parts that sum to the original number. 'index' tracks the current position,
    const checkSplit = (squareAsString: string, index: number, targetSum: number): boolean => {
        const length = squareAsString.length;
        if (index == length) {
            return targetSum == 0;
        }
        let currentSum = 0;
        for (let j = index; j < length; ++j) {
            currentSum = currentSum * 10 + Number(squareAsString[j]);
            if (currentSum > targetSum) {
                break;
            }
            // Recursively check the next part of the string
            if (checkSplit(squareAsString, j + 1, targetSum - currentSum)) {
                return true;
            }
        }
        return false;
    };

    // Initialize the sum variable to keep track of the total sum of special numbers
    let sumOfSpecialNumbers = 0;

    // Loop through each number from 1 to n to find all special numbers
    for (let i = 1; i <= n; ++i) {
        const square = i * i; // Calculate the square of the current number
        const squareAsString = square.toString(); // Convert the square to a string for processing

        // Check if the square can be split into integers that sum to the original number
        if (checkSplit(squareAsString, 0, i)) {
            sumOfSpecialNumbers += square; // If it can be, add the square to the sum
        }
    }

    // Finally, return the sum of all special numbers found
    return sumOfSpecialNumbers;
}
```

Time and Space Complexity

The given Python code defines a method `punishmentNumber` which calculates the sum of squares of certain integers up to `n` that satisfy a particular condition. The condition is verified using a helper method `check` that implements a recursive approach to determine if the square of the number can be represented as a consecutive sum of integers ending with the number itself.

Time Complexity:

The main time complexity comes from two sources:

1. The outer for loop iterating from 1 to `n`.
2. The recursive `check` function called for each `i`.

For each `i`, the `check` function explores different partitions of the digits of `i*i` into sums that could add up to `i`. In the worst case, this might result in checking each partition which leads to a search space that is exponential in the number of digits of `i*i`. However, the recursive search is pruned whenever a partition sum exceeds `x`, which can significantly reduce the number of searched partitions.

The number of digits of `i*i` is `O(log(i))` which means the worst case complexity of the `check` function is `O(2^d)`. Since `d` is `O(log(i))`, the recursive calls for each `i` may reach a complexity of `O(2^(2 * log(i)))` due to the squaring operation, simplifying the expression to `O(i^2)` for each `i`. Multiplying this by `n` for the for-loop, we get a total worst-case time complexity of `O(n^3)`.

Space Complexity:

The space complexity of the code is primarily determined by the maximum depth of the recursive call stack used by the `check` function.

The depth of the recursion is limited by the number of digits in `x` (the square of `i`), which is `O(log(x)) = O(log(i^2)) = O(2*log(i)) = O(log(i))`. Therefore, the space complexity is `O(log(n))`.

Note, the space used to store integers and for iteration is only `O(1)`, which does not contribute significantly compared to the recursive stack.