2787. Ways to Express an Integer as Sum of Powers

Medium   Dynamic Programming   Leetcode Link

Problem Description

The objective of this problem is to find the total number of unique ways we can represent a given positive integer `n` as a sum of the `xth` powers of distinct positive integers. In other words, for two positive numbers `n` and `x`, we need to figure out in how many different ways we can find sets of unique positive integers `[n1, n2, ..., nk]` such that when each number `ni` is raised to the power of `x` and then all of these `xth` powers are added together, the sum equals `n`. The final result must be returned modulo `10^9 + 7` to manage large numbers.

For a given number `n = 160` and `x = 3`, an example of such a representation is `160 = 2^3 + 3^3 + 5^3`. However, the problem asks for the count of *all* possible such representations, not just one.

Intuition

The intuition behind the solution is to use dynamic programming to keep track of the number of ways we can form different sums using `xth` powers of numbers up to `i`. We define a 2-dimensional array `f` where `f[i][j]` will store the number of ways we can form the sum `j` using the `xth` power of integers from 1 to `i`.

To populate this array, we'll start with the understanding that there's exactly one way to achieve a sum of 0 (by using no numbers), so `f[0][0] = 1`. Then, we iterate over each number from 1 to `n`, calculating its `xth` power (`k = pow(i, x)`) and updating the `f` array. For each `i`, we iterate through all possible sums `j` from 0 to `n`, and we do the following:

- We first set `f[i][j]` to `f[i-1][j]`, which represents the number of ways to get a sum `j` without using the `i`th number.
- If `k` (the `xth` power of `i`) is less than or equal to `j`, it means we can include `i` in a sum that totals `j`. Therefore, we add the number of ways to get the remaining sum (`j - k`) using integers up to `i - 1`. That is, `f[i][j] += f[i-1][j - k]`.
- Every time we update `f[i][j]`, we take the result modulo `10^9 + 7` to keep the number within bounds.

At the end of the iterations, `f[n][n]` will hold the total number of ways we can represent `n` using the `xth` powers of unique integers, and this is what we return.

Solution Approach

The solution uses dynamic programming, a method often used to solve problems that involve counting combinations or ways of doing something. To tackle this particular problem, we create a two-dimensional list `f` that serves as our DP table. This table has dimensions `(n+1) x (n+1)` where `n` is the input integer whose expressions we need to find.

The idea is to gradually build up the number of ways to reach a certain sum `j` using `xth` powers of numbers up to `i`. Let's take a closer look at how the code implements this:

- We start by initializing our DP table, `f`, with zeros and setting `f[0][0]` to 1, as there's only one way to have a sum of 0 (using none of the numbers).

```
f = [[0] * (n + 1) for _ in range(n + 1)]
f[0][0] = 1
```

- We then iterate through each number `i` from 1 to `n`, since we are considering sums of numbers raised to the power `x`. And for each number, we calculate its `xth` power and store it in `k`.

```
for i in range(1, n + 1):
    k = pow(i, x)
```

- For each `i`, we then go through all possible sums `j`. We set `f[i][j]` to the number of ways to form the same sum `j` without using `i`. This is obtained from `f[i - 1][j]`.

```
for j in range(n + 1):
    f[i][j] = f[i - 1][j]
```

- If the `xth` power of `i` (`k`) is less than or equal to `j`, we find the number of ways to make up the remaining value (`j - k`) with the previous numbers. We then add this count to our current count of ways for sum `j`.

```
if k <= j:
    f[i][j] = (f[i][j] + f[i - 1][j - k]) % mod
```

- The `% mod` ensures that the resulting count is within the bounds set by the problem statement.

By the end of these iterations, `f[n][n]` contains the desired count of ways to express `n` as a sum of the `xth` power of unique positive integers. Because we iterated over all numbers from 1 to `n` and for each considered all subsets of these numbers and ways to sum up to `j`, our final count is comprehensive.

Finally, the solution function returns this count:

```
return f[n][n]
```

By systematically adding the ways to form smaller sums and building up to the larger sums, the dynamic programming approach eliminates redundant calculations, thus optimizing the process of finding all possible combinations that sum up to `n` using powers of `x`.

Example Walkthrough

Let's take a small example to illustrate the solution approach. Consider `n = 10` and `x = 2`. We want to find out how many unique ways can we express `n` as a sum of squares of distinct positive integers.

1. Initialize the two-dimensional list, `f`, with zeros and set `f[0][0]` to 1. This represents the number of ways to represent `0` as the sum of squares of numbers from `0`. There is only one way, which is to use none of the numbers:

```
f = [
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    ... (up to f[10][0] which are all zeros)
]
```

2. We iterate through numbers `i` from 1 to `n` (1 to 10), and for each number, we compute its square (`k = i^2`).

   Example for `i = 1`: `k = 1^2 = 1`

3. With `k = 1` for `i = 1`, we iterate through sums `j` from 0 to `n`. If `k` (`1` in this case) is less than or equal to `j`, we add the count for `j-k` from the previous iterations. This represents including number `i` in our sum.

   Example updates for `i = 1` and `j = 1 to 10` (with modulo operation omitted for simplicity):

```
f[1][1] = f[0][1] + f[0][0] = 1 (only using 1^2)
f[1][2] = f[0][2] + f[0][1] = 0 (can't express 2 as only square of 1)
...
f[1][10] = f[0][10] + f[0][9] = 0 (can't express 10 as only square of 1)
```

4. We repeat step 3 for all `i` up to `n`. For instance, when `i = 2`, `k = 2^2 = 4`.

   Example updates for `i = 2`:

```
f[2][4] = f[1][4] + f[1][0] = 1 (using 2^2)
f[2][5] = f[1][5] + f[1][1] = 1 (using 1^2 and 2^2)
...
f[2][10] = f[1][10] + f[1][6] = 1 (using 1^2 and 3^2)
```

5. As we continue this process and update `f` for increasing `i` and `j`, we eventually build up the count. By the time we reach `f[10][10]`, it will have been updated with all possible ways to express `10` as a sum of squares of unique integers from 1 to 3, as 4^2 already exceeds 10, and higher powers won't be considered.

After evaluating `f[i][j]` with the above steps while including the necessary modulo operation, the answer for `f[10][10]` will give us the total number of ways we can represent `10` using the squares of unique integers. In this case, the unique ways are `{1^2, 3^2}` and `{1^2, 2^2, 3^2}` minus `{2^2, 3^2}` since 2^2 + 3^2 is greater than 10, giving us a final answer as 2.

Python Solution

```python
class Solution:
    def numberOfWays(self, total_sum: int, exponent: int) -> int:
        # Modulo constant to prevent overflow issues for large numbers
        MOD = 10 ** 9 + 7

        # Initializing a dynamic programming table where
        # dp[i][j] represents the number of ways to write j as a sum
        # of i-th powers of first i natural numbers.
        dp = [[0] * (total_sum + 1) for _ in range(total_sum + 1)]

        # There is one way to form the sum 0 using 0 numbers: use no numbers at all.
        dp[0][0] = 1

        # Iterate over all numbers from 1 to total_sum
        for i in range(1, total_sum + 1):
            # Calculate the i-th power of the number
            power = pow(i, exponent)
            # Iterate over all possible sums from 0 to total_sum
            for j in range(total_sum + 1):
                # The number of ways to form the sum j without using the i-th number
                dp[i][j] = dp[i - 1][j]
                # If the power is less than or equal to the sum j, then
                # add the number of ways to form the previous sum j-power
                if power <= j:
                    dp[i][j] = (dp[i][j] + dp[i - 1][j - power]) % MOD

        # Return the number of ways to write total_sum as a sum of
        # powers of the first total_sum natural numbers.
        return dp[total_sum][total_sum]

```

Java Solution

```java
class Solution {

    /**
     * This method calculates the number of ways to reach a target sum `n`
     * using unique powers `x` of the numbers from 1 to `n` inclusive.
     *
     * @param n The target sum we want to achieve.
     * @param x The power to which we will raise numbers.
     * @return The number of ways to achieve the target sum using the powers.
     */
    public int numberOfWays(int n, int x) {
        // Define the modulo constant to prevent overflow issues
        final int MOD = (int) 1e9 + 7;

        // Initialize a 2D array to store intermediate results
        // f[i][j] will store the number of ways to reach the sum `j`
        // using powers of numbers from 1 to `i`
        int[][] dp = new int[n + 1][n + 1];

        // There is exactly one way to reach the sum 0, which is by using no numbers
        dp[0][0] = 1;

        // Loop over every number up to `n` to compute the powers and the ways to reach each sum `j`
        for (int i = 1; i <= n; ++i) {
            // Calculate the power of the current number `i`
            long power = (long) Math.pow(i, x);

            // Loop over all sums from 0 to `n`
            for (int j = 0; j <= n; ++j) {
                // Initialize dp[i][j] with the number of ways to reach the sum `j` without using the current number
                dp[i][j] = dp[i - 1][j];

                // If adding the current power does not exceed the sum `j` (it's a valid choice to reach the sum `j`)
                if (power <= j) {
                    // Update the number of ways by adding the ways to reach the reduced sum `j - power`
                    // and take modulo to handle large numbers
                    dp[i][j] = (dp[i][j] + dp[i - 1][j - (int) power]) % MOD;
                }
            }
        }

        // Return the number of ways to reach the sum `n` using all numbers from 1 to `n` raised to the power `x`
        return dp[n][n];
    }
}

```

C++ Solution

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

class Solution {
public:
    int numberOfWays(int n, int x) {
        const int MOD = 1e9 + 7;                                        // Modular constant for large numbers
        std::vector<std::vector<int>> dp(n + 1, std::vector<int>(n + 1, 0)); // DP table to store intermediate results

        dp[0][0] = 1;                                                   // Base case: one way to sum up 0 using 0 elements
        for (int i = 1; i <= n; ++i) {                                  // Loop through numbers from 1 to n
            long long powerOfI = (long long)std::pow(i, x);             // Calculate the i-th power of x
            for (int j = 0; j <= n; ++j) {                              // Loop through all the possible sums from 0 to n
                dp[i][j] = dp[i - 1][j];                                // Without the current number i, ways are from previous
                if (powerOfI <= j) {
                    // If current power of i fits into sum j, add ways where i contributes to sum j
                    dp[i][j] = (dp[i][j] + dp[i - 1][j - powerOfI]) % MOD;
                }
            }
        }
        return dp[n][n]; // Return the number of ways to get sum n using numbers 1 to n to the power of x
    }
};

```

Typescript Solution

```typescript
function numberOfWays(n: number, x: number): number {
    // Modulo constant for the result as we only need the remainder
    // after dividing by this large prime.
    const MODULO = 10 ** 9 + 7;

    // Creating a 2D array 'ways' of dimensions (n+1)x(n+1) to store
    // the number of ways to write numbers as the sum of powers of x.
    const ways = Array.from({ length: n + 1 }, () =>
                  Array(n + 1).fill(0));

    // Base case: there is only one way to write 0 - as an empty sum.
    ways[0][0] = 1;

    // Iterate over each number from 1 to n.
    for (let i = 1; i <= n; ++i) {

        // Calculate the current power of i.
        const power = Math.pow(i, x);

        // Iterate over all the numbers from 0 to n to find the number
        // of ways to write each number j as a sum of powers of integers.
        for (let j = 0; j <= n; ++j) {

            // The number of ways to write j without using i.
            ways[i][j] = ways[i - 1][j];

            // If current power does not exceed j, we can use it in the sum.
            // We add the number of ways to write the remaining part
            // (j - power) with numbers up to (i - 1).
            if (power <= j) {
                ways[i][j] = (ways[i][j] + ways[i - 1][j - power]) % MODULO;
            }
        }
    }

    // The answer is the number of ways to write n
    // using powers of all numbers up to n.
    return ways[n][n];
}

```

Time and Space Complexity

The given Python code defines a function `numberOfWays` that calculates the number of ways to reach a total sum `n` by adding powers of integers up to `n`, each raised to the power of `x`. Below is an analysis of the code's time and space complexity:

Time Complexity

The algorithm consists of a nested loop where the outer loop runs `n` times, and the inner loop also runs `n` times. In every iteration of the inner loop, the algorithm performs constant-time operations, except for the `pow(i, x)` calculation, which is done once per outer loop iteration.

The `pow` function is typically implemented with a logarithmic time complexity concerning the exponent. However, since the exponent `x` is constant for the entire run of the algorithm, each call to `pow` will have a constant time complexity. Thus, we can consider the `pow(i, x)` part to have a time complexity of `O(1)` for each iteration of `i`.

Hence, the time complexity is primarily governed by the nested loop, resulting in a time complexity of `O(n^2)`.

Space Complexity

The code uses a 2-dimensional list `f` with dimensions `n + 1` by `n + 1`, which stores the computed values. This list is the primary consumer of memory in the algorithm.

The space complexity for the list `f` is `O(n^2)`, which reflects the amount of memory used with respect to the input size `n`.