2961. Double Modular Exponentiation

Medium Array Math Simulation

Problem Description

In this problem, you are given a two-dimensional array named `variables` where each element is a list of four integers `[a_i, b_i, c_i, m_i]`. In addition, you are provided with an integer `target`. An index `i` in the `variables` array is considered *good* if it satisfies a certain mathematical condition based on the values of `a_i`, `b_i`, `c_i`, and `m_i`. The condition is that the expression `((a_i^b_i % 10)^c_i) % m_i` must be equal to the `target`. Here the `^` represents exponentiation, and `%` is the modulo operation.

The task is to identify all the indices that are *good* and return them as an array in any order. An index is *good* if the above formula matches the target when evaluated with the values at that index in the variables array. You need to consider each index `i` from `0` to `variables.length - 1` and check the formula for each.

Intuition

The intuition behind the solution lies in understanding how modular arithmetic can be used to simplify calculations and the properties of exponents. Since the final expression involves taking `a_i` to the power of `b_i`, then to the power of `c_i`, and applying modulo `m_i` to check if it equals the `target`, we can take advantage of the following properties:

1. **Exponentiation by Squaring (Fast Power Method):** This method allows us to compute `a^b` efficiently by squaring and reducing the number of multiplications we need to perform. This is especially useful when `b` is very large.
2. **Modulo Properties:** We can use the property `(a * b) % m = ((a % m) * (b % m)) % m`, which states that the result of a product modulo `m` is the same regardless of whether we perform the modulo before or after multiplication. We can also apply this principle to exponentiation.

By applying these optimizations, we can effectively reduce the calculations needed to verify whether an index `i` satisfies the condition. Hence, the process for each index `i` is to apply the modulo operation as soon as possible during the exponentiation steps to keep the numbers manageable and prevent overflow. This is done by calculating `a^b % 10` and then `((a^b % 10)^c) % m_i`.

The solution applies the Python `pow` function, which takes an additional modulo argument, making these operations straightforward and efficient. By doing so, we loop through each `variables[i]` and apply the property `(a^b % 10)^c % m_i` and compare it to the `target`. If it matches, we add the index `i` to our output list. This approach is both clean and efficient, allowing us to handle even large powers effectively.

Solution Approach

The implementation of the solution uses a relatively simple approach aligned with the properties of modulo arithmetic and exponentiation. Here's how it works:

1. **Iteration:** We iterate through each index `i` of the input `variables` array using a for-loop.
2. **Exponentiation:** For every triple `(a, b, c, m)` at index `i`, we need to calculate the expression `((a^b % 10)^c) % m` and check if it equals the `target`.
3. **Exponentiation with Modulo:** The Python `pow` function is perfect for this task because it allows us to efficiently calculate powers with a modulo. The regular power operation `a^b` may lead to very large numbers, especially when `b` and `c` are large, which is computationally expensive and can cause integer overflows. However, by using the `pow` function with a modulo, we can calculate `a^b % 10` much more quickly and safely.
   - The first power operation is `pow(a, b, 10)` which computes `a^b % 10`. We modulo by `10` because exponentiation modulo `10` gives us the last digit of `a^b`, which is all we need for the next power.
   - The resulting number then needs to be exponentiated again with `c` and taken modulo `m`. This is continued with another `pow` operation - `pow(result, c, m)` where `result` is `a^b % 10`.
4. **Check Against Target:** The final result of the nested `pow` function is compared to the `target`. If they match, the index `i` is considered *good*.
5. **Building the Output List:** The indices where the condition holds true are gathered into a list using a list comprehension. This list is returned as the final output.

The process uses a direct simulation of the problem's formula. The fast power method (also known as exponentiation by squaring) is implicitly applied through Python's built-in `pow` function with mod argument, as this is a known efficient way to compute powers in modular arithmetic.

Here's a breakdown of the algorithm based on the solution code:

```
class Solution:
    def getGoodIndices(self, variables: List[List[int]], target: int) -> List[int]:
        return [
            i
            for i, (a, b, c, m) in enumerate(variables)
            if pow(pow(a, b, 10), c, m) == target
        ]
```

In the provided code, `enumerate(variables)` gives us both the index `i` and the list `[a, b, c, m]`. For each index and list, `if pow(pow(a, b, 10), c, m) == target` evaluates the condition. If the condition is true, the `i` is included in the output list. This is an elegant and effective way to solve the problem with minimal code and high readability.

Example Walkthrough

Let's work through a small example to illustrate the solution approach.

Assume we have the following `variables` array and the `target` value:

```
variables = [
    [2, 3, 1, 4],  # Index 0
    [3, 2, 2, 3],  # Index 1
    [2, 5, 2, 1],  # Index 2
]

target = 1
```

We need to find all indices where the condition `((a_i^b_i % 10)^c_i) % m_i == target` holds true.

Step 1: We start by iterating over each index in the `variables` array.

For index 0: The tuple is `[2, 3, 1, 4]`, representing `a=2`, `b=3`, `c=1`, `m=4`.

- We calculate `2^3 % 10`, which is `8 % 10` giving us `8`.
- Next, we calculate `8^1 % 4`, which is simply `8 % 4` giving us `0`.
- The result is `0`, which does not match the `target` of `1`. Hence, index `0` is not *good*.

For index 1: The tuple is `[3, 2, 2, 3]`, representing `a=3`, `b=2`, `c=2`, `m=3`.

- We calculate `3^2 % 10`, which is `9 % 10` giving us `9`.
- Next, we calculate `9^2 % 3`, which is `81 % 3` giving us `0`.
- The result is `0`, which does not match the `target` of `1`. Hence, index `1` is not *good*.

For index 2: The tuple is `[2, 5, 2, 1]`, representing `a=2`, `b=5`, `c=2`, `m=1`.

- We calculate `2^5 % 10`, which is `32 % 10` giving us `2`.
- Next, we calculate `2^2 % 1`, which is `4 % 1` giving us `0`.
- However, since any number modulo `1` is `0`, this does not match our `target` of `1`. Hence, index `2` is not *good*.

At the end of the iteration, unfortunately, no indices satisfy the condition for our small example, and therefore the resulting array of *good* indices would be empty - `[]`.

The solution code using this approach would be:

```
class Solution:
    def getGoodIndices(self, variables: List[List[int]], target: int) -> List[int]:
        return [
            i
            for i, (a, b, c, m) in enumerate(variables)
            if pow(pow(a, b, 10), c, m) == target
        ]
```

The code works efficiently through the properties of exponents and modulo arithmetic, using Python's `pow` function to simplify the computations and avoid overflow, while checking each index against the target.

Solution Implementation

Python

```
from typing import List

class Solution:
    def get_good_indices(self, variables: List[List[int]], target: int) -> List[int]:
        # Iterate over the list enumerating the variables
        # where 'i' is the index and 'variable' is a list [a, b, c, m]
        return [
            i
            for i, variable in enumerate(variables)
            # Check if the complex power expression matches the target
            # a raised to the power of b mod 10, raised to the power of c mod m should equal target
            if pow(pow(variable[0], variable[1], 10), variable[2], variable[3]) == target
        ]

# Example usage:
# solution = Solution()
# result = solution.get_good_indices([[2, 3, 1, 10], [3, 3, 2, 12]], 8)
# print(result) # Output: list of indices that match the criteria
```

Java

```
class Solution {
    // Method to find all the indices in 'variable' array where the recursive power operation result equals 'target'.
    public List<Integer> getGoodIndices(int[][] variables, int target) {
        List<Integer> goodIndices = new ArrayList<>(); // List to hold the indices that meet the criteria.
        // Iterate over each variable array.
        for (int i = 0; i < variables.length; ++i) {
            int[] variableSet = variables[i];
            // Extract individual variables from the current set.
            int a = variableSet[0];
            int b = variableSet[1];
            int c = variableSet[2];
            int m = variableSet[3];
            // If the recursive power operation result equals 'target', add the index to the list.
            if (quickPow(quickPow(a, b, 10), c, m) == target) {
                goodIndices.add(i);
            }
        }
        return goodIndices; // Return the list of indices.
    }

    // Method to compute (a^b) % mod quickly using binary exponentiation.
    private int quickPow(long a, int b, int mod) {
        long result = 1; // Initialize result to 1.
        // Loop until b is zero.
        for (; b > 0; b >>= 1) {
            // If the current bit in b is set, multiply result by 'a' and take mod.
            if ((b & 1) == 1) {
                result = (result * a) % mod;
            }
            // Square 'a' and take mod to use for the next iteration.
            a = (a * a) % mod;
        }
        return (int) result; // Cast the long result back to int and return.
    }
}
```

C++

```
#include <vector>

class Solution {
public:
    // Function to find all the indices of variables that meet a certain condition.
    std::vector<int> getGoodIndices(std::vector<std::vector<int>>& variables, int target) {
        std::vector<int> goodIndices; // This will store the indices that meet the condition

        // Lambda function for fast exponentiation under modulo (to avoid integer overflow)
        auto fastPower = [](long base, int exponent, int modulo) -> int {
            long long result = 1; // Start with a result of 1

            // Continuously square base and multiply by base when the exponent's current bit is 1
            while (exponent > 0) {
                if (exponent & 1) { // Check if the current bit is 1
                    result = (result * base) % modulo; // Multiply by base and take modulo
                }
                base = (base * base) % modulo; // Square the base and take modulo
                exponent >>= 1; // Shift exponent right by 1 bit (divide by 2)
            }

            return static_cast<int>(result); // Cast result to int before returning
        };

        // Iterate over all the variable sets
        for (int i = 0; i < variables.size(); ++i) {
            // Unpacking variables for readability // Reference for better performance and readability
            int a = variableSet[0]; // Base part of the power
            int b = variableSet[1]; // Exponent part
            int c = variableSet[2]; // Exponent for the result of a^b
            int modulo = variableSet[3]; // The modulo value

            // Apply the fast power modulo function twice as specified by the problem
            // and compare with the target value
            if (fastPower(fastPower(a, b, 10), c, modulo) == target) {
                goodIndices.push_back(i); // If condition is met, add index to the list
            }
        }

        return goodIndices; // Return the list of good indices
    }
};
```

TypeScript

```
// Function to perform quick exponentiation by squaring, module a given modulus 'mod'.
const quickPowMod = (base: number, exponent: number, modulus: number): number => {
    let answer = 1;
    while(exponent > 0) {
        if(exponent & 1) {
            answer = Number((BigInt(answer) * BigInt(base)) % BigInt(modulus));
        }
        base = Number((BigInt(base) * BigInt(base)) % BigInt(modulus));
        exponent >>= 1;
    }
    return answer;
};

// Function to find and return the indices of the 'variables' array where the double exponentiation
// result matches the target after applying the modulus 'm' in each inner calculation.
function getGoodIndices(variables: number[][], target: number): number[] {
    const resultIndices: number[] = [];

    for(let index = 0; index < variables.length; ++index) {
        // Destructuring each 'variables' element into separate constants for clarity.
        const [base, firstExponent, secondExponent, modulus] = variables[index];

        // Compute the nested power-mod operation: ((a^b)^c) % m.
        const doubleExponentiationResult = quickPowerMod(
            quickPowerMod(base, firstExponent, 10),
            secondExponent,
            modulus
        );

        // If the result matches the target, add the current index to the 'resultIndices' array.
        if(doubleExponentiationResult === target) {
            resultIndices.push(index);
        }
    }

    // Return the array with the good indices.
    return resultIndices;
}
```

```
from typing import List

class Solution:
    def get_good_indices(self, variables: List[List[int]], target: int) -> List[int]:
        # Iterate over the list enumerating the variables
        # where 'i' is the index and 'variable' is a list [a, b, c, m]
        return [
            i
            for i, variable in enumerate(variables)
            # Check if the complex power expression matches the target
            # a raised to the power of b mod 10, raised to the power of c mod m should equal target
            if pow(pow(variable[0], variable[1], 10), variable[2], variable[3]) == target
        ]

# Example usage:
# solution = Solution()
# result = solution.get_good_indices([[2, 3, 1, 10], [3, 3, 2, 12]], 8)
# print(result) # Output: list of indices that match the criteria
```

Time and Space Complexity

The given Python code in the `Solution` class contains a method called `getGoodIndices` which computes good indices based on the power operation condition.

The time complexity of this code is primarily dependent on the `enumerate` function that iterates through the `variables` list and the `pow` function used inside the list comprehension. The `pow` function in Python has a time complexity of `O(log n)`, where `n` is the exponent.

Here, the `pow` function is used twice; the first call executes `pow(a, b, 10)`, and the second call executes `pow(result_of_first_call, c, m)`. Since `b` and `c` are the exponents used in each call, the time complexity for each iteration is controlled by these values.

Considering `M` as the maximum value among all values of `b_i` and `c_i`, and with `M` being limited to `10^3` in the problem, the time complexity per iteration is `O(log M)`. Since there are `n` elements in `variables`, where `n` is the length of `variables`, the total time complexity is `O(n * log M)`.

The space complexity of the code is `O(1)` because the extra space required is constant and does not depend on the input size. There are no additional data structures that grow with the size of `variables`. The list comprehension simply produces the final list of indices, which the function returns and does not count towards extra space as it is the required output.