372. Super Pow

Medium Math Divide and Conquer Leetcode Link

Problem Description

The problem requires us to calculate `a^b mod 1337`, where `a` is a positive integer and `b` is an extremely large positive integer presented as an array. The notation `a^b` refers to `a` raised to the power `b`. The array represents the number `b` where each element is a digit of `b` in its base-10 (decimal) representation. The least significant digit is at the end of the array.

Intuition

To deal with the extremely large exponent `b` efficiently, we need to apply modular exponentiation. Calculating `a^b` directly, even before applying the modulus, is not feasible because `b` can be very large. Instead, we can repeatedly use the property `(x * y) mod z = [(x mod z) * (y mod z)] mod z` to keep the intermediary results small.

Also, we can apply the right-to-left binary method for exponentiation, which goes through each digit of the exponent, in our case each element in array `b`. For each digit `d` in `b` from right to left, we compute `(a^d) mod 1337` and multiply it with the running `ans`. After processing a digit `d`, we must update `a` to `a^10 mod 1337` to account for the positional value of `d` in `b`. This is because each step to the left in the array represents a decimal place increase by a factor of 10, effectively multiplying our base `a` by `10^d`.

By starting from the least significant digit of `b` (the end of the array) and working our way back to the most significant digit, each iteration computes the result for one decimal place, while updating `a` to its new base raised to the power of 10.

The reason we perform operations under modulo 1337 at every step is to prevent overflow and ensure that all intermediate multiplication results are manageable for large inputs.

Solution Approach

The solution uses a basic loop and modular arithmetic to solve the problem of computing `(a^b) mod 1337`. Let's go through the steps:

1. Initialize `ans` to 1 because anything raised to the power 0 is 1, and it will be our cumulative product for the result.
2. Iterate over the digits of `b` in reverse order, i.e., from the least significant digit to the most significant digit.
3. In each iteration of the loop, we update `ans` using the formula `ans = ans * (a^e) mod 1337`. Here `e` is the current digit of `b`. The expression `(a^e) mod 1337` computes the contribution of the current digit to the overall exponentiation.
4. After processing each digit `e`, we need to update `a` to account for shifts in decimal place. We use the formula `a = (a^10) mod 1337`.
5. After the loop ends, `ans` holds the final value of `(a^b) mod 1337`.

In this approach, `pow(x, y, z)` is a powerful built-in Python function that calculates `(x^y) mod z` efficiently (*modular exponentiation*), even for large values of `y`.

By iteratively handling each digit of `b` and applying modular arithmetic at every step, we maintain manageable numbers throughout the computation process. This is necessary because working with significantly large numbers directly would be impractical due to computational and memory constraints.

The pattern used here is that of repeated squaring, which is a standard technique in modular exponentiation algorithms where the base is squared (or here raised to the power of 10) for each successive digit. This is premised on the mathematical insight that squaring the base adjusts for the exponential increase as you move through each digit of the exponent.

This also preserves the order of computation represented by the array `b`, properly accounting for each digit's positional value, ending with a correct answer modulo 1337.

Example Walkthrough

Let's illustrate the solution approach with a small example.

Assume `a = 2`, and `b` is represented by the array `[1, 0, 2]`, which corresponds to the integer `201`.

The goal is to compute `(2^201) mod 1337`.

Following the solution steps:

1. Initialize `ans` to 1. Initially, `ans = 1`.
2. The array `[1, 0, 2]` represents the number `201`. We iterate over it from the end (right-to-left):
   - Starting with the least significant digit, which is `2` in this case, we compute `(2^2) mod 1337`. This is `4`.
   - Update `ans` with this value by multiplying: `ans = 1 * 4 mod 1337`, which simplifies to `ans = 4`.
3. Now, before moving to the next digit, we must update `a` to `a^10 mod 1337` to account for the positional value:
   - Update `a` with `(2^10) mod 1337`, which is equal to `1024` mod `1337`. This is `1024`.
   - The new value of `a` is now `1024`.
4. Moving to the second digit from the end, which is `0`:
   - Compute `(1024^0) mod 1337`, which is `1`.
   - Update `ans = 4 * 1 mod 1337`, so `ans` remains `4`.
5. Again, we must update `a` to the next power of ten:
   - Update `a` with `(1024^10) mod 1337`. Here we use the power function `pow(1024, 10, 1337)`. Let's assume this calculation gives us the value `X` for simplicity.
   - Now `a` becomes `X`.
6. Finally, moving to the most significant digit, which is `1`:
   - Compute `(X^1) mod 1337`, which just yields `X`.
   - Update `ans = 4 * X mod 1337`.
7. After processing all digits, the value stored in `ans` is `(2^201) mod 1337`.

This example walked through each iteration, updating `ans` with the contribution of each digit in the array `b`, and recalculating the base `a` as `a^10 mod 1337` after each step.

Keep in mind that for the actual computing of the updated value of `a` as `a^10 mod 1337`, especially for large values of `a`, the `pow` function is used due to its efficient computation of large exponents under a modulus.

Python Solution

```python
class Solution:
    def superPow(self, base: int, exponent_digits: List[int]) -> int:
        # Define the modulus to ensure the result is within a reasonable range
        mod = 1337

        # Initialize the answer to 1, as we'll be multiplying it iteratively
        result = 1

        # Reverse the list of digits to process the exponent starting from the least significant digit
        for digit in reversed(exponent_digits):
            # Multiply the current result by base to the power of the current digit,
            # then take the modulus to keep it within the defined range.
            result = (result * pow(base, digit, mod)) % mod

            # Update the base to be raised to the next power of 10, again within the modulus range.
            # This accounts for shifting one decimal place to the left in the exponent.
            base = pow(base, 10, mod)

        # Return the final result after processing all digits in the exponent
        return result

```

Java Solution

```java
class Solution {
    private final int MOD = 1337; // Using all-uppercase for the constant value to follow Java naming conventions

    /**
     * Calculates a^b mod 1337 where a is an integer and b is represented as an array of digits.
     *
     * @param a The base number.
     * @param b The exponent represented as an array of digits.
     * @return The result of a^b mod 1337.
     */
    public int superPow(int a, int[] b) {
        long result = 1; // Initialize result to 1.

        // Loop through the array from the last element to the first element.
        for (int i = b.length - 1; i >= 0; --i) {
            // Multiply the result with the fast power of a raised to the current digit, then take mod.
            result = result * fastPower(a, b[i]) % MOD;
            // Update a to a^10 to get the next power level for the next digit.
            a = fastPower(a, 10);
        }
        return (int) result; // Cast result back to int before returning.
    }

    /**
     * Fast power algorithm calculates a^n mod 1337.
     *
     * @param a The base number.
     * @param n The exponent.
     * @return The result of a^n mod 1337.
     */
    private long fastPower(long a, int n) {
        long ans = 1; // Result of the exponentiation.

        // Loop through the bits of n until n is 0.
        for (; n > 0; n >>= 1) {
            // Whenever the current bit is 1, multiply ans with a and take mod.
            if ((n & 1) == 1) {
                ans = ans * a % MOD;
            }
            // Square a and take mod for the next iteration (or next bit).
            a = a * a % MOD;
        }
        return ans; // Return the power result.
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int superPow(int base, vector<int>& digits) {
        // Define the modulus constant as required by the problem statement.
        const int MOD = 1337;

        // Result of the super power operation, initialized to 1.
        long long result = 1;

        // Define a lambda function for fast exponentiation under modulus.
        // It calculates (base^exponent) % MOD using the binary exponentiation method.
        auto quickPow = [&](long long base, int exponent) -> int {
            long long res = 1; // Initialize the result of exponentiation to 1.
            while (exponent > 0) {
                if (exponent & 1) {
                    // If the current exponent bit is set, multiply the result by base.
                    res = (res * base) % MOD;
                }
                // Square the base and reduce it modulo MOD for the next bit of exponent.
                base = (base * base) % MOD;
                // Right-shift exponent by one bit to process the next bit.
                exponent >>= 1;
            }
            // Cast the long long result to int before returning.
            return static_cast<int>(res);
        };

        // Process the array of digits in reverse order to calculate the super power.
        for (int i = digits.size() - 1; i >= 0; --i) {
            // Multiply the result by (base^(current digit)) % MOD.
            result = (result * quickPow(base, digits[i])) % MOD;
            // Update base to base^10 for the next iteration (digit place).
            base = quickPow(base, 10);
        }

        // Return the final result as an integer.
        return static_cast<int>(result);
    }
};

```

Typescript Solution

```typescript
function superPow(base: number, digits: number[]): number {
    let result = 1; // Initialize result variable
    const modulo = 1337; // Define the modulo value for calculations to avoid large numbers

    // Helper function for quick exponentiation under modulus
    const quickPow = (base: number, exponent: number): number => {
        let localResult = 1; // Initialize local result for this function
        while (exponent > 0) { // Loop until exponent is zero
            if (exponent & 1) { // If the current bit is set
                localResult = Number((BigInt(localResult) * BigInt(base)) % BigInt(modulo));
            }
            base = Number((BigInt(base) * BigInt(base)) % BigInt(modulo)); // Square the base
            exponent >>= 1; // Right shift exponent by 1 (divide by 2 and take floor)
        }
        return localResult; // Return the calculated power
    };

    // Start from the least significant digit of the array
    for (let i = digits.length - 1; i >= 0; --i) {
        result = Number((BigInt(result) * BigInt(quickPow(base, digits[i]))) % BigInt(modulo));
        base = quickPow(base, 10); // Elevate the base to the 10th power for the next iteration
    }

    return result; // Return final result
}

```

Time and Space Complexity

Time Complexity

The time complexity of the algorithm is determined by the for-loop which iterates through the list `b`. Since the length of `b` dictates the number of iterations, let's call the length of `b` be `n`.

During each iteration, there are two main operations.

1. `ans = ans * pow(a, e, mod) % mod`: Computational complexity for `pow(a, e, mod)` function is generally `O(log(e))` for each iteration. However, since `e` is a single digit (0-9) as per the problem statement of `superPow` on LeetCode, the time complexity for `pow(a, e, mod)` becomes `O(1)` for each iteration.

2. `a = pow(a, 10, mod)`: Here the base `a` is raised to the power of 10. Similar to the previous point, this would normally be `O(log(10))`, but since 10 is a constant, this operation is also `O(1)`.

The loop runs `n` times, and each iteration performs a constant number of `O(1)` operations. Therefore, the overall time complexity is `O(n)`.

Space Complexity

The space complexity of the solution is determined by the extra space used in addition to the input.

1. The variable `ans` and other temporary variables use a constant amount of space, contributing to an `O(1)` space complexity.
2. The input list `b` is not duplicated, and reversing the list using `b[::-1]` does not create a new list, it creates an iterator, which is more space-efficient.
3. There are no recursive calls or additional data structures that would use extra space.

Therefore, the space complexity of the algorithm is `O(1)`, as it uses a constant amount of extra space.