Documentation

The problem asks to calculate a^b 1337, where a is a positive integer and b is a huge integer represented as an array of digits. The challenge here is that b can have up to 2000 digits, which makes directly computing a^b infeasible due to the potential size of the resulting number. Instead, the solution uses modular arithmetic to compute the result efficiently. The core idea is to utilize **modular exponentiation**, which allows calculating large powers modulo a number without ever having to compute the large numbers themselves.

To achieve this, the solution defines a helper function that performs **exponentiation by squaring**. This method computes $x^n \bmod m$ in logarithmic time, which is crucial when dealing with large exponents. By repeatedly squaring the base x and halving the exponent, the function ensures that even large powers can be computed quickly. The main function processes the array b digit by digit, updating the result incrementally by first raising the current result to the power of 10 (due to the structure of the number b) and then multiplying it by the value of a^b[i]mod 1337, where b[i] is each digit of b.

Since b is provided as an array of digits, it ensures that we don't directly deal with the massive value of b. Instead, we break down the exponentiation process into smaller and manageable parts. For each digit in the array, the intermediate results are kept within the modulus 1337, thus preventing overflow and unecessary computations. This method also benefits from the fact that exponentiation by squaring reduces the time complexity significantly, allowing the solution to handle even the largest cases efficiently.

The time complexity of this approach is O(klogn), where k is the number of digits in b and n is the maximum possible value for any digit in b, which is 9. Since the length of b can be up to 2000 digits, the algorithm remains feasible even for the upper limit. Each operation involving a digit of b takes logarithmic time in terms of the current exponent, ensuring that even with large exponents, the solution runs efficiently.

Space complexity for this approach is O(1), since only a few variables are used to store the intermediate results and no extra space is needed for large data structures. This makes the solution not only time-efficient but also space-efficient. The use of modular arithmetic and exponentiation by squaring ensures that the solution can handle extremely large numbers while staying within manageable bounds.

This problem exemplifies the power of modular arithmetic and efficient algorithms like exponentiation by squaring when dealing with large numbers. By breaking down the problem into smaller, manageable parts and using efficient techniques, the solution can handle even the largest inputs without runing into performance issues. In conclusion, the approach leverages both mathematical principles and algorithmic optimizations to solve a problem that would otherwise be computationally impractical.