479. Largest Palindrome Product

Math Hard

Problem Description

palindromic number is one that remains the same when its digits are reversed, such as 12321 or 45654. Since the resulting palindromic number could be quite large, the problem requires the final solution to be presented modulo 1337. This means you should take the remainder after dividing the palindromic number by 1337 before returning it. When n = 1, the largest palindromic number that can be formed is simply 9 (which is the product of two 1-digit numbers: 3 * 3), so the function returns 9. Intuition

Given an integer n, the task is to find the largest palindromic integer that can be obtained by multiplying two n-digit numbers. A

start multiplying from the largest n-digit numbers. The algorithm constructs the palindromic number by appending the reverse of the left half to itself. This ensures that the number is palindromic. We initiate b with the value of a and x as a, then extend x to be the full palindromic number by repeatedly

To find the largest palindromic integer product of two n-digit numbers, we start from the maximum n-digit number possible, mx =

10ⁿ - 1, and decrement from there. This is because we are interested in the largest possible product, which means we should

multiplying x by 10 (to shift digits to the left) and then adding the last digit of b to x after which b is divided by 10 (effectively dropping the last digit of b).

After constructing the palindromic number x, we try to find a divisor t that is a maximum n-digit number such that x % t == 0. If such a t exists, then x is a product of two n-digit numbers and we return x % 1337 as the result. We only need to check divisors down to the square root of x because if x is divisible by some number greater than its square root, the corresponding factor

would have to be less than the square root of x, which we would already have checked.

decreases the value of a, hence trying with a new, smaller n-digit number.

makes use of simple arithmetic operations and loops to achieve the outcome.

with the value of mx (t = 99), and check if x is divisible by t while t * t >= x.

So, we check if 9999 % 99 == 0. This is not the case, so we decrement t by 1 and check again.

that can be obtained by multiplying two 2-digit numbers.

99), and x to a (so x is also 99 initially).

descending n-digit numbers until it finds a palindromic product. If n is 1, the loop doesn't need to run since the only 1-digit palindromic product is 9, so in this case the algorithm returns 9. **Solution Approach**

The implementation uses a single for loop to iterate over all possible n-digit numbers, starting from the largest $(mx = 10^n - 1)$

and decrementing to the lowest possible n-digit number, which is 10^(n-1). Since we're interested in the largest palindromic

If a palindromic number cannot be found with the current a, the algorithm decrements a and tries again, continuing to loop over

product, we do not need to check any numbers less than mx // 10. For each iteration:

We initialize b to the value of a and x to a as well. Then, we construct a palindromic number x by reversing the digits of a and appending them to the original a. This is done within a while loop — each time we take the last digit of b using b % 10,

append it to x by multiplying x by 10 and adding this last digit, and remove the last digit from b using integer division b //= 10. We then initialize another temporary variable t, also with the value of mx, and perform another while loop with the condition t

* t >= x. This allows us to check if x is divisible (x % t == 0) by any n-digit number t. If it is, that means x is a palindromic product of two n-digit numbers and we return x % 1337. The condition t * t >= x works because if no factor is found before t reaches √x, then x cannot have two n-digit factors — it would require one factor to be smaller than n digits.

The loop decrements t after every iteration. If no divisor is found for the current palindromic number x, the outer loop

- If there's no palindromic number found for n > 1, the algorithm will naturally exit the for loop. This does not occur for any n > 11 as per the problem's constraints. However, if n = 1, then the function directly returns 9, as this is the only single-digit palindromic number product (since n-digit numbers in this case are between 1 and 9). The key to this solution is the efficient construction of palindromic numbers and the expedited search for divisors starting from
- **Example Walkthrough** Let's illustrate the solution approach with an example where n = 2. This means we are looking for the largest palindromic number

No additional data structures are used apart from a few variables to store intermediate values and the result. The implementation

The maximum 2-digit number is 99 ($mx = 10^n - 1$, with n being 2 in this case, so $10^2 - 1 = 100 - 1 = 99$). We start with a = mx, which is 99 in this example, and try to construct a palindromic number using a. We initialize b to a (b =

To create the palindromic number, we need to append the reverse of b to x. Since b = 99, we take the last digit of b by b % 10,

which is 9, multiply x by 10 to make space for the new digit, and add 9 to x. After this, b is divided by 10 to drop the last digit

% 1337.

(since 9009 % 99 == 0).

Solution Implementation

equals 123.

Python

class Solution:

the largest possible n-digit number.

(b becomes 9). We repeat this until b becomes 0, finishing with x as the palindromic number x = 9999. Now, we want to see if this palindromic number 9999 is a product of two 2-digit numbers. We initialize a temporary variable t

However, for this example, we will eventually find that no such t exists for x = 9999. Therefore, we decrement a and start over. a becomes 98, and we repeat the above steps to construct a new palindromic number x = 9889 and search again for a divisor t.

Eventually, we'll construct the palindromic number x = 9009 (from a = 91), and we'll find out that it is divisible by t = 99

Once the valid t is found, we take x % 1337 to find the result modulo 1337. For x = 9009, the result is 9009 % 1337, which

We continue this process, testing each t (98, 97, 96, ..., until we find x % t == 0). If we find such a t, we would then return x

- In this example walkthrough, the function would return 123 as the largest palindromic number that can be obtained by multiplying two 2-digit numbers modulo 1337.
- def largestPalindrome(self, n: int) -> int: # Define the maximum value for the given digit count. $max_num = 10**n - 1$ # Loop backwards from the maximum number to find the largest palindrome.

Stop at a reduced max_num (maximum divided by 10), to not check smaller sizes.

reversed_half //= 10 # Remove last digit from reversed_half.

This is the largest single-digit palindrome and is a special case for n = 1.

factor -= 1 # Reduce the factor and keep checking.

If no palindrome product of two n-digit numbers is found, return 9

Create the palindrome by reflecting the first half to create the second half.

palindrome = palindrome * 10 + reversed_half % 10 # Append reversed digit.

for first_half in range(max_num, max_num // 10, -1):

second_half = reversed_half = first_half

return palindrome % 1337

palindrome = first_half

while reversed_half:

factor = max_num

Check if the palindrome can be divided exactly by a number in the range. while factor * factor >= palindrome: if palindrome % factor == 0: # If so, return the palindrome modulo 1337 as per problem constraints.

return 9

Java

```
class Solution {
   // This method finds the largest palindrome that is a product of two n-digit numbers
    public int largestPalindrome(int n) {
       // The largest possible n-digit number
       int maxDigitValue = (int) Math.pow(10, n) - 1;
       // Looping from the largest n-digit number down to the smallest n-digit number
        for (int firstFactor = maxDigitValue; firstFactor > maxDigitValue / 10; --firstFactor) {
           // The second factor starts off identical to the first factor
           int secondHalf = firstFactor;
           // Beginning to construct the palindrome by considering the first factor
            long possiblePalindrome = firstFactor;
           // Mirroring the digits of the first factor to compose the second half of the palindrome
           while (secondHalf != 0) {
                possiblePalindrome = possiblePalindrome * 10 + secondHalf % 10;
               secondHalf /= 10;
           // Trying to find if the constructed palindrome has factors with n digits
            for (long potentialFactor = maxDigitValue; potentialFactor * potentialFactor >= possiblePalindrome; --potentialFactor
               // If the possiblePalindrome is divisible by potentialFactor, it is a palindrome that is a product of two n-digit
               if (possiblePalindrome % potentialFactor == 0) {
                    // Return the palindrome modulo 1337 as per the problem's requirement
                    return (int) (possiblePalindrome % 1337);
       // If no palindrome can be found that is a product of two n-digit numbers, return 9
       // This happens in the case where n = 1
       return 9;
C++
```

```
function largestPalindrome(n: number): number {
 // Calculate the maximum number based on the digit count n
 const maxNumber: number = Math.pow(10, n) - 1;
```

TypeScript

};

return 9;

class Solution {

int largestPalindrome(int n) {

while (reverse) {

int maxNumber = pow(10, n) - 1;

// Calculate the maximum number based on the digit count n

for (int number = maxNumber; number > maxNumber / 10; --number) {

reverse /= 10; // Remove the last digit from reverse

// Start from the maximum number and try to find a divisor

// Check if palindrome is divisible by testDivisor

// Return the palindrome modulo 1337

long palindrome = number; // This will be used to form the palindrome

// Create the palindrome by appending the reverse of "number" to itself

int reverse = number; // We will reverse "number" to create the palindrome

palindrome = palindrome * 10 + reverse % 10; // Add the last digit of reverse to palindrome

for (long testDivisor = maxNumber; testDivisor * testDivisor >= palindrome; --testDivisor) {

// Start from the maximum number and iterate downwards

if (palindrome % testDivisor == 0) {

// This is a special case for one-digit palindromes,

// which is the largest palindrome with one digit (9)

// since the loop iterations start for n > 1.

// Start from the maximum number and iterate downwards

for (let number = maxNumber; number > maxNumber / 10; number--) {

let palindrome: number = number; // The current half of the palindrome

let reverse: number = number; // Will reverse the number to create the palindrome

return palindrome % 1337;

public:

```
// Create the palindrome by appending the reverse of "number" to itself
      while (reverse > 0) {
        palindrome = palindrome * 10 + reverse % 10; // Add the last digit of reverse to the palindrome
        reverse = Math.floor(reverse / 10); // Remove the last digit from reverse
      // Try to find a divisor starting from the maximum number
      for (let testDivisor = maxNumber; testDivisor * testDivisor >= palindrome; testDivisor--) {
        // Check if palindrome is divisible by testDivisor
        if (palindrome % testDivisor === 0) {
         // Return the palindrome modulo 1337
          return palindrome % 1337;
    // This is a special case for one-digit palindromes,
    // which is the largest palindrome with one digit (9)
    // since the loop iterations start for n > 1.
    return 9;
class Solution:
   def largestPalindrome(self, n: int) -> int:
       # Define the maximum value for the given digit count.
       max_num = 10**n - 1
       # Loop backwards from the maximum number to find the largest palindrome.
       # Stop at a reduced max_num (maximum divided by 10), to not check smaller sizes.
        for first_half in range(max_num, max_num // 10, -1):
           # Create the palindrome by reflecting the first half to create the second half.
           second half = reversed half = first half
           palindrome = first_half
           while reversed half:
               palindrome = palindrome * 10 + reversed_half % 10 # Append reversed digit.
               reversed_half //= 10 # Remove last digit from reversed_half.
           factor = max_num
           # Check if the palindrome can be divided exactly by a number in the range.
           while factor * factor >= palindrome:
               if palindrome % factor == 0:
                   # If so, return the palindrome modulo 1337 as per problem constraints.
                   return palindrome % 1337
               factor -= 1 # Reduce the factor and keep checking.
       # If no palindrome product of two n-digit numbers is found, return 9
       # This is the largest single-digit palindrome and is a special case for n = 1.
       return 9
Time and Space Complexity
```

Time Complexity: The time complexity of the code is determined by the nested loops and the operations within those loops.

numbers.

decrements until it reaches mx / 10. This gives us the loop running in $0(10^{(n-1)})$. Inside the first loop, we create a palindromic number x based on the current value of a. The inner while loop used to create x

- iterates once for each digit in a, which is n times. Therefore, the palindromic number creation runs in O(n) time. The second while loop iterates at most t times, where t starts at mx and is decremented until t * t >= x. Since x can be as
 - large as mx^2, and we are decrementing t one by one, the upper bound for the number of iterations is mx, leading to a worstcase time complexity of O(mx) for this loop, which is $O(10^n)$.

The first loop runs for approximately mx / 10 times, which is $10^n / 10 = 10^{(n-1)}$ iterations, as it starts from mx and

The given Python code is designed to find the largest palindromic number that can be produced from the product of two n-digit

The time complexity is exponential, primarily due to the two nested loops where the outer loop is proportional to 10^(n-1) and the inner loop could potentially iterate up to 10ⁿ.

The nested loops result in a combined time complexity of $0(10^{(n-1)} * n * 10^n) = 0(n * 10^{(2n-1)})$.

Space Complexity:

The space complexity of the code is determined by the variables used and any additional space allocated during the execution of the algorithm. Variables a, b, x, t, and mx use a constant amount of space.

- The creation of the palindromic number does not use any additional data structures that grow with the input size.
 - Therefore, the space complexity of the code is 0(1), indicating constant space usage regardless of the input size n.