479. Largest Palindrome Product

Hard Math Leetcode Link

Problem Description

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

To find the largest palindromic integer product of two `n`-digit numbers, we start from the maximum `n`-digit number possible, `mx = 10^n - 1`, and decrement from there. This is because we are interested in the largest possible product, which means we should 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 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.

If a palindromic number cannot be found with the current `a`, the algorithm decrements `a` and tries again, continuing to loop over 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 product, we do not need to check any numbers less than `mx // 10`.

For each iteration:

1. 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`.
2. 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.
3. The loop decrements `t` after every iteration. If no divisor is found for the current palindromic number `x`, the outer loop decreases the value of `a`, hence trying with a new, smaller `n`-digit number.
4. 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 > 1` 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 the largest possible `n`-digit number.

No additional data structures are used apart from a few variables to store intermediate values and the result. The implementation makes use of simple arithmetic operations and loops to achieve the outcome.

Example Walkthrough

Let's illustrate the solution approach with an example where `n = 2`. This means we are looking for the largest palindromic number that can be obtained by multiplying two 2-digit numbers.

1. 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`).
2. 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 = 99`), and `x` to `a` (so `x` is also `99` initially).
3. 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 (`b` becomes 9). We repeat this until `b` becomes 0, finishing with `x` as the palindromic number `x = 9999`.
4. Now, we want to see if this palindromic number `9999` is a product of two 2-digit numbers. We initialize a temporary variable `t` with the value of `mx` (`t = 99`), and check if `x` is divisible by `t` while `t * t >= x`.
5. So, we check if `9999 % 99 == 0`. This is not the case, so we decrement `t` by 1 and check again.
6. 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 % 1337`.
7. 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`.
8. Eventually, we'll construct the palindromic number `x = 9009` (from `a = 91`), and we'll find out that it is divisible by `t = 99` (since `9009 % 99 == 0`).
9. 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 equals `123`.

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`.

Python Solution

```python
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
```

Java Solution

```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 num
                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++ Solution

```cpp
class Solution {
public:
    int largestPalindrome(int n) {
        // Calculate the maximum number based on the digit count n
        int maxNumber = pow(10, n) - 1;

        // Start from the maximum number and iterate downwards
        for (int number = maxNumber; number > maxNumber / 10; --number) {
            long palindrome = number; // This will be used to form the palindrome
            int reverse = number;     // We will reverse "number" to create the palindrome

            // Create the palindrome by appending the reverse of "number" to itself
            while (reverse) {
                palindrome = palindrome * 10 + reverse % 10; // Add the last digit of reverse to palindrome
                reverse /= 10;  // Reverse the last digit from reverse
            }

            // Start from the maximum number and try to find a divisor
            for (long 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;
    }
};
```

Typescript Solution

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

    // 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;
}
```

Time and Space Complexity

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

Time Complexity:

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

1. The first loop runs for approximately `mx / 10` times, which is `10^n / 10 = 10^(n-1)` iterations, as it starts from `mx` and decrements until it reaches `mx / 10`. This gives us the loop running in `O(10^(n-1))`.
2. 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.
3. 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 worst-case time complexity of `O(mx)` for this loop, which is `O(10^n)`.
4. The nested loops result in a combined time complexity of `O(10^(n-1) * n * 10^n) = O(n * 10^(2n-1))`.

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 `10^n`.

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.

1. Variables `a`, `b`, `x`, `t`, and `mx` use a constant amount of space.
2. 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 `O(1)`, indicating constant space usage regardless of the input size `n`.