866. Prime Palindrome

Medium Math

Problem Description

The problem requires us to find the smallest prime palindrome greater than or equal to a given integer `n`. A prime palindrome is an integer that is both a prime number and a palindrome. A prime number is one that has exactly two distinct positive divisors: 1 and itself, whereas a palindrome is a number that reads the same backward as forward. For instance, `7`, `3`, `5`, `7`, `11` are prime numbers, and `101`, `131`, `151` are examples of prime palindromes.

Intuition

The brute force method is to search for prime palindromes starting at `n` and proceeding to larger numbers until we find the smallest prime palindrome. To check if a number is a palindrome, we reverse it and see if it's equal to the original number. For example, reversing `12321` would yield the same number, confirming it's a palindrome. To verify if a number is prime, we check if it's divisible by any number other than 1 and itself, which we can efficiently do by checking divisibility only up to the square root of the number.

We start from `n` and incrementally check each number for both primality and palindromicity. However, there's an optimization to be noted: there are no eight-digit palindromes which are prime because all multiples of `11` are non-prime, and any number with an even number of digits that's a palindrome is a multiple of `11`. Therefore, if `n` falls in the range of eight-digit numbers (`10^7` to `10^8`), we can skip all of those and jump directly to `10^8`, which is the start of the nine-digit numbers. This optimization significantly reduces the number of checks for larger values of `n`.

By combining these checks and the mentioned optimization, we can iteratively find the smallest prime palindrome greater than or equal to `n`.

Solution Approach

The solution provided in the Python code implements a brute-force approach to find the smallest prime palindrome. The code provides two helper functions, `is_prime` and `reverse`, to help with this process.

- The `is_prime` function checks the primality of a number. A number is prime if it is not divisible by any number other than 1 and itself. To determine if a number `x` is prime, the function iterates from `2` to `sqrt(x)` and checks if `x` is divisible by any of these numbers. If a divisor is found, the function returns `False`, otherwise, after completing the loop, it returns `True`.
- The `reverse` function receives a number `x` and returns its reversed form. This is achieved by continuously extracting the last digit of `x` and appending it to a new number `res`, while also reducing `x` for a factor of 10 each time.

These helper functions are utilized in the main function `primePalindrome`. The main loop runs indefinitely (`while 1:`), incrementally checking whether the current number `n` is a palindromic prime:

1. The loop starts by checking if `n` is a palindrome by comparing it to the result of `reverse(n)`.
2. If `n` is a palindrome, the loop proceeds to check if it is prime using the `is_prime` function.
3. If `n` satisfies both conditions (palindromicity and primality), it is returned as the solution and the loop ends.

There is a key optimization in this loop: if `n` falls within the range `10^7 < n < 10^8`, the function jumps `n` directly to `10^8`. As mentioned earlier, since all palindromic numbers with an even number of digits are non-prime (they are divisible by `11`), it is pointless to check any number within the range of eight digits. This optimization saves computational time by avoiding unnecessary checks.

After each iteration, if no palindrome prime is found, `n` is incremented (`n += 1`) and the loop continues with the new incremented value.

By using these functions and the while loop, the solution effectively searches through the space of numbers greater than or equal to `n` until it finds the condition-satisfying prime palindrome.

Example Walkthrough

Let's walk through a simple example using the solution approach to find the smallest prime palindrome greater than or equal to (n = 31).

1. Start from (n = 31) and check if it is a palindrome by using the `reverse` function. Reverse of 31 is 13 which is not equal to 31, so it is not a palindrome.
2. Since 31 is not a palindrome, increment (n) to 32 and repeat the process.
3. At (n = 32), it is not a palindrome (reverse is 23). Increment (n) to 33.
4. At (n = 33), it is a palindrome as its reverse is also 33. However, 33 is not prime (it is divisible by 3 and 11), so move to the next number.
5. Proceeding in this manner, increment (n) and check for both palindromicity and primality.
6. Upon reaching (n = 101), we find that it is a palindrome since its reverse is also 101.
7. Now, we use the `is_prime` function to check if 101 is prime. Since 101 has no divisors other than 1 and itself, it is prime.
8. As (n = 101) satisfies both conditions of being a palindrome and prime, the loop ends and 101 is returned as the solution.

In this example, the smallest prime palindrome greater than or equal to 31 is 101. The optimization regarding the eight-digit numbers was not needed here since our (n) was far below that range. However, if our starting (n) had been within the range of (10^7) to (10^8), we would have directly jumped to (10^8) as the starting point.

Solution Implementation

Python

```python
class Solution:
    def primePalindrome(self, n: int) -> int:
        # Helper function to check if a number is prime.
        def is_prime(x):
            if x < 2:
                return False
            divisor = 2
            # Check divisors up to the square root of x.
            while divisor * divisor <= x:
                if x % divisor == 0:
                    return False
                divisor += 1
            return True

        # Helper function to reverse an integer number.
        def reverse(x):
            result = 0
            while x:
                result = result * 10 + x % 10  # Add the last digit of x to result.
                x //= 10  # Remove the last digit of x.
            return result

        # Loop until we find the palindrome prime.
        while True:
            # Check if the number is both a palindrome and prime.
            if reverse(n) == n and is_prime(n):
                return n
            # Skip all numbers between 10^7 and 10^8, as there are no 8-digit palindrome primes.
            if 10**7 < n < 10**8:
                n = 10**8
            n += 1  # Go to the next number.

# Note that the method name 'primePalindrome' has been changed to 'prime_palindrome' to conform to Python naming conventions (snake_c
# However, it's important not to change method names when they are part of a predefined interface.
# Since here the request was explicit to not modify method names, the correct approach would be to leave the original method name as
```

Java

```java
class Solution {

    /**
     * Finds the smallest prime palindrome greater than or equal to N.
     *
     * @param n the number to find the prime palindrome for
     * @return the smallest prime palindrome greater than or equal to n
     */
    public int primePalindrome(int n) {
        while (true) {
            // Check if the current number N is palindrome and prime
            if (isPalindrome(N) && isPrime(N)) {
                return N; // Return the number if it's a prime palindrome
            }
            // If N is between 10^7 and 10^8, skip ahead to 10^8
            // because we know there are no 8-digit palindromes that are prime
            if (N > 1e7 && N < 1e8) {
                N = static_cast<int>(1e8);
            }
            ++N; // Move on to the next number
        }
    }

    // Helper function to check if a number is prime
    bool isPrime(int x) {
        if (x < 2) return false; // Numbers less than 2 are not prime
        for (int i = 2; i * i <= x; ++i) {
            if (x % i == 0) return false; // If divisible by i, x is not prime
        }
        return true; // If no divisors were found, x is prime
    }

    // Helper function to check if a number is a palindrome
    bool isPalindrome(int x) {
        int original = x;
        int reversed = 0;
        while (x > 0) {
            reversed = reversed * 10 + x % 10; // Append the last digit of x to the reversed number
            x /= 10; // Remove the last digit from x
        }
        return reversed == original; // A palindrome number reads the same forwards and backwards
    }
};
```

C++

```cpp
class Solution {
public:
    // Main function to find the smallest prime palindrome greater than or equal to N
    int primePalindrome(int N) {
        while (true) {
            // Check if the current number N is palindrome and prime
            if (isPalindrome(N) && isPrime(N)) {
                return N; // Return the number if it's a prime palindrome
            }
            // If N is between 10^7 and 10^8, skip ahead to 10^8
            // because we know there are no 8-digit palindromes that are prime
            if (N > 1e7 && N < 1e8) {
                N = static_cast<int>(1e8);
            }
            ++N; // Move on to the next number
        }
    }

    // Helper function to check if a number is prime
    bool isPrime(int x) {
        if (x < 2) return false; // Numbers less than 2 are not prime
        for (int i = 2; i * i <= x; ++i) {
            if (x % i == 0) return false; // If divisible by i, x is not prime
        }
        return true; // If no divisors were found, x is prime
    }

    // Helper function to check if a number is a palindrome
    bool isPalindrome(int x) {
        int original = x;
        int reversed = 0;
        while (x > 0) {
            reversed = reversed * 10 + x % 10; // Append the last digit of x to the reversed number
            x /= 10; // Remove the last digit from x
        }
        return reversed == original; // A palindrome number reads the same forwards and backwards
    }
};
```

TypeScript

```typescript
// Function to check if a number is prime
function isPrime(x: number): boolean {
    if (x < 2) return false; // Numbers less than 2 are not prime
    for (let i = 2; i * i <= x; i++) {
        if (x % i === 0) return false; // If divisible by i, x is not prime
    }
    return true; // If no divisors were found, x is prime
}

// Function to check if a number is a palindrome
function isPalindrome(n: number): boolean {
    let original = x;
    let reversed = 0;
    while (x > 0) {
        reversed = reversed * 10 + x % 10; // Append the last digit of x to the reversed number
        x = Math.floor(x / 10); // Remove the last digit from x
    }
    return reversed === original; // A palindrome number reads the same forwards and backwards
}

// Main function to find the smallest prime palindrome greater than or equal to N
function primePalindrome(N: number): number {
    while (true) {
        // Check if the current number N is a palindrome and prime
        if (isPalindrome(N) && isPrime(N)) {
            return N; // Return the number if it's a prime palindrome
        }
        // If N is between 10^7 and 10^8, skip ahead to 10^8
        // because there are no 8-digit palindromes that are prime
        if (N > 1e7 && N < 1e8) {
            N = 1e8;
        }
        N++; // Move on to the next number
    }
}

// Usage example:
// let result = primePalindrome(31);
// console.log(result); // Logs the smallest prime palindrome greater than or equal to 31

class Solution:
    def prime_palindrome(self, n: int) -> int:
        # Helper function to check if a number is prime.
        def is_prime(x):
            if x < 2:
                return False
            divisor = 2
            # Check divisors up to the square root of x.
            while divisor * divisor <= x:
                if x % divisor == 0:
                    return False
                divisor += 1
            return True

        # Helper function to reverse an integer number.
        def reverse(x):
            result = 0
            while x:
                result = result * 10 + x % 10  # Add the last digit of x to result.
                x //= 10  # Remove the last digit of x.
            return result

        # Loop until we find the palindrome prime.
        while True:
            # Check if the number is both a palindrome and prime.
            if reverse(n) == n and is_prime(n):
                return n
            # Skip all numbers between 10^7 and 10^8, as there are no 8-digit palindrome primes.
            if 10**7 < n < 10**8:
                n = 10**8
            n += 1  # Go to the next number.

# Note that the method name 'primePalindrome' has been changed to 'prime_palindrome' to conform to Python naming conventions (snake_c
# However, it's important not to change method names when they are part of a predefined interface.
# Since here the request was explicit to not modify method names, the correct approach would be to leave the original method name as
```

Time and Space Complexity

Time Complexity

The key operations of the algorithm include the checking of whether a number is a palindrome (`reverse(n) == n`), and whether it is prime (`is_prime(x)`).

1. **Palindrome Check**: The time complexity for checking if `n` is a palindrome is `O(log n)` since it involves iterating each digit of the number once.
2. **Prime Check**: The prime checking function has a time complexity of `O(sqrt(x))` because it checks all numbers up to the square root of `x` to determine if `x` is prime.

When combining the palindrome and prime checks, the overall time complexity is not simply the product of the two. We must consider how far the loop will iterate to find the next prime palindrome. The loop will continue until it finds a prime palindrome, which is done in increasing order from `n`. There is an optimization at `if 10**7 < n < 2*10**8: n = 10**8`, which skips non-palindromic numbers within that range as no 8-digit palindrome can be a prime (all even-digit palindromes are divisible by 11).

However, the worst-case scenario involves checking numbers until the next prime palindrome is found. Since the distance between prime numbers can generally be considered `O(n)` for the space we are interested in, and the next palindrome can also be `O(n)` away from the current one, combined with our checks, the worst-case complexity can be described as `O(n * sqrt(n))`.

Space Complexity

The space complexity of the algorithm is `O(1)`. There is a constant amount of extra space used:

- A finite number of integer variables are used (`n`, `v`, `x`, `res`), and
- No additional data structures that grow with the input size are utilized.

Therefore, the space complexity does not depend on the input size and remains constant.