906. Super Palindromes

Leetcode Link

Problem Explanation

In the problem, we are asked to count Super-Palindromes between two given positive integers L and R. A positive integer is called a Super-Palindrome if it is a Palindrome and it is a square of a palindrome as well.

A Palindrome is a string which is similar when reading from start to end and end to start. e.g. "121", "12321", "1"

Solution Approach

If we analyze the problem, it is quite clear that we need to find all the palindrome numbers between sqrt(L) and sqrt(R) then check if the square of that number is also a palindrome.

The time complexity of this program is O(sqrt(n)) and space complexity is O(1).

Let's take an example to understand the approach:

Example 1: Input: L = "4", R = "1000" Output: 4

The numbers between sqrt(L) = sqrt(4) = 2 and sqrt(R) = sqrt(1000) = 31 are 2, 3, 11, 22, 121, 111, 121, 222, 232,....., 313.

Among these 11, 22, 111, 121, 212 are palindromes. Their squares are 121, 484, 12321, 14641, 44944 which are also palindromes. Therefore we conclude that there are 4 superpalindromes in the range.

Python Solution

```python

class Solution:
    def superpalindromesInRange(self, L: str, R: str) -> int:
        def is_palindrome(n):
            return str(n) == str(n)[::-1]

        def next_palindrome(n):
            s = str(n)
            l = len(s)
            half = s[:l // 2]
            ret = int(half + s[(l - 1) // 2] + half[::-1])
            if ret < n:
                return int(str(int(s[:l // 2 + 1]) + (s[:l // 2 + 1][::-1])[l%2:]))
            return ret

        l, r = int(L), int(R)
        lb, ub = int(l ** 0.5), int((r + 1) ** 0.5)
        ans = lb * lb == l
        if lb * lb != l:
            lb = next_palindrome(lb)
        for x in range(lb, ub + 1):
            if is_palindrome(x ** 2):
                ans += 1
            x = next_palindrome(x)

        return ans
```

Java Solution

```java

class Solution {
    public int superpalindromesInRange(String L, String R) {
        long l = Long.parseLong(L), r = Long.parseLong(R);
        int cnt = 0;
        for (long i = (long)Math.ceil(Math.sqrt(l)); i <= (long)Math.sqrt(r);) {
            long palindrome = updateToNextPalindrom(i);
            if (Math.sqrt(palindrome) == Math.floor(Math.sqrt(palindrome)) && isPalindrom(String.valueOf(palindrome))) cnt++;
            i++;
        }
        return cnt;
    }

    private boolean isPalindrom(String s) {
        int i = 0, j = s.length();
        while (i < j) {
            if (s.charAt(i++) != s.charAt(j--)) return false;
        }
        return true;
    }

    private long updateToNextPalindrom(long n) {
        char[] s = String.valueOf(n).toCharArray();
        int len = s.length;
        for (int i = len / 2; i < len; i++) {
            s[i] = s[len - i - 1];
        }
        long ret = Long.parseLong(new String(s));
        if (ret < n) {
            int half = Integer.parseInt(new String(s, 0, len / 2 + 1));
            half++;
            char[] halfArr = String.valueOf(half).toCharArray();
            for (int i = 0; i <= len / 2; i++) {
                s[i] = halfArr[i];
            }
            for (int i = len / 2; i < len; i++) {
                s[i] = s[len - i - 1];
            }
            ret = Long.parseLong(new String(s));
        }
        return ret;
    }
}

```

Note: Unfortunately, I am not proficient in JavaScript, C++, C#. However, the logic would remain the same; only syntax would differ.# JavaScript Solution

JavaScript solution for this problem goes like this:

We keep generating next higher palindrome numbers, check whether the square of the palindrome is a palindrome and also lies within the given range.

To get next palindrome, we can do that by splitting the number half, reverse it and attach it to the number. Increment the number if it is smaller than the current number.

Here is the JavaScript Solution for the problem:

```javascript
javascript
function superpalindromesInRange(left, right) {
    let L = parseInt(left), R = parseInt(right);
    let LOW = Math.floor(Math.sqrt(L)), HIGH = Math.ceil(Math.sqrt(R));
    let cnt = 0;

    for(let i = LOW; i <= HIGH;){
        let p = createPalindrome(i);
        if (p > 1) i = p;
        else {
            if (p*p <= R && isPalindrome(p*p)) cnt++;
            i = nextNumber(i);
        }
    }
    return cnt;
};

function createPalindrome(n) {
    let s = n.toString().split('');
    let len = s.length;
    let ret = s.slice(0, Math.floor(len/2)).concat(s[Math.floor((len - 1)/2)]).concat(s.slice(0, Math.floor(len/2)).reverse()).join
    return parseInt(ret);
}

function nextNumber(n) {
    let len = n.toString().length;
    let increment = Math.pow(10, Math.floor(len/2));
    n = Math.floor(n / increment) * increment + increment;
    return n;
};

function isPalindrome(n) {
    let s = n.toString();
    let len = s.length;
    for(let i = 0; i < len / 2; i++) {
        if(s[i] !== s[len - 1 - i]) {
            return false;
        }
    }
    return true;
};
```

In all solutions, the key idea for counting super-palindromes is to generate palindrome candidates and filter out those that are not super-palindromes. This approach significantly reduces the search space and allows for efficient computation.

Got a question? Ask the Teaching Assistant anything you don't understand.