564. Find the Closest Palindrome

Hard Math String Leetcode Link

Problem Description

The problem asks for the closest integer to a given integer `n`, where the closest integer must be a palindrome. A palindrome is a number that reads the same backward as forward. We're not allowed to consider the integer `n` itself, and if there are two palindromes equally close to `n`, we must return the smaller one. The closeness between two integers is defined by the absolute difference between them, so we are looking for the integer with the minimum absolute difference from `n` that is also a palindrome.

Intuition

To find the closest palindrome, we can consider different cases. The following intuition helps us arrive at a solution:

1. The nearest palindrome could be smaller or larger than `n`: We must check in both directions.
2. Palindromes of different lengths can be candidates: For example, if `n` is a 3-digit number, then 99 and 1001 (palindromes just below and above the nearest 3-digit ones) might be closer than any other 3-digit palindrome.
3. We can focus on changing the first half of `n`: Any palindrome can be reflected by its first half. For example, to form palindromes near 12345, we could reflect 123 to form 12321, or change it slightly to 12221 (by reflecting 122) or 12421 (by reflecting 124).
4. There's no need to examine too many candidates: Because we're looking for the *nearest* palindrome, we only need to look at a very small range of numbers around `n`.
5. We pick the closest palindrome and resolve ties by choosing the smaller: This is done by comparing absolute differences and values directly.

The solution creates potential candidates by examining numbers formed by reflecting the digits around the center of `n` and by considering edge cases like 999→1001 (where all digits are 9). It then discards the original number `n` if it's already a palindrome and compares the remaining candidates to find the one with the minimum absolute difference to `n`.

Solution Approach

The provided solution uses algorithmic techniques and a logical approach to determine the nearest palindrome. Here's how the implementation tackles the problem:

First, the solution creates a set `res` to store potential palindromic numbers, including two edge cases:

- `10 ** (l - 1) - 1` which handles the case where the closest palindrome has one less digit and consists of all 9's (for example, 999 is the closest palindrome to 1000).
- `10**l + 1` which handles the case of a palindrome that has one more digit and starts and ends with a 1 (for example, 1001 is the closest palindrome to 999).

The `int(n[: (l + 1) >> 1])` gets the first half of the digits (more precisely, the first half rounded-up if the length is odd, due to the `>> 1` which is equivalent to dividing by 2 using bit shifting).

The loop `for i in range(left - 1, left + 2):` generates palindromes by considering the reflected versions of numbers just below, equal to, and just above the first half of `n`. The `if l % 2 == 0` condition checks the length of `n` to decide on whether or not to consider the middle digit (for odd lengths) when creating the mirrored palindromes.

Within the loop, `while j:` is used to append the reversed digits of the first half to generate the full palindrome (`i`). Once complete, it adds these potential palindromes to the `res` set.

The line `res.discard(x)` ensures that the original number `n` is not considered if it is already a palindrome because the problem statement asks for a different number.

Finally, the solution iterates through the set `res` to find the candidate with the smallest absolute difference to `x`. It does this by considering both the difference and the value of `t`, using the `abs()` function to find the smallest absolute difference and resolve ties by choosing the smaller number.

In conclusion, the algorithm very efficiently narrows down the potential palindromes to consider and then picks the closest one by numerical comparison. By doing so, it avoids the brute force approach of checking each number in the range individually, which would be much less efficient.

Example Walkthrough

Let's work through a small example using the number `123`. According to the solution approach, our goal is to find the closest integer to `123` that is a palindrome, considering both smaller and larger potential palindromes.

First, we determine the length `l` of `n`. For `123`, `l` is `3`. We create a set `res` that we will use to store our potential palindrome candidates.

Now, we add our two edge cases to the set `res`:

- `10 ** (l - 1) - 1`, which for `123` gives us `99` (the largest 2-digit palindrome).
- `10 ** l + 1`, which for `123` gives us `1001` (the smallest 4-digit palindrome).

Next, we extract the first half of `n`. For `123`, the first half (rounded up in the case of odd lengths) is `12`. We will use this to generate palindromes that have the same first half as `123`.

The solution then considers the numbers just below, equal to, and just above `12`: these are `11`, `12`, and `13`.

For each of these numbers, we generate a palindrome by mirroring the first half. For instance:

- Mirroring `11` gives us `111`.
- Mirroring `12` (which is `n`'s exact first half) gives us `121`.
- Mirroring `13` gives us `131`.

We add these palindromes to our set `res`. Now, `res` contains `99`, `1001`, `111`, `121`, and `131`.

Since `123` is not a palindrome, we don't need to discard it from our set `res`. If `123` were a palindrome, we would remove it from the set since we are looking for a different integer.

Finally, we iterate through our set `res` to determine which number has the smallest absolute difference to `123`. We calculate as follows:

- `abs(99 - 123)` gives us `24`.
- `abs(1001 - 123)` gives us `878`.
- `abs(111 - 123)` gives us `12`.
- `abs(121 - 123)` gives us `2`.
- `abs(131 - 123)` gives us `8`.

The smallest difference is `2` for the palindrome `121`. Since we are asked to return the closest palindrome and in case of a tie return the smaller number, `121` is our answer as it is the closest palindrome to `123`.

Python Solution

```python
class Solution:
    def nearestPalindromic(self, n: str) -> str:
        # Convert the string to an integer for numerical operations
        num = int(n)
        num_length = len(n)

        # Initialize a set with the smallest and largest possible palindromes
        # with different digit lengths compared to the input number
        candidates = {
            10**(num_length - 1) - 1,  # Smallest palindrome with one less digit
            10**num_length + 1          # Smallest palindrome with one more digit
        }

        # Find the higher and lower palindromes around the input number
        # 'prefix' is the first half of the input number
        prefix = int(n[:(num_length + 1) // 2])

        # Generate palindromes by varying the prefix by -1, 0, +1
        for i in range(prefix - 1, prefix + 2):
            # For even lengths, use the entire prefix.
            # For odd lengths, exclude the last digit of the prefix.
            j = i if num_length % 2 == 0 else i // 10

            # Append the reverse of 'j' to 'i' to construct the palindrome
            palindrome = i
            while j > 0:
                palindrome = palindrome * 10 + j % 10
                j //= 10

            # Add the constructed palindrome to the candidate set
            candidates.add(palindrome)

        # Remove the original number itself if it's in the set
        candidates.discard(num)

        # Find the closest palindrome to the original number
        closest_palindrome = -1
        for candidate in candidates:
            # Check for the smallest absolute difference
            # If the absolute difference is the same, choose the smaller number
            if (closest_palindrome == -1 or
                abs(candidate - num) < abs(closest_palindrome - num) or
                (abs(candidate - num) == abs(closest_palindrome - num) and candidate < closest_palindrome)):
                closest_palindrome = candidate

        # Convert the closest palindrome back to a string and return
        return str(closest_palindrome)

```

Java Solution

```java
class Solution {

    // Function to find the nearest palindromic number in string form
    public String nearestPalindromic(String n) {
        // Convert the input string to a long integer for comparison
        long number = Long.parseLong(n);

        // Variable to store the closest palindrome number
        long closestPalindrome = -1;

        // Get all potential palindrome candidates
        for (long candidate : getPalindromeCandidates(n)) {
            // If this is the first candidate or it's closer to the input number than the current closest
            // or equally close but smaller, then update the closest palindrome
            if (closestPalindrome == -1 ||
                Math.abs(candidate - number) < Math.abs(closestPalindrome - number) ||
                (Math.abs(candidate - number) == Math.abs(closestPalindrome - number) && candidate < closestPalindrome)) {

                closestPalindrome = candidate;
            }
        }
        // Convert the closest palindrome back to a string and return it
        return Long.toString(closestPalindrome);
    }

    // Helper function to generate palindrome candidates based on the input string
    private Set<Long> getPalindromeCandidates(String n) {
        int length = n.length(); // Length of the input number
        Set<Long> candidates = new HashSet<>(); // Set to store palindrome candidates

        // Add 9's (One less digit than n and all 9's) e.g. 999 for n=1000
        candidates.add((long)Math.pow(10, length - 1) - 1);
        // Add 1 followed by all zeros and then a 1 (One more digit than n) e.g. 10001 for n=999
        candidates.add((long)Math.pow(10, length) + 1);

        // Get the first half of n (if odd, include the middle digit)
        long firstHalf = Long.parseLong(n.substring(0, (length + 1) / 2));
        // Generate candidates by varying the first half from -1 to 1 and mirroring to get palindromes
        for (long i = firstHalf - 1; i <= firstHalf + 1; ++i) {
            StringBuilder candidateBuilder = new StringBuilder();
            candidateBuilder.append(i); // Append the first half
            // Mirror and append the reverse of the first half (excluding the middle digit if odd length)
            candidateBuilder.append(new StringBuilder(Long.toString(i)).reverse().substring(length % 2));
            // Add the generated number to candidates
            candidates.add(Long.parseLong(candidateBuilder.toString()));
        }

        // Remove the number itself if it's a palindrome, as we want the nearest different palindrome
        candidates.remove(Long.parseLong(n));

        return candidates; // Return the set of candidates
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Main function to find the nearest palindrome
    string nearestPalindromic(string n) {
        long originalNumber = stol(n); // Convert the original string to a long
        long closestPalindrome = -1;   // Initialize the closest palindrome

        // Loop through all possible palindromes and choose the nearest one
        for (long candidate : getCandidates(n)) {
            if (closestPalindrome == -1 ||
                abs(candidate - originalNumber) < abs(closestPalindrome - originalNumber) ||
                (abs(candidate - originalNumber) == abs(closestPalindrome - originalNumber) && candidate < closestPalindrome)) {
                closestPalindrome = candidate;
            }
        }
        return to_string(closestPalindrome); // Return the string representation of the nearest palindrome
    }

    // Helper function to generate possible palindromes
    unordered_set<long> getCandidates(string& n) {
        int length = n.size();
        unordered_set<long> candidates;

        // Add the edge cases: the largest number with one less digit and the smallest number with one more digit
        candidates.insert((long) pow(10, length - 1) - 1);
        candidates.insert((long) pow(10, length) + 1);

        // Get the first half of the original number and consider the number one less, one greater, and itself
        long firstHalf = stol(n.substr(0, (length + 1) / 2));
        for (long i = firstHalf - 1; i <= firstHalf + 1; ++i) {
            string prefix = to_string(i); // First half as a string

            // Create a palindrome by appending the reverse of the prefix, considering odd/even length
            string candidate = prefix + string(prefix.rbegin() + (length & 1), prefix.rend());

            // Add the generated palindrome to the set of candidates
            candidates.insert(stol(candidate));
        }

        // Remove the original number from the candidate set to ensure it's not considered as its own nearest palindrome
        candidates.erase(originalNumber);

        return candidates; // Return the set of candidate palindromes
    }
};

```

Typescript Solution

```typescript
// Main function to find the nearest palindrome
function nearestPalindromic(n: string): string {
    const originalNumber: number = parseInt(n, 10); // Convert the original string to a number
    let closestPalindrome: number = -1; // Initialize the closest palindrome

    // Loop through all possible palindromes and choose the nearest one
    for (const candidate of getCandidates(n)) {
        if (closestPalindrome === -1 ||
            Math.abs(candidate - originalNumber) < Math.abs(closestPalindrome - originalNumber) ||
            (Math.abs(candidate - originalNumber) === Math.abs(closestPalindrome - originalNumber) && candidate < closestPalindrome
            closestPalindrome = candidate;
        }
    }
    return closestPalindrome.toString(); // Return the string representation of the nearest palindrome
}

// Helper function to generate possible palindrome candidates
function getCandidates(n: string): Set<number> {
    const length: number = n.length;
    const candidates: Set<number> = new Set<number>();

    // Add the edge cases: the largest number with one less digit and the smallest number with one more digit
    candidates.add(Math.pow(10, length - 1) - 1);
    candidates.add(Math.pow(10, length) + 1);

    // Get the first half of the original number and consider the number one less, one greater, and itself
    const firstHalf: number = parseInt(n.substr(0, Math.ceil(length / 2)), 10);
    for (let i: number = firstHalf - 1; i <= firstHalf + 1; i++) {
        const prefix: string = i.toString(); // First half as a string

        // Create a palindrome by appending the reverse of the prefix, considering odd/even length
        const reversedPrefix: string = prefix.split('').reverse().join('');
        const candidateStr: string = prefix + reversedPrefix.substr(length % 2);

        // Add the generated palindrome to the set of candidates
        candidates.add(parseInt(candidateStr, 10));
    }

    // Remove the original number from the candidate set to ensure it's not considered
    // as its own nearest palindrome
    candidates.delete(originalNumber);

    return candidates; // Return the set of candidate palindromes
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code can be broken down as follows:

1. Calculating `res` by adding the smallest and largest possible palindrome numbers around the length of `n`, which is `O(1)` since size of the `res` set remains constant regardless of the size of `n`.

2. Generation of near-by candidate palindromes involves constant time operations by iterating over a small range `left - 1` to `left + 2`, which is `O(1)`.

3. Constructing palindromes involves a while loop that iterates approximately `l/2` times, where `l` is the length of `n`, resulting in `O(len(n))` for each palindrome creation. Since we create at most three palindromes, the overall contribution is `O(len(n))`.

4. Discarding the original number `x` from the set `res` is also `O(1)` as it's a single operation.

5. The final for loop to choose the best palindrome candidate iterates over the elements in `res`. Since `res` has a maximum of 5 elements (fixed size), the loop runs in `O(1)` time.

Overall, summing these contributions, the time complexity of the code is primarily dependent on the length of `n`, leading to `O(len(n))`.

Space Complexity

The space complexity analysis is as follows:

1. A set `res` is created which holds at most 5 integers, thus contributing `O(1)` space.

2. Temporary variables such as `x`, `l`, `left`, `i`, `j`, `ans`, and `t` are all constant space overhead, adding up to `O(1)`.

Therefore, the total space complexity of the code is `O(1)`.