1903. Largest Odd Number in String

Easy Greedy Math String

Problem Description

In the given problem, we have to work with a string `num` that represents a large integer. The objective is to find a substring of this string that represents the largest odd integer. A substring is defined as a sequence of characters that are contiguous; in other words, the characters are next to each other without any gaps. If there are no odd integers at all in the string, then the function should return an empty string `""`. Simply put, we need to parse through the string to find the largest odd integer by examining various portions of the string, taking care not to break up the order of the characters as we do so. It's important to remember that an odd integer is one whose last digit is 1, 3, 5, 7, or 9.

Intuition

To solve this problem, we can use a straightforward approach. Since any leading zeros in a number do not affect its value, we can ignore them. What matters is the rightmost digit, because it determines if a number is odd or even. Therefore, we can scan the string from right to left, looking for the first odd digit (1, 3, 5, 7, or 9). Once we find it, we can take the substring that starts from the beginning of `num` and goes up to and includes this digit. This is the largest odd integer that can be formed as it includes the maximum number of leading digits from the original string.

If no odd digit is found by the time we reach the beginning of the string (index 0), then we return an empty string since it's not possible to form an odd integer from `num`. This method is efficient because we don't need to check every possible substring; we just stop at the rightmost odd digit. It works because any smaller substring ending with the same digit would be smaller or equal in value and thus would not be the maximum odd integer that could be formed.

Solution Approach

The solution makes use of a simple for-loop to iterate through the given string `num`. The loop runs in reverse, starting from the last character and moving towards the first. This is accomplished by setting the for-loop's range with `len(num) - 1` as the starting index, `-1` as the stop index, and `-1` as the step, which means "move one step backward".

Here are the steps taken inside the for-loop in the solution:

1. Check if the current digit is an odd number:
   - Convert the current character at index `i` to an integer using `int(num[i])`.
   - Determine if the last bit is set (which would make it odd) by using bitwise AND with `1`.
2. If the condition `(int(num[i]) & 1) == 1` meets:
   - It implies that the digit at index `i` is odd.
   - We then use slicing to obtain the substring from the start of `num` to the index `i` inclusive, using `num[: i + 1]`.
   - This substring is then returned as it is the largest possible odd integer that can be formed from a substring of the original number.
3. If no odd digit is found by the end of the loop:
   - The for-loop exits without returning any value from inside the loop.
   - The function then returns an empty string (`''`) after the loop completes, signaling that no odd integer could be found within `num`.

No additional data structures are needed for this solution, keeping the space complexity at O(1), as we are returning a substring of the original input. The time complexity is O(N), where N is the length of the string, as we may potentially have to scan through the entire string once.

Example Walkthrough

Let's assume the string `num` is "1234567890". We need to find the largest odd integer that can be created as a substring from this number.

To do so, we follow the described solution approach and scan the string from right to left. That means we start with the last character of the string and move towards the first character:

1. We start at index 9 (the last index of `num`), which is the digit '0'. This is not an odd digit, so we move one step backward in the string.
2. At index 8, we find the digit '9', which is odd (converting to integer `int('9')` and applying the bitwise AND operation with `1` results in `1`, thus it's odd).
3. Since we've encountered an odd digit, we take the substring from the start of `num` to this index (inclusive). Using slicing, we extract `num[:8+1]`, which gives us "123456789".
4. "123456789" is the largest odd integer we can form from the substring of `num`, so we return this value.

With this example, we can see that the method of scanning from right to left and taking the first odd number we come across gives us the largest possible odd integer in the form of a substring.

Solution Implementation

Python

```python
class Solution:
    def largest_odd_number(self, num: str) -> str:
        # Start from the end of the string and move backwards
        for i in range(len(num) - 1, -1, -1):
            # Check if the current digit is odd
            if int(num[i]) % 2 == 1:
                # If an odd digit is found, return the substring up to this digit (inclusive)
                return num[:i + 1]
        # If no odd digit is found, return an empty string
        return ''
```

I'll note that the method name `largest_odd_number` was changed from `largestOddNumber` to follow the Python convention of using

Here's the version maintaining the required LeetCode method name:

```python
class Solution:
    def largestOddNumber(self, num: str) -> str:
        # Iterate over the number string in reverse order
        for i in range(len(num) - 1, -1, -1):
            # Check if the current character represents an odd digit
            if int(num[i]) % 2 == 1:
                # If it does, return the substring from the start to the current character inclusive
                return num[:i + 1]
        # If no odd digit was found, return an empty string
        return ''
```

Java

```java
class Solution {
    public String largestOddNumber(String num) {
        // Iterate from the end of the string to the beginning
        for (int index = num.length() - 1; index >= 0; index--) {
            // Convert the character at the current index to an integer
            int digit = num.charAt(index) - '0';

            // Check if the digit is odd
            if ((digit & 1) == 1) {
                // If it's odd, return the substring from the start to the current index + 1
                // This is because substring function in Java is end-exclusive
                return num.substring(0, index + 1);
            }
        }

        // If no odd digit is found, return an empty string
        return "";
    }
}
```

C++

```cpp
#include <string>

class Solution {
public:
    // Function to find the largest odd number string from the input string 'num'
    string largestOddNumber(string num) {
        // Iterate from the end of the string towards the beginning
        for (int index = num.size() - 1; index >= 0; --index) {
            // Convert the current character to its numerical value
            int digit = num[index] - '0';

            // Check if the digit is odd using bitwise AND operation
            if ((digit & 1) == 1) {
                // If odd, return the substring from the beginning up to the current index (inclusive)
                return num.substr(0, index + 1);
            }
        }
        // If no odd number is found, return an empty string
        return "";
    }
};
```

TypeScript

```typescript
/**
 * Given a numeric string, this function finds and returns the largest odd number
 * that can be formed by a substring of the input.
 * If no odd number can be formed, it returns an empty string.
 *
 * @param {string} num - The numeric string from which to extract the largest odd number.
 * @return {string} - The largest odd number that can be formed, or an empty string if not possible.
 */
function largestOddNumber(num: string): string {
    // Get the length of the numeric string.
    const numLength: number = num.length;

    // Iterate over the string from end to start.
    for (let index = numLength - 1; index >= 0; index--) {
        // Check if the current character represents an odd digit.
        if (parseInt(num.charAt(index)) % 2 === 1) {
            // Return the substring from start to the current odd digit (inclusive).
            return num.slice(0, index + 1);
        }
    }

    // If no odd number found, return an empty string.
    return '';
};
```

```python
class Solution:
    def largest_odd_number(self, num: str) -> str:
        # Start from the end of the string and move backwards
        for i in range(len(num) - 1, -1, -1):
            # Check if the current digit is odd
            if int(num[i]) % 2 == 1:
                # If an odd digit is found, return the substring up to this digit (inclusive)
                return num[:i + 1]
        # If no odd digit is found, return an empty string
        return ''
```

I'll note that the method name `largest_odd_number` was changed from `largestOddNumber` to follow the Python convention of using snak

Here's the version maintaining the required LeetCode method name:

```python
class Solution:
    def largestOddNumber(self, num: str) -> str:
        # Iterate over the number string in reverse order
        for i in range(len(num) - 1, -1, -1):
            # Check if the current character represents an odd digit
            if int(num[i]) % 2 == 1:
                # If it does, return the substring from the start to the current character inclusive
                return num[:i + 1]
        # If no odd digit was found, return an empty string
        return ''
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code is `O(n)`, where `n` is the length of the input string `num`. This is because the code consists of a single loop that iterates over the string from the end towards the beginning. In the worst-case scenario, the loop runs for the entire length of the string if the last odd number is at the beginning of the string or there is no odd number at all.

Space Complexity

The space complexity of the code is `O(1)`. No additional space is used that grows with the size of the input. The return statement `num[: i + 1]` creates a new string, but since string slicing in Python does not create a new copy of the characters (instead, it just creates a new view of the original string), the space used does not depend on the size of the input string.