1016. Binary String With Substrings Representing 1 To N

Medium String

Leetcode Link

Problem Description

The problem requires us to determine if a given binary string `s` contains all the binary representations of numbers from 1 to a given positive integer `n` as substrings. A substring is a sequence of characters that occur in order without any interruptions within a string. For example, if `s = "0110"` and `n = 3`, we need to check if the binary representations of `1` (`"1"`), `2` (`"10"`), and `3` (`"11"`) are all present as substrings in `s`.

To solve the problem, we need to generate the binary representation of each number from 1 to `n` and verify if each representation is a substring of `s`.

Intuition

The solution involves a few key observations and steps:

1. **Binary Length**: The binary representation of a number grows in length when the number doubles. Therefore, as we approach `n`, more significant numbers would have longer binary strings, and if they are not found in `s`, we can determine the answer is `false` without checking smaller numbers.

2. **Efficient Checking**: We start checking from `n` and go down to `n//2`. The reasoning is that every binary string that would represent a number from `1` to `n//2` will also be a substring of a string representing a number from `n//2` to `n`. For example, "10" for `2` is contained within "110" for `6`.

3. **Performance Boundaries**: Since the binary length grows with the number size, the code includes a quick exit condition when `n` is greater than 1000, possibly to avoid performance issues with extremely large strings. However, this condition seems arbitrary and depends on constraints not mentioned in the problem description. It may not be necessary if the input string `s` is always large enough to potentially contain all representations.

4. **All-encompassing Check**: To verify if all binary representations are in `s`, a Python built-in function `all()` is used, which checks for the truthiness of all elements in an iterable. In this case, a generator expression checks if each binary representation as a string is a substring of `s` (`bin(i)[2:] in s`).

The intuition behind this approach is that by checking the larger half of the range first using string containment operations, one can determine if the binary representations of numbers in the range `[1, n]` are substrings of the given string `s` with higher efficiency than checking every single number starting from 1.

Solution Approach

The solution approach utilizes a simple and direct method for checking the presence of binary substrings within the given string `s`. Below are the components and steps of the implementation using the provided Python code:

1. **Function Signature**:
   - `def queryString(self, s: str, n: int) -> bool`: This is the function signature where `s` is the input string and `n` is the integer until which we need to check the binary representations. The function returns a Boolean value.
2. **Early Return Condition**:
   - `if n > 1000: return False`: The code immediately returns `False` if `n` is more than 1000, implying a performance optimization for large numbers, but as mentioned earlier, this condition may be arbitrary.
3. **Main Checking Loop**:
   - `return all(bin(i)[2:] in s for i in range(n, n // 2, -1))`: Here the algorithm employs Python's built-in function `all()` which tests if all elements in an iterable are true. The iterable, in this case, is a generator expression that goes through the range of integers from `n` to `n // 2` (integer division by 2) in descending order.
4. **Binary Conversion and Substring Check**:
   - `bin(i)[2:] in s`: For each `i` in the specified range, the built-in `bin()` function generates its binary representation as a string and strips off the '0b' prefix with `[2:]`. Then we check whether this binary representation is a substring of `s`.
5. **Data Structures**:
   - No additional data structures are used in this solution. The input string `s` and integer `n` are directly utilized within the algorithm.
6. **Algorithmic Patterns**:
   - The algorithm does not use complex patterns or advanced data structures. It relies on Python's expressive syntax and built-in functions to perform substring checking efficiently.

The loop iterates through only half of the specified range (from `n` to `n // 2`) in reverse order because if `i` can be represented within `s` as a binary string, all smaller numbers can also be substrings of those representations or will have occurred earlier within the binary sequence. This approach takes advantage of the fact that binary representations of numbers are nested within those of larger numbers, providing a significant performance improvement over checking every single number individually from 1 to `n`.

In summary, the code leverages Python's capabilities to check for the presence of each binary representation of numbers in the given range within the string `s` succinctly and efficiently.

Example Walkthrough

To illustrate the solution, let's consider an example where `s = "01101101"` and `n = 4`. We would like to determine if the binary representations of numbers from 1 to 4, which are "1" (1), "10" (2), "11" (3), and "100" (4), appear as substrings in `s`.

Steps:

1. **Check for Early Return**: The first check in our solution is to see if `n` is greater than 1000. In this case, `n = 4`, so we do not return early and proceed.

2. **Initialize the Main Loop**: Starting with `i = 4` and iterating down to `i = 2` (half of `n`), we check each binary representation to see if it's a substring in `s`.

3. **Binary Conversion**:
   - For `i = 4`, we convert `4` to binary, getting `"100"`. Check if `"100"` is in `s`. It is not, so we could already return `False`. However, continue for illustration purposes.
   - For `i = 3`, convert `3` to binary, yielding `"11"`. Check if `"11"` is a substring of `s`, which it is, as `s` is `"01101101"`.
   - For `i = 2`, convert `2` to binary to get `"10"`. We then check if `"10"` is part of `s`, and indeed, it is present.
4. **Evaluate with `all()` Function**: Use the `all()` function to verify whether all these checks (`i = 4` to `i = 2`) are `True`. Since `"100"` was not found, `all()` will return `False`.

5. **No Additional Data Structures**: We have not used additional data structures outside the string `s` and the numbers we're checking.

6. **Conclusion**: Since not all binary representations from 1 to 4 were found as substrings in `s` (`"100"` was missing), the `all()` function will evaluate to `False`. Therefore, the given string `s` does not contain all binary representations as substrings.

This step-by-step walkthrough demonstrates the simplicity and efficiency of the approach by focusing on checking only the necessary binary representations and utilizing Python's built-in tools.

Python Solution

```python
class Solution:
    def query_string(self, string: str, upper_bound: int) -> bool:
        # If the upper bound (n) is greater than 1000, return false as per the given logic.
        if upper_bound > 1000:
            return False

        # 'all' checks whether all elements in the iterable are True.
        # The loop starts from 'upper_bound' and goes till 'upper_bound // 2' (integer division by 2), moving backwards.
        # 'bin(i)[2:]' converts the integer 'i' to its binary representation in string format, stripping off the '0b' prefix.
        # 'in string' checks if each binary representation is a substring of the input string 'string'.
        return all(bin(i)[2:] in string for i in range(upper_bound, upper_bound // 2, -1))

# Example usage:
# sol = Solution()
# result = sol.query_string("0110", 3)
# print(result) # This would print 'True' if all binary numbers from n to n//2 are substrings of "0110"

```

Java Solution

```java
class Solution {

  /**
   * Checks if all binary representations of numbers from 1 to n are substrings of the string s.
   *
   * @param s The string in which binary representations are to be searched.
   * @param n The maximum value up to which binary representations should be checked.
   * @return True if all binary representations from 1 to n are found in s, otherwise False.
   */
  public boolean queryString(String s, int n) {
    // If n is greater than the maximum allowed value (as binary representation within the string),
    // which is 2^10 - 1 = 1023 for a 10-bit binary number, return false as the condition can't be met.
    if (n > 1023) {
      return false;
    }
    // Iterate from n down to n / 2 since every number smaller than n/2 is a binary substring of a number
    // that is larger than n/2 (because the binary representation of a number is also a suffix of the
    // binary representation of its double).
    // If a substring representing the binary of i is not found within string s, return false.
    for (int i = n; i > n / 2; i--) {
      if (!s.contains(Integer.toBinaryString(i))) {
        return false;
      }
    }
    // Return true if all required binary substrings have been found.
    return true;
  }
}

```

C++ Solution

```cpp
#include <bitset>
#include <string>

class Solution {
public:
    // Function to check if all binary representations of numbers
    // from 1 to n are substrings of the input string s.
    bool queryString(std::string s, int n) {
        // Early exit condition if n is greater than the maximum
        // value representable with 10 binary digits.
        if (n > 1023) {
            return false;
        }
        // Loop through numbers starting from n to half of n
        // since the bit representation of numbers less than n/2
        // will always be contained in the bit representation of
        // numbers between n/2 and n.
        for (int i = n; i > n / 2; --i) {
            // Convert the number to binary (bitset) and then to string.
            std::string binaryString = std::bitset<32>(i).to_string();
            // Remove leading zeroes from the binary string.
            binaryString.erase(0, binaryString.find_first_not_of('0'));
            // Check if the resulting binary string is a substring of s.
            if (s.find(binaryString) == std::string::npos) {
                // If not found, return false immediately.
                return false;
            }
        }
        // All required binary representations are substrings of s. Return true.
        return true;
    }
};

```

Typescript Solution

```typescript
/**
 * Checks if a binary string represents all numbers from 1 to n.
 *
 * @param {string} binaryString - The string consisting of binary digits.
 * @param {number} maxNumber - The maximum number to check up to (inclusive).
 * @returns {boolean} - Returns true if all numbers from 1 to maxNumber are
 * represented in the binaryString, otherwise false.
 */
function queryString(binaryString: string, maxNumber: number): boolean {
    // Check if the maxNumber is larger than the largest number that can be represented
    // with a 10-bit binary number, which is 1023.
    if (maxNumber > 1023) {
        return false;
    }
    // Iterate from the maxNumber down to half of it, since every number from 1 to n / 2
    // will be a substring of the binary representation of numbers from n / 2 + 1 to n.
    for (let currentNum = maxNumber; currentNum > maxNumber / 2; --currentNum) {
        // Convert the current number to its binary string representation.
        const binaryRepresentation = currentNum.toString(2);
        // Check if the current binary representation is a substring of binaryString.
        // If it is not found, the method returns -1, therefore, return false.
        if (binaryString.indexOf(binaryRepresentation) === -1) {
            return false;
        }
    }
    // If all necessary binary representations are found in the string, return true.
    return true;
}

```

Time and Space Complexity

The time complexity of the provided code can be determined by analyzing the two main operations: the `all` function and the string containment check `in`.

Time Complexity:

- The `all` function iterates over the range from `n` to `n // 2`, decreasing by 1 each time. This results in approximately `n/2` iterations.
- For each iteration, the string containment check `in` is performed, which, in the worst case, has a complexity of `O(m)`, where `m` is the length of the string `s`.

Therefore, the overall worst-case time complexity is `O(m * n/2)`, as the containment check is performed `n/2` times.

Space Complexity:

- No extra space is used for data storage that scales with the input size; only a fixed number of variables are used.
- The binary representation string created for each number `i` is temporary and its length is at most `O(log(n))`.

Hence, the space complexity is `O(1)`, which means it is constant since the space required does not grow with the input size `n` or the string length `m`.