2311. Longest Binary Subsequence Less Than or Equal to K

Medium Greedy Memoization String Dynamic Programming

Problem Description

In this problem, you are given a binary string `s` which only consists of `0`s and `1`s. You are also given a positive integer `k`. The task is to find the length of the longest subsequence of the given binary string `s` such that when the subsequence is treated as a binary number, it is less than or equal to `k`. There are a couple of extra points to keep in mind:

- Leading zeroes in the subsequence are allowed.
- An empty subsequence is considered as binary `0`.
- A subsequence can be derived from the string by deleting some or no characters without changing the order of the remaining characters.

To clarify, a binary string is a sequence of bits (where each bit is either `0` or `1`). The length of a subsequence is the total number of bits it contains.

Intuition

For solving this problem, we start thinking about the properties of binary numbers. A crucial observation is that the least significant bits have the least effect on the value of the binary number. For example, in binary `1011`, if we drop the last `1`, the number changes from `11` (in decimal: 3) to `1010` (in decimal: 10), which is a much smaller change compared to dropping a higher bit.

Given that we can only choose subsequences (i.e., we can't rearrange the bits), the logical approach is to consider bits from the least significant to the most significant (from right to left in the string) and decide whether to include them in our subsequence.

One key insight is to realize that we should always take as many `0`s as possible because they do not increase the value of our number. On the other hand, taking a `1` would increase the value, so we should only take a `1` if it doesn't cause the subsequence to exceed `k`. Also, we should process the string in reverse to ensure we are considering the bits from low to high significance.

Another detail is that because integer values can only be accurately represented in most programming languages up to a certain limit (typically 32 bits for a `signed int` in many languages), and because `k` can be at most 10^9, we only need to consider the last 30 bits of any subsequence when determining if its value is less than or equal to `k`. This significantly simplifies the problem because we don't need to work with potentially very long binary strings; we can ignore any bits past the 30th most significant bit.

The provided code snippet shows an efficient realization of these ideas. It initializes a counter `ans` to keep track of the length of our subsequence and a value `v` which is the decimal representation of the subsequence being constructed. It iterates over the binary string `s` in reverse. When it encounters a `0`, it can safely include it, so it increments the `ans`. When it encounters a `1`, it checks whether including this `1` would keep the value `v` under `k`. If so, it sets the corresponding bit in `v` (using bitwise OR and bitwise shift operations) and increments `ans`. For this check, it also makes sure that the number of bits processed is less than 30 to avoid overflow issues.

By the end of this process, `ans` holds the maximum length of a subsequence that represents a number less than or equal to `k`.

Solution Approach

The implementation of the solution is quite straightforward once the intuition is understood.

Here's a step-by-step breakdown of the implemented code:

1. Initialize two variables:
   - `ans` set to `0`, which will ultimately contain the length of the longest valid subsequence.
   - `v` set to `0`, which will keep track of the current numeric value of the subsequence being considered in decimal.
2. Loop through the binary string in reverse order using `for c in s[::-1]:`. This ensures that we examine bits from least significant to most significant.
3. Inside the loop, check if the current character `c` is "0":
   - If that's the case, since a zero does not change the current value of `v` but can still be part of the subsequence, increment `ans` by 1.
4. If the character is "1", there are two conditions to check:
   - First, check if the length of the subsequence is less than 30 (`ans < 30`). This is important to prevent overflow and because we are only interested in subsequences that can affect the value within the integer limit, as explained in the intuition part.
   - Second, we want to make sure that by setting the current bit (which we obtain by `1 << ans`) the value of `v` would still be less than or equal to `k`. This is done by the operation `(v | 1 << ans) <= k`, which uses a bitwise OR (`|`) coupled with a left shift (`<<`) to add the current bit to `v`.
5. If both conditions hold true for "1", we update `v` to include the current bit by setting `v |= 1 << ans` and increment `ans`.
6. Once the loop finishes, `ans` would have the length of the longest valid subsequence by iteratively building it from the lowest order bit towards the highest order bit considered, ensuring that at any time the subsequence's value does not exceed `k`.

The code uses a bitwise approach to manipulate and evaluate the binary numbers without converting them to their decimal counterparts. This pattern leverages the efficiency of bitwise operations and keeps the implementation clean and straightforward.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Consider the following scenario:

- binary string `s` = "10101"
- integer `k` = 5 (binary representation: "101")

We aim to find the longest subsequence of `s` that, when treated as a binary number, does not exceed `k` (5 in decimal). Following the steps from the solution approach:

1. Initialize `ans` to 0 and `v` to 0. `ans` will track the length of our subsequence, and `v` will track the numeric value of the subsequence as we build it.
2. Process the binary string in reverse. Our string `s` is "10101", so we start from the last character and move to the first (i.e., "10101" -> "10101").
3. The string in reverse is "10101", which we process from left to right:
   - 1st char ('1'): `ans` is 0, so `1 << ans` is `1`. We can't include this '1' because `v | 1` would become `1`, which is not less than or equal to `k` (5).
   - 2nd char ('0'): We include this zero because adding zeros doesn't change the value. Now, `ans` is incremented to 1.
   - 3rd char ('0'): Include this zero as well, increment `ans` to 2.
   - 4th char ('1'): Check if `ans` is less than 30 and if `(v | 1 << ans) <= k`. `ans` is 2, so `1 << ans` equals `4`, and the current `v` is `0`. The result of `v | 4` is `4`, which is less than or equal to `k` (5), so include this '1' and increment `ans` to 3.
   - 5th char ('1'): Now `ans` is 3, so `1 << ans` equals `8`, and the current `v` is `4`. `v | 8` would be `12`, which exceeds `k`. Therefore, we cannot include this '1'.
4. The loop ends with `ans` equal to 3. The subsequence that we can form is "001" (when read in the original order it's "100"), which equals 4 in decimal, and is less than or equal to `k`.

The longest valid subsequence of the original binary string "10101", which represents a value less than or equal to `5`, is "100" (in the subsequence order it's "001"), and its length is 3. Therefore, our `ans` is 3, which is the final result.

Solution Implementation

Python

```python
class Solution:
    def longestSubsequence(self, s: str, k: int) -> int:
        # Initialize the count of the subsequence and the value of the binary string seen so far
        subsequence_length = value_so_far = 0

        # Iterate over the string in reverse order since the least significant bit contributes less to the overall value
        for character in reversed(s):
            # If the character is '0', it doesn't affect the value but can increase the length of the subsequence
            if character == "0":
                subsequence_length += 1
            # If the character is '1', check if adding this bit exceeds the value of k
            # Since we're looking at the binary string from right to left, we shift 1 left by the current subsequence length
            # The subsequence length represents the binary digit's position (0-indexed)
            # Also note that we perform this check only if subsequence length is less than 30 since 2^30 is the first power of 2 that
            elif subsequence_length < 30 and (value_so_far | (1 << subsequence_length)) <= k:
                # If adding '1' to the current position does not exceed k, update the value_so_far
                value_so_far |= 1 << subsequence_length
                # Increase subsequence length as this '1' is part of the longest subsequence not exceeding k
                subsequence_length += 1

        # Return the length of the longest subsequence not exceeding the value k
        return subsequence_length
```

Java

```java
class Solution {

    /**
     * Returns the length of the longest subsequence with a decimal value less than or equal to k.
     *
     * @param s The binary string.
     * @param k The upper bound for decimal value of the subsequence.
     * @return The length of the longest subsequence.
     */
    public int longestSubsequence(String s, int k) {
        int longestLength = 0; // The length of the longest valid subsequence
        int decimalValue = 0;  // The decimal value of the considered subsequence

        // Iterate over the string in reverse because the least significant bits
        // can be considered in isolation for the smallest possible addition to the value.
        for (int index = s.length() - 1; index >= 0; --index) {
            // If we find a '0', it doesn't add to the value,
            // so we can always include it in the subsequence
            if (s.charAt(index) == '0') {
                ++longestLength;
            }
            // Only consider '1's if the length of the sequence is less than 30
            // and adding the '1' wouldn't exceed k. We check length < 30
            // because 2^30 exceeds Integer.MAX_VALUE and cannot be represented by int.
            else if (longestLength < 30 && (decimalValue | (1 << longestLength)) <= k) {
                // '|' is the bitwise OR operator. Here we add the value represented by
                // a '1' at the current position to the decimalValue (if it does not exceed k).
                decimalValue |= 1 << longestLength;
                // Increment the length because we've added a '1' to the subsequence.
                ++longestLength;
            }
        }
        return longestLength; // Return the computed length of the longest subsequence
    }
}
```

C++

```cpp
class Solution {
public:
    int longestSubsequence(string s, int k) {
        int answer = 0;          // Used to track the length of the longest subsequence
        int value = 0;           // Used to store the current value of the binary subsequence
        // Loop through the string backwards to consider the least significant bits first
        for (int i = s.size() - 1; i >= 0; --i) {
            if (s[i] == '0') {
                // If the current character is '0', we can always include it without
                // affecting the value of the binary number it represents.
                ++answer;
            } else if (answer < 30 && (value | (1 << answer)) <= k) {
                // Check if the current length of the subsequence is less than 30
                // (Because 2^30 is the first number larger than 10^9, which is outside the constraint for k)
                // and if by setting the current bit to 1 we still get a value less than or equal to k.
                // 1 << answer is the value of setting the current bit (at position 'answer') to 1.
                value |= 1 << answer;  // Set the current bit to 1.
                ++answer;              // Increment the length of the longest subsequence.
            }
            // If the bit is '1' and including it would make the value exceed k or
            // the length of the subsequence is already equal or longer than 30, skip it.
        }
        return answer;  // Return the length of the longest subsequence found.
    }
};
```

TypeScript

```typescript
function longestSubsequence(s: string, k: number): number {
    let longestLength = 0; // Initialize the longest subsequence length to 0
    let currentValue = 0; // Initialize the current value of the binary subsequence to 0

    // Traverse the string from right to left (least significant bit to most significant bit)
    for (let index = s.length - 1; index >= 0; --index) {
        // If the character at the current index is '0', it doesn't contribute to the value
        // of the binary number, so we can safely include it without worrying about the
        // value exceeding `k`, and increment the length of the subsequence.
        if (s[index] === '0') {
            longestLength++;
        } else {
            // If the character is '1' and the length of the subsequence is less than 30
            // (since 2^30 exceeds JavaScript's safe integer for bitwise operations),
            // and the value of including this '1' does not exceed k, include it
            // in the subsequence. The use of `<< k` is crucial since `k` is inclusive.
            if (longestLength < 30 && (currentValue | (1 << longestLength)) <= k) {
                currentValue |= 1 << longestLength; // Include '1' in the currentValue.
                longestLength++; // Increment the subsequence length after including '1'.
            }
        }
    }
    // Return the length of the longest subsequence that satisfies the condition.
    return longestLength;
}
```

```python
class Solution:
    def longestSubsequence(self, s: str, k: int) -> int:
        # Initialize the count of the subsequence and the value of the binary string seen so far
        subsequence_length = value_so_far = 0

        # Iterate over the string in reverse order since the least significant bit contributes less to the overall value
        for character in reversed(s):
            # If the character is '0', it doesn't affect the value but can increase the length of the subsequence
            if character == "0":
                subsequence_length += 1
            # If the character is '1', check if adding this bit exceeds the value of k
            # Since we're looking at the binary string from right to left, we shift 1 left by the current subsequence length
            # The subsequence length represents the binary digit's position (0-indexed)
            # Also note that we perform this check only if subsequence length is less than 30 since 2^30 is the first power of 2 that
            elif subsequence_length < 30 and (value_so_far | (1 << subsequence_length)) <= k:
                # If adding '1' to the current position does not exceed k, update the value_so_far
                value_so_far |= 1 << subsequence_length
                # Increase subsequence length as this '1' is part of the longest subsequence not exceeding k
                subsequence_length += 1

        # Return the length of the longest subsequence not exceeding the value k
        return subsequence_length
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is $O(n)$, where n is the length of the input string `s`. The reason for this is that the code consists of a single loop that traverses the string in reverse, performing a constant amount of work for each character. There are no nested loops, recursive calls, or operations that would increase the complexity beyond linear.

Space Complexity

The space complexity of the code is $O(1)$ because it uses a fixed amount of additional space regardless of the input size. Variables `ans` and `v` are the only variables that occupy space and their space requirement does not scale with the input string's length. The input string `s` and integer `k` are given, and we do not allocate any additional space that is dependent on the size of the input.