1545. Find Kth Bit in Nth Binary String

Medium | Recursion | String

Leetcode Link

Problem Description

The problem requires us to generate a binary string `Sn` using a specified algorithm and then find the `k`th bit in this string for the given `n`. The binary string is constructed following these rules:

- Start with `S1` as "0".
- For any `i > 1`, the string `Si` is formed by concatenating the previous string `Si-1` with "1" and then adding the reversed and inverted version of `Si-1`.

To clarify, the `invert` function flips all bits in a string (0 becomes 1 and 1 becomes 0), and the `reverse` function reverses the order of the characters in the string.

A simple illustration shows that the strings grow exponentially as `n` increases and the process of creating them is recursive.

The challenge is to find the `k`th bit in the constructed string without actually building the entire string, as doing so would be highly inefficient for large `n`.

Intuition

The provided solution navigates through the structure of the generated strings using the defined construction algorithm without having to construct the entire string `Sn`. Here's how it achieves this:

1. The length of the final string `Sn` is determined by a recursive function `calcLength`. Since the length is doubled and incremented by 1 with each iteration, this function keeps track and uses a set to record specific positions that would contain the bit "1".
2. Base cases are checked: if either `n` is 1 or `k` is 1, it's clear from the construction rules that the `k`th bit must be "0".
3. Knowing that the string is symmetric with a "1" in the middle for any `n > 1`, there are three cases:
   - If `k` is at the midpoint, return "1".
   - If `k` is before the midpoint, recursively find the `k`th bit in `Sn-1`.
   - If `k` is after the midpoint, find the corresponding bit before the midpoint in `Sn-1` and invert it.
4. A helper function `r` is used to invert a given bit.

The recursive strategy works by leveraging symmetry and the known structure of the string, avoiding unnecessary computation of the full string and thus maintaining efficiency even for large `n`.

Solution Approach

The solution utilizes a recursive approach, coupled with a set for bookkeeping, to simplify the task of finding the `k`th bit in the sequence `Sn`. The implementation is structured as follows:

- The key to the solution lies in understanding the symmetry of the string `Sn`. Each string `Si` for `i > 1` is a palindrome with a center bit '1'. This property allows us to only focus on half of the string to find the corresponding value.
- The `calcLength` function calculates the length of `Sn` recursively based on the formula that the length of `Si` is twice the length of `Si-1` plus one (`len(Si) = 2 * len(Si-1) + 1`). The function also adds the length `len(Si)` plus one to the set `set` for each iteration, marking positions in the string which result in the bit '1'.
- The `findKthBit` function acts as the primary function to determine the `k`th bit:
  - Base cases check if `k == 1` or `n == 1`, and since we know the first character is always '0', it returns '0'.
  - If the `k`th position has been marked in the set (which was done in `calcLength`), it means that the `k`th bit is '1'.
  - Otherwise, the function determines if the `k`th bit is before or after the midpoint of `Sn`:
    - If `k` is less than the midpoint, recursion is performed to find the `k`th bit in `Sn-1`.
    - If `k` is at or beyond the midpoint:
      - The midpoint bit is always '1', which is directly returned if `k` is exactly the midpoint.
      - For bits after the midpoint, it must find the corresponding bit before the midpoint in `Sn-1` and invert it using the `r` function.
- The `r` function is the inverting function, which takes a bit character and returns its inversion.

This implementation forgoes the need to construct the entire `Sn` string by smartly exploiting its properties, specifically the palindrome characteristic and marking of certain positions with known bits. The algorithm's efficiency arises from avoiding the explicit generation of the binary string and instead working through recursive calls that "simulate" the construction process minimally to find the required bit.

Example Walkthrough

Let's illustrate the solution approach using a small example by constructing the string `S3` and finding the 5th bit of `S3`.

Starting with `S1` as "0" and following the problem rules, we can construct the strings `S2` and `S3`:

- `S1` is "0"
- To get `S2`, we take `S1`, concat a "1", and then add the inverted and reversed `S1`: "0 1 1" (where the space indicates the midpoint)
- For `S3`, the process is repeated with `S2`: "011 1 100" (again, space shows the midpoint)

The length of `S3` is 7 bits, and it is a palindrome with the 4th bit being the middle "1". We can visualize it as: `[0] [1] [1] [1] [0] [0] [0]` where brackets show the positions: 1 2 3 4 5 6 7

We want to find the 5th bit:

1. As per the rule, the midpoint of `S3` is bit 4 and is "1". So we split the string into two halves around the midpoint:
   - First half (left of midpoint): [0] [1] [1]
   - Second half (right of midpoint): [0] [0] [0]
2. We see that the 5th bit is on the right side of the midpoint, which is the reverse and inverted of the first half. So the bit corresponding to the 5th bit is the 3rd bit from the first half (`S2`).
3. Now we invert that bit. The 3rd bit of `S2` is "1" (since `S2` is "011"), so its inversion is "0".
4. We conclude that the 5th bit of `S3` is "0".

This example highlights the application of known symmetry and inversion to determine the value of the `k`th bit without constructing the full string `Sn`, demonstrating the algorithm's efficiency.

Python Solution

```python
class Solution:
    def findKthBit(self, n: int, k: int) -> str:
        if k == 1 or n == 1:
            # The first bit of any sequence is '0' or the sequence of n=1 is '0'
            return '0'

        length_set = set()
        length = self.calculate_length(n, length_set)

        if k in length_set:
            # All the added lengths in the set represent the middle '1'
            return '1'

        if k < length // 2:
            # If k is in the first half, it's the same as the previous sequence
            return self.findKthBit(n - 1, k)
        else:
            # If k is in the second half, invert the (length - k + 1)th bit of the previous sequence
            return self.invert_bit(self.findKthBit(n - 1, length - k + 1))

    def invert_bit(self, bit: str) -> str:
        # Inverts the bit '0' to '1' or '1' to '0'
        return '1' if bit == '0' else '0'

    def calculate_length(self, n: int, length_set: set) -> int:
        if n == 1:
            # Base case: the sequence of length 1 has only a single bit '0'
            return 1

        # Calculate the current length as twice the previous length plus one for the middle '1'
        current_length = 2 * self.calculate_length(n - 1, length_set) + 1
        # Add the position of the middle '1' to the length set
        length_set.add(current_length // 2 + 1)
        return current_length

```

Java Solution

```java
class Solution {
    public char findKthBit(int n, int k) {
        if (k == 1 || n == 1) {
            // The first bit of any sequence is '0' or the sequence of n=1 is '0'
            return '0';
        }
        Set<Integer> lengthSet = new HashSet<>();
        int length = calculateLength(n, lengthSet);
        if (lengthSet.contains(k)) {
            // All the added lengths in the set represent the middle '1'
            return '1';
        }

        if (k < length / 2) {
            // If k is in the first half, it's the same as the previous sequence
            return findKthBit(n - 1, k);
        } else {
            // If k is in the second half, invert the (length - k + 1)th bit of the previous sequence
            return invertBit(findKthBit(n - 1, length - k + 1));
        }
    }

    private char invertBit(char bit) {
        // Inverts the bit '0' to '1' or '1' to '0'
        return (bit == '0') ? '1' : '0';
    }

    private int calculateLength(int n, Set<Integer> lengthSet) {
        if (n == 1) {
            // Base case: the sequence of length 1 has only a single bit '0'
            return 1;
        }

        // Calculate current length as twice the previous length plus one for the middle '1'
        int currentLength = 2 * calculateLength(n - 1, lengthSet) + 1;
        // Add the length plus one (position of middle '1') to the set
        lengthSet.add(currentLength + 1);
        return currentLength;
    }
}

```

C++ Solution

```cpp
#include <unordered_set>
using namespace std;

class Solution {
public:
    char findKthBit(int n, int k) {
        // If k is the first bit or n is 1, the k-th bit is always '0'
        if (k == 1 || n == 1) {
            return '0';
        }

        unordered_set<int> lengthSet;
        int length = calculateLength(n, lengthSet);

        // If k corresponds to the position of the middle '1', return '1'
        if (lengthSet.count(k)) {
            return '1';
        }

        // If k is less than halfway through the string, the k-th bit is the
        // same as in the sequence for n-1
        if (k < length / 2) {
            return findKthBit(n - 1, k);
        } else {
            // If k is in the second half, find the bit at the symmetric position
            // in the sequence for n-1 and invert it
            return invertBit(findKthBit(n - 1, length - k + 1));
        }
    }

private:
    char invertBit(char bit) {
        // Inverts the bit: if '0' returns '1', if '1' returns '0'
        return (bit == '0') ? '1' : '0';
    }

    int calculateLength(int n, unordered_set<int>& lengthSet) {
        // Base case for the recursion: the length of the sequence for n=1
        if (n == 1) {
            return 1;
        }

        // Recursive call to calculate the length for n, which is twice the length
        // of n-1, plus one for the middle '1'
        int currentLength = 2 * calculateLength(n - 1, lengthSet) + 1;

        // Store current length into the set
        lengthSet.insert(currentLength);

        return currentLength;
    }
};

```

Typescript Solution

```typescript
// Use a global set to store the lengths that have a '1' at the center.
const lengthSet = new Set<number>();

function findKthBit(n: number, k: number): string {
    if (k === 1 || n === 1) {
        // The first bit of any S sequence is '0' or the entire sequence for n=1 is '0'
        return '0';
    }
    const length = calculateLength(n);
    if (lengthSet.has(k)) {
        // All the added lengths in the set represent the middle '1'
        return '1';
    }

    if (k < length / 2) {
        // If k is in the first half of the sequence, it's equivalent to the (k)th bit of S_n-1
        return findKthBit(n - 1, k);
    } else {
        // If k is in the second half, invert the (length - k + 1)th bit of S_n-1
        return invertBit(findKthBit(n - 1, length - k + 1));
    }
}

function invertBit(bit: string): string {
    // Inverts the bit: '0' becomes '1' and '1' becomes '0'
    return bit === '0' ? '1' : '0';
}

function calculateLength(n: number): number {
    if (n === 1) {
        // Base case: S_1 has a length of 1
        return 1;
    }

    // Recursive calculation of the length; length of S_n is twice the length of S_n-1 plus 1 for the '1' in the middle
    const currentLength = 2 * calculateLength(n - 1) + 1;
    // Add the position of the '1' in the middle to the set
    lengthSet.add(currentLength / 2 + 1);
    return currentLength;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the `findKthBit` function involves multiple recursive calls. The `calcLength` function is called recursively to calculate the length of the Sn string. Due to the nature of the construction of Sn, where `Sn = Sn-1 + "1" + reverse(invert(Sn-1))`, the length of each subsequent string is double the previous plus one, making it an exponential growth in terms of n.

Let's denote the time complexity of `calcLength` as T(n). Therefore, we have:

`T(n) = T(n-1) + O(1)`

Since this recurses n times, and each recursion is just adding a constant amount of work (besides the recursive call), this simplifies to a linear complexity in terms of n:

`T(n) = O(n)`

Next, analyzing the recursive calls in the main `findKthBit` function, we notice that in the worst case, it can end up recursively calling itself by decreasing n by one each time until n reaches 1. This gives us, in the worst case, a recursion depth of n.

However, we also need to consider that not every recursive call goes all the way down to n=1 due to the early return conditions, so not every call will branch out fully. The recursion only happens to find the kth bit before the midpoint. After the midpoint, the work reduces due to the reuse of the computation as we call `findKthBit(n - 1, len - k + 1)`.

As a result, the time complexity is not straightforward to calculate, but we can approximate it by considering that at each level of recursion, the size of the problem is roughly halved (similar to a binary search). The worst-case time complexity thus approximates to:

`O(n * log(len))`

where `len` is the length of the final string Sn which is `O(2^n)`.

Combining all the recursive calls and operations within each call, the total time complexity roughly approximates to:

`O(n^2)`

This is considering that each recursive call to `findKthBit` has a cost that doubles each time with a diminishing number of calls due to halving of k as well.

Space Complexity

The space complexity is governed by both the recursive call stack and the space taken by the HashSet to store specific lengths at each level of recursion.

The maximum depth of the recursion stack is n, for the recursive calls to `findKthBit`.

The HashSet grows in size linearly with n, since a new length is added for each level of recursion in `calcLength`.

Therefore, the space complexity is:

`O(n) + O(n) = O(n)`

Overall, the space complexity of the function is linear with respect to n due to the recursion stack and the HashSet storage:

`O(n)`