2802. Find The K-th Lucky Number

Medium

Leetcode Link

Problem Description

In this problem, we have two digits that are considered "lucky" – 4 and 7. A number is termed as a "lucky number" if and only if every digit in the number is either a 4 or a 7. We are given a task to find the k^th^ lucky number when all the lucky numbers are sorted in increasing order. We need to provide this lucky number as a string.

For instance, the first few lucky numbers are: 1st lucky number is "4" 2nd lucky number is "7" 3rd lucky number is "44" (and so on)

The challenge here is to calculate the k^th^ lucky number without generating all previous lucky numbers.

Intuition

To solve this problem, we can draw an analogy from binary numbers. If we replace every binary digit 0 with 4 and every binary digit 1 with 7, we get a system where each binary number corresponds to a lucky number. For example, binary 0 (which in our modified system would be "4"), binary 1 ("7"), binary 10 ("44"), binary 11 ("47") and so on.

To find the k^th^ lucky number, we follow a similar approach as we would finding the k^th^ number in binary. However, unlike a standard binary system where each position can be 0 or 1, we only have two digits 4 and 7, representing the two possible states at each position of a lucky number. We determine the length of our lucky number (in digits) by finding the smallest number `n` such that k is less than 2^n. This represents the level at which the k^th^ lucky number exists if we were to visualize all lucky numbers in a binary tree form.

Once we have the number of digits `n`, we can construct the lucky number from most significant digit to least significant digit, by checking if k is in the lower half (which would correspond to '4') or the upper half (which would correspond to '7') of the values for that digit's position—this is akin to deciding between 0 and 1 in binary representation. If it's in the upper half, we know this digit is '7', and we subtract the size of the lower half (2^(n - 1)) from k to continue finding the rest of the digits. If it's in the lower half, we simply assign '4' to this current digit. We iterate this process until we have all `n` digits of our lucky number.

By approaching the problem in this manner, we efficiently calculate the k^th^ lucky number without the need to list or check all previous lucky numbers.

Solution Approach

The solution implements the intuition discussed above with a focus on optimizing the process. Here's the detailed explanation of the code step-by-step:

1. We start by initializing `n` to 1. This `n` variable will eventually indicate the number of digits in our k^th^ lucky number.

2. The first `while` loop checks how many digits the k^th^ lucky number will have. It works under the principle that there are `2^n` lucky numbers with `n` digits (`2^n - 1` possible combinations plus the all-4s combination). So, if `k` is greater than `2^n`, `k` is not within the range of lucky numbers that have `n` digits. Therefore, we subtract `2^n` from `k` and increment `n` by 1, and iterate until `k` is less than or equal to `2^n`. This locates the correct 'level' of the binary-tree-like structure where our number sits.

3. Once we have the number of digits `n`, we initialize an empty list `ans`, which will store each digit of the k^th^ lucky number as we compute it.

4. The second `while` loop executes `n` times, decreasing `n` with each iteration. Each iteration of the loop decides one digit of the lucky number, starting from most significant to least significant. The conditional within this loop `if k <= 1 << n` is checking whether `k` fits in the lower half of the range for the current digit (which would correspond to '4'). If so, '4' is appended to `ans`. Otherwise, '7' is appended, and `k` is decremented by `2^(n-1)` or `1 << n` to reflect that we're now looking in the upper half of the range for the next digit.

5. After the loop completes, `ans` holds the digits of our k^th^ lucky number in order. We finally return the number as a string by joining each element of the list with `"".join(ans)`.

This implementation uses a list (`ans`) as the primary data structure to build the lucky number and follows a binary search pattern to decide each digit of the lucky number, improving the efficiency by avoiding unnecessary computation that would come from generating all previous lucky numbers.

Example Walkthrough

Let's illustrate the solution approach using an example. We want to find the 5th lucky number.

1. Initialize `n` to 1, because every lucky number has at least one digit.

2. Begin the first `while` loop to find the number of digits our 5th lucky number will have. We know there are `2^n` lucky numbers for each digit length. Initially, `n` is 1, and `2^n` is 2, which is less than 5. So we increment `n` to 2, and now `2^n` is 4, which is still less than 5. Incrementing `n` once more gives us `2^n` as 8, which is greater than 5. So, we stop here and know that our 5th lucky number has 3 digits.

3. We now start with `k = 5` and `n = 3`. Initialize an empty list `ans` to store the digits.

4. Enter the second `while` loop, which will run three times (since our lucky number has 3 digits):

   - In the first iteration, `1 << n` equals `1 << 3` which equals 8. This is greater than `k` (5), so `k` is in the lower half. We append '4' to `ans` and do not change `k`.
   - In the second iteration, we decrement `n` to 2. Now `1 << n` is 4, which is less than `k`. Hence, `k` is in the upper half, and we need to subtract `1 << (n-1)` (which is 2) from `k`. `k` becomes 3, and we append '7' to `ans`.
   - In the third and final iteration, `n` is decremented to 1, making `1 << n` equal to 2. `k` is 3, which is greater; therefore, we are again in the upper half. We subtract `1 << (n-1)` from `k`, which is 1, making `k` now equal to 2. We append '7' to `ans`.
5. The second `while` loop completes and our list `ans` has the values `['4', '7', '7']`.

6. We join these to form the 5th lucky number: `"".join(ans)` equals "477".

Therefore, the 5th lucky number is "477". This example clearly shows how the binary-like approach works by deciding one digit at a time, based on whether `k` is in the lower or upper half of the range for that digit.

Python Solution

```python
class Solution:
    def kth_lucky_number(self, k: int) -> str:
        # Initialize the number of digits to be considered to 1
        num_digits = 1

        # Find the number of digits the kth lucky number must have
        while k > (1 << num_digits):
            # Decrease k by the count of lucky numbers with num_digits digits
            k -= 1 << num_digits
            # Increment the digit count as we're moving on to numbers with more digits
            num_digits += 1

        # Initialize the answer as an empty list to hold the digits
        answer_digits = []

        # Construct the kth lucky number by going through each digit place
        while num_digits:
            num_digits -= 1  # Decrement digits count as we build the number from high to low
            if k <= (1 << num_digits):
                # If k is within the range of the first half, append 4, as it is the smaller digit
                answer_digits.append("4")
            else:
                # If k is in the second half, append 7 and adjust k accordingly
                answer_digits.append("7")
                k -= 1 << num_digits  # Decrement k as we have used one of the 7s

        # Join all the individual digits to form the kth lucky number and return it
        return "".join(answer_digits)

```

Java Solution

```java
class Solution {
    public String kthLuckyNumber(int k) {
        // 'n' represents the number of digits in the lucky number
        int n = 1;

        // Find the number of digits in the kth lucky number by comparing k with powers of 2
        while (k > (1 << n)) {
            k -= (1 << n);
            ++n;
        }

        // Build the kth lucky number starting with the most significant digit
        StringBuilder ans = new StringBuilder();
        while (n > 0) {
            // Check the kth bit of n to decide whether to append '4' or '7'
            // If k is in the first half of the range for the current digit length, append '4'
            if (k <= (1 << (n - 1))) {
                ans.append('4');
            } else {
                // If k is in the second half, append '7' and update k
                ans.append('7');
                k -= (1 << (n - 1));
            }
            n--;
        }

        return ans.toString();
    }
}

```

C++ Solution

```cpp
#include <string>

class Solution {
public:
    // Function to find the kth lucky number where lucky numbers are
    // positive integers whose decimal representation contains only the digits 4 and 7.
    std::string kthLuckyNumber(int k) {
        // Start counting digits from 1
        int numDigits = 1;

        // Find the number of digits in the kth lucky number by using powers of 2
        // Each additional digit doubles the count of lucky numbers available
        while (k > (1 << numDigits)) {
            k -= 1 << numDigits; // Subtract the number of numbers with `numDigits` digits
            ++numDigits;         // Move to the next digit length
        }

        // Initialize an empty string to store the kth lucky number
        std::string luckyNumber;

        // Construct the lucky number digit by digit
        while (numDigits--) {
            // If the remaining k is less or equal to the number of lucky numbers with the current number of digits
            if (k <= (1 << numDigits)) {
                luckyNumber.push_back('4'); // A '4' is appended when k is in the first half within the current digit's range
            } else {
                luckyNumber.push_back('7'); // Otherwise, a '7' is appended
                k -= 1 << numDigits;       // And we adjust k to reflect that we're now considering the second half range
            }
        }

        // Return the constructed lucky number as a string
        return luckyNumber;
    }
};

```

Typescript Solution

```typescript
function kthLuckyNumber(k: number): string {
    // Initialize counter 'n' which represents the number of binary digits.
    let counter = 1;

    // As long as 'k' is greater than '2^n', decrease 'k' by '2^n' and increment 'n'.
    while (k > (1 << counter)) {
        k -= 1 << counter;
        ++counter;
    }

    // Initialize an array 'luckyNumbers' to store the lucky number digits.
    const luckyNumbers: string[] = [];

    // Build the lucky number by determining if each digit is a '4' or a '7'.
    while (counter-- > 0) {
        if (k <= (1 << counter)) {
            // If 'k' is less than or equal to '2^n', the digit is '4'.
            luckyNumbers.push('4');
        } else {
            // If 'k' is greater, the digit is '7' and we adjust 'k' accordingly.
            luckyNumbers.push('7');
            k -= 1 << counter;
        }
    }

    // Join the digits and return the resulting lucky number as a string.
    return luckyNumbers.join('');
}

```

Time and Space Complexity

The provided code calculates the k-th lucky number where lucky numbers are composed only of the digits `4` and `7`.

Time Complexity

The main component of the code involves a while loop that runs until `k` is less than `1 << n`, where `n` starts at `1` and gets incremented. Inside the loop, `k` is decremented by `1 << n`. After finding the value of `n` such that `k` is within bounds, the code executes another while loop that decreases `n` on each iteration and constructs the lucky number. In the worst-case scenario, `n` will be proportional to the number of digits in the k-th lucky number.

The actual number of loop iterations is related to the bit length of the input `k`. Therefore, the time complexity is determined by the length of the binary representation of `k`, which can be described as `O(log k)`.

The construction of the lucky number (`ans.append("4")` or `ans.append("7")`) depends on the number of digits `n`. Since the loops' maximum number of iterations is equal to the number of digits, the overall time complexity of constructing the lucky number is `O(log k)`.

Space Complexity

The extra space used in the solution is allocated for the list `ans` to construct the return string. In the worst case, the length of `ans` will be equal to `n`, the number of digits in the k-th lucky number. Since the value of `n` is dependent on the logarithm of `k`, the space complexity is `O(log k)` for storing the resulting string.

The code does not use any other data structures that grow with the input size, so the overall space complexity remains `O(log k)`.