K-Sparse Binary Count

codility
WE TEST CODERS

Another month has passed and it is time to reveal how the Codility challenge codenamed Xi can be solved. You can still give it a try, but no certificate will be awarded.

In the Xi certificate problem, we consider non-negative integers in whose binary representation any two consecutive 1s are separated by at least K 0s, for some given integer K. Such positive integers are called K-*sparse*. The problem is to compute the number of K-sparse integers in a range $[A..B]$ for given positive integers A and B (modulo $1\,000\,000\,007$). A and B can be as large as $10^{300\,000}$.

For the sake of simplicity, from now on we will perform all the computations modulo $1\,000\,000\,007$ without explicit further notice. The first observation to simplify the problem is that the number of K-sparse integers in the range $[A..B]$ is equal to the number of K-sparse integers smaller than $B+1$ minus the number of K-sparse integers smaller than A. So, the problem reduces to finding the number of K-sparse integers smaller than a given positive integer N. Let `increase` be a simple procedure to increase a binary representation of an integer (stored in a string) by 1, and `below` be a function to find the number of K-sparse integers smaller than a given positive integer. A relevant piece of code (in Python) might be as follows:

```
ModP = 1000000007;

sparse_binary_count(A, B, K):
    return (below(increase(B), K) - below(A, K) + ModP) % ModP
```

We will show how to solve this problem for K-sparse values of N. But what if N is not K-sparse? In such a case, we can increase N to the smallest K-sparse integer larger than N. If N is not K-sparse, it means that there are at least two consecutive 1s separated by fewer than K 0s. We should focus on the most significant such pair of 1s, as changing the lower bits wouldn't produce a K-sparse number. Moreover, if we are looking for a K-sparse number larger than N, we have to shift the more significant of the two 1s up. On the other hand, if we do shift it up, even by one position, then all the lower bits can be reset to 0 and the resulting number will still be larger than N. Would it be K-sparse? Not necessarily. There is one tricky point: If the more significant of the two 1s is separated by exactly K 0s from the next more significant 1, then, after shifting it up, the distance between these two 1s becomes smaller than K. Therefore, we should focus on the most significant and longest sequence of bits of the form:

$$1\ K \text{ times } 0\ 1\ K \text{ times } 0\ 1 \ldots 1\ K \text{ times } 0\ 1 \text{ fewer than } K \text{ 0s } 1$$

The smallest K-sparse integer larger than N can be obtained by shifting the most significant of these 1s one position up and setting all lower bits to 0. The following piece of code in Python implements this and calls a procedure `below_k_sparse`, which solves the problem for K-sparse values of N:

```
def below(N, K):
    N = '0' + N
    l = len(N)
    c = K+1
    i = 0
    j = 0
    while (i < l):
        if (N[i] == '1'):
            if (c < K):
                break
            else:
                if (c > K):
                    j = i
            c = 0
        else:
            c += 1
        i += 1
    if (i < l):
        N = N[:j-1] + '1' + '0'*(l-j)
    return below_k_sparse(N, K)
```

So, what is the number of K-sparse numbers smaller than a given K-sparse number N? For the sake of simplicity, let us include zero among K-sparse numbers. Let the most significant 1 in a binary representation of N be at position representing 2^I. In other words, $2^I \leqslant N < 2^{I+1}$. K-sparse numbers smaller than N can be divided into two groups:

1. K-sparse numbers smaller than 2^I: let us denote the number of such K-sparse numbers by $F[I]$;
2. K-sparse numbers smaller than N but not smaller than 2^I: the binary representations of such numbers contain 1 at the position representing 2^I and 0s at the positions representing $2^{I-1}, 2^{I-2}, \ldots, 2^{I-K}$; the remaining bits represent a number smaller than $N - 2^I$.

Hence, the result is a sum of values of $F[I]$ for values of I in which the binary representation of N contains 1 at the position representing 2^I. The following code in Python implements this:

```
def below_k_sparse(N, K):
    l = len(N)
    res = 0
    for i in xrange(l):
        if (N[i] == '1'):
            res = (res + F[l-1-i]) % ModP
    return res
```

What remains is to calculate the values of $F[I]$. Again, K-sparse numbers smaller than 2^I can be divided into two groups:

1. numbers smaller than 2^{I-1}: there are $F[I-1]$ of them; and
2. numbers smaller than 2^I but not smaller than 2^{I-1}: their binary representations contain 1 at the position representing 2^{I-1} and 0s at the positions representing $2^{I-2}, 2^{I-3}, \ldots, 2^{I-K-1}$; hence, there are $F[I-K-1]$ of them.

From this, we obtain the following recursive equation defining $F[I]$:

$$\begin{array}{rcll} F[I] & = & F[I-1]+F[I-K-1] & \text{for } I \geqslant 0 \\ F[I] & = & 1 & \text{for} I < 0 \end{array}$$

This leads to the following code for precomputing values of $F[I]$, which can be added to the procedure `sparse_binary_count`:

```
1        F = [1]*(len(B)+2)
2        for i in xrange(1,len(F)):
3            if (i > K):
4                F[i] = (F[i-1] + F[i-K-1]) % ModP
5            else:
6                F[i] = (F[i-1] + 1) % ModP
7
8      def below_k_sparse(N, K):
9        l = len(N)
10       res = 0
11       for i in xrange(l):
12           if (N[i] == '1'):
13               res = (res + F[l-1-i]) % ModP
14       return res
```

What remains is to calculate the values of $F[I]$. Again, K-sparse numbers smaller than 2^I can be divided into two groups:

1. numbers smaller than 2^{I-1}: there are $F[I-1]$ of them; and
2. numbers smaller than 2^I but not smaller than 2^{I-1}: their binary representations contain 1 at the position representing 2^{I-1} and 0s at the positions representing $2^{I-2}, 2^{I-3}, \ldots, 2^{I-K-1}$; hence, there are $F[I-K-1]$ of them.

From this, we obtain the following recursive equation defining $F[I]$:

$$\begin{array}{rcll} F[I] & = & F[I-1]+F[I-K-1] & \text{for } I \geqslant 0 \\ F[I] & = & 1 & \text{for } I < 0 \end{array}$$

This leads to the following code for precomputing values of $F[I]$, which can be added to the procedure `sparse_binary_count`:

```
1        F = [1]*(len(B)+2)
2        for i in xrange(1,len(F)):
3            if (i > K):
4                F[i] = (F[i-1] + F[i-K-1]) % ModP
5            else:
6                F[i] = (F[i-1] + 1) % ModP
```

Note that, for $K = 2$, $F[I]$ are Fibonacci numbers.

Each step of the computation can be performed in a length of time proportional to the length of the processed string. Strings representing numbers A and B are of lengths $O(\log A)$ and $O(\log B)$ respectively. Hence, the overall time complexity of the presented solution is $O(\max(\log A, \log B)) = O(\log(A+B)) = O(\log B)$ time.