Binary Subsequences

-editorial-

Author: Oncescu Costin
Preparation: Oncescu Costin

The first thing we should think about is how to solve the initial problem (that for which we are supposed to help in building test data): that is, for a given string containing 0s and 1s, to count how many different non-empty subsequences it has. We need an as easy as possible method that we can later on use in solving the actual task. There are multiple ways to count the different subsequences, among which, we'll explain the easiest one:

Let's say we have a string S, with length denoted as |S| and i^{th} digit S[i], S being 1-indexed. We'll use dynamic programming: let dp[i][j] be the number of distinct non-empty subsequences of the prefix of length i of S, which end in digit j.

Of course dp[0][0] and dp[0][1] are both 0, as we should count the non-empty subsequences ending in 0, respectively 1 of an empty string.

dp[i][S[i] ^ 1] is equal to dp[i - 1][S[i] ^ 1] as there is no new subsequence ending in the digit different from S[i] (as S[i] is the only novelty, the only difference from the prefix i - 1)

In order to compute how many subsequences end in a digit equal to S[i], we could simply assume that i is the last position in the subsequence, as we shouldn't miss any subsequence this way (it's not like position i could help us in any other way), so dp[i][S[i]] becomes equal to the number of different subsequences of the prefix of length i - 1 of S, which is dp[i - 1][0] + dp[i - 1][1] + 1 (1 is because, as you noticed, we should count the empty subsequence too, as the final subsequence will be obtained by adding S[i] to the end of these subsequences, so the subsequences will be non-empty anyway).

Once we've computed this dynamic programming, the number of different subsequences is dp[|S|][0] + dp[|S|][1].

30.1 points

Now, we're already ready to score some points. We'll get 30.1 points by simply generating all strings of length at most 16 and computing for each of them the number of different non-empty subsequences in the way described above. This way, we'll be able to answer all the questions of type 2 for K <= 2000 (as you can find out by running this brute force).

43 points

We can no longer improve the brute force as we cannot find out all the strings for fixed K(for example string 11..1 has length K and should be counted), so we cannot count all of them. We need some observations. Hopefully, by inspecting the recurrence, we find out that dp[i][] depends only on dp[i - 1][]. That's why we should try to store the values dp[size][0] and dp[size][1] for some string of length size.

Also, dp[i][0] + dp[i][1] >= dp[i - 1][0] + dp[i - 1][1] (we just extend the set of subsequences by increasing the length of the prefix). As we want dp[|S|][0] + dp[|S|][1] to be K, we have dp[i][0] + dp[i][1] <= K for each i.

By the above 2 observations, we come up with the idea of using a graph of states. We have an oriented edge from a state to some other state with a label of 0 or 1. We'll define a state as a pair of non-negative integers with sum at most K (dp[something][0] and dp[something][1]). We trace an edge between state (a, b) with label 0 to state (a + b + 1, b) and an edge from (a, b) with label 1 to state (a, a + b + 1). This way, if we begin with a state (0, 0) and start moving along the edges corresponding to the current digit, we end up in a state (dp[|S|][0], dp[|S|][1]). We'll denote this path as the path of the string S in our graph. Also, this graph has only O(K^2) nodes and O(K^2) edges due to observation 2.

Now we get the idea of using 2 other dynamic programming: minL[i][j] the minimum length of a string whose path ends in (i, j) and cnt[i][j] the number of strings with paths ending in (i, j).

We can get a forward kind of recurrence: we add cnt[i][j] to both cnt[i + j + 1][j] and cnt[i][i + j + 1] and we update minL[i + j + 1][j] with the minimum of how it was before and minL[i][j] + 1 and the same for minL[i][i + j + 1].

This will run in O(maxK^2) and will work without any problems for maxK <= 2000.

82 points

In order to improve the previous solution, we must first notice that, if we're interested in only one K (or generally few values of K), most of the states are useless. Actually, only K + 1 states could be final states, those with a form of (t, K - t) for t some non-negative integer less than K + 1. That gives us the idea of using memoization.

If we try to reverse the recurrence we used in the previous state (that is, doing it backward in order to make the memoization work), we can observe that a state just seems to have 2 possible previous states: (i - j - 1, j) and (i, j - i - 1), but, actually, at most one of them is a valid state! At least one of them will have one negative term depending on whether i is greater than j or not. What does that mean? It means our oriented graph is actually a tree!!! Now, we don't need to use memoization anymore. We can simply fix a state (t, K - t), walk through the fathers until we reach the root (that is state (0, 0)) and write down the labels on the path in reversed order, this way generating all the possible strings.

Let's notice that the answer is rather small (at most K + 1) there is no need for it to be computed modulo 1.000.000.007. Even though it may seem that the worst case is K^2, this algorithm works much better in practice (we tested it on all the possible tests).

100 points

We already have a very good solution, so it doesn't make any sense to try a new approach. We just have to make the last one better. What we did was basically: go from all pairs (t, K - t) up to (0, 0) by doing either i = i - j - 1 or j = j - i - 1 depending on whether i > j or i < j. Does it look familiar to you? It sounds a lot like gcd, doesn't it? That's why we get the idea of

doing more subtractions at once, that is doing i = i - (j + 1) * k where k is [i / (j + 1)] for i > j and analogous for i < j. The time complexity of an iteration is the same as that of gcd: logK and, thus, the total time complexity is KlogK.