App Versioning

With the advancement of AI, writing code is now much faster than ever before. Many AI-pilled companies estimate that the number of lines of code they will produce in 2026 will be at least 20 times that of 2025. More code means more releases, and app versioning will become much more complicated. To prepare for this future, you are building a new versioning system.

A *version* is a sequence of one or more non-negative integers, called *components*, separated by dots. For example, `1.2.3`, `2.1` and `10` are versions. Each component must be written in decimal **without leading zeros**. In particular, `0` is a valid component, but `02` is not, so `1.02.3` is not a valid version.

Unlike the versions used by humans today, which usually have at most four components, versions in your system can have hundreds of thousands of components. This is obviously not an issue for a system whose target audience is AI agents.

Two versions are compared component by component, from left to right:

- At the first position where the components of the two versions are different, the version whose component is smaller **as an integer** is the smaller version. For example, `1.2.3` < `1.2.4`, `1.9` < `1.10` and `1.3` < `2.1`.
- If there is no such position, i.e., one version is a prefix of the other, the version with fewer components is the smaller version. For example, `1.3` < `1.3.1` and `1.3` < `1.3.0`.

Unfortunately, when testing your system, you found two versions where the dots have been lost. More precisely, you have two strings of digits A and B. You need to insert dots into A and into B to turn them back into versions V_A and V_B. A dot can be inserted between any two adjacent digits, and at most one dot can be inserted between each pair of adjacent digits. Inserting no dots at all is also allowed. The digits themselves cannot be changed, removed or reordered.

Count the number of ways to insert dots such that both V_A and V_B are valid versions and $V_A < V_B$. Two ways are different if the resulting pairs (V_A, V_B) are different. Since the answer can be very large, print it modulo $10^9 + 7$.

Input

Each test contains multiple test cases. The first line contains the number of test cases τ ($1 \le \tau \le 5 \times 10^5$). The description of the test cases follows.

Each test case consists of a single line containing two strings A and B ($1 \le |A|, |B| \le 5 \cdot 10^5$), separated by a space. Both strings consist only of digits `0`–`9`.

It is guaranteed that the sum of $|A| + |B|$ over all test cases does not exceed 10^6.

Output

For each test case, print the number of ways modulo $10^9 + 7$.

Sample Input 1	Sample Output 1
3 12 13 102 11 0 00	3 3 1

Sample Explanation

In the first test case, V_A can be `12` or `1.2`, and V_B can be `13` or `1.3`. There are 3 ways such that $V_A < V_B$: `12` $<$ `13`, `1.2` $<$ `13` and `1.2` $<$ `1.3`.

In the second test case, V_A can be `102`, `10.2` or `1.0.2` (but not `1.02`, which has a leading zero), and V_B can be `11` or `1.1`. There are 3 ways such that $V_A < V_B$: `10.2` $<$ `11`, `1.0.2` $<$ `11` and `1.0.2` $<$ `1.1`.

In the third test case, the only valid way is $V_A =$ `0` and $V_B =$ `0.0`, and indeed `0` $<$ `0.0`.