ACM-ICPC Indonesia National Contest 2016

Problem A

String Matching

Time Limit: 1 second

String matching is an important problem where we want to find whether a string (or sometimes called *pattern*) can be found within a larger string. In other words, whether a pattern P exists as a substring of string S. Naively, we can solve this problem by iterating through all possible substrings of S in $O(|S|^2)$ time, but usually the length of S and P are quite large such that a quadratic time-complexity solution like this is not fast enough. This problem has been studied extensively (e.g., in computational biology) and many algorithms have been proposed to solve this problem. You may have heard (or better, familiar with) some of these well-known algorithms: Knuth-Morris-Prat (KMP) algorithm, Rabin-Karp algorithm, and Aho-Corasick algorithm. These algorithms can solve the string matching problem in linear time-complexity.

Red is the founder of a new start-up game developer company. When Red developed his first game with his team, Red found the exact problem which he has learnt back in his undergraduate study, the string matching problem. However, being an ignorant person, Red did not pay much attention on this subject and managed to barely pass the exam. Red delegated this problem to one of his new programmer which also is a fresh-graduate, with the hope that this new guy still remember the linear time-complexity solution for this problem.

Unfortunately, instead of implementing a (correct) string matching algorithm, this new guy implemented a wrong one:

1. Let P be the pattern and S be the string.
2. If $|P| > |S|$, then output "NO" and terminate.
3. If P is a prefix of S, then output "YES" and terminate; otherwise
4. Let x be the smallest index where $P_x \neq S_x$,
5. Update S as suffix of S starting from index x + 1, then back to step 2.

Knowing that his solution is linear time-complexity, this new guy is confident that this solution works. However, of course you, being a competitive programmer, realize that this solution is simply wrong.

For example, let P = "ABABC" and S = "ABABABCABA".

First round:

```
S: ABABABCABA
P: ABABC
```

P is not a prefix of S and $P_4 \neq S_4$ (x = 4 in 0-based index), so update S as suffix of S starting from index 4 + 1 (= 5): ABABABCABA → BCABA.

Second round:

```
S: BCABA
P: ABABC
```

P is not a prefix of S and $P_0 \neq S_0$ (x = 0 in 0-based index), so update S as suffix of S starting from index 0 + 1 (= 1): BCABA → CABA.

Third round:

```
S: CABA
P: ABABC
```

|S| is lower than |P| (4 < 5), so output "NO" and terminate.

Therefore, this algorithm will produce "NO" output for P = ABABC and S = ABABABCABA, even though we can find P in S: AB(ABABC)ABA.

You want to analyze the damages caused by this algorithm, so, as the first step, you should reproduce this algorithm. Given a pattern P and a string S, output whether P exists in S according to the aforementioned algorithm.

Input

The first line of input contains an integer T (T ≤ 100) denoting the number of cases. Each case contains two string P and S separated by a single space denoting the pattern and the string, respectively. P and S consist of uppercase alphabetic characters only (A-Z) and have length between 1 and 20,000 characters.

Output

For each case, output in a line "`Case #X: Y`" where `X` is the case number, starts from 1, and `Y` is the output of the algorithm described in the problem statement (YES or NO).

Sample Input

```
5
ABABC ABABABCABA
ABC ABABCAB
ICPC ACMICPCJAKARTA
BABAT BABABABABAT
INC INC
```

Output for Sample Input

```
Case #1: NO
Case #2: NO
Case #3: YES
Case #4: YES
Case #5: YES
```