**19I510 Design and Analysis of Algorithms**

**Exercise 4 – String Matching Algorithms**

1. 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 programmers who is also a fresh-graduate, with the hope that this new guy still remembers 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 Px ≠ Sx,
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 P4 ≠ S4 (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: **A**BABC | P is not a prefix of S and P0 ≠ S0 (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 Format**It 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 Format**For each case, output either YES or NO

**Sample Input**

ABABC ABABABCABA

**Sample Output**  
NO

1. Implement a linear time correct algorithm O(n) for finding the pattern in the given text.

Hint: Try to find any longest proper prefix that is also a suffix, as in the LPS (longest proper prefix that is also a suffix) and construct a table, called the LPS table.

In the LPS table, map every character of the pattern to a value.

**Input Format**It 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 Format**For each case, output either YES or NO

**Sample Input**

ABABC ABABABCABA

**Sample Output**

YES

1. **Z algorithm (Linear time pattern searching Algorithm)**

This algorithm finds all occurrences of a pattern in a text in linear time. Let length of text be n and of pattern be m, then total time taken is O(m + n) with linear space complexity.

**Text will be Pattern$Text or PatternText**

For a string str[0..n-1], Z array is of same length as string. An element Z[i] of Z array stores length of the longest substring starting from str[i] which is also a prefix of str[0..n-1]. The first entry of Z array is meaning less as complete string is always prefix of itself.

Example:

Index 0 1 2 3 4 5 6 7 8 9 10 11

Text a a b c a a b x a a a z

Z values X 1 0 0 3 1 0 0 2 2 1 0

**More Examples:**

str = "aaaaaa"

Z[] = {x, 5, 4, 3, 2, 1}

str = "aabaacd"

Z[] = {x, 1, 0, 2, 1, 0, 0}

str = "abababab"

Z[] = {x, 0, 6, 0, 4, 0, 2, 0}

**How is Z array helpful in Searching Pattern in Linear time?**   
The idea is to concatenate pattern and text, and create a string “P$T” where P is pattern, $ is a special character should not be present in pattern and text, and T is text. Build the Z array for concatenated string. In Z array, if Z value at any point is equal to pattern length, then pattern is present at that point.

Example:

Pattern P = "aab", Text T = "baabaa"

The concatenated string is = "aab$baabaa"

Z array for above concatenated string is {x, 1, 0, 0, 0,

**3**, 1, 0, 2, 1}.

Since length of pattern is 3, the value 3 in Z array

indicates presence of pattern.