Append and Delete

locked

by zemen

Problem | Submissions | Leaderboard | Discussions | Editorial

Statistics

Difficulty: Easy

Time Complexity: O(|s| + |

Publish D

03 2016

Editorial by zemen

If we can obtain string t by performing x operations, then we can also obtain it in $x + 2 \cdot i$ operations for any $i \geq 1$ by repeatedly deleting and re-appending the last character. Thus, it's sufficient to find some minimal even and minimal odd v such that t can be obtained in v operations.

We denote the length of the longest common prefix of s and t to be p. Because we know that the first p characters in s and t are the same in both strings, that tells us the minimum number of operations we must perform is $d = |s| + |t| - 2 \cdot p$. If d is of the same parity as k, we can simply check that $k \geq d$.

If d and k do not have the same parity, the only way we can change the parity of the length of the string after k steps is to perform a deletion on an empty word. This means we must erase s, perform an additional delete operation, and then append each character in t for a total of $f = |s| + |t| + 1$ operations. Then we print `Yes` if and only if $k \geq f$.

Set by zemen

Problem Setter's code :

```
#include <bits/stdc++.h>
using namespace std;
#define sz(x) ((int) (x).size())

int main() {
    #ifdef LOCAL
    assert(freopen("test.in", "r", stdin));
    #endif
    string s, t;
    int k;
    cin >> s >> t >> k;
    int p = 0;
    while (p < min(sz(s), sz(t)) && s[p] == t[p])
        ++p;
    int vmin;
    if (k % 2 == (sz(s) + sz(t)) % 2)
        vmin = sz(s) + sz(t) - 2 * p;
    else
        vmin = sz(s) + sz(t) + 1;
    if (k < vmin)
        cout << "No\n";
    else
        cout << "Yes\n";
}
```

Tested by AllisonP

Problem Tester's code :

```
import java.util.*;

class Solution {

    public static boolean solve(char[] s, char[] t, int k) {
```

```
            if (s[i] != t[i]) {
                break;
            }
        }

        // The strings are the same
        if (i == s.length && s.length == t.length) {
            // if k is odd, there will always be 1 operation left over
            // else, you can delete and re-append last character to use up k opera
tions
            return ((k & 1) == 1) ? false : true;
        }

        // Else
        // Reduce k by number of necessary deletions and insertions
        k = k - (s.length - i) - (t.length - i);

        // If k < 0 or there is an odd number of operations left over, false
        // else we need exactly k operations or the number of extra ops is even, t
rue
        return (k < 0 || (k & 1) == 1) ? false : true;

    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String s = in.next();
        String t = in.next();
        int k = in.nextInt();
        in.close();

        System.out.println( (solve(s.toCharArray(), t.toCharArray(), k))
                            ? "Yes"
                            : "No"
                            );
    }
}
```