One Edit Distance

Given two strings S and T, determine if they are both one edit distance apart.

Solution 1

```
public boolean isOneEditDistance(String s, String t) {
    if(Math.abs(s.length()-t.length()) > 1) return false;
    if(s.length() == t.length()) return isOneModify(s,t);
    if(s.length() > t.length()) return isOneDel(s,t);
    return isOneDel(t,s);
}
public boolean isOneDel(String s,String t){
    for(int i=0,j=0;i<s.length() && j<t.length();i++,j++){
        if(s.charAt(i) != t.charAt(j)){
            return s.substring(i+1).equals(t.substring(j));
        }
    }
    return true;
}
public boolean isOneModify(String s,String t){
    int diff =0;
    for(int i=0;i<s.length();i++){
        if(s.charAt(i) != t.charAt(i)) diff++;
    }
    return diff==1;
}
```

written by zq670067 original link here

Solution 2

Java:

```
for (int i = 0; i < Math.min(s.length(), t.length()); i++) {
    if (s.charAt(i) != t.charAt(i)) {
        return s.substring(i + (s.length() >= t.length() ? 1 : 0)).equals(t.subst
ring(i + (s.length() <= t.length() ? 1 : 0)));
    }
}
return Math.abs(s.length() - t.length()) == 1;
```

C++:

```
    for (int i = 0; i < min(s.size(), t.size()); i++) {
        if (s.at(i) != t.at(i)) {
            return s.substr(i + (s.size() >= t.size() ? 1 : 0)).compare(t.substr(
i + (s.size() <= t.size() ? 1 : 0))) == 0;
        }
    }
    return s.size() - t.size() == 1 || s.size() - t.size() == -1;
```

Python:

```
    for i in range(min(len(s), len(t))):
        if s[i] != t[i]:
            return s[i + (1 if len(s) >= len(t) else 0):] == t[i + (1 if len(s) <
= len(t) else 0):]
    return abs(len(s) - len(t)) == 1
```

written by xcv58 original link here

Solution 3

To solve this problem, you first need to know what is*edit distance*. You may refer to this wikipedia article for more information.

For this problem, it implicitly assumes to use the classic**Levenshtein distance**, which involves **insertion**, **deletion** and **substitution** operations and all of them are of the same cost. Thus, if `S` is one edit distance apart from `T`, `T` is automatically one edit distance apart from `S`.

Now let's think about all the possible cases for two strings to be one edit distance apart. Well, that means, we can transform `S` to `T` by using exactly one edit operation. There are three possible cases:

1. We insert a character into `S` to get `T`.
2. We delete a character from `S` to get `T`.
3. We substitute a character of `S` to get `T`.

For cases 1 and 2, `S` and `T` will be one apart in their lengths. For cases 3, they are of the same length.

It is relatively easy to handle case 3. We simply traverse both of them and compare the characters at the corresponding positions. If we find exactly one mismatch during the traverse, they are one edit distance apart.

Now let's move on to cases 1 and 2. In fact, they can be merged into one case, that is, to delete a character from the longer string to get the shorter one, or equivalently, to insert a character into the shorter string to get the longer one.

We will handle cases 1 and 2 using the shorter string as the reference. We traverse the two strings, once we find a mismatch. We know this position is where the deletion in the longer string happens. For example, suppose `S = "kitten"` and `T = "kiten"`, we meet the first mismatch in the `4`-th position (`1`-based), which corresponds to the deleted character below, shown in between `*`. We then continue to compare the remaining sub-string of `T` (`en`) with the remaining sub-string of `S` (`en`) and find them to be the same. So they are one edit distance apart.

```
S: k i t t e n
```

```
T: k i t *t* e n
```

In fact, cases 1, 2 and 3 can be further handled using the same piece of code. For strings of the same length, once we find a mismatch, we just substitute one to be another and check whether they are now the same. For strings of one apart in lengths, we insert the deleted character of the longer string into the shorter one and compare whether they are the same.

The code is as follows. If you find the first half of the return statement (`!mismatch && (n - m == 1)`) hard to understand, run the code on cases that the mismatch only occurs at the last character of the longer string, like `S = "ab"` and `T = "abc"`.

```cpp
class Solution {
public:
    bool isOneEditDistance(string s, string t) {
        int m = s.length(), n = t.length();
        if (m > n) return isOneEditDistance(t, s);
        if (n - m > 1) return false;
        bool mismatch = false;
        for (int i = 0; i < m; i++) {
            if (s[i] != t[i]) {
                if (m == n) s[i] = t[i];
                else s.insert(i, 1, t[i]);
                mismatch = true;
                break;
            }
        }
        return (!mismatch && n - m == 1) || (mismatch && s == t);
    }
};
```

written by jianchao.li.fighter original link here