### Calculating Percentage Similarity of 2 Question

#### Problem

When a teacher imports a question list into the question bank, some questions may be duplicated. This can lead to a waste of server’s storage.

#### Solution

In order to solve the mentioned problem, we decide to apply the Levenshtein distance (thêm citation) algorithm.

Levenshtein distance algorithm is used to identify the differences between the range of 2 sequences. The range between these sequences is the minimum steps to make one sequence become the other one. This algorithm includes 3 changing functions:

- Remove a character

- Add a new character

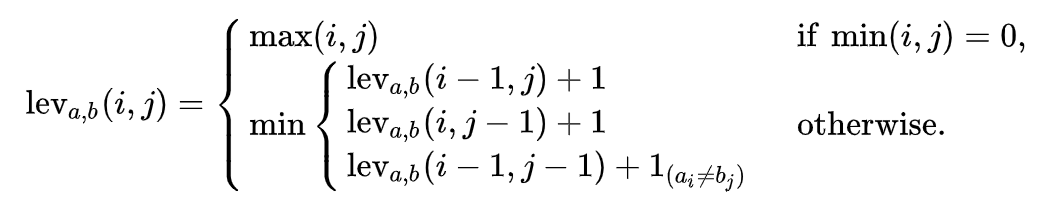
- Replace with another character

Example: To change “kitten” to “sitting”, we have to use at least 3 changing times as following:

1. kitten -> sitten (replace "k" with "s")
2. sitten -> sittin (replace "e" with "i")
3. sittin -> sitting (add "g")

Therefore, the range between "kitten" and sitting" is 3.

The nature of Levenshtein distance algorithm is based on Dynamic Programming. Mathematically, the Levenshtein distance between two strings {\displaystyle a,b}a, b (of length {\displaystyle |a|}|a| and {\displaystyle |b|}|b| respectively) is given by ,{\displaystyle \operatorname {lev} \_{a,b}(|a|,|b|)} where:



With

* {\displaystyle 1\_{(a\_{i}\neq b\_{j})}}  equal to 0 when {\displaystyle a\_{i}=b\_{j}} and equal to 1 otherwise
* {\displaystyle \operatorname {lev} \_{a,b}(i,j)}  is the distance between the first {\displaystyle i} characters of {\displaystyle a} and the first {\displaystyle j} characters of {\displaystyle b}

Note that the first element in the minimum corresponds to deletion (from {\displaystyle a} to {\displaystyle b}), the second to insertion and the third to match or mismatch, depending on whether the respective symbols are the same.

Here is a straightforward pseudocode for a function called LevenshteinDistance that takes two strings:

* *s* of length *m*
* *t* of length *n*

and returns the Levenshtein distance between them:

function LevenshteinDistance(char s[1..m], char t[1..n]):

// for all i and j, d[i,j] will hold the Levenshtein distance between

// the first i characters of s and the first j characters of t

// note that d has (m+1)\*(n+1) values

declare int d[0..m, 0..n]

set each element in d to zero

// source prefixes can be transformed into empty string by

// dropping all characters

for i from 1 to m:

d[i, 0] := i

// target prefixes can be reached from empty source prefix

// by inserting every character

for j from 1 to n:

d[0, j] := j

for j from 1 to n:

for i from 1 to m:

if s[i] = t[j]:

substitutionCost := 0

else:

substitutionCost := 1

d[i, j] := minimum(d[i-1, j] + 1, // deletion

d[i, j-1] + 1, // insertion

d[i-1, j-1] + substitutionCost) // substitution

return d[m, n]

Two examples of the resulting matrix (hovering over a tagged number reveals the operation performed to get that number):

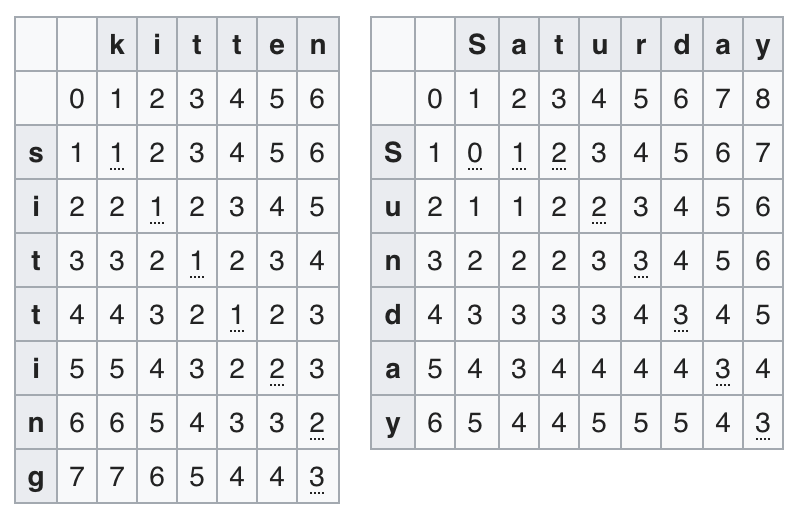


Figure 38 Levenshtein distance example result

#### Complexity

In totally, the complexity of this algorithm is O(n\*m) with n is length of first question and m is length of second question need to compare