SUYASH PRATAP SINGH(181B226)

TASKS:-

1. Implement the Levenshtein distance in python.
2. Create a vocabulary of a minimum of 100 words.
3. Take any sample word as input.
4. The code should calculate the edit distance from every word in vocabulary and display only those words as output havin levenshtein distance of less tahn or equal to 2.

In [1]:
```
import numpy as np
import pandas as pd
```

In [2]:
```
def med(word1,word2):
    word1,word2=word1.upper(),word2.upper()
    #Initializing matrix
    len1,len2 = len(word1),len(word2)
    matrix = np.array([['0 ']*(len2+2)]*(len1+2))
    for i in range(len1):
        matrix[i+2][0],matrix[i+2][1] = word1[i],i+1
    for i in range(len2):
        matrix[0][i+2],matrix[1][i+2] = word2[i],i+1
    #Calculating Minimum Edit Distance
    for i in range(2,len1+2):
        for j in range(2,len2+2):
            prev = min(matrix[i-1][j-1],matrix[i-1][j],matrix[i][j-1])
            matrix[i][j] = int(prev)+(matrix[i][0] != matrix[0][j])
    return int(matrix[-1,-1])
```

In [3]:
```
med('asdf','ardg')
```

Out[3]:
```
2
```

In [4]:
```
file=open(r'C:\Users\Admin\Downloads\Suyash.txt')
mafia=file.readlines()
```

In [5]:
```
#Removing punctuations
import re
mafia = [re.sub(r'[^\w\s]', '', x).lower() for x in mafia]
```

```
In [6]: mafia
```

```
Out[6]: ['in information theory linguistics and computer science the levenshtein dist
ance is a string metric for measuring the difference between two sequences in
formally the levenshtein distance between two words is the minimum number of
singlecharacter edits insertions deletions or substitutions required to chang
e one word into the other it is named after the soviet mathematician vladimir
levenshtein who considered this distance in 1965 levenshtein distance may als
o be referred to as edit distance although that term may also denote a larger
family of distance metrics known collectively as edit distance232 it is close
ly related to pairwise string alignments in approximate string matching the o
bjective is to find matches for short strings in many longer texts in situati
ons where a small number of differences is to be expected the short strings c
ould come from a dictionary for instance here one of the strings is typically
short while the other is arbitrarily long this has a wide range of applicatio
ns for instance spell checkers correction systems for optical character recog
nition and software to assist natural language translation based on translati
on memory the levenshtein distance can also be computed between two longer st
rings but the cost to compute it which is roughly proportional to the product
of the two string lengths makes this impractical thus when used to aid in fuz
zy string searching in applications such as record linkage the compared strin
gs are usually short to help improve speed of comparisons in linguistics the
levenshtein distance is used as a metric to quantify the linguistic distance
or how different two languages are from one another3 it is related to mutual
intelligibility the higher the linguistic distance the lower the mutual intel
ligibility and the lower the linguistic distance the higher the mutual intell
igibility computing the levenshtein distance is based on the observation that
if we reserve a matrix to hold the levenshtein distances between all prefixes
of the first string and all prefixes of the second then we can compute the va
lues in the matrix in a dynamic programming fashion and thus find the distanc
e between the two full strings as the last value computed iterative with two
matrix rows it turns out that only two rows of the table are needed for the c
onstruction if one does not want to reconstruct the edited input strings the
previous row and the current row being calculated the levenshtein distance ma
y be calculated iteratively using the following algorithm this two row varian
t is suboptimalâthe amount of memory required may be reduced to one row and o
ne index word of overhead for better cache locality hirschbergs algorithm com
bines this method with divide and conquer it can compute the optimal edit seq
uence and not just the edit distance in the same asymptotic time and space bo
unds the dynamic variant is not the ideal implementation an adaptive approach
may reduce the amount of memory required and in the best case may reduce the
time complexity to linear in the length of the shortest string and in the wor
st case no more than quadratic in the length of the shortest string the idea
is that one can use efficient library functions to check for common prefixes
and suffixes and only dive into the dp part on mismatch']
```

```
In [8]: tokens=set(mafia[0].split())
```

```
In [9]: def similar(word):
            for i in tokens:
                n = med(word,i)
                if n < 3:
                    print(n,i)
```

In [10]:
```
import numpy
def levenshteinDistanceDP(token1, token2):
    distances = numpy.zeros((len(token1) + 1, len(token2) + 1))

    for t1 in range(len(token1) + 1):
        distances[t1][0] = t1

    for t2 in range(len(token2) + 1):
        distances[0][t2] = t2

    a = 0
    b = 0
    c = 0

    for t1 in range(1, len(token1) + 1):
        for t2 in range(1, len(token2) + 1):
            if (token1[t1-1] == token2[t2-1]):
                distances[t1][t2] = distances[t1 - 1][t2 - 1]
            else:
                a = distances[t1][t2 - 1]
                b = distances[t1 - 1][t2]
                c = distances[t1 - 1][t2 - 1]

                if (a <= b and a <= c):
                    distances[t1][t2] = a + 1
                elif (b <= a and b <= c):
                    distances[t1][t2] = b + 1
                else:
                    distances[t1][t2] = c + 1


    return distances[len(token1)][len(token2)]
```

In [11]:
```
distance = levenshteinDistanceDP('asdf','ardg')
print(distance)
```

```
2.0
```

In [12]:
```
teja=[]
while _ in range(100):
    teja.append(input())
```

In [13]:
```
teja=['comparison', 'bot', 'influence', 'exciting', 'outgoing', 'damage', 'mic
e', 'wretched', 'early', 'nonstop', 'song',
   'aback', 'laborer', 'guard', 'aunt', 'lumber', 'grape', 'beautiful', 'passe
nger', 'ornament', 'barbarous', 'vein',
   'claim', 'detailed', 'love', 'zipper', 'smoggy', 'language', 'protective',
'rifle', 'coast', 'guess', 'spiky',
   'multiply', 'birth', 'railway', 'stiff', 'discover', 'book', 'fall', 'mount
ainous', 'assorted', 'lyrical',
   'jittery', 'murky', 'smelly', 'ban', 'realize', 'rhetorical', 'efficient',
'prefer', 'cows', 'cheerful',
   'toad', 'deafening', 'stop', 'invent', 'cook', 'interesting', 'useless', 'c
lear', 'jog', 'cheap', 'raise',
   'avoid', 'increase', 'noisy', 'serious', 'sea', 'spiteful', 'loud', 'rod',
'replace', 'oranges','border', 'second', 'cactus',
   'perpetual', 'grip', 'dispensable','sprout', 'pat', 'friend', 'boat', 'hal
f', 'elbow', 'poor', 'beak', 'busy', 'monkey',
   'control', 'cautious', 'trace', 'deletion', 'disarm', 'mundane', 'divergen
t', 'horn', 'cars', 'prose']
print(len(teja))
```

```
100
```

In [14]:
```
def edit_distance(w1, w2, m, n):
    if m == 0:
        return n
    if n == 0:
        return m
    if w1[m-1] == w2[n-1]:
        return edit_distance(w1, w2, m-1, n-1)
    return 1 + min(edit_distance(w1, w2, m, n-1),     # Insertion property foll
ow
                   edit_distance(w1, w2, m-1, n),     # deletion property follo
w
                   edit_distance(w1, w2, m-1, n-1)    # substituion property f
ollow
                   )
```

In [15]:
```
s='loud'
for i in teja:
    dist=edit_distance(s, i, len(s), len(i))
    if(dist<=2):
        print (i)
```

```
love
toad
loud
rod
```

THANK YOU