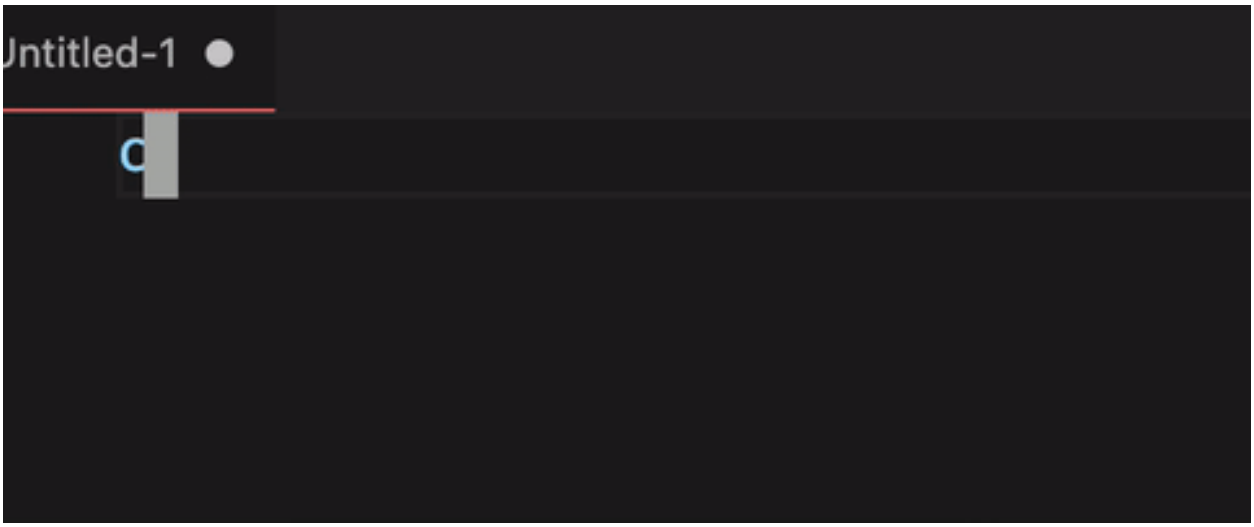
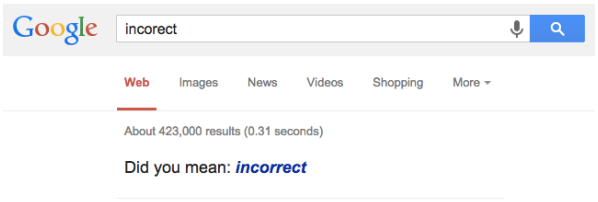


Autocorrect

Introduction

Spellcheckers and A**utocorrect** can feel like magic. They're at the core of everyday applications, our phones, office software, google search. What a spellchecker does ? It takes in a misspelled word — for example 'teh' — and returns the best guess at the correct spelling — 'the'.

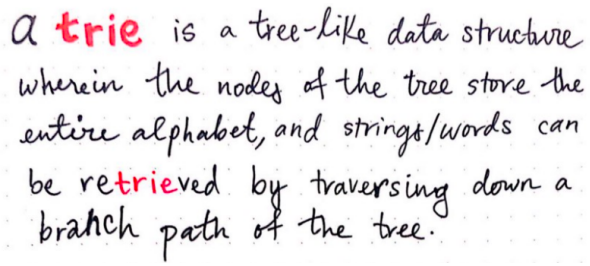


For most spell checkers, a candidate word is considered to be spelled correctly if it is found in a long list of valid words called a *dictionary*. In case of spellcheck provided by Google , if it doesn't find the keyword in the dictionary, it suggests a most likely replacement. To do this it associates with every word in the dictionary a frequency, the number of the times that word is expected to appear in a large document. When a word is misspelled (i.e. it is not found in the dictionary) Google suggests a "similar" word whose frequency is larger or equal to any other "similar" word.

Implemenation

Implementing the autocorrect feature involves the use of trie data structure, edit distance and hash maps.

Trie



First of all, we need a dictionary, a sorted list of words stored in a large text file. The dictionary is built using the trie data structure, discussed in the

section. Here is how we define a trie node - it has an array of 26 nodes , it has a variable, *count* to store the frequency of the word, a boolean variable isEndOfWord to indicate whether the node is the end of a word.

Edit Distance

The problem of edit distance is to find the minimum number of insert/delete/overwrite operations to transform one string to another. Another perspective to look at the problem will be that a natural measure of the distance between two strings is the extent to which they can be aligned, or matched up.

		E	L	E	P	H	A	N	T
	0	1	2	3	4	5	6	7	8
R	1	1	2	3	4	5	6	7	8
E	2	1	2	2	3	4	5	6	7
L	3	2	1	2	3	4	5	6	7
E	4	3	2	1	2	3	4	5	6
V	5	4	3	2	2	3	4	5	6
A	6	5	4	3	3	3	3	4	5
N	7	6	5	4	4	4	4	3	4
T	8	7	6	5	5	5	5	4	3

Solution

What Are The Sub-Problems ?

E(i, j) - edit distance between some prefix of the first string, x[1 ... i], and some prefix of the second, y[1 ... j], m = strlen(x), n=strlen(y).

Our goal is to find the edit distance between strings x[1.....m]and y[1.....n], that is, E(m, n).

Expressing `E(i, j)` in terms of smaller sub-problems

Expressing `E(i, j)` in terms of smaller sub-problems

> $E(i,j) = min(1 + E(i-1,j), 1 + E(j,i-1), diff(i,j) + E(i-1,j-1))$, where $diff(i,j) = 0$ when $x[i] = y[j]$ and 1 otherwise.

> When aligning $x[1....i]$ and $y[1....j]$, the righmost column can be

1. $\begin{pmatrix} x[i] \\ - \end{pmatrix}$

> delete $x[i]$, $E(i,j) = 1 + E(i-1,j)$

2. $\begin{pmatrix} - \\ y[j] \end{pmatrix}$

> $inserty[j]$, $E(i,j) = 1 + E(i,j-1)$

3. $\begin{pmatrix} x[i] \\ y[j] \end{pmatrix}$

> overwrite $x[i]$ to $y[j]$ $E(i,j) = diff(i,j) + E(i-1,j-1)$

In what order should these sub-problems be solved ?
E(i - 1, j), E(i , j - 1), E(i - 1, j - 1)E(i−1,j),E(i,j−1),E(i−1,j−1) are to be solved before E(i , j). So, we go solving every sub-problem row-wise.

Algorithm

```
for i = 0,1,2,...m:
    E(i, 0) = i
for j = 0,1,2,...n:
    E(j, 0) = j
for i = 0,1,2,...m:
    for j = 0,1,2,...n:
            E(i, j) = min(1 + E(i-1, j), 1 + E(j, i - 1), diff(i, j) + E(i-1, j-1))
return E(m, n)
```

> Time Complexity - O(m, n) to find the edit distance between two strings of length m and n.

Autocorrect Algorithm

Now, to implement the autocorrect feature itself, first, we load all the words found in the provided dictionary text file into the **trie**. Next, the user's input string is compared against the

trie, if the input string is found in the trie, the program will indicate that it is spelled correctly. If the input string is not found, the program will return the node of **the most "similar" word.**

How a word is determined to be most similar:

1. It has the "closest" **edit distance** from the input string
2. If multiple words have the same edit distance, the most similar word is the one with higher frequency.
3. If multiple words are the same edit distance and have the same count/frequency, the most similar word is the one that is first alphabetically.

A word *w* in the dictionary has an edit distance of x from the input string if there exists a string *s* such that *s* has an edit distance of x/2 from the input string and w has an edit distance of x/2 from *s*.

If there is no word in the dictionary that has an edit distance of 1 or 2, there is no word in the dictionary "similar" to the input string. In that case, return NULL.