AUTOCORRECT AND AUTOCOMPLETE

CS16: Introduction to Data Structures & Algorithms
Outline

1. Autocorrect
2. Levenshtein Edit Distance
3. Autocomplete
4. Tries
5. Autocorrect with a Trie
Autocorrect – How would you do it?

• Finding words similar to the one typed...

• What is a reasonable measure of similarity?
Edit Distance

• The “edit distance” between two words is the number of insertions, deletions, and/or substitutions needed to transform one word into the other. Gives rise to alignment:

\[
\begin{align*}
&\text{C A N A D A - (start word)} \\
&\text{B A N A N A S (end word)}
\end{align*}
\]

• Edit distance: 3 (1 insertion, 2 substitutions)
• Also called “Levenshtein edit distance” (LED)
Applying Alignment to Autocorrect

• How does this look like in code? Discuss with your neighbor.
Edit Distance Defined Recursively

- If we remove the last column (just S), we’ll have to find the edit distance between “CANADA” and “BANANA” to build the edit distance for “CANADA” and “BANANAS”

- We can define the optimal edit distance recursively using shorter sub words!
Breaking it into Sub-problems

• Ex. - Editing “AN” into “TO”.

• Our final minimum edit distance will be the best of the solutions to the following three sub-problems
  • Min. edit distance between “A” and “TO” + 1 deletion
  • Min. edit distance between “A” and “T” + 1 substitution for the “N” → “O” transition
  • Min. edit distance between “AN” and “T” + 1 insertion

• We can further solve each of these sub-problems in a similar manner.

• The base case is when one of the sub-words is an empty string and the edit distance is inserting as many letters as the other sub-word currently has.
Recursive Definition

```python
function recursiveEditDist(word1, word2):
    // base cases
    if word1.length == 0: return word2.length
    if word2.length == 0: return word1.length

    return
    min(
        // deletion
        recursiveEditDist(word1[0:endLetter-1], word2) + 1,
        // insertion
        recursiveEditDist(word1, word2[0:endLetter-1]) + 1,
        // substitution
        recursiveEditDist(word1[0:endLetter-1], word2[0:endLetter-1])
        + (0 if word1[endLetter] == word2[endLetter] else 1)
    )
```
Diagram of the Recursion

AN → TO

A → TO

A → T

AN → T

Remove “N”

Remove “O” and “N”

Remove “O”

(Deletion)  (Substitution)  (Insertion)
Analysis of Recursive Approach

• As we see from the recursive tree, sub-problems are often recomputed in this top-down recursive approach
  • Ex. – “A” → “T” is recomputed thrice

• Runtime is exponential, which is suboptimal

• Our recursive definition gives us a recurrence relation though, so there may be a way to store already computed solutions in a table

• We can take the bottom-up approach and store sub-problem solutions starting from the base cases.
Dynamic Programming Recap

1. Make a table to hold values from smaller versions of the problem (Table “T”)
2. For any cell in T, describe in words what that cell represents
3. For any cell in T, define the recurrence relation used to calculate its value
4. Decide what order you’ll fill in the table
5. Fill in the base case(s) in the table and then fill in the rest of it based on the decided order
6. Look up the final answer in the table
Step 1: Make a Table

<table>
<thead>
<tr>
<th></th>
<th>H</th>
<th>E</th>
<th>L</th>
<th>L</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘’</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>‘’</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- colWord = “HELLO”
- rowWord = “SMELLY”
- Leave an empty row and column at the front of each word – they represent the base cases of the recurrence
Step 2: Describe each Cell in Words

<table>
<thead>
<tr>
<th></th>
<th>H</th>
<th>E</th>
<th>L</th>
<th>L</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>“”</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>“”</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- \( T[i,j] \) holds the edit distance between the sub words \( \text{rowWord}[0:i] \) and \( \text{colWord}[0:j] \)

- Ex. – \( T[4,2] \) is the edit distance between “SMEL” and “HE”
### Step 3: Recurrence Relation

The recurrence relation is:

$$ T[i,j] = \min(T[i,j-1] + 1, \ T[i-1,j] + 1, \ T[i-1][j-1] + \{1,0\}) $$

depending on whether $\text{rowWord}[i] == \text{colWord}[j]$.

<table>
<thead>
<tr>
<th></th>
<th>H</th>
<th>E</th>
<th>L</th>
<th>L</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Start word

Final word
### Step 4: Decide the Order of Filling in the Table

- Can think of the table as a DAG
  - Directed edges represent dependencies
- Cells must be filled in a topologically sorted order
- You can fill this one out left to right, top to bottom, i.e. increasing i and increasing j order

<table>
<thead>
<tr>
<th>Start word</th>
<th>“”</th>
<th>H</th>
<th>E</th>
<th>L</th>
<th>L</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>“”</td>
<td>“”</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Final word</td>
<td>“”</td>
<td>H</td>
<td>E</td>
<td>L</td>
<td>L</td>
<td>O</td>
</tr>
</tbody>
</table>
Step 5: Fill in the Table

<table>
<thead>
<tr>
<th>Final word</th>
<th>Start word</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>S</td>
<td>1 1 2 3 4 5</td>
</tr>
<tr>
<td>M</td>
<td>2 2 2 3 4 5</td>
</tr>
<tr>
<td>E</td>
<td>3 3 2 3 4 5</td>
</tr>
<tr>
<td>L</td>
<td>4 4 3 2 3 4</td>
</tr>
<tr>
<td>L</td>
<td>5 5 4 3 2 3</td>
</tr>
<tr>
<td>Y</td>
<td>6 6 5 4 3 3</td>
</tr>
</tbody>
</table>

- First row and first col: base cases
  - Edit distance between empty string and sub-word is always just the length of the sub-word
  - The extra space adds the base case
- Everything else is the regular case

Final answer! 3
function editDistance(word1, word2):
    word1 = “ ” + word1  // pad out words with leading space
    word2 = “ ” + word2

    numCols = word1.size
    numRows = word2.size
    T = 2D array of size [numRows x numCols]

    for r from 0 to numRows-1:  // base cases!
        T[r][0] = r
    for c from 0 to numCols-1:
        T[0][c] = c

    for r from 1 to numRows-1:
        for c from 1 to numCols-1:
            T[r][c] = min(T[r-1][c] + 1,  // insertion
                           T[r][c-1] + 1,  // deletion
                           T[r-1][c-1] + (0 if word1[c] == word2[r] else 1))  // substitution

    return T[numRows-1][numCols-1]  // get the final answer
Edit Distance Analysis

• Analysis of dynamic programming algorithms is straightforward
  • Runtime is the size of the table

• If M and N are the lengths of the 2 input words, what is the runtime?
  • $O(M \times N)$

• The bottom-up approach reduced runtime from exponential to polynomial
Recreating the Alignment

We can trace back to determine what edits transform “HELLO” to “SMELLY”

- Diagonal -> match/substitution
- Vertical -> insertion
- Horizontal -> deletion

H - E L L O
S M E L L Y
Outputting the Alignment

• Trace back along the path, following the pointer with minimum weight
• Going from the bottom right:
  • ‘O’ and ‘Y’ is formed from a match/substitution, so we output ‘O’ and ‘Y’ to our alignment:

```
H E L L O
E L L Y
```
• Now we have two options because H/M could have come from multiple directions; we can pick arbitrarily

```
H - E L L O
S M E L L Y
```

Equally valid:
```
- H E L L O
S M E L L Y
```
Activity 1 – Edit distance
Applications of Edit Distance

• A good measure of checking similarities between words to correct misspellings, i.e. autocorrecting strings

• Computational Biology – used very often for tasks such as comparing different DNA strings and finding matches in the genome
Autocomplete
Trie

• Most people pronounce it as “try”

• A trie stores an entire corpus of words in lesser space than it would take to store the words in a set

• A trie is a prefix tree  
  • Every node represents a prefix of a word

Yellow marks the end of a word
Trie Nodes

• Each node has a dictionary of children so it can easily map a letter to the correct child node

• We also need to keep track of which nodes are the ends of words

class Node:
    children = dict()
    isWordEnd = false

Yellow marks the end of a word
function insert(node, word):
    if word.length == 0:
        node.isWordEnd = true
        return

    firstLetter = word[0]
    if firstLetter not in node.children:
        node.children[firstLetter] = new Node()
    insert(node.children[firstLetter], word[1:end])
function find(node, prefix):
    // Output: the last node of the prefix, if it exists
    if prefix.length == 0:
        return node
    firstLetter = prefix[0]
    if firstLetter not in node.children:
        return "prefix not found"
    return find(node.children[firstLetter],
                prefix[1:end])
Autocomplete

• Call `find()` to find the last node in the prefix chain

• Run DFS using that node as a root to compile a list of matches

• Example: Autocomplete “AN”
Autocomplete

function autocomplete(prefix):
    start = find(Trie.root, prefix) //find node representing prefix
    return dfs(start, prefix) //find all children words

function dfs(node, wordSoFar):
    //wordSoFar is the current prefix path from the root
    matches = []
    if node.isWordEnd:
        matches.append(wordSoFar)
    for child in node.children:
        childMatches = dfs(child, wordSoFar + child’s key)
        matches += childMatches
    return matches
Trie Analysis

• `find()` is worst case $O(M)$, where $m$ is the length of the longest word in the trie
  • For this reason, tries are sometimes a good replacement for hashtables, especially for storing short strings, like real words
  • Sometimes requires more memory than a hashtable (for long strings)

• `autocomplete()` is $O(n*m)$, where $n$ is the number of words and $m$ is the length of the longest word
Activity 2 - Autocomplete
Autocorrect without a Trie...

• How do I find the best Levenshtein distances from one word to all other words in a dictionary of words?
  • You have to compare a given word to all words in the dictionary
  • Extremely slow and impractical as number of strings and length of strings grows
**Autocorrect with a Trie**

- How do I find the best Levenshtein distances to other words?
  1. Set a maximum edit distance for suggestions (Ex. – 3 edits)
  2. Compute edit distance from root of trie
  3. Recursively compute edit distances going down all children, stopping when the minimum edit distance is above the maximum we have set

- Thus, only go through nodes that could be similar enough to the target word to be reasonable

- You can (and should) use a trie for *both* autocomplete and autocorrect
Autocorrect with Trie Example

• Autocorrect “AP” with an edit distance of up to 2 edits
• Go down the trie starting at “A”
• We’re 1 edit away from “AP” so “A” counts
• Then visit “A”’s children
  • We see “AN” is 1 edit away as well so “AN” counts
• Then visit “N”’s children
  • We see “AND” is 2 edits away so “AND” counts
  • “ANI” is not a valid word but is still 2 edits away
• We stop here on this branch because going further down the trie means the edit distance will be > 2
• Similarly, we check the “I” and “P” branches
• Finally, the autocorrect options are “A”, “AN”, “AND”, “I”, “IT”

Yellow marks the end of a word