



BMS College of Engineering
Department of Information Science and Engineering

“Autocompletion and spell checker using dynamic programming”

By :-

ADITYA A KAMAT (1BM18IS003)

ARYAN (1BM18IS018)

Abstract

- Autocomplete is a feature in which an application predicts the rest of a word a user is typing. Many autocomplete algorithms learn new words after the user has written them a few times, and can suggest alternatives based on the learned habits of the individual user.
- a spell checker (or spelling checker or spell check) is a software feature that checks for misspellings in a text. Spell-checking features are often embedded in software or services, such as a word processor, email client, electronic dictionary, or search engine.

Introduction

- The levenshtein distance algorithm is a dynamic programming algorithm which is a text similarity measure that compares two words and returns a numeric value representing the distance between them.
- The more similar the two words are, the less the distance between them, and vice versa.

Problem Statement

Implement Autocomplete and Spell checking using Levenshtein Distance algorithm

Relevance

- Autocompletion is one of the most important concept applied in smartphone keyboards and is also used in softwares such as Microsoft Word and Google Slides.
- Spell Checker is the core application of softwares such as Grammarly, which performs spell checking while writing emails or drafting reports.

Literature Survey

SL.NO	Title of the Paper	Problem Addressed	Authors Approach / Method	Results
1	Autocomplete and Spell Checking Levenshtein Distance Algorithm to Getting Text Suggest Error Data Searching in Library	To apply the autocomplete feature and spell checking with Levenshtein distance algorithm to get text suggestion in an error data searching in library and determine the level of accuracy on data search trials.	The feature that was used to provide autocomplete and spell checking was the Levenshtein distance algorithm.	Spell checking accuracy rate obtained from the system as 86%.
2	Research on String Similarity Algorithm based on Levenshtein Distance	Similarity algorithm based on Levenshtein Distance is improved.	Longest Common Subsequence (LCS) and Longest Continuous Common Substring (LCCS) are used.	The calculation of string similarity is further improved taking into account the important influence of LCS and LCCS on the computation of string similarity.

Literature Survey

SL.NO	Title of the Paper	Problem Addressed	Authors Approach / Method	Results
3	Levenshtein Distance Algorithm for Efficient and Effective XML Duplicate Detection	The aim of this project is to present a novel algorithm to find duplicate objects in the hierarchical structures, like XML files.	Proposed system uses Levenshtein distance algorithm to find duplicates in structured data.	Using Levenshtein Distance algorithm gives better result than Normalized Edit Distance algorithm.
4	Levenshtein Distance Algorithm Analysis on Enrollment and Disposition of Letters Application	This project results in the form of a letter filing and disposition application by applying the Levenshtein distance algorithm in letter search to speed up the letter search process.	To make it easier to letter archiving and speed up searching for letters, the Levenshtein distance algorithm is used.	Archiving design and letter disposition can display data by the word being searched even though there are some errors when writing the word you want to search for.

Literature Survey

SL.NO	Title of the Paper	Problem Addressed	Authors Approach / Method	Results
5	Using the Levenshtein Edit Distance for Automatic Lemmatization: A Case Study for Modern Greek and English	Performing an accurate lemmatization can be a quite difficult and time-consuming task especially for morphologically complex languages with highly inflexional structure, such as the Modern Greek language.	The Levenshtein Distance is implemented on a dictionary-based algorithm in order to achieve the automatic induction of the normalized form (lemma) of regular and mildly irregular words with no direct supervision.	In this paper, a new language independent lemmatization algorithm based on the Levenshtein distance was demonstrated and its performance was evaluated both on Modern Greek and English languages.
6	Levenshtein Distance, Sequence Comparison and Biological Database Search	The advent of modern genomic sequencing and the volume of data it generates has resulted in a return to the problem of local alignment.	Given two biological sequences of length n , the basic problem of biological sequence comparison can be recast as that of determining the Levenshtein distance between them.	Although some modern bioinformatics heuristics using k-mer matching have partially supplanted the direct optimization of Levenshtein distance, the algorithm is still relevant.

Literature Survey

SL.NO	Title of the Paper	Problem Addressed	Authors Approach / Method	Results
7	Learning String-Edit Distance	Determination of similarity of two strings	The authors implement a string-edit distance that reduces the error rate of the untrained Levenshtein distance by a factor of 4.7, to within 4 percent of the minimum error rate achievable by any classifier.	The authors demonstrate the efficacy of their techniques by correctly recognizing over 87 percent of the unseen pronunciations of syntactic words in conversational speech, which is within 4 percent of the maximum success rate achievable by any classifier.
8	Kernels Based on Weighted Levenshtein Distance	To determine the similarity or dissimilarity of two strings in many training algorithms.	The authors replace distance measure in the RBF and exponential kernels and inner product in polynomial and form a new class of string kernels: Levenshtein kernels.	It is demonstrated that incorporating prior knowledge about problems at hand into kernels can improve the performance of kernel machines effectively.

Literature Survey

SL.NO	Title of the Paper	Problem Addressed	Authors Approach / Method	Results
9	Computing the Levenshtein Distance of a Regular Language	In this paper, the problem of computing the edit distance of a regular language is considered.	The authors distinguish two cases depending on whether the given automaton is deterministic or nondeterministic.	The authors show why the problem of computing the edit distance of a given regular language is of polynomial time complexity.
10	Plagiarism Detection Using the Levenshtein Distance and Smith-Waterman Algorithm	Demonstrating the practicality of the Levenshtein algorithm for plagiarism detection.	Using the Levenshtein Distance algorithm and Smith-Waterman Algorithm.	The authors have described how the Levenshtein distance can be used to change the likely scarcity, which can improve both time and space efficiency.

Literature Survey

SL.NO	Title of the Paper	Problem Addressed	Authors Approach / Method	Results
11	A Normalized Levenshtein Distance Metric	To improve the metric between two strings and satisfy the triangle inequality.	The Generalised Levenshtein Distance algorithm is used.	The main contribution of the paper is to prove that the new distance is a metric valued in $[0, 1]$ and can generally achieve similar accuracies to two other normalized edit distances.
12	String correction using the Damerau-Levenshtein distance	In this paper, the authors focus on the development of algorithms whose space complexity is less and whose actual runtime and energy consumption are less than those of the algorithm of Lowrance and Wagner.	The authors develop space- and cache-efficient algorithms to compute the Damerau-Levenshtein (DL) distance between two strings.	The authors' algorithms are able to handle much larger sequences than earlier algorithms due to the reduction in space requirements.

Literature Survey

SL.NO	Title of the Paper	Problem Addressed	Authors Approach / Method	Results
13	Adapting the Levenshtein Distance to Contextual Spelling Correction	To screen and refine the results obtained by Levenshtein Distance and apply to errors in Arabic words.	The authors developed a system for correcting spelling errors in the Arabic language based on language models and Levenshtein algorithm.	From the results obtained, it is clear that this combination helps improve satisfactorily scheduling solutions returned by our method.
14	A Comparison of Standard Spell Checking Algorithms and a Novel Binary Neural Approach	To create an approach that is aimed towards isolated word error correction.	The authors integrate Hamming Distance and n-gram algorithms that have high recall for typing errors and a phonetic spell-checking algorithm in a single novel architecture.	The proposed hybrid methodology has the highest recall rate of the techniques evaluated. This method has a high recall rate and low-computational cost.

Literature Survey

SL.NO	Title of the Paper	Problem Addressed	Authors Approach / Method	Results
15	Spelling Checker Algorithm Methods for Many Languages	To implement spell checking in languages such as Arabic, Chinese, Hindi, Japanese etc.	The approach is divided into several parts, which are: determine the sources of research, define the keyword model for the search process, start inclusion and exclusion criteria, extract data and analyze the result to answer a research question.	It is known that every language has a different method of spell checking. The Damerau-Levenshtein algorithm method is most frequently used for spell checkers.

Methodology

We used Levenshtein Distance Algorithm to calculate the similarity between the input word and the target word in both spell check and auto-complete programs.

Here 3 techniques can be used for editing the input word into target word:

- Insertion
- Deletion
- Replacement

Methodology

The algorithm:

Step 1: Initialization

I. Set a is the length of document 1 say d_1 , set b is length of document 2 say d_2 .

II. Create a matrix that consists $0 - b$ rows and $0 - a$ columns.

III. Initialize the first row from 0 to a .

IV. Initialize the first column from 0 to b .

Methodology

Step2: Processing

- I. Observe the value of d_2 (i from 1 to a).
 - II. Observe the value of d_1 (j from 1 to b).
 - III. If the value at $d_2[i]$ is equals to value at $d_1[j]$, the cost becomes 0.
 - IV. If the value at $d_2[i]$ does not equal $d_1[j]$, the cost becomes 1.
 - V. Set block of matrix $M[d_1, d_2]$ of the matrix equal to the minimum of:
 - I. the block immediately above add 1: $M[d_2-1, d_1]$
 - li. the block immediately to the left add 1: $M[d_2, d_1-1] + 1$.
 - lii. The block is diagonally above and to the left adds the cost: $M[d_2-1, d_1-1] + \text{cost}$.
- Step 2 is repeated till the distance $M[a, b]$ value is found.

Methodology

Step 3: Result [1, 5].

2.1 Computing Techniques

$Dis(i, j)$ = score of best alignment from $d11..d1i$ to $d21.....d2j$

$Dis(i-1, j-1) + d(d1i, d2j)$ //copy $Dis(i-1, j) + 1$ //insert

$Dis(i, j-1) + 1$ //delete

Cost depend upon following factors:

$Dis(0, 0) = 0$ cost // if both strings are same

$Dis(i, 0) = dis(i-1, 0) + 1 = 0$ // if source string is empty

$Dis(0, j) = dis(0, j-1) + 1 = 0$ // if target string is empty [6, 7]

Methodology

For example:

- Given two words, **hello** and **hello**, the Levenshtein distance is zero because the words are identical.
- For the two words **helo** and **hello**, it is obvious that there is a missing character "l". Thus to transform the word **helo** to **hello** all we need to do is **insert** that character. The distance, in this case, is 1 because there is only one edit needed.
- On the other hand, for the two words **kelo** and **hello** more than just inserting the character "l" is needed. We also need to substitute the character "k" with "h". After such edits, the word **kelo** is converted into **hello**. The distance is therefore 2, because there are two operations applied: substitution and insertion.
- For the two words **kel** and **hello**, we must first replace "k" with "h", then add a missing "l" followed by an "o" at the end. As a result, the distance is 3 because there are three operations applied.

Methodology

		h	e	l	l	o
	0	1	2	3	4	5
k	1	1	2	3	4	5
e	2	2	1	2	3	4
l	3	3	2	1	2	2
m	4	4	3	2	2	3

$$\text{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \text{lev}_{a,b}(i-1,j) + 1 \\ \text{lev}_{a,b}(i,j-1) + 1 \\ \text{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

Here “hello” is input word that has to be transformed into “kelm” which is target word. By this method we can find the levenshtein distance.

Implementation

Autocomplete using **Levenshtein Distance algorithm**:

```
In [1]: import numpy

def printDistances(distances, token1Length, token2Length):
    for t1 in range(token1Length + 1):
        for t2 in range(token2Length + 1):
            print(int(distances[t1][t2]), end=" ")
            print()

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

        printDistances(distances, len(token1), len(token2))
        return distances[len(token1)][len(token2)]
    distance = levenshteinDistanceDP("kelm", "hello")

def calcDictDistance(word, numWords):
    file = open('text.txt', 'r')
    lines = file.readlines()
    file.close()
    dictWordDist = []
    wordIdx = 0
```

Implementation

Autocomplete using **Levenshtein Distance** algorithm:

OUTPUT:

```
jupyter Levenshtein Distance Last Checkpoint: an hour ago (autosaved)
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3
+ - * < > ↩ ⌂ Run C Code
dictwordDist = []
wordIdx = 0

for line in lines:
    wordDistance = levenshteinDistanceOP(word, line.strip())
    if wordDistance >= 18:
        wordDistance = 9
    dictwordDist.append(str(int(wordDistance)) + "-" + line.strip())
    wordIdx = wordIdx + 1

closestWords = []
wordDetails = []
currWordDist = 0
dictwordDist.sort()
print(dictwordDist)
for i in range(numWords):
    currWordDist = dictwordDist[i]
    wordDetails = currWordDist.split("-")
    closestWords.append(wordDetails[1])
return closestWords
print(calcDictDistance("pape", 3))

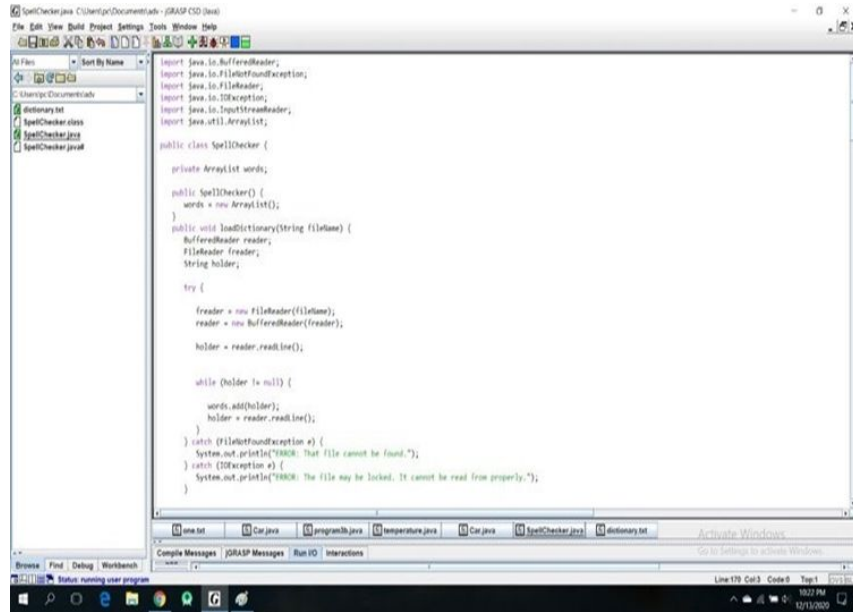
0 1 2 3 4 5
1 1 2 3 4 5
2 2 1 2 3 4
3 3 2 1 2 3
4 4 3 2 2 3
```

```
In [3]: print(calcDictDistance("pape", 3))

['1-page', '1-paper', '2-age', '2-are', '2-base', '2-came', '2-care', '2-case', '2-ease', '2-face', '2-game', '2-gave',
'2-have', '2-hope', '2-lake', '2-late', '2-made', '2-make', '2-map', '2-name', '2-pair', '2-part', '2-pass', '2-past', '2
-path', '2-pay', '2-place', '2-plane', '2-pose', '2-race', '2-rope', '2-safe', '2-same', '2-save', '2-shape', '2-space',
'2-take', '2-type', '2-wave', '3-a', '3-able', '3-act', '3-add', '3-ago', '3-air', '3-all', '3-am', '3-an', '3-and', '3-a
ny', '3-apple', '3-area', '3-arm', '3-art', '3-as', '3-ask', '3-at', '3-baby', '3-back', '3-bad', '3-ball', '3-band', '3-b
ank', '3-bar', '3-bat', '3-be', '3-blue', '3-bone', '3-call', '3-camp', '3-can', '3-can', '3-card', '3-cat', '3-cause',
'3-come', '3-copy', '3-dad', '3-dance', '3-dark', '3-day', '3-die', '3-done', '3-each', '3-ear', '3-east', '3-eat', '3-ed
ge', '3-else', '3-eye', '3-fact', '3-fair', '3-fall', '3-far', '3-farm', '3-fast', '3-fat', '3-fine', '3-fire', '3-five',
'3-free', '3-gas', '3-give', '3-gone', '3-had', '3-hain', '3-half', '3-hand', '3-happen', '3-happy', '3-hard', '3-has',
'3-hat', '3-he', '3-here', '3-hole', '3-home', '3-huge', '3-ice', '3-kept', '3-lady', '3-land', '3-large', '3-last', '3-l
aw', '3-lay', '3-leave', '3-lie', '3-life', '3-like', '3-line', '3-live', '3-lone', '3-love', '3-main', '3-man', '3-man
y', '3-mark', '3-mass', '3-may', '3-me', '3-mile', '3-mine', '3-more', '3-move', '3-nine', '3-nose', '3-note', '3-once',
'3-one', '3-open', '3-paint', '3-parent', '3-party', '3-people', '3-phrase', '3-pick', '3-piece', '3-plain', '3-plan', '3
-planet', '3-plant', '3-play', '3-please', '3-poem', '3-poor', '3-port', '3-post', '3-power', '3-proper', '3-prove', '3-p
ull', '3-push', '3-put', '3-rail', '3-rain', '3-raise', '3-ran', '3-range', '3-ride', '3-rise', '3-rose', '3-rule', '3-sa
id', '3-sail', '3-salt', '3-sand', '3-sat', '3-saw', '3-say', '3-scale', '3-see', '3-share', '3-she', '3-shoe', '3-side',
'3-size', '3-slave', '3-some', '3-spoke', '3-state', '3-sure', '3-table', '3-tail', '3-talk', '3-tall', '3-the', '3-tie',
'3-time', '3-tire', '3-tone', '3-top', '3-trade', '3-tree', '3-true', '3-tube', '3-up', '3-use', '3-value', '3-vary', '3-
wait', '3-walk', '3-wall', '3-want', '3-wan', '3-warm', '3-was', '3-wash', '3-water', '3-way', '3-we', '3-were', '3-wid
e', '3-wife', '3-wire', '3-yard', '4-I', '4-above', '4-after', '4-again', '4-agree', '4-also', '4-anger', '4-appear', '4-
```

Implementation

Spell check using Levenshtein Distance Algorithm



The screenshot shows a Java IDE with the following code in the SpellChecker.java file:

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;

public class SpellChecker {

    private ArrayList words;

    public SpellChecker() {
        words = new ArrayList();
    }

    public void loadDictionary(String filename) {
        BufferedReader reader;
        FileReader fr;
        String holder;

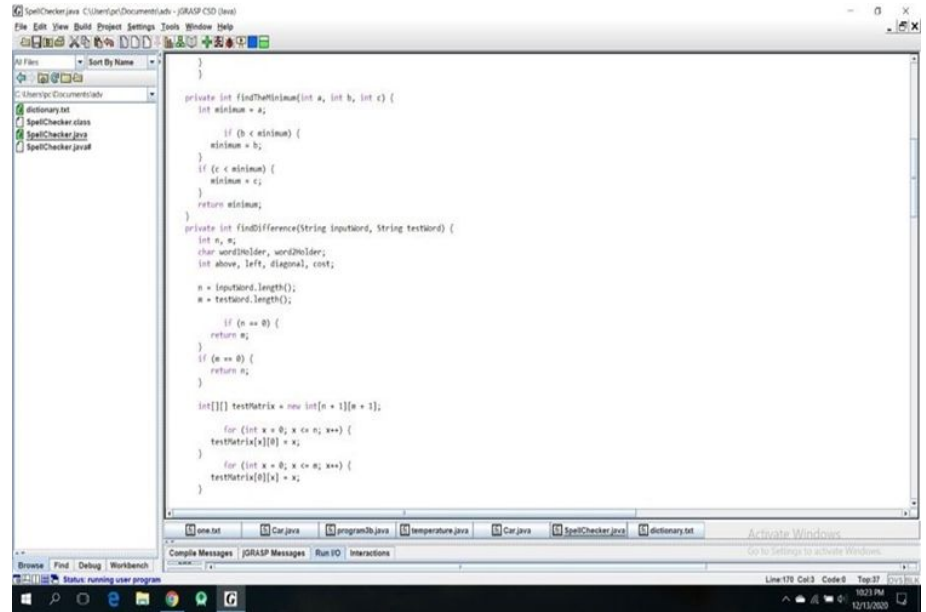
        try {

            fr = new FileReader(filename);
            reader = new BufferedReader(fr);

            holder = reader.readLine();

            while (holder != null) {

                words.add(holder);
                holder = reader.readLine();
            }
        } catch (FileNotFoundException e) {
            System.out.println("ERROR: That file cannot be found.");
        } catch (IOException e) {
            System.out.println("ERROR: The file may be locked. It cannot be read from properly.");
        }
    }
}
```



The screenshot shows the continuation of the SpellChecker.java file with the following code:

```
}

private int findMinimum(int a, int b, int c) {
    int minimum = a;

    if (b < minimum) {
        minimum = b;
    }
    if (c < minimum) {
        minimum = c;
    }
    return minimum;
}

private int findDifference(String inputWord, String testWord) {
    int n, m;
    char wordHolder, word2Holder;
    int above, left, diagonal, cost;

    n = inputWord.length();
    m = testWord.length();

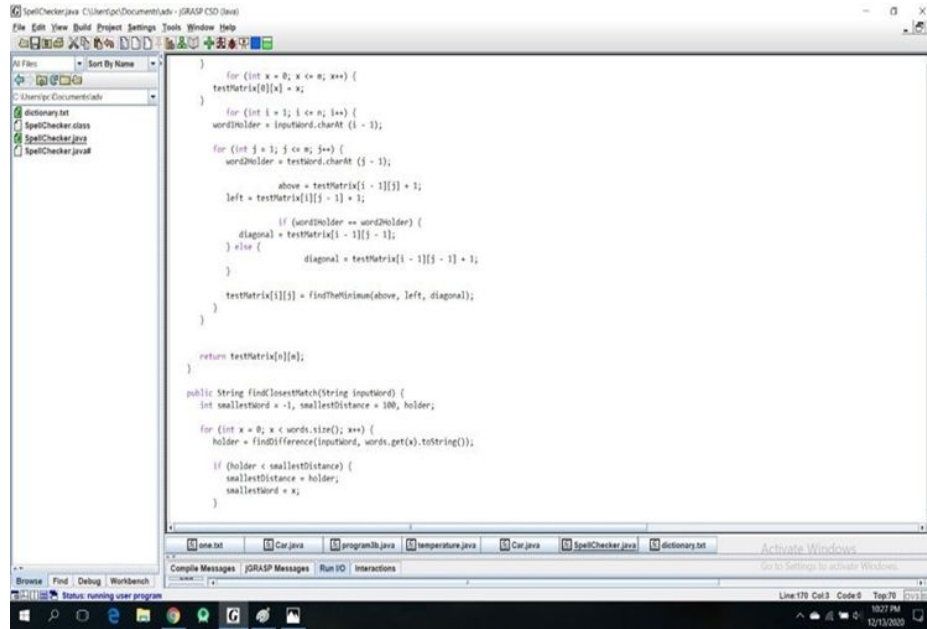
    if (n == 0) {
        return m;
    }
    if (m == 0) {
        return n;
    }

    int[][] testMatrix = new int[n + 1][m + 1];

    for (int x = 0; x <= n; x++) {
        testMatrix[x][0] = x;
    }
    for (int x = 0; x <= m; x++) {
        testMatrix[0][x] = x;
    }
}
```

Implementation

Spell check using Levenshtein Distance Algorithm



```
SpellChecker.java C:\Users\pc\Documents\GRASP CSD (Java)
File Edit View Build Project Settings Tools Window Help
C:\Users\pc\Documents\
dictionary.txt
SpellChecker.class
SpellChecker.java
SpellChecker.java

    for (int x = 0; x <= n; x++) {
        testMatrix[0][x] = x;
    }

    for (int i = 1; i <= n; i++) {
        wordHolder = inputWord.charAt(i - 1);

        for (int j = 1; j <= m; j++) {
            wordHolder = testWord.charAt(j - 1);

            above = testMatrix[i - 1][j] + 1;
            left = testMatrix[i][j - 1] + 1;

            if (wordHolder == testWord.charAt(j - 1)) {
                diagonal = testMatrix[i - 1][j - 1];
            } else {
                diagonal = testMatrix[i - 1][j - 1] + 1;
            }

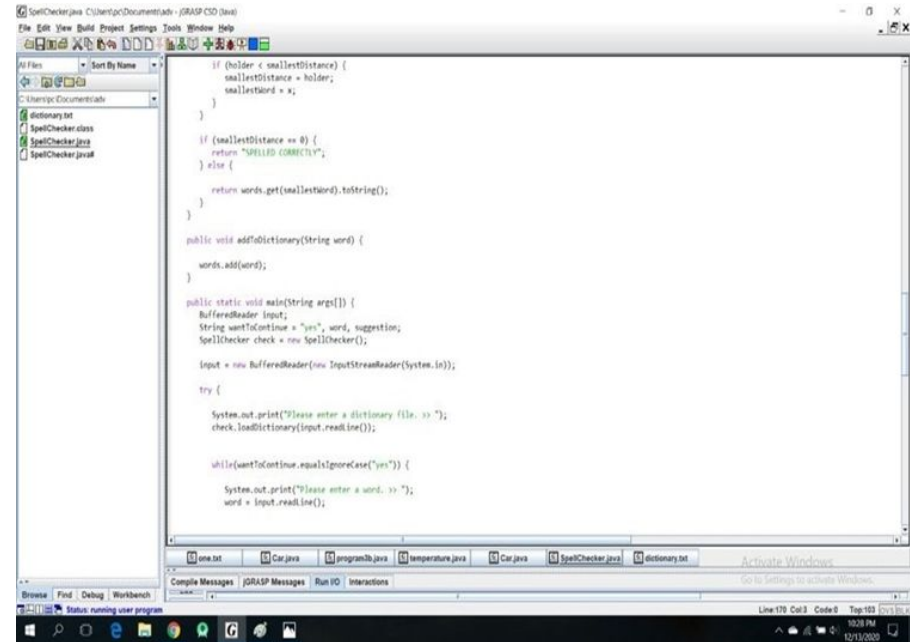
            testMatrix[i][j] = findTheMinimum(above, left, diagonal);
        }
    }

    return testMatrix[n][m];
}

public String findClosestMatch(String inputWord) {
    int smallestWord = -1, smallestDistance = 100, holder;

    for (int x = 0; x < words.size(); x++) {
        holder = findDifference(inputWord, words.get(x).toString());

        if (holder < smallestDistance) {
            smallestDistance = holder;
            smallestWord = x;
        }
    }
}
```



```
SpellChecker.java C:\Users\pc\Documents\GRASP CSD (Java)
File Edit View Build Project Settings Tools Window Help
C:\Users\pc\Documents\
dictionary.txt
SpellChecker.class
SpellChecker.java
SpellChecker.java

    if (holder < smallestDistance) {
        smallestDistance = holder;
        smallestWord = x;
    }

    if (smallestDistance == 0) {
        return "SPELLLED CORRECTLY";
    } else {
        return words.get(smallestWord).toString();
    }

    public void addDictionary(String word) {
        words.add(word);
    }

    public static void main(String args[]) {
        BufferedReader input;
        String wantToContinue = "yes", word, suggestion;
        SpellChecker check = new SpellChecker();

        input = new BufferedReader(new InputStreamReader(System.in));

        try {
            System.out.print("Please enter a dictionary file. >> ");
            check.loadDictionary(input.readLine());

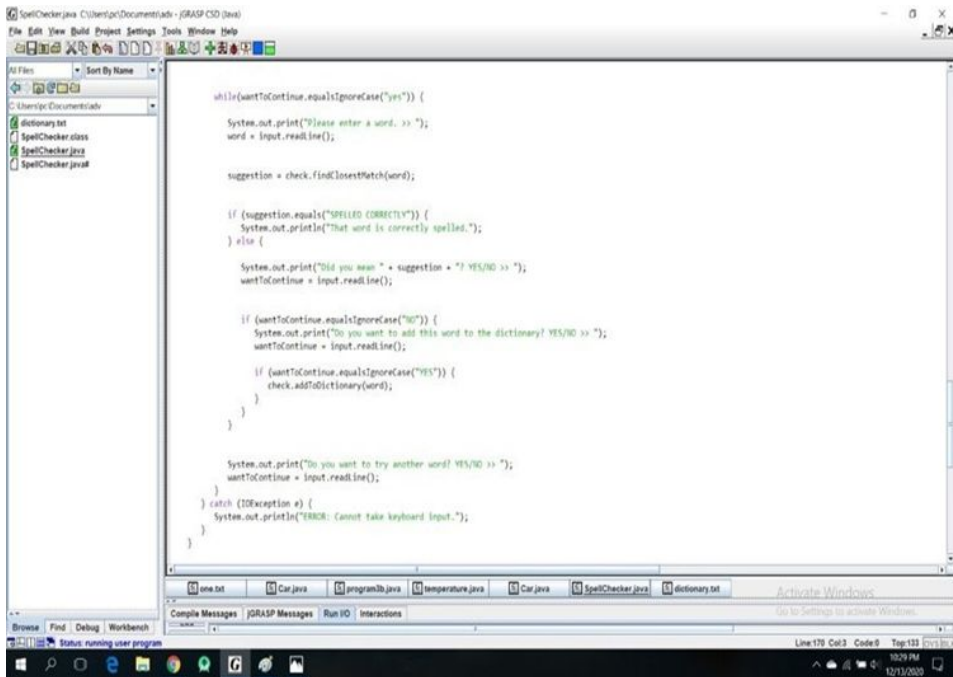
            while(wantToContinue.equalsIgnoreCase("yes")) {
                System.out.print("Please enter a word. >> ");
                word = input.readLine();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void loadDictionary(String filename) {
        try {
            BufferedReader reader = new BufferedReader(new FileReader(filename));
            String line;
            while ((line = reader.readLine()) != null) {
                words.add(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Implementation

Spell check using **Levenshtein Distance Algorithm**

OUTPUT



```
spellChecker.java C:\Users\pc\Documents\ads - jGRASP CSD (java)
File Edit View Build Project Settings Tools Window Help

All Files | Sort By Name
C:\Users\pc\Documents\ads
  dictionary.txt
  SpellChecker.class
  SpellChecker.java
  SpellChecker.java~

while(wantToContinue.equalsIgnoreCase("yes")) {
    System.out.print("Please enter a word. >> ");
    word = input.readLine();

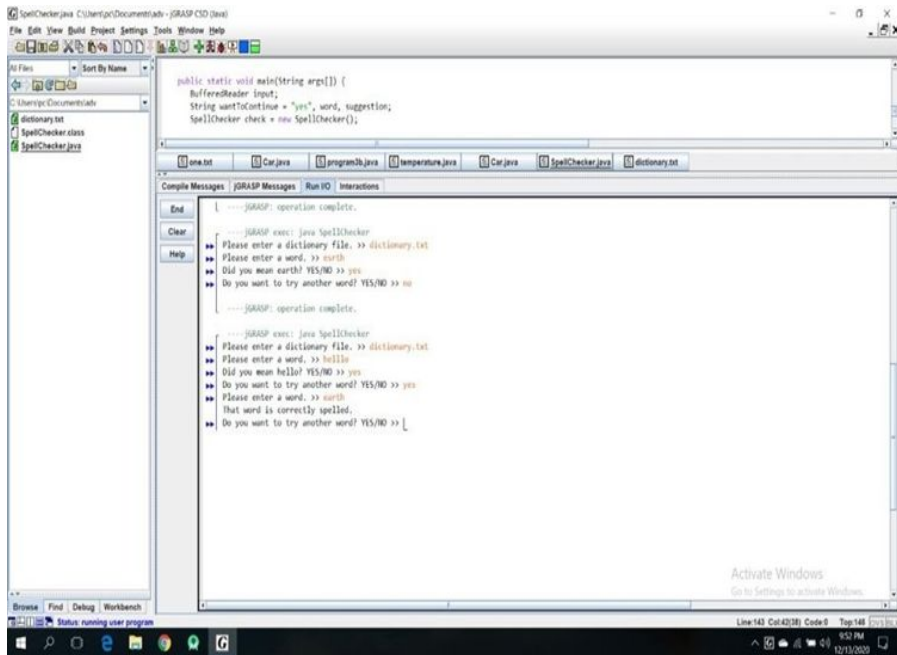
    suggestion = check.findClosestMatch(word);

    if (suggestion.equals("SPELLED CORRECTLY")) {
        System.out.println("That word is correctly spelled.");
    } else {
        System.out.print("Did you mean " + suggestion + "? YES/NO >> ");
        wantToContinue = input.readLine();

        if (wantToContinue.equalsIgnoreCase("no")) {
            System.out.print("Do you want to add this word to the dictionary? YES/NO >> ");
            wantToContinue = input.readLine();

            if (wantToContinue.equalsIgnoreCase("YES")) {
                check.addToDictionary(word);
            }
        }

        System.out.print("Do you want to try another word? YES/NO >> ");
        wantToContinue = input.readLine();
    }
} catch (IOException e) {
    System.out.println("ERROR: Cannot take keyboard input.");
}
```



```
spellChecker.java C:\Users\pc\Documents\ads - jGRASP CSD (java)
File Edit View Build Project Settings Tools Window Help

All Files | Sort By Name
C:\Users\pc\Documents\ads
  dictionary.txt
  SpellChecker.class
  SpellChecker.java

public static void main(String args[]) {
    BufferedReader input;
    String wantToContinue = "yes", word, suggestion;
    SpellChecker check = new SpellChecker();

    -----jGRASP: operation complete.

    -----jGRASP exec: java SpellChecker
    Please enter a dictionary file. >> dictionary.txt
    Please enter a word. >> earth
    Did you mean earth? YES/NO >> yes
    Do you want to try another word? YES/NO >> no
    -----jGRASP: operation complete.

    -----jGRASP exec: java SpellChecker
    Please enter a dictionary file. >> dictionary.txt
    Please enter a word. >> hello
    Did you mean hello? YES/NO >> yes
    Do you want to try another word? YES/NO >> yes
    Please enter a word. >> earth
    That word is correctly spelled.
    Do you want to try another word? YES/NO >> [
```


Conclusion

Here we are solving two problems i.e. spell checking and auto-complete, these both can be solved using Levenshtein Distance Algorithm where we are deciding the similarity of two words based on their levenshtein distance.

With the help of similarity we can auto-complete the words and also check if the word is spelled correctly.

Conclusion

Time analysis:

Best case: $O(m*n)$

As we are using 2 for loops to fill the edit distance matrix and get the levenshtein distance, $O(m*n)$ where m is the length of the test word and n is length of input word.

Worst case: $O(m*n*p)$

$O(m*n*p)$ where m is length of target word, n is length of input word, p is the place or index of the target word in the imported dictionary as we have to check the Levenshtein distance of each word to find the match in spell check as well as in autocomplete.

References

1. **Autocomplete and Spell Checking Levenshtein Distance Algorithm to Getting Text Suggest Error Data Searching in Library**
Muhammad Maulana Yulianto, Riza Arifudin, Alamsyah
2. **Research on String Similarity Algorithm based on Levenshtein Distance**
Shengnan Zhang, Yan Hu, Guangrong Bian
3. **Levenshtein Distance Algorithm for Efficient and Effective XML Duplicate Detection**
Mrs.Shital Gaikwad, Prof.Nagaraju Bogiri
4. **Levenshtein Distance Algorithm Analysis on Enrollment and Disposition of Letters Application**
Sugiarto, I Gede Susrama Mas Diyasa, Ilvi Nur Diana
5. **Using the Levenshtein Edit Distance for Automatic Lemmatization: A Case Study for Modern Greek and English**
Dimitrios P. Lyras, Kyriakos N. Sgarbas, Nikolaos D. Fakotakis
6. **Levenshtein Distance, Sequence Comparison and Biological Database Search**
Bonnie Berger, Michael S. Waterman, and Yun William Yu

References

7. **Learning String-Edit Distance**
Eric Sven Ristad and Peter N. Yianilos
8. **Kernels Based on Weighted Levenshtein Distance**
Jianhua XU and Xuegong ZHANG
9. **Computing the Levenshtein Distance of a Regular Language**
Stavros Konstantinidis
10. **Plagiarism Detection Using the Levenshtein Distance and Smith-Waterman Algorithm**
Zhan Su, Byung-Ryul Ahn, Ki-yol Eom, Min-koo Kang, Jin-Pyung Kim, Moon-Kyun Kim
11. **A Normalized Levenshtein Distance Metric**
Li Yujian and Liu Bo
12. **String correction using the Damerau-Levenshtein distance**
Chunchun Zhao and Sartaj Sahni
13. **Adapting the Levenshtein Distance to Contextual Spelling Correction**
Si Lhoussain Aouragh, Hicham Gueddah and Abdellah Yousfi
14. **A Comparison of Standard Spell Checking Algorithms and a Novel Binary Neural Approach**
Victoria J. Hodge and Jim Austin
15. **Spelling Checker Algorithm Methods for Many Languages**
Novan Zukarnain, Agung Trisetyarso, Bahtiar Saleh Abbas, Chul Ho Kang and Suparta Wayan