# String processing algorithms

Ordean Tiberiu-Vasilian - 322CD
University Politehnica of Bucharest, Faculty of Automatic Control and Computers
tiberiu.ordean@stud.acs.upb.ro

**Keywords**: Boyer-Moore, Edit distance, Levenshtein, LCS (Longest Common Subsequence)

**Abstract:** This project presents the analysis of 2 string processing algorithms , Edit distance and Boyer-Moore and focuses on their performance , their complexity and on different approaches of using string operations regarding these algorithms .

# 1. Introduction

### 1.1. Description of the solved problem

String processing algorithms are some of the most important algorithms used in programming. They can be used in different situations and with different purposes . For instance, string matching algorithms are widely used as they help finding close matches to what someone may give as input. String editing algorithms help analyse  the differences between strings and offer the possibility  to edit them or find specific information such as the length ,the position of a specific char, etc.

### 1.2.  Practical use

The string processing algorithms are used daily by a wide variety of people in different tools such as Google, Bing, DuckDuckGo or other search engines. One of their roles is to find the closest match to what someone offers as input from their keyboard . Another practical use of string processing algorithms is in Plagiarism Detection such as MOSS or Copyleaks . Furthermore, they are used in Spell Checker tools used by various messaging and social media applications or even by Google .

### 1.3.  Chosen solutions

My choices for this project are the Boyer-Moore algorithms and also the Edit distance algorithm . For the edit distance I decided to use The Levenshtein distance and the Longest common subsequence(LCS) distance.

The edit distance algorithm is widely used for evaluating how dissimilar two strings are by counting the minimum number of operations necessary for matching two different strings.

The Boyer-Moore algorithm is considered one of the most efficient string-matching algorithms and is is the benchmark for literature string-searching.

### 1.4. Validation criteria

The validation of the implementation will be done by 10 manually generated tests (for each algorithm) that are specifically made in order to evaluate the complexity and the run time of the algorithms. The correctitude of the algorithm si done by the comparison of the input and the output, also done manually.

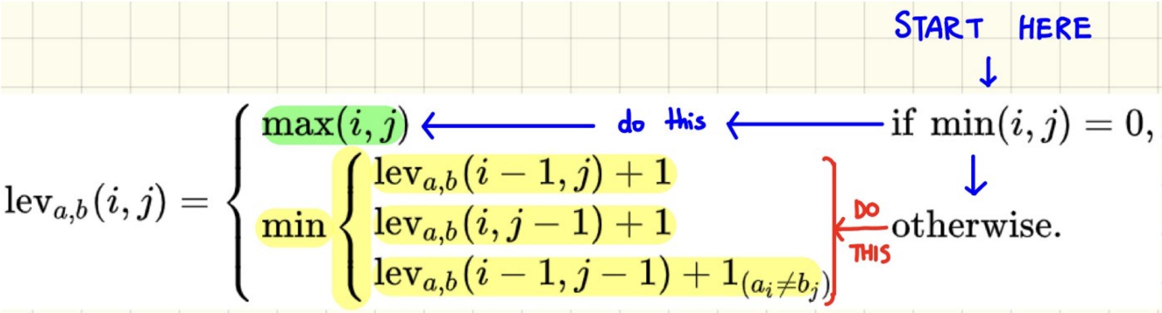
## 2. Analysis of the chosen algorithms

### 2.1. EDIT DISTANCE

**a. Functionality**

Edit distance is a classic problem used in dynamic programming and has a base complexity of O(mn) if we are using a matrix or O(n) if we are using two arrays . The basic edit distance accepts the uses of insertion , deletion and substitution. The main idea of the edit distance is to compare two string character by character and find the similarities . After that, it is necessary to delete , insert, or substitute a character in order to have the same word. The output consists in the number of operation used in order to get to the desired result. For example , let’s say the two words given are “march” and “mars” and we also have the length of both strings , in this case 5 and 4. The first step is to find the similar letters , which are ‘mar’ , the first three. Then, in order to make the first string be the same as the second we need to replace ‘c’ with an ‘s’. Then we need to delete the ‘h’. The output in this case will be 2 because we needed 2 operations in order to obtain our result.

In the picture below I present the mathematical formula I used for implementing the Levensthein distance.



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

The difference between the implementation of Levenshtein distance and LCS distance is that the first one allows using all three of the basic Edit distance operations, while LCS allows only using insertion and deletion.

### b. Complexity Analysis

Normally the classic solution for both LCS and Levenshtein has a basic time complexity of O(mn) . For this solution and for better understanding of the complexity I will present the following calculus :

- The matrix m[i][j] = the minimum number of operations that need to be made on the first ‘i’ characters from the first string in order to get the first ‘j’ characters from the second string
- The recursion is m[i][j] = min(m[i-1][j] + 1, m[i][j-1] + 1, m[i-1][j-1] + eval(i, j) )
- eval will return true if the character on the position ‘i’ from the first string is different than the character on the position ‘j’ in the second string, and will return false otherwise
- The dynamic will be initialised as such : m[i][0] = m[0][i] = i and the needed result will be in m[N][M]

- Let’s say there is an additional restriction such as the following: ’the edit distance of two strings must be less or equal than k’ . This restriction can be used in order to lessen the number of calculus made . For any line ‘i’ of the matrix ‘m’ is sufficient to calculate only the cells whose column is between ‘i - k’ and ‘i + k’ because the cells outside of this interval represents strings for which the length difference is > k.  This means that at least k + 1 operations will be necessary in order to make the strings to be the same . Because only the lines ‘i - 1’ and ‘i’ are used, the memory consumption will be reduced . This way we can apply the edit distance using only two arrays , performing a conversion of complexity from O(n^2) to O(n). Using these optimisations the time complexity will be O(nk).

### c. Advantages and disadvantages

Edit distance is an easy algorithm to implement recursively, with very compact code . Moreover , it is pretty fast with shorter strings in both LCS and Levenshtein form . This algorithm behaves very well even if we do not use alphabetical characters, meaning that it accepts as input any of the ASCII characters , making its capabilities broad enough for future practical uses.

On the other hand , when given longer sentences the speed of the algorithm will decrease significantly ,in both LCS and Levensthein forms. This limits the algorithm to only perform at maximum capacity when given shorter strings that require less operations.

## 2.2. BOYER-MOORE

### a. Functionality /b. Complexity Analysis

Boyer-Moore is a fast pattern searching algorithm that outputs the position where a given pattern is found within a given string. This algorithm has a base complexity of O(mn) . For example if we are given as input string1 = “MTPEWGAKSD” and string2 = “ WGA” , string2 is considered the pattern and the algorithm will search in string1 for the appearances of string2. In this case “WGA” appears only once in string1, after the fourth position and the output in this case will be 4.

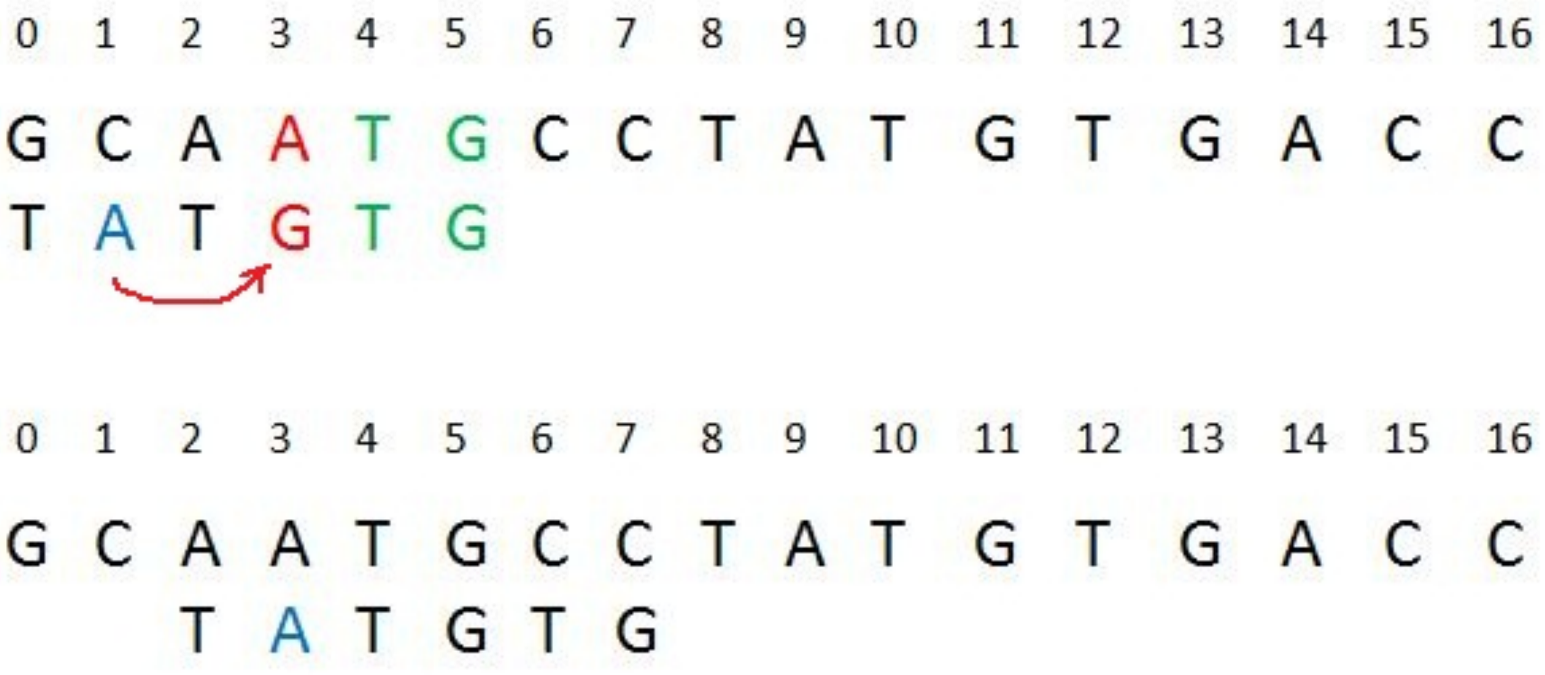


fig. 1

In the picture above we are looking in the pattern(“T A T G T G”) for the last matching characters ’T’ and ‘G’ which are at the positions 4 and 5 . Doing that we find the last mismatching character which is ‘A’ , highlighted in red . The next step is to look in the pattern for the last appearance of ‘A’. In this case we find it at position 1 . Next we need to shift ‘x’ times in order to align the ‘A’(highlighted) in blue with the mismatching ‘A’ in the string . In this case we are shifting 2 times.

In the second picture we are repeating the process, only this time we are starting with a mismatching character, ‘C’(position 7). Looking for ‘C’ in the pattern we couldn’t find it, so we need to shift the pattern after the 7th position in order for the algorithm to properly work. The reason for this shift is the following: because ‘C’ does not exist in the pattern , any shift before position 7 would have been a mismatch . In this case , after the last shift, we get a match(highlighted in green). In this case our output will be 8 .

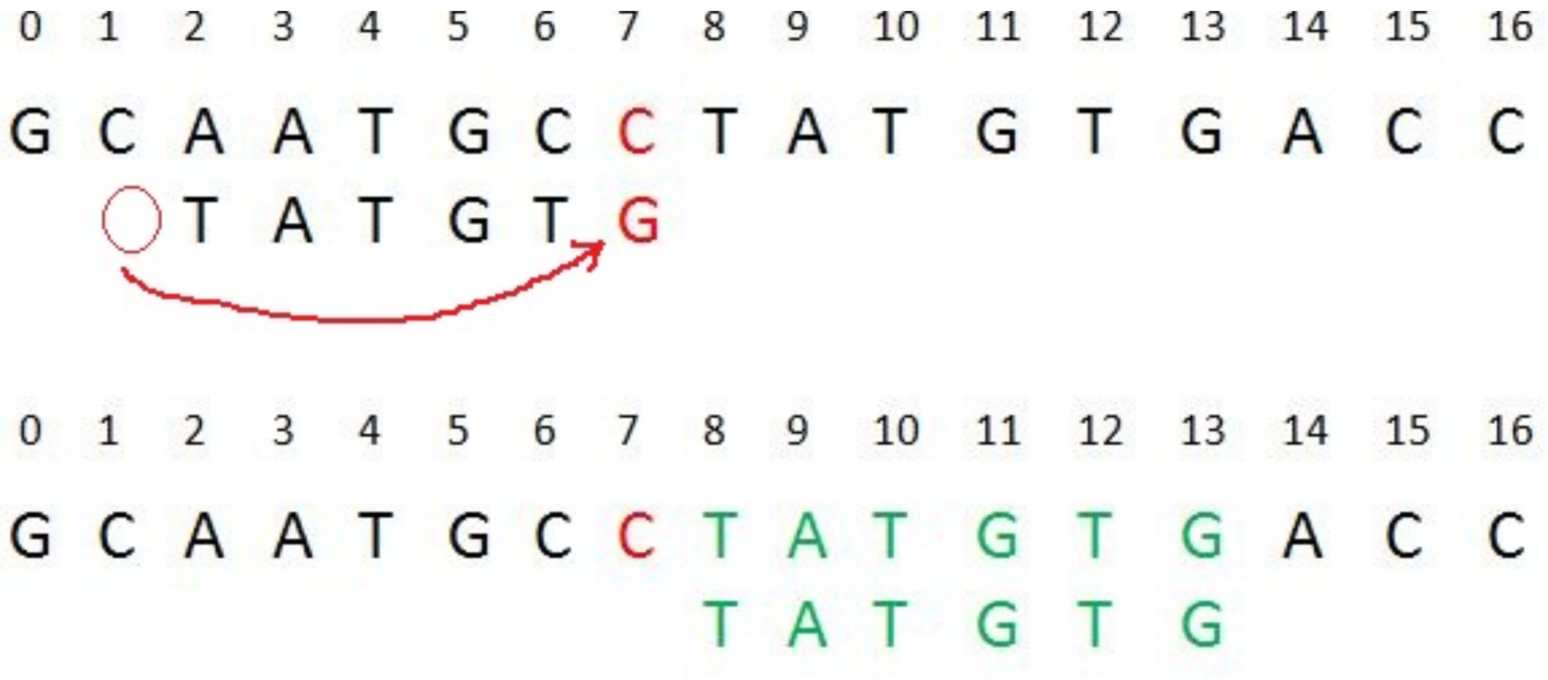


fig. 2

This example explains very clear the functionality of Boyer-Moore and also suggests the importance of a fast pattern searching algorithm in computer science.

The preprocessing phase is done in O(m+σ) time and space complexity.

The Boyer-Moore searching phase has a worst case running time of O(m + n) if the pattern is not found in the initial text. If the pattern is found in the initial text , the complexity is O(mn) in the worst case.

The best performance of the Boyer-Moore algorithm is O(m/n) .

### c. Advantages and disadvantages

The biggest advantage of the Boyer-Moore algorithm is its best case complexity of O(m/n) which is the absolute minimum for any string-matching algorithm in the model where the pattern only is preprocessed. Another advantage is that the longer the pattern , the faster the algorithm . Moreover , Boyer-Moore is considered the standard benchmark for practical string-search literature, meaning that the practical use of this algorithm is at an increased level. In addition The Boyer–Moore algorithm uses information gathered during the preprocess phase to skip sections of the text, resulting in a lower constant factor than many other string search algorithms.

On the other hand , the algorithm has a pretty difficult implementation because if it is not implemented correctly , it will only output the position of the first appearance of the pattern, even if the pattern can be found more than once in the initial text.

## 3. Performance evaluation

### 3.1. Evaluation criteria

- For evaluating the algorithms I created 10 tests for each algorithm that receive as input some different kind of data such as strings with a shorter or a longer length ,with and without special characters such as 'spaces' , 'question marks', 'comas' etc.
- The first step was to manually create 10 tests for each algorithm and put them in an input folder. Then, I read all the info from each test and printed them in an output.
- I manually verified each output to assure it writes a correct answer.
- For the Edit distance algorithm I used strings that need all three(insertion , deletion and substitution for the Levenshtein distance) or

two(insertion and deletion for LCS distance) operations in the same process.

- Running the tests for the Edit distance it was visible that for strings with less resemblance that needed more operations , the run-time was longer. On my computer it became visible after 20 necessary operations. This factor displays the biggest disadvantage of the edit distance algorithm  which is the longer running time, for longer sentences with more differences between them.
- For the Boyer-Moore algorithm I used smaller patterns that repeat multiple times in the given string in order to see the complexity and the functionality of the algorithm. I also tried to use patterns that can not be found in the string.
- In addition , I used non-alphabetical characters for both pattern and initial string .
- For the Boyer-Moore algorithm I did not have any problems with the running speed and with the memory consumption as I had with the edit distance algorithms.

### 3.2. System specification

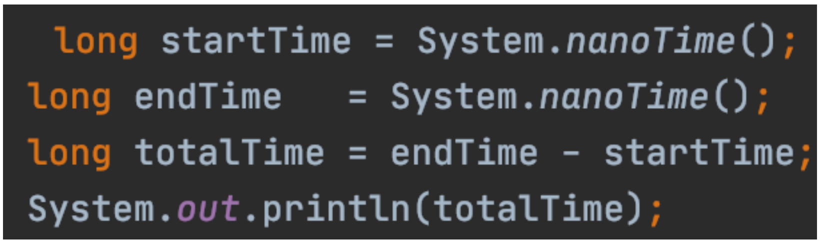
The code has been compiled using Javac, the the main compiler included in the JDK package . It was run on a system with macOS 64-bit machine, Intel Core i5 Quad-Core 1.4 GHz processor , 8GB LPDDR3 2133 RAM and SSD Drive with remaining memory of approximative 40GB.

I want to mention that I have also tested the code for the second phase of the project on a 64-bit system of a Linux virtual machine.

### 3.3. Evaluation of solutions on the test set

- All the tests I provided were successful but some required a longer running time than others.

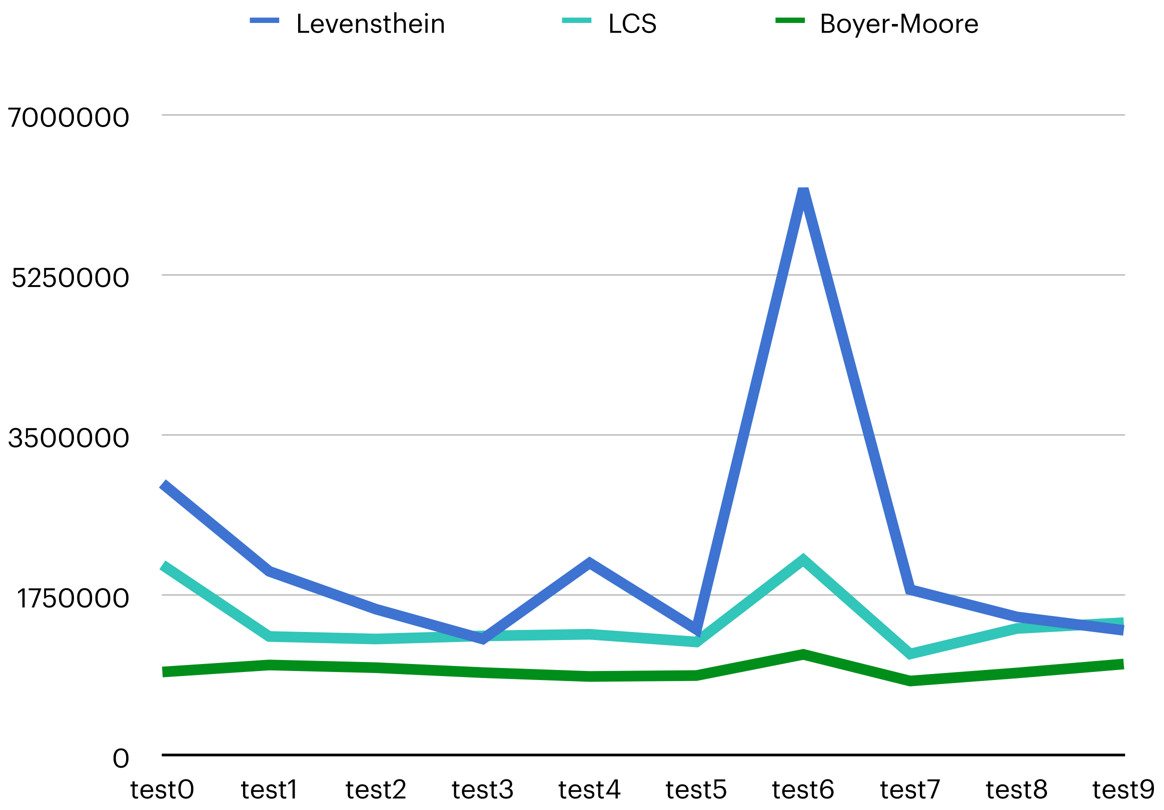
- To obtain the running time for each test I used the method below:



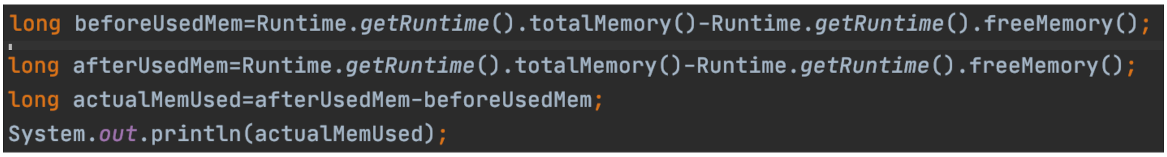
```
 long startTime = System.nanoTime();
long endTime   = System.nanoTime();
long totalTime = endTime - startTime;
System.out.println(totalTime);
```

- The table for the running time of each test(**in nanoseconds**)

| Test Number | Levenshtein | LCS | Boyer-Moore |
|---|---:|---:|---:|
| **test0** | 2980150 | 2090746 | 921138 |
| **test1** | 2020727 | 1309003 | 997601 |
| **test2** | 1607029 | 1281490 | 968280 |
| **test3** | 1280983 | 1315146 | 914046 |
| **test4** | 2112194 | 1332937 | 872210 |
| **test5** | 1386073 | 1249050 | 882434 |
| **test6** | 6199910 | 2146346 | 1113351 |
| **test7** | 1820779 | 1117767 | 822645 |
| **test8** | 1522459 | 1396398 | 910479 |
| **Test9** | 1376072 | 1459754 | 1008124 |



- To obtain the memory consumption I used the following method and I divided everything by 1024 to get the value in “KB”.

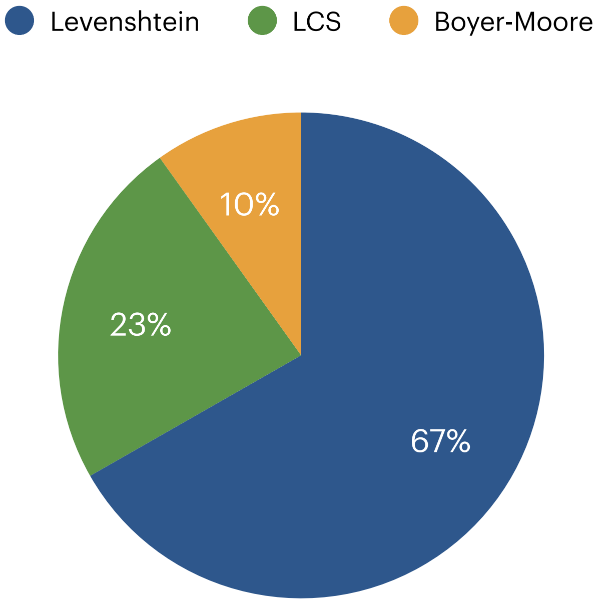


```
long beforeUsedMem=Runtime.getRuntime().totalMemory()-Runtime.getRuntime().freeMemory();
long afterUsedMem=Runtime.getRuntime().totalMemory()-Runtime.getRuntime().freeMemory();
long actualMemUsed=afterUsedMem-beforeUsedMem;
System.out.println(actualMemUsed);
```

- The memory consumption table:

| Levensthein(KB) | LCS(KB) | Boyer-Moore(KB) |
|---:|---:|---:|
| 1513 | 530 | 224 |

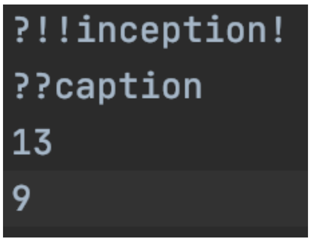
- The memory consumption chart that shows how the entire memory of the project is distributed:



### 3.4. Test values

- The difference  of run-time between LCS & Levensthein and Boyer-Moore is significant . The  majority of the tests for the first two algorithms are in the 7 digit zone, while for Boyer-Moore it varies in the 6 digit zone, proving the fact that the Boyer-Moore algorithm is one of the fastest string-processing algorithms.

- The run time graphic above represents in the best way the diversity of the tests I created for each algorithm  and also outputs the run-time for each algorithm.
- The resulting values suggest that the Levensthein algorithm has a longer running-time, but yet very similar to the LCS algorithm running time in some cases(for test3, test5, test8).
- The only unexpected result comes from test6 and I believe the difference is only made by the difficulty of the test as it can be seen that both LCS and Levenshtein running time increased .



```
?!!inception!
??caption
13
9
```

I believe that both the length of the strings and the differences between them cause for the longer running time of the program.

## 4. Conclusion

Considering all the data and the analysis, the Boyer-Moore algorithm is best suited for literature string-searching because it is very fast and offers a good performance . It is sufficiently optimised for real usage scenario , and it is already used in different domains , giving the security of a correct usage of the algorithm. Moreover, the Edit distance algorithm could be used in Plagiarism Detection , but not on a very advanced level , because of the difficulty the algorithm presents when given longer inputs.

## 5. References

https://www.geeksforgeeks.org/edit-distance-dp-5/

https://en.wikipedia.org/wiki/String-searching_algorithm

https://www.geeksforgeeks.org/applications-of-string-matching-algorithms/

https://www.geeksforgeeks.org/edit-distance-and-lcs-longest-common-subsequence/

https://en.wikipedia.org/wiki/Edit_distance

https://en.wikipedia.org/wiki/Boyer–Moore_string-search_algorithm

https://www.infoarena.ro/girls-programming-camp-2011/selectie/solutii/edist