Wagner-Fiscer Algorithm: A comparison between algorithmic approaches

Pedro Miguel Rocha Oliveira

***Resumo* – O algoritmo de Wagner-Fischer é uma maneira de calcular a distância de Levenshtein entre palavras ou strings, usado, por exemplo, no campo de teoria de informação.**

Existem várias maneiras de implementar o algoritmo. Neste artigo, vou apresentar três abordagens: uma baseada em recursividade, outra em recursividade com memoization e outra baseada em dynamic programming; e comparar os três algoritmos na sua complexidade de tempo e de memória, assim como a eficiência em tempo de execução e número de operações realizadas.

***Abstract* – The Wagner-Fischer algorithm is a way to calculate the Levenshtein distance between words or strings used, for instance, in the field of Information Theory.**

There are various ways of implementing the algorithm. In this article, I will go over three approaches: a recursive-based function, a function that uses recursivity and memoization, and a dynamic programming algorithm. I will then compare the three algorithms in terms of their complexity in time and memory, as well as their efficiency in execution time and number of operations.

I. INTRODUCTION

The Wagner-Fischer algorithm is an algorithm that calculates the Levenshtein distance between two strings of characters. This distance tells us the minimum number of changes that need to be made so that one string changes to another. For example, consider the words 'Sunday' and 'Saturday'; for them to become the same string, we need to follow three changes:

1. Sunday → Saunday (insertion of an 'a')
2. Saunday → Satunday (insertion of a 't')
3. Satunday → Saturday (replace the 'n' with an 'r')

This means the two words have a Levenshtein distance of 3.

This value is used in information theory, for example, in spell checkers, to detect small typos in a text or in a search query, to improve the user experience of someone using a text editor or a search engine. Furthermore it also works as a metric to quantify linguistic distance and check how different two languages are from one another.[1]

The standard way of implementing this algorithm is with a dynamic programming approach, which means implementing an iterative algorithm that would save the values needed in memory and go bottom-up until it reaches the words needed to compare. This means the words "Sunday" and "Saturday" would start by comparing the empty strings and go change by change until it reaches the original two words.

This is distinctly different from the recursive implementation, that simply starts with the original words and, using a top-down strategy, calculates the distance until it reaches the empty string. In addition to this algorithm, it's possible to implement a memoization technique and save the new results in a data structure. This improves the algorithm, since it means there's no need to redo previous calculations.

In this article, I will go over the execution times and the overall computational complexity of each algorithm. In section 2, I will go over the pseudo-code for each algorithm, and go over their complexity. In section 3, I will explain the tests done to test the algorithms and compare each result. Finally, I will conclude in section 4 how scalable each algorithm is. All the algorithms were implemented in Python 3.7.

II. ALGORITHMIC IMPLEMENTATIONS

The more famous implementation of the Wagner-Fischer algorithm is based on dynamic programming. Taking this version as my base, I then implemented the variations necessary for the recursive function and the memoization algorithm.

A. Dynamic Programming Implementation

As seen in [2], the pseudo-code for the algorithm is fairly simple to implement. Considering the strings *s* and *t* as our starting parameters, we first need to declare an integer matrix *d* MxN set to 0s, with M being the length of string *s* plus 1 and N being the length of string *t* plus 1.

Next, we set each element in the first column and the first row equal to its index. This means in that in order to turn *s* or *t* into the empty string we'd need to remove all characters, making it a Levenshtein distance of M or N respectively.

Finally, we set the rest of the elements in the matrix as the minimum of the following:

1. d[i-1, j] +1, implying we deleted a character
2. d[i, j-1] +1, implying we inserted a character
3. d[i-1, j-1] + sub, implying we have replaced a character

This sub variable is dependent on wether or not the letter from *s* at position *i* is the same as the letter from *t* at

position *j*. If it is the same, sub is set to 0, as it means we don't actually need to replace anything, otherwise it's set to 1.

As such, it's possible to understand how this is a dynamic programming algorithm, as we're starting from the initial positions in the matrix and going one by one from there until we reach the final position (m, n) which would represent our Levenshtein distance. This makes it an iterative algorithm.

While this article will prove this empirically on Section 3, it's already possible to estimate the complexity of this implementation, both in terms of execution time and in terms of memory. Since I'm storing everything in a matrix that's MxN, the memory used is O(MxN), and since we have to check and fill every element in this matrix, the execution complexity is around O(MxN) as well.

B. Recursive Implementation

Based on the dynamic programming implementation explained above, it was fairly easy to alter it in order to become a purely recursive method.

Again, I have to start with string *s* and *t* as arguments. My exit conditions are simple, if either of these strings are empty, the Levenshtein distance is equal to the length of the non-empty string.

The recursive call would come next as three conditions specified in the last implementation, except now we're taking only a substring of the original. Meaning instead of using a matrix, we're only cutting the original word and taking out the last index, as follows:

1. rec(s[:-1], t) + 1, implying we deleted a character
2. rec(s, t[:-1]) + 1, implying we inserted a character
3. rec(s[:-1], t[:-1]) + sub, implying we have replaced a character

where rec is our recursive function.

This follows the basic logic of a recursive function, meaning we're starting from the most complex element, the full strings, and making recursive calls until we find a simple case, i.e. the empty string.

Again, it's possible to estimate the complexity of this function. In terms of memory, other than recursive calls, there is nothing to be saved on memory, so this is simply an O(a), with *a* being a constant. In terms of computation, this becomes a $O(a^{m*n})$, which means it grows exponentially as the strings grow in size.

C. Recursive Implementation with Memoization

The final variation is one that uses the last algorithm's recursive calls, as well as a data structure like the one defined in the Dynamic Programming algorithm.

With this data structure, I'll be saving the results from the recursive call, and another exit condition was added. Now I'm checking if calculation has been previously done by checking the element at the position equal to the length of both strings that are being sent as arguments. If it is, then that value is returned.

So in this implementation, the complexity is very similar to that in the Dynamic Programming method. So, in terms of memory, we need to reserve a O(M*N) matrix, and in terms of computation, we can expect a O(M*N) times.

III. TESTS AND RESULTS

Now that all the algorithms have been implemented and tested with simple examples ('Saturday'/'Sunday', 'Kitten'/'Sitting', 'Surgery'/'Survey'), all the algorithms are ready to be tested in terms of execution times and number of basic operations.

The test uses a random string generator function that will return a string formed with random letters (both lowercase and uppercase) with the length that was passed as argument.

The initial plan was testing every algorithm with an exponentially growing string, starting with 1 character up until 5000 characters. Unfortunately, it was quickly made obvious that the recursive function was too computationally taxing to be able to return results in a proper time frame.

So instead I grew the string iteratively, and stopped it at the 12 character mark for all algorithms. Then I resumed the running for the rest of the tests as planned previously, in a exponential growth stopping at 5000 characters.

There was, however, another setback. At 200 characters, the recursive algorithm would reach the limit defined in python as the recursive depth. While I increased this limit to 5000 (as opposed to being previously at 1500) it only allowed me to reach a string with 1000 characters. This then became my stopping point.

With the basic control variable now fixed for the tests, I only had to keep track of execution time and number of basic operations that were done. While I made sure not to have other programs running in the background, the execution time might still be influenced by other SO-related processes. Furthermore, the calculation of the number of basic operations was based in how many explicit sums were done, ignoring function calls and data access.

Finally, I will start showcasing each algorithm's overall results and a final comparison in the end.

A. Results with the Recursive Implementation

As expected from the section above, the recursive implementation takes an exponential growth in execution time and operation count. As can be seen in Figure 1, which relates the Operation Count with the length of the String, and in Figure 2 with relates to the Execution Times, it's possible to see the exponential curve.

However, this might still not be good enough, as it might be an unexpected polynomial of higher degree. Therefore, I'm also comparing with graphs done with the y-axis in the logarithmic scale. If we're seeing things in the

logarithmic scale, we can expect the graph to look closer to a straight line, while a polynomial would tend to have a downward slope. So, as shown in Figure 3 and 4, both operation count and execution time grow exponentially by a certain facto X, finally confirming the initial hypothesis that this would be an algorithm with $O(a^{m*n})$ in terms of computational complexity.

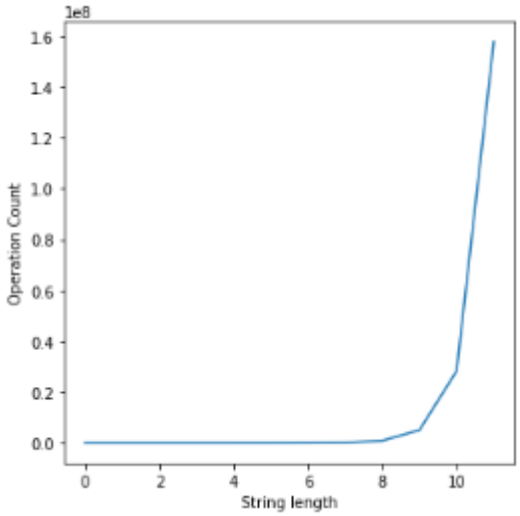

Fig. 1 - Graph relating the number of characters in the strings with the number of operations done.

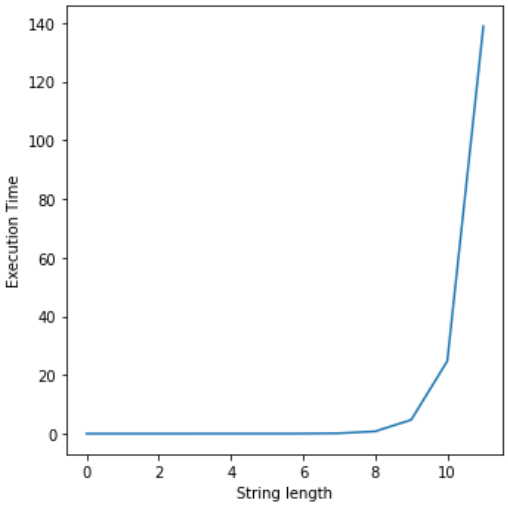

Fig. 2 - Graph relating the number of characters in the strings with the execution time in seconds.

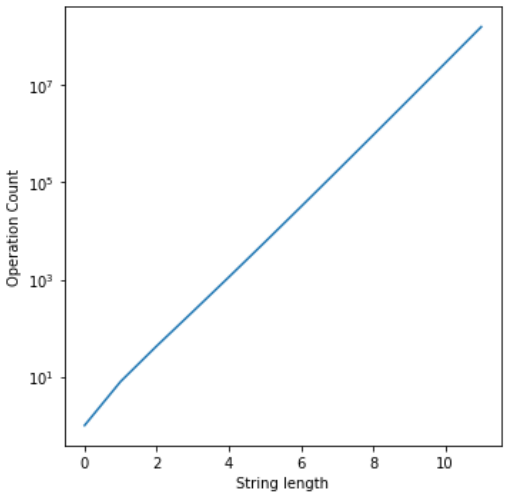

Fig. 3 - Graph relating the number of characters in the strings with the number of operations, using a logarithmic scale.

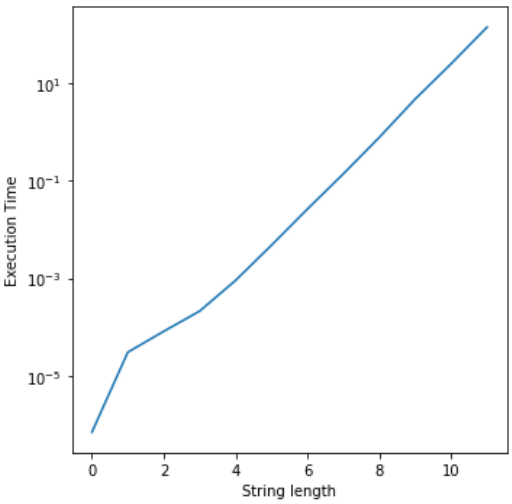

Fig. 4 - Graph relating the number of characters in the strings with the execution time, using a logarithmic scale.

B. Results with the Recursive and Memoization Implementation

With the iterative component of the test, Figures 5 and 6 showcase a graph that is showing a growth much smaller than the last algorithm, indicating that the algorithm's complexity isn't exponential, but rather polynomial.

This is further evidenced by the graphs 7 and 8 that, much like previously, were altered to have a logarithmic scale in its Y axis. The existence of the downward slope shows that the growth slows down. This means that the complexity of the algorithm is of $O(n^2)$, or as I had estimated, O(MxN) with M being the length of one string and N the length of the other.

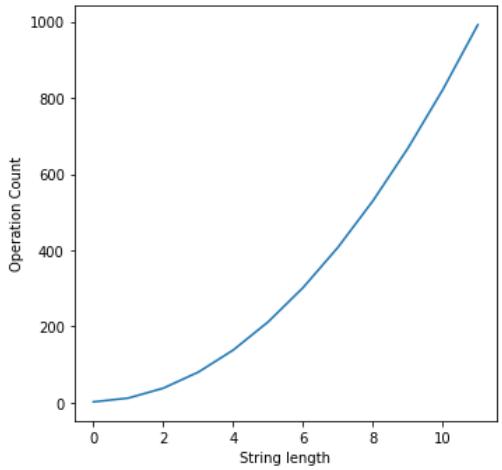

Fig. 5 - Graph relating the number of characters in the strings with number of operations for the memorization algorithm.

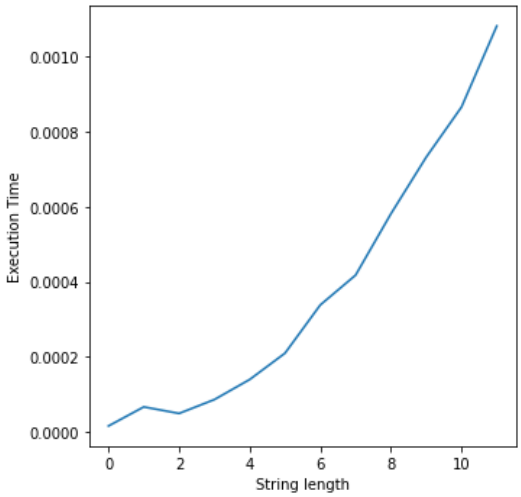

Fig. 6 - Graph relating the number of characters in the strings with the execution time for the memorization algorithm.

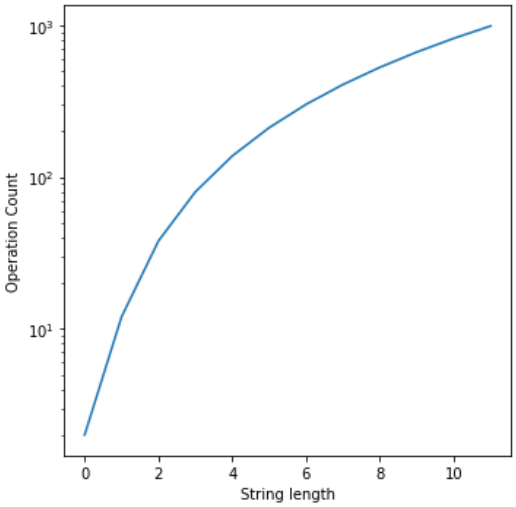

Fig. 7 - Graph relating the number of characters in the strings with the number of operations for the memorization algorithm, using a logarithmic scale.

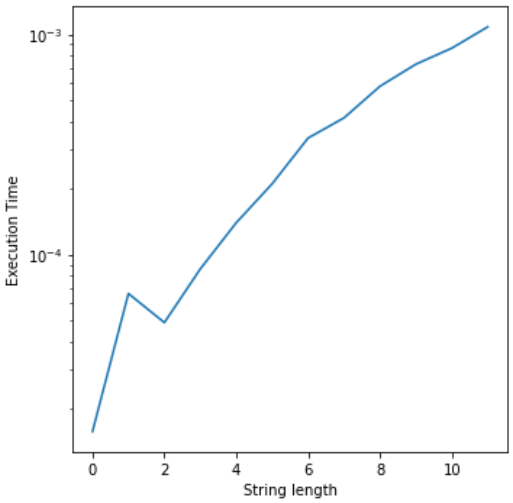

Fig. 8 - Graph relating the number of characters in the strings with the execution time for the memorization algorithm, using a logarithmic scale.

C. Results with the Dynamic Programming Implementation

As with the previous algorithms, in this section I'll show the results this algorithm had with the iteratively growing strings. The results look pretty similar to the previous algorithm, however with slightly better times and number of operations.

It should be noted, however, that the slopes present in the Figures 9, 10, 11 and 12 look parallel to the ones obtained with the results from the Memoization algorithm. This means the complexity of the algorithm is similar to the latter, as was stated in Section 2, i.e O(MxN) with M and N being the length of the strings passed as parameters.

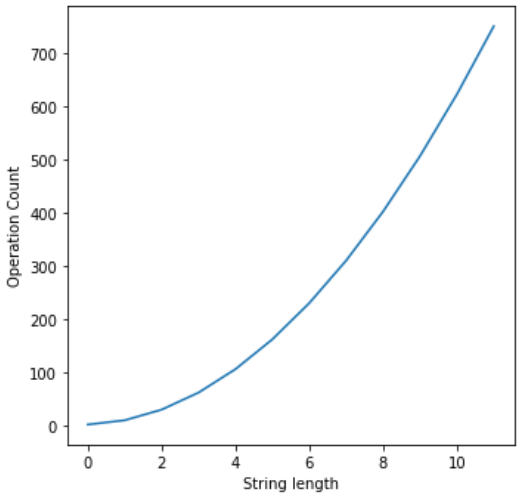

Fig. 9 - Graph relating the number of characters in the strings with number of operations for the dynamic programming algorithm.

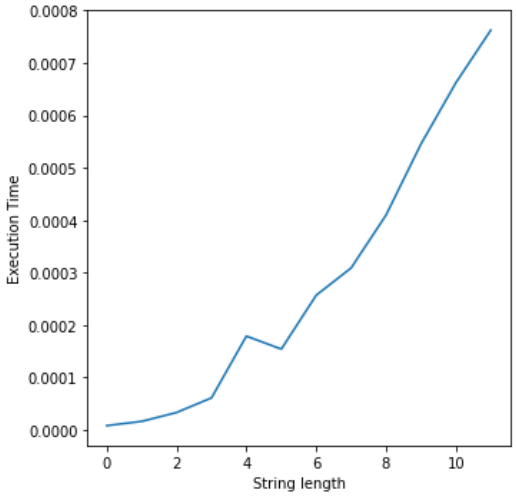

Fig. 10 - Graph relating the number of characters in the strings with the execution time for the dynamic programming algorithm.

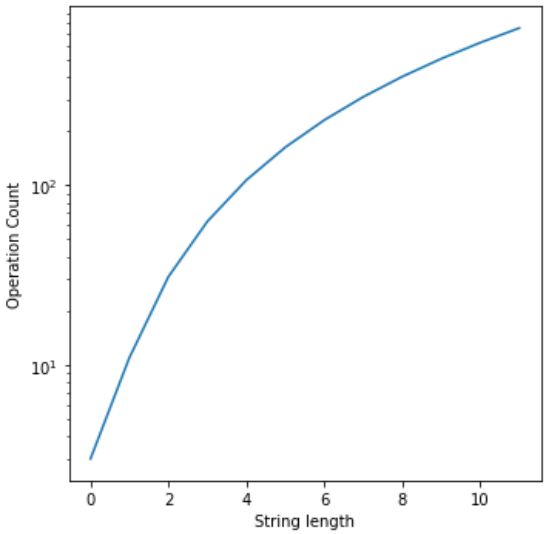

Fig. 11 - Graph relating the number of characters in the strings with the number of operations for the dynamic programming algorithm, using a logarithmic scale.

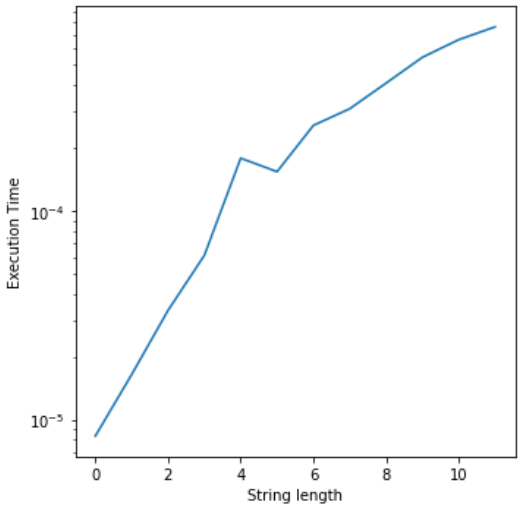

Fig. 12 - Graph relating the number of characters in the strings with the execution time for the dynamic programming algorithm, using a logarithmic scale.

D. Comparison between the Algorithms

Now that we have seen the results for each individual algorithm, this doesn't quite show us how they fare against each other. While the purely recursive algorithm clearly lags behind against the other two algorithms, there isn't a clear answer between the memoization algorithm and the dynamic programming.

So in the second part of the test, with a string that is growing exponentially until it reaches 1000 characters, the results in Figure 13 and 14 show a more clear comparison. In these figures, the memoization algorithm consistently presents higher results in terms of number of operations executed and in terms of execution time.

Interestingly, when the Y axis scale was changed to a logarithmic scale, like in Figures 15 and 16, it was

possible to see that the Memoization results are almost parallel to the Dynamic results, making the difference between both constant.

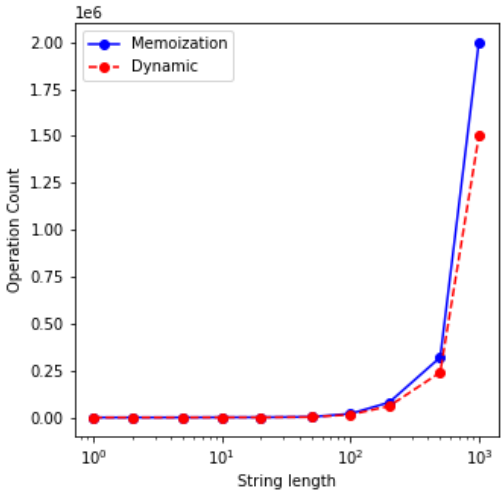

Fig. 13 - Graph comparing the operation count for larger strings of characters between the memoization and the dynamic programming algorithms

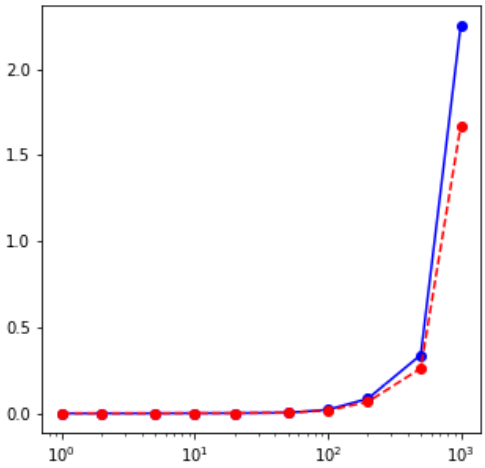

Fig. 14 - Graph comparing the execution time for larger strings of characters between the memoization and the dynamic programming algorithms

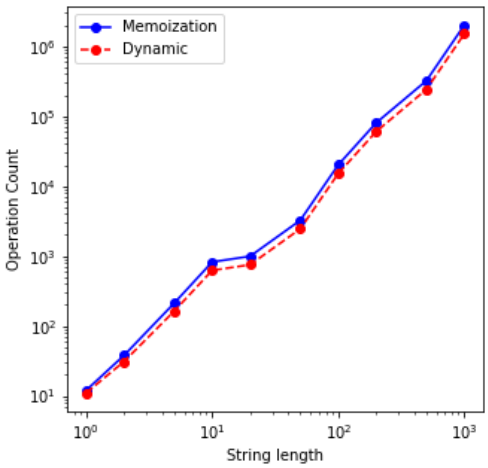

Fig. 15 - Graph comparing the operation count for larger strings of characters between the memoization and the dynamic programming algorithms, in a log scale

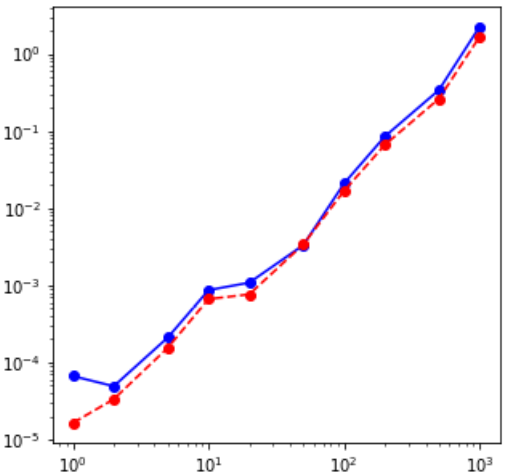

Fig. 16 - Graph comparing the execution time for larger strings of characters between the memoization and the dynamic programming algorithms, in a log scale

IV. CONCLUSION

In this article, I have gone over three different implementation of the Wagner-Fischer Algorithm, showcasing three different approaches: recursive, recursive with memoization and dynamic programming.

After implementing this, the tests with increasingly greater input parameters, we can see that the dynamic programming approach slightly beats out the memoization approach, and that the recursive approach is too computationally difficult to be a genuine solution.

Furthermore, by following the graphs, we can expect that for an input string length of around 5000 characters would actually take around 250 seconds with the memoization algorithm and 144 with the dynamic programming algorithm, while the purely recursive solution would take 7000 seconds to complete a string with 15 characters.

Again showing that the Dynamic programing algorithm has the best performance out of the three.

REFERENCES

Use the Style "referencia" to the references. Example:

[1] Navarro, Gonzalo (2001). "A guided tour to approximate string matching". ACM Computing Surveys.

[2] https://en.wikipedia.org/wiki/Wagner–Fischer_algorithm