**Intro**

The Needleman-Wunsch (String alignment) algorithm is a dynamic programming algorithm. As such, it divides a large problem (e.g., finding similarities) into a series of smaller problems used to find the optimal solution to the larger problem. Common use cases include but are not limited to Genome assembly, Gene Finding, Evolution Theory, and Protein network analysis. The String alignment algorithms can also be used to detect changes between different versions of the same document (as in version control systems) or to detect verbatim copying between different documents. In this app, the algorithm is used to find similarities between two words. Exchange the two words for essays and you have a very efficient and lightweight plagiarism detector.

In this app, the string alignment algorithm takes as input two strings x and y. For a fixed symbol set Σ, it returns a minimal-cost set of edit operations for transforming the string x into string y.

Worst-case performance: O(m\*n)

Worst-case space complexity: O(m\*n)

**How To use the app?**

**Main Screen:**

The main screen is divided into 2 sections, left and right.

The left side is for selecting two words that you want to run the algorithm on. You will see these words in the visualization and the app remembers/keeps the last words you used highlighted. It also displays them in the right section under the app title for your convince.

The right side displays the app title at the top followed by the last two words you used. At the bottom of the screen, the gear takes you to settings in case you want to mess with the algorithm parameters, speed, and visualization settings (highly recommended). When you’re ready to start the visualization, you can tap the start button.

**Settings Screen:**

The settings screen is divided into 2 sections, left and right. Where the right section takes up the most space.

The left side is a legend that you can come back to from the visualization screen. It tells you what each color and square represents in the visualization. Tip: you can tap on a square to change its color.

The right side has 7 text buttons around the middle of the screen.

The speed button lets you control the speed of the visualization. I recommended extra slow for learning and extreme for a good time. Regardless of the speed option, the visualization can be stopped in real-time when in the visualization.

The animation button lets you control how much animation is displayed in the visualization.

As a reminder, the goal of the algorithm is to determine the CHEAPEST operations that TRANSFORM one string into another.

The "delete cost" button lets you decide how expensive it is to delete a letter. In the visualization, this is represented with an animation/operation with the square directly above the “\*current square”.

The "Insert cost" button lets you insert/add a letter. In the visualization, this is an animation/operation with the left square in relation to the “\*current square”. This operation is different from substituting (i.e., swapping a letter.)

The "substitute cost" button lets you decide how expensive it is to substitute a letter. In the visualization, this is an upper diagonal square operation in relation to the “\*current square”.

The "no-operation cost" button is similar to substitute, but no letter is substituted. In the visualization, this is an upper diagonal square operation in relation to the “\*current square”.

\*The current square is a square that’s about to get processed by the algorithm. It’s represented by the “square” color in the legend and has a “processing halo” (also in the legend) to better highlight it.

Minimum letter repeat is how many letters you want to consider “plagiarized”. For example, if you select 3. The algorithm will show where 3+ letters are the same between the two words. If they're not at least 3 letters similar the visualization will not display any results in the final phase of the visualization. I suggest keeping this at 1. Real-world example: how many words did this student essay match IN ORDER from a Wikipedia article.

The 6 icons displayed at the bottom of the right-side section. From left to right. (dismiss this screen, go to help screen (this screen), turn on/off visualization sounds, step through the algorithm (coming soon), switch between dark and light mode, go to the home screen.)

**Visualization screen:**

The app displays the two words you selected on the top and left side of the matrix. It also displays them at the top of the screen.

On the lower-left side of the screen, you will find the operation counter. If you tap it, you can see the number of operations that took place. Note: the color of the button matches the color of the operation.

On the lower right side of the screen. You will see three buttons. Starting from the left. The play button lets you play/pause the animation. The home button takes you home and the gear button takes you to the settings screen.

Visualization settings can be tweaked in the settings screen. The visualization begins with a two-dimensional matrix animation. It's fast so you might miss it ;). This is followed by the string alignment (align string) animation. This is where most of the colors change. This part calculates the costs of each operation. Each square starts as infinity because that’s the worst-case scenario for the square. Slowly the algorithm calculates the cheapest operation available to the square (no operation, insert a letter, delete letter, and substitute letter.) After this animation is finished the (extractalighnment) animation runs. This is a diagonal animation from the lower right of the screen to the top left of the screen. \*These are the optimal operations. After this animation finishes the commonSubstrings animation starts. This animation highlights similar letters.

\*How to check the optimal path: starting from the bottom right square “current square”, Select the smallest number from the following in reference to the current square: top square, left square, top left diagonal square. The cheapest square is now the current square. Keep doing this until you get to the topmost left square. All paths lead to the topmost left square. There could be more than one optimal path. If the path you/the app selected is the smallest (in terms of sum of the values in the squares in the path), then you found the optimal path.

**Algorithm Deep Dive:**

The algorithm is subdivided into 3 parts:

(I) alignStrings() takes as input two ASCII strings x and y, and the cost of the operations, and runs a dynamic programming algorithm to return the cost matrix, which contains the optimal costs for all the subproblems for aligning these two strings.

How was it implemented? alignStrings is in charge of making and populating a two-dimensional array. Therefore, I used a matrix data structure and arrays to solve this entire problem. I used a couple if statements to perform the required operations on the array. the first one places a 0 in the first spot. Two if ‘s are in charge of populating the first x and y columns in the matrix. The remaining two if‘s append the smallest value while taking into consideration if the diagonal is a no-op or a sub. After the entire matrix is populated the matrix is returned.

(ii) extractAlignment(), takes as input an optimal cost matrix, strings x, y, cost of the operations, and returns a vector a that represents an optimal sequence of edit operations to convert x into y. This optimal sequence is recovered by finding a path on the implicit DAG of decisions made by alignStrings to obtain the values starting from Square[0,0].

How was it implemented? extractAlignment is in charge of finding the best path in the matrix. There are a couple of try-except statements to prevent out-of-range errors. In a nutshell, this function consists of 4 checks. One if checks if the top location is the smallest. Another if checks if the left is the smallest. Finally, the remaining two if‘s check if the diagonal is the smallest. There are two types of diagonals a no-op and a sub. This function handles both appropriately. After this is done an array is returned containing the optimal path.

(iii) commonSubstrings() takes as input the ASCII string x, an integer, and an optimal sequence of edits to x, which would transform x into y. This function returns each of the substrings of length at least L in x that aligns exactly, via a run of no-ops, to a substring in y.

How was it implemented? commonSubstrings essentially is a while loop that traverses the entire array passed in from the last function extractAlignment. In this new function, I create two ”pointers”. One pointer is used to track where L no-ops begin and the other is used to note where L no-ops end. After I find a section in the array that is at least L long. The pointers are used as markers to append the section to the result. The ”pointers” are updated accordingly. In this function, no-ops are key because this signifies no change in the text file which is a key aspect of the plagiarism checker.

**Runtime Complexity Justifications:**

Function 1: alignStrings = (startString \* endString)

This is because there are two strings that the algorithm is using to create a two-dimensional matrix. Every value in that two-dimensional matrix needs to be computed therefore its running time is thata (startString \* endString)

Function 2: extractAlignment = (startString + endString)

This function finds the optimal path from the bottom right corner to the top left corner. In terms of traversing the matrix, you can’t go right or down. The worst case is if the algorithm only utilizes left and top but no diagonal. Therefore, the worst case is the length of the startString + the length of endString.

Function 3: commonSubstrings = len(startString + endString)

This function takes the array from extractAlignment as an argument. We know that startString + endString is part of the worst case. We use the length of that array as a condition for a while loop. therefore the worst case is len(startString + endString).