Final Project: Decoding the relationships between Genes

1. Python Implementation that gives all the LCSs for two stings and their corresponding lengths:

You can find the implementation of the code in the Appendix B section. The code returns all of the Longest Common Subsequences (LCSs) for any two given arbitrary strings as well as the LCSs corresponding length. Then, I added several test cases to evaluate the ability of my code with different edge cases like when the string is empty or multiple LCS with the same length or no LCS.

I used the bottom-up dynamic programming approach. It builds a lookup table that is a 2D list that has the lengths of LCSs in substring between the two arbitrary strings (X and Y). Then, the table is filled using a bottom-up approach. It calculates the length of LCS for each pair of substrings of X and Y.

2. Matrix of LCS length:

I created an empty 2D numpy array to store the length of LCSs for each pair of strings. The dimensions are set to be 7 rows x 7 columns since we have 7 strings in Set_Strings. I then iterated through cell in the matrix by iterating through rows and columns, I used the longest_common_subsequence function from question 1 to calculate the length of LCS between the two strings that correspond to that cell. I inserted that LCS length value in that cell. After that, I printed the matrix 2D array in a matrix shape as in figure (1).

LCSs lengths matrix

	a	b	c	d	e	f	g
a	124	90	104	82	93	83	80
b	90	100	91	83	82	88	83
c	104	91	113	81	99	82	80
d	82	83	81	115	80	101	93
e	93	82	99	80	118	80	79
f	83	88	82	101	80	107	96
g	80	83	80	93	79	96	113

Figure(1): Matrix for the length of the longest common sequence between the DNA strings

2.a In the nested loops, the lengths of LCSs are calculated for each pair of strings, and the results are stored in len_lcs_matrix.

2.b We also achieved this criterion as the matrix has [7,7] dimensions and we can use len_lcs_matrix[row, column] to give us the length of the LCS between two given strings in the Set_Strings list.

2.c Yes, we can infer from the matrix how close the strings are to each other. We can look at which pair has the longest LCS in common. This pair is more probably related to each other since they have fewer changes. The longer LCSs, the stronger “closer” the relation between the two strings compared. For example, ‘d’ and ‘f’ have a common length of LCS= 101, and ‘d’ and ‘g’ have LCS= 93. That means that d is more related to f than g. Also, d and f are more related to each other more than any other string d or f. Thus, d is the closest relative to f. We can also notice that the longest LCS happens between each string and itself, which makes common sense because there is no other string that is more related to a string than itself. However, we can’t say anything about the parental relationships between these strings.

3.a Local strategy/ Greedy approach:

We’ll leverage the similarity percentage between pairs of DNA strings as the guiding factor for establishing relationships. This percentage is computed by taking the length of the Longest Common Subsequence (LCS) between two strings and dividing it by the length of the longer

string. This approach is chosen because it accommodates both insertions and deletions, considering the changes in the strings.

Opting for the length of the longest string is crucial since it provides an estimate that encompasses various types of changes. In contrast, using the length of the shortest string wouldn't effectively capture differences due to potential insertions in the longer string. Thus, dividing the LCS by the length of the longest string offers a sensible measure, providing a reliable estimation for the shared percentage.

The greedy algorithm exploits this percentage similarity property for each string in relation to others. It helps identify which two strings act as the children of a particular parent string. The subsequent explanation will delve into this process.
We will use the percentage of similarity between a pair of two DNA strings as the greedy property to infer the parental relationships. Then, I created a matrix as you can see in figure(2) to calculate the percentage by dividing the length of the LCS between the two strings by the length of the longest of the two strings. We choose the length of the longest as both additions and deletions are allowed between the two strings and we want to capture an estimate for all types of changes. If we divide by the length of the shortest string, it won't capture the difference between the two strings due to the additions in the longer one. Dividing the longest common sequence between the two strings by the length of the longest makes sense for a good approximation of the percentage of the similarity between them.

percentage of lcs length relative to the row sequence length

	a	b	c	d	e	f	g
a	1.00	0.73	0.84	0.66	0.75	0.67	0.65
b	0.90	1.00	0.91	0.83	0.82	0.88	0.83
c	0.92	0.81	1.00	0.72	0.88	0.73	0.71
d	0.71	0.72	0.70	1.00	0.70	0.88	0.81
e	0.79	0.69	0.84	0.68	1.00	0.68	0.67
f	0.78	0.82	0.77	0.94	0.75	1.00	0.90
g	0.71	0.73	0.71	0.82	0.70	0.85	1.00

Figure (2) : Matrix for percentage of LCS between each two strings relative to their length.

We will start at the first row in the table corresponding tothe string 'a' and we look for the two strings in the columns that have the highest percentage with 'a'. In that case, c and e with percentages with a are 0.84 and 0.75 respectively, and thus will be the children of a. Since we already assigned a, d, and e to parents, they no longer need adoption and thus we will not consider them again as potential children for other strings. We will go for the row of the first child, c, in the matrix and look for the two strings with the highest percentage with c between the

strings left for adoption (b, d, f, g). Since b and f have the highest percentage with c, they are the children of c. We will then go for the second child of a, which is e, and look at its corresponding row. We have only 2 strings left (d and g) so we will assign them as children of e. They aren't the highest percentage with c but we chose them for e just because they were left as we are following a greedy algorithm. The final tree of string relationships is as in figure (3).

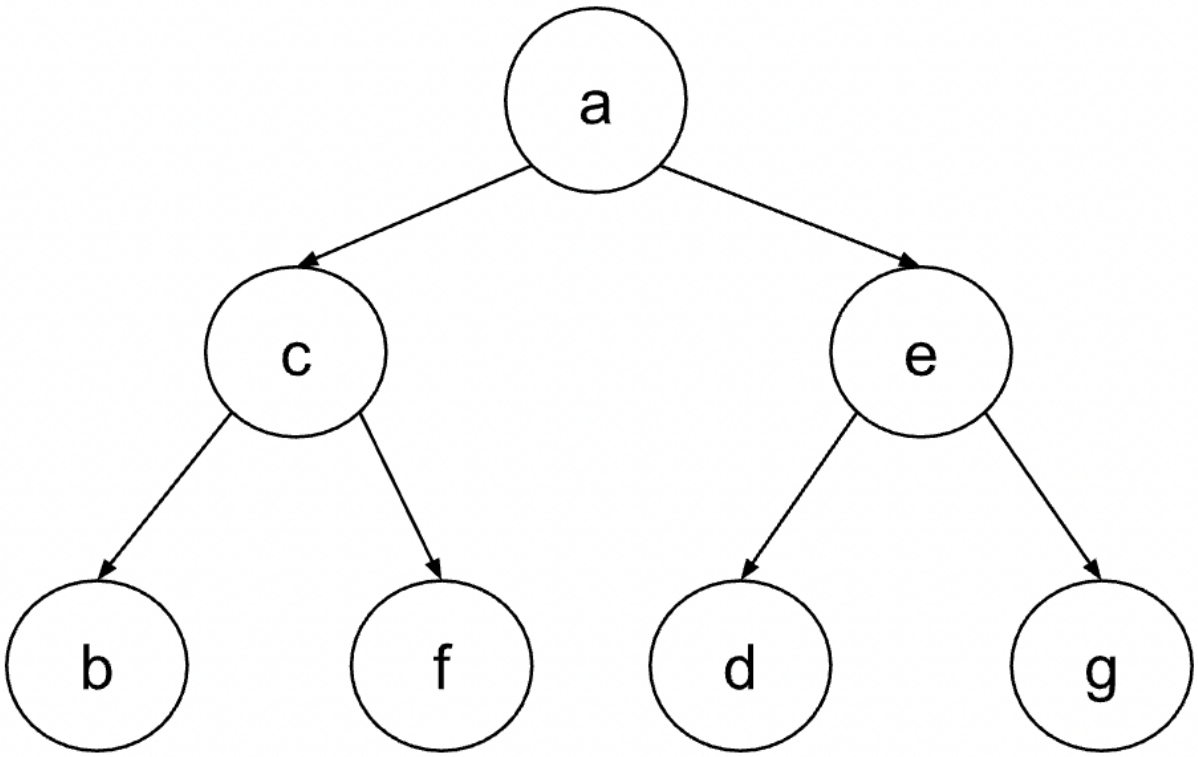

Figure (3) : The tree of the relationships between DNA strings generated by Local or Greedy algorithm.

This is a greedy algorithm because we did not consider all relationships at once in the tree of relationships, but we only compared percentage similarity for a given parent to identify its immediate children. We selected a random great-grandparent and then assigned children for a given parent based on their percentage similarity.

We did not consider other relationships or connections in the tree at the same time. This algorithm is not expected to give us the global optimum all the time especially because we do not know if the great-grandparent is in the correct position as we just start with any random string that is at the top and we connect children to it.

3.b Global or Dynamic Programming Algorithm:

To reach a global solution, we need to identify a metric that defines the relationships between strings more accurately. To reach from a parent to a child, at least a given number of changes, such as mutations, insertions, or deletions for one character in the string, happen to the parent. The goal is to minimize the number of these changes between a given string and its child or parent in the tree of relationships between all strings. Thus, our metric is the number of modifications or the modification distance between the two strings. We want to minimize that modification distance or find the shortest distance or find the shortest path of modifications between two strings. This approach of least modification distance is applied in actual software that builds phylogenetic trees using the genomes of a given number of organisms to show the evolutionary relationship between these organisms. We will implement dynamic programming here as the problem has optimal substructure as well as overlapping subproblems.

The approach uses dynamic programming as it has an *optimal substructure and overlapping subproblems*. It has an optimal substructure because we can solve the main problem of finding the modification distance between each two strings by breaking down that problem to finding the shortest modification distance between the first given number of characters (say x) in the first string and the first character in the second string. We can solve the main problem when we have the optimal solutions for that subproblem. We can notice that it also has overlapping subproblems because we may need to solve this same subproblem multiple times to reach the solution of the main problem which is finding the minimum modification distance between two strings. To take advantage of this repetition or overlapping property, we will use memoization to solve this subproblem once.

To identify the great-grandfather, we will select the string that minimizes the modification distance between it and all other strings considered together at the same time. To solve this, for each string, we will calculate the sum of the distance between that string and all other strings given. After getting all the distances between every two strings, we add all the distances from each string to the others (for example, the distance from a to b + a to c+ a to d… etc.) and save them as the sum of distances from a. Since we do the same for all the strings, we obtain a list of all the sums. Those sums show how connected each string is to all other strings. The grandparent string is the most connected (having the shortest distances) to all other strings. Thus, the string with the lowest distance sum value is the grandparent as you can see in the appendix it is ‘b’. After identifying the grandparent by comparing it to all the values, we identify the parents by finding the most connected strings “shortest distance” to the grandparent by the code in the appendix.

After that, we will identify the two strings that have the shortest modification distance with that great-grandfather. These two strings will be the great-grandfather's children. For each of these two strings, we will repeat the process to identify their two children. We will keep doing so until we run out of strings. We build a tree based on that. The resulting tree is in figure (4) and I did it by tree_maker function that constructs the genealogy tree using the aforementioned strategy to double-check. This is a global algorithm because we considered all possible connections between strings at once to identify their relationships as described earlier.

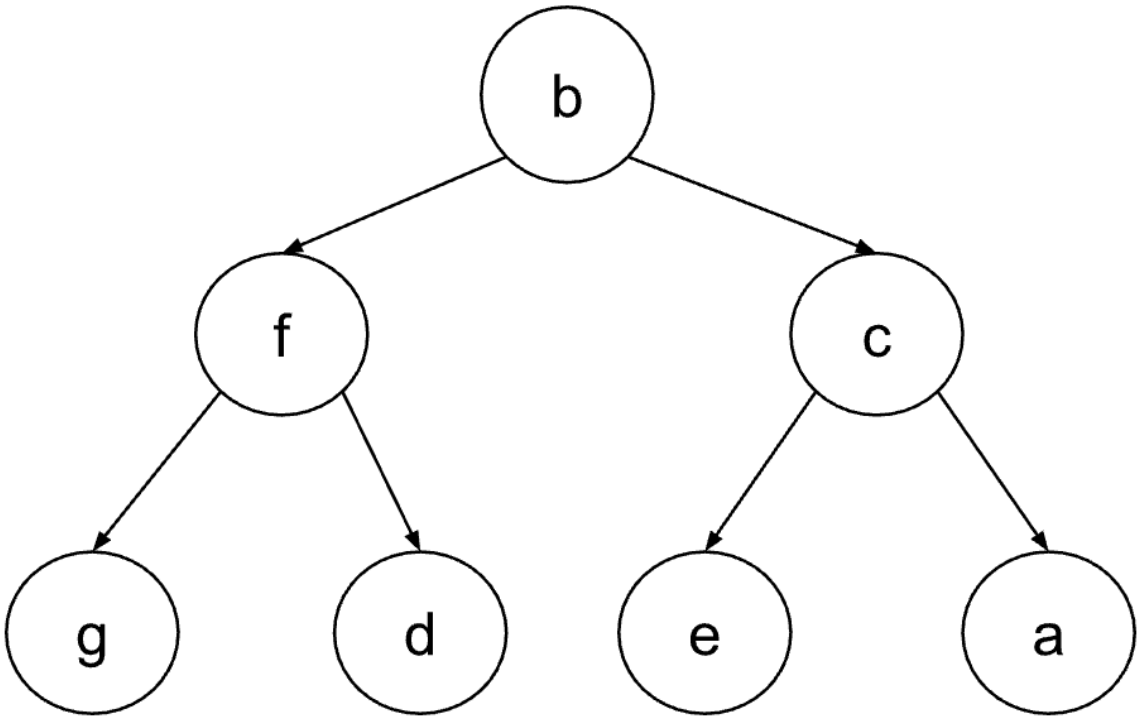

Figure (4) : The tree of the relationships between DNA strings generated by Global or Dynamic Programming algorithm.

3.C Comparison between the two algorithms' results

As you can see in Figures (3) and (4) both of the trees aren't the same. They are totally different from the great-grandfathers to the children. That is expected because the first tree is generated by the greedy approach that focuses on the immediate relationships, connecting each parent with its children based on the percentage similarity without accounting for the possibility that

suboptimal local connections may lead to a more optimal arrangement. We picked a random string to be the great-grandfather which is the first string ‘a’ and the results are highly dependent on the starting point. It doesn’t agree with the great-grandfather in the global strategy in the second figure and led to a totally different tree structure. The local strategy only picks the optimum possible solution at a given connection level, but not necessarily the tree as a whole. On the other hand, the global strategy/ dynamic programming compares all possible strings in the position of the great-grandparent based on the relationships with other strings in the tree to identify the great-grandparent. Also, the global or dynamic programming approach guarantees the global optimum solution at every time no matter how we shuffle the strings or add more. The global dynamic programming approach finds the global strongest relation value to all other strings “lowest sum of distances,” considers it the grandparent, and builds the tree based on the strongest relations to each member of a certain generation. Thus, it guaranteed an overall optimal solution to the problem of reconstructing the genealogy tree. The insights from 2. c are different now since as I mentioned the great-grandfather is different which led to the overall different parent-child relationship. However, it has some similarities in the closeness but not the same parental relationships between the strings since we inferred that ‘b’ and ‘c’ are relatives of each other. In the first tree, ‘b’ is the parent of ‘c’ but in the second tree, ‘c’ is the parent of ‘b’. The same case for ‘a’ and ‘c’. Although in the first tree ‘d’ and ‘g’ were both children to ‘e’, they are still both children but to ‘f’ in the second tree.

4. Complexity Analysis:

Greedy Algorithm:

In this algorithm, we build a matrix for the length of the longest common subsequence (LCS) between each two strings. We then use this matrix to generate a percentage matrix that we use as an estimation for the similarity between the two strings. We then use this as a greedy property of each string to assign relationships. Thus, the main part of creating this algorithm to create the relationships tree is generating the matrix, and the complexity of building that matrix is the same as the algorithm.

As mentioned, we will assume we have N number of strings and M characters in each string. The code that generates the matrix has two main loops. The outer loop iterates over all possible pairs of strings, which is equivalent to N^2, the square of the number of strings. The inside loop iterates over the characters of the two strings in a nested loop form. Since each string has M characters, the inside loop has a time complexity of M^2. Thus, we can conclude that our

algorithm has a time complexity of O(N^2× M^2). The big O notation means the worst-case scenario of the algorithm and we are interested in it as it represents most of the cases and we are interested in knowing how our algorithm will behave in those cases.
In case we keep the length of the strings (M) constant and only change the number of strings, we do not have to worry about the complexity component with respect to M as it will be a constant value. Thus, in that case, the time complexity of our greedy algorithm is O(N^2).
To test that experimentally, we designed an experiment where we changed the number of strings in the relationship tree. We calculated the average run time as we ran every instance (number of strings) of the experiment multiple times to take the average. We produced a graph for the results. This is present in the corresponding part in the appendix. If we look at our experimental results (fig. 5), we can notice that the scaling of the graph as the input size increases is quadratic which means that when the input size increases by 2, the running time increases by (2^2=4). Since the time complexity basically represents the time growth scaling of the algorithm as the input size increases, we can notice that the experimental results indeed agree with our analytical results.

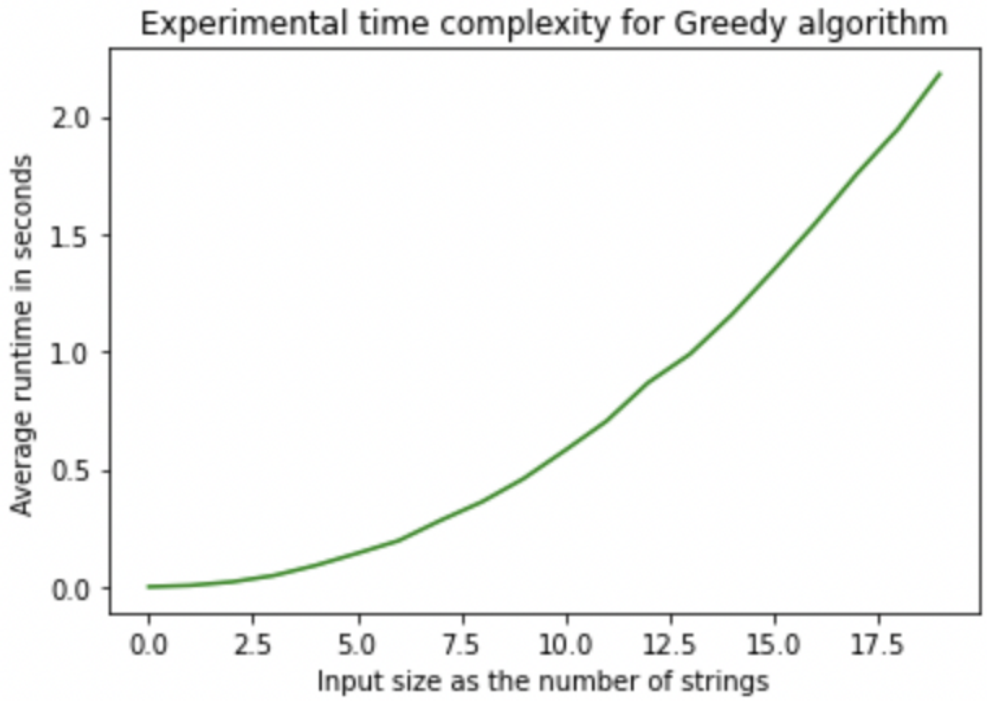

Figure (5) : The experimental runtime growth for the greedy algorithm with input size varied with number of strings. The runtime scaling is quadratic with input size.

Dynamic programming

The main algorithm of the dynamic programming approach depends on a function named `edit_distance_dynamic_programming`, which calculates the least modification distance between each pair of strings. This function implements a dynamic programming approach, creating a two-dimensional array to record the solutions of the subproblems. The size of that array depends on the number of characters in the strings. Since the number of characters is denoted as M, the size of the array is M^2. This function is then called by the `distances` function for every possible pair of strings in the list. As we have N number of strings, this function is called N^2 times. Thus, the overall time complexity of that algorithm is O(N^2 × M^2).

Similarly, as mentioned earlier, in case we keep the length of the strings (M) constant and only vary the number of strings, we do not have to worry about the complexity component with respect to M as it will be a constant value. Thus, in that case, the time complexity of our greedy algorithm is O(N^2).

As depicted in the corresponding part in the Appendix, we tested this algorithm experimentally by changing the number of strings (N) and representing the input size. We then calculated the time that the algorithm takes to run. We repeated the time calculation step several times per input size so that we took the average runtime per input size. We see the graph in figure (6). We can notice that the graph scales quadratically as the input size increases. This agrees with our time complexity value calculated analytically.

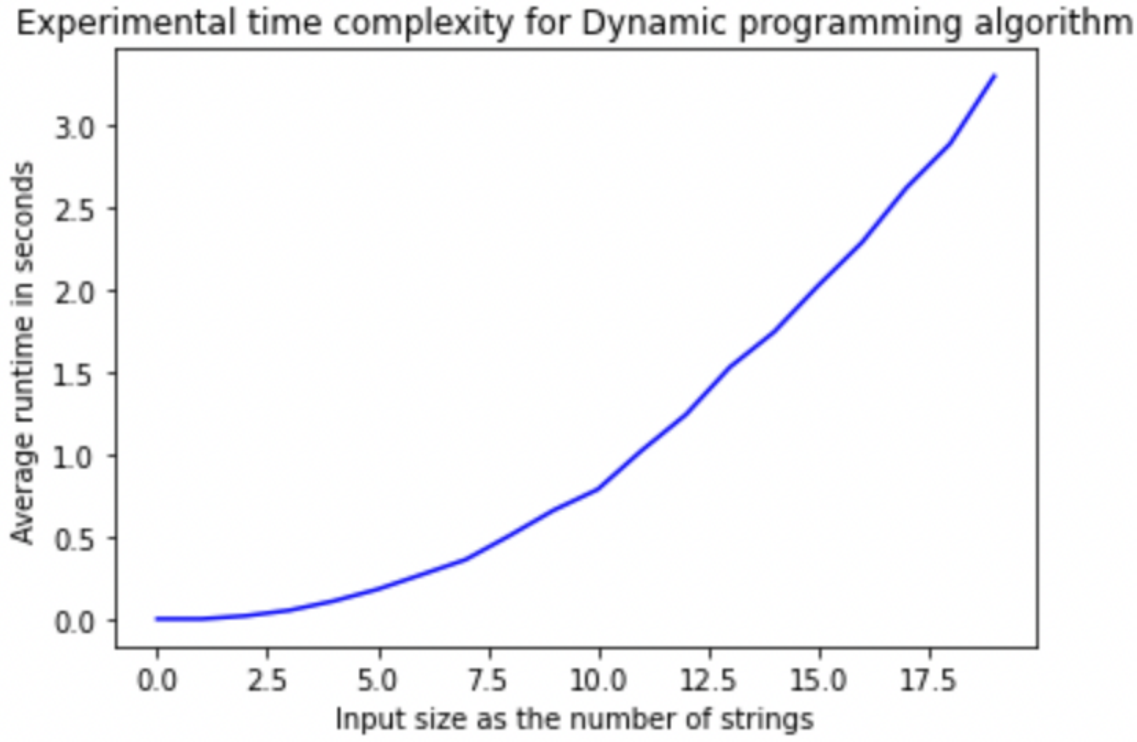

Figure (6) : The experimental runtime growth for the dynamic programming algorithm with input

size varied with number of strings. The runtime scaling is quadratic with input size.

Comparing greedy and dynamic programming approaches

From the analytical approach, we can notice that the time complexity for both algorithms is the same O(N2 × M2) or O(N2) when only considering the number of strings and making the number of characters per string constant. That means that both algorithms have a runtime that scales in the same manner as the input size increases.
We can also confirm that from the experimental approaches, we notice that they both scaled quadratically as we increased the number of strings as the input size as we see in figure (7). Also, the greedy algorithm approach looks a little better than the dynamic in the figure since it doesn't need memoization that could affect the time complexity or check subproblems that cause overhead but we know that even if it has better time complexity, we can't depend on the greedy since as we showed above how it doesn't give optimal solution and just check immediate aspects without looking for other strings so it isn't guaranteed to give the global optimal solution. Also, this problem has overlapping subproblems and optimal substructure so it is better to go with dynamic programming, and greedy would be better for simpler problems.

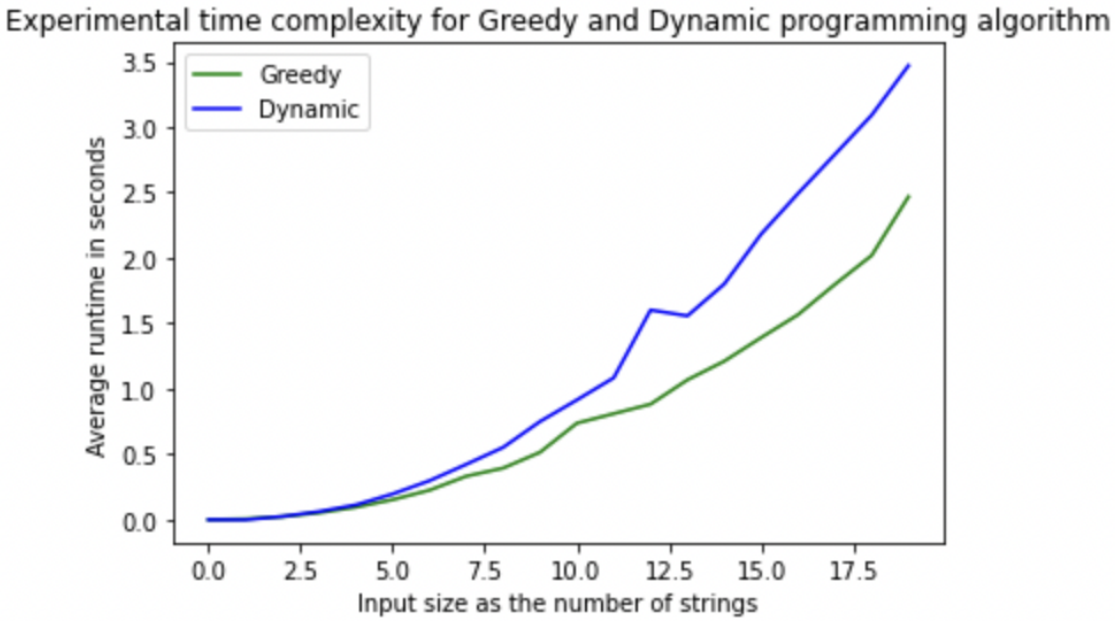

Figure (7) : Comparison between experimental runtime growth for the greedy and dynamic programming algorithm with input size represented in number of strings. The runtime for both algorithms scale quadratically as the input size increases.

5. Estimating the probabilities of insertions, deletions, and mutations:

To estimate the probabilities of a certain type of modifications in DNA, we can use our DNA strings and the parent-child relationships in the tree generated by our algorithm. We will count the number of each type of modification between a given parent and its child. After that, we will take the average of the number of that given type of modification between all parent-child connections in the tree.

To do so, we will start by aligning each parent and child sequence. The alignment will be that all the sequences that agree will be in positions corresponding to each other.
If there is a missing character in one of them, it will be represented at an empty slot or a dash (-) at that position.
If there is a change in the character, there will be a mismatch at that position in the alignment. To do this alignment step, we will utilize a very famous simple alignment algorithm called the **Needleman-Wunsch algorithm**. We implemented the algorithm strategy and the Python code from Slowikowski, K. (2020).

Basically, the Needleman-Wunsch algorithm uses a dynamic programming approach to align genetic code or protein sequence and is used widely in genomics and proteomics computational applications for its relative simplicity and efficiency in comparing biological sequences. The algorithm divides the large problem, which is the full sequence into smaller subproblems and solves these small problems to find the large problem's optimal solution. It starts by building a matrix between the characters of the two strings. It moves through the cells row by row and assigns a score to the degree of alignment between the characters based on comparing the scores of the ones around in the direction left, top, or top left, and adding appropriate scores for match, and mismatch (modification) or deletion/insertion. After filling up the matrix, the algorithm finds the optimal path from the bottom right to the top left that will have the highest possible score. More importantly, the output of the algorithm is two strings, where the gaps in one of the sequences (due to an insertion in the other sequence or a deletion in that sequence) are represented as a dash. A mutation is represented by just a mismatch at that particular site in the alignment.

For every parent-child string pair, we will align them using the algorithmic strategy described above. After that, we want to count the number of modifications that happened between the parent and the child. As we explained earlier, the parent string is the original one that produced

the child and, thus, we will take it as a reference for identifying the type of modifications. Based on that, we will have three types of modifications:

- Mutations: it will be represented as a mismatch between the aligned parent and child strings
- Insertions: it is when we find a gap or a (-) in the parent string but a character in the child string
- Deletions: it is when we find a character in the parent string but a gap or a (-) in the child string.

We will divide the number of each type of modification to determine the probability of a given type of modification happening per a character in that parent string. After we find the results for the probability of each type of modification (mutations, insertions, and deletions) between each pair of parent-child strings, we will calculate the average by summing them up and dividing by the number of parents-children pairs we have. That way, we will get an estimated probability for the types of modifications for a given string. You can see all of these calculations in the code implementation in the appendix B

b. Python implementation and estimation results reflection

We implemented this algorithmic strategy to estimate the probability of the three different types of modifications (mutations, insertions, and deletions) using python as in the corresponding part of the code in Appendix B. We believe this estimation approach best utilize the data given to make a strong and plausible estimation. We got these estimated probability values:

- Deletions probability is 0.0358
- Insertions probability is 0.1146
- Mutations probability is 0.0596

The relatively low probability of deletions suggests that, on average, there's a small chance of a character being deleted when transitioning from parent to child. This aligns with the biological intuition that deletions might be less frequent events.

A higher probability of insertions indicates that, on average, there's a more significant chance of introducing new characters in the child sequence compared to deletions. This could be attributed to the dynamic nature of DNA, where insertions might be more common in certain contexts.

The probability of mutations falls between deletions and insertions, suggesting that, on average, there's a moderate chance of character changes occurring between parent and child sequences. This is in line with the understanding that mutations are fundamental to genetic diversity.

We can notice that these values seem plausible as they are low, which corresponds to our knowledge about mutations or changes in genetic codes that are quite rare events.The probabilities seem reasonable from a biological standpoint, but it's important to note that these estimates are dependent on the dataset and the specific characteristics of the DNA sequences we are working with. Real-world biological data can be complex, and these estimates might vary across different scenarios.

Appendix A:

LOs Applications:

#ComputationalCritique:
I explicitly critiqued the local and global approaches and how they ended up with an optimal solution. I set the limitations for the greedy algorithms that resulted in a different tree structure than the dynamic programming. I explained the effectiveness of the implementations of algorithms throughout the whole project. I explained why the results made sense and whether they met the expectations. I discussed how dynamic programming in this project provides an optimal solution.

#AlgoStratDataStruct:
I justified the choice of the greedy for the local approach and dynamic programming for the global approach. I explained their strategies for finding the parental relationship between the strings. I also explained the algorithm strategy to estimate the probability of the changes. I explained how each algorithm works and how they are guaranteed to provide the expected answer.

#PythonProgramming:
I used the feedback from the previous assignments to run the code more than once before the submission, as there was a bug every time after the submission. I made sure to add many test cases to evaluate the code broadly. I printed the output of the function that isn't in the same

order as the assertion statement in the prompt to show how the algorithm works, but the order is not the same.

#CodeReadability:
I ensured all the functions had docstring and the comments were short enough. I named the functions properly to make them clearer for the reader. I set the variables to reasonable names to avoid confusion from the reader. Also, I ensured that the code line didn't exceed the margin of the page. In the pdf version, I made sure there is no statements cut-off in the code as in the previous assignment.

#ComplexityAnalysis:
I set the hypothetical and experimental time complexity for both strategies, Greedy and dynamic programming, and showed how the results align as expected. I analyzed the functions that dominate in both strategies that give rise to the overall time complexity. I set the formula to show how it ended up by the quadratic time complexity.

#professionalism:
I put time and effort into presenting the data in a structured and organized report. I reviewed the context in Grammarly to avoid any typos. I cited the external algorithm to estimate the insertion, deletion, and mutation probability.

word count: 300 words.

HCs:

#dataviz:
I added clear graphs and a description to each one. I added title, x_axis, and y_axis label for the graphs. I explained the graphs and mentioned the figures in the relevant context.

#estimation:
I produced an estimation for the probabilities of the modification (insertions, deletions, and mutations). I explained the calculation strategy well and how it is used in this context. I explained the approach step by step and implemented it in the code to find the final probabilities. I interpreted the values of the estimations in the context.

#audience:
The audience here is the professor, who I thought as one of my peers. I ensured that I explained

everything and the language was clear for them. When I used an external strategy (Needleman-Wunsch algorithm), I explained the full approach since I knew that we hadn't dealt with this algorithm before.

word count: 140 words.

References

Slowikowski, K. (2020). A simple version of the Needleman-Wunsch algorithm in Python [Source Code].
https://gist.github.com/slowkow/06c6dba9180d013dfd82bec217d22eb5

AI statement:

I used Grammarly to evaluate the grammer of the texts. I didn't use other types of AI.

Appendix B:

```
#DNA strings given in the problem
a='''GCCTCCGTTCATGACGTGTGTATTTTATTCCGAGCAGGATTCAATCGGACATCCAG\
TTCTGCTACATTCCTAGCTAATGAAGAAACTAGACAGCGTCATAGTCTCTATTCTCATAGTGAATAAC'''



b='GACCTCGTCAGCTTCAGTTTATCCAGCAGAATTCAGATGTCATAGTT\
CGTATCATTCCTGCAAAGAGTACTAGAAGCGTCATAGTCTTTTCTAATAGTAC'
c='GTCCCTCGTCAAGACGTTTCTATTTTATTCCAGCAGGATTCAATCGGCATCA\
GTTCTGTACATTCCTGCAAAGAAGTACTAGACAGCGTCATAGTCTCTATCTAACTAATTAA'

d='ACCTCTCACTAAGTTTCATCAGGACGAGAGAATAAAGACTTCACGTTTCAGTAGCACT\
TCCTGGCCCACACGAGGTACCTAGCAAGCGGTATATAGTCTTTTTTTAGATAGGGAT'

e='GTCCTCTGTCAAAGATGTATTACTGTTTTGCACAGGAATTCAACGGGCATTCAGTTTT\
GTACATTACTCGCAAAGACAGTTACTAGACCAACGTCATAAGTCTCTACAAACTAATTAA'

f='ACCTCTCACTGCAGTTTATCAGGACGAGAGAATAAGATGTCATGTTTCAGTATCATTC\
CTGCCACACGAGTACTAGAAGCGGTATATAGTCTTTTTCTAGATAGGAT'
```

```
g='ACGTCATCACCTCCAGATTTATCTAGGCACGCGAGAATAAGATGTACATGATTTACAGTA\
ACATTCCTGCCACACAGTTAGAAGTGATATAGTCTGTCTTCTTAGATCAGGAT'


Set_Strings= [a,b,c,d,e,f,g]

def longest_common_subsequence(X, Y):
    """
 Find the Longest Common Subsequence (LCS) between two strings and return
 all unique LCSs along with their length.

    Parameters:
    X (str): The first input string
    Y (str): The second input string

    Returns:
    tuple: A tuple containing a list of all unique LCSs and the length of the
    first LCS in the list.
    """

    # Lookup table used to store the lengths of LCS between substrings of X (X[0...
    #and Y (Y[0...j-1])
    lookup_table = [[0 for x in range(len(Y) + 1)] for y in range(len(X) + 1)]

    # Use this function (LCS_table) to fill out the lookup table:
    LCS_table(X, Y, lookup_table)

    # Use this function to find all the longest common subsequences (LCSs)
    lcs = LCSs(X, Y, len(X), len(Y), lookup_table)  # Avoid duplicates
    lcs_set = set(lcs)
    lcs_list = list(lcs_set)
    all_lcs = (lcs_list, len(lcs_list[0]))

    return all_lcs

def LCS_table(X, Y, lookup_table):
    """
    Use a bottom-up dynamic programming approach to calculate the LCS between
    two strings and fill out the lookup table.

    Parameters:
```

```
        X (str): The first input string
        Y (str): The second input string
        lookup_table (list): A 2D list used to store the intermediate results
        of calculating LCS

        Returns:
        list: The filled lookup table.
        """

        # Using a bottom-up approach, fill out the lookup table
        for i in range(1, len(X) + 1):
            for j in range(1, len(Y) + 1):
                # In case the current elements of X and Y are the same
                if X[i - 1] == Y[j - 1]:
                    lookup_table[i][j] = lookup_table[i - 1][j - 1] + 1
                # In case they are different:
                else:
                    lookup_table[i][j] = max(lookup_table[i - 1][j], lookup_table[i][j

        return lookup_table



    def LCSs(X, Y, m, n, lookup_table):
        """
        This function uses the lookup table to determine and return
        all the possible longest common subsequences between X and Y.
        It implements a dynamic programming approach as it utilizes the lookup
        table we built to store the lengths of the longest common subsequences
        between the substrings of X and Y.
        Inputs
        ----------
        X, Y: strings
            Strings to compute the LCS
        m: int
            length of string X
        n: int
            length of string Y
        lookup_table: 2d list
            2D list containing the lookup table constructed
        Returns
        ----------
        list
```

```
            list of all possible LCSs between X and Y
        """
        # if we have no charachters in one of the strings left, return list of empty st
        if m == 0 or n == 0:
            return ['']
        # if the last charachters in both strings match
        if X[m-1] == Y[n-1]:
    # we chop the last charachters in X and Y and find all LCS of the substrings # whic
    #are X[0...m-2] and y[0...n-2] now, and add to the list
            lcs = LCSs(X, Y, m - 1, n - 1, lookup_table)
            # we will also append that last charachter to all LCS of these substrings
            for i in range(len(lcs)):
                lcs[i] = lcs[i] + (X[m-1])
            return lcs
        # else, When the last charachters do not match, proceed with the following
        else:
    # we look at the table, if the left cell of the current one has a higher value
    #that the top cell
    # we disregard the current character in the Y string
    # and proceed with finding LCSs in the substrings X[0...m-1] and y[0...n-2].
            if lookup_table[m][n - 1] > lookup_table[m - 1][n]:
                return LCSs(X, Y, m, n - 1, lookup_table)
    # we look at the table, if the opposite is true, or if the top cell of the current
    # has a higher value that the left cell, we disregard the current charachter of
    #the X string and we proceed with finding LCSs in the substring X[0...m-2] and y[0.
            if lookup_table[m - 1][n] > lookup_table[m][n - 1]:
                return LCSs(X, Y, m - 1, n, lookup_table)
            # in case the top and the left cells have equal values, we consider both:
            else:
                top_cell = LCSs(X, Y, m - 1, n, lookup_table)
                left_cell = LCSs(X, Y, m, n - 1, lookup_table)
                # return them together merged
                return top_cell + left_cell


    longest_common_subsequence(b, a)



    ## test cases:
    # two empty
    x1 , y1 = '', ''
    assert(longest_common_subsequence(x1, y1) == (['', 0))
    # one empty
```

```
x2 , y2 = '', 'a'
assert(longest_common_subsequence(x2, y2) == ([''], 0))
# two one charachter and exactly the same
x3 , y3 = 'a', 'a'
assert(longest_common_subsequence(x3, y3) == (['a'], 1))
# two have many charachters and exactly the same
x4 , y4 = 'abcdefglmnobqrsxyz', 'abcdefglmnobqrsxyz'
assert(longest_common_subsequence(x4, y4) == (['abcdefglmnobqrsxyz'], 18))
# two have many charachters that are the same with duplicates
x5 , y5 = 'abacbdcedfegflgmlnmonpoqprqsrxsyxzyz', 'abacbdcedfegflgmlnmonpoqprqsrxsy
assert(longest_common_subsequence(x5, y5) == (['abacbdcedfegflgmlnmonpoqprqsrxsyxzy
                                               36))
# two have many charachters with nothing in common not the same length
x6 , y6 = 'abcde', 'xyz'
assert(longest_common_subsequence(x6, y6) == (['abcde'], 0))
# two have charachters in common but they do not have the same length
x7 , y7 = 'abcde', 'ace'
assert(longest_common_subsequence(x7, y7) == (['ace'], 3))
# there are multiple LCS with the same length
#this sentence is going to give assertion error since the order of the output
#isn't the same as you can see
#in the second code cell but it gives the same result with different order.
x8, y8 = 'abcbdabacddcab', 'bdcaba'
assert(longest_common_subsequence(x8, y8) == (['bcaba', 'bdcab', 'bdaba'], 5))


# Test cases
x1, y1 = 'ABCBDAB', 'BDCABA'
x2, y2 = 'abc', ''
x3, y3 = 'abc', 'a'
x4, y4 = 'abc', 'ac'

# Test case 1
result1 = longest_common_subsequence(x1, y1)
print(f"Test Case 1: x={x1}, y={y1}")
print(f"Expected Result: (['BDAB', 'BCBA', 'BCAB'], 4)")
print(f"Actual Result: {result1}")
print()

# Test case 2
result2 = longest_common_subsequence(x2, y2)
print(f"Test Case 2: x={x2}, y={y2}")
print(f"Expected Result: ([''], 0)")
```

```
print(f"Actual Result: {result2}")
print()

# Test case 3
result3 = longest_common_subsequence(x3, y3)
print(f"Test Case 3: x={x3}, y={y3}")
print(f"Expected Result: (['a'], 1)")
print(f"Actual Result: {result3}")
print()

# Test case 4
result4 = longest_common_subsequence(x4, y4)
print(f"Test Case 4: x={x4}, y={y4}")
print(f"Expected Result: (['ac'], 2)")
print(f"Actual Result: {result4}")



import pandas as pd
import numpy as np

def build_matrix(Set_Strings, len_lcs_matrix):
    """
    Build a matrix of the lengths of Longest Common Subsequences (LCS) for each pai
    of strings in Set_Strings.

    Parameters:
    Set_Strings (list): A list containing strings to be compared.
    len_lcs_matrix (numpy.ndarray): An empty matrix to store the lengths of the
    LCSs for each pair of strings.

    Returns:
    None: The function prints the resulting LCSs lengths matrix.
    """

    # Iterate over each pair of strings in Set_Strings
    for row in range(7):
        for column in range(7):
            # Get the ith and jth strings from Set_Strings
            string1 = Set_Strings[row]
            string2 = Set_Strings[column]

            # Initialize the lengths of the strings
```

```
            n = len(string1)
            m = len(string2)

            # Create a two-dimensional array of size (n+1) x (m+1) to store
            #the lengths of the LCSs
            lcs_matrix = np.zeros((n+1, m+1))

            # Iterate over the characters in string1 and string2
            for x in range(1, n+1):
                for y in range(1, m+1):
                    # If the characters match, add 1 to the length of the LCS
                    if string1[x-1] == string2[y-1]:
                        lcs_matrix[x, y] = lcs_matrix[x-1, y-1] + 1
                    # If the characters don't match, take the maximum of
                    #the previous values
                    else:
                        lcs_matrix[x, y] = max(lcs_matrix[x-1, y], lcs_matrix[x, y-

            # Store the length of the LCS for this pair of strings in len_lcs_matri
            len_lcs_matrix[row, column] = lcs_matrix[n, m]

    # Identifying columns and rows for formatting
    column_names = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
    row_names = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
    matrix = pd.DataFrame(len_lcs_matrix, columns=column_names, index=row_names)

    print("LCSs lengths matrix ")
    print(matrix.astype(int))


# Set_Strings contains the strings to be compared
Set_Strings = [a, b, c, d, e, f, g]

# len_lcs_matrix will store the lengths of the LCSs for each pair of strings
len_lcs_matrix = np.zeros((7, 7))

build_matrix(Set_Strings, len_lcs_matrix)



#percentage of lcs length relative to the row sequence length

lcs_percentages = []
```

```
sequences_lengths = [len(n) for n in Set_Strings]


#looping through the LCSs
for i in range(0,7):
    fraction_row = []
    for j in range(0,7):
        #getting the fraction of LCS to the length of the row string
        fraction_row.append(round(len_lcs_matrix[i,j]/sequences_lengths[i], 2))
    lcs_percentages.append(fraction_row)
len_lcs_matrix_fract = np.array(lcs_percentages)

#identifying columns and rows for formatting
column_names = ['a', 'b', 'c','d', 'e', 'f', 'g']
row_names    = ['a', 'b', 'c','d', 'e', 'f', 'g']
fract_matrix = pd.DataFrame(len_lcs_matrix_fract, columns=column_names, index=row_n
print("percentage of lcs length relative to the row sequence length")
print(fract_matrix)


#finding the least modifications distance between a given two strings using dynamic
# show the great-grandfather based on the calculations
def edit_distance_dynamic_programming(str1, str2, m, n):
    """
    Calculate the edit distance between two given strings using dynamic programming
    The edit distance is the minimum number of operations
    (insertions, deletions, and mutations)
    required to transform one string into the other.

    Parameters:
    str1 (str): The first input string
    str2 (str): The second input string
    m (int): The length of str1
    n (int): The length of str2

    Returns:
    int: The minimum number of operations required to transform str1 into str2.
    """
    dp = [[0 for x in range(n + 1)] for x in range(m + 1)]

    for i in range(m + 1):
        for j in range(n + 1):
```

```
                if i == 0:
                    dp[i][j] = j  # Min. operations = j
                elif j == 0:
                    dp[i][j] = i  # Min. operations = i
                elif str1[i-1] == str2[j-1]:
                    dp[i][j] = dp[i-1][j-1]
                else:
                    dp[i][j] = 1 + min(dp[i][j-1], dp[i-1][j], dp[i-1][j-1])

        return dp[m][n]


    def retrieve_name(x, Vars=vars()):
        """
        Get the name of a variable.

        Inputs:
        x: The variable to find the name for

        Returns:
        str: The name of the given variable
        """
        for k in Vars:
            if isinstance(x, type(Vars[k])):
                if x is Vars[k]:
                    return k


    def distances(strings):
        """
        Get the distances between each two strings.

        Inputs:
        strings: A list of strings

        Returns:
        list: A list of distances between each two pair of strings
        """
        vals = []
        for i in strings:
            for j in strings:
                if [i][0] == [j][0]:
                    pass
```

```
            else:
                vals.append([retrieve_name(i), retrieve_name(j), \
                             edit_distance_dynamic_programming\
                             (i, j, len(i), len(j))])
    return vals


def all_sums(strings):
    """
    Calculate the sum of all distances between each string and others.

    Inputs:
    strings: A list of strings

    Returns:
    list: A list of the sum of distances between each two pair of strings
    """
    a_sum, b_sum, c_sum, d_sum, e_sum, f_sum, g_sum = 0, 0, 0, 0, 0, 0, 0
    all_sums_vals = distances(strings)

    for i in all_sums_vals:
        if i[0] == 'a':
            a_sum += i[2]
        elif i[0] == 'b':
            b_sum += i[2]
        elif i[0] == 'c':
            c_sum += i[2]
        elif i[0] == 'd':
            d_sum += i[2]
        elif i[0] == 'e':
            e_sum += i[2]
        elif i[0] == 'f':
            f_sum += i[2]
        else:
            g_sum += i[2]

    all_sums = [["a", int(a_sum)], ["b", int(b_sum)], ["c", int(c_sum)], \
                ["d", int(d_sum)],
                ["e", int(e_sum)], ["f", int(f_sum)], ["g", int(g_sum)]]

    return all_sums
```

```
def grandparent_finder(strings):
    """
    Find the grandparent node value from the all sum.

    Inputs:
    strings: A list of strings

    Returns:
    int: The value of the grandparent node
    """
    a_sum, b_sum, c_sum, d_sum, e_sum, f_sum, g_sum = 0, 0, 0, 0, 0, 0, 0
    all_sums_vals = distances(strings)

    for i in all_sums_vals:
        if i[0] == 'a':
            a_sum += i[2]
        elif i[0] == 'b':
            b_sum += i[2]
        elif i[0] == 'c':
            c_sum += i[2]
        elif i[0] == 'd':
            d_sum += i[2]
        elif i[0] == 'e':
            e_sum += i[2]
        elif i[0] == 'f':
            f_sum += i[2]
        else:
            g_sum += i[2]

    sum_values = [int(a_sum), int(b_sum), int(c_sum), int(d_sum), \
                  int(e_sum), int(f_sum), int(g_sum)]
    grandparent_value = int(min(sum_values))

    return grandparent_value


# Driver code
strings = [a, b, c, d, e, f, g]

print(all_sums(strings))
print('grandparent=', grandparent_finder(strings))
```

```
distances(strings)
```

```
#this cell shows the tree based on the dynamic programming strategy
def tree_maker(all_of_sums, grandpaernt, all_distances):

    """

    A function that takes all sums list, grandparent value, all the distances
    from one string to others and returns the genaology tree.


    Parameters:
    all_of_sums (arr): array of sum of all distances from a certain string
    grandpaernt (int): grandparent distance value
    all_distances (arr): array of all distances from each string to others

    Returns:
    3 lists : each list holds generations by level

    """

    #filling first level with grandparent
    first_level=[]
    for i in all_of_sums:
        if grandpaernt == i[1]:
            first_level.append(i[0])
    print(first_level)


    #filling second level with parents that have strongest connections to grandpare
    second_level=[]
    for j in all_distances:
        if j[0] == first_level[0] and j[2] < 30:
            second_level.append(j[1])
    print(second_level)


    #filling third level with childer that have strongest connections to parents
    third_level=[]
    for k in all_distances:
        if second_level[0] == k[0]:
```

```
            #making sure it doesn't reuse parents if they have strong connection
                if k[1] == second_level[1] or k[1] == first_level[0]:
                    pass
                elif k[0] == second_level[0] and k[2] < 30:
                    third_level.append(k[1])

            if second_level[1] == k[0]:
                #making sure it doesn't reuse parents if they have strong connection
                if k[1] == second_level[0] or k[1] == third_level[0] or\
                k[1] == third_level[1] or \
                k[1] == first_level[0]:
                    pass
                elif k[0] == second_level[1] and k[2] < 30:
                    third_level.append(k[1])


        print(third_level)



    all_of_sums = all_sums(strings)
    grandparent = grandparent_finder(strings)
    all_distances = distances(strings)


    tree_maker(all_of_sums, grandparent, all_distances)


    import random

    def strings_maker(n):
        '''
        This function makes a random n number of strings. We will fix them at size 100.

        Inputs
        -------
        n: int
            Number of strings to be generated

        Returns
        -------
        Set_Strings: lst
            List of n random strings
```

```
        '''
        Random_Strings_Set = []
        for i in range(n):
            string = ""
            chars = ["A", "C", "G", "T"]
            for j in range(100):
                string += random.choice(chars)
            Random_Strings_Set.append(string)
        return Random_Strings_Set

    strings_maker(4)



    import numpy as np

    def length_matrix_experimental(lenlcs_matrix, Set_Strings, n):
        '''
        This function generates a matrix for the LCS lengths between
        each pair of a number of strings.
        This version is modified for experiments.

        Inputs
        ----------
        Set_Strings: lst
            A list for the strings to generate the matrix for
        lenlcs_matrix: 2D array
            A 2D array that holds the matrix structure with 7*7 cells

        Returns
        ----------
        matrix: 2D array
            The function prints the filled matrix in a matrix shape
        '''
        for i in range(n):
            for j in range(n):
                row_string = Set_Strings[i]
                column_string = Set_Strings[j]
                temp_matrix = np.zeros((len(row_string) + 1, \
                                        len(column_string) + 1))
                for x in range(1, len(row_string) + 1):
                    for y in range(1, len(column_string) + 1):
                        if row_string[x - 1] == column_string[y - 1]:
```

```
                    temp_matrix[x, y] = temp_matrix[x - 1, y - 1] + 1
                else:
                    temp_matrix[x, y] = max(temp_matrix[x - 1, y],\
                                            temp_matrix[x, y - 1])
            lenlcs_matrix[i, j] = temp_matrix[len(row_string), len(column_string)]



#greedy
#the plot code cells take some time to run
import time
import matplotlib.pyplot as plt

def experimental_greedy(n):
    '''
    This function performs an experimental average runtime calculation
    for the greedy algorithm
    as we vary the input size which is the number of strings.

    Inputs
    --------
    n: int
        The maximum number of strings to test the algorithm for

    Returns
    ---------
    None
        This function produces a graph for the average runtime per input size
    '''
    # Random_Strings_Set = strings_maker(n)
    average_runtimes = []

    for i in range(n):
        # generate input size of length i
        Random_Strings_Set = strings_maker(i)
        runtime = 0

        # repeat each experiment 10 times
        for t in range(10):
            greedy_start_time = time.time()
            lenlcs_matrix = np.zeros((len(Random_Strings_Set),\
                                      len(Random_Strings_Set)))
            length_matrix_experimental(lenlcs_matrix, \
```

```
                                        Random_Strings_Set, i)
            greedy_end_time = time.time()
            runtime += (greedy_end_time - greedy_start_time)

        average_runtimes.append(runtime / 10)

    plt.plot(range(0, n, 1), average_runtimes, color='green')
    plt.title('Experimental time complexity for Greedy algorithm')
    plt.xlabel('Input size as the number of strings')
    plt.ylabel('Average runtime in seconds')
    plt.show()

experimental_greedy(20)



#dynamic
def experimental_dynamic(n):
    '''
    This function performs an experimental average runtime calculation for the
    Dynamic Programming algorithm
    as we vary the input size which is the number of strings.

    Inputs
    --------
    n: int
        The maximum number of strings to test the algorithm for

    Returns
    ---------
    None
        This function produces a graph for the average runtime per input size
    '''
    average_runtimes = []

    for i in range(n):
        # generate input size of length i
        Random_Strings_Set = strings_maker(i)
        runtime = 0

        # repeat each experiment 10 times
        for t in range(10):
            dynamic_start_time = time.time()
```

```
            all_sums(Random_Strings_Set)
            grandparent_finder(Random_Strings_Set)
            dynamic_end_time = time.time()
            runtime += (dynamic_end_time - dynamic_start_time)

        average_runtimes.append(runtime / 10)

    plt.plot(range(0, n, 1), average_runtimes, color='blue')
    plt.title('Experimental time complexity for Dynamic programming algorithm')
    plt.xlabel('Input size as the number of strings')
    plt.ylabel('Average runtime in seconds')
    plt.show()

experimental_dynamic(20)



#this code cell takes time to run
#Comparison of greedy and dynamic programming runtime experiment


import time
import matplotlib.pyplot as plt
import numpy as np

def compare_greedy_dynamic(n):
    '''
    This function performs an experimental average runtime calculation for
    the greedy and Dynamic Programming algorithm to compare
    as we vary the input size which is the number of strings.

    Inputs
    --------
    n: int
        The maximum number of strings to test the algorithm for

    Returns
    ---------
    None
        This function produces a graph for the average runtime per input
        size for greedy and dynamic
    '''
    # this section is for the greedy algorithm
```

```
    greedy_average_runtimes = []

    for i in range(n):
        # generate input size of length i
        Random_Strings_Set = strings_maker(i)
        runtime = 0

        # repeat each experiment 5 times
        for t in range(5):
            greedy_start_time = time.time()
            len_lcs_matrix = np.zeros((len(Random_Strings_Set),\
                                       len(Random_Strings_Set)))
            length_matrix_experimental(len_lcs_matrix, Random_Strings_Set, i)
            greedy_end_time = time.time()
            runtime += (greedy_end_time - greedy_start_time)

        greedy_average_runtimes.append(runtime / 5)

    # this section is for the dynamic programming algorithm
    dynamic_average_runtimes = []

    for i in range(n):
        # generate input size of length i
        Random_Strings_Set = strings_maker(i)
        runtime = 0

        # repeat each experiment 5 times
        for t in range(5):
            dynamic_start_time = time.time()
            all_sums(Random_Strings_Set)
            grandparent_finder(Random_Strings_Set)
            dynamic_end_time = time.time()
            runtime += (dynamic_end_time - dynamic_start_time)

        dynamic_average_runtimes.append(runtime / 5)

    plot1 = plt.plot(range(0, n, 1), greedy_average_runtimes, color='green',\
                     label='Greedy')
    plot2 = plt.plot(range(0, n, 1), dynamic_average_runtimes, color='blue', \
                     label='Dynamic')

    plt.title('Experimental time complexity for Greedy and Dynamic programming \
    algorithm')
```

```
    plt.xlabel('Input size as the number of strings')
    plt.ylabel('Average runtime in seconds')
    plt.legend()
    plt.show()

compare_greedy_dynamic(20)



#this code isn't generated by me.
'''
Slowikowski, K. (2020). A simple version of the Needleman-Wunsch
algorithm in Python [Source Code].
https://gist.github.com/slowkow/06c6dba9180d013dfd82bec217d22eb5
'''
import numpy as np

def needleman_wunsch_alignment(x, y, match=1, mismatch=1, gap=1):
    """
    Aligns two strings, highlighting differences with dashes and
    unmatching characters in the alignment.

    Parameters:
    x (str): The first input string.
    y (str): The second input string.
    match (int): The score for a match (default is 1).
    mismatch (int): The score for a mismatch (default is 1).
    gap (int): The score for a gap (default is 1).

    Returns:
    list: A list containing two strings representing the aligned sequences.
    """
    nx = len(x)
    ny = len(y)

    # Optimal score at each possible pair of characters.
    F = np.zeros((nx + 1, ny + 1))
    F[:, 0] = np.linspace(0, -nx * gap, nx + 1)
    F[0, :] = np.linspace(0, -ny * gap, ny + 1)

    # Pointers to trace through an optimal alignment.
    P = np.zeros((nx + 1, ny + 1))
    P[:, 0] = 3
```

```
    P[0, :] = 4

    # Temporary scores.
    t = np.zeros(3)

    for i in range(nx):
        for j in range(ny):
            if x[i] == y[j]:
                t[0] = F[i, j] + match
            else:
                t[0] = F[i, j] - mismatch
            t[1] = F[i, j + 1] - gap
            t[2] = F[i + 1, j] - gap

            tmax = np.max(t)
            F[i + 1, j + 1] = tmax

            if t[0] == tmax:
                P[i + 1, j + 1] += 2
            if t[1] == tmax:
                P[i + 1, j + 1] += 3
            if t[2] == tmax:
                P[i + 1, j + 1] += 4

    # Trace through an optimal alignment.
    i = nx
    j = ny
    rx = []
    ry = []

    while i > 0 or j > 0:
        if P[i, j] in [2, 5, 6, 9]:
            rx.append(x[i - 1])
            ry.append(y[j - 1])
            i -= 1
            j -= 1
        elif P[i, j] in [3, 5, 7, 9]:
            rx.append(x[i - 1])
            ry.append('-')
            i -= 1
        elif P[i, j] in [4, 6, 7, 9]:
            rx.append('-')
            ry.append(y[j - 1])
```

```
            j -= 1

    # Reverse the strings.
    rx = ''.join(rx)[::-1]
    ry = ''.join(ry)[::-1]

    aligned_sequences = [str(rx), str(ry)]
    return aligned_sequences


def generate_alignments(parental_relationships):
    """
    Generates alignments between each parent and child pair.

    Parameters:
    parental_relationships (list): A list of tuples, where each tuple
    represents a parent-child relationship.
    Each tuple contains two strings, representing the parent and child sequences.

    Returns:
    list: A list containing aligned sequences for each parent-child pair.
          Each element of the list is a tuple with two strings representing the
          aligned sequences.
    """
    aligned_strings = [needleman_wunsch_alignment(parent, child) for parent, \
                       child in parental_relationships]
    return aligned_strings


def count_operations(parent, child):
    """
    Counts the number of each operation for each pair of parent and child.

    Parameters:
    parent (str): The parent sequence.
    child (str): The child sequence.

    Returns:
    list: A list containing the count of deletions, insertions, and mutations.
          The elements are in the order [deletions, insertions, mutations].
    """
    deletions = 0
```

```
        insertions = 0
        mutations = 0

        for i in range(len(parent)):
            # If parent has '-' and child has a character in the same place,
            it's an insertion
            if parent[i] == '-' and child[i] != '-':
                insertions += 1
            # If parent has a character and child has '-' in the same place,
            it's a deletion
            elif parent[i] != '-' and child[i] == '-':
                deletions += 1
            # If parent has a character different than the child, it's a mutation
            elif parent[i] != '-' and child[i] != '-' and parent[i] != child[i]:
                mutations += 1

        operations = [deletions, insertions, mutations]

        return operations



    def count_operations_for_relationships(parental_relationships):
        """
        Gets the number of each operation for all pairs of parents and children.

        Parameters:
        parental_relationships (list): A list of lists where each sublist
        contains the parent and child strings.

        Returns:
        list: A list containing the number of operations
        (deletions, insertions, mutations) for each pair.
        """
        # Generate alignments for all parent-child pairs
        parent_child_alignments = generate_alignments(parental_relationships)

        # Initialize a list to store operation counts for each pair
        operation_counts = []

        # Iterate through each alignment and count operations
        for alignment in parent_child_alignments:
            operation_counts.append(count_operations(alignment[0], alignment[1]))
```

```
    return operation_counts


# Example usage
parental_relationships = [[b, c], [b, f], [c, a], [c, e], [f, d], [f, g]]
count_operations_for_relationships(parental_relationships)



def print_modification_probabilities(parental_relationships):
    """
    Prints the probability of deletion, insertion, and mutation
    for each character
    in the parent string to form the child string, and the average
    across all relationships.

    Parameters:
    parental_relationships (list): A list of lists where each sublist
    contains the parent and child strings.
    """
    # Count operations for all parent-child pairs
    operation_counts = count_operations_for_relationships(parental_relationships)

    # Extract parents from the relationships
    parents = [relationship[0] for relationship in parental_relationships]

    p_order = 0
    total_deletion_probability = 0
    total_insertion_probability = 0
    total_mutations_probability = 0

    # Iterate through each pair's operation count
    for count in operation_counts:
        # Calculate final probabilities for each relationship
        final_deletion_probability = count[0] / len(parents[p_order])
        final_insertion_probability = count[1] / len(parents[p_order])
        final_mutations_probability = count[2] / len(parents[p_order])

        # Accumulate probabilities for averaging
        total_deletion_probability += final_deletion_probability
        total_insertion_probability += final_insertion_probability
        total_mutations_probability += final_mutations_probability
```

```
        # Print the estimation for each parental-child pair
        print(f"Estimation for Parent-Child Pair {p_order + 1}:")
        print(f"Deletion Probability: {final_deletion_probability}")
        print(f"Insertion Probability: {final_insertion_probability}")
        print(f"Mutation Probability: {final_mutations_probability}\n")

        p_order += 1

    # Calculate and print the average estimation across all parental-child pairs
    avg_deletion_probability = total_deletion_probability / len(parental_relationsh
    avg_insertion_probability = total_insertion_probability /\
    len(parental_relationships)
    avg_mutations_probability = total_mutations_probability /\
    len(parental_relationships)

    print("Average Estimation Across All Parental-Child Pairs:")
    print(f"Average Deletion Probability: {avg_deletion_probability}")
    print(f"Average Insertion Probability: {avg_insertion_probability}")
    print(f"Average Mutation Probability: {avg_mutations_probability}")


# Example usage
parental_relationships = [[b, c], [b, f], [c, a], [c, e], [f, d], [f, g]]
print_modification_probabilities(parental_relationships)
```