Project 3 - Final Computational Application

Minerva University

CS110: Data Structures and Algorithms

Prof. B. Richard

December 26, 2022

Decoding the relationships between genes

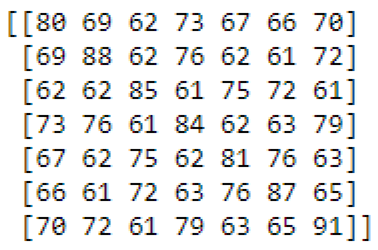
In this assignment, I investigate the relationship between genes to establish the genealogical tree and estimate the probability of each possible change a string can undergo: insertion, deletion, or mutation.

1. I created a function that calculated the longest common sequence between any 2 strings. I included the suggested test cases and my function passed all of them. Apart from the suggested test cases, I included three more. I examined the output of the function if the 2 strings given have no characters in common and if they are totally identical. I also examined the output of 2 empty strings and included them in my code guard clause to prevent the function of running in any case when at least of the strings is empty because we do not need it to run to know that there are no common characters.

I used the function to examine the set of strings given. As per my examination, the maximum number of combinations of possible longest common subsequences (lcs) is 13428744 between b and f, and the minimum number of combinations of lcs is 4 between e and f.

I created the matrix that shows the length of the longest common subsequence between any 2 strings. The matrix is represented in figure 1.

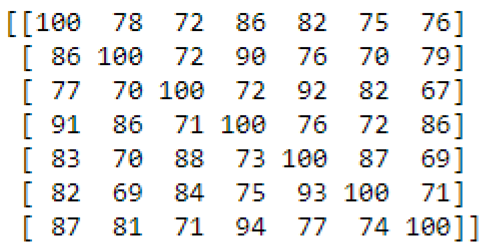
2. Generate the matrix



```
[[80 69 62 73 67 66 70]
 [69 88 62 76 62 61 72]
 [62 62 85 61 75 72 61]
 [73 76 61 84 62 63 79]
 [67 62 75 62 81 76 63]
 [66 61 72 63 76 87 65]
 [70 72 61 79 63 65 91]]
```

Figure 1: matrix of the longest common sequence between any 2 strings

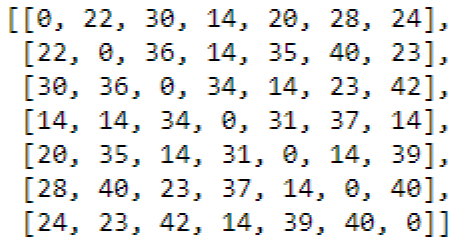
Based on the matrix obtained above we can see that there are some stings that are more closely related to each other than others. We can also see that our matrix is mirrored. However, given that the strings have different lengths it does not necessarily reflect the real relationship between them. For example, if a string A and a string B have the longest common subsequence of 50 when A has a length of 50 and B has a length of 150 A would be much stronger related to B than B to A. There might be a third-string C that would have lengths of 300 and 90 in the longest common subsequence with B. In this case in the presented table, we will see that C is more strongly related to B than A. To consider the length of the strings I incorporated an additional part in my function to create the same table but with the information represented in percentages. As we can see in figure 2 if we take a look at the percentages the matrix would not be mirrored anymore and we will get a deeper insight in the relationship between the strings.



```
[[100  78  72  86  82  75  76]
 [ 86 100  72  90  76  70  79]
 [ 77  70 100  72  92  82  67]
 [ 91  86  71 100  76  72  86]
 [ 83  70  88  73 100  87  69]
 [ 82  69  84  75  93 100  71]
 [ 87  81  71  94  77  74 100]]
```

Figure 2: matrix represents the longest common sequence in percentages.

To examine the precise relationships between strings I would use the minimum edit distance which is also known as Levenshtein distance. This metric shows how many changes should be made to a string A so that it becomes a string B.



```
[[0, 22, 30, 14, 20, 28, 24],
 [22, 0, 36, 14, 35, 40, 23],
 [30, 36, 0, 34, 14, 23, 42],
 [14, 14, 34, 0, 31, 37, 14],
 [20, 35, 14, 31, 0, 14, 39],
 [28, 40, 23, 37, 14, 0, 40],
 [24, 23, 42, 14, 39, 40, 0]]
```

Figure 3: the figure represents the minimum edit distance.

I examined figure 3 and noted the two closest connections for each element. Examining only 1 relationship might not be enough to see a clear pattern and examining 3 connections of every node would result in too much noise in the data analyzed.

A is the most closely related to D and E.

A→ D, A→E

B is the most closely related to D and A

B→D, B→A

C is the most closely related to E and F

C→E, C→F

D is the most closely related to A and B and G. Here there are 3 connections since all of them have the same value.

D→A, D→G, D→B

E is the most closely related to C and F

E→C, E→F

F is the most closely related to E and C

F→E, F→C

G is the most closely related to D and B

G→D, G→B

I focus on detecting mutual connections. The set of nodes E C F are all the closest to each other (mutually connected) so we can assume they are a part of one subfamily. To understand who is the parent node and who are the children nodes we will take a look at the strength of the relationships. The longest distance will be between 2 brothers since they are connected with each other not directly but throught the common parent. Since 23 is the largest distance and it is between C and F they are brothers. Therefore E is a parent node of C and F.

Another set of 3 although not that distinct is B G D. The largest distance is between G and B so they are brothers. D has a strong connection with both of them so it is a parent node. A third strong connection the D node has is with A. So A is a parent node of D. A also has a strong connection with E, the previously found parent of another parent of the tree. Therefore we can say that A is a parent of both E and D. Therefore A is a grandparent of the tree.

3. Examine the precise relationships between strings.

Local strategy:

My local strategy will consist of creating all the possible trees and choosing the best one. I use the top-down approach. That means that I start from the top of the tree parent node and I finish at the bottom level. I will take one step at a time and make a decision looking only at one row. I will use greedy properties by making a local optimal solution (choosing the best match from a single row) to reach global optimal solution.

I will choose each element as a parent and will build a tree for each of those parents. I will take a row that corresponds to the index of the parents. All the values in this row represent the minimum edit distance between the parent and every string. I will choose 2 elements with the smallest value. Since this is a root it cannot have parents so the closets element for it would be its children. After that, I will take each of those children as a parent and will find their children and continue until all the nodes are used.

When all the trees are constructed I will sum up all the edit distances between them to determine which tree has the lowest edit distance, thus is the best.

This strategy is very complex because we have to consider every combination possible and create every possible tree to find the one that would be optimal. It does not guarantee reaching the globally optimal solution since it does not consider the whole picture and makes only the locally optimal choice. For example, it might be possible that if the distance between the grandparent and parent is a bit bigger it would drastically decrease the distance between the parent and child which would result in the overall decrease of the edit distance of the tree. However, we will not be able to figure it out with this approach since we do not look at parent-child relationships when we find grandparent-parent relationships. Therefore it is not guaranteed to find the globally optimal solution with this strategy.

Global strategy:

My global strategy would be looking at the whole matrix instead of examining it row by row. I will use the bottom-up approach by starting with the lower level (in this case it is grandchildren) and moving up to the parent node. I will find the highest edit distance on the matrix. The highest edit distance means that the elements are on opposite sides from each other. In our case of 7 elements they are located on the third level from different sides (have different parent nodes). When we found the combination of elements that produce the highest edit distance we can see which exact elements result in that number. Those elements are considered “cousins”. After that, we examine each of these “cousins” to find their “brother” nodes, the second node from the same parent. We know that the minimum edit value of the element is when we compare it to itself since no changes should be made. The second minimum value should be the connection with a parent or a connection with a child. Since we focus on the lowest level, the node does not have children so the second highest connection will be with its parent. The third highest connection will be with another child node of the same parent, its ‘brother’. We find ‘brothers’ for each of these distant elements. As we have both children we can merge them to consider them as one entity. In those merged columns we find the highest element that will represent the most distant element to the children - “uncle”. If we switch those elements we will get the set of parent nodes and children. After we finished those steps we have only 1 element left - the root node of the tree. We run loops to determine which element was not used. As we did so we will construct the tree in the form of lists. This strategy will help to get the globally optimal solution since we are looking at all relationships at the same time. It allowes us to make some sacrifices (less optimal solutions) on a local level to reach the best solution possible on a global level. The ability of this approach to reach a globally optimal solution was supported by the evidence that the output of the global strategy function matched with the theoretically hypothesized tree created above. On the other hand, the output from the function based on the local approach did not produce the expected output.

The resulting tree from the global solution (an index-based version) is presented in figures 4 and 5.

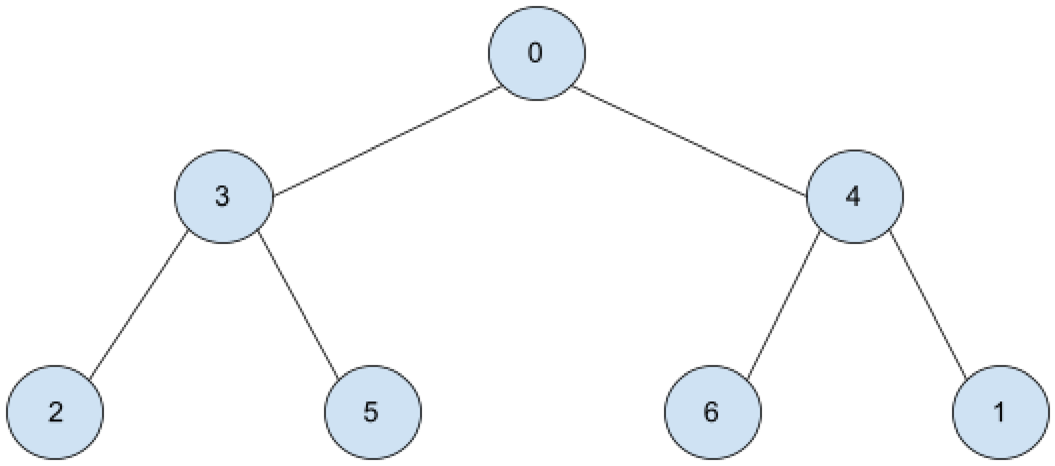


Figure 4: index-based version of the global tree

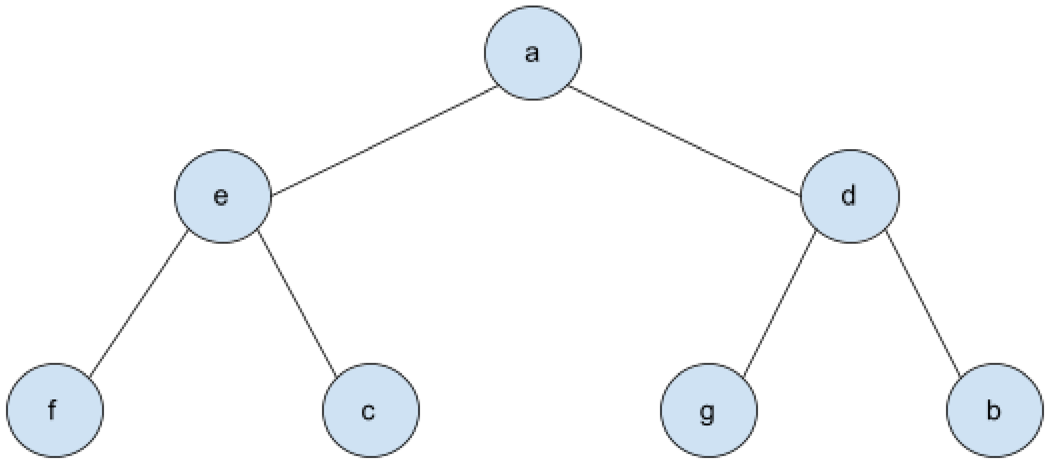


Figure 5: letter-based version of the global tree,

However, the local approach is much easier to implement on a larger scale since the code is easy to scale. The global approach is very efficient but unfortunately, the scalability of my implementation is limited and it should be edited for a bigger scale.

4. For each algorithmic approach, what is your solution's computational complexity to produce genealogy binary trees?

Time complexities for both trees depend on n - a number of genes and m - the length of the genes. Both algorithms share in common the calculation of the Edit distance since it outputs the matrix both approaches work with. Edit distance has a time complexity of $O(n * m)$. Matrix is constructed using 2 loops so the time complexity for the matrix part will be $O(n^2)$. So the total time complexity of the edit distance calculation and matrix construction will be $O(n^2(n * m))$ (GeeksforGeeks, 2022b).

This is the only part where the algorithm's time complexity relies on the length of the string since after the matrix is calculated we rely solely on the integers in the matrix and depend on the number of the strings. As we see in the time complexity notation the number of the genes matters 3 times more than the length of the genes when it comes to calculating the time matrix which is the input for both trees.

Time complexity for the local approach:

Local approach will have comparatively worse time complexity because we have to deconstruct a tree starting with each parent node. It will scale as the number of genes increases because the more genes we add the more tree we will have to construct.

To calculate the tie complexity for the whole approach I will calculate the time complexity for each function consisting of:

- Find_min: it contains 2 for loops, 1 of them is based on the length of the row which is basically the number of genes n. Second repeats 2 times. So time complexity is about $O(2n)$
- Creating matrix time complexity was defined above and it is $O(n^2(n * m))$
- Cleaning table takes $O(n^2)$ because of the 2 for loops.
- Finding children will take about $O(n\ )$ because of the for loop. While loop although exists on a large scale the complexity it adds would not be significant.
- Greedy approach takes $O(n\ )$ because of the for loop
- Finding sum is $O(n\ )$ because of the for loop

Total time complexity is $O(2n + 3n + n^2 + n^2(n * m)) = O(5n + n^2 + n^2(n * m))$

Since 5n will not be that significant compared to exponential we can cancel it out.

$O(n^2 + n^3 + nm))$

So we will see exponential growth

Also, we will see more extreme growth if the number of genes increases and the length of genes stays the same the vice versa since n contributes to the time complexity much more than m.

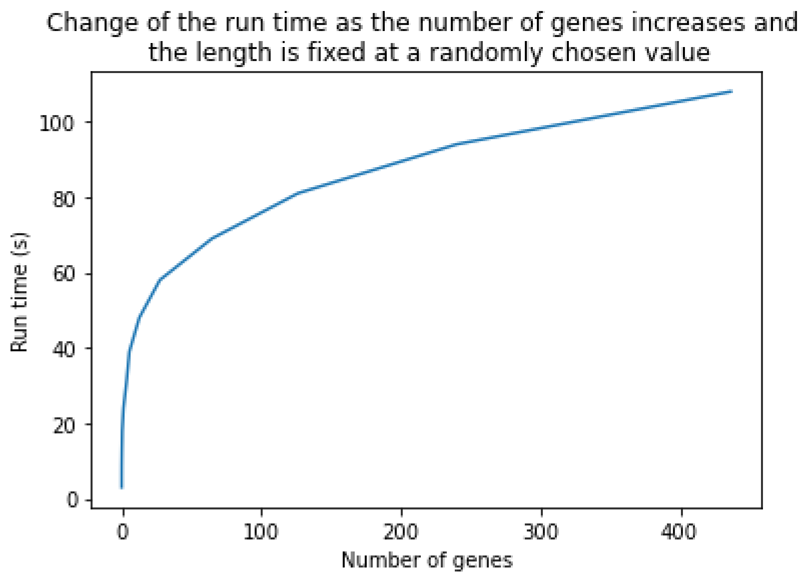


Figure 6: figure represents the growth of the run time of the greedy approach over time. The length of strings stays the same but the number of strings increases from 0 to 400.

Figure 6 shows that the dependency between the increase in the run time and the increase in the number of genes is exponential as it was hypothesized. It means that for each gene added to the tree the time to construct the tree will increase by 2+m times.

For the global approach:

Global approach would be more efficient than the local because we do not run the function for each node, so its time complexity is expected to be smaller that the one of the greedy.

- Constructing a matrix takes $O(n^2(n * m))$
- To find cousins we will need to do 1 for loop which will result in $O(n)$ time complexity.
- In finding the brother function we will need to do sorting 2 times. The time complexity of

the in-build sort function in Python in $O(2nlogn)$(GeeksforGeeks, 2022a).

- Finding parents have 2 loops one runs through all the rows and the second only through 2 elements. It results in the $O(2n)$ time complexity
- Merging will take $O(n^2)$ because of 2 loops
- Building the tree will take $O(2n)$

Total time complexity is

$$O(2n + 2n + n^2 + 2nlogn + n + n^2(n * m)) = 5n + 2nlogn + n^2(n * m))$$

If we follow the same principle as before and cancel out $5n$ we will end up with

$O(2nlogn + n^2(n * m))$ which is significantly smaller than the time complexity of the greedy approach.

So if we compare the scaling of both approaches for an increase of 1 gene greedy approach will increase by exponentially more time than the dynamic approach.

5. How would you estimate the probabilities of insertions, deletions, and mutations?

a. Algorithmic approach for estimating the probabilities of insertion, deletion, and mutation

We know the longest common subsequence for any two strings. It means that the rest that is not common was either added, deleted, or changed. My algorithmic strategy would focus on analyzing the part of the strings where they differ from each other. Let's assume that we know that string B was created as a result of the mutilation, insertion, and deletion of the elements into string A. (A is a parent string of B).

If string A is smaller than string B, the insertion would be found by simply finding the difference between the two strings since to get from A to B we need to insert additional elements. If we want to consider the opposite case, string B is smaller than string A we would assign the difference between the strings to the deletion since we need to remove some elements to get from B to A. The mutilation is equal to the length of the smaller string without the common values (LCS). Example:

String A has the length of 30 elements

String B has the length of 25 elements

The longest common subsequence of A and B is 10, so we subtract 10 from the length of both strings and will operate with the values we get.

$$30 - 10 = 20$$

$$25 - 10 = 15$$

We see that the string B is smaller than A. Since A is the ancestor of B the difference between the strings is the result of deletion.

$$20 - 15 = 5$$

As we subtract the remaining length of one element from another we get the number of deletions that were made.

The number of mutations is the part of A that was not deleted and does not match B. The number of mutations is equal to 15.

To find the probability we have to divide the number of all manipulations and divide the number of each manipulation by the total.

In this case:

$total = 5 + 15 = 20$

$P_{deletion} = 5/20 = 0.25$

$P_{mutation} = 15/20 = 0.75$

$P_{insertion} = 0/20 = 0$

The total number will be found by dividing the number of changes by the length of the string that was changed.

b. Produce an estimation in Python that would take your algorithmic strategy into practice.

I produced an estimation in Python. I obtained the following results:

$P_{mutation} \approx 0.195$

$P_{deletion} \approx 0$

$P_{insertion} \approx 0.049$

These probabilities seem reasonable since they are less than 5% . We also know that the minimum edit distance between the closest nodes in the tree is relatively small (14-20 edits) which serves as evidence supporting these results. The probability for deletion is 0 because the length of the tree increases by levels. This is supported by figure 1 in Appendix.

6. List all the LOs and HCs you have exercised while working on this final assignment.

a. LO Applications:

#CodeReadability

I added docstrings and comments where it was appropriate. All my docstrings were clean and formatted in the same way. I visually separated the description of the function and the input/output based on the feedback from the previous assignment. I assigned logical and self-explanatory names for my variables so that I did not have to write extra comments for more explanation. I also made sure that the code itself is clear and I avoided unnecessary steps that might be confusing. I kept my code as simple and succinct as possible.

Word Count: 91 words

#Professionalism

I did the explanation part of the assignment clearly. I have also proofread my assignment to avoid typos as I was advised in the previous assignment. I also proofread my comments and docstrings. I was also staying within the word count where it was specified. I formatted my assignment according to the APA style. I used professional language but at the same time avoided complicated expressions because my audience was a confused student and I did not want to confuse them even more.

Word Count: 83 words

#ComplexityAnalysis:

I applied this LO by analyzing each function both local and global approaches consist of and creating equations of the time complexity. I pointed out which parts of the functions contribute tot he time complexity and how they affect the run time. I explained the significance of time complexity in the real life - what it means for the time and change in the number of genes.

Word Count: 67 words

#ComputationalCritique

I made a strong application of this LO by providing efficiently connecting analytical and experimental analysis. I also ensured that I created appropriate experiments that provide useful insight into the efficiency of the code. I liked my reasoning on this LO to the LO #ComplexityAnalysis to make my application even stronger. I also explained which features of the global algorithm make it more efficient than the local algorithm.

Word Count: 68 words

#AlgoStratDataStrat

I explained my algorithmic strategy for both local and global algorithms in detail and went beyond a simple translation of the pseudocode to plain English. I exampled why I did what I did and how it improved my code. I discussed the usefulness of both local and global approaches. I pointed out the disadvantages of my implementation of the global strategy and an idea of how it could have been solved.

Word Count: 71 words

#PythonProgramming

All my code is working. I fixed all the bugs in my code. I wrote efficient code by using such strategies as guard clauses to prevent the function from running when it is not necessary. I avoided unnecessary loops that would negatively impact my time complexity. I also used multiple functions to separate pieces of the code and make it more clear.

Word Count: 62 words

b. HC Applications

#dataviz:

I applied this HC by producing high-quality data visualizations. I appropriately labeled the axis. I added an informative title and created captions for all the visualizations presented in this paper. I provided different types of data visualizations like graphs, line plots, and matrixes. I included multiple figures in this paper.

Word Count: 50 words

#audience:

I took into consideration my target audience of a confused student and explained everything in a detailed enough way. I avoided complex language and jargon. I also used examples to ease explanations. I used clear language and avoided vague statements.

Word Count: 40 words

#deduction:

I applied deductive reasoning when I was analytically analyzing a matrix to construct a tree. I defined premises and if the premises are true the conclusion should be true so I made the conclusion about the structure of the tree based on the proven premises about the parents. I concluded about the parents based on the distance in the interconnected triangles identified. I identified the connections in triangles by finding the mutually connected couples.

Word Count: 74 words

#algorithms:

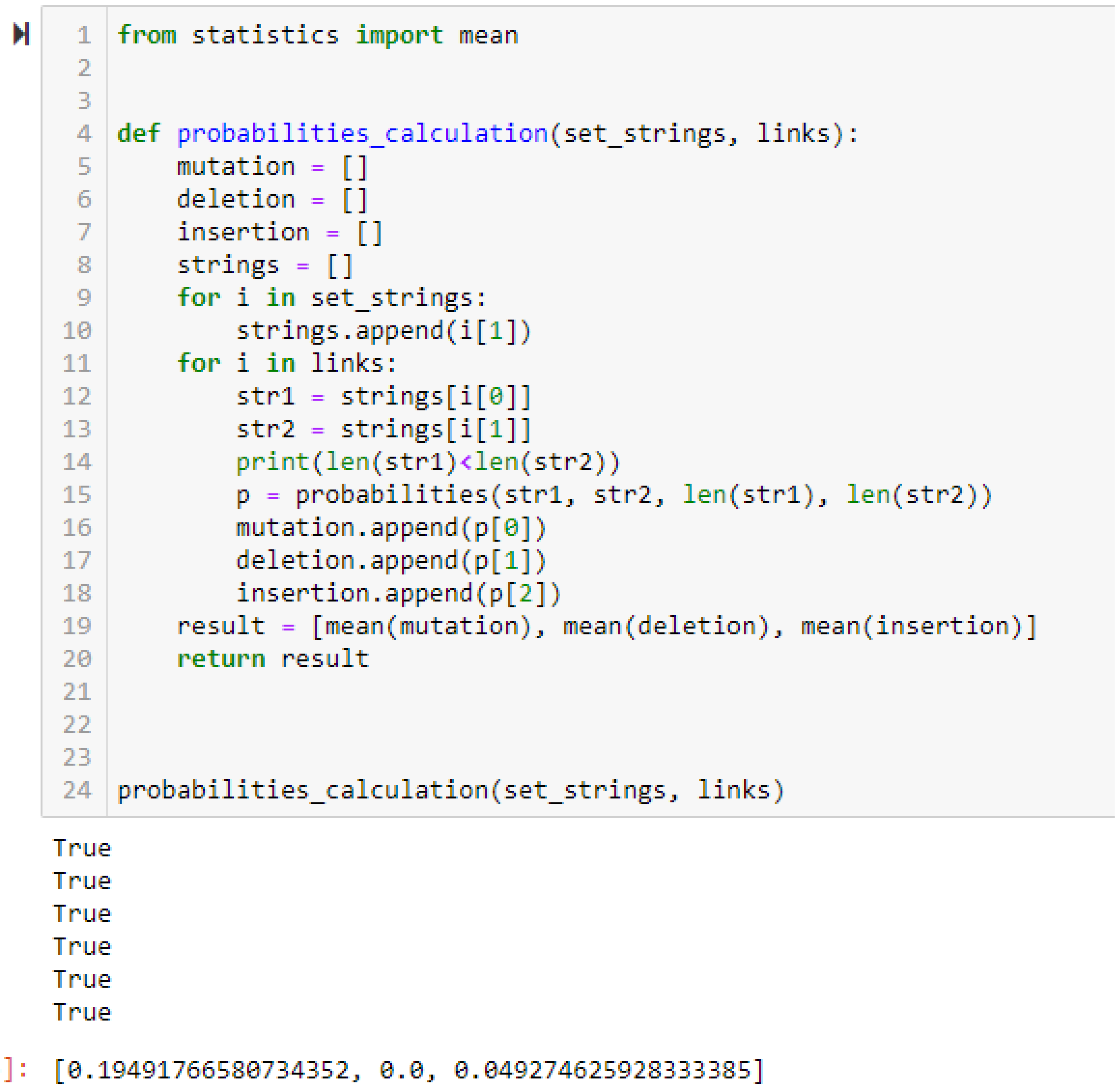
I used this HC since developing the solution for the given problem will require strong algorithmic thinking. I accounted for all the potential edge cases and made sure that my code would be able to deal with them. I identified an appropriate algorithmic strategy (dynamical programming - global strategy) and I justified my choice. Before writing the code I developed the pseudo code to have a between understanding of what I am aiming for. I was also drawing the execution of the code step by step while fixing bugs. I provided a thorough explanation of my code.

Word Count: 96 words

.

Total Word Count: 2823 words

Appendix:

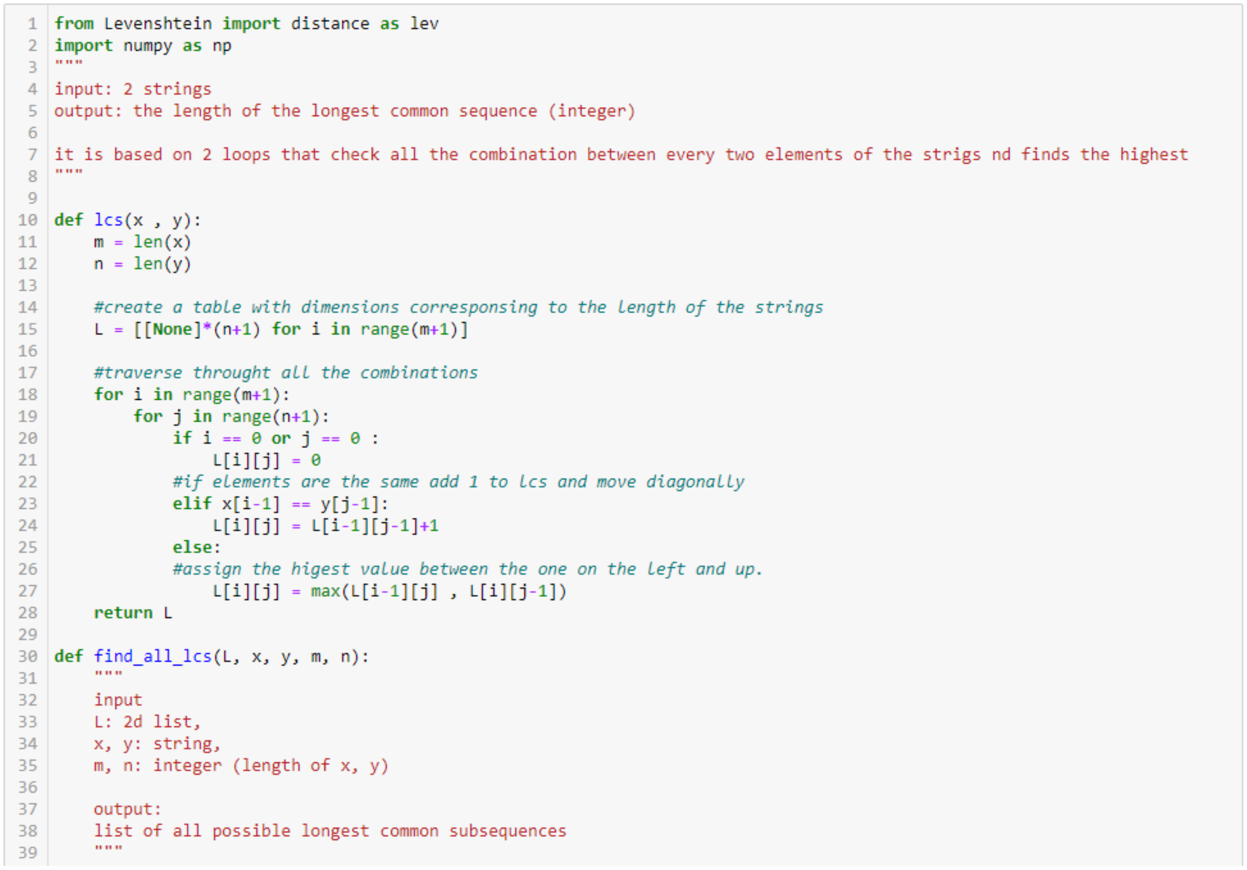


```
from statistics import mean


def probabilities_calculation(set_strings, links):
    mutation = []
    deletion = []
    insertion = []
    strings = []
    for i in set_strings:
        strings.append(i[1])
    for i in links:
        str1 = strings[i[0]]
        str2 = strings[i[1]]
        print(len(str1)<len(str2))
        p = probabilities(str1, str2, len(str1), len(str2))
        mutation.append(p[0])
        deletion.append(p[1])
        insertion.append(p[2])
    result = [mean(mutation), mean(deletion), mean(insertion)]
    return result



probabilities_calculation(set_strings, links)
```

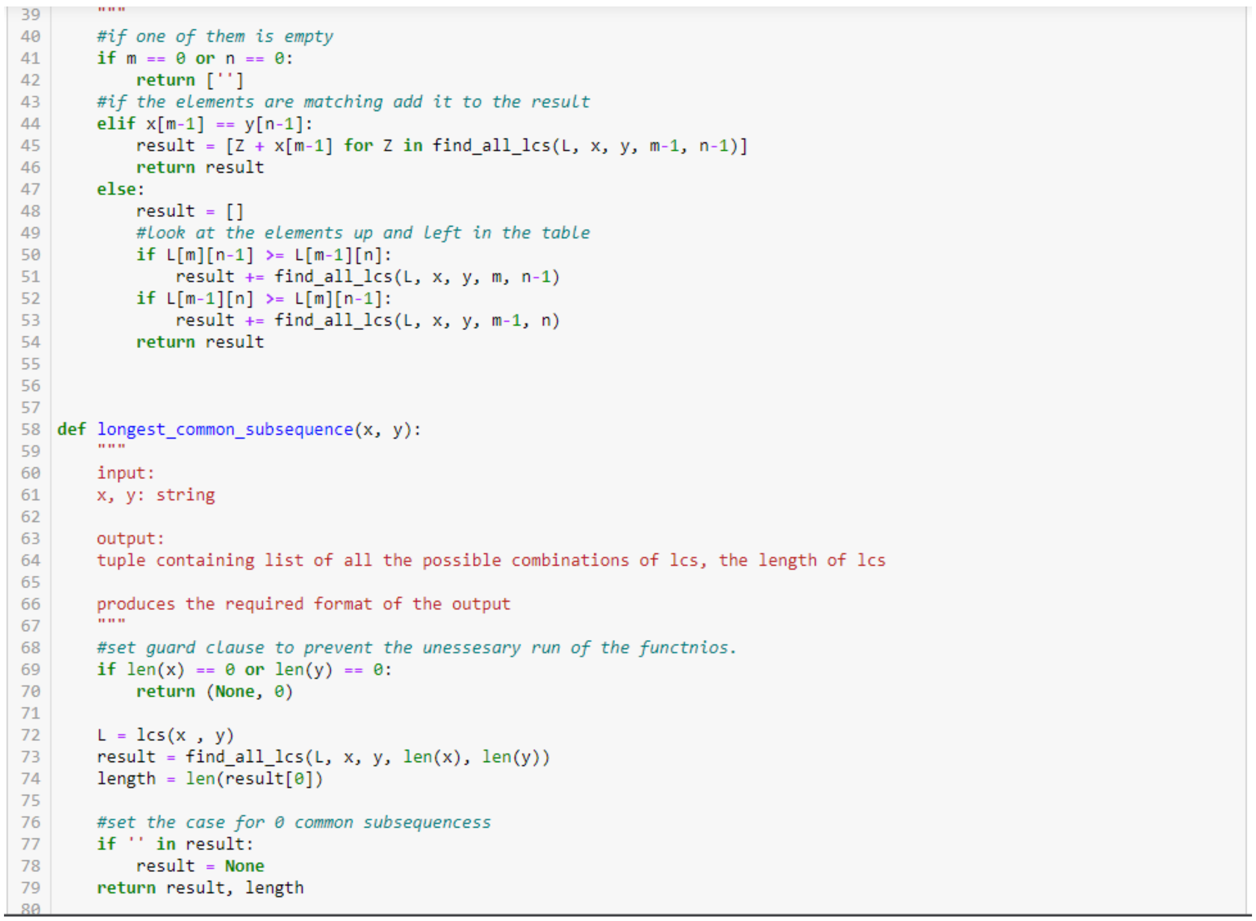
```
True
True
True
True
True
True

5]: [0.19491766580734352, 0.0, 0.0492746259283333385]
```

Figure 6: This figure proves the plausibility of the 0% deletion probability since every other string in the tree is longer than its parent.

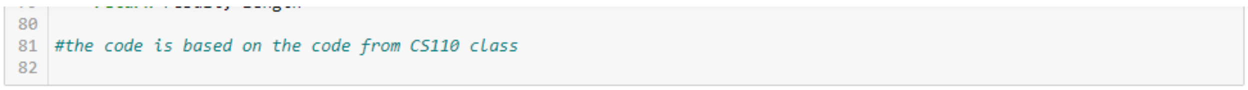
Code:



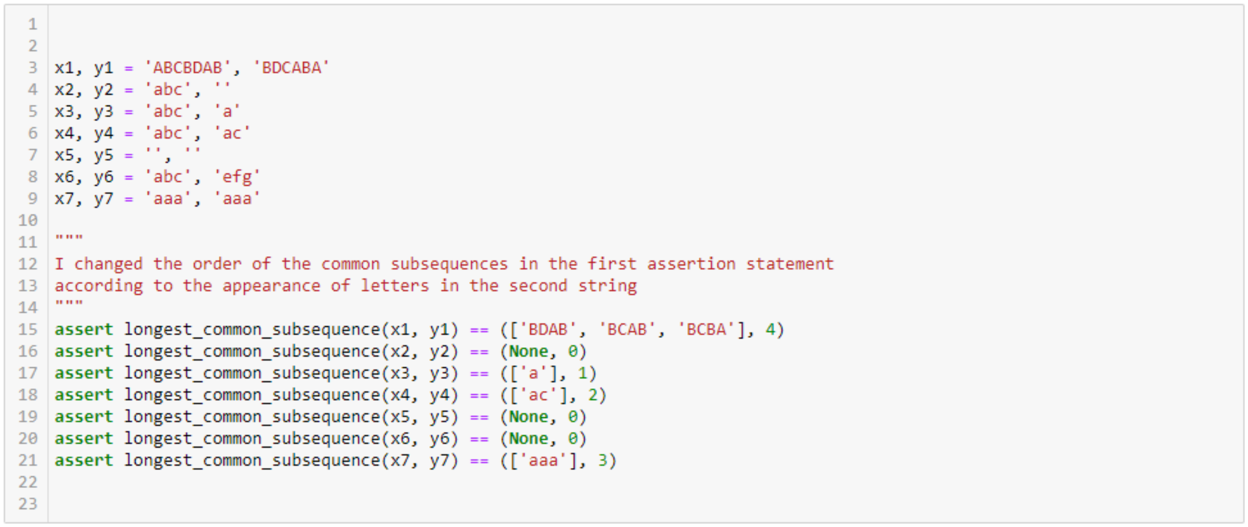
```python
from Levenshtein import distance as lev
import numpy as np
"""
input: 2 strings
output: the length of the longest common sequence (integer)

it is based on 2 loops that check all the combination between every two elements of the strigs nd finds the highest
"""

def lcs(x , y):
    m = len(x)
    n = len(y)

    #create a table with dimensions corresponsing to the length of the strings
    L = [[None]*(n+1) for i in range(m+1)]

    #traverse throught all the combinations
    for i in range(m+1):
        for j in range(n+1):
            if i == 0 or j == 0 :
                L[i][j] = 0
            #if elements are the same add 1 to lcs and move diagonally
            elif x[i-1] == y[j-1]:
                L[i][j] = L[i-1][j-1]+1
            else:
            #assign the higest value between the one on the left and up.
                L[i][j] = max(L[i-1][j] , L[i][j-1])
    return L

def find_all_lcs(L, x, y, m, n):
    """
    input
    L: 2d list,
    x, y: string,
    m, n: integer (length of x, y)

    output:
    list of all possible longest common subsequences
    """
```



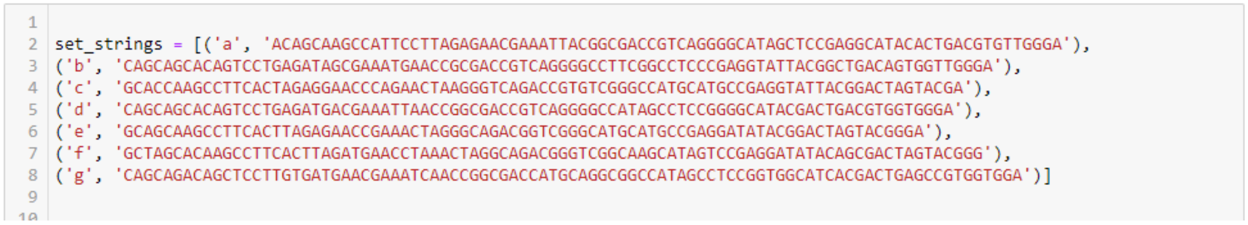
```python
    """
    #if one of them is empty
    if m == 0 or n == 0:
        return ['']
    #if the elements are matching add it to the result
    elif x[m-1] == y[n-1]:
        result = [Z + x[m-1] for Z in find_all_lcs(L, x, y, m-1, n-1)]
        return result
    else:
        result = []
        #look at the elements up and left in the table
        if L[m][n-1] >= L[m-1][n]:
            result += find_all_lcs(L, x, y, m, n-1)
        if L[m-1][n] >= L[m][n-1]:
            result += find_all_lcs(L, x, y, m-1, n)
        return result



def longest_common_subsequence(x, y):
    """
    input:
    x, y: string

    output:
    tuple containing list of all the possible combinations of lcs, the length of lcs

    produces the required format of the output
    """
    #set guard clause to prevent the unessesary run of the functnios.
    if len(x) == 0 or len(y) == 0:
        return (None, 0)

    L = lcs(x , y)
    result = find_all_lcs(L, x, y, len(x), len(y))
    length = len(result[0])

    #set the case for 0 common subsequencess
    if '' in result:
        result = None
    return result, length
```



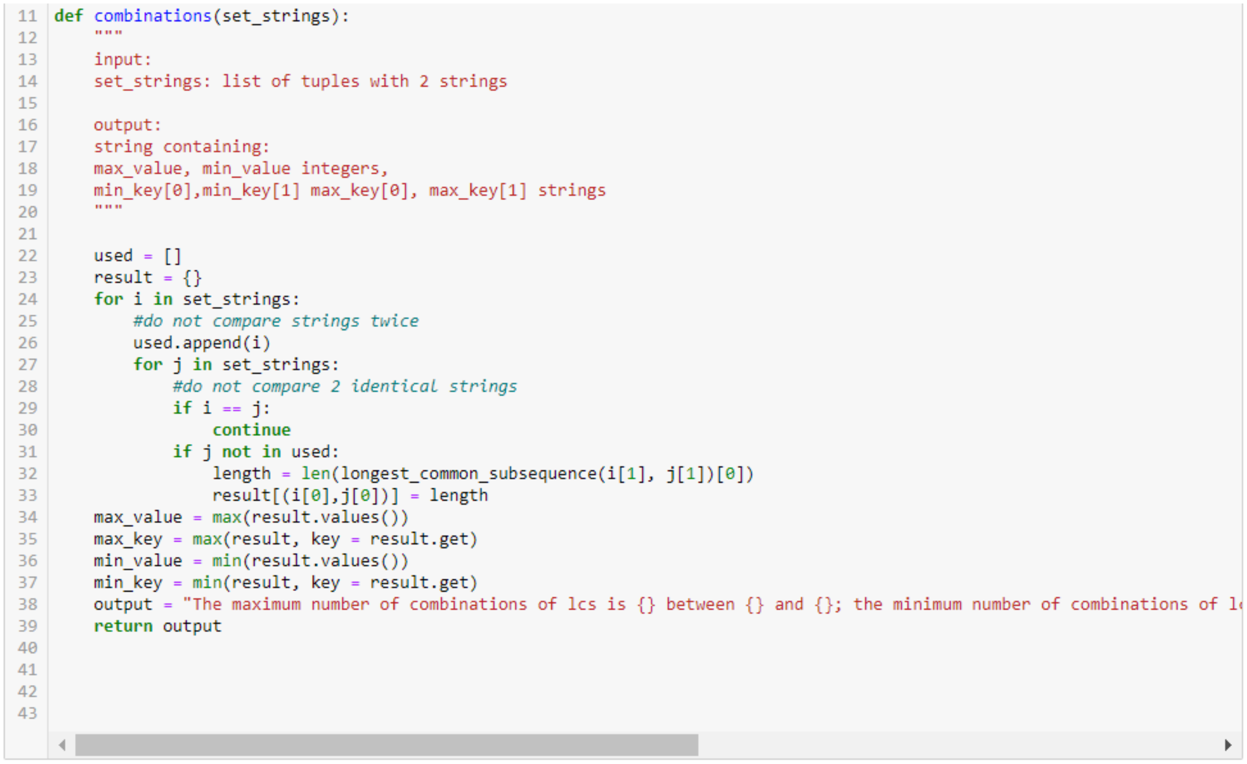
```python

#the code is based on the code from CS110 class

```



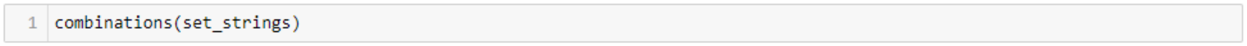
```python


x1, y1 = 'ABCBDAB', 'BDCABA'
x2, y2 = 'abc', ''
x3, y3 = 'abc', 'a'
x4, y4 = 'abc', 'ac'
x5, y5 = '', ''
x6, y6 = 'abc', 'efg'
x7, y7 = 'aaa', 'aaa'

"""
I changed the order of the common subsequences in the first assertion statement
according to the appearance of letters in the second string
"""
assert longest_common_subsequence(x1, y1) == (['BDAB', 'BCAB', 'BCBA'], 4)
assert longest_common_subsequence(x2, y2) == (None, 0)
assert longest_common_subsequence(x3, y3) == (['a'], 1)
assert longest_common_subsequence(x4, y4) == (['ac'], 2)
assert longest_common_subsequence(x5, y5) == (None, 0)
assert longest_common_subsequence(x6, y6) == (None, 0)
assert longest_common_subsequence(x7, y7) == (['aaa'], 3)


```



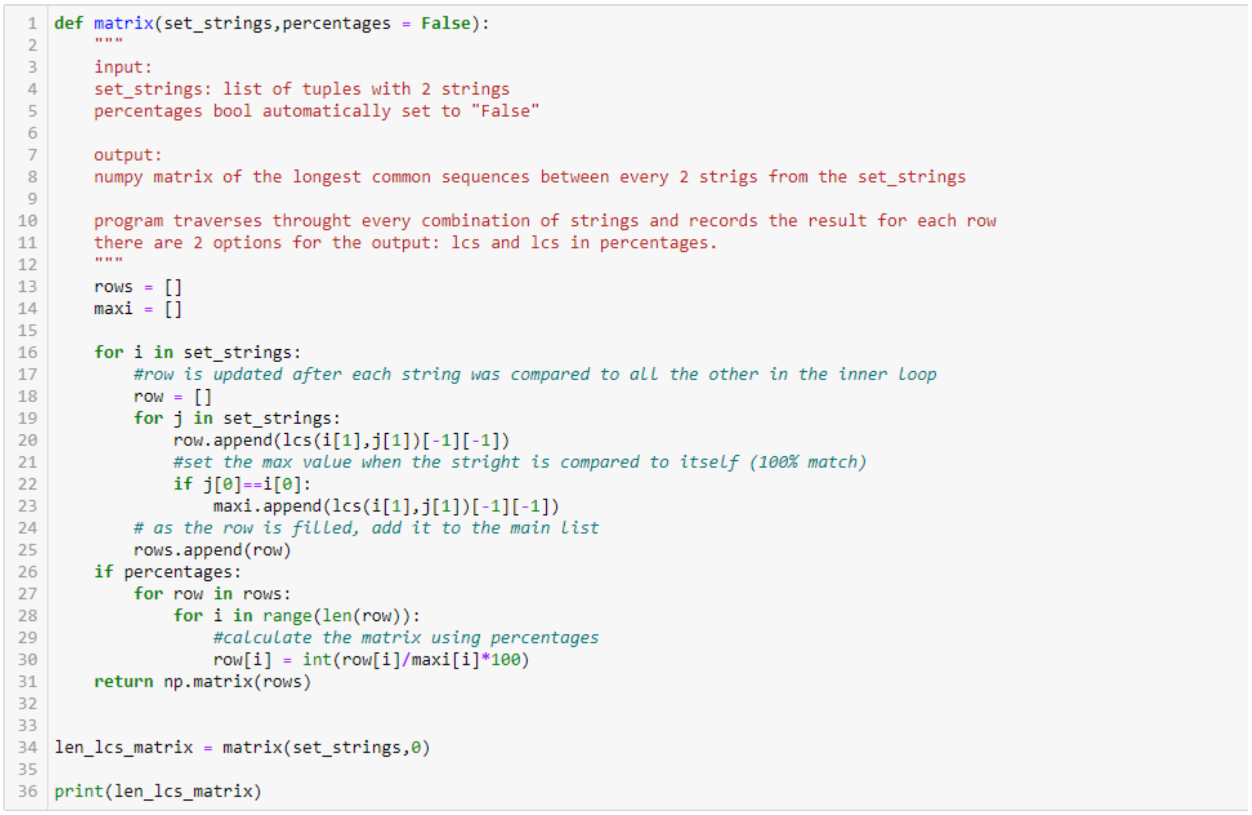
```python

set_strings = [('a', 'ACAGCAAGCCATTCCTTAGAGAACGAAATTACGGCGACCGTCAGGGGCATAGCTCCGAGGCATACACTGACGTGTTGGGA'),
('b', 'CAGCAGCACAGTCCTGAGATAGCGAAATGAACCGCGACCGTCAGGGGCCTTCGGCCTCCCGAGGTATTACGGCTGACAGTGGTTGGGA'),
('c', 'GCACCAAGCCTTCACTAGAGGAACCCAGAACTAAGGGTCAGACCGTGTCGGGCCATGCATGCCGAGGTATTACGGACTAGTACGA'),
('d', 'CAGCAGCACAGTCCTGAGATGACGAAATTAACCGGCGACCGTCAGGGGCCATAGCCTCCGGGGCATACGACTGACGTGGTGGGA'),
('e', 'GCAGCAAGCCTTCACTTAGAGAACCGAAACTAGGGCAGACGGTCGGGCATGCATGCCGAGGATATACGGACTAGTACGGGA'),
('f', 'GCTAGCACAAGCCTTCACTTAGATGAACCTAAACTAGGCAGACGGGTCGGCAAGCATAGTCCGAGGATATACAGCGACTAGTACGGG'),
('g', 'CAGCAGACAGCTCCTTGTGATGAACGAAATCAACCGGCGACCATGCAGGCGGCCATAGCCTCCGGTGGCATCACGACTGAGCCGTGGTGGA')]

```



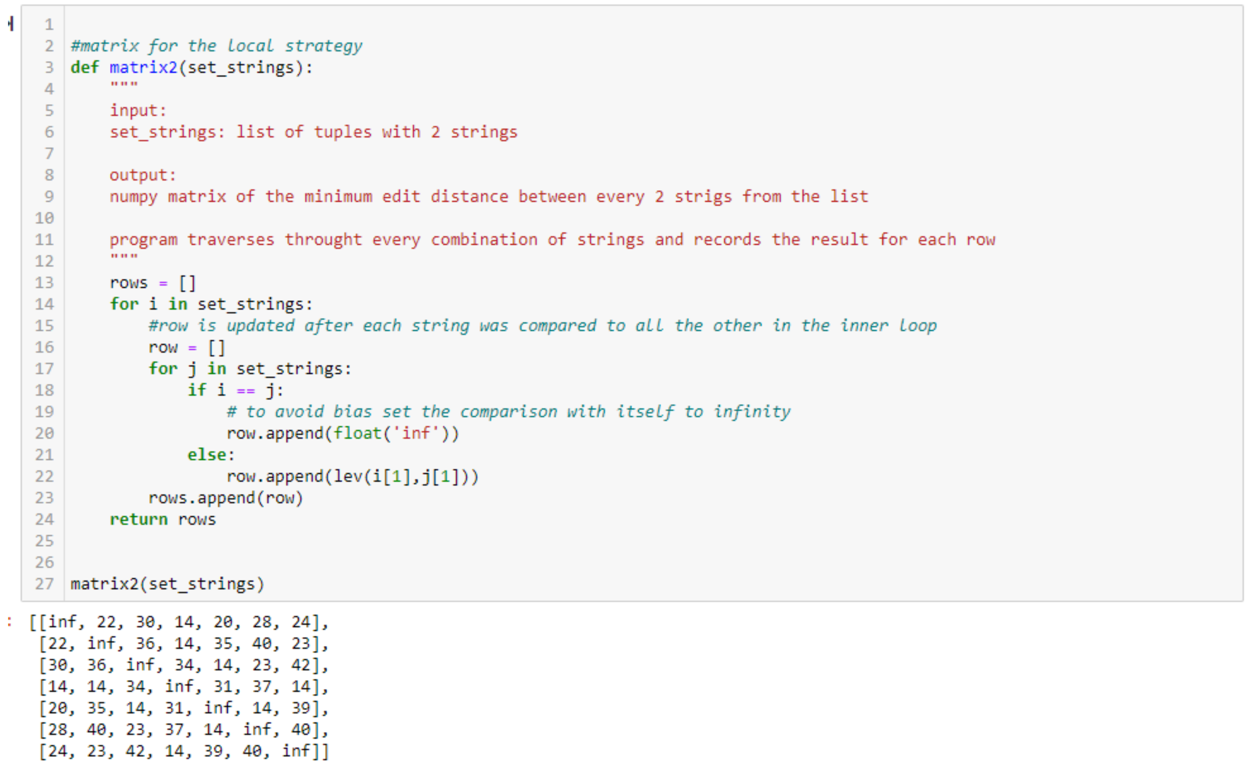
```python
def combinations(set_strings):
    """
    input:
    set_strings: list of tuples with 2 strings

    output:
    string containing:
    max_value, min_value integers,
    min_key[0],min_key[1] max_key[0], max_key[1] strings
    """

    used = []
    result = {}
    for i in set_strings:
        #do not compare strings twice
        used.append(i)
        for j in set_strings:
            #do not compare 2 identical strings
            if i == j:
                continue
            if j not in used:
                length = len(longest_common_subsequence(i[1], j[1])[0])
                result[(i[0],j[0])] = length
    max_value = max(result.values())
    max_key = max(result, key = result.get)
    min_value = min(result.values())
    min_key = min(result, key = result.get)
    output = "The maximum number of combinations of lcs is {} between {} and {}; the minimum number of combinations of l
    return output




```



```python
combinations(set_strings)
```

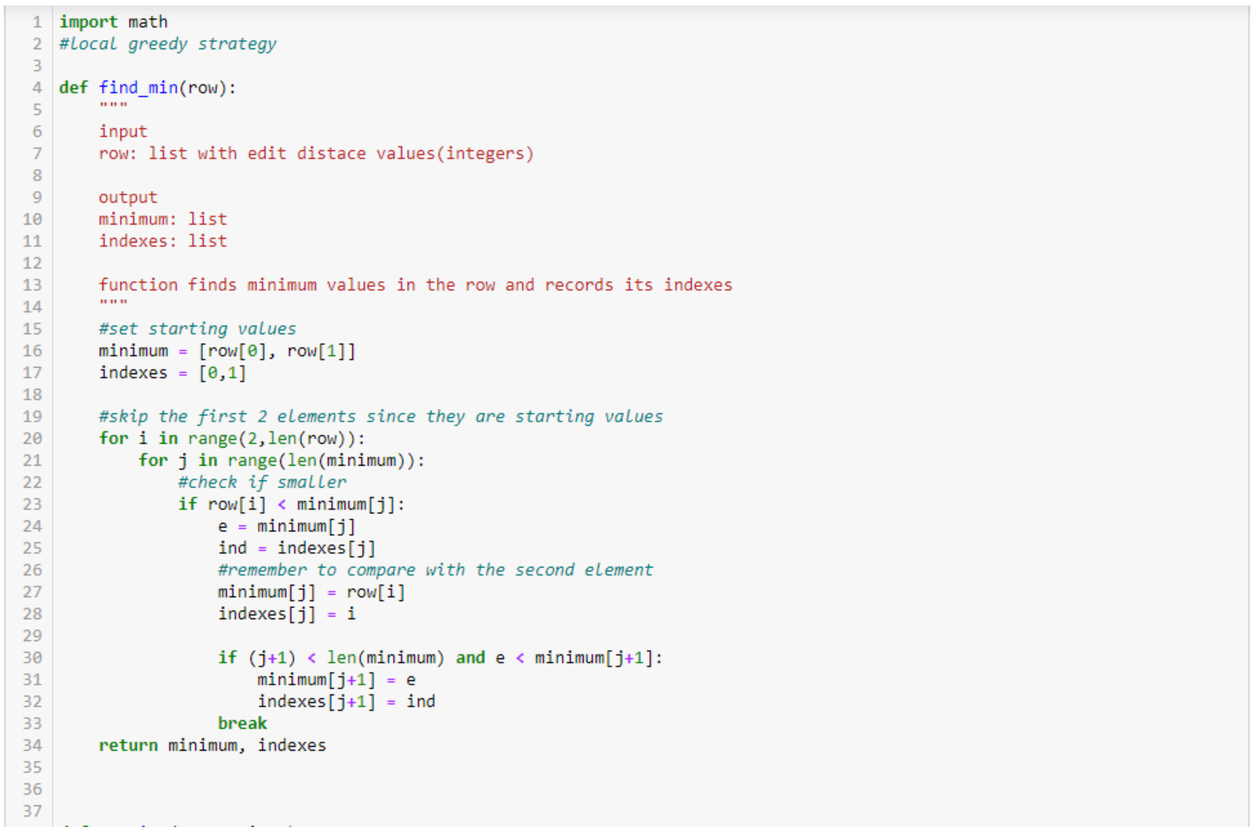


```python
def matrix(set_strings,percentages = False):
    """
    input:
    set_strings: list of tuples with 2 strings
    percentages bool automatically set to "False"

    output:
    numpy matrix of the longest common sequences between every 2 strigs from the set_strings

    program traverses throught every combination of strings and records the result for each row
    there are 2 options for the output: lcs and lcs in percentages.
    """
    rows = []
    maxi = []

    for i in set_strings:
        #row is updated after each string was compared to all the other in the inner loop
        row = []
        for j in set_strings:
            row.append(lcs(i[1],j[1])[-1][-1])
            #set the max value when the stright is compared to itself (100% match)
            if j[0]==i[0]:
                maxi.append(lcs(i[1],j[1])[-1][-1])
        # as the row is filled, add it to the main list
        rows.append(row)
    if percentages:
        for row in rows:
            for i in range(len(row)):
                #calculate the matrix using percentages
                row[i] = int(row[i]/maxi[i]*100)
    return np.matrix(rows)


len_lcs_matrix = matrix(set_strings,0)

print(len_lcs_matrix)
```

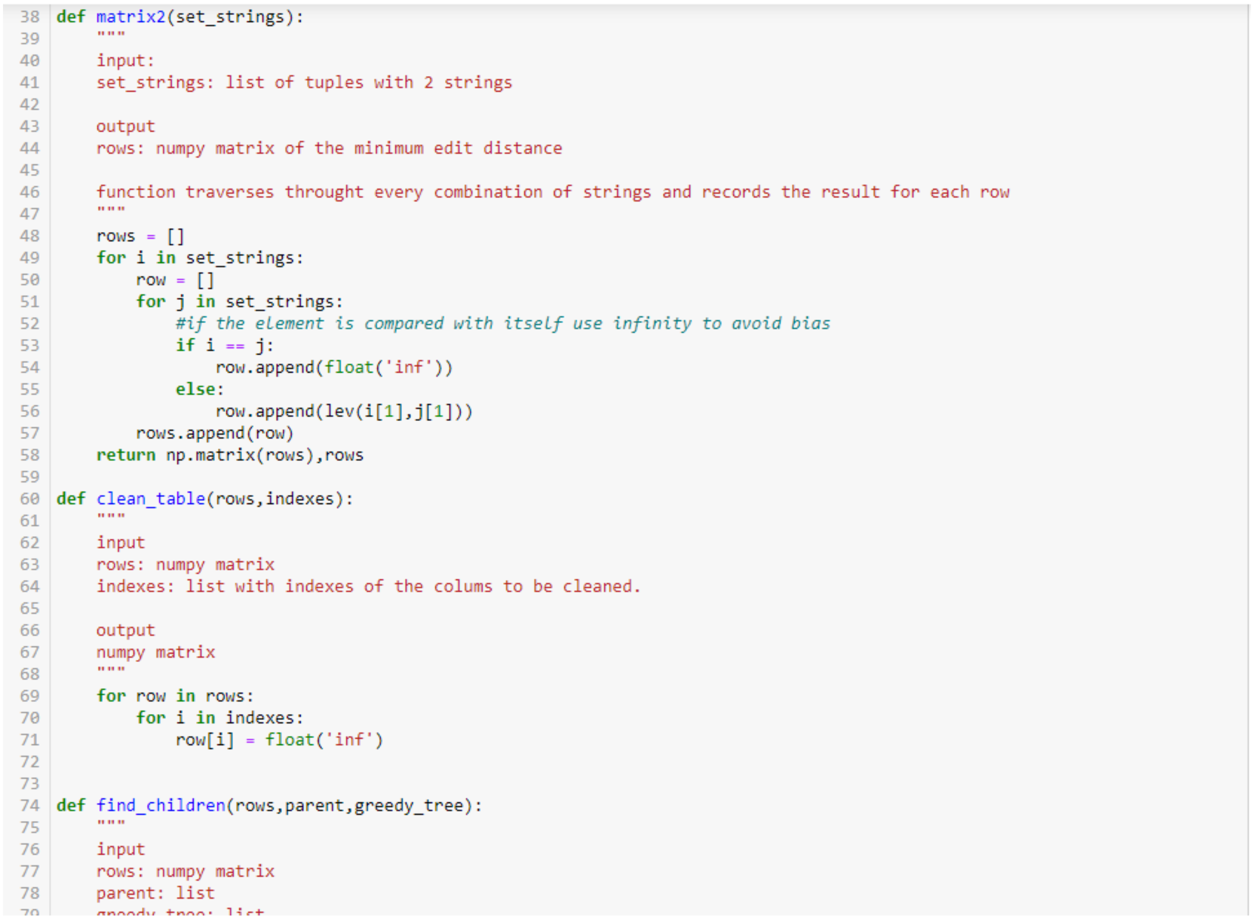


```

#matrix for the local strategy
def matrix2(set_strings):
    """
    input:
    set_strings: list of tuples with 2 strings

    output:
    numpy matrix of the minimum edit distance between every 2 strigs from the list

    program traverses throught every combination of strings and records the result for each row
    """
    rows = []
    for i in set_strings:
        #row is updated after each string was compared to all the other in the inner loop
        row = []
        for j in set_strings:
            if i == j:
                # to avoid bias set the comparison with itself to infinity
                row.append(float('inf'))
            else:
                row.append(lev(i[1],j[1]))
        rows.append(row)
    return rows


matrix2(set_strings)
```

```
: [[inf, 22, 30, 14, 20, 28, 24],
   [22, inf, 36, 14, 35, 40, 23],
   [30, 36, inf, 34, 14, 23, 42],
   [14, 14, 34, inf, 31, 37, 14],
   [20, 35, 14, 31, inf, 14, 39],
   [28, 40, 23, 37, 14, inf, 40],
   [24, 23, 42, 14, 39, 40, inf]]
```



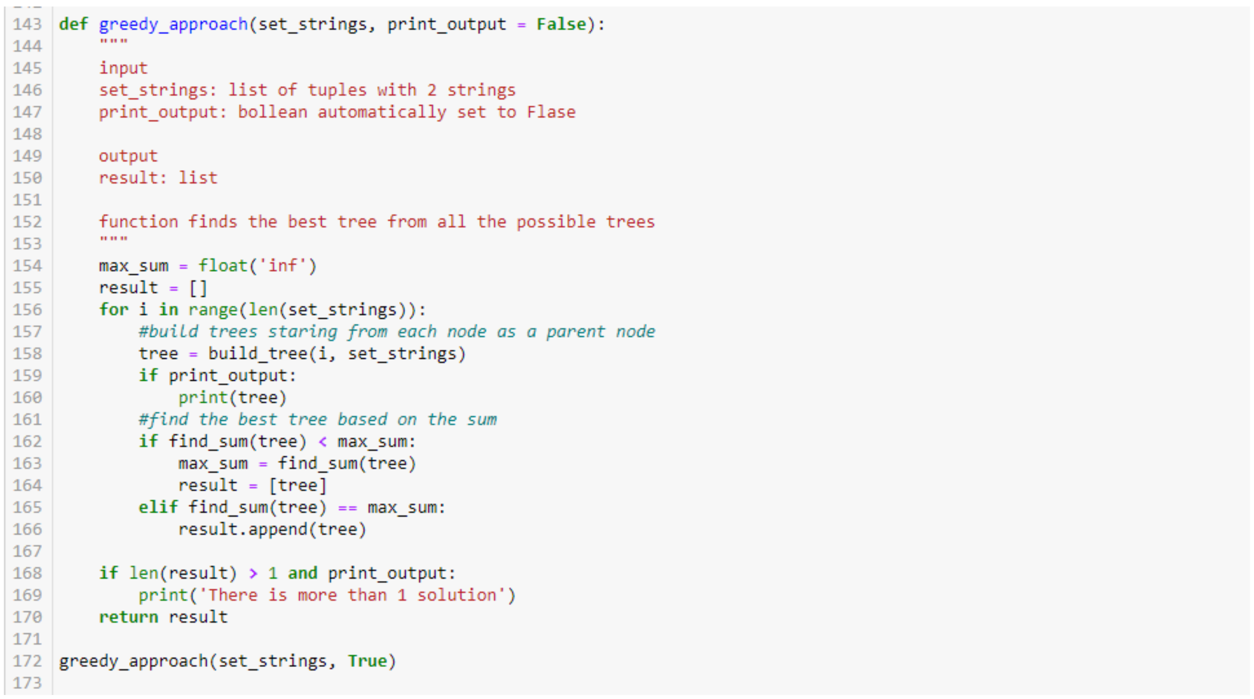
```
import math
#local greedy strategy

def find_min(row):
    """
    input
    row: list with edit distace values(integers)

    output
    minimum: list
    indexes: list

    function finds minimum values in the row and records its indexes
    """
    #set starting values
    minimum = [row[0], row[1]]
    indexes = [0,1]

    #skip the first 2 elements since they are starting values
    for i in range(2,len(row)):
        for j in range(len(minimum)):
            #check if smaller
            if row[i] < minimum[j]:
                e = minimum[j]
                ind = indexes[j]
                #remember to compare with the second element
                minimum[j] = row[i]
                indexes[j] = i

                if (j+1) < len(minimum) and e < minimum[j+1]:
                    minimum[j+1] = e
                    indexes[j+1] = ind
                break
    return minimum, indexes


```



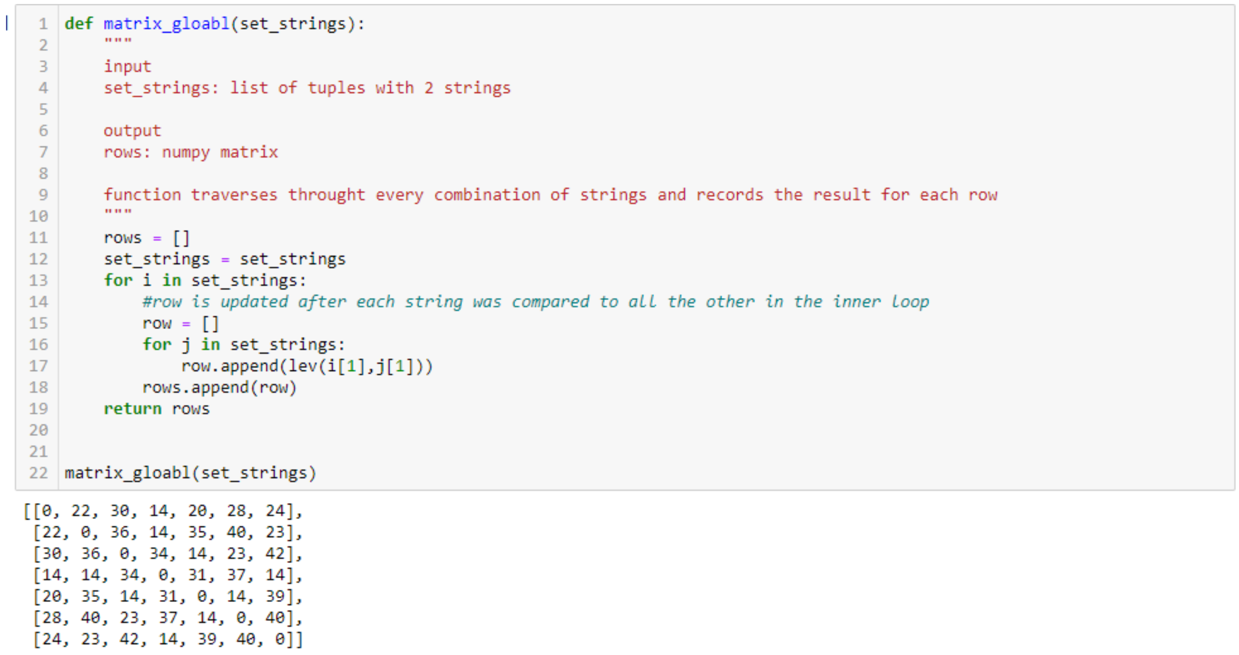
```python
def matrix2(set_strings):
    """
    input:
    set_strings: list of tuples with 2 strings

    output
    rows: numpy matrix of the minimum edit distance

    function traverses throught every combination of strings and records the result for each row
    """
    rows = []
    for i in set_strings:
        row = []
        for j in set_strings:
            #if the element is compared with itself use infinity to avoid bias
            if i == j:
                row.append(float('inf'))
            else:
                row.append(lev(i[1],j[1]))
        rows.append(row)
    return np.matrix(rows),rows

def clean_table(rows,indexes):
    """
    input
    rows: numpy matrix
    indexes: list with indexes of the colums to be cleaned.

    output
    numpy matrix
    """
    for row in rows:
        for i in indexes:
            row[i] = float('inf')


def find_children(rows,parent,greedy_tree):
    """
    input
    rows: numpy matrix
    parent: list
    greedy_tree: list
```

```python
def find_children(rows,parent,greedy_tree):
    """
    input
    rows: numpy matrix
    parent: list
    greedy_tree: list

    output
    children: list
    indexes: list

    function find the children of a prenet based on its proximity
    """
    row = rows[parent]
    #recors the value of the child and its location
    children = find_min(row)[0]
    indexes = find_min(row)[1]

    #termination condition
    if children.count(float('inf')) == 2:
        return

    #remove spare nodes from the tree
    while children.count(float('inf')) > 0:
        children.remove(float('inf'))

    #add children to the tree
    greedy_tree.append(children)
    clean_table(rows,indexes)

    for i in indexes:
        find_children(rows,i,greedy_tree)

    return children,indexes

```

```python
def build_tree(parent, set_strings):
    """
    input
    parent: integer, parent node of the tree
    set_strings: list of tuples with 2 strings

    output
    greedy_tree: list representing a tree

    function builds the tree suing the defined above functinos
    """
    rows = matrix2(set_strings)[1]
    clean_table(rows, [parent])
    greedy_tree=[parent]
    find_children(rows,parent,greedy_tree)
    return greedy_tree


def find_sum(greedy_tree):
    """
    input
    greedy_tree: list representing the tree

    output
    summ: integer

    function calculated the sum of all the minmum edit distace of the tree
    """
    summ = 0
    for i in greedy_tree[1:]:
        summ+= sum(i)
    node = greedy_tree[0]
    return summ

```

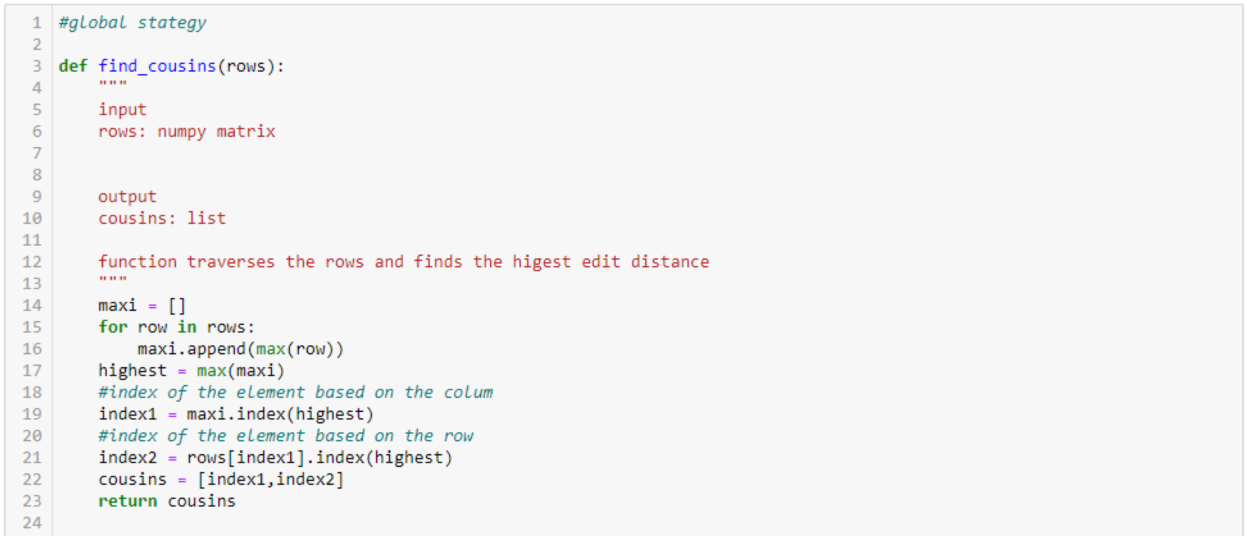


```python
def greedy_approach(set_strings, print_output = False):
    """
    input
    set_strings: list of tuples with 2 strings
    print_output: bollean automatically set to Flase

    output
    result: list

    function finds the best tree from all the possible trees
    """
    max_sum = float('inf')
    result = []
    for i in range(len(set_strings)):
        #build trees staring from each node as a parent node
        tree = build_tree(i, set_strings)
        if print_output:
            print(tree)
        #find the best tree based on the sum
        if find_sum(tree) < max_sum:
            max_sum = find_sum(tree)
            result = [tree]
        elif find_sum(tree) == max_sum:
            result.append(tree)

    if len(result) > 1 and print_output:
        print('There is more than 1 solution')
    return result

greedy_approach(set_strings, True)
```

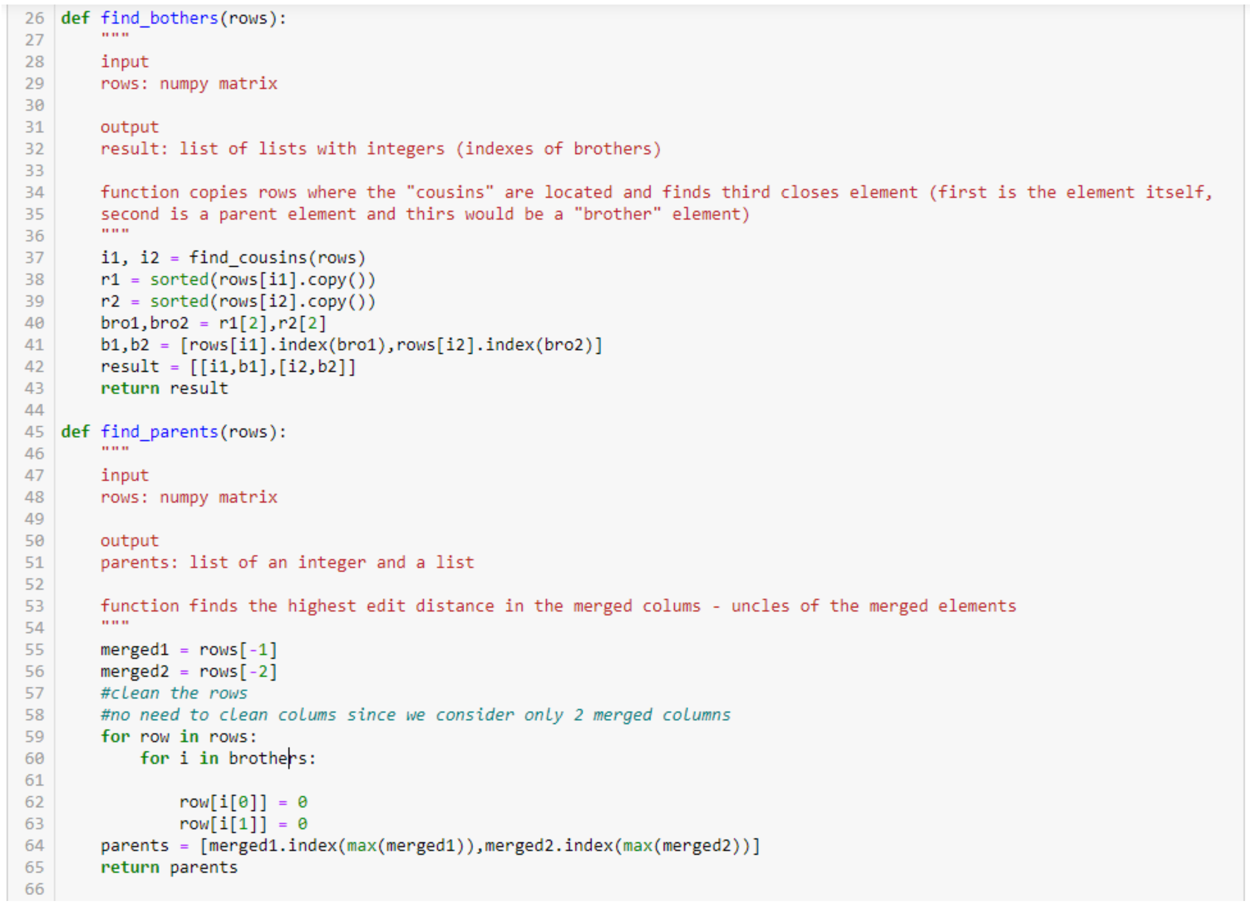


```python
def matrix_gloabl(set_strings):
    """
    input
    set_strings: list of tuples with 2 strings

    output
    rows: numpy matrix

    function traverses throught every combination of strings and records the result for each row
    """
    rows = []
    set_strings = set_strings
    for i in set_strings:
        #row is updated after each string was compared to all the other in the inner loop
        row = []
        for j in set_strings:
            row.append(lev(i[1],j[1]))
        rows.append(row)
    return rows


matrix_gloabl(set_strings)
```

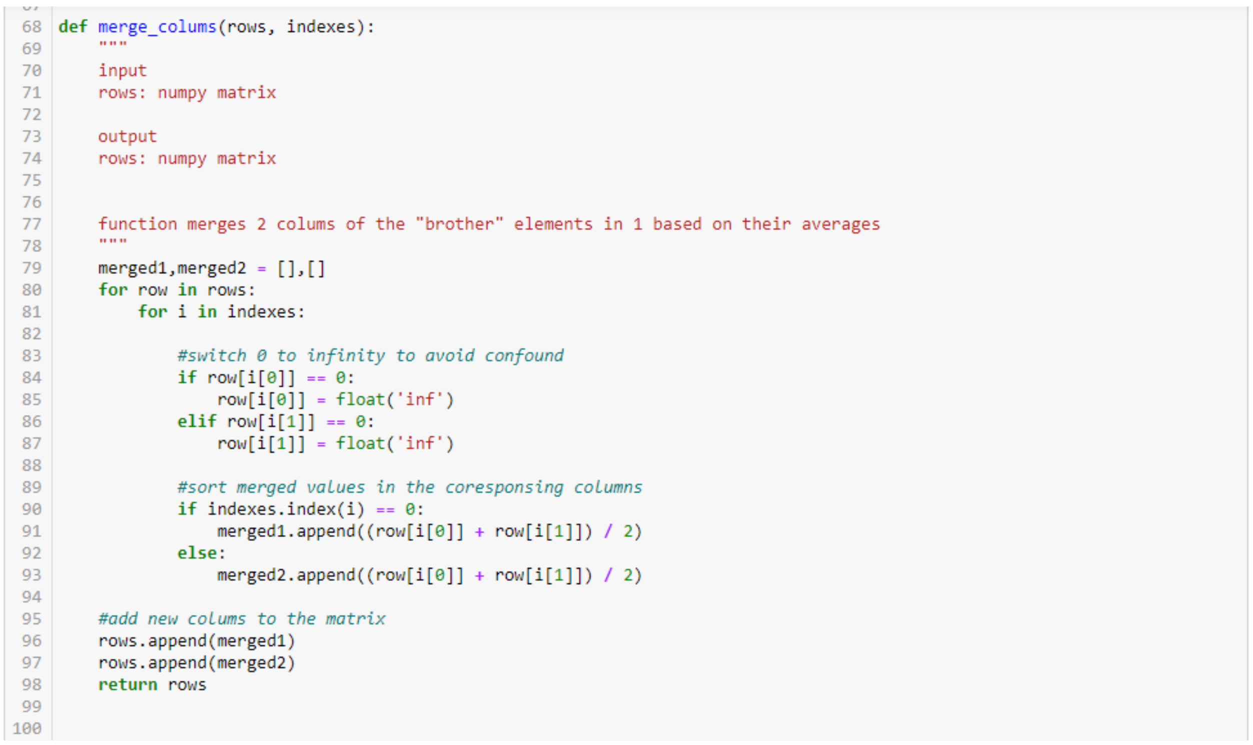
```
[[0, 22, 30, 14, 20, 28, 24],
 [22, 0, 36, 14, 35, 40, 23],
 [30, 36, 0, 34, 14, 23, 42],
 [14, 14, 34, 0, 31, 37, 14],
 [20, 35, 14, 31, 0, 14, 39],
 [28, 40, 23, 37, 14, 0, 40],
 [24, 23, 42, 14, 39, 40, 0]]
```



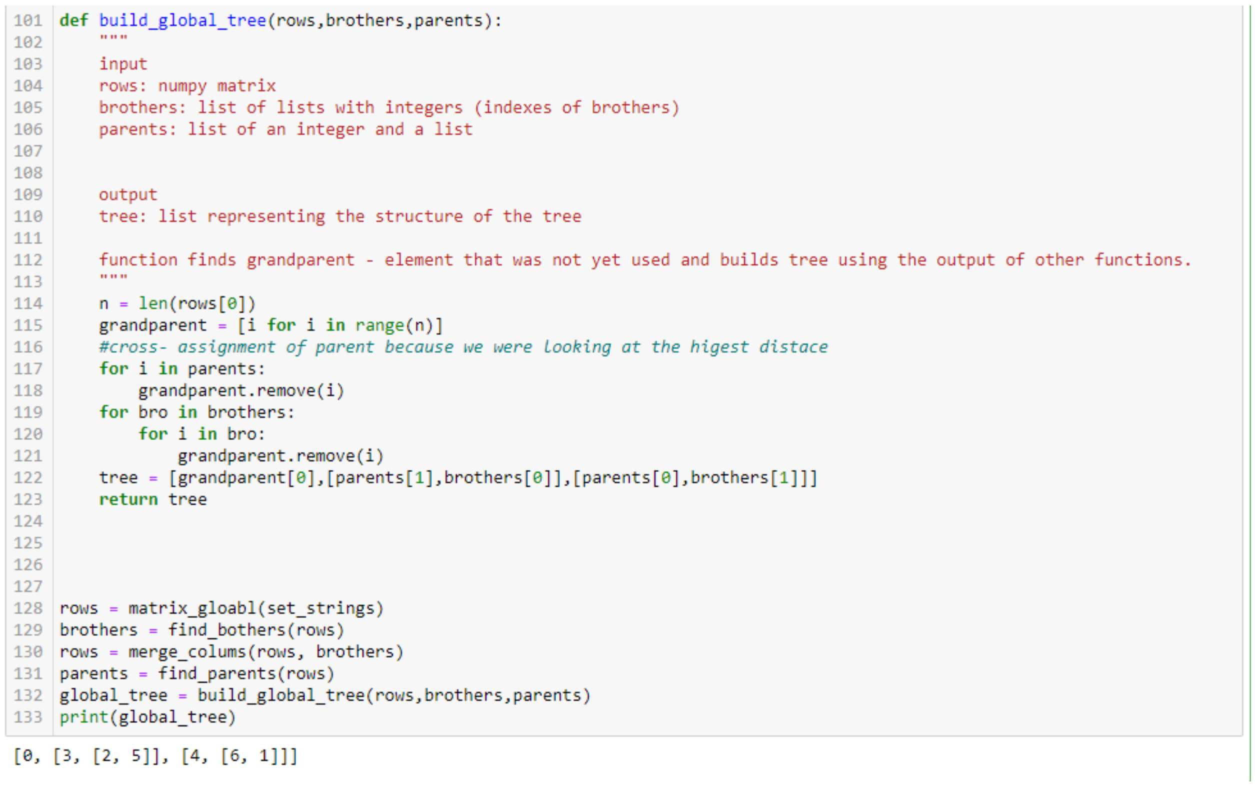
```python
#global stategy

def find_cousins(rows):
    """
    input
    rows: numpy matrix


    output
    cousins: list

    function traverses the rows and finds the higest edit distance
    """
    maxi = []
    for row in rows:
        maxi.append(max(row))
    highest = max(maxi)
    #index of the element based on the colum
    index1 = maxi.index(highest)
    #index of the element based on the row
    index2 = rows[index1].index(highest)
    cousins = [index1,index2]
    return cousins

```



```python
def find_bothers(rows):
    """
    input
    rows: numpy matrix

    output
    result: list of lists with integers (indexes of brothers)

    function copies rows where the "cousins" are located and finds third closes element (first is the element itself,
    second is a parent element and thirs would be a "brother" element)
    """
    i1, i2 = find_cousins(rows)
    r1 = sorted(rows[i1].copy())
    r2 = sorted(rows[i2].copy())
    bro1,bro2 = r1[2],r2[2]
    b1,b2 = [rows[i1].index(bro1),rows[i2].index(bro2)]
    result = [[i1,b1],[i2,b2]]
    return result

def find_parents(rows):
    """
    input
    rows: numpy matrix

    output
    parents: list of an integer and a list

    function finds the highest edit distance in the merged colums - uncles of the merged elements
    """
    merged1 = rows[-1]
    merged2 = rows[-2]
    #clean the rows
    #no need to clean colums since we consider only 2 merged columns
    for row in rows:
        for i in brothers:

            row[i[0]] = 0
            row[i[1]] = 0
    parents = [merged1.index(max(merged1)),merged2.index(max(merged2))]
    return parents

```

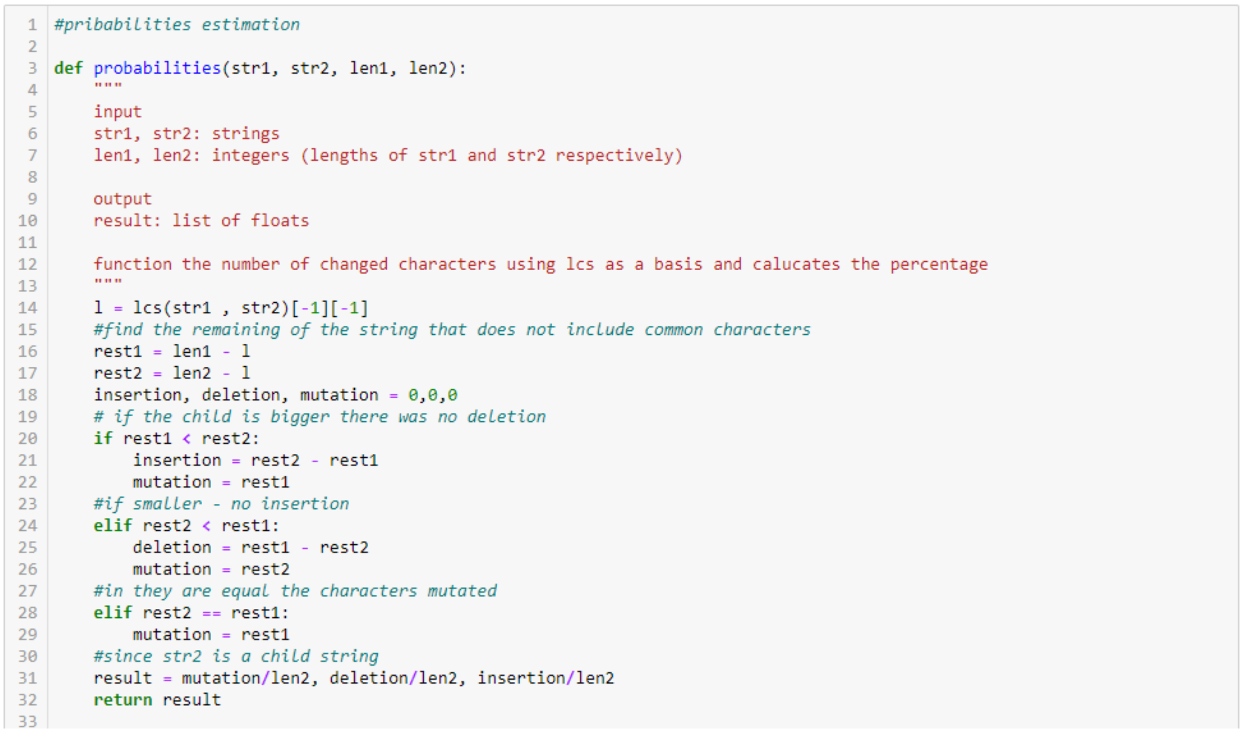


```python
def merge_colums(rows, indexes):
    """
    input
    rows: numpy matrix

    output
    rows: numpy matrix


    function merges 2 colums of the "brother" elements in 1 based on their averages
    """
    merged1,merged2 = [],[]
    for row in rows:
        for i in indexes:

            #switch 0 to infinity to avoid confound
            if row[i[0]] == 0:
                row[i[0]] = float('inf')
            elif row[i[1]] == 0:
                row[i[1]] = float('inf')

            #sort merged values in the coresponsing columns
            if indexes.index(i) == 0:
                merged1.append((row[i[0]] + row[i[1]]) / 2)
            else:
                merged2.append((row[i[0]] + row[i[1]]) / 2)

    #add new colums to the matrix
    rows.append(merged1)
    rows.append(merged2)
    return rows
```

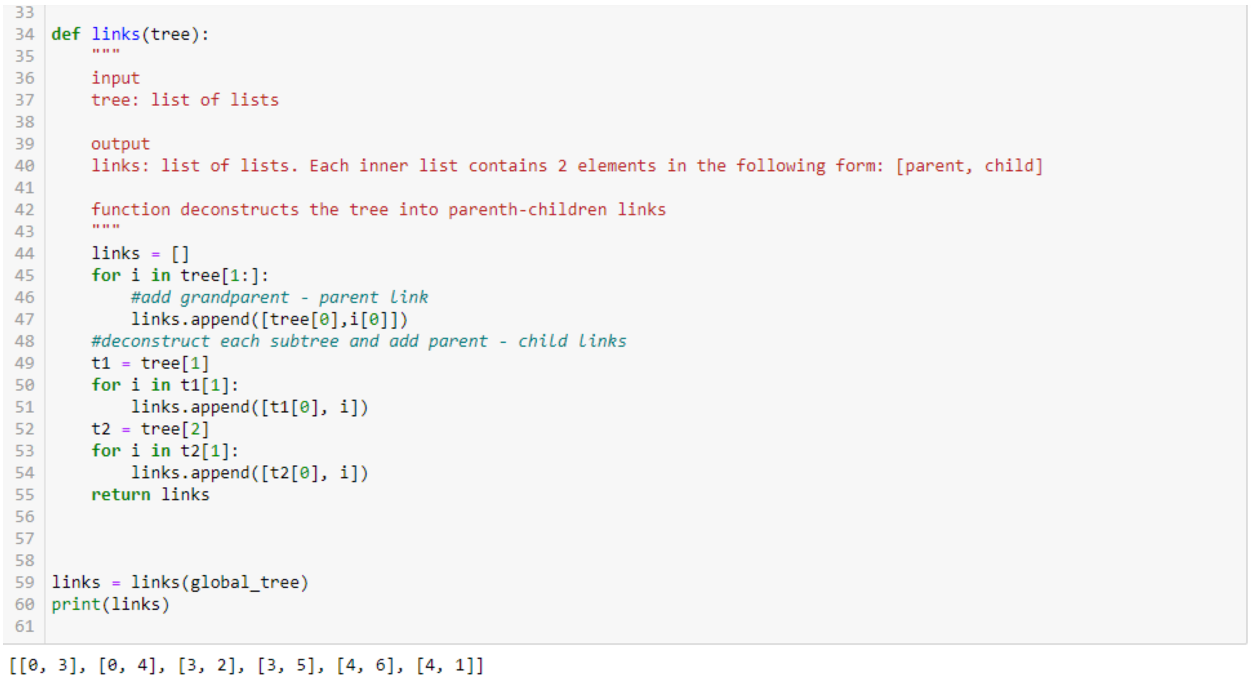


```python
def build_global_tree(rows,brothers,parents):
    """
    input
    rows: numpy matrix
    brothers: list of lists with integers (indexes of brothers)
    parents: list of an integer and a list


    output
    tree: list representing the structure of the tree

    function finds grandparent - element that was not yet used and builds tree using the output of other functions.
    """
    n = len(rows[0])
    grandparent = [i for i in range(n)]
    #cross- assignment of parent because we were looking at the higest distace
    for i in parents:
        grandparent.remove(i)
    for bro in brothers:
        for i in bro:
            grandparent.remove(i)
    tree = [grandparent[0],[parents[1],brothers[0]],[parents[0],brothers[1]]]
    return tree




rows = matrix_gloabl(set_strings)
brothers = find_bothers(rows)
rows = merge_colums(rows, brothers)
parents = find_parents(rows)
global_tree = build_global_tree(rows,brothers,parents)
print(global_tree)
```

```
[0, [3, [2, 5]], [4, [6, 1]]]
```

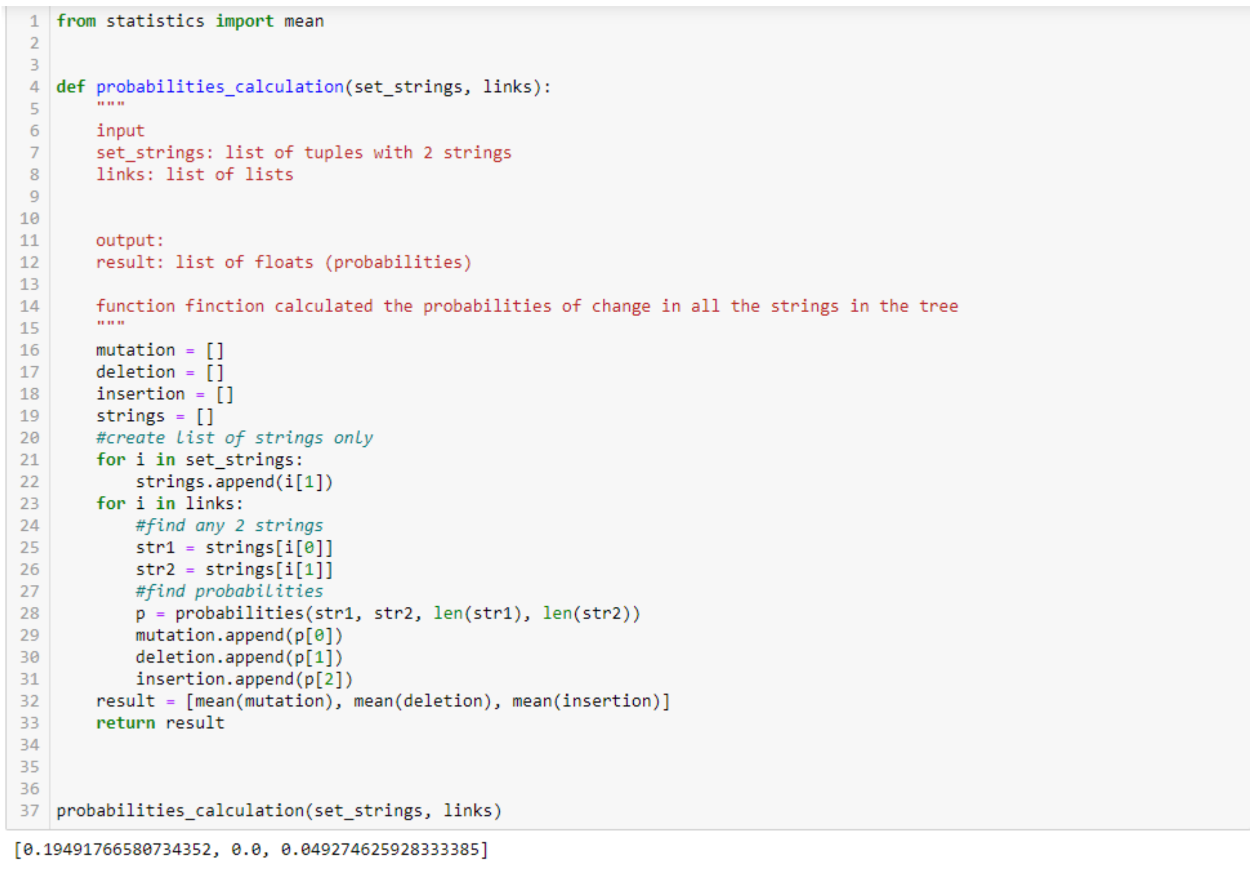


```python
#pribabilities estimation

def probabilities(str1, str2, len1, len2):
    """
    input
    str1, str2: strings
    len1, len2: integers (lengths of str1 and str2 respectively)

    output
    result: list of floats

    function the number of changed characters using lcs as a basis and calucates the percentage
    """
    l = lcs(str1 , str2)[-1][-1]
    #find the remaining of the string that does not include common characters
    rest1 = len1 - l
    rest2 = len2 - l
    insertion, deletion, mutation = 0,0,0
    # if the child is bigger there was no deletion
    if rest1 < rest2:
        insertion = rest2 - rest1
        mutation = rest1
    #if smaller - no insertion
    elif rest2 < rest1:
        deletion = rest1 - rest2
        mutation = rest2
    #in they are equal the characters mutated
    elif rest2 == rest1:
        mutation = rest1
    #since str2 is a child string
    result = mutation/len2, deletion/len2, insertion/len2
    return result

```



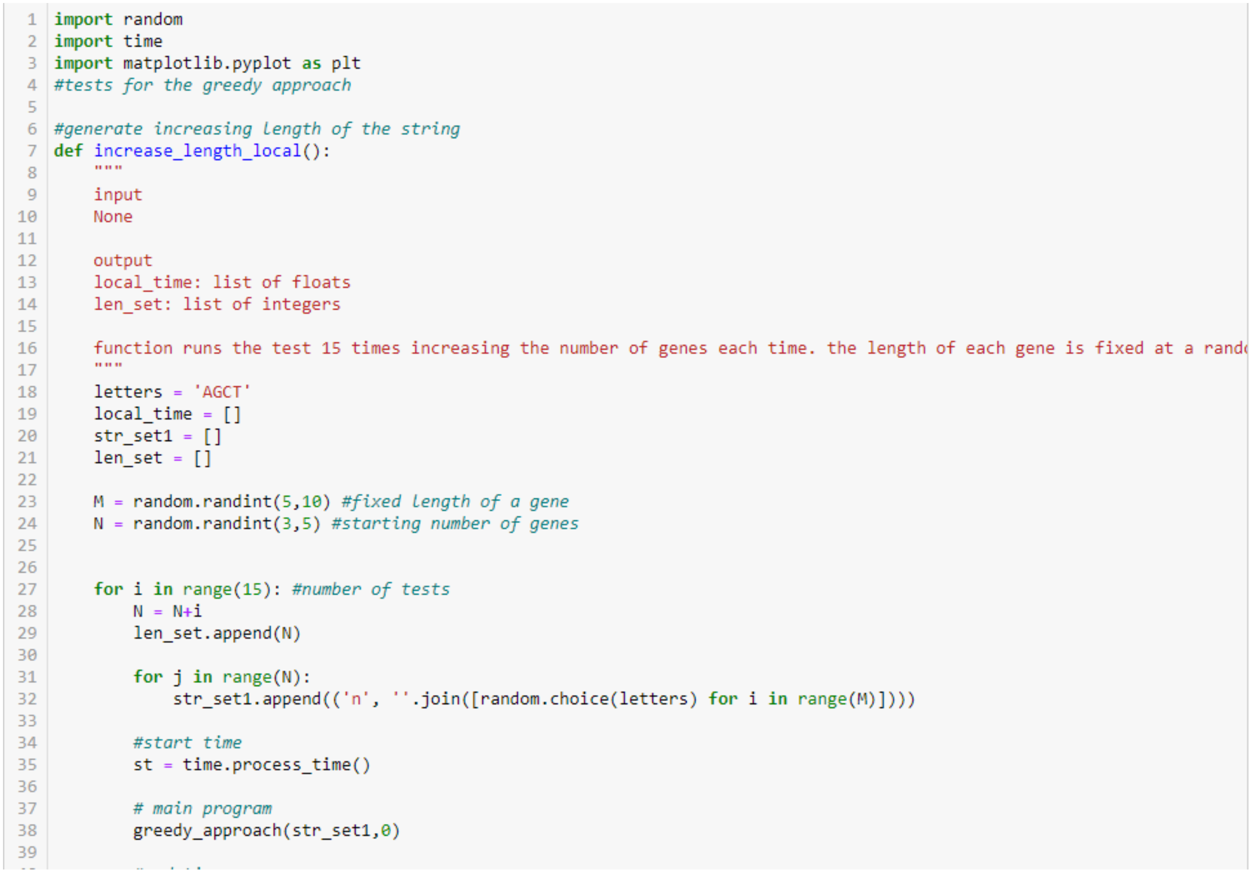
```python

def links(tree):
    """
    input
    tree: list of lists

    output
    links: list of lists. Each inner list contains 2 elements in the following form: [parent, child]

    function deconstructs the tree into parenth-children links
    """
    links = []
    for i in tree[1:]:
        #add grandparent - parent link
        links.append([tree[0],i[0]])
    #deconstruct each subtree and add parent - child links
    t1 = tree[1]
    for i in t1[1]:
        links.append([t1[0], i])
    t2 = tree[2]
    for i in t2[1]:
        links.append([t2[0], i])
    return links



links = links(global_tree)
print(links)

```

```
[[0, 3], [0, 4], [3, 2], [3, 5], [4, 6], [4, 1]]
```

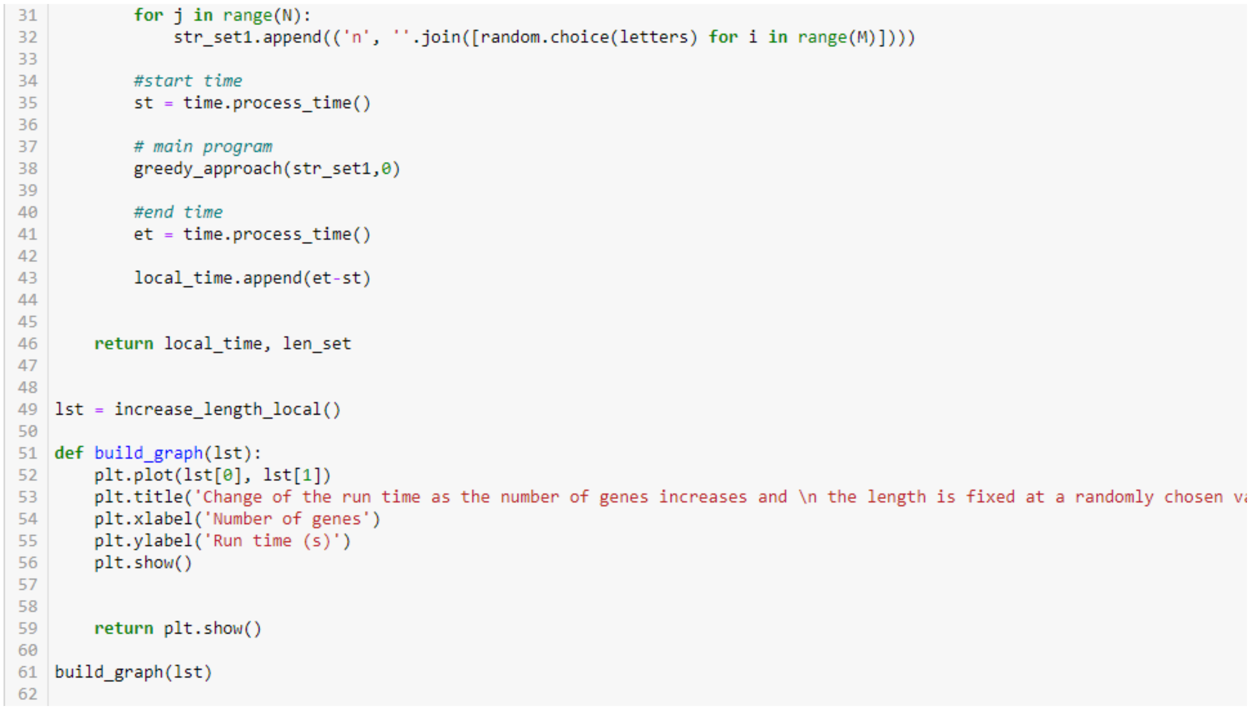


```python
from statistics import mean


def probabilities_calculation(set_strings, links):
    """
    input
    set_strings: list of tuples with 2 strings
    links: list of lists


    output:
    result: list of floats (probabilities)

    function finction calculated the probabilities of change in all the strings in the tree
    """
    mutation = []
    deletion = []
    insertion = []
    strings = []
    #create list of strings only
    for i in set_strings:
        strings.append(i[1])
    for i in links:
        #find any 2 strings
        str1 = strings[i[0]]
        str2 = strings[i[1]]
        #find probabilities
        p = probabilities(str1, str2, len(str1), len(str2))
        mutation.append(p[0])
        deletion.append(p[1])
        insertion.append(p[2])
    result = [mean(mutation), mean(deletion), mean(insertion)]
    return result



probabilities_calculation(set_strings, links)
```

```
[0.19491766580734352, 0.0, 0.049274625928333385]
```

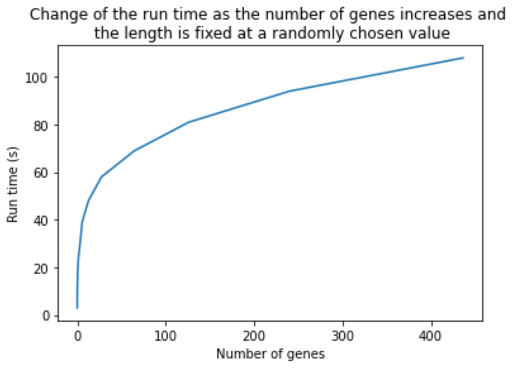


```
import random
import time
import matplotlib.pyplot as plt
#tests for the greedy approach

#generate increasing length of the string
def increase_length_local():
    """
    input
    None

    output
    local_time: list of floats
    len_set: list of integers

    function runs the test 15 times increasing the number of genes each time. the length of each gene is fixed at a rand
    """
    letters = 'AGCT'
    local_time = []
    str_set1 = []
    len_set = []

    M = random.randint(5,10) #fixed length of a gene
    N = random.randint(3,5) #starting number of genes


    for i in range(15): #number of tests
        N = N+i
        len_set.append(N)

        for j in range(N):
            str_set1.append(('n', ''.join([random.choice(letters) for i in range(M)])))

        #start time
        st = time.process_time()

        # main program
        greedy_approach(str_set1,0)

```



```
        for j in range(N):
            str_set1.append(('n', ''.join([random.choice(letters) for i in range(M)])))

        #start time
        st = time.process_time()

        # main program
        greedy_approach(str_set1,0)

```

```
        #end time
        et = time.process_time()

        local_time.append(et-st)


    return local_time, len_set


lst = increase_length_local()

def build_graph(lst):
    plt.plot(lst[0], lst[1])
    plt.title('Change of the run time as the number of genes increases and \n the length is fixed at a randomly chosen v
    plt.xlabel('Number of genes')
    plt.ylabel('Run time (s)')
    plt.show()


    return plt.show()

build_graph(lst)

build_graph(lst)
```



References

Algorithm Implementation/Strings/Longest common subsequence - Wikibooks, open books for an open world. (n.d.). https://en.wikibooks.org/wiki/Algorithm_Implementation/Strings/Longest_common_subsequence

Back To Back SWE. (2019, January 11). *Edit Distance Between 2 Strings - The Levenshtein Distance ("Edit Distance" on LeetCode)*. YouTube. https://www.youtube.com/watch?v=MiqoA-yF-0M

Levenshtein distance with substitution, deletion and insertion count. (2020, May 13). Stack Overflow. https://stackoverflow.com/questions/61784300/levenshtein-distance-with-substitution-deletion-and-insertion-count

GeeksforGeeks. (2022, December 10). *Edit Distance | DP-5*. https://www.geeksforgeeks.org/edit-distance-dp-5/

GeeksforGeeks. (2022a, May 24). *sort() in Python*. https://www.geeksforgeeks.org/sort-in-python/

GeeksforGeeks. (2022b, December 14). *Print all longest common sub-sequences in lexicographical order*. https://www.geeksforgeeks.org/print-longest-common-sub-sequences-lexicographical-order/

GeeksforGeeks. (2022b, December 10). *Edit Distance | DP-5*. https://www.geeksforgeeks.org/edit-distance-dp-5/