Assignment 3

Genetic heritage:

The following 7 strings were generated by taking an existing string and with a small probability either inserting a new character, deleting an existing character, or changing to a new character randomly. This created two child strings, each of these strings with a single character changed from their parent. And from the two child strings, four grandchild strings were created, two from each child. The 4 grandchild strings also have a single character changed from their parents. As result, we got 7 strings but unfortunately the order of the strings has been lost.

```
(0,
'CAGCGGGTGCGTAATTTGGAGAAGTTATTCTGCAACGAAATCAATCCTGTTTCGTTAGCTTACGGACTACGACGAGAGGGTACTTCCCTGATATAGTCAC')
(1, 'CAAGTCGGGCGTATTGGAGAATATTTAAATCGGAAGATCATGTTACTATGCGTTAGCTCACGGACTGAAGAGGATTCTCTCTTAATGCAA')
(2, 'CATGGGTGCGTCGATTTTGGCAGTAAAGTGGAATCGTCAGATATCAATCCTGTTTCGTAGAAAGGAGCTACCTAGAGAGGATTACTCTCACATAGTA')
(3, 'CAAGTCCGCGATAAATTGGAATATTTGTCAATCGGAATAGTCAACTTAGCTGGCGTTAGCTTTACGACTGACAGAGAGAAACCTGTCCATCACACA')
(4, 'CAAGTCCGGCGTAATTGGAGAATATTTTGCAATCGGAAGATCAATCTTGTTAGCGTTAGCTTACGACTGACGAGAGGGATACTCTCTCTAATACAA')
(5,
'CACGGGCTCCGCAATTTTGGGTCAAGTTGCATATCAGTCATCGACAATCAAACACTGTTTTGCGGTAGATAAGATACGACTGAGAGAGGACGTTCGCTCGA
(6,
'CACGGGTCCGTCAATTTTGGAGTAAGTTGATATCGTCACGAAATCAATCCTGTTTCGGTAGTATAGGACTACGACGAGAGAGGACGTTCCTCTGATATAGTTA
```

Task 1

Write python code to give the length of the longest common subsequence for two strings.

In [8]:
```python
def lcs(str_1, str_2):
    m = len(str_1)
    n = len(str_2)
    lcs_len = []

    for i in range(m):
        lcs_len.append([])
        for j in range(n):
            lcs_len[i].append(0)

    for i in range(1, m):
        for j in range(1, n):
            if str_1[i] == str_2[j]:
                lcs_len[i][j] = lcs_len[i - 1][j - 1] + 1 # subproblem's LCS + 1
            elif lcs_len[i - 1][j] > lcs_len[i][j - 1]:
                lcs_len[i][j] = lcs_len[i - 1][j] # subproblem's LCS
            else:
                lcs_len[i][j] = lcs_len[i][j - 1] # subproblem's LCS

    return lcs_len[-1][-1] + 1

str_1 = "CAGCGGGTGCGTAATTTGGAGAAGTTATTCTGCAACGAAATCAATCCTGTTTCGTTAGCTTACGGACTACGACGAGAGGGTACTTCCCTGATATAGTCA
str_2 = "CAAGTCCGGCGTAATTGGAGAATATTTTGCAATCGGAAGATCAATCTTGTTAGCGTTAGCTTACGACTGACGAGAGGGATACTCTCTCTAATACAA"
lcs(str_1, str_2)
lcs("abfeb", "abcfeabee")
```

Out[8]:
```
5
```

Task 2

Generate the table of the lengths of the longest common subsequences for every pair of strings.

In [83]:

```python
seqs = [
    (0, 'CAGCGGGTGCGTAATTTGGAGAAGTTATTCTGCAACGAAATCAATCCTGTTTCGTTAGCTTACGGACTACGACGAGAGGGTACTTCCCTGATATAGTCA
    (1, 'CAAGTCGGGCGTATTGGAGAATATTTAAATCGGAAGATCATGTTACTATGCGTTAGCTCACGGACTGAAGAGGATTCTCTCTTAATGCAA'),
    (2, 'CATGGGTGCGTCGATTTTGGCAGTAAAGTGGAATCGTCAGATATCAATCCTGTTTCGTAGAAAGGAGCTACCTAGAGAGGATTACTCTCACATAGTA')
    (3, 'CAAGTCCGCGATAAATTGGAATATTTGTCAATCGGAATAGTCAACTTAGCTGGCGTTAGCTTTACGACTGACAGAGAGAAACCTGTCCATCACACA'),
    (4, 'CAAGTCCGGCGTAATTGGAGAATATTTTGCAATCGGAAGATCAATCTTGTTAGCGTTAGCTTACGACTGACGAGAGGGATACTCTCTCTAATACAA'),
    (5, 'CACGGGCTCCGCAATTTTGGGTCAAGTTGCATATCAGTCATCGACAATCAAACACTGTTTTGCGGTAGATAAGATACGACTGAGAGAGGACGTTCGCTC
    (6, 'CACGGGTCCGTCAATTTTGGAGTAAGTTGATATCGTCACGAAATCAATCCTGTTTCGGTAGTATAGGACTACGACGAGAGAGGACGTTCCTCTGATATA
]

seq_ct = len(seqs)
dists = []

for i in range(seq_ct):
    dists.append([])
    for j in range(seq_ct):
        lcs_len = lcs(seqs[i][1], seqs[j][1])
        dists[i].append(lcs_len)

print(dists)
```

```
[[100, 74, 76, 73, 82, 84, 91], [74, 90, 67, 72, 80, 70, 71], [76, 67, 97, 65, 69, 81, 84], [73, 72, 65, 96,
81, 71, 69], [82, 80, 69, 81, 96, 74, 75], [84, 70, 81, 71, 74, 111, 97], [91, 71, 84, 69, 75, 97, 104]]
```

In [84]:

```
import pandas as pd

dists_dct = {}
for i in range(len(dists)):
    dists_dct[i] = dists[i]

df = pd.DataFrame(dists_dct)
df
```

Out[84]:

	0	1	2	3	4	5	6
0	100	74	76	73	82	84	91
1	74	90	67	72	80	70	71
2	76	67	97	65	69	81	84
3	73	72	65	96	81	71	69
4	82	80	69	81	96	74	75
5	84	70	81	71	74	111	97
6	91	71	84	69	75	97	104

Fig. 1-1: Table of lengths of common subsequences (LCS) of all sequence pairs. Main diagonal (up left to down right) contains the lengths of original sequences, since they are LCS of the same string.

Seeing numbers might be irrelevant, so instead we could take a look at what fraction of bases is retained in all pairs of sequences (by dividing by the actual sequence length).

```
In [85]:
frac_dct = {}
for i in range(len(dists)):
    frac_dct[i] = list(map(lambda elem: elem / max(dists[i]), dists[i]))

df = pd.DataFrame(frac_dct)
df
```

```
Out[85]:
```

	0	1	2	3	4	5	6
0	1.00	0.822222	0.783505	0.760417	0.854167	0.756757	0.875000
1	0.74	1.000000	0.690722	0.750000	0.833333	0.630631	0.682692
2	0.76	0.744444	1.000000	0.677083	0.718750	0.729730	0.807692
3	0.73	0.800000	0.670103	1.000000	0.843750	0.639640	0.663462
4	0.82	0.888889	0.711340	0.843750	1.000000	0.666667	0.721154
5	0.84	0.777778	0.835052	0.739583	0.770833	1.000000	0.932692
6	0.91	0.788889	0.865979	0.718750	0.781250	0.873874	1.000000

Fig. 1-2: Table of fractions of retained bases compared to the gene number in the column. Note: this table is non-symmetrical.

To determine which sequences are related, we will scan through column-wise and see which two (one parent and one sister; one parent and one grandparent; two children) have the highest retainment rate compared to the column number sequence. We then will "untangle" these relationships into a binary tree.

Task 3

Manually examine the table, and infer the relationships between strings.

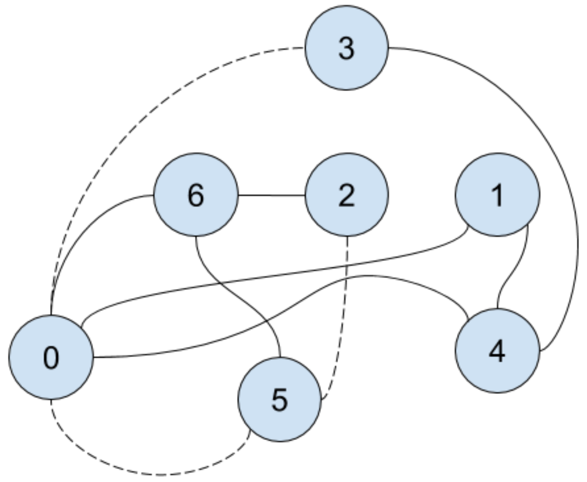

Fig. 2-1: Relationships between sequences. Solid lines indicate the relationship with the highest base-retainment rate (strong relationship). Dotted line shows weaker connections (2nd or 3rd highest base-retainment).

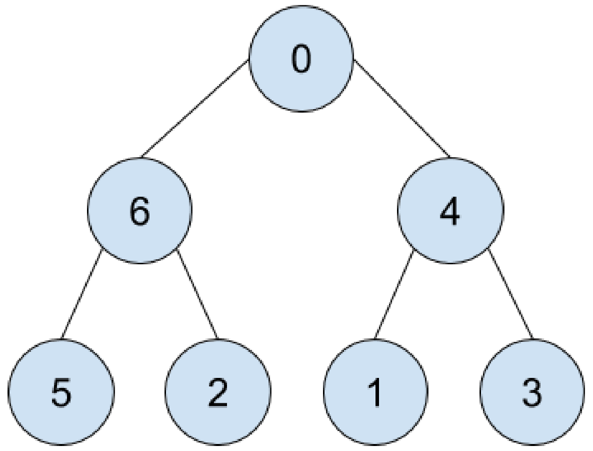

Fig. 2-2: Tree built on the relationships above, putting strongest relationships in a parent-child relationship in a tree and 2nd strongest as either parent-child or 2 generations apart or sisters.

Task 4

How would you estimate the probabilities of mutation, insertions and deletions? (There might not be enough data to give meaningful estimates, but at least have a clear idea of the approach.)

To estimate the probabilities of these operations, we can guess the set and order of the operations that happened to tranform one sequence into another. We will assume that nature will choose the shortest path to modify the sequences and not have infinite operation loops (e.g. "abc" → *insertion at 2* → "abbc" → *mutation at 1* → "aabc" → *deletion at 0* → "abc"). Then, using the Wagner–Fischer algorithm, we can find what operations were involved in a sequence transformation.

Having gathered the amounts of each operation and the length of sequences, we can then approximate the probabilities by finding the mean modification percentage over different lengths and sequences. The more data we have, the closer the estimated probability will align to the actual probability of an operation because of the regression to the mean property of probability distributions of these operations.

In [86]:

```python
def edit_dist(str_1, str_2):
    s_1 = " " + str_1
    s_2 = " " + str_2
    m = len(s_1)
    n = len(s_2)
    dists = [] # min operations to get from suffix str_1[:i] to str_2[:j], init to 0

    # init
    for i in range(m):
        if i == 0:
            dists.append(list(range(n)))
        else:
            dists.append([i] + list(map(lambda x: 0, range(1, n))))

    for i in range(1, m):
        for j in range(1, n):
            if s_1[i] == s_2[j]:
                dists[i][j] = dists[i - 1][j - 1]
            else:
                dists[i][j] = min(
                    dists[i - 1][j - 1], # mutation
                    dists[i - 1][j], # deletion
                    dists[i][j - 1] # insertion
                ) + 1

    return dists

def backtrack_ops(dists):
    mut_ct = del_ct = ins_ct = 0

    i, j = len(dists) - 1, len(dists[0]) - 1
    while i > 0 or j > 0:
        cur_min = dists[i][j] # to store best next step
        opt_coor = (i, j)

        if i - 1 >= 0 and j - 1 >= 0 and cur_min >= dists[i - 1][j - 1]:
            cur_min = dists[i - 1][j - 1]
            opt_coor = (i - 1, j - 1)
            if dists[i - 1][j - 1] != dists[i][j]:
                mut_ct += 1
        else:
            if i - 1 >= 0 and cur_min >= dists[i - 1][j]:
                cur_min = dists[i - 1][j]
                opt_coor = (i - 1, j)
                del_ct += 1
            if j - 1 >= 0 and cur_min >= dists[i][j - 1]:
                cur_min = dists[i][j - 1]
                opt_coor = (i, j - 1)
                ins_ct += 1
        (i, j) = opt_coor

    return mut_ct, del_ct, ins_ct


dists = edit_dist("aaaba", "abaa")
ops = backtrack_ops(dists)
print("Mutations:", ops[0], "Deletions:", ops[1], "Insertions:", ops[2])
```

```
Mutations: 1 Deletions: 1 Insertions: 0
```

In [96]:
```
P_mut = P_del = P_ins = 0

for i in range(seq_ct):
    for j in range(seq_ct):
        dists = edit_dist(seqs[i][1], seqs[j][1])
        ops = backtrack_ops(dists)

        factor = float(len(seqs[i][1])) * seq_ct**2
        P_mut += ops[0] / factor
        P_del += ops[1] / factor
        P_ins += ops[2] / factor

print("Mutation probability:", str(P_mut*100)[:5] + "%",
      "\nDeletion probability:", str(P_del*100)[:4] + "%",
      "\nInsertion probability:", str(P_ins*100)[:4] + "%")
```

```
Mutation probability: 23.78%
Deletion probability: 3.20%
Insertion probability: 3.59%
```

Task 5

Can you devise an algorithm in the general case which might be able to infer such a tree of relationships? Give any strengths or weaknesses of your suggested algorithm.

To do so, we cant calculate the number of mutations, deletions, and insertions for each pair for sequences and check if the percentages of those operations compared the original sequence are almost equal to the probabilities above. Then, these sequences would likely be of 1st order relationship. If the percentages happen to be in the order of squares of the probabilities above, then the sequences most likely would be of 2nd order relationship (and so forth, for all pairs).

Then, assuming that one sequence can only have one parent (e.g. in asexual reproduction), we can build a tree by looking at which node has highest *connectivity* $\sum_{i=1}^{n} \frac{count_{ord}}{ord}$, where n is the number of sequences, ord is the order of relationship (1st being the closest) and $count_{ord}$ is the number of relationships of order ord.

The universal ancestor would have the highest connectivity because ord is at most the height of the tree. For all other nodes, $height(tree) \leq max(ord(node_i, node_j), j = 1 \rightarrow n) \leq 2height(tree)$. Then, the summation terms would be smaller.

Next step is to sort the nodes in an array based on their connectivity in the descending order. Lastly, using the greedy approach, we keep track of the order of the current node compared to the first node in the array, and whenever the order increases, we will evaluate the node's ***edit_dist*** with the nodes in the previous layers, and whichever is minimum, that one would be the parent of the current node. This cross-comparison will take $O((n)^2)$ time complexity as we go through the whole list, and for each element we compare $O(n)$ times to find its parent.

For large amounts of data, this algorithm would be highly inefficient as it scales quadratically with the number of sequences.

Task 6

Describe the complexity of your solution to identify related “genes” for this assignment. (Let M be the length of a gene, and N be the number of genes.)

My solution is of $O(M^2N^2)$ time complexity because we scan through each pair of suffixes (M^2) of the each pair of words (N^2) and do a constant amount of comparisons at each step. Then, I spend $O(M)$ for backtracking the path taken in each pair and $O(MN^2)$ in total for all pairs. The space complexity is $O(M^2)$ for each pair of genes, and total space complexity is $O(M^2N^2)$ to store the matrices of number of operations to transform each suffix to another in each pair of sequences.

The space complexity to this problem could be improved by storing only 2 rows in **edit_dist(str_1, str_2)** (current and previous row). Then, the exact operations can be derived from the following:

1. When we insert a base, the LCS b/w original and modified string does not change, but the length increases by 1

2. When we delete a base, the LCS drops by one, and the length drops by 1
3. When we mutate a base, the LCS drops by one, and the length does not change

Using these three conditions, we can infer the number of each operations by solving this system of equations:

$$\begin{cases} len(seq_1) = len(seq_2) + I - D \\ LCS(seq_1, seq_2) = len(seq_1) - D - M \\ num_{ops} = I + D + M \end{cases}$$

where I, D, M are the number of insertions, deletions, and mutations, respectively. $LCS(s_1, s_2)$ in this case gives the length of the longest common subsequence of two strings.

Since this is a linear system of equations, we can easily express each of the unknowns:

$$\begin{cases} I = len(seq_1) - len(seq_2) + D \\ D = len(seq_1) - LCS(seq_1, seq_2) - M \\ M = I + D - num_{ops} \end{cases}$$

$$\begin{cases} M = len(seq_1) - len(seq_2) + 2(len(seq_1) - LCS(seq_1, seq_2) - M) - num_{ops} \\ I = len(seq_1) - len(seq_2) + D \\ D = len(seq_1) - LCS(seq_1, seq_2) - M \end{cases}$$

$$\begin{cases} M = len(seq_1) - \frac{1}{3}len(seq_2) - \frac{2}{3}LCS(seq_1, seq_2) - \frac{1}{3}num_{ops} \\ D = \frac{1}{3}len(seq_2) - \frac{1}{3}LCS(seq_1, seq_2) + \frac{1}{3}num_{ops}) \\ I = len(seq_1) - \frac{2}{3}len(seq_2) - \frac{1}{3}LCS(seq_1, seq_2) + \frac{1}{3}num_{ops}) \end{cases}$$

Then, to calculate the number of each operation, we need only O(1) time complexity.

This improved solution then needs:

1. $O(M^2)$ time complexity to find the matrix of optimal transformations from each suffix of one string to each suffix of another. $O(M^2N^2)$ to find the optimal transformations for all pairs.
2. $O(M)$ space complexity to store the current row and upper row at each step, $O(MN^2)$ for all pairs.
3. $O(1)$ time complexity to determine the number of insertions, deletions, and mutations. $O(N^2)$ for all pairs.

Task 7

Give differences between this assignment and inferring relationships in real gene sequences.

In real life, whole genome sequences are ~3B long, so using Wagner-Fischer's algorithm to determine the relationship between two sequences would be very inefficient. There are methods more suitable for this, such as indexing (current state-of-art tool is Tabix) to efficientlyquery for certain parts of genomes.

Bibliography:

1. Wikipedia contributors. (2018, February 6). Wagner–Fischer algorithm. In Wikipedia, The Free Encyclopedia. Retrieved 01:55, December 13, 2018, from https://en.wikipedia.org/w/index.php?title=Wagner%E2%80%93Fischer_algorithm&oldid=824346754 (https://en.wikipedia.org/w/index.php?title=Wagner%E2%80%93Fischer_algorithm&oldid=824346754)
2. Li, H. (2017, January 5). Tabix: Fast retrieval of sequence features from generic TAB-delimited files. Retrieved from https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3042176/ (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3042176/)

Appendix:

#algorithms: I built upon Wagner-Fischer's algorithm to precisely calculate the amount of insertions, deletions, and mutations from the table of number of optimal operations and explained how such calculations could be improved in terms of time complexity.
#dataviz: Presented explanatory figures and tables and provided descriptive captions for all of them.

```
In [ ]:
```