CS110 - Assignment 4

April 14, 2019

1 Longest Common Subsequence

In [36]:
```
'''
Longest Common Subsequence
Dynamic Solution Adapted from Cormen et. al
'''
def LCS_length(X,Y):
    m = len(X)
    n = len(Y)
    c = [[0 for _ in range(n+1)] for _ in range(m+1)] #bottom up storage table
    for i in range(m+1):
        for j in range(n+1):#iterate through storage table
            if i == 0 or j == 0 :
                c[i][j] = 0
            elif X[i-1] == Y[j-1]:
                c[i][j] = c[i-1][j-1]+1
            else:
                c[i][j] = max(c[i-1][j] , c[i][j-1])

    return c[m][n]  #return len of LCS of X[0..n-1] and Y[0..m-1]
```

In [37]:
```
'''
Our set of Genes assambled into a list.
'''
g1='CAGCGGGTGCGTAATTTGGAGAAGTTATTCTGCAACGAAATCAATCCTGTTTCGTTAGCTTACGGACTACGACGAGAGGGTA
g2='CAAGTCGGGCGTATTGGAGAATATTTAAATCGGAAGATCATGTTACTATGCGTTAGCTCACGGACTGAAGAGGATTCTCTC
g3='CATGGGTGCGTCGATTTTGGCAGTAAAGTGGAATCGTCAGATATCAATCCTGTTTCGTAGAAAGGAGCTACCTAGAGAGGA
g4='CAAGTCCGCGATAAATTGGAATATTTGTCAATCGGAATAGTCAACTTAGCTGGCGTTAGCTTTACGACTGACAGAGAGAAA
g5='CAGTCCGGCGTAATTGGAGAATATTTTGCAATCGGAAGATCAATCTTGTTAGCGTTAGCTTACGACTGACGAGAGGGATAC
g6='CACGGGCTCCGCATCTATTTTGGGTCAAGTTGCATATCAGTCATCGACAATCAAACACTGTTTTGCGGTAGATAAGATACG
g7='CACGGGTCCAATTTTGGAGTAAGTTGATATCGTCACGAAATCAATCCTGTTTCGGTAGTATAGGACTACGACGAGAGAGGA
g8='GGTCCGTCAATTTTGGAGTAAGTTGATATCGTCACGAAATCAATCCTGTTTCGGTAGTATAGGACTACGACGAGAGAGGAC
g9='CACGGGAATCCGTCAATTTTGGAGTAAGTTGATATCGTCACGAAATCAATCCTGTTTCGGTAGTATAGGACTACGACGAGA
g10='CACGGGTCCGTCAATTTTGGAGTAAGTTGATATCGTCACGAAATCAATCCTGTTTCGGTAGTATAGGACTACGACGAGAG

set_strings=[g1,g2,g3,g4,g5,g6,g7,g8,g9,g10]
```

In [38]:
```
'''
Draw a table of LCS between every gene
'''
import pandas
string_names=['Gene 1','Gene 2','Gene 3','Gene 4','Gene 5','Gene 6', 'Gene 7','Gene 8

#create storage matrix
lcs_table=[[0 for _ in range(len(set_strings))] for _ in range(len(set_strings))]

#fill in table with LCS
for i in range(len(set_strings)):
    for j in range(len(set_strings)):
        lcs_table[i][j]=LCS_length(set_strings[i],set_strings[j])

#Table of LCS between every gene
pandas.DataFrame(lcs_table, string_names, string_names)
```

Out[38]:

	Gene 1	Gene 2	Gene 3	Gene 4	Gene 5	Gene 6	Gene 7	Gene 8	\
Gene 1	100	74	76	73	82	84	89	87	
Gene 2	74	90	67	72	79	71	69	68	
Gene 3	76	67	97	65	69	82	82	81	
Gene 4	73	72	65	96	80	72	68	67	
Gene 5	82	79	69	80	95	74	74	73	
Gene 6	84	71	82	72	74	114	95	93	
Gene 7	89	69	82	68	74	95	101	97	
Gene 8	87	68	81	67	73	93	97	100	
Gene 9	91	71	84	69	75	97	101	100	
Gene 10	91	71	84	69	75	97	101	100	

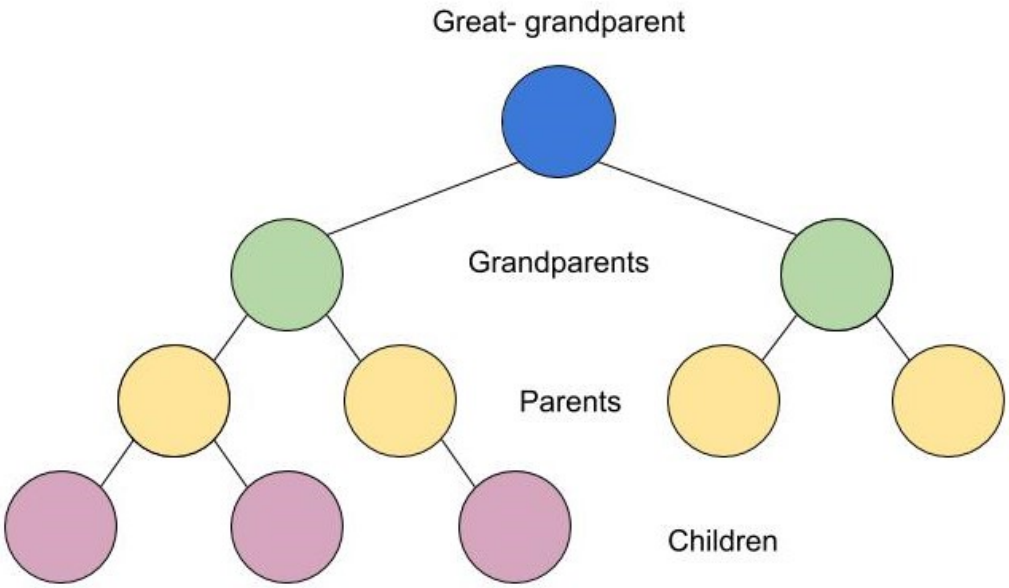
	Gene 9	Gene 10
Gene 1	91	91
Gene 2	71	71
Gene 3	84	84
Gene 4	69	69
Gene 5	75	75
Gene 6	97	97
Gene 7	101	101
Gene 8	100	100
Gene 9	106	104
Gene 10	104	104

2 Finding a Genealogy

Any genealogy tree constructed from the table above should have the below structure. Note that we do not know where exactly the children nodes belong. What matters is that there are three of them.

In [49]:
```
from IPython.display import Image
Image(filename='genealogy.jpg')
```

Out[49]:



In [30]:
```
'''
GENEALOGY FINDER

My algorithm will analyse and identify kinship relationships
by looking at the distribution of mutations.

1. Construct Binary Family Trees such that they have a 1-2-4-3 structure,
by choosing each gene as a GREAT GRANDPARENT ONCE and building the tree by
choosing the next gene which has the largest LCS.

2. Find a statistical measure and record the distribution of LCS's
on each level (Grandparent level, Parent Level, etc...). This measure will be
Standar Deviation

3. Assuming that mutations have a similar effect of change from every level, we can
identify implausible trees (where LCS distribution on a level are really skewed). In
our case, the most plausible trees will be the ones with the lowest levels of standar
deviation per level.

4. Select trees with the most plausible distribution on each level, return it
as a plausible genealogy. Draw Trees.
```

```
'''
import statistics

# 1. Create Node class to build the Binary Trees from
class Node(object):
    def __init__(self,key,name):
        self.l_child = None
        self.r_child = None
        self.parent = None
        self.data = key
        self.name= name


    def display(self):
        lines, _, _, _ = self._display_aux()
        for line in lines:
            print(line)

    def _display_aux(self):
        """Returns list of strings, width, height, and horizontal coordinate of the r
        # No child.
        if self.l_child is None and self.l_child is None:
            line = '%s' % self.name
            line = line+' ('+str(self.data)+')'
            width = len(line)
            height = 1
            middle = width // 2
            return [line], width, height, middle

        # Only l_child child.
        if self.r_child is None:
            lines, n, p, x = self.l_child._display_aux()
            s = '%s' % self.name
            s = s +' ('+str(self.data)+')'
            u = len(s)
            first_line = (x + 1) * ' ' + (n - x - 1) * '_' + s
            second_line = x * ' ' + '/' + (n - x - 1 + u) * ' '
            shifted_lines = [line + u * ' ' for line in lines]
            return [first_line, second_line] + shifted_lines, n + u, p + 2, n + u // 1

        # Only r_child child.
        if self.l_child is None:
            lines, n, p, x = self.r_child._display_aux()
            s = '%s' % self.name
            s = s +' ('+str(self.data)+')'
            u = len(s)
```

```
            first_line = s + x * '_' + (n - x) * ' '
            second_line = (u + x) * ' ' + '\\' + (n - x - 1) * ' '
            shifted_lines = [u * ' ' + line for line in lines]
            return [first_line, second_line] + shifted_lines, n + u, p + 2, u // 2

        # Two children.
        l_child, n, p, x = self.l_child._display_aux()
        r_child, m, q, y = self.r_child._display_aux()
        s = '%s' % self.name
        s = s +' ('+str(self.data)+')'
        u = len(s)
        first_line = (x + 1) * ' ' + (n - x - 1) * '_' + s + y * '_' + (m - y) * ' '
        second_line = x * ' ' + '/' + (n - x - 1 + u + y) * ' ' + '\\' + (m - y - 1) *
        if p < q:
            l_child += [n * ' '] * (q - p)
        elif q < p:
            r_child += [m * ' '] * (p - q)
        zipped_lines = zip(l_child, r_child)
        lines = [first_line, second_line] + [a + u * ' ' + b for a, b in zipped_lines]
        return lines, n + m + u, max(p, q) + 2, n + u // 2

#insert function for creating a tree
def insert(arr, root, i,n, names_list):
    if i < n:
        temp = Node(arr[i],names_list[i])
        root = temp
        # insert left child
        root.l_child = insert(arr, root.l_child,2 * i + 1, n,names_list)
        # insert right child
        root.r_child = insert(arr, root.r_child,2 * i + 2, n,names_list)
    return root


#create tree function
def create_tree(lcs_table,string_names,i):
    printlist=[]
    #create list of lcs values for gene i
    lcs_data=lcs_table[i][:]
    index_list=[]

    for j in range(len(lcs_data)):
        index_list.append([lcs_data[j],j])

    index_list[i][0]=-float('inf')
    index_list=sorted(index_list, key=lambda x: x[0], reverse=True)
    index_list=[index_list[i][1] for i in range(len(index_list))]
    names_list=[string_names[x] for x in index_list]
    names_list= (names_list[-1:] + names_list[:-1])
```

```
            index_list.pop()

            for j in range(len(lcs_data)):
                if j==0:
                    printlist.append(lcs_table[i][i])
                else:
                    k=index_list.pop(0)
                    printlist.append(lcs_data[k])
            family_tree=None
            family_tree=insert(printlist,family_tree,0,len(printlist),names_list)

            return printlist,family_tree

        #calculate standard deviation for each level
        import statistics

        def std_dev(arr):
            std_list=[]
            #calculate std dev for each level
            level2=statistics.stdev(arr[1:3])
            level3=statistics.stdev(arr[3:7])
            level4=statistics.stdev(arr[7:10])

            #create a descriptive statistic (sum of stdev for level 2, 3 and 4 of tree)

            descriptive=level2+level3+level4

            return descriptive
```

2.1 Complexity

Assuming a list of n gene sequences, our LCS algorithm will take $O(n^2)$ to generate a table of LCS between every gene. Then it would take $O(l * n)$ time to construct every tree, where l is the length of the sequence. Finding the most plausible genealogy is also a $O(n)$ operation. Implementing these functions together, the task of finding the most plausible genealogy will be of $O(n^2)$ complexity.

```
In [31]: '''
         Build a tree with each gene as a root. Calculate Descriptive Statistics for each tree
         Return best trees.

         '''
         trees_arr=[create_tree(lcs_table,string_names,i) for i in range(len(string_names))]
         stat_arr=[[std_dev(trees_arr[i][0]),i] for i in range(len(string_names))]
         stat_arr=sorted(stat_arr, key=lambda x: x[0])

         '''
```

```
        '''
        Print First Best Tree.
        '''
        best_tree=trees_arr[stat_arr[0][1]]
        best_tree[1].display()
        print('\n',"This tree has a sum total of",round(stat_arr[0][0],3), 'standard deviatio
```

```
                                  _____________________________Gene 1 (100)_________________
                                 /
                  _________________Gene 9 (91)_________________                      _____Gene
                 /                                             \                    /
       _____Gene 7 (89)_____                                    _____Gene 8 (87)          Gene 6 (84)
      /                     \                                  /
Gene 3 (76)           Gene 2 (74)           Gene 4 (73)
```

This tree has a sum total of 4.637 standard deviation for all levels.

```
In [32]: '''
         Print Second Best Tree.
         '''
         best_tree=trees_arr[stat_arr[1][1]]
         best_tree[1].display()
         print('\n',"This tree has a sum total of",round(stat_arr[1][0],3), 'standard deviatio
```

```
                                  _____________________________Gene 3 (97)_________________
                                 /                                                          \
                  _________________Gene 9 (84)_________________                      _____Gene 10
                 /                                             \                    /
       _____Gene 6 (82)_____                                    _____Gene 7 (82)          Gene 8 (81)
      /                     \                                  /
Gene 5 (69)           Gene 2 (67)           Gene 4 (65)
```

This tree has a sum total of 4.872 standard deviation for all levels.

```
In [33]: '''
         Print Third Best Tree.
         '''
         best_tree=trees_arr[stat_arr[2][1]]
         best_tree[1].display()
         print('\n',"This tree has a sum total of",round(stat_arr[2][0],3), 'standard deviatio
```

```
                                  _____________________________Gene 2 (90)_________________
                                 /                                                        \
                  _________________Gene 5 (79)_________________                      _____Gene 1
                 /                                             \                    /
       _____Gene 4 (72)_____                                    _____Gene 6 (71)          Gene 9 (71)
      /                     \                                  /
Gene 7 (69)           Gene 8 (68)           Gene 3 (67)
```

```
This tree has a sum total of 5.036 standard deviation for all levels.
```

3 Probabilities

In order to find the probabilities of insertion, mutation and removal, P_i, P_m and P_r, we would want to calculate what it takes to change the sequence of one gene to another. I will write an algorithm that calculates that smallest amount of possible steps with which we can achieve this.

I will then use the algorithm to calculate the probabilities mentioned above on our three most plausible trees. This way, we can get relatively good estimates for P_i, P_m and P_r for populations larger than our current gene pool.

In [34]:
```
'''
Minimum Edit Distance

1. Finds number of edits to get Gene2 from Gene1

A bottom-up dynamic solution, similar to the LCS problem,
that when employed on our PLAUSIBLE trees, should allow us
to calculate good probability estimates.

'''
def edit_dist(g1, g2, m, n):
    # Storage table for subproblem solutions
    table = [[0 for x in range(n+1)] for x in range(m+1)]

    #bottom-up filling of table
    for i in range(m+1):
        for j in range(n+1):

            # case 1: gene1 empty
            if i == 0:
                table[i][j] = j

            #case 2: gene2 empty
            elif j == 0:
                table[i][j] = i

            # check last character equality, recurrence for remaining gene sequence
            elif g1[i-1] == g2[j-1]:
                table[i][j] = table[i-1][j-1]

            # if last character different, decide what operation is the best
            else:
                table[i][j] = 1 + min(table[i][j-1],  # insertion
                                      table[i-1][j],    # deletion
```

```
                                        table[i-1][j-1]) # mutate

            #backtracking steps
            steps=[]
            i,j=m,n
            while i!=0 and j!=0:
                diff=table[i][j]==table[i-1][j-1] or table[i][j]==table[i-1][j-1]-1
                if min(table[i][j-1],table[i-1][j], table[i-1][j-1])==table[i-1][j-1] and dif
                    if table[i-1][j-1]==table[i][j]-1:
                        steps.append('mutate')
                    i-=1
                    j-=1
                elif min(table[i][j-1],table[i-1][j], table[i-1][j-1])==table[i][j-1] and dif
                    steps.append('insert')
                    j-=1
                else:
                    steps.append('remove')
                    i-=1


            return steps

        def edit_distance_steps(g1,g2):
            #return the probability of different operations to make g2 from g1
            steps=edit_dist(g1,g2,len(g1),len(g2))
            insert=steps.count('insert')
            mutate=steps.count('mutate')
            remove=steps.count('remove')
            n=insert+mutate+remove

            return insert/n,mutate/n,remove/n

        edit_distance_steps(g1,g2)
```

Out[34]: (0.4074074074074074, 0.0, 0.5925925925925926)

In [48]:
```
'''
Calculating Probabilites

Now that we have an algorithm to find us the
probability of steps, assuming that gene mutation
would take the shortest path from a sequence to
another, we can use our most plausible trees to
get an estimate for the total probability of our
operations:

1. Insert
2. Mutate
```

```
3. Delete
'''
def genes_index(set_strings,lcs_table,i):
    #given a gene find corresponding list of genes in level-order
    lcs_data=lcs_table[i][:]
    index_list=[]

    for j in range(len(lcs_data)):
        index_list.append([lcs_data[j],j])

    index_list[i][0]=-float('inf')
    index_list=sorted(index_list, key=lambda x: x[0], reverse=True)
    index_list=[index_list[i][1] for i in range(len(index_list))]
    index_list= (index_list[-1:] + index_list[:-1])
    return index_list



def check_prob(set_strings,gi):
    #check probability of mutations between parents and children
    total=[]
    total.append(edit_distance_steps(set_strings[gi[0]],set_strings[gi[1]]))
    total.append(edit_distance_steps(set_strings[gi[0]],set_strings[gi[2]]))
    total.append(edit_distance_steps(set_strings[gi[1]],set_strings[gi[3]]))
    total.append(edit_distance_steps(set_strings[gi[1]],set_strings[gi[4]]))
    total.append(edit_distance_steps(set_strings[gi[2]],set_strings[gi[5]]))
    total.append(edit_distance_steps(set_strings[gi[2]],set_strings[gi[6]]))
    total.append(edit_distance_steps(set_strings[gi[3]],set_strings[gi[7]]))
    total.append(edit_distance_steps(set_strings[gi[3]],set_strings[gi[8]]))
    total.append(edit_distance_steps(set_strings[gi[4]],set_strings[gi[9]]))

    insert,mutate,remove=0,0,0

    for i in total:
        insert+=i[0]
        mutate+=i[1]
        remove+=i[2]

    return insert/9,mutate/9,remove/9


#calculate probs based on best three trees
probs=[]
#tree1
probs.append(check_prob(set_strings,genes_index(set_strings,lcs_table,0)))
#tree2
probs.append(check_prob(set_strings,genes_index(set_strings,lcs_table,2)))
#tree3
```

```
        probs.append(check_prob(set_strings,genes_index(set_strings,lcs_table,1)))

        insert,mutate,remove=0,0,0
        #print out total probabilites
        for i in probs:
            insert+=i[0]
            mutate+=i[1]
            remove+=i[2]
        insert=insert/3
        mutate=mutate/3
        remove=remove/3

        print('Probability of insertion: ',round(insert,3))
        print('Probability of mutation: ',round(mutate,3))
        print('Probability of removal: ',round(remove,3))
```

```
Probability of insertion:  0.448
Probability of mutation:  0.0
Probability of removal:  0.552
```

Calculating the probabilities of these different type of inheritance operations from our most plausible trees we get that $P_i = 0.448, P_m = 0$ and $P_r = 0.552$. These results are especially interesting, since they tell us that mutations are not plausible changes amongst our genes, and it was only insertion and removal that took place.

However, we must keep in mind that these results rely upon multiple assumptions, the most important amongst which:

1. The trees that we identified as plausible make up a representative sample of gene to gene differences.
2. The solution to our edit-distance algorithm corresponds to the edits that would have taken place during the process of inheritance.

4 HC Applications

#sampling: In this assignment, when calculating the probabilities of insertion, mutation and removal I accurately describe what sampling method I would use (sampling the top 3 most plausible trees) and assert that such a small sampling size is too great for accurate estimates, yet it is the best solution we have.

#descriptivestats: When having to decide for the most plausible geneaologies, I devised a method by which I calculate an overall descriptive statistic from the standard deviations between the LCS results for each level. In my accompanying description I justified why this is an appropriate choice, and following that I implemented my method correctly.

5 References

My tree printing method in the Node object definition was adopted from @J.V.'s comment at https://stackoverflow.com/questions/34012886/print-binary-tree-level-by-level-in-python.