In [1]:

```
### Question 1 and 2
#LCS code adopted from https://en.wikipedia.org/wiki/Longest_common_subsequence_problem
# BackTrackAll code modified by using
#https://en.wikibooks.org/wiki/Algorithm_Implementation/Strings/Longest_common_subsequenc
#as reference

import string
import numpy as np
from tabulate import tabulate


def LCS(X, Y):
    m = len(X)
    n = len(Y)
    # An (m+1) times (n+1) matrix
    C = [[0] * (n + 1) for _ in range(m + 1)]
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if X[i - 1] == Y[j - 1]:
                C[i][j] = C[i - 1][j - 1] + 1
            else:
                C[i][j] = max(C[i][j - 1], C[i - 1][j])
    return C


def backTrackAll(C, X, Y, i=None, j=None):
    if i is None:
        i = len(X)
    if j is None:
        j = len(Y)
    if i == 0 or j == 0:
        return ""
    elif X[i - 1] == Y[j - 1]:
        return backTrackAll(C, X, Y, i - 1, j - 1) + X[i - 1]
    else:
        if C[i][j - 1] >= C[i - 1][j]:
            return backTrackAll(C, X, Y, i, j - 1)
        else:
            return backTrackAll(C, X, Y, i - 1, j)
    #results = LCS("abcd", "accbc")
#print(backTrackAll(results, "abcd", "accbc"))

genes =[(0,'TTCTACGGGGGGAGACCTTTACGAATCACACCGGTCTTCTTTGTTCTAGCCGCTCTTTTTCATCAGTTGCAGCTAGT
        (1,'TCTACGGGGGGCGTCATTACGGAATCCACACAGGTCGTTATGTTCATCTGTCTCTTTTCACAGTTGCGGCTTGTGCA
        (2,'TCTACGGGGGGCGTCTATTACGTCGCCAACAGGTCGTATGTTCATTGTCATCATTTTCATAGTTGCGGCCTGTGCGT
        (3,'TCCTAACGGGTAGTGTCATACGGAATCGACACGAGGTCGTATCTTCAATTGTCTCTTCACAGTTGCGGCTGTCCATA
        (4,'TATCAGTAGGGCATACTTGTACGACATTCCCCGGATAGCCACTTTTTTCCTACCCGTCTCTTTTTCTGACCCGTTCC
        (5,'TAATCTATAGCATACTTTACGAACTACCCCGGTCCACGTTTTTCCTCGTCTTCTTTCGCTCGATAGCCATGGTAACT
        (6,'TATCATAGGGCATACTTTTACGAACTCCCCGGTGCACTTTTTTCCTACCGCTCTTTTTCGACTCGTTGCAGCCATGA

table = np.zeros([8,8])
table[0,0]=float("nan")

for i in range (0, 7):
    table[0,i+1]= i
    table[i+1,0]= i

for i in range (len(genes)-1):
    for j in range (i+1, len(genes)):
```

```
        lcs = LCS(genes[i][1], genes[j][1])

        backtrack = backTrackAll(lcs, genes[i][1], genes[j][1])
        table[i+1][j+1] = len(backtrack)

for i in xrange(len(genes)):
    for j in xrange(i, len(genes)):
        if i == j:
            table[i+1][j+1] = len(genes[i][1])
        else:
            table[j+1][i+1] = table[i+1][j+1]

present_table = tabulate(table)

print present_table
```

```
---  ---  --  --  --  --  --  --
nan    0   1   2   3   4   5   6
  0  100  82  73  72  72  70  80
  1   82  96  83  81  67  65  70
  2   73  83  94  73  62  61  67
  3   72  81  73  97  62  60  63
  4   72  67  62  62  98  71  82
  5   70  65  61  60  71  89  79
  6   80  70  67  63  82  79  94
---  ---  --  --  --  --  --  --
```

Question 3

Since the first generation two child 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, so those child strings will have the higest LCS with that parent string. Also the second generation child strings follow the same pattern. We could look at the table and for each given string, find out the two highest strings (except for the cell where the string self LCSs or the cells that have the LCS of the strings with their parents) and thus form a binary tree with the grandparents, parents and children.

--> Looking at the first row, String 0 has the highest LCS with String 1 (82) and String 6 (80) . So it is confident to say that String 0 is the parent of String 1 and String 6.

--> Now looking at the second row, String 1 has the highest LCS with String 2 (83) and String 3 (81) . So it is confident to say that String 0 is the parent of String 2 and String 3.

--> Looking at the sixth row, String 6 has the highest LCS with String 4 (82) and String 5 (79) . So it is confident to say that String 6 is the parent of String 4 and String 5.

So summarizing, we can get the binary tree as Grandparent: 0 Parent: 1 6 Children: 2 3 4 5

Question 4

#In order to induce the probability of the mutation, insertion and deletion, we have to assume that the relation I made from question 3 is correct. My LCS_pattern function compares the parent/child string with the LCS to find the differences between them. My genes_operation function starts comparing the parent pattern with the

children pattern. There are 3 cases that can occur. Case 1 - When parent's pattern has something that the child's pattern doesnt. In this case, a deletion has occured in the offspring.

Case 2 - When parent's pattern doesn't have something that the child's pattern does. In this case, an insertion has occured in the offspring

Case 3 - Mutation has occured.

In [2]:

```
from tabulate import tabulate
def LCS_pattern(x, r):
    count = 0
    word = ""
    for i in xrange(len(x)):
        if count == len(r):
            break
        if x[i] == r[count]: #if there is no difference between the LCS and the
                             #string, then "-" is assigned. Else " " is assigned.
            word += "-"
            count += 1
        else:
            word += " "
    return word

def genes_operation(parent, child): #gets two strings, the parent string and its
                                    #child string, according to the tree above.
    c = LCS(parent, child) #finds the LCS between them
    r = backTrackAll(c, parent, child) #Returns the LCS string.
    parent_code = LCS_pattern(parent, r) #Returns the LCS pattern for the parent.
    children_code = LCS_pattern(child, r) #Returns the LCS pattern for the child.
    insert = 0
    delete = 0
    mutate = 0
    i = 0
    j = 0
    while i < len(parent_code):
        if parent_code[i] == "-":
            if children_code[j] == "-":
                i += 1
                j += 1
            else:
                delete += 1
                j += 1
        else:
            if children_code[j] == "-":
                insert += 1
                i += 1
            else:
                mutate += 1
                j += 1
                i += 1
    result = insert, delete, mutate

    return result

birth = np.zeros((6, 3))
pairs = ((0, 1), #pairs of the format (parent, child) based on the tree
                 #constructed from Question 3
         (0, 6),
         (1, 2),
         (1, 3),
         (6, 4),
         (6, 5),)

for i in xrange(len(pairs)):
    birth[i] = genes_operation(genes[pairs[i][0]][1], genes[pairs[i][1]][1])
present_birth = tabulate(birth)
print "Insertion, Deletion and Mutation table for six gene splits"
```

```
print present_birth

print " "

chance = birth / np.sum(birth, axis=1).reshape((6, 1))
present_chance = tabulate(chance)
print "Following is the probability table for Insertion, Deletion, and Mutation."
"The probabilities sum up to 1 for each row"
print present_chance
print " "

print "Following are average probabilities of Insertion, Deletion, and Mutation"
"for the genes"
ave = np.average(chance, axis=0)
print "Insertion:", ave[0]
print "Deletion:", ave[1]
print "Mutation:", ave[2]
print " "

print "Another approach of finding average probabilites is by summing up all the"
"changes in all six genes split column wise"
print np.sum(birth, axis=0) / np.sum(birth)
```

```
Insertion, Deletion and Mutation table for six gene splits
--  --  -
13   9  5
16  10  4
10   7  3
11  12  3
12  16  0
12   7  3
--  --  -

Following is the probability table for Insertion, Deletion, and Mutation.The
probabilities sum up to 1 for each row
--------  --------  --------
0.481481  0.333333  0.185185
0.533333  0.333333  0.133333
0.5       0.35      0.15
0.423077  0.461538  0.115385
0.428571  0.571429  0
0.545455  0.318182  0.136364
--------  --------  --------

Following are average probabilities of Insertion, Deletion, and Mutation for
the genes
Insertion: 0.485319618653
Deletion: 0.394635919636
Mutation: 0.120044461711

Another approach of finding average probabilites is by summing up all the ch
anges in all six genes split
[ 0.48366013  0.39869281  0.11764706]
```

Question 5

While there might be many ways to infer a tree relationship, the one that I can think of is an algorithm that takes the LCS length table for each gene pair and somehow ranks it and then chooses a gene by row and finds the

worst and best pairing for that gene. This will give us a better idea of which genes to pair together to create the best offspring.

In [3]:

```
from scipy.stats import rankdata
from tabulate import tabulate
ranks = np.zeros([8,8])
ranks[0,0]=float("nan")
for i in range (0,7):
    ranks[0,i+1]= i
    ranks[i+1,0]= i
    for i in range(0,7):
        ranks[i+1] = rankdata(table[i+1], method='min')

#just because rankdata ranks horizontally for each line, I had to make
#the following function to restore the table presentation
#and get the actual ranks
for i in range(1,8):
    ranks[i] = ranks[i]-1
for i in range (0,7):
    ranks[i+1,0]= i
present_ranks = tabulate(ranks)
print "The ranking table for each gene pair"
print present_ranks
```

```
The ranking table for each gene pair
---  -  -  -  -  -  -  -
nan  0  1  2  3  4  5  6
  0  7  6  4  2  2  1  5
  1  5  7  6  4  2  1  3
  2  4  6  7  4  2  1  3
  3  4  6  5  7  2  1  3
  4  5  3  1  1  7  4  6
  5  4  3  2  1  5  7  6
  6  5  3  2  1  6  4  7
---  -  -  -  -  -  -  -
```

Strength of the Ranking Algorithm

1. The Algoritm can perform very fast as it already has the LCS table in hand.
2. The rankings can be further used to choose the best gene pairs. (1 is the worst and 6 is the best. Not 7 because it is self gene multiplication)

Weakness of the Ranking Algorithm

1. It depends on the previously generated LCS table to rank.
2. It ranks horizontally for each line, due to which I had to add two other for loops to get the correct ranking and maintain my table presentation. So not very efficient.

Question 6

As stated in answer 5, The time complexity of my algorithm in question 5 (not taking the LCS code complexity from the first cell into account and assuming that we already have the LCS table) does not require M (the length of the gene) as it works based on the LCS table. (Also I don't take into account the extra two loops I had to insert into the fuction because that was only required because of the orientation of my LCS table)

The algorithm starts ranking the table row by row. So for a general case, there will be N rows and each row will have N collumns. So it will take O(N x N) = O(N^2) time to go through the table.

After going through each row, it behaves as a sorting algorithm by ranking the higest values with a high rank. This can be done in O(log N) time.

So the overall time complexity of the ranking algorithm stands at O(N^2).O(log N) = O(N^2 . log N)

However, the LCS algorithm from cell one has a complexity of O(M x N) itself.

So the overall time complexity of the algorithm becomes O(M x N . N^2 . log N) = O(MN^3.log N)