Lab 06 Ancestral states

The goal of Lab 06 | Ancestral states is to calculate the likelihood of observed character states given a phylogenetic tree along with branch lengths. The likelihoods will also be used to infer the most likely ancestral state at each position in the tree. This can be done manually or using dynamic programming and a recursion to implement Felsenstein's pruning algorithm.

The lab is split into the following sections:

- Phylogenetic models
- Ancestral states
- Recursion and Felsenstein's pruning algorithm

Assignment

Follow the instructions in this document and answer the questions in the cell below each question. Submit your answers by uploading a PDF file to gradescope. To generate the pdf, first export the notebook as HTML: >File, >Export to ..., >HTML. Then, open the HTML in a browser and use your browser to print to PDF.

Check to make sure all your cells have been run and the **results** displayed in the PDF file.

Reminder, provide comments for any code you write to ensure partial credit.

Phylogenetic Models

In the previous lab, we calculated the likelihood of two aligned sequences under various nucleotide substitution models. However, in many instances we are interested in the likelihood of a multiple sequence alignment (MSA). For a given column at position i of a multiple sequence alignment, we want to calculate $P(D_i|\tau; M)$, where τ is a phylogenetic tree and M is a nucleotide substitution model. Assuming sites and lineages evolve independently of one another, the probability is thus the product of the probabilities at each site:

$$L = P(D|\tau; M) = \prod_{i=1}^{m} P(D_i|\tau; M) \tag{1}$$

where D_i is the ith site. This means we only need to know how to compute the likelihood at a single site.

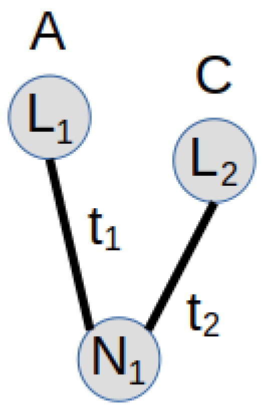
The conditional probability given a tree brings a complex new dimension to the problem. In phylogenetics the goal is to find a tree that maximizes the likelihood. Each tree that is considered consists of both a single topology but also the branch lengths separating the nodes and leaves on the tree. Thus, to consider the likelihood of a single tree topology one must find the branch lengths that give the highest likelihood of the data given the topology. Finding the best tree is hard since there are so many tree topologies that need to be evaluated.

In this lab we will focus on how to calculate the likelihood of the data *given* both a tree and branch lengths. Even in this simpler situation, the likelihood is not straightforward to calculate because we don't know the ancestral states present at each node.

Evaluating the likelihood of the data over all possible ancestral states is not only relevant to evaluating the likelihood of various tree topologies, it is also needed to infer ancestral states, estimate branch lengths in a tree, compare different substitution models and in many other likelihood based methods that uses multiple sequence alignments.

Ancestral states

Consider the following tree τ_1, with two leaves L_1 and L_2, one ancestral node N_1 and two branch lengths t_1 and t_2:



The likelihood of the data given the tree is:

$$P(D|\tau_1) = \prod_{N_1 \in [A,G,C,T]} P(A|N_1,t_1) \cdot P(C|N_1,t_2)\Big) \tag{2}$$

In other words, the likelihood is the product of transitioning between the leaf nodes and all four ancestral states `[A, G, C, T]` at N_1.

To find the most likely state of N_1 we should calculate the likelihood for each of the four possibilities. Lets first just consider the possibility that $N_1 = C$:

$P(D|N_1 = C, \tau_1) = P_{CA}(t_1) \cdot P_{CC}(t_2)$

If $t_1 = 0.5$, $t_2 = 0.25$, we can use the JC69 model to get the transition probabilities using:

$P(t) = e^{Qt}$

In [1]:
```
import numpy as np
```

```
from scipy.linalg import expm
def JC69():
    Q = np.full((4,4),0.25)
    np.fill_diagonal(Q,-.75)
    Q = Q/0.75
    return Q

# Initialize rate matrix
Q = JC69()
P1 = expm(Q*0.5)
P2 = expm(Q*0.25)

# Calculate P
print(P1)
print(P2)

# Calculate P(D|N=C)
PNC = P1[2][0]*P2[2][2]
print(PNC)
```

```
[[0.63506284 0.12164572 0.12164572 0.12164572]
 [0.12164572 0.63506284 0.12164572 0.12164572]
 [0.12164572 0.12164572 0.63506284 0.12164572]
 [0.12164572 0.12164572 0.12164572 0.63506284]]
[[0.78739848 0.07086717 0.07086717 0.07086717]
 [0.07086717 0.78739848 0.07086717 0.07086717]
 [0.07086717 0.07086717 0.78739848 0.07086717]
 [0.07086717 0.07086717 0.07086717 0.78739848]]
0.095783655573403
```

Now lets consider all the cases, i.e. N can be A, G, C or T:

In [2]:
```
PNA = P1[0][0]*P2[0][2] # AA and AC with N=A and A to A on one lineage and A to
print(PNA)
PNG = P1[1][0]*P2[1][2] # GA and GC
print(PNG)
PNC = P1[2][0]*P2[2][2] # CA and CC
print(PNC)
PNT = P1[3][0]*P2[3][2] # TA and TC
print(PNT)
```

```
0.045005107688810372
0.0086206882228 1631
0.095783655573403
0.008620688222816312
```

Not surprisingly, A and C are more likely than G and T, but C is the most likely. There is an easier way to do this using NumPy's dot products. If L_1 is the probability of `[A,G,C,T]` on a leaf then the left and right parts of N_1 are:

In [3]:
```
L1 = np.array([1,0,0,0])
L2 = np.array([0,0,1,0])
PNL = L1.dot(P1) # Left probability
PNR = L2.dot(P2) # Right probability
print("Left ",PNL)
print("Right",PNR)
PN1 = PNR*PNL
print("Both ",PN1)
```

```
Left  [0.63506284 0.12164572 0.12164572 0.12164572]
Right [0.07086717 0.07086717 0.78739848 0.07086717]
Both  [0.04500511 0.00862069 0.09578366 0.00862069]
```

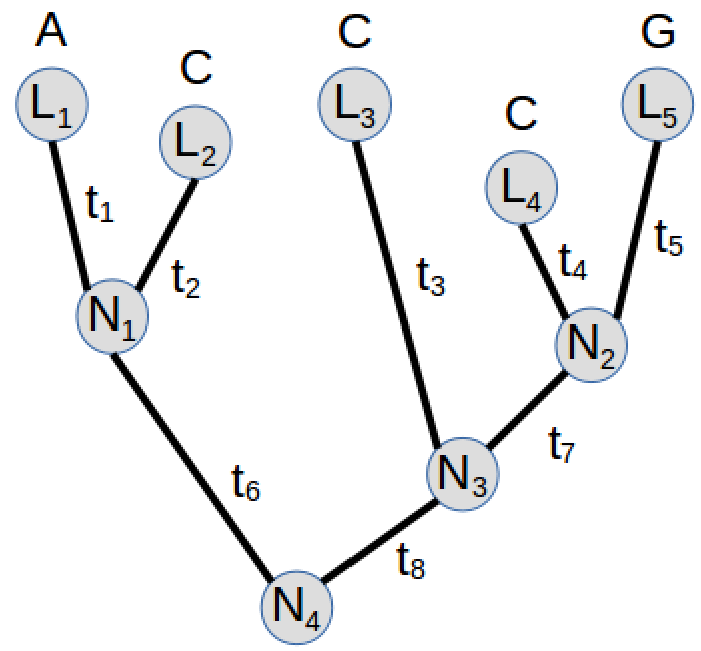
This is essentially the same thing as calculating $P_{AN}(t_1) \times P_{CN}(t_2)$ = `P1[0,:]*P2[2,:]`

In [4]:
```
print( P1[0,:]*P2[2,:] )
```

```
[0.04500511 0.00862069 0.09578366 0.00862069]
```

Multiple nodes

What about doing the same thing for Node 4? This is the product of the left and right descendants. The likelihood at N_4 given the left descendant is the transition probabilities multiplied by each of the likelihoods at N_1.



But, there are lots of paths to consider: The likelihood of A at Node 4 $P(N_4 = A)$ is the sum of:

$$P(N_4 = A) = N_1(A)P_{AA}(t_6) + N_1(G)P_{GA}(t_6) + N_1(C)P_{CA}(t_6) + N_1(T)P_{TA}(t_6)$$

Which is the sum of:

- the likelihood of A at N_1 times the probability of going from A at Node 1 to A at Node 4
- the likelihood of G at N_1 times the probability of going from G at Node 1 to A at Node 4
- the likelihood of C at N_1 times the probability of going from C at Node 1 to A at Node 4
- the likelihood of T at N_1 times the probability of going from T at Node 1 to A at Node 4

Note that $P_{GA} = P_{AG}$, the JC69 as well as other models are reversible.

We'll use $t_6 = 0.1$ and we already calculated the probabilities of each base at N_1.

In [5]:
```
P6 = expm(Q*0.1)
```

```
print(P6)

PN4A = PN1[0]*P6[0,0]+PN1[1]*P6[1,0]+PN1[2]*P6[2,0]+PN1[3]*P6[3,0]
print( PN4A )
```

```
[[0.90637999 0.03120667 0.03120667 0.03120667]
 [0.03120667 0.90637999 0.03120667 0.03120667]
 [0.03120667 0.03120667 0.90637999 0.03120667]
 [0.03120667 0.03120667 0.03120667 0.90637999]]
0.044318863926988396
```

Great, now we just have to do the same thing for $P(N_4 = C)$, $P(N_4 = G)$ and $P(N_4 = T)$:

In [6]:
```
PN4A = PN1[0]*P6[0,0]+PN1[1]*P6[1,0]+PN1[2]*P6[2,0]+PN1[3]*P6[3,0]
PN4G = PN1[0]*P6[0,1]+PN1[1]*P6[1,1]+PN1[2]*P6[2,1]+PN1[3]*P6[3,1]
PN4C = PN1[0]*P6[0,2]+PN1[1]*P6[1,2]+PN1[2]*P6[2,2]+PN1[3]*P6[3,2]
PN4T = PN1[0]*P6[0,3]+PN1[1]*P6[1,3]+PN1[2]*P6[2,3]+PN1[3]*P6[3,3]
print( PN4A, PN4G, PN4C, PN4T)
```

```
0.044318863926988396 0.012476190782101982 0.08875889421594703 0.0124761907821019
79
```

We can see that C is the most likely, *just* based on the left descenant.

But there is an easier way to do these calculations. The dot product of the likelihood of Node 1 and the P6 matrix makes the same calculations:

In [7]:
```
PN4 = PN1.dot(P6)
print( PN4 )
```

```
[0.04431886 0.01247619 0.08875889 0.01247619]
```

But remember this is only the left side. We need the right side as well to get the likelihood of Node 4. The right side depends on Node 3 and consequently Node 2 as well.

We'll use the following vector for the 8 branch lengths (time):

`t = [0.5, 0.25, 0.75, 0.1, 0.15, 0.1, 0.1, .5]`

Such that t_1 = `t[0]` = 0.5, etc.

In [8]:
```
L1 = np.array([1,0,0,0])
L2 = np.array([0,0,1,0])
L3 = np.array([0,0,1,0])
L4 = np.array([0,0,1,0])
L5 = np.array([0,1,0,0])
t = [0.5, 0.25, 0.75, 0.1, 0.15, 0.1, 0.1, .5]
PN1 = L1.dot(expm(Q*t[0])) * L2.dot(expm(Q*t[1]))
PN2 = L4.dot(expm(Q*t[3])) * L5.dot(expm(Q*t[4]))
PN3 = L3.dot(expm(Q*t[2])) * PN2.dot(expm(Q*t[6]))
PN4 = PN1.dot(expm(Q*t[5])) * PN3.dot(expm(Q*t[7]))

# Check PN1
print(PN1)
# Check left side of PN4
print( PN1.dot(expm(Q*t[5])) )
# PN4
print( PN4 )
```

```
[0.04500511 0.00862069 0.09578366 0.00862069]
[0.04431886 0.01247619 0.08875889 0.01247619]
[1.48460804e-04 6.44278661e-05 1.18700233e-03 4.17931589e-05]
```

This tells us that the likelihood at Node 4 of A, G, C, T is highest for C.

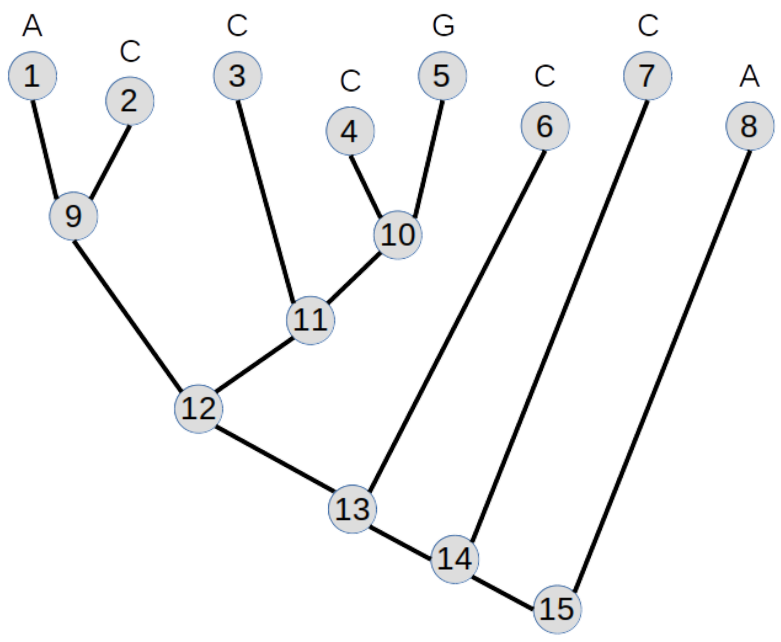
To calculate the posterior probability of the ancestral state, we must multiply by the prior probabilities, i.e. the equilibrium values of A, G, C, T which under JC69 model are 0.25. Of course since they are all 0.25, this doesn't change which is the most likely ancestral state. It is still C.

In [9]:
```
pi = np.array([.25, .25, .25, .25])
print (pi*PN4)
```

```
[3.71152011e-05 1.61069665e-05 2.96750582e-04 1.04482897e-05]
```

Question 1

Use the tree, observed states at the leaves, branch lengths and the JC69, calculate the posterior probability of [A,G,C,T] at node 15. (An explanation of how the tree is encoded is given at the beginning of the next section).

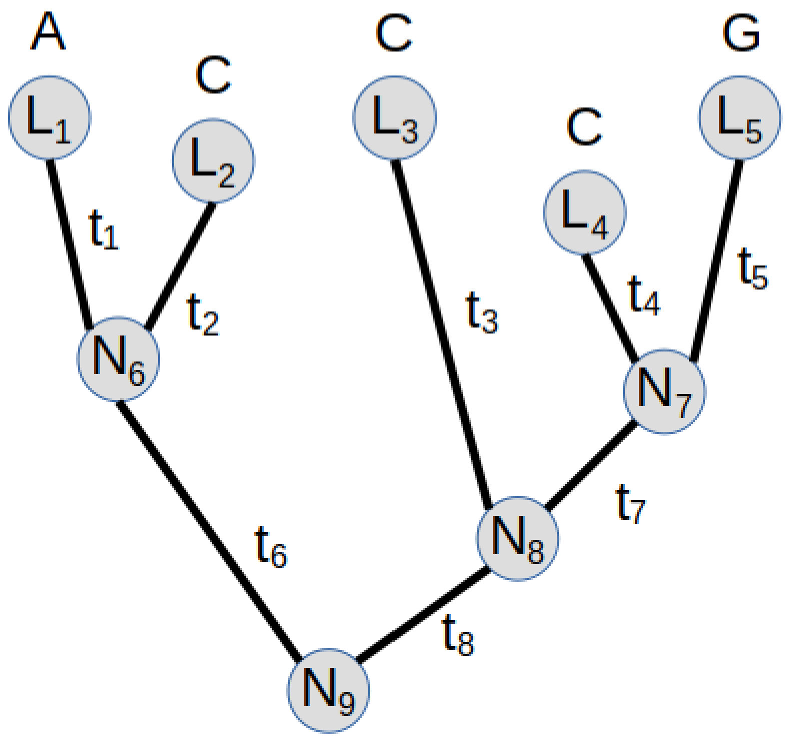


(4 points)

In [24]:
```
# Use this character state vector indicating
# the nucleotide at leaf nodes 1-8
site1=["A","C","C","C","G","C","C","A"]

# Use this tree structure
tree4 = {
    15: ['N',{"node": 14, "branch": 0.02}, {"node": 8, "branch": 1.1}],
    14: ['N',{"node": 13, "branch": 0.11}, {"node": 7, "branch": 1.2}],
    13: ['N',{"node": 12, "branch": 0.12}, {"node": 6, "branch": 0.8}],
    12: ['N',{"node": 11, "branch": 0.64}, {"node": 9, "branch": 0.11}],
    11: ['N',{"node": 3, "branch": 0.26}, {"node": 10, "branch": 0.24}],
    10: ['N',{"node": 4, "branch": 0.02}, {"node": 5, "branch": 0.08}],
    9: ['N',{"node": 1, "branch": 0.4}, {"node": 2, "branch": 0.6}]
}
for i in range(1, 9):
    tree4[i] = ['L']

# JC69 model to generate Q matrix
def JC69():
    Q = np.full((4,4),0.25)
    np.fill_diagonal(Q,-.75)
    Q = Q/0.75
    return Q

# Initialize rate matrix
Q = JC69()

# Answer
```

Recursion and the pruning algorthm

A recursive algorithm is one that calls itself. Suppose we have the following tree structure:



We can represent this tree using a dictionary, a list or a combination of the two. Using a dictionary of lists, each key is a numbered leaf or node in the tree and each value is a list containing three items: a letter indicating whether it is a node "N" or leaf "L", and two dictionaries for the left and right descenants. These dictionaries have two key-value pairs: the descendent *node* and the descendent *branch* length. In this dictionary, $L_1 - L_5$ is represented by keys $1 - 5$ and $N_6 - N_9$ is represented by keys $6 - 9$. Since leaves have no descendents, the list has only one item.

In [11]:

```
tree = {
    9: ['N',{'node': 6, 'branch': 0.1}, {'node': 8, 'branch': 0.2}],
    8: ['N',{'node': 3, 'branch': 0.2}, {'node': 7, 'branch': 0.5}],
    7: ['N',{'node': 4, 'branch': 0.3}, {'node': 5, 'branch': 0.4}],
    6: ['N',{'node': 1, 'branch': 0.2}, {'node': 2, 'branch': 0.1}],
    5: ['L'],
    4: ['L'],
    3: ['L'],
    2: ['L'],
    1: ['L']
}

# Print node 7, left descenant
print(tree[7][1]['node'])

# Print node 7, right descenant
print(tree[7][2]['node'])

# Print node 7, state (leaf or node)
print(tree[7][0])

# Print node 7, left descenant branch length
print(tree[7][1]['branch'])
```

```
4
5
N
0.3
```

Lets use a recursion to print **all** the descenant leaves of a key (node/leaf) in the dictionary; so all the leaves of the left and right descendants.

A recursive function is one that calls itself and is particularly useful for obtaining information from the tree-like data structure but without knowing the actual topology of the tree.

In [12]:

```
def descendants(tree, node):
    if tree[node][0] == 'N':
        #print(tree[node][1]['node'])
        #print(tree[node][2]['node'])
        descendants(tree, tree[node][1]['node'])
        descendants(tree, tree[node][2]['node'])
    else:
        print(node)
    return(None)

# All leaves should be printed
print("Node 9:")
descendants(tree,9)
# Just those below 8
print("Node 8:")
descendants(tree,8)
# Input of leaf, just gives the leaf
print("Node 1:")
descendants(tree,1)
```

```
Node 9:
1
2
```

```
3
4
5
Node 8:
3
4
5
Node 1:
1
```

Question 2

Write a recursive function that calculates the sum of the branch lengths given a tree and node number. The function should return the sum of all branches leading from the node to all descendant leaves. The function should return 0 if a leaf is given and return an error message if the node/leaf doesn't exist.

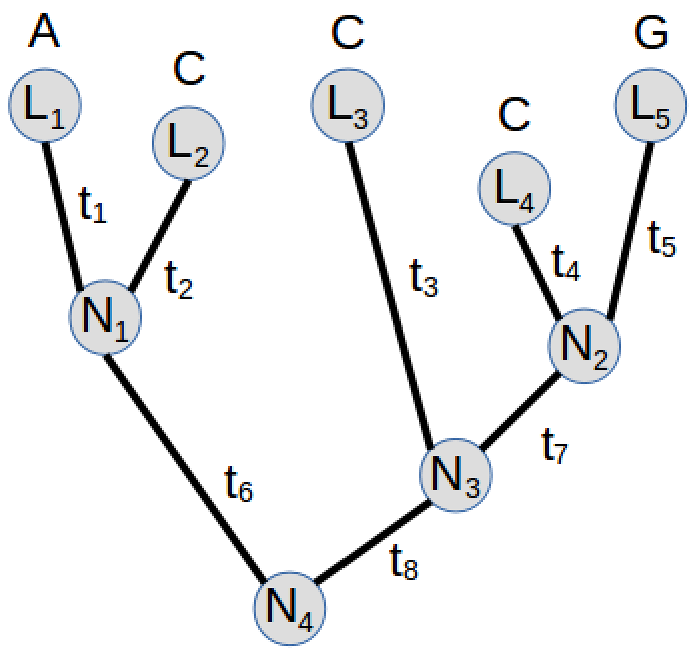
Use your function and print the output for nodes: 17, 15, 13, 9, 2

(4 points)

In [23]:
```python
tree3 = {
    15: ['N',{'node': 14, 'branch': 0.1}, {'node': 12, 'branch': 0.3}],
    14: ['N',{'node': 13, 'branch': 0.2}, {'node': 1, 'branch': 1.5}],
    13: ['N',{'node': 11, 'branch': 0.3}, {'node': 9, 'branch': 0.4}],
    12: ['N',{'node': 10, 'branch': 0.3}, {'node': 8, 'branch': 0.7}],
    11: ['N',{'node': 2, 'branch': 0.25}, {'node': 3, 'branch': 0.35}],
    10: ['N',{'node': 6, 'branch': 0.1}, {'node': 7, 'branch': 0.12}],
    9: ['N',{'node': 4, 'branch': 0.02}, {'node': 5, 'branch': 0.08}],
    8: ['L'],
    7: ['L'],
    6: ['L'],
    5: ['L'],
    4: ['L'],
    3: ['L'],
    2: ['L'],
    1: ['L']
}

# Answer
```

Felsenstein's pruning algorithm

Consider a tree with observed leaves L_1- L_5, unknown nodes N_1- N_4 and unknown branch lengths t_1- t_8:



To find the probability of the observed states $L1 - L5$ we need to consider all 4 possible states `[A,G,C,T]` at each of the nodes along with the branch lengths. Assuming independence among lineages, the probability of a state at each node can be **recursively** calculated using the states of the descendant node or leaf and the branch length. This recursive algorithm is known as **Felsenstein's pruning algorithm**.

The key insight of the pruning algorithm is that once probabilities of observing A, G, C, T at Node 1 are calculated, they can be recursively used to calculate subsequent probabilities, e.g. at Node 4. This results in the general recursion formula for the pruning algorithm:

$$L_k(s) = \left(\sum_x P(x|s, t_l) L_l(x) \right) \left(\sum_y P(y|s, t_m) L_m(y) \right) \tag{3}$$

where the likelihood of state s at node k is a product of the likelihoods at descendant nodes l and m having states x and y transitioning to s after time t_l and t_m, respectively, summed over all possible descendant states. The equation is a recursion since evaluation of $L_k(s)$ requires evaluation of two other calls of the same likelihood equation $L_l(x)$ and $L_m(y)$. The recursion ends when the tips are reached since the states at the tips are observed. The posterior probability of the overall tree is the likelihood of the ancestral ancestral node (Node 4), weighted by their prior probabilities, equal to the stationary frequencies of each nucleotide, π_x:

$$L = \sum_x \pi_x L_{anc}(x) \tag{4}$$

The posterior probability and likelihood are closely related as they differ in whether the prior is included. In maximum likelihood one typically uses a uniform prior and so the prior isn't part of the equation. If we are not using a JC69 model but rather a HKY85 model our priors are the expected (equilibrium) frequencies of A, G, C, T.

Lets make a function to encode this algorithm.

In [14]:

```python
# Define the observed leaf states and the tree
site=["A","C","C","C","G"]
tree = {
    9: ['N',{'node': 6, 'branch': 0.1}, {'node': 8, 'branch': 0.5}],
    8: ['N',{'node': 3, 'branch': 0.75}, {'node': 7, 'branch': 0.1}],
    7: ['N',{'node': 4, 'branch': 0.1}, {'node': 5, 'branch': 0.15}],
    6: ['N',{'node': 1, 'branch': 0.5}, {'node': 2, 'branch': 0.25}],
    5: ['L'],
    4: ['L'],
    3: ['L'],
    2: ['L'],
    1: ['L']
}
# To make a recursion, we also need to keep track of the node likelihoods for ea
# Doing so using a dictionary of lists (length 4 for each nucleotide) makes sens
# Lets initialize this to zero
node_likelihoods = {}
for node in range(9):
    node_likelihoods[node+1] = np.array([0,0,0,0])
# We should also initialize the leaf node likelihoods. Since they are observed t
nuc_map = {
    "A": 0,
    "G": 1,
    "C": 2,
    "T": 3
}
for i, nuc in enumerate( site ):
    node_likelihoods[i+1][nuc_map[nuc]] = 1

# Check our initiation
for node in node_likelihoods:
    print(node, node_likelihoods[node])
```

```
1 [1 0 0 0]
2 [0 0 1 0]
3 [0 0 1 0]
4 [0 0 1 0]
5 [0 1 0 0]
6 [0 0 0 0]
7 [0 0 0 0]
8 [0 0 0 0]
9 [0 0 0 0]
```

In [15]:

```python
# Recursive function that returns the likelihood for any node
def likelihood(node, Q, node_likelihoods, tree):
    if (tree[node][0] == 'L'):
        return node_likelihoods[node]
    else:
        # Find left and right descendants
        left_descendant = tree[node][1]["node"]
        right_descendant = tree[node][2]["node"]
        # Recursion to calculate probabilities at descendant nodes
        posterior_left = likelihood(left_descendant, Q, node_likelihoods, tree)
        posterior_right = likelihood(right_descendant, Q, node_likelihoods, tree
        # Calculate the likelihood at the given node
        P_left = expm(tree[node][1]["branch"]*Q)
        P_right = expm(tree[node][2]["branch"]*Q)
        likelihood_left = P_left.dot(posterior_left)
        likelihood_right = P_right.dot(posterior_right)
        node_likelihoods[node] = likelihood_left*likelihood_right
        return node_likelihoods[node]

# Initialize rate matrix
Q = JC69()
# Lets check and see if it matches our earlier calculation in the lab for Node 9

print( likelihood(9, Q, node_likelihoods, tree) )
print( "Earlier PN4: ")
print( PN4)
```

```
[1.48460804e-04 6.44278661e-05 1.18700233e-03 4.17931589e-05]
Earlier PN4:
[1.48460804e-04 6.44278661e-05 1.18700233e-03 4.17931589e-05]
```

Lets put it all together into a single function: `pruning` that will take a tree and nucleotide sites for the leaves and calculate the likelihood of each base in the ancestral node of the entire tree using the JC69 model.

In [16]:

```
def pruning(tree, site):
    node_likelihoods = {}
    total_nodes = 2*len(site)-1
    for node in range(total_nodes):
        node_likelihoods[node+1] = np.array([0,0,0,0])
    nuc_map = {
        "A": 0,
        "G": 1,
        "C": 2,
        "T": 3
    }
    for i, nuc in enumerate( site ):
        node_likelihoods[i+1][nuc_map[nuc]] = 1
    Q = JC69()
    return likelihood(total_nodes, Q, node_likelihoods, tree)
print( pruning(tree, site) )
```

```
[1.48460804e-04 6.44278661e-05 1.18700233e-03 4.17931589e-05]
```

Question 3

Calculate the most likely ancestral sequence using the five sequences present in SSA_regions.fasta using the pruning function and tree above. Print the most likely base and the likelihoods of A,G,C,T at each site.

(4 points)

In [22]:

```
tree = {
    9: ['N',{'node': 6, 'branch': 0.1}, {'node': 8, 'branch': 0.5}],
    8: ['N',{'node': 3, 'branch': 0.75}, {'node': 7, 'branch': 0.1}],
    7: ['N',{'node': 4, 'branch': 0.1}, {'node': 5, 'branch': 0.15}],
    6: ['N',{'node': 1, 'branch': 0.5}, {'node': 2, 'branch': 0.25}],
    5: ['L'],
    4: ['L'],
    3: ['L'],
    2: ['L'],
    1: ['L']
}
# Answer
```

Question 4

The file **SARS-CoV-2.fasta** contains five genomes of SARS-CoV-2: the ancestor (Wuhan, the first sequenced genome), and representatives of four common variants (Alpha, Delta, Gamma, Omicron).

Evolutionary rates and the most evolved lineages can be inferred through comparison of each variant genome to the Wuhan ancestor. Calculate the number of differences between each variant genome sequence and the ancestor (Wuhan). Print the name of the variant and the number of differences. Ignore gaps and Ns when comparing two sequences.

The relationship amoung these variants can be inferred using parsimony informative sites. However, positive selection can also result in the same mutation occurring on two independent lineages. (We won't distinguish between these, but we can find the relevant sites). Find all positions where two or more of the variant genomes differ from the ancestor. For these positions, print the position in the alignment and the nucleotides present in each of the five genomes. Do not include differences caused by 'N' or gaps, but if there are other nucleotide differences at the same site then they should be included.
For example, don't include sites like these:
`ANAAA` (only difference is N amoung the five genomes)
`G---G` (only difference is gap)
`GAGGG` (only one difference)

But include sites like these:
`AGGAA` (two differences)
`A-GGA` (two difference not including gap)
`AGGGG` (if Wuhan is 'A' then there are four differences and this site should be included)

Based on the output of shared sites, which variant does omicron have the most shared changes with?

(4 points)

In [20]:
```
# Answer
```

Question 5

Find all shared gaps between the SARS-CoV-2 genomes, ie where two or more variant genomes have an insertion or deletion in comparison to Wuhan. Print the position in the alignment and the sequence present in each genome for these positions. Do not print gaps at the beginning or end of the genomes since these are are just regions that weren't sequenced and aren't true insertion/deletion events.

(4 points)

In [21]:

```
# Answer
```

In [ ]:

```
```