Question 1

```
!pip install Bio
```

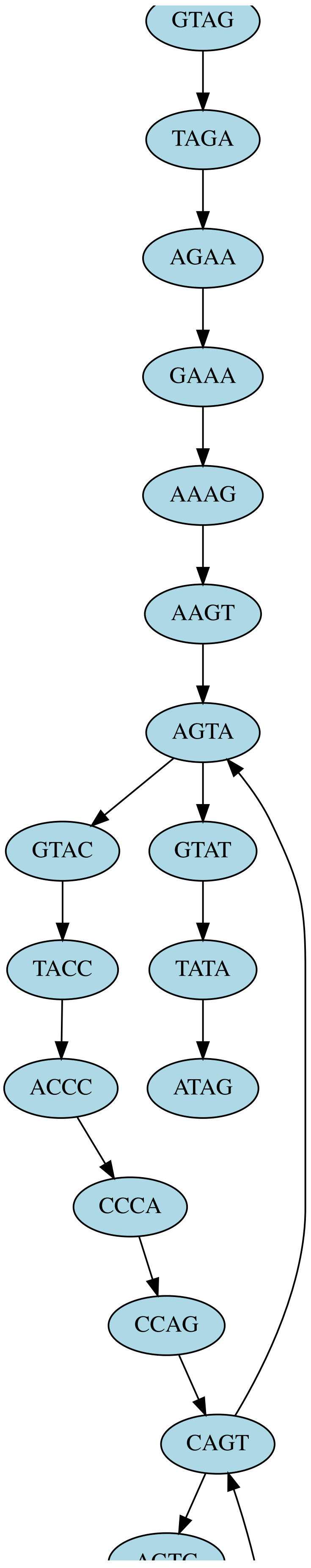
```
from Bio import AlignIO
from Bio import pairwise2

# Function to identify mutations at each position in a multiple sequence alignment
def identify_mutations_at_positions(alignment):
    # Create an empty dictionary to store mutations
    mutations = {}

    # Iterate through columns in the alignment
    for i in range(len(alignment[0])):
        # Extract the i-th column from the alignment
        column = alignment[:, i]

        # Check if there are multiple unique characters (amino acids) in the column
        if len(set(column)) > 1:
            # Store the position (i + 1) and the column in the mutations dictionary
            mutations[i + 1] = column

    # Return the dictionary of mutations
    return mutations

# Function to calculate the percent identities between sequences in the alignment
def calculate_percent_identities(alignment):
    # Nested function to calculate the percent identity between two sequences
    def calculate_percent_identity(seq1, seq2):
        # Count matching amino acids and total non-gap positions
        matches = sum(aa1 == aa2 for aa1, aa2 in zip(seq1, seq2) if aa1 != '-' and aa2 != '-')
        total = sum(1 for aa1, aa2 in zip(seq1, seq2) if aa1 != '-' and aa2 != '-')

        # Calculate and return the percent identity, handling cases with no valid positions
        return (matches / total) * 100 if total > 0 else 0

    # Create a dictionary to store percent identities between sequence pairs
    percent_identities = {}

    # Get the number of sequences in the alignment
    num_sequences = len(alignment)

    # Iterate through all pairs of sequences in the alignment
    for i in range(num_sequences):
        for j in range(i + 1, num_sequences):
            # Calculate the percent identity between the i-th and j-th sequences
            identity = calculate_percent_identity(alignment[i].seq, alignment[j].seq)

            # Store the percent identity in the dictionary using a tuple of sequence IDs as the key
            percent_identities[(alignment[i].id, alignment[j].id)] = identity

    # Return the dictionary of percent identities
    return percent_identities


def Q1(file_path):
    #   accession numbers
    # GenBank: MT291827.1
    # GenBank: MZ437368.1
    # GenBank: MN908947.3

    # Load the .aln file
    alignment = AlignIO.read(file_path, "clustal")

    # Identify mutations at each position
    mutations = identify_mutations_at_positions(alignment)

    # Calculate percent identity between the sequences
    percent_identities = calculate_percent_identities(alignment)

    #printing the results
    print("Mutations at each position:")
    for position, bases in mutations.items():
        print(f"Position {position}: {bases}")

    print("\nPercent identities between sequences:")
    for pair, identity in percent_identities.items():
        print(f"{pair}: {identity:.2f}%")
    # Return the results as a tuple
    return mutations, percent_identities


# Replace 'your_sequences.aln' with your actual file path
file_path = 'clustalw.aln'
Q1(file_path)
```

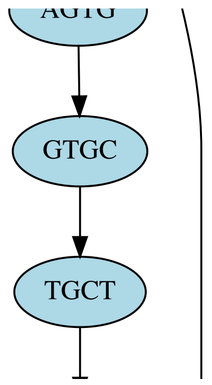
```
Mutations at each position:
Position 1: A--
Position 2: T--
Position 3: T-T
Position 4: A-A
Position 5: A-A
Position 6: A-A
Position 7: G-G
Position 8: G-G
Position 9: T-T
Position 10: T-T
Position 11: T-T
Position 12: A-A
Position 13: T-T
Position 14: A-A
Position 15: C-C
Position 16: C-C
Position 17: T-T
Position 18: T-T
Position 19: C-C
Position 20: C-C
Position 21: C-C
Position 22: A-A
Position 23: G-G
Position 24: G-G
Position 25: T-T
Position 26: A-A
Position 27: A-A
Position 28: C-C
Position 29: A-A
Position 30: A-A
Position 31: A-A
Position 32: C-C
Position 33: C-C
Position 34: A-A
Position 35: A-A
Position 36: C-C
Position 37: C-C
Position 210: GGT
Position 241: CCT
Position 1191: CCT
Position 1267: CCT
Position 3037: CCT
Position 5184: CCT
Position 6539: CCT
Position 7113: CCT
Position 9891: CCT
Position 11418: TTC
Position 12946: TTC
Position 14262: CCT
Position 14408: CCT
Position 15451: GGA
Position 16466: CCT
Position 20262: AAG
Position 20320: CCT
Position 21618: CCG
Position 21987: GGA
Position 22029: AA-
```

Mutations at Each Position: Positions 1 to 37 have various mutations, but without knowledge of the specific amino acids involved and their functional significance, it's challenging to assess their pathogenic impact. Positions 210, 241, 1191, 1267, 3037, 5184, 6539, 7113, 9891, 11418, 12946, 14262, 14408, 15451, 16466, 20262, 20320, 21618, 21987, and 22029 onwards also have mutations. Further analysis is needed to determine the effect of these mutations on protein function or pathogenesis. Some positions have insertions (e.g., 22029, 22030, etc.), which can potentially disrupt protein structure and function. The pathogenic impact depends on the specific amino acid changes, their location in the protein, and the functional regions of the protein. In-depth analysis is required to assess their significance. Percent Identities Between Sequences: The percent identity values between sequences ('MN908947.3', 'MT291827.1') and ('MN908947.3', 'MZ437368.1') are very high, indicating that these sequences are closely related and share a high degree of similarity. High percent identity suggests that these sequences are likely from the same or very closely related strains or variants. The high similarity between these sequences may indicate limited evolutionary divergence, which can impact pathogenicity and transmission dynamics. For a more detailed analysis of the pathogenic impact, it's important to know the specific amino acid changes, their location in the protein (e.g., spike protein, polymerase, etc.), and their potential effects on protein function, viral fitness, or antigenicity. Further study and analysis are required to draw specific conclusions regarding the pathogenicity of these mutations.

Question 2

```
from graphviz import Digraph, Source
from IPython.display import display

def Q2(st, k, n):
    # Define a class for a node in the De Bruijn graph
    class Node:
        def __init__(self, kmer):
            self.kmer = kmer
            self.in_deg = 0
            self.out_deg = 0
            self.ngbr = []

        def __str__(self):
            return self.kmer

        def __len__(self):
            return len(self.kmer)

    # Define a class for the De Bruijn Graph
    class Graph:
        def __init__(self, st, k, n):
            self.seq = st
            self.k = k
            self.olp = n

            # Create a list of k-mers and their left and right neighbors
            splits = [(st[i:i+k], st[i:i+k-1], st[i+1:i+k]) for i in range(len(st) - (k - 1))]
            self.graph = {}
            self.nodes = {}

            for i in splits:
                kit = i[0]
                nL = i[1]
                nR = i[2]
                Ln, Rn = None, None

                # Create nodes for left and right neighbors if they don't exist
                if nL not in self.nodes:
                    Ln = Node(nL)
                    self.nodes[nL] = Ln
                    self.graph[Ln] = set()
                else:
                    Ln = self.nodes[nL]

                if nR not in self.nodes:
                    Rn = Node(nR)
                    self.nodes[nR] = Rn
                    self.graph[Rn] = set()
                else:
                    Rn = self.nodes[nR]

                # Connect the left neighbor to the right neighbor
                self.graph[Ln].add(Rn)
                #increment in and out degrees of the node
                Rn.in_deg += 1
                Ln.out_deg += 1
            # Set the head and tail of the graph
            self.head = self.nodes[splits[0][1]]
            if splits[-1][2] not in self.nodes:
                self.nodes[splits[-1][2]] = Node(splits[-1][2])
            self.tail = self.nodes[splits[-1][2]]

        def eulerian(self):
            g = self.graph.copy()

            # add a temporary edge from head to tail
            g[self.tail] = {self.head}
            path = []

            # Define a function to explore the Eulerian path, similar to recursive dfs
            def explore(node):
                while len(g[node]) > 0:
                    next_node = g[node].pop()
                    explore(next_node)
                    path.append(node)

            #start from head and end at tail, finally reverse the path as tail is connected to the head and dfs g
            explore(self.head)
            path = path[::-1][:-1]
            start_idx = path.index(self.head)
            path = path[start_idx:] + path[:start_idx]

            path_str = [str(node) for node in path]
            path_output = path_str[0] + ''.join([node[-1] for node in path_str[1:]])
            # Create the result string
            result = f"Re-constructed Sequence: {path_output}\nEulerian Path: {' -> '.join(path_str)}"
            return result, path_output

        def to_dot(self):
            # Create a DOT representation of the De Bruijn graph
            dot = Digraph(format='png', node_attr={'style': 'filled', 'fillcolor': 'lightblue'})
            for node in self.graph:
                dot.node(node.kmer, label=node.kmer)
                for neighbor in self.graph[node]:
                    dot.edge(node.kmer, neighbor.kmer)
            return dot

    # Create an instance of the Graph class
    dbj_grp = Graph(st, k, n)
    # Generate the DOT representation of the graph
    dot = dbj_grp.to_dot()
    src = Source(dot.source)

    # Get the Eulerian path and display the result
    result, ret_val = dbj_grp.eulerian()
    print(result)
    print("The Following is the De Bruijn Graph")
    display(src)

    # returns the reconstructed answer from the De Bruijn Graph
    return ret_val

# Call the Q2 function with the specified parameters
Q2(st="TGTAGAAAGTACCCAGTGCTCAGTATAG", k=5, n=2)
```


GTAG
TAGA
AGAA
GAAA
AAAG
AAGT
AGTA
GTAC
GTAT
TACC
TATA
ACCC
ATAG
CCCA
CCAG
CAGT



In Question 2, we assume that De Bruijn Graph needs to be made only using the entire sequence, instead of splitting the given sequence into random reads. This allows for uniform splitting and overlapping across all edges. Hence, for required size of kmer = 5, we have k-1 mer nodes of size 4 which will always have an overlapping of 3 with the a unique k-1 mer if an edge is created. Since the problem statement asks for a minimum overlapp of 2, overlapp of 3 therefore satisfies the requirements.

It is also assumed that the graph generated is a balanced directed graph, to make an Eulerian Walk possible.

Question 3

```
def Q3(inp_seq):
    def bwt_fnc(text):
        rotations = [text[i:] + text[:i] for i in range(len(text))]  # Generate all cyclic rotations
        rotations.sort()  # Sort the rotations lexicographically
        bwt_text = ''.join(rot[-1] for rot in rotations)  # Extract the last character of each rotation
        return bwt_text

    # Construct the BWT
    bwt_result = bwt_fnc(inp_seq)

    # Retrun the answer
    return bwt_result
Q3("GCGTGCCTGGTCA$")
```

```
'ACTGGCT$TGCGGC'
```