1. Write a Python program that count the ATGC content of a given DNA sequence.

In [1]:
```
def count_nucleotides(dna_sequence):
    counts = {'A': 0, 'C': 0, 'G': 0, 'T': 0}
    for nucleotide in dna_sequence:
        counts[nucleotide] += 1
    return counts

# Example DNA sequence
dna_sequence = "AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGT"
print(f"Nucleotide Counts: {count_nucleotides(dna_sequence)}")
```

```
Nucleotide Counts: {'A': 8, 'C': 9, 'G': 9, 'T': 15}
```

2. Write a Python function that returns the complement of a DNA strand. A complements T, and C complements G.

In [2]:
```
def complement_dna(dna_sequence):
    complement = {'A': 'T', 'T': 'A', 'C': 'G', 'G': 'C'}
    return ''.join([complement[base] for base in dna_sequence])

# Example DNA sequence
dna_sequence = "AAGCT"
print(f"Complement: {complement_dna(dna_sequence)}")
```

```
Complement: TTCGA
```

3.Write a Python program that converts a DNA sequence into an RNA sequence by replacing all occurrences of "T" with "U".

In [3]:
```
def dna_to_rna(dna_sequence):
    return dna_sequence.replace('T', 'U')

# Example DNA sequence
dna_sequence = "AAGCT"
print(f"RNA: {dna_to_rna(dna_sequence)}")
```

```
RNA: AAGCU
```

4.Given an RNA sequence, write a Python program that translates the RNA into its protein sequence, using the standard genetic code.

In [4]:
```
def translate_rna(rna_sequence):
    genetic_code = {
        'AUG': 'Methionine', 'UUU': 'Phenylalanine', 'UUC': 'Phenylalanine',
        # Add remaining codons as needed
    }
    protein_sequence = [genetic_code[rna_sequence[i:i+3]] for i in range(0, len(rna_sequence)
    return protein_sequence

# Example RNA sequence
rna_sequence = "AUGUUUUUC"
print(f"Protein: {translate_rna(rna_sequence)}")
```

```
Protein: ['Methionine', 'Phenylalanine', 'Phenylalanine']
```

5. Write a Python function that finds all occurrences of a motif (substring) in a given DNA sequence and returns their positions (1-based indexing).

```
In [5]: def find_motif(dna_sequence, motif):
            positions = []
            start = 0
            while True:
                start = dna_sequence.find(motif, start) + 1
                if start > 0:
                    positions.append(start)
                else:
                    break
            return positions

        # Example
        dna_sequence = "GATATATGCATATACTT"
        motif = "ATAT"
        print(f"Positions: {find_motif(dna_sequence, motif)}")
```

```
Positions: [2, 4, 10]
```

6. Write a Python program that calculates the total mass of a protein sequence. Each amino acid has a specific mass (e.g., A=71.03711, C=103.00919, etc.).

```
In [6]: def calculate_protein_mass(protein_sequence):
            amino_acid_mass = {
                'A': 71.03711, 'C': 103.00919, # Add remaining amino acids as needed
            }
            return sum([amino_acid_mass[aa] for aa in protein_sequence])

        # Example Protein sequence
        protein_sequence = "AC"
        print(f"Total Mass: {calculate_protein_mass(protein_sequence)}")
```

```
Total Mass: 174.0463
```

7. Write a Python program that counts the number of occurrences of each nucleotide in a DNA sequence.

```
In [7]: def count_nucleotides(dna_sequence):
            counts = {'A': 0, 'C': 0, 'G': 0, 'T': 0}
            for nucleotide in dna_sequence:
                counts[nucleotide] += 1
            return counts

        # Example DNA sequence
        dna_sequence = "AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGT"
        print(f"Nucleotide Counts: {count_nucleotides(dna_sequence)}")
```

```
Nucleotide Counts: {'A': 8, 'C': 9, 'G': 9, 'T': 15}
```

8. Write a python code for finding most frequent k-mers in the DNA sequence

```
In [8]: def find_most_frequent_kmers(dna_sequence, k):
            # Dictionary to count k-mer occurrences
            kmers_count = {}

            # Slide the window of length k across the sequence
            for i in range(len(dna_sequence) - k + 1):
                # Extract the current k-mer
                kmer = dna_sequence[i:i+k]
                # Increment the count for this k-mer
                if kmer in kmers_count:
```

```
            kmers_count[kmer] += 1
        else:
            kmers_count[kmer] = 1

    # Find the maximum frequency of k-mers
    max_count = max(kmers_count.values())

    # Identify all k-mers that have the maximum frequency
    most_frequent = [kmer for kmer, count in kmers_count.items() if count == max_count]

    return most_frequent, max_count

# Example usage
dna_sequence = "ACGTTGCATGTCGCATGATGCATGAGAGCT"
k = 4
most_frequent_kmers, count = find_most_frequent_kmers(dna_sequence, k)
print(f"Most frequent {k}-mers: {most_frequent_kmers} with count: {count}")
```

```
Most frequent 4-mers: ['GCAT', 'CATG'] with count: 3
```

9. Write a python code for finding the reverse complement of a DNA Sequence

In [9]:
```
def reverse_complement(dna_sequence):
    # Define the complement of each nucleotide
    complement = {'A': 'T', 'T': 'A', 'C': 'G', 'G': 'C'}

    # Reverse the DNA sequence
    reversed_sequence = dna_sequence[::-1]

    # Replace each nucleotide with its complement
    reverse_complement_sequence = ''.join([complement[nucleotide] for nucleotide in reversed_

    return reverse_complement_sequence

# Example usage
dna_sequence = "ATCG"
print("Original sequence:", dna_sequence)
print("Reverse complement:", reverse_complement(dna_sequence))
```

```
Original sequence: ATCG
Reverse complement: CGAT
```

In [ ]:
```
10. Write a python code for designing primers for a DNA Sequence
```

In [10]:
```
def calculate_tm(seq):
    """Calculate the melting temperature of a DNA sequence."""
    a_t = seq.count('A') + seq.count('T')
    g_c = seq.count('G') + seq.count('C')
    tm = 2 * a_t + 4 * g_c
    return tm

def gc_content(seq):
    """Calculate the GC content of a DNA sequence."""
    g_c = seq.count('G') + seq.count('C')
    return (g_c / len(seq)) * 100

def design_primer(dna_sequence, primer_length=20):
    """Design primers for a given DNA sequence."""
    primers = []
    for i in range(len(dna_sequence) - primer_length + 1):
        primer = dna_sequence[i:i+primer_length]
        if 40 <= gc_content(primer) <= 60:
            primers.append(primer)
    return primers

# Example DNA sequence
dna_sequence = "ATGCTGCACTCGGTCGACTGATCGATCGTACGTCGATCG"

# Design primers
forward_primers = design_primer(dna_sequence)
reverse_primers = design_primer(dna_sequence[::-1])  # Reverse complement for simplicity

print("Forward Primers:", forward_primers)
print("Reverse Primers:", reverse_primers)
```

```
Forward Primers: ['ATGCTGCACTCGGTCGACTG', 'TGCTGCACTCGGTCGACTGA', 'GCTGCACTCGGTCGACTGAT', 'CTG
CACTCGGTCGACTGATC', 'TGCACTCGGTCGACTGATCG', 'GCACTCGGTCGACTGATCGA', 'CACTCGGTCGACTGATCGAT', 'A
CTCGGTCGACTGATCGATC', 'CTCGGTCGACTGATCGATCG', 'TCGGTCGACTGATCGATCGT', 'CGGTCGACTGATCGATCGTA',
'GGTCGACTGATCGATCGTAC', 'GTCGACTGATCGATCGTACG', 'TCGACTGATCGATCGTACGT', 'CGACTGATCGATCGTACGT
C', 'GACTGATCGATCGTACGTCG', 'ACTGATCGATCGTACGTCGA', 'CTGATCGATCGTACGTCGAT', 'TGATCGATCGTACGTCG
ATC', 'GATCGATCGTACGTCGATCG']
Reverse Primers: ['GCTAGCTGCATGCTAGCTAG', 'CTAGCTGCATGCTAGCTAGT', 'TAGCTGCATGCTAGCTAGTC', 'AGC
TGCATGCTAGCTAGTCA', 'GCTGCATGCTAGCTAGTCAG', 'CTGCATGCTAGCTAGTCAGC', 'TGCATGCTAGCTAGTCAGCT', 'G
CATGCTAGCTAGTCAGCTG', 'CATGCTAGCTAGTCAGCTGG', 'ATGCTAGCTAGTCAGCTGGC', 'TGCTAGCTAGTCAGCTGGCT',
'GCTAGCTAGTCAGCTGGCTC', 'CTAGCTAGTCAGCTGGCTCA', 'TAGCTAGTCAGCTGGCTCAC', 'AGCTAGTCAGCTGGCTCAC
G', 'GCTAGTCAGCTGGCTCACGT', 'CTAGTCAGCTGGCTCACGTC', 'TAGTCAGCTGGCTCACGTCG', 'AGTCAGCTGGCTCACGT
CGT', 'GTCAGCTGGCTCACGTCGTA']
```

11. Write a python code to evaluate the origin of replication

In [15]:
```
def find_pattern_in_dna(sequence, pattern):
    """
    Find all occurrences of a pattern in a DNA sequence.

    :param sequence: A string representing the DNA sequence.
    :param pattern: A string representing the pattern to search for.
    :return: A list of start indices where the pattern is found in the DNA sequence.
    """
    pattern_length = len(pattern)
    sequence_length = len(sequence)
    indices = []

    for i in range(sequence_length - pattern_length + 1):
        if sequence[i:i+pattern_length] == pattern:
            indices.append(i)

    return indices

# Example DNA sequence
dna_sequence = "AACATGACGATGCTACGATC"

# Pattern to search for (e.g., a simple motif associated with the origin of replication)
pattern = "ATG"

# Find and print the start indices of the pattern in the DNA sequence
start_indices = find_pattern_in_dna(dna_sequence, pattern)
print("Pattern found at indices:", start_indices)
```

```
Pattern found at indices: [3, 9]
```

12. Write a python code to evaluate the Open Reading Frames (ORF) of a nucleotide

In [16]:
```
def reverse_complement(dna_seq):
    complement = {'A': 'T', 'T': 'A', 'G': 'C', 'C': 'G'}
    return ''.join([complement[base] for base in reversed(dna_seq)])
```

```
def find_orfs(dna_seq):
    start_codon = 'ATG'
    stop_codons = ['TAA', 'TAG', 'TGA']
    orfs = []

    for strand, seq in [('+', dna_seq), ('-', reverse_complement(dna_seq))]:
        for frame in range(3):
            trans_start_pos = None
            for pos in range(frame, len(seq) - 2, 3):
                codon = seq[pos:pos + 3]
                if trans_start_pos is None and codon == start_codon:
                    trans_start_pos = pos
                elif trans_start_pos is not None and codon in stop_codons:
                    orfs.append((strand, frame, trans_start_pos, pos + 3))
                    trans_start_pos = None

    return orfs

def main(dna_seq):
    orfs = find_orfs(dna_seq)
    for orf in orfs:
        print(f"Strand: {orf[0]}, Frame: {orf[1]}, Start: {orf[2]}, End: {orf[3]}, Length: {o

dna_seq = "ATGAAAATGAAATAATAGTAA"
main(dna_seq)
```

```
Strand: +, Frame: 0, Start: 0, End: 15, Length: 15
```

In [ ]:
```
13. Write a python code to identify the number of RFLP markers in the DNA Sequence
```

In [17]:
```
def count_rflp_markers(dna_sequence, restriction_enzymes):
    """
    Count the number of RFLP markers in a given DNA sequence based on specified restriction e

    Parameters:
    dna_sequence (str): The DNA sequence to search.
    restriction_enzymes (dict): A dictionary of restriction enzymes and their cut sites.

    Returns:
    dict: A dictionary with the enzyme names as keys and the counts of their cut sites in the
    """
    rflp_marker_counts = {}
    for enzyme, cut_site in restriction_enzymes.items():
        count = dna_sequence.count(cut_site)
        rflp_marker_counts[enzyme] = count
    return rflp_marker_counts

# Example usage:
dna_sequence = "ATCGGATCCAGTCAAGCTTGAATTCGGATCCAAGCTTGGATCC"
restriction_enzymes = {
    "EcoRI": "GAATTC",
    "BamHI": "GGATCC",
    "HindIII": "AAGCTT"
}

rflp_marker_counts = count_rflp_markers(dna_sequence, restriction_enzymes)
print(rflp_marker_counts)
```

```
{'EcoRI': 1, 'BamHI': 3, 'HindIII': 2}
```

14. Write a python code to evaluate the frequency of mutations between two sequences

```
In [18]: def evaluate_mutations(sequence1, sequence2):
             """
             Evaluates the frequency of mutations between two sequences.

             Parameters:
             - sequence1: A string representing the first DNA sequence.
             - sequence2: A string representing the second DNA sequence.

             Returns:
             - A tuple containing the number of mutations and the mutation frequency as a percentage.
             """
             if len(sequence1) != len(sequence2):
                 raise ValueError("Sequences must be of equal length.")

             mutation_count = sum(1 for base1, base2 in zip(sequence1, sequence2) if base1 != base2)
             mutation_frequency = (mutation_count / len(sequence1)) * 100

             return mutation_count, mutation_frequency

         # Example usage
         sequence1 = "AGCT"
         sequence2 = "AGTT"
         mutation_count, mutation_frequency = evaluate_mutations(sequence1, sequence2)
         print(f"Number of mutations: {mutation_count}")
         print(f"Mutation frequency: {mutation_frequency:.2f}%")
```

```
Number of mutations: 1
Mutation frequency: 25.00%
```

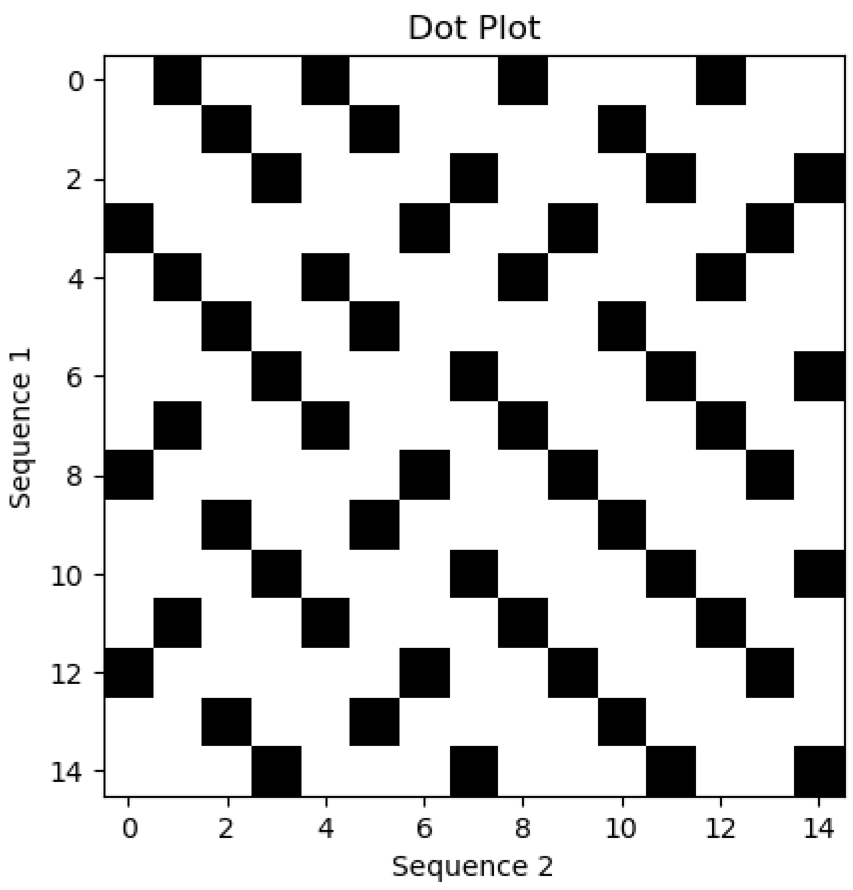
15. Write a python program to calculate the mass of the peptide

```
In [19]: def calculate_peptide_mass(sequence):
             # Average masses of the common amino acids in g/mol
             mass_table = {
                 'A': 71.0788, 'R': 156.1875, 'N': 114.1038, 'D': 115.0886,
                 'C': 103.1388, 'E': 129.1155, 'Q': 128.1307, 'G': 57.0519,
                 'H': 137.1411, 'I': 113.1594, 'L': 113.1594, 'K': 128.1741,
                 'M': 131.1926, 'F': 147.1766, 'P': 97.1167, 'S': 87.0782,
                 'T': 101.1051, 'W': 186.2132, 'Y': 163.1760, 'V': 99.1326
             }
             water_mass = 18.015

             # Calculate the total mass of the amino acids
             total_mass = sum(mass_table[aa] for aa in sequence)

             # Subtract the mass of water for each peptide bond formed
             peptide_bonds = len(sequence) - 1
             total_mass -= peptide_bonds * water_mass

             return total_mass

         # Example usage
         sequence = "AGCT"
         peptide_mass = calculate_peptide_mass(sequence)
         print(f"The mass of the peptide {sequence} is: {peptide_mass} g/mol")
```

```
The mass of the peptide AGCT is: 278.32959999999997 g/mol
```

16. Write a python code to perform dot plot between two nucleotides

In [21]:
```python
import matplotlib.pyplot as plt

def create_dot_plot(seq1, seq2):
    # Create a matrix initialized with zeros
    matrix = [[0]*len(seq2) for _ in range(len(seq1))]

    # Fill the matrix: 1 for matches, 0 for mismatches
    for i in range(len(seq1)):
        for j in range(len(seq2)):
            if seq1[i] == seq2[j]:
                matrix[i][j] = 1

    # Plot the dot plot
    plt.imshow(matrix, cmap='Greys', interpolation='none')
    plt.title('Dot Plot')
    plt.xlabel('Sequence 2')
    plt.ylabel('Sequence 1')
    plt.show()

# Example sequences
seq1 = 'ACTGACTAGCTAGCT'
seq2 = 'GACTACGTAGCTAGT'

create_dot_plot(seq1, seq2)
```



17. Write a python code to calculate the hamming distance between two nucleotides

In [22]:
```python
def hamming_distance(seq1, seq2):
    """Calculate the Hamming distance between two nucleotide sequences"""
    if len(seq1) != len(seq2):
        raise ValueError("Sequences must be of equal length")

    distance = sum(ch1 != ch2 for ch1, ch2 in zip(seq1, seq2))
    return distance
```

```
# Example usage
sequence1 = "AGCT"
sequence2 = "ACGT"

distance = hamming_distance(sequence1, sequence2)
print(f"The Hamming distance between the sequences is: {distance}")
```

```
The Hamming distance between the sequences is: 2
```

In [ ]:
```
18. write a pyton code to calculate the levinstein distance between two nucleotides
```

In [23]:
```
def levenshtein_distance(seq1, seq2):
    """Calculate the Levenshtein distance between two sequences."""
    if len(seq1) < len(seq2):
        return levenshtein_distance(seq2, seq1)

    # If one of the sequences is empty, return the length of the other (all insertions)
    if len(seq2) == 0:
        return len(seq1)

    previous_row = range(len(seq2) + 1)
    for i, c1 in enumerate(seq1):
        current_row = [i + 1]
        for j, c2 in enumerate(seq2):
            insertions = previous_row[j + 1] + 1
            deletions = current_row[j] + 1
            substitutions = previous_row[j] + (c1 != c2)
            current_row.append(min(insertions, deletions, substitutions))
        previous_row = current_row

    return previous_row[-1]

# Example nucleotide sequences
nucleotide_seq1 = "AGCT"
nucleotide_seq2 = "ACGT"

# Calculate Levenshtein distance
distance = levenshtein_distance(nucleotide_seq1, nucleotide_seq2)
print(f"The Levenshtein distance between the two nucleotide sequences is: {distance}")
```

```
The Levenshtein distance between the two nucleotide sequences is: 2
```

19. Write a python code to perform global alignment between two nucleotides

In [26]:
```
def needleman_wunsch(seq1, seq2, match_score=1, mismatch_score=-1, gap_penalty=-2):
    m, n = len(seq1), len(seq2)
    # Initialize the score matrix
    score_matrix = [[0 for _ in range(n + 1)] for _ in range(m + 1)]
    # Initialize the traceback matrix
    traceback_matrix = [[0 for _ in range(n + 1)] for _ in range(m + 1)]

    # Fill the first row and column of the score matrix
    for i in range(m + 1):
        score_matrix[i][0] = i * gap_penalty
    for j in range(n + 1):
        score_matrix[0][j] = j * gap_penalty

    # Fill the score matrix
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            match = score_matrix[i-1][j-1] + (match_score if seq1[i-1] == seq2[j-1] else mism
            delete = score_matrix[i-1][j] + gap_penalty
            insert = score_matrix[i][j-1] + gap_penalty
            score_matrix[i][j] = max(match, delete, insert)
            # Track the direction of the maximum score for traceback
            if score_matrix[i][j] == match:
                traceback_matrix[i][j] = "diag"
            elif score_matrix[i][j] == delete:
                traceback_matrix[i][j] = "up"
            else:
                traceback_matrix[i][j] = "left"

    # Traceback
    align1, align2 = "", ""
    i, j = m, n
    while i > 0 or j > 0:
        if traceback_matrix[i][j] == "diag":
            align1 = seq1[i-1] + align1
            align2 = seq2[j-1] + align2
            i -= 1
            j -= 1
        elif traceback_matrix[i][j] == "up":
            align1 = seq1[i-1] + align1
            align2 = "-" + align2
            i -= 1
        else:  # left
            align1 = "-" + align1
            align2 = seq2[j-1] + align2
            j -= 1

    return align1, align2

# Example usage
seq1 = "GATTACA"
seq2 = "GCATGCU"
alignment = needleman_wunsch(seq1, seq2)
print("Alignment 1:", alignment[0])
print("Alignment 2:", alignment[1])
```

```
Alignment 1: GATTACA
Alignment 2: GCATGCU
```

20. Write a python code to perform local alignment between two nucleotide

In [ ]:

```
def smith_waterman(seq1, seq2, match_score=2, gap_cost=1):
    m, n = len(seq1), len(seq2)
    # Score matrix
    score = [[0 for _ in range(n+1)] for _ in range(m+1)]
    # Traceback matrix
    traceback = [[0 for _ in range(n+1)] for _ in range(m+1)]

    max_score = 0
    max_pos = None

    # Scoring
    for i in range(1, m+1):
        for j in range(1, n+1):
            match = score[i-1][j-1] + (match_score if seq1[i-1] == seq2[j-1] else -gap_cost)
            delete = score[i-1][j] - gap_cost
            insert = score[i][j-1] - gap_cost
            score[i][j] = max(0, match, delete, insert)
            if score[i][j] == match:
                traceback[i][j] = '↖'
            elif score[i][j] == delete:
                traceback[i][j] = '↑'
```

```python
            elif score[i][j] == insert:
                traceback[i][j] = '←'
            else:
                traceback[i][j] = None
            if score[i][j] >= max_score:
                max_score = score[i][j]
                max_pos = (i, j)

    # Traceback
    align1, align2 = '', ''
    i, j = max_pos
    while traceback[i][j] is not None:
        if traceback[i][j] == '↖':
            align1 = seq1[i-1] + align1
            align2 = seq2[j-1] + align2
            i -= 1
            j -= 1
        elif traceback[i][j] == '↑':
            align1 = seq1[i-1] + align1
            align2 = '-' + align2
            i -= 1
        elif traceback[i][j] == '←':
            align1 = '-' + align1
            align2 = seq2[j-1] + align2
            j -= 1

    return align1, align2, max_score

# Example usage
seq1 = "GATTACA"
seq2 = "GCATGCU"
align1, align2, score = smith_waterman(seq1, seq2)
print("Alignment 1:", align1)
print("Alignment 2:", align2)
print("Score:", score)
```