ABIN - Assignment 2

Group-16

```
!pip install biopython
```

```
from Bio import Entrez

# Define a list of accession numbers or relevant identifiers.
accession_numbers = ["LR721773.1", "GMB10646.1", "MBQ2671914.1"]

# Define a list of file names corresponding to the sequences.
sequence_file_names = ["q1", "q2_1", "q2_2"]

try:
    # Fetch the sequence from NCBI
    handle = Entrez.efetch(db="nucleotide", id=accession_numbers[0], rettype="fasta", retmode="text")
    fasta_data = handle.read()
    handle.close()

    # Save the FASTA data to a file
    with open(f"{sequence_file_names[0]}.fasta", "w") as fasta_file:
        fasta_file.write(fasta_data)

    print(f"FASTA file saved as {sequence_file_names[0]}.fasta")

except Exception as e:
    print(f"An error occurred: {str(e)}")

for i in range(1,len(accession_numbers)):
  try:
      handle = Entrez.efetch(db="protein", id=accession_numbers[i], rettype="fasta", retmode="text")
      fasta_data = handle.read()
      handle.close()

      # Save the FASTA data to a file
      with open(f"{sequence_file_names[i]}.fasta", "w") as fasta_file:
          fasta_file.write(fasta_data)

      print(f"FASTA file saved as {sequence_file_names[i]}.fasta")

  except Exception as e:
      print(f"An error occurred: {str(e)}")
```

Fetching FASTA files

This code uses the Biopython library to fetch sequence data (nucleotide and protein sequences) from the NCBI (National Center for Biotechnology Information) database and save it to FASTA files with specified names.

Here, we have saved three sequences:

For Q1 : Nucleotide sequence

Accession Number - LR721773.1 (967 bps)

For Q2, Q3 : Protein sequences

Accession Numbers - GMB10646.1 (1000 aa), MBQ2671914.1 (984 aa)

Methods and arguments:

Entrez.efetch(): This function is used to retrieve data from the NCBI database. In this case, it fetches a nucleotide sequence (db="nucleotide") with the specified accession number (id=accession_numbers[0]).

It requests the data in FASTA format (rettype="fasta") and text mode (retmode="text").

handle.read(): Reads the fetched data from the handle.

handle.close(): Closes the connection to the NCBI database.

The fetched FASTA data is saved to a file with the name "q1.fasta".

```
# Define the filenames for the Fasta file
```

```
fasta_file = "q1.fasta"

# Function to read a Fasta file and extract the sequence
def get_seq(filename):
    sequence = ""
    with open(filename, 'r') as file:
        for line in file:
            if line.startswith(">"):  # Header line, skip it
                continue
            sequence += line.strip()  # Concatenate sequence lines
    return sequence


get_seq(fasta_file)
```

```
'AAATGTTTATGGCTGATGCTTATTTTGCAGACACTGTGTGGTATGTGGGGCAAATAATTTTTATAGTTGCCA
TTTGTTCATTGGTTATAATAGTTGTAGTGGCATTTTTGGCAACTTTTAAATTGTGTATTCAACTTTGCGGTAT
GTGTAATACCTTAGTACTGTCCCCTTCTATTTATGTGTTTAATAGAGGTAGGCAGTTTTATGAGTTTTACAAT
GATGTAAAACCACCAGTTCTTGATGTGGATGACGTTTAGTTAATCCAAACATTATGAGTAGTATAACTACACC
```

Reading DNA Sequence from a FASTA File

In this code cell, we have a function get_seq that reads a DNA sequence from a FASTA file. Here's what the code does:

- It takes the filename of a FASTA file as input.
- It initializes an empty string called sequence to store the DNA sequence.
- The code reads the file line by line and skips the header line (lines starting with ">").
- The non-header lines (sequence lines) are concatenated to form the complete DNA sequence.
- Finally, the get_seq function returns the extracted DNA sequence. You can call this function with the appropriate filename to obtain the DNA sequence from a FASTA file.

```
fasta_files = [get_seq(f'{i}.fasta') for i in sequence_file_names]

for i in fasta_files:
  print(f"Sequence {fasta_files.index(i)+1} : {i}")
  print()
```

```
Sequence 1 : AAATGTTTATGGCTGATGCTTATTTTGCAGACACTGTGTGGTATGTGGGGCAAATAATTTTTATAGTTGCCATTTGTTCATTGGTTATAATAGTTGTAGTGGCATTTTTGGCAACTTT

Sequence 2 : MKFKKNIAFLLVILNIFICSFKPEVSAIRKIKNITLFNQVCSIPKEAHGFISTKQLQYKNTQVFHYIHAKSGANLVFQKNNNINKVIEFNFKTPPKDNTGVNHVLEHSLLEANQRYPG

Sequence 3 : MFKKPISRIIALVLTSLLTFNVLPLAKEAPKSQISVQNIGETVCGFKLEKKLNYNGTEVCYFIHEKSGAHAVVEKNSNKEKSFQIGFRTPAENDKGINHIIEHSVLNGSENYPYKNIM
```

Question 1

```
from collections import defaultdict

def count_kmers(fasta_file, k):
    # Initialize a dictionary to store k-mer counts
    kmer_counts = defaultdict(int)

    sequence = get_seq(fasta_file)

        # Process the last sequence in the file
    for i in range(len(sequence) - k + 1):
        kmer = sequence[i:i + k]
        kmer_counts[kmer] += 1

    return kmer_counts

# Example usage:
k = 3  # Replace with the desired k-mer length

fasta_file = 'q1.fasta'

kmer_counts = count_kmers(fasta_file, k)

total_kmers = 0

# Print the k-mer counts
for kmer, count in kmer_counts.items():
    print(f'K-mer: {kmer}, Count: {count}')
    total_kmers += count
```

```
print('Total kmers: ',total_kmers)
```

```
K-mer: TTA, Count: 35
K-mer: TAT, Count: 43
K-mer: TGG, Count: 24
K-mer: GGC, Count: 13
K-mer: GCT, Count: 10
K-mer: CTG, Count: 11
K-mer: TGA, Count: 17
K-mer: GAT, Count: 16
K-mer: TGC, Count: 13
K-mer: CTT, Count: 20
K-mer: ATT, Count: 30
K-mer: TTG, Count: 30
K-mer: GCA, Count: 11
K-mer: CAG, Count: 11
K-mer: AGA, Count: 8
K-mer: GAC, Count: 8
K-mer: ACA, Count: 12
K-mer: CAC, Count: 9
K-mer: ACT, Count: 19
K-mer: GTG, Count: 22
K-mer: GGT, Count: 16
K-mer: GTA, Count: 22
K-mer: GGG, Count: 5
K-mer: CAA, Count: 18
K-mer: ATA, Count: 28
K-mer: TAA, Count: 22
K-mer: TAG, Count: 17
K-mer: AGT, Count: 19
K-mer: GCC, Count: 7
K-mer: CCA, Count: 13
K-mer: CAT, Count: 15
K-mer: TTC, Count: 14
K-mer: TCA, Count: 17
K-mer: AAC, Count: 14
K-mer: GCG, Count: 5
K-mer: CGG, Count: 3
K-mer: TAC, Count: 17
K-mer: ACC, Count: 13
K-mer: CCT, Count: 7
K-mer: GTC, Count: 6
K-mer: TCC, Count: 4
K-mer: CCC, Count: 7
K-mer: TCT, Count: 11
K-mer: CTA, Count: 14
K-mer: GAG, Count: 6
K-mer: AGG, Count: 15
K-mer: GGA, Count: 13
K-mer: ACG, Count: 4
K-mer: CGT, Count: 6
K-mer: ATC, Count: 12
K-mer: AGC, Count: 5
K-mer: GAA, Count: 11
K-mer: AAG, Count: 13
K-mer: CCG, Count: 4
K-mer: CGC, Count: 2
K-mer: CGA, Count: 3
K-mer: CTC, Count: 1
K-mer: TCG, Count: 1
Total kmers:  965
```

we define a function count_kmers that takes a FASTA file and a k-mer length 'k' as input. The function initializes a dictionary to store k-mer counts and then processes the sequences within the FASTA file to count k-mers. The code also demonstrates how to use this function to count k-mers for a specific k-mer length ('k').

we have also provided an example of how to use the count_kmers function to count 3-mers within the DNA sequence from the FASTA file. The code then prints the counts of each 3-mer and calculates the total count of all 3-mers in the sequence.

Question 2

```
def preprocess(seq1, seq2):
    '''
    Initialises the 2d dp array for the tabulation base case
    '''
    dp = [[0]*(len(seq2)+1) for i in range(len(seq1)+1)] #initialise a 2d list which will store the global alignment values
    dp[0][0] = 0 #base case when both sequence end
    for x in range(1, len(seq1)+1): #base case when sequence 2 ends first and gaps need to be filled for sequence 1
        dp[x][0] = 1 + dp[x-1][0]
    for y in range(1, len(seq2)+1):#base case when sequence 1 ends first and gaps need to be filled for sequence 2
```

```
        dp[0][y] = 1 + dp[0][y-1]
    return dp

#example
seq1 = "PLEASANTLY" # sequence 1 hard coded
seq2 = "MEANLY"   # sequence 1 hard coded
preprocess (seq1, seq2)
```

```
[[0, 1, 2, 3, 4, 5, 6],
 [1, 0, 0, 0, 0, 0, 0],
 [2, 0, 0, 0, 0, 0, 0],
 [3, 0, 0, 0, 0, 0, 0],
 [4, 0, 0, 0, 0, 0, 0],
 [5, 0, 0, 0, 0, 0, 0],
 [6, 0, 0, 0, 0, 0, 0],
 [7, 0, 0, 0, 0, 0, 0],
 [8, 0, 0, 0, 0, 0, 0],
 [9, 0, 0, 0, 0, 0, 0],
 [10, 0, 0, 0, 0, 0, 0]]
```

Preprocessing Sequences

we have a function preprocess that initializes a 2D dynamic programming (DP) array dp. This array is used for global sequence alignment, which is a part of sequence comparison algorithms. Here's what the code does:

- seq1 and seq2 are the input sequences for which we want to compute the edit distance.
- The dp array is initialized as a 2D list with dimensions (len(seq1) + 1) x (len(seq2) + 1). This array will store alignment values.
- Base cases are set in the dp array:
    - dp[0][0] = 0 represents the case when both sequences are empty, and no operations are required.
    - The code then fills the DP array to handle cases when one sequence ends before the other, requiring gap insertions or deletions.
- The preprocess function returns the initialized DP array.

```
def edit_distance(seq1, seq2):
    '''
    Given two strings seq1 and seq2,
    the function returns the minimum number of operations(insertions, deletion and replacements)
    required to convert seq1 to seq2.
    '''
    dp = preprocess(seq1, seq2)
    for x in range(1, len(dp)):
        for y in range(1, len(dp[0])):
            if(seq1[x-1] == seq2[y-1]):# charecters match
                dp[x][y] = dp[x - 1][y - 1]
                # index filled by match value, as match is always prioritised over gaps

            else:# chareters do not match
                dp[x][y] = 1+min(dp[x - 1][y - 1],
                                 dp[x][y - 1],
                                 dp[x - 1][y])
                # selects between mismatch, gap in sequence 1 and gap in sequence 2 and selects the maximum value among the three
    return dp[-1][-1]

#example
seq1 = "PLEASANTLY" # sequence 1 hard coded
seq2 = "MEANLY"   # sequence 1 hard coded
edit_distance(seq1, seq2)
```

```
5
```

Calculating Edit Distance

we define a function edit_distance that calculates the edit distance between two input sequences, seq1 and seq2. Here's what the code does:

- The edit_distance function calls the preprocess function to initialize the DP array.
- It then iterates through the DP array, filling in values based on character matches or mismatches and gap insertions or deletions.
- If characters match, the value is filled from the diagonal (indicating a match operation).
- If characters do not match, the code selects the minimum value among three options: mismatch, gap insertion in seq1, or gap insertion in seq2.
- Finally, the function returns the edit distance, which is the value stored in the bottom-right cell of the DP array.

```
#actual protein fasta sequences of length 1000 aa

seq1 = get_seq('q2_1.fasta')
```

```
seq2 = get_seq('q2_2.fasta')

print(edit_distance(seq1,seq2))
```

```
692
```

Question 3

```
def calculate_alignment(s, t):
    # Lengths of the input strings
    m, n = len(s), len(t)

    # Create a matrix to store edit distances
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Initialize the first row and column
    for i in range(m + 1):
        dp[i][0] = i * -2  # Gap penalty for string s
    for j in range(n + 1):
        dp[0][j] = j * -2  # Gap penalty for string t

    # Fill in the matrix using dynamic programming
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            match = dp[i - 1][j - 1] + (1 if s[i - 1] == t[j - 1] else -1)  # Match or mismatch score
            delete = dp[i - 1][j] - 2  # Gap in string s
            insert = dp[i][j - 1] - 2  # Gap in string t
            dp[i][j] = max(match, delete, insert)

    # The final value in the matrix represents the edit distance
    edit_distance = dp[m][n]

    # Backtrack to find the optimal alignment (s' and t')
    s_aligned, t_aligned = "", ""
    i, j = m, n
    while i > 0 or j > 0:
        if i > 0 and dp[i][j] == dp[i - 1][j] - 2:
            s_aligned = s[i - 1] + s_aligned
            t_aligned = "-" + t_aligned
            i -= 1
        elif j > 0 and dp[i][j] == dp[i][j - 1] - 2:
            s_aligned = "-" + s_aligned
            t_aligned = t[j - 1] + t_aligned
            j -= 1
        else:
            s_aligned = s[i - 1] + s_aligned
            t_aligned = t[j - 1] + t_aligned
            i -= 1
            j -= 1

    return edit_distance, s_aligned, t_aligned

# Example usage:
s = "PRETTY"
t = "PRTTEIN"
edit_distance, s_aligned, t_aligned = calculate_alignment(s, t)
print(f"Edit Distance: {edit_distance}")
print(s_aligned)
print(t_aligned)

print() #gap

# Input protein strings
s = get_seq('q2_1.fasta')
t = get_seq('q2_2.fasta')

# Calculate edit distance and obtain optimal alignment
edit_distance, s_aligned, t_aligned = calculate_alignment(s, t)

# Print the results
print(f"Edit Distance: {edit_distance}")
print(s_aligned)
print(t_aligned)
```

```
Edit Distance: -2
PRETTY-
PRTTEIN
```

```
Edit Distance: -440
MKFKKNIAFL--LVILNIFICSFKPEVSAIRKIKNITLFNQVCSIPKEAHGFISTKQLQYKNTQVFHYIHAKSGANLVFQKNNNINKVIEFNFKTPPKDNTGVNHVLEHSLLEANQRYPGF---FKLHD-S
M-FKKPISRIIALV-LTSLLT-FNVLPLAKEAPKSQISV-QN--IGETVCGFKLEKKLNYNGTEVCYFIHEKSGAHAVVEKNSNKEKSFQIGFRTPAENDKGINHIIEHSVLNGSENYPYKNIMFELDNLS
```

We have written a Python function called calculate_alignment that computes the edit distance between two strings, s and t, using dynamic programming. The edit distance quantifies the minimum number of operations (insertions, deletions, substitutions) needed to transform one string into the other. Additionally, our code generates an optimal alignment of these two strings.

Here's how our code operates:

- Initially, we calculate the lengths of the input strings s and t and store these lengths in m and n, respectively.
- To keep track of the edit distances for different substrings of s and t, we create a matrix called dp.
- We set up the first row and column of the matrix to represent gap penalties for string s and string t, respectively.
- The heart of our algorithm lies in dynamic programming. We fill in the dp matrix by considering match scores, mismatch penalties, and gap penalties.
- The value in the last cell of the matrix, dp[m][n], signifies the edit distance between the entire strings s and t.
- To discover the optimal alignment, we engage in backtracking, commencing from the bottom-right corner of the matrix. Depending on whether the value in the current cell originated from a match, deletion, or insertion, we construct the aligned versions of s and t.
- Our function delivers three results: the edit distance and the aligned versions of s and t.

Scoring scheme

- Match : + 1
- Mismatch : -1
- Gap : -2