BIOQUANTS - GENOME ASSEMBLY ASSIGNMENT

Author - Elisabetta Roviera (s328422)

Date of exam - June 9, 2025

Repository - https://github.com/elisabettaroviera/Bioquants. It contains the homework delivery, the notebook, the dataset and this report.

In this report, the functions required to complete **Assignment #6 (Python genome assembly)** are described.

Here is the proposed structure.

- **Overview**. A concise introduction and brief description of the problem addressed.
- **Tasks / Implementation Steps**. This section details each specific subproblem or function required for the solution. For each task, it includes:
  1. A description of the function to be implemented;
  2. The implementation code for the function;
  3. Tests to verify the function's correctness.
- **Final Demonstration**. A concluding section presenting an example of how to use all the implemented functions together to produce the final result (e.g., the assembled genome sequence).

1 Overview

This assignment focuses on introducing an approach to the task of genome assembly. The process of genome assembly involves taking numerous short DNA fragments, known as reads, generated by sequencing machines, and piecing them together to form much longer continuous sequences, ultimately representing chromosomes. This is achieved by identifying regions where reads overlap and determining their correct order to reconstruct the original genomic sequence. The task will be to process the provided sequence reads, detect overlaps between them, establish their proper arrangement, and finally reconstruct the genomic sequence based on these overlapping segments.

For the purpose of this task, I will operate under two significant simplifying assumptions:

1. I assume the **absence of sequencing errors**, meaning any overlaps between reads are exact matches.

2. I assume that **no read is entirely contained** within another read.

Throughout this exercise, I will develop functions designed to collectively address the genome sequence assembly problem. The sequence reads are located in the file named genome-assembly.txt. Each line in this file corresponds to a single read. The first piece of information on each line represents the read's identifier or name, and the second part contains the actual read sequence.

2 Problem 1 - Read and Parse Input Data

The initial step of this task is to acquire the input data. To accomplish this, I am asked to develop a function named readDataFromFile that accepts a single argument: a string representing the fileName. The required output of this function is a dictionary. Within this dictionary, each key should correspond to a read name, and its associated value should be the respective read sequence. It is essential that both the keys and the values in the returned dictionary are of the string data type.

[1]:
```
# Write a function `readDataFromFile(fileName)` that reads a file and
# returns a dictionary mapping read names to sequences
def readDataFromFile(fileName):
    reads = {} # Creating the dictionary

    with open(fileName, 'r') as f: # Opening text file
        for line in f:
            name, seq = line.strip().split(':') # Reading the file and splitting␣
 ↪in key-value
            reads[name] = seq

    return reads
```

[2]:
```
# Test
fileName = 'genome-assembly.txt'
reads = readDataFromFile(fileName)

reads
```

[2]:
```
{'1': 'GGCTCCCCACGGGGTACCCATAACTTGACAGTAGATCTCGTCCAGACCCCTAGC',
 '3': 'GTCTTCAGTAGAAAATTGTTTTTTTCTTCCAAGAGGTCGGAGTCGTGAACACATCAGT',
 '2': 'CTTTACCCGGAAGAGCGGGACGCTGCCCTGCGCGATTCCAGGCTCCCCACGGG',
 '5': 'CGATTCCAGGCTCCCCACGGGGTACCCATAACTTGACAGTAGATCTC',
 '4': 'TGCGAGGGAAGTGAAGTATTTGACCCTTTACCCGGAAGAGCG',
 '6': 'TGACAGTAGATCTCGTCCAGACCCCTAGCTGGTACGTCTTCAGTAGAAAATTGTTTTTTTCTTCCAAGAGGTC
GGAGT'}
```

3 Problem 2 - Mean Read Length

Given that the number of reads is often too large for manual examination, it is necessary to find an automated way to analyze their properties. Specifically, the objective is to determine the average length of these sequences. A function needs to be created, named meanLength(fileName), which accepts a single string argument representing the name of an input file formatted like the one previously described. This function is required to compute the mean sequence length and return the result as a floating-point number.

[3]:
```
# Write a function `meanLength(fileName)` that returns the mean
# sequence length as a float
```

```
def meanLength(fileName):
    reads = readDataFromFile(fileName) # Calling the function to read the file

    if not reads: # Being sure that the file is read in a proper way
        return "File not read in a proper way"

    total_length = sum(len(seq) for seq in reads.values())
    mean = total_length / len(reads) # Computing the mean

    return mean
```

[4]:
```
# Test
mean_length = meanLength(fileName)

mean_length
```

[4]: 55.333333333333336

4 Problem 3 - Compute Overlap Between Two Reads

The subsequent step involves identifying which reads overlap each other. To facilitate this, a function is required that takes two read sequences and computes their overlap. Given the characteristics of the input data, it is known that none of the reads are completely nested within another. Consequently, when two reads overlap, the overlap occurs between the 3' end of the read designated as 'left' and the 5' end of the read designated as 'right'. Furthermore, with the assumption of no sequencing errors, any overlap identified will represent a perfect sequence match. The approach to finding this overlap involves iterating through all possible overlap lengths, considering the specified 'left' and 'right' orientations. Within the iterative process, it is necessary to begin with the largest possible overlap length (equivalent to min(len(left), len(right))) and progressively evaluate smaller overlap lengths until an exact match is discovered. A function is to be implemented with the signature def getOverlap(left, right): which accepts two string arguments, left and right, each containing a read sequence. The function must return the identified overlapping sequence as a string. In the case where no overlap exists between the two sequences, an empty string should be returned.

[5]:
```
# Write a function `getOverlap(left, right)` that returns the overlapping
# substring between the end of `left` and the start of `right`
def getOverlap(left, right):
    max_overlap = ''
    max_len = min(len(left), len(right))

    # Iterating from the maximum length down to 1
    # This ensures the longest overlap is found first
    for i in range(max_len, 0, -1):
        # Checking if the last 'i' chars of 'left' match the first 'i' chars of␣
 ↪'right'
        if left[-i:] == right[:i]:
            max_overlap = left[-i:]
```

```
            break

    return max_overlap
```

[6]:
```
# Test e.g. s1 and s2
s1 = 'CGATTCCAGGCTCCCCACGGGGTACCCATAACTTGACAGTAGATCTC'
s2= 'GGCTCCCCACGGGGTACCCATAACTTGACAGTAGATCTCGTCCAGACCCCTAGC'

overlaps1s2 = getOverlap(s1, s2)
overlaps2s1 = getOverlap(s2, s1)

print(f'getOverlap(s1, s2) → {overlaps1s2}')
print(f'getOverlap(s2, s1) → {overlaps2s1}')
```

```
getOverlap(s1, s2) → GGCTCCCCACGGGGTACCCATAACTTGACAGTAGATCTC
getOverlap(s2, s1) → C
```

5 Problem 4 - Compute All Pairwise Overlaps

With the capability to evaluate the overlap between two reads in a specific orientation now established, the subsequent task is to compute overlaps between all possible pairs of reads. This computation must consider both left-right and right-left orientations. A function is to be implemented with the signature def getAllOverlaps(reads): that accepts one argument: a dictionary named reads, formatted identically to the output produced by the readDataFromFile function. The required return value of this function is a dictionary of dictionaries. This nested dictionary structure should store the number of overlapping bases for a pair of reads in a given orientation. Overlaps of a read with itself are not meaningful in this context and must be excluded from the results. As an illustration, if the overlap size between read '2' placed in the left position and read '5' in the right position is found to be 21 bases, then the resulting dictionary, if named d, should satisfy the condition d['2']['5'] == 21.

[7]:
```
# Write a function `getAllOverlaps(reads)` that returns a nested dict
# of overlap lengths for each ordered read pair
def getAllOverlaps(reads):
    overlaps = {} # Creating the final dictionary

    # Iterating through each read (and its sequence)
    # as the potential 'left' read (read_a)
    for a, seq_a in reads.items():
        overlaps[a] = {}

        # Iterating through each read (and its sequence)
        # as the potential 'right' read (read_b)
        for b, seq_b in reads.items():
            # Skipping calculating overlap of a read with itself
            if a == b:
                continue
```

```
            # Getting the overlapping sequence between seq_a (end) and seq_b␣
↪(beginning)
            overlap_seq = getOverlap(seq_a, seq_b)

            # Storing the length of the overlap in the nested dictionary
            overlaps[a][b] = len(overlap_seq)

    return overlaps
```

[8]:
```
# Test
overlap_dict = getAllOverlaps(reads)
ovlp25 = overlap_dict['2']['5']

print(f'ex. d[2][5] = {ovlp25}')
overlap_dict
```

```
ex. d[2][5] = 21
```

[8]:
```
{'1': {'3': 0, '2': 1, '5': 1, '4': 0, '6': 29},
 '3': {'1': 0, '2': 0, '5': 0, '4': 1, '6': 1},
 '2': {'1': 13, '3': 1, '5': 21, '4': 0, '6': 0},
 '5': {'1': 39, '3': 0, '2': 1, '4': 0, '6': 14},
 '4': {'1': 1, '3': 1, '2': 17, '5': 2, '6': 0},
 '6': {'1': 0, '3': 43, '2': 0, '5': 0, '4': 1}}
```

6 Problem 5 - Pretty-Print Overlap Matrix

The dictionary structure produced by getAllOverlaps is not easily readable. To improve visualization of which read pairs overlap and in what orientations, it is desired to display this data in a clear, matrix-like format. A function is required with the signature def prettyPrint(overlaps): that accepts one argument: a dictionary named overlaps, containing the dictionary of dictionaries produced by the getAllOverlaps function. This function should not yield a return value; its sole purpose is to print the matrix representation, ensuring all columns are nicely aligned and right-justified. The first column is designated to list the names of the reads considered in the "left conformation." The top row is to contain the names of the reads considered in the "right conformation." The interior cells of the matrix will each contain the number of overlapping bases found for the corresponding left-right read pair. The cells situated along the main diagonal correspond to overlaps of a read with itself; these specific cells must be populated with dashes ('-').

[9]:
```
# Write a function `prettyPrint(overlaps)` that prints an aligned matrix
# of overlaps with dashes on the diagonal
def prettyPrint(overlaps):
    names = sorted(overlaps.keys(), key=lambda x: int(x))

    # Header
    print('   ' + ' '.join(f'{name:>3}' for name in names))
```

```
    # Iterating through each read name as the 'left' read (row label)
    for i in names:
        # Start building the current row with the 'left' read name
        row = [f'{i:>3}']

        # Iterating through each read name as the 'right' read (column label)
        for j in names:

            # Checking if the current cell is on the diagonal (read overlapping␣
↪itself)
            if i == j:
                cell = ' -' # Putting a dash '-' for self-overlaps

            else:
                # Getting the overlap length between read 'i' and read 'j'.
                # .get(j, 0) safely gets the value for key 'j' from overlaps[i],
                # returning 0 if the key 'j' doesn't exist in overlaps[i]
                cell = f'{overlaps[i].get(j, 0):>3}'

            # Add the formatted cell content to the current row list
            row.append(cell)

        print(' '.join(row)) # Print the completed row,
```

[10]:
```
# Test
prettyPrint(overlap_dict)
```

```
    1   2   3   4   5   6
1   -   1   0   0   1  29
2  13   -   1   0  21   0
3   0   0   -   1   0   1
4   1  17   1   -   2   0
5  39   1   0   0   -  14
6   0   0  43   1   0   -
```

7 Problem 6 - Find the First Read

With the overlap relationships between reads now established, it becomes possible to chain them together pair by pair from left to right to reconstruct the order in which they represent the genomic sequence. This process is typically achieved by starting with the initial (leftmost) read, identifying which subsequent read has the largest overlap at its right end, then taking that identified read and finding the read with the largest overlap to its right end, and continuing this sequence until the rightmost (last) read is reached. The immediate requirement in this process is to identify the first (leftmost) read, as this serves as the starting point. This initial read is identified as the one that exhibits a significant overlap (specifically, an overlap greater than 2 bases) only at its right end; in other words, it demonstrates a substantial overlap when positioned to the left of other

reads, but no significant overlap occurs when other reads are positioned to its left. Referencing the example output from prettyPrint previously, read '4' would be identified as the first read because its corresponding column (representing reads overlapping '4' on its right) shows no significant overlaps (none larger than two bases). A function is to be implemented with the signature def findFirstRead(overlaps): that accepts one argument, the dictionary of dictionaries named overlaps returned by the getAllOverlaps function. This function must yield a string containing the name of the identified first read.

[11]:
```
# Write a function `findFirstRead(overlaps)` to identify the read
# with no significant overlaps (>2) as the first (leftmost)
def findFirstRead(overlaps):
    reads = overlaps.keys()

    # Iterate through each read, considering it as a potential first read
    for candidate in reads:
        # Checking if all other reads 'a' have an overlap of 2 or less
        # when 'a' is on the left and the 'candidate' is on the right
        # This looks down the 'candidate' column in the overlap matrix,
        # excluding the diagonal (self-overlap)
        # .get(candidate, 0) safely handles cases where a pair might not be in␣
↪the dict
        if all(overlaps[a].get(candidate, 0) <= 2 for a in reads if a !=␣
↪candidate):
            # If the condition is true for a candidate, it means no other read
            # significantly overlaps its left end, so it's the first read
            return candidate

    return None
```

[12]:
```
# Test
first_read = findFirstRead(overlap_dict)

first_read
```

[12]: '4'

8 Problem 7 - Determine Read Order

Following the identification of the initial read, the correct ordering of the reads can be determined recursively. A list containing the read names arranged in the correct sequential order is desired as the output. This problem lends itself to a recursive solution, where the primary task can be decomposed into a recurring subproblem: finding the read that exhibits the largest overlap at the right end of the current read (this constitutes the recursive case). This process continues until the final (rightmost) read is reached, which is defined as not having any significant overlap (greater than 2 bases) at its right end (this constitutes the base case for the recursion). Within the recursive step, it is necessary to identify precisely which read comes next; that is, the read possessing the largest overlap to the right end of the current read. The overlaps dictionary is utilized for this purpose.

For example, if the current read is named '4', the dictionary overlaps['4'] provides information about overlaps with reads positioned to the right of read '4'. Therefore, to ascertain the name of the read with the largest overlap in such a scenario, a dedicated function is required that finds the key associated with the largest value within a given dictionary. As a preliminary step, a function with the signature def findKeyForLargestValue(d): is to be implemented. This function should accept a dictionary argument d and return the key linked to the dictionary's largest value. This helper function is then to be employed as a tool within the subsequent primary function. Implement the main function with the signature def findOrder(name, overlaps). This function accepts two arguments: a string name containing the name of the first read (identified in the previous task), and the dictionary argument overlaps returned by getAllOverlaps. The required return value is a list containing the names of the reads arranged in the correct order representing the genomic sequence.

[13]:
```
# Write `findKeyForLargestValue(d)` to find the key with the max value in `d`
def findKeyForLargestValue(d):
    # Returns the key corresponding to the largest value in dict d
    return max(d, key=d.get)
```

[14]:
```
# Test
d = overlap_dict['4']
second_read = findKeyForLargestValue(d)

second_read
```

[14]:
```
'2'
```

[15]:
```
# Write `findOrder(name, overlaps)` that returns the ordered list of reads
def findOrder(name, overlaps):
    # Given the first read name and overlaps dict, returns the list of reads in␣
 ↪order
    order = [name]
    current = name

    # Loop iteratively to find the next read based on overlaps
    while True:
        # Get the dictionary of overlaps from the current read to others on its␣
 ↪right
        next_overlaps = overlaps[current]
        # Filtering only significant overlaps >2
        significant = {r: o for r, o in next_overlaps.items() if o > 2}

        # Base case: If there are no significant overlaps from the current read,
        # it means we have reached the last read in the sequence
        if not significant:
            break

        # Recursive step (implemented iteratively):
```

```
        # Find the name of the read (key) that has the largest significant␣
↪overlap
        # with the current read using the helper function (assumed to exist)
        next_read = findKeyForLargestValue(significant)

        order.append(next_read) # Add the found 'next_read' to the order list

        # Set the 'next_read' as the new 'current' read for the next iteration
        current = next_read

    return order
```

[16]:
```
# Test
reads_order = findOrder(first_read,overlap_dict)

reads_order
```

[16]:
```
['4', '2', '5', '1', '6', '3']
```

9 Problem 8 - Assemble Genome

With the number of overlapping bases between reads and the correct order of the reads now determined, the genomic sequence can be reconstructed. A function is to be implemented with the signature def assembleGenome(readOrder, reads, overlaps). This function accepts three arguments: a list named readOrder containing the names of the reads in their correct sequential order, a dictionary named reads as returned by the readDataFromFile function, and a dictionary named overlaps as returned by the getAllOverlaps function. The required return value of this function is a single string containing the fully assembled genomic sequence.

[17]:
```
# Write `assembleGenome(readOrder, reads, overlaps)`
# to reconstruct the full genome string
def assembleGenome(readOrder, reads, overlaps):
    # Starting the genome sequence with the sequence of the first read in the␣
↪ordered list
    genome = reads[readOrder[0]]

    # Iterating through consecutive pairs of reads in the determined order
    # zip(readOrder, readOrder[1:]) creates pairs like (read1, read2), (read2,␣
↪read3), etc.
    for prev, curr in zip(readOrder, readOrder[1:]):
        # Getting the overlap length between the end of the 'prev' read
        # and the beginning of the 'curr' read
        o_len = overlaps[prev][curr]

        # Appending the non-overlapping part of the current read's sequence to␣
↪the genome
        # We slice the 'curr' read's sequence starting after the overlap length
```

```
        genome += reads[curr][o_len:]

    return genome
```

[18]:
```
# Test
genomic_sequence = assembleGenome(reads_order, reads, overlap_dict)
correct_sequence =␣
↪'TGCGAGGGAAGTGAAGTATTTGACCCTTTACCCGGAAGAGCGGGACGCTGCCCTGCGCGA'\
                  ␣
↪'TTCCAGGCTCCCCACGGGGTACCCATAACTTGACAGTAGATCTCGTCCAGACCCCTAGCT'\
                  ␣
↪'GGTACGTCTTCAGTAGAAAATTGTTTTTTTCTTCCAAGAGGTCGGAGTCGTGAACACATCAGT'

# Checking if the genomic_sequence is equal to the correct_sequence
genomic_sequence == correct_sequence
```

[18]:
```
True
```

10 Application Example

Upon the successful implementation of all the required functions, these can be utilized, as an example, in the following manner to produce the assembled genome.

[19]:
```
# Problem 1: Read and parse input data
print("### Problem 1: Read Data")

filename = 'genome-assembly.txt'

reads = readDataFromFile(filename)
print(f"reads (names → sequences):\n{reads}")
```

```
### Problem 1: Read Data
reads (names → sequences):
{'1': 'GGCTCCCCACGGGGTACCCATAACTTGACAGTAGATCTCGTCCAGACCCCTAGC', '3':
'GTCTTCAGTAGAAAATTGTTTTTTTCTTCCAAGAGGTCGGAGTCGTGAACACATCAGT', '2':
'CTTTACCCGGAAGAGCGGGACGCTGCCCTGCGCGATTCCAGGCTCCCCACGGG', '5':
'CGATTCCAGGCTCCCCACGGGGTACCCATAACTTGACAGTAGATCTC', '4':
'TGCGAGGGAAGTGAAGTATTTGACCCTTTACCCGGAAGAGCG', '6': 'TGACAGTAGATCTCGTCCAGACCCCTAG
CTGGTACGTCTTCAGTAGAAAATTGTTTTTTTCTTCCAAGAGGTCGGAGT'}
```

[20]:
```
# Problem 2: Compute mean read length
print("### Problem 2: Mean Read Length")

mean_len = meanLength(filename)
print(f"The mean_length of the sequence is: {mean_len:.2f}")
```

```
### Problem 2: Mean Read Length
The mean_length of the sequence is: 55.33
```

[21]:
```
# Problem 3: Compute overlap between two example reads
print("### Problem 3: Example Overlap")

# Using reads '1' and '2' as an example
s1 = reads['1']
s2 = reads['2']

overlap_1_2 = getOverlap(s1, s2)
overlap_2_1 = getOverlap(s2, s1)

print(f"s1 → {s1}")
print(f"s2 → {s2}")

print(f'getOverlap(s1, s2) → {overlap_1_2}, (length: {len(overlap_1_2)})')
print(f'getOverlap(s2, s1) → {overlap_2_1}, (length: {len(overlap_2_1)})')
```

```
### Problem 3: Example Overlap
s1 → GGCTCCCCACGGGGTACCCATAACTTGACAGTAGATCTCGTCCAGACCCCTAGC
s2 → CTTTACCCGGAAGAGCGGGACGCTGCCCTGCGCGATTCCAGGCTCCCCACGGG
getOverlap(s1, s2) → C, (length: 1)
getOverlap(s2, s1) → GGCTCCCCACGGG, (length: 13)
```

[22]:
```
# Problem 4: Compute all pairwise overlaps
print("### Problem 4: All Pairwise Overlaps (dict form)")

overlaps = getAllOverlaps(reads)
print(f"The overlaps dictionary is:\n{overlaps}")
```

```
### Problem 4: All Pairwise Overlaps (dict form)
The overlaps dictionary is:
{'1': {'3': 0, '2': 1, '5': 1, '4': 0, '6': 29}, '3': {'1': 0, '2': 0, '5': 0,
'4': 1, '6': 1}, '2': {'1': 13, '3': 1, '5': 21, '4': 0, '6': 0}, '5': {'1': 39,
'3': 0, '2': 1, '4': 0, '6': 14}, '4': {'1': 1, '3': 1, '2': 17, '5': 2, '6':
0}, '6': {'1': 0, '3': 43, '2': 0, '5': 0, '4': 1}}
```

[23]:
```
# Problem 5: Pretty-print overlap matrix
print("### Problem 5: Overlap Matrix")

prettyPrint(overlaps)
```

```
### Problem 5: Overlap Matrix
     1   2   3   4   5   6
  1  -   1   0   0   1  29
  2  13  -   1   0  21   0
  3   0   0  -   1   0   1
  4   1  17   1  -   2   0
  5  39   1   0   0  -  14
  6   0   0  43   1   0  -
```

[24]:
```
# Problem 6: Find the first (leftmost) read
print("### Problem 6: First Read")

first_read = findFirstRead(overlaps)
print(f"first_read: {first_read}")
```

```
### Problem 6: First Read
first_read: 4
```

[25]:
```
# Problem 7: Determine the read order
print("### Problem 7: Read Order")

order = findOrder(first_read, overlaps)
print(f"order: {order}")
```

```
### Problem 7: Read Order
order: ['4', '2', '5', '1', '6', '3']
```

[26]:
```
# Problem 8: Assemble the genome
print("### Problem 8: Assembled Genome")
genome_sequence = assembleGenome(order, reads, overlaps)

print(f"The assembled_genome_sequence is: \n{genome_sequence}")
```

```
### Problem 8: Assembled Genome
The assembled_genome_sequence is:
TGCGAGGGAAGTGAAGTATTTGACCCTTTACCCGGAAGAGCGGGACGCTGCCCTGCGCGATTCCAGGCTCCCCACGGGGT
ACCCATAACTTGACAGTAGATCTCGTCCAGACCCCTAGCTGGTACGTCTTCAGTAGAAAATTGTTTTTTTCTTCCAAGAG
GTCGGAGTCGTGAACACATCAGT
```

11 Conclusion

In this assignment, a complete pipeline for genome assembly from short sequencing reads was successfully implemented. Through a series of modular and rigorously tested functions, the input reads were parsed, analyzed for overlaps, ordered correctly, and finally assembled into the reconstructed genome sequence. The approach, while based on simplifying assumptions (perfect overlaps and no read containment), demonstrates the fundamental concepts of sequence assembly in computational genomics.