BT3051 - ASSIGNMENT 4

BE23B016

K.VARUNKUMAR

26/09/2024

Contents

1 Problem 1

1.1 Requirements

- The question requires the function to calculate the minimum number of fragments needed to reconstruct the entire genome sequence.
- Each DNA fragment is represented by a tuple consisting of its start and end positions.
- The genome sequence has a total length denoted by n, and it is required to cover the entire sequence from position 1 to n.
- The function needs to determine the minimum number of fragments such that the fragments completely cover the entire genome from position 1 to n without leaving any gaps.
- The problem requires the use of a **greedy algorithm** to solve it efficiently.
- If it is impossible to cover the genome with the given fragments, the function should return an indication (e.g., return -1).

1.2 Implementation

- **Sorting Fragments:** The input list of DNA fragments is sorted by their starting positions. This ensures that fragments are processed in the order of their appearance in the genome.
- **Sequence Initialization:** The sequence list ('seq') is initialized by checking whether the first fragment starts at position 0. If not, the function prints an error indicating an incomplete sequence.
- **Handling Holes:** During the iteration, the algorithm checks for "holes" in the sequence (gaps between the current end of the sequence and the next fragment's start). If a gap is detected, the function returns -1 to indicate an incomplete sequence.
- **Greedy Extension:** The algorithm uses a greedy approach to extend the sequence by selecting the fragment that covers the furthest possible position at each step.
- **Completion Check:** After processing all fragments, the algorithm checks if the sequence covers the entire genome. If the final fragment does not cover position n, the function prints an error and returns -1.
- **Count of Fragments:** The function keeps track of the number of fragments added to the sequence and returns this count if the sequence successfully covers the entire genome.

All Edge cases are thus considered as illustrated above and a few of the outputs are shown below

Note:

The code was modified slightly to obtain the above output.

The function returns the length of the seq, if it is valid, Otherwise it returns -1.

1.3 Running Test cases

```
[(0, 2), (3, 5), (7, 10)]
Incomplete Sequence, Hole in between.
[(0, 2), (3, 5)]
======================================================================
[(0, 3), (4, 6), (8, 9)]
Incomplete Sequence, Hole in between.
[(0, 3), (4, 6)]
======================================================================
[(0, 1), (2, 4), (6, 8)]
Incomplete Sequence, Hole in between.
[(0, 1), (2, 4)]
======================================================================
[(0, 5), (6, 9)]
Incomplete Sequence, Total Sequence cannot be obtained from given data.
n= 10
[(0, 5), (6, 9)]
======================================================================
[(0, 2), (3, 4), (5, 7)]
Incomplete Sequence, Total Sequence cannot be obtained from given data.
n= 8
[(0, 2), (3, 4), (5, 7)]
======================================================================
[(0, 2), (3, 5), (6, 9), (11, 12)]
Incomplete Sequence, Hole in between.
[(0, 2), (3, 5), (6, 9)]
======================================================================
[(0, 1), (2, 3), (5, 7), (8, 10)]
Incomplete Sequence, Hole in between.
[(0, 1), (2, 3)]
======================================================================
[(0, 2), (3, 5), (7, 8)]
Incomplete Sequence, Hole in between.
[(0, 2), (3, 5)]
======================================================================
[(0, 3), (4, 6), (7, 10)]
Complete Sequence.
3
[(0, 3), (4, 6), (7, 10)]
======================================================================
[(1, 4), (3, 5), (0, 6), (5, 10), (8, 12)]
Complete Sequence.
3
[(0, 6), (5, 10), (8, 12)]
======================================================================
```

2 Problem 2

2.1 Requirements

- The problem requires selecting the maximum number of non-overlapping gene expression intervals.
- Each gene expression interval is represented as a tuple (start_time, end_time), where:
  - **start_time**: the time when the gene starts expressing.
  - **end_time**: the time when the gene stops expressing.
- The algorithm must return the maximum number of non-overlapping intervals that can be scheduled.
- The solution should use a **greedy algorithm**.
- The input consists of a list of n intervals.
- The output should be the maximum number of non-overlapping gene expression intervals that can be scheduled.

2.2 Implementation

- **Input:** The function takes a list of intervals (gene expression periods) as input, where each interval is represented as a tuple (start, end).
- **Reversing the Intervals:** Each interval is reversed to create a list where each tuple is in the form (end, start), facilitating sorting by the end times.
- **Sorting by End Times:** The reversed list is sorted by the end times, which is the first element of each tuple after reversing. This step ensures that intervals are processed in the order of their earliest end time.
- **Greedy Selection:**
  - The algorithm begins by selecting the first interval (the one with the earliest end time).
  - It iterates over the remaining intervals, adding an interval to the schedule if its start time is greater than or equal to the end time of the last selected interval.
- **Optimal Schedule:** The final list of non-overlapping intervals is stored in the `schedule` list, which contains the optimal set of intervals.
- **Output:** The function returns the number of non-overlapping intervals in the optimal schedule.
- **It is to be noted that the function assume all inputs are well behaved that is no tuple in the data has start_time > end_time.**

2.3 Running Test cases

```
The input data is: [(1, 10), (1, 3), (1, 4), (1, 2), (1, 5)]
The optimum scheduling as per requirement is : [(1, 2)]
1
================================================
The input data is: [(1, 3), (2, 3), (3, 3), (3, 5), (3, 6)]
The optimum scheduling as per requirement is : [(1, 3), (3, 3), (3, 5)]
3
================================================
The input data is: [(1, 3), (1, 3), (1, 3), (2, 3), (3, 3)]
The optimum scheduling as per requirement is : [(1, 3), (3, 3)]
2
================================================
The input data is: [(1, 3), (2, 4), (3, 5), (4, 6), (5, 7)]
The optimum scheduling as per requirement is : [(1, 3), (3, 5), (5, 7)]
3
================================================
```

---

3 Problem 3

3.1 Requirements

- The problem involves partitioning a given DNA sequence into subsequences where each subsequence is a palindrome.
- A **palindromic DNA sequence** is one that reads the same forward and backward.
- The input is a string, `dna_sequence`, consisting of nucleotides `'A', 'T', 'C', 'G'`.
- The goal is to return all possible partitions of the DNA sequence such that every subsequence in the partition is palindromic.
- The solution should use a **Dynamic Programming** approach to efficiently find all valid partitions.
- The output should be a list of lists, where each inner list represents a valid partition of palindromic subsequences.
- Each partition is made of subsequences that are palindromes, and all possible partitions need to be listed.

3.2 Implementation

- **Creating a DP Table:** A 2D array `dp` of size $n \times n$ (where n denotes the length of the input sequence) is initialized to store whether a substring starting at the i^{th} index and ending at j^{th} the is a palindrome.
- **Precomputing Palindromes:** Using the `dp` table, all substrings that are palindromes are identified in $O(n^2)$ time.
- **Dynamic Programming with Iterative Partitioning:** A dictionary-based approach is used to iteratively construct all possible palindromic partitions by extending previously computed valid partitions.
- **Output:** The final exhaustive list containing all possible palindromic partitions is returned.

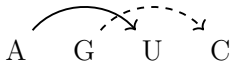
3.3 Running Test cases

```
Input string: ATGGGCTGTT
['A', 'T', 'G', 'G', 'G', 'C', 'T', 'G', 'T', 'T']
['A', 'T', 'G', 'G', 'G', 'C', 'T', 'G', 'TT']
['A', 'T', 'G', 'G', 'G', 'C', 'TGT', 'T']
['A', 'T', 'G', 'GG', 'C', 'T', 'G', 'T', 'T']
['A', 'T', 'G', 'GG', 'C', 'T', 'G', 'TT']
['A', 'T', 'G', 'GG', 'C', 'TGT', 'T']
['A', 'T', 'GG', 'G', 'C', 'T', 'G', 'T', 'T']
['A', 'T', 'GG', 'G', 'C', 'T', 'G', 'TT']
['A', 'T', 'GG', 'G', 'C', 'TGT', 'T']
['A', 'T', 'GGG', 'C', 'T', 'G', 'T', 'T']
['A', 'T', 'GGG', 'C', 'T', 'G', 'TT']
['A', 'T', 'GGG', 'C', 'TGT', 'T']
============================================================
Input string: ATTA
['A', 'T', 'T', 'A']
['A', 'TT', 'A']
['ATTA']
============================================================
Input string: ATGGGCTGTTCA
['A', 'T', 'G', 'G', 'G', 'C', 'T', 'G', 'T', 'T', 'C', 'A']
['A', 'T', 'G', 'G', 'G', 'C', 'T', 'G', 'TT', 'C', 'A']
['A', 'T', 'G', 'G', 'G', 'C', 'TGT', 'T', 'C', 'A']
['A', 'T', 'G', 'GG', 'C', 'T', 'G', 'T', 'T', 'C', 'A']
['A', 'T', 'G', 'GG', 'C', 'T', 'G', 'TT', 'C', 'A']
['A', 'T', 'G', 'GG', 'C', 'TGT', 'T', 'C', 'A']
['A', 'T', 'GG', 'G', 'C', 'T', 'G', 'T', 'T', 'C', 'A']
['A', 'T', 'GG', 'G', 'C', 'T', 'G', 'TT', 'C', 'A']
['A', 'T', 'GG', 'G', 'C', 'TGT', 'T', 'C', 'A']
['A', 'T', 'GGG', 'C', 'T', 'G', 'T', 'T', 'C', 'A']
['A', 'T', 'GGG', 'C', 'T', 'G', 'TT', 'C', 'A']
['A', 'T', 'GGG', 'C', 'TGT', 'T', 'C', 'A']
============================================================
```

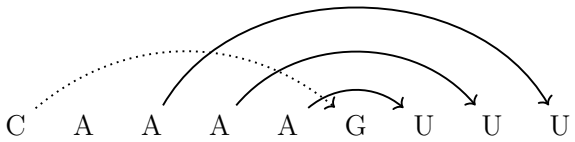
4 Problem 4

4.1 Consideration 1:

We shall consider Non-Overlapping base pairs to mean something as shown below.



Here the bond formation can be between either A-U or G-C, both cannot form as highlighted by the dashed bond line showing overlap. Another example to illustrate this is shown below.



Here the maximum possible bond pairs that can be formed without overlap is 3, due to the fact that the C-G bond overlaps with the other A-U bonds indicated by it's dashed nature.

"Another sample chosen as the test is shown here

It is possible another arrangement might also give 5, however it is impossible for the total maximum pairs to exceed 5.

This makes it clear that the maximum possible bond pairs here is 5, without any overlap.

Here the constrain boils down to the fact that, any subsequent base pair that is considered must have both it's bases in the range of a previous pair or be completely outside the previous pairs range.

4.1.1 Implementation

- **Input:** The function `max_rna_pairs1` takes a string `seq` representing a RNA sequence consisting of nucleotides.
- **Initialization:**
  - The length of the sequence is calculated and stored in variable n.
  - A 2D array `map` of size $n \times n$ is initialized to store the maximum number of pairs for each subsequence.
  - Valid nucleotide pairs are defined in the `pairs` list as `["A", "U"]`, `["U", "A"]`, `["G", "C"]`, and `["C", "G"]`.
- **Base Cases:** A loop initializes the diagonal of the DP table, setting `map[i,i] = 0` since no pairs can form with a single nucleotide.

- **Dynamic Programming Table Filling:**
  - The outer loop iterates over possible substring lengths (difference) from 1 to $n-1$.
  - The inner loop iterates through the sequence, calculating the maximum number of pairs for each subsequence defined by indices j and i.
  - A temporary variable `tmax` keeps track of the maximum pairs obtained by splitting the sequence at various points.
- **Pairing Logic:**
  - If the characters at positions j and i form a valid pair, the table entry `map[j,i]` is updated to reflect the maximum pairs obtainable, including the current valid pair and the maximum pairs from the inner subsequence.
  - If the characters do not form a valid pair, the maximum value from splits is retained.
- **Output:** The function prints the input sequence and the maximum number of pairs found in the full sequence (`map[0, n-1]`).

4.1.2 Running Test cases

```
UACGGUCACGUA
Maximum number of pairs: 5.0
============================================================
AUGCUAGC
Maximum number of pairs: 4.0
============================================================
CAAAGUUU
Maximum number of pairs: 3.0
============================================================
AGUGAC
Maximum number of pairs: 2.0
============================================================
AGUC
Maximum number of pairs: 1.0
============================================================
```

4.2 Consideration 2:

The other implication for "Non-Overlapping" could be that a single nucleotide can only be involved in bonding with a single other nucleotide.

This condition however reduces the problem to the mere sum of min(count(A,U)) and min(count(G,C)) of a given sequence.

4.2.1 Implementation

- **Input:** The function `max_rna_pairs2` takes a string `seq` representing a RNA sequence consisting of nucleotides.
- **Data Structure Initialization:**
  - A dictionary `data` is initialized to keep track of the counts of each nucleotide: `"A"`, `"U"`, `"G"`, and `"C"`, all set to zero.
- **Counting Nucleotides:**
  - The input sequence is converted to uppercase and stored in `temp` for uniformity.
  - A loop iterates through each nucleotide in `temp` and increments the corresponding count in the `data` dictionary based on the nucleotide type.
- **Calculating Maximum Pairs:**
  - The maximum number of valid RNA pairs is calculated as the sum of the minimum counts of pairs:
    - The minimum of `data["A"]` and `data["U"]` (for adenine-uracil pairs).
    - The minimum of `data["G"]` and `data["C"]` (for guanine-cytosine pairs).
- **Output:** The function returns the total maximum number of valid RNA pairs (`maxlen`).

4.2.2 Running Test cases

```
UACGGUCACGUA
6
================================================
AUGCUAGC
4
================================================
CAAAGUUU
4
================================================
AGUGAC
2
================================================
AGUC
2
================================================
```