BT3051 - ASSIGNMENT 5

BE23B016

K.VARUNKUMAR

11/10/2024

Contents

1 Problem 1

1.1 Requirements

- The training field is represented by an $m \times n$ grid.
- The robot starts at the top-left corner: grid[0][0].
- The robot's goal is to reach the bottom-right corner: grid[m-1][n-1].
- The field contains obstacles and open spaces:
  - Obstacles are marked as 1.
  - Open spaces are marked as 0.
- The robot can only move in two directions:
  1. Right.
  2. Down.
- The robot cannot move through obstacles (cells marked as 1).
- The task is to determine the total number of unique paths that the robot can take to reach the goal.

1.2 Implementation

We shall assume the well behaved nature of our sample data-set that is field[0][0] and field[r-1][c-1] are always 0, as it is our objective to reach there.

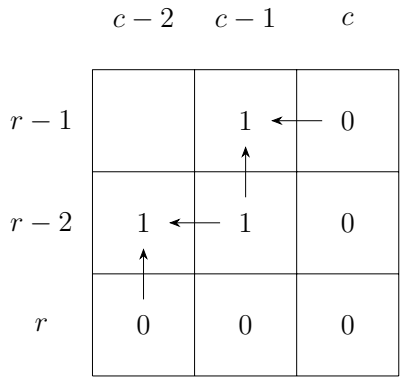
1.2.1 Filling the DP table

Initializing the data in the DP table

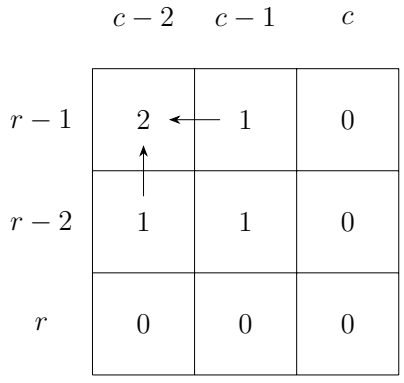
The 0's mark the boundary of the field, the 1 at DP[r-1][c-1] indicates from that point there is one way to get to the end.

	$c-2$	$c-1$	c
$r-1$			0
$r-2$		1	0
r	0	0	0

Fill the table diagonally as shown, this will ensure each coordinate has the total possible starting at it to reach the end.



Continue filling the entire table in a similar fashion until the top is reached.



If an obstacle is encountered, it is marked with 0 as there are 0 paths from it to the end (You can not end up in an obstacle).

Time Complexity Analysis

The final answer is the value obtained at dp[0][0], As it contains the total number of paths possible from [0,0] to the end.

This Solution utilizes a Dynamic Programming approach and is of $O(mn)$ complexity.

1.3 Running Test Cases

```
  0   0   1   0   0   0
  0   0   0   0   1   0
  0   0   0   0   0   0
  0   1   0   1   0   0
============================================================
 14   6   0   3   1   1   0
  8   6   4   2   0   1   0
  2   2   2   2   2   1   0
  0   0   0   0   1   1   0
  0   0   0   0   0   0   0
============================================================
 14
============================================================
```

In the test case below there are no paths that lead you to the end from the start, the program captures this quite elegantly as shown.

```
  0   1   0   0   0
  0   0   1   1   0
  1   0   0   1   0
============================================================
  0   0   1   1   1   0
  0   0   0   0   1   0
  0   0   0   0   1   0
  0   0   0   0   0   0
============================================================
  0
============================================================
```

2 Problem 2

2.1 Requirements

2.1.1 Task

- Read the FASTA File for the DNA Sequence.
- Find the optimal overlap alignment between a suffix of sequence s and a prefix of sequence t to maximize the alignment score.
- The scoring scheme for alignment is as follows:
  - $+1$ point for matching characters.
  - -2 points for mismatches (substitutions).
  - -2 points for gaps (insertions or deletions).
- If multiple alignments have the same optimal score, return any one of them (The choice has been made here to return the longest sequence with the maximum score).

2.1.2 Output

To return the :

- The optimal alignment score.
- The aligned suffix of s and the aligned prefix of t that achieves this score.

The output is returned in the form of list :
Output : $[\langle max\_score \rangle, \langle suffix\_sequence \rangle, \langle prefix\_sequence \rangle]$

2.2 Implementation

2.2.1 Constructing the DP Table

- The function initializes a 2D DP table of size $(n+1) \times (m+1)$, where n is the length of the prefix and m is the length of the suffix.
- The DP table is filled using dynamic programming based on matching characters, mismatches (substitutions), and gap penalties.
- The first row and the first column of the DP table are initialized to handle gaps in the alignment.

2.2.2 Finding Maximum Score

- While filling the DP table, the algorithm tracks the maximum score encountered in the last column (i.e., aligning the suffix of string s with the prefix of string t).
- The coordinates of this maximum score are saved to facilitate backtracking for the optimal alignment.

- Maximum scores are tracked at the end of the match with the suffix sequence, as all the characters till the end must be considered for it to be a suffix.

2.2.3 Backtracking for the Alignment

- Starting from the cell with the maximum score, the algorithm backtracks through the DP table to reconstruct the optimal alignment.
- Backtracking employs a greedy strategy when multiple outputs give the same maximum score, since the problem statement requires us to output only 1 matching sequence.
- It traces back the path that led to the maximum score, determining whether the alignment step is a match, mismatch, or gap, and constructs the aligned suffix and prefix.

Time Complexity Analysis

- The time complexity of the algorithm is $O(n \times m)$, where n is the length of the prefix and m is the length of the suffix. This is because the DP table is filled in a nested loop structure, iterating over all cells in the table.
- The space complexity is also $O(n \times m)$, as the DP table stores values for each combination of the suffix and prefix up to length n and m, respectively.

2.3 Running Test Cases

```
The suffix sequence is : TAGCATGGC
=======================================================
The prefix sequence is : AGCATATGCTACT
=======================================================
The max score is :  3
=======================================================
From the suffix : AGCAT-GGC
From the prefix : AGCATATGC
=======================================================
```

```
The suffix sequence is :
AGCCCTCCAGGACAGGCTGCATCAGAAGAGGCCATCAAGCAGGTCTGTTCCAAGGGCCTTTGCGTCAGGTGG
GCTCAGGATTCCAGGGTGGCTGGACCCCAGGCCCCAGCTCTGCAGCAGGGAGGACGTGGCTGGGCTCGTGAA
GCATGTGGGGGTGAGCCCAGGGGCCCCAAGGCAGGGCACCTGGCCTTCAGCCTGCCTCAGCCCTGCCTGTCT
CCCAGA
=========================================================================
The prefix sequence is :
AGGCGGGCACTGTGTCTCCCTGACTGTGTCCTCCTGTGTCCCTCTGCCTCGCCGCTGTTCCGGAACCTGCTC
TGCGCGGCACGTCCTGGCAGTGGGGCAGGTGGAGCTGGGCGGGGGCCCTGGTGCAGGCAGCCTGCAGCGGGG
GAAGGAGGTGGGACATGTGGGCGTTGGGGCC
=========================================================================
The max score is :  3
=========================================================================
From the suffix : AGGCAGGGCACCT-GG-CCTTCAGCCTG-CCTCAG-CCCTGCCTGTCTCCCAGA
From the prefix : AGGC-GGGCAC-TGTGTC-T-C--CCTGAC-TGTGTCC-T-CCTGTGTCCCTCT
=========================================================================
```

3 Problem 3

3.1 Requirements

3.1.1 Task

- Find the longest increasing subsequence of the gene expression levels.
- Find the longest decreasing subsequence of the gene expression levels.
- If multiple subsequences of the same maximum length exist, return any one.

3.1.2 Output

- First, output the longest increasing subsequence of the gene expression levels.
- Then, output the longest decreasing subsequence of the gene expression levels.

3.2 Implementation

- The function `longest_inde_ss` takes two inputs: n (the length of the list) and l (the list of gene expression levels).
- Two dictionaries, `inc_sub` and `dec_sub`, are used to store the Longest Increasing Subsequence (LIS) and Longest Decreasing Subsequence (LDS) respectively for each index. The key is the index, and the value is a tuple containing:
  - The length of the subsequence starting at that index.
  - The actual subsequence itself.
- The algorithm uses a dynamic programming approach, with the base case occuring at the last index, whose LIS and LDS respectively are itself.
- A loop runs backward from $n-2$ to 0, comparing each element $l[x]$ with all subsequent elements $l[y]$ (where $y > x$).
  - For the LIS: If $l[x] < l[y]$ and appending $l[y]$ results in a longer subsequence, the LIS at index x is updated.
  - For the LDS: If $l[x] > l[y]$ and appending $l[y]$ results in a longer subsequence, the LDS at index x is updated.
- The algorithm keeps track of the starting index of the longest LIS and LDS during this process.
- After the loop finishes, the actual subsequences are retrieved using the indices of the longest LIS and LDS and returned as the result.

Time Complexity Analysis

- The outer loop runs from $n-2$ to 0, iterating over the gene expression levels.
- For each element $l[x]$, an inner loop checks all subsequent elements $l[y]$, where $y > x$.
- As a result, the algorithm examines every pair of elements in the list, leading to a time complexity of $O(n^2)$.

3.3 Running Test Cases

NOTE : As stated in the requirements, the longest increasing subsequence is printed first followed by the longest decreasing subsequence,
The program however returns this as a list of lists, and the sequence is printed in this format by a function **cleanprint** .

```
The sample data is :
[16, 26, 16, 1, 4, 9, 19, 4, 18, 17, 35, 36, 24, 47, 32]
================================================================
[1, 4, 9, 19, 35, 36, 47]
================================================================
[26, 16, 9, 4]
================================================================
```

```
The sample data is :
[32, 43, 48, 22, 22, 39, 19, 27, 29, 33, 46, 11, 12, 13, 5]
================================================================
[19, 27, 29, 33, 46]
================================================================
[48, 22, 19, 11, 5]
================================================================
```

4 Problem 4

4.1 Requirements

- You are given a sequence of characters (the scroll) of length N.
- You can perform up to k substitutions, where each substitution allows you to change any character in the scroll to any other character.
- The task is to determine the length of the longest palindromic substring that can be formed from the scroll after performing up to k substitutions.

Input

- The length of the scroll N.
- A string of N characters representing the scroll.
- An integer k representing the maximum number of substitutions allowed.

Output

- The length of the longest palindromic substring that can be obtained from the scroll after up to k substitutions.

4.2 Implementation

- A sliding window approach was used to identify all the possible substrings which could be converted into a palindrome with k or fewer changes.
- For each potential center of the palindrome (both odd and even-length centers are considered), the algorithm expands the window while counting how many characters need to be substituted to make the substring a palindrome (Even centers are empty gaps in the parent string, and odd centers are the characters in the base string themselves, the string was slightly modify to ensure the capture of all centers) .
- If the number of necessary substitutions does not exceed k, the window is considered valid, and the length of the palindromic substring is tracked.
- The algorithm then returns the maximum length of such a valid substring.

Time Complexity Analysis

- In the worst case, for each character in the string (there are N characters), the algorithm attempts to expand around each possible center to check for palindromes.
- For every center, expanding to the left and right takes $O(N)$ time.
- Thus, in the worst case, the time complexity is $O(N^2)$, where N is the length of the scroll.

4.3 Running Test Cases

```
scroll contains : xsabcabcabcgta
Substitutions allowed : 3
The length of the longest palindromic substring is : 9
```

SABCABCAB

SABCABCAB

```
scroll contains :
Level is a device used to find plane surfaces. Levels
are commonly used by engineers for determining flat
surfaces.
Substitutions allowed : 18
The length of the longest palindromic substring is : 52
```

LEVEL IS A DEVICE USED TO FIND PLANE SURFACES. LEVEL

LEVEL IS A DEVICE USED TO FIND PLANE SURFACES. LEVEL

The Underlined characters denote the one we would like to substitute to obtain the longest palindromic sequence.

---

5 Problem 5

5.1 Requirements

- You are given two DNA sequences:
  - A reference DNA sequence S of length n, which is typically a long strand of DNA.
  - A target gene sequence T of length m, which is a shorter subsequence representing a gene.
- The goal is to find the starting index (0-based) of all occurrences of the target gene T in the reference sequence S using the Knuth-Morris-Pratt (KMP) algorithm for efficient string matching.
- If the target gene appears multiple times in the reference sequence, return all starting indices of the occurrences.
- If the target gene does not appear in the reference sequence, return an empty list.

5.2 Implementation

FT (Fail Table Construction)

- The function `FT(gene)` computes the fail table, which stores the length of the longest proper prefix of the `gene` that is also a suffix for each index of the string.
- This fail table is used to skip unnecessary comparisons during the KMP string matching process, reducing the overall time complexity.

KMP (Knuth-Morris-Pratt Algorithm)

- The function `KMP(genome, gene)` searches for all occurrences of the `gene` sequence within the `genome` sequence by using the fail table to avoid redundant comparisons.
- After a successful match or mismatch, the fail table is utilized to restart matching at the appropriate position in the `genome`, which prevents re-checking already matched portions of the string.

Time Complexity Analysis

- The time complexity of constructing the fail table is $O(m)$, where m is the length of the gene sequence.
- The time complexity of the KMP search algorithm is $O(n + m)$, where n is the length of the genome sequence. Each character of the genome is processed at most once due to the fail table, which ensures that once a mismatch occurs, the algorithm uses previously computed information to skip unnecessary checks.

5.3 Running Test Cases

```
Genome is :
TGCAGTCTGAAAGCCTGCAATCGTACGGCTTAGCTGTACAGCTGAAGCCTGCAAACGTTAGCATGGCCTGC
TGATCGGATCGTAGCGAAGCCTGCAAGTACGTGGCTAAAGCTGTCCGTAACGGCTGCAACTGCAGTACCGT
GAACGTCGTGGAAGCCTGCAAATCGGCTATGCTTACCGTAGGTTACAGCTGAAGGCTGCACGCTCGTTGAA
GCCTGCAAGCATGATCGTTCGAACGGTTAACGTCGATTAGCCTGCTAGTTAAACCGTATCGCGT
Gene is : AAGCCTGCAA
[10, 44, 87, 153, 211]
================================================================================
```

```
Genome is :
CTGAAAGCCTGCAACGTATGCGGTTAGACCGTAAAGCCTGCAATCTAGCTTACGCGTTAGGAGGTCGTAAGC
CTGCAATCTGTAAGGCTGTTAGCGTATGCGAGTCCGAAAGCCTGCATGGTACGTGTAAGCCTGCAAACGTGC
AAGGTTAGGTCGTATGCGTACCTGGAAGCCTGCAAATCGTTAGCTAGCGTACGGTAGACGTAGTAGTCGTTA
GGTCGTGCTTAGTCCAGCAGATGAGTGTTTG
Gene is : AAGCCTGCAA
[4, 33, 68, 128, 169]
================================================================================
```
