**OBSERVATION BOOK :**

**Lab – 2 : Edit Distance and Applications**

* **Submitted by Oishanee Mukherjee, 4MDS, 2448042**

**Program Question – 1 :**

*Solve exercise 2.4 and 2.5 from Text book of Speech and Language Processing of Daniel*

*Jurafsky and team :*

**2.4.** Compute the edit distance (using insertion cost 1, deletion cost 1, substitution cost 1) of “leda” to “deal”. Prepare an edit distance grid to complete your work.

**2.5.** Figure out whether the “drive” is closer to “brief” or to “divers” and what the edit distance is to each. You may use any version of distance that you like.

***Program Description :***

This question involves calculating the edit distance (Levenshtein distance) between pairs of words manually using a cost model: Insertion = 1, Deletion = 1 & Substitution = 1

Our aim is to:

1. Construct the edit distance grid for the strings "leda" and "deal".
2. Compute and compare the edit distances of "drive" to "brief" and "divers" to determine which one is closer.

***Program Logic :***

***Algorithm : Dynamic Programming (Lavenshtein Distance)***

*if str1[i-1] == str2[j-1]:*

*cost = 0*

*else:*

*cost = 1*

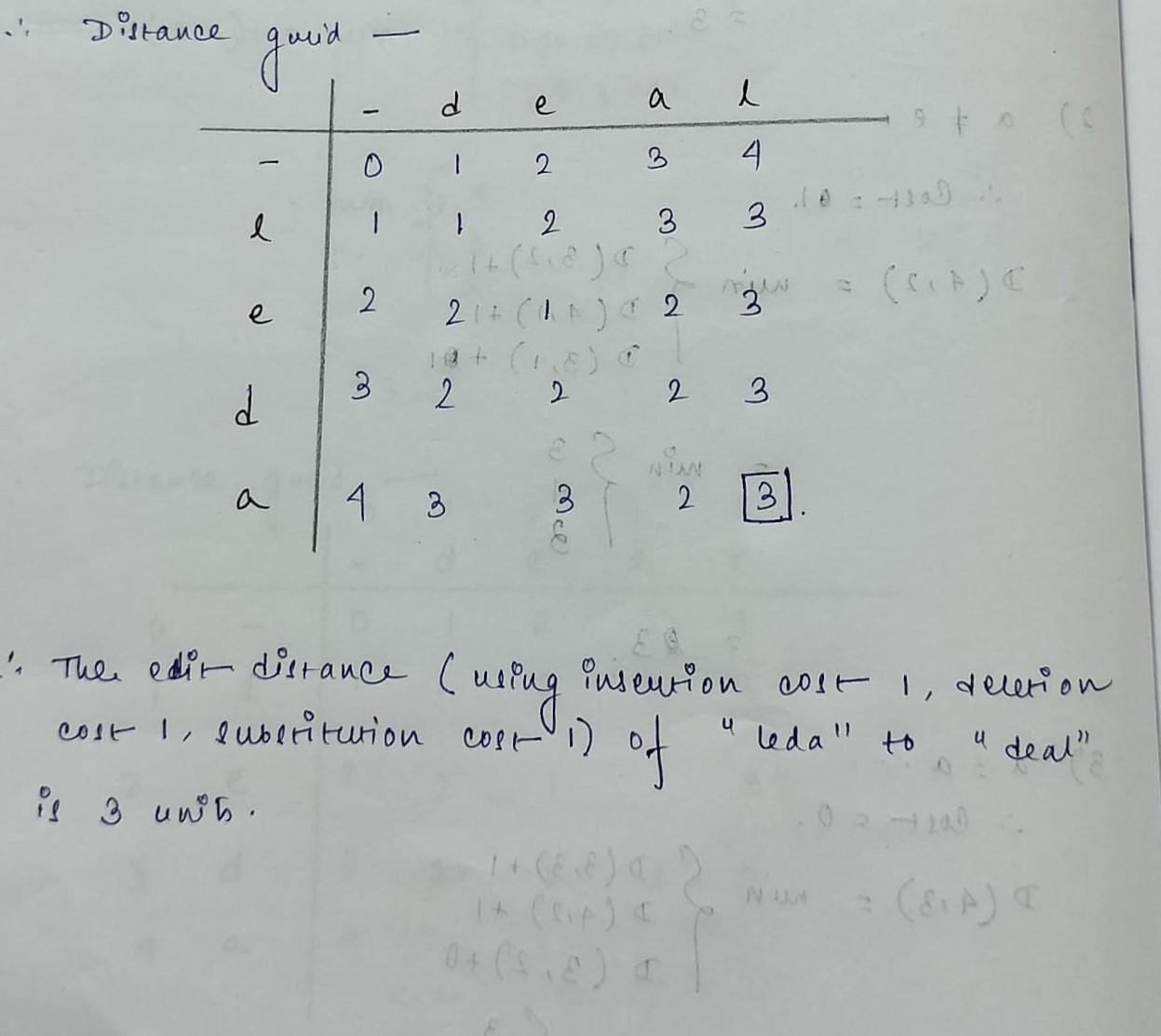
*dp[i][j] = min(dp[i-1][j] + 1, # deletion*

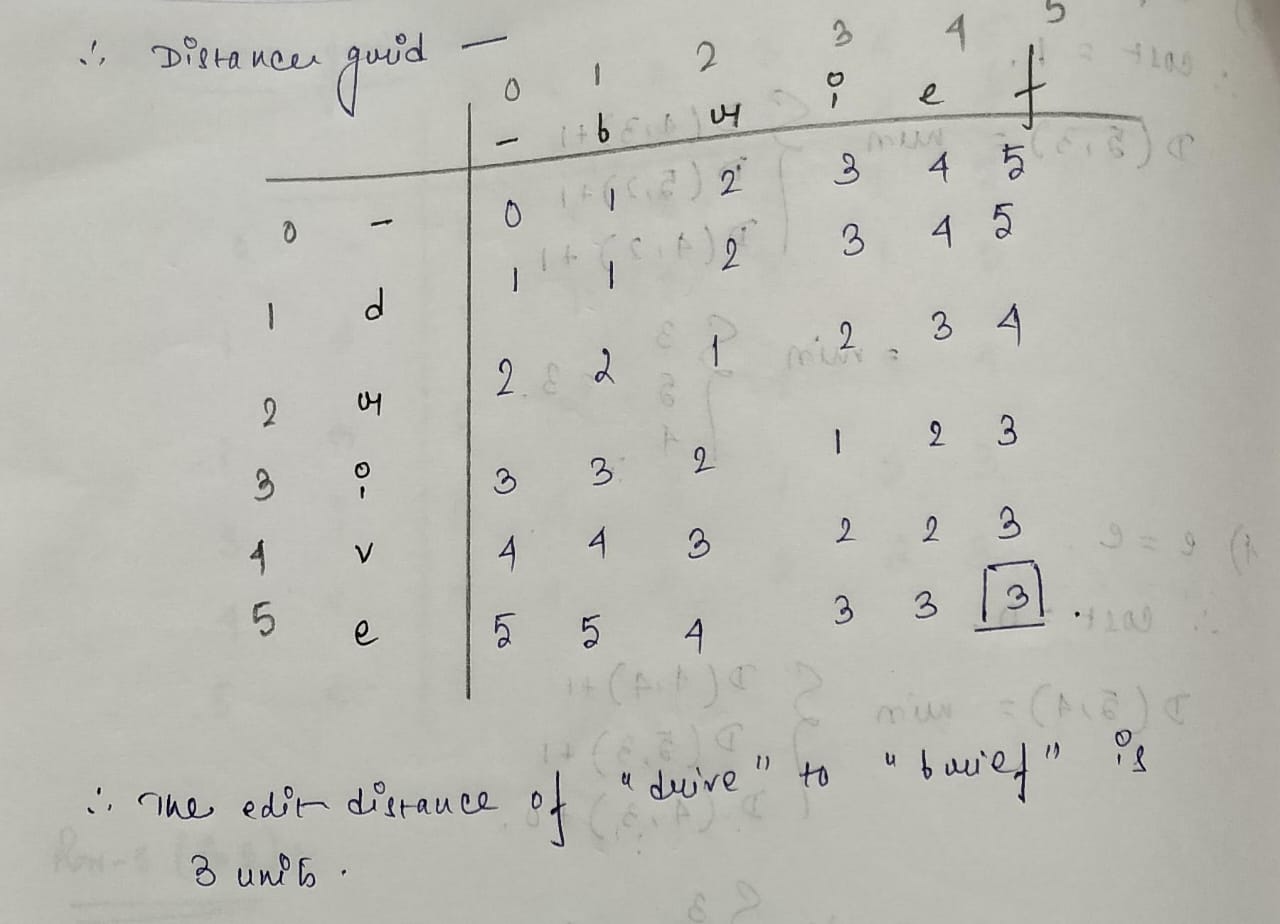
*dp[i][j-1] + 1, # insertion*

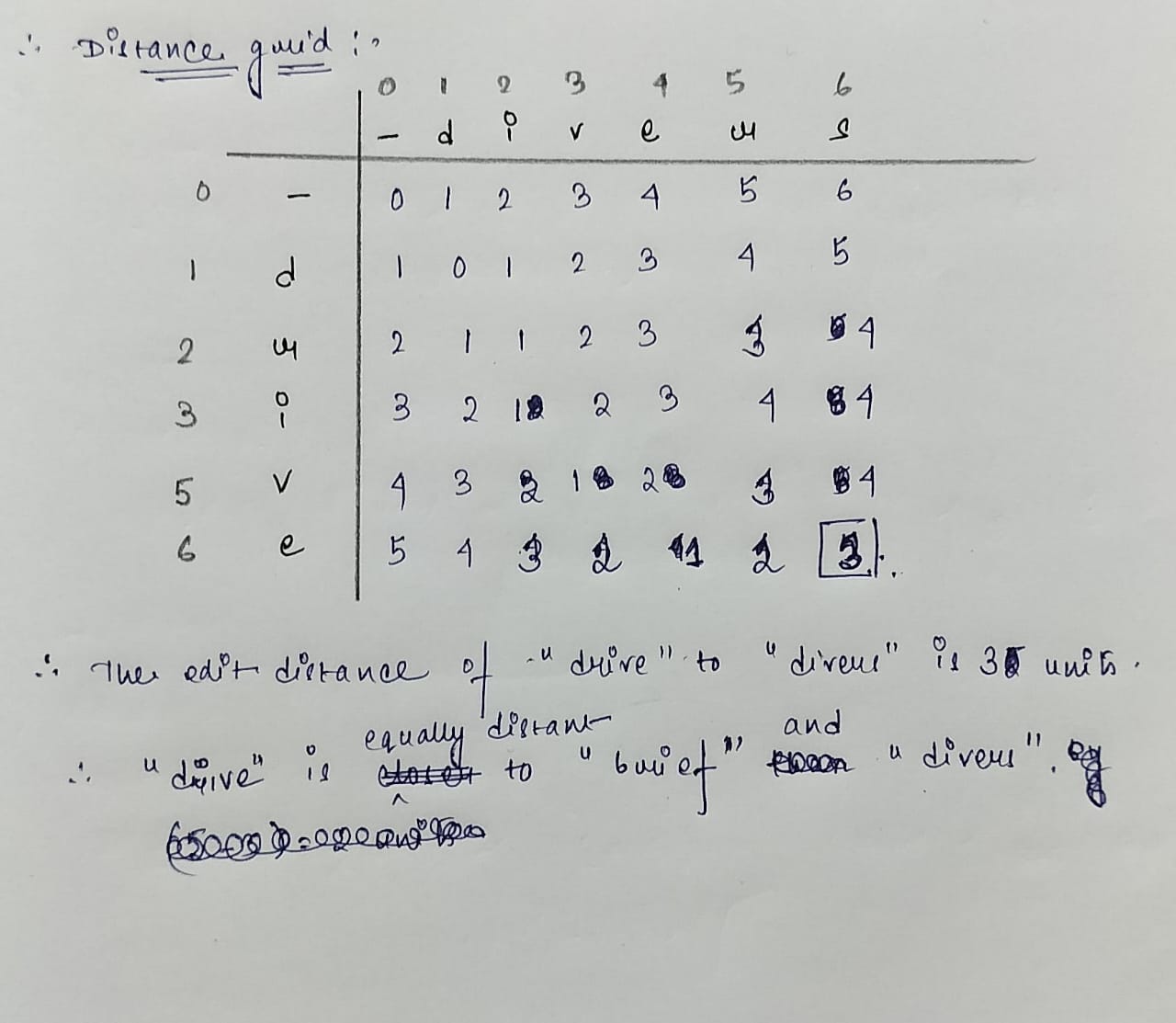
*dp[i-1][j-1] + cost) # substitution*

**where dp[m][n] is the minimum edit distance.**

***Manual Computation :***

**2.4. **

**2.5.** 



**Program Question – 2 : Edit Distance**

**2.6.** Implement a minimum edit distance algorithm and use your hand-computed results to check your code.

str1 = "leda"

str2 = "deal"

distance = levenshtein\_distance(str1, str2)

print(f"The edit distance between '{str1}' and '{str2}' is {distance}")

***Program Description :***

This program calculates the Levenshtein distance (also known as minimum edit distance) between two user-input strings. The edit distance is the minimum number of operations required to convert one string into another, where the allowed operations are: Insertion, Deletion and Substitution.

***Program Logic :***

1. ***Algorithm : Dynamic Programming***

*if str1[i-1] == str2[j-1]:*

*cost = 0*

*else:*

*cost = 1*

*dp[i][j] = min(dp[i-1][j] + 1, # deletion*

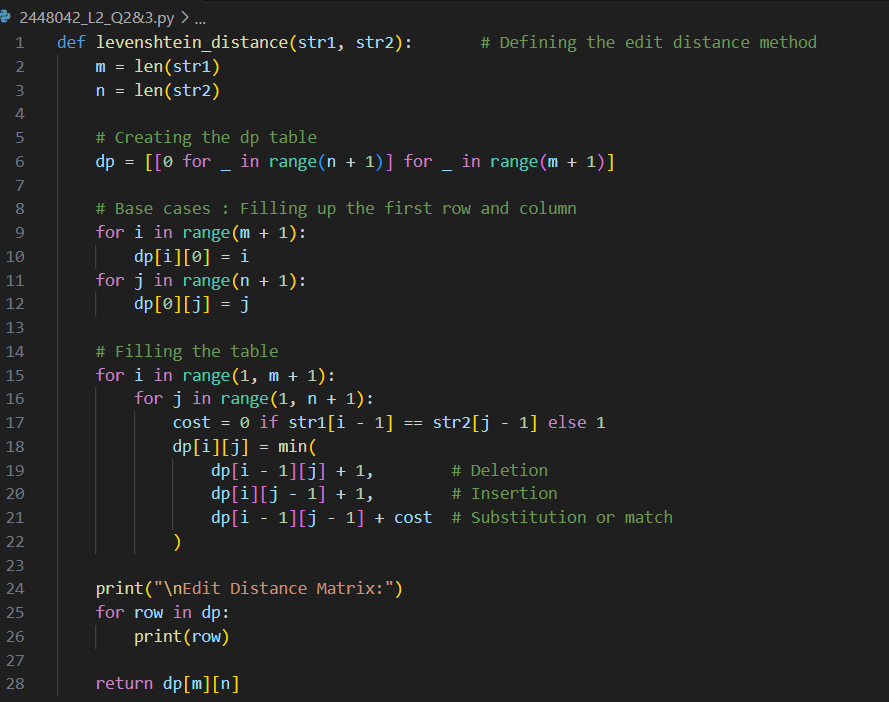
*dp[i][j-1] + 1, # insertion*

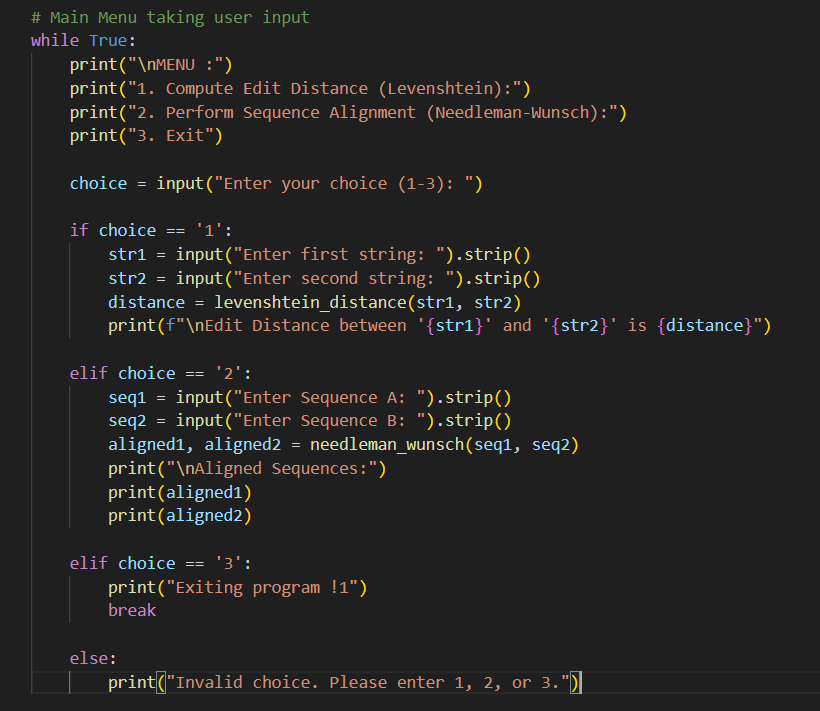
*dp[i-1][j-1] + cost) # substitution*

**where dp[m][n] is the minimum edit distance.**

1. ***Data Types:*** Strings, Integers and 2D list.

***Python Source Code :***





***Test Cases :***

|  |  |
| --- | --- |
| INPUT | OUTPUT |
| leda, deal | 3 |
| flaw, lawn | 2 |
| abc, abc | 0 |
| a,b | 1 |
| book, back | 2 |
|  |  |

**Program Question – 3 : ( Implement Sequence Alignment )**

Write a program to align the given sequence of input text A and B

**Input:**

**Text A:** AGGCTATCACCTGACCTCCAGGCCGATGCCC

**Text B:** TAGCTATCACGACCGCGGTCGATTTGCCCGAC

**Output:**

-AGGCTATCACCTGACCTCCAGGCCGA--TGCCC---

TAG-CTATCAC--GACCGC—GGTCGATTTGCCCGAC

***Program Description :***

This program implements a global sequence alignment algorithm (Needleman-Wunsch) to align two DNA or protein sequences entered by the user. The purpose is to determine the best possible alignment between two sequences by introducing gaps (-) and matching/mismatching characters based on a scoring system.

***Program Logic :***

1. ***Algorithm : Needleman-Wunsch Algorithm (Global Alignment)***

*score[i][j] = max(*

*score[i-1][j-1] + (match or mismatch),*

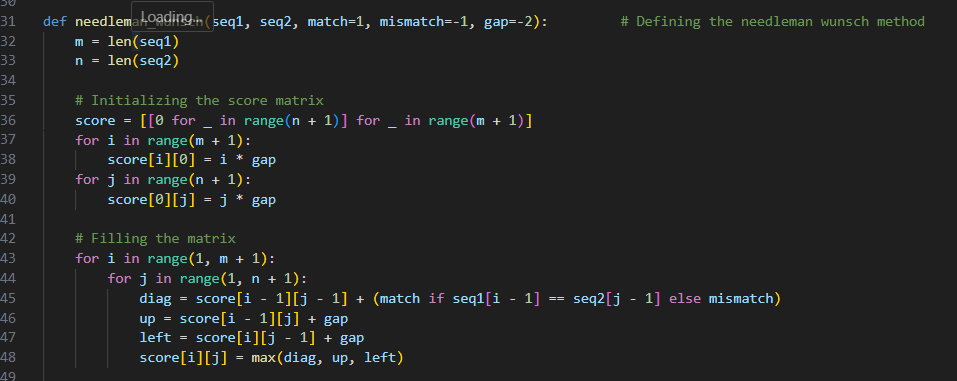
*score[i-1][j] + gap,*

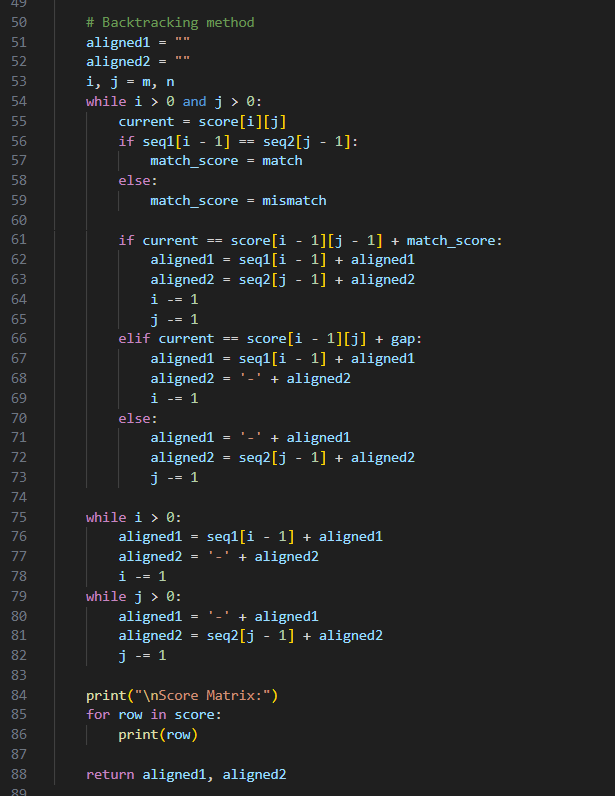
*score[i][j-1] + gap*

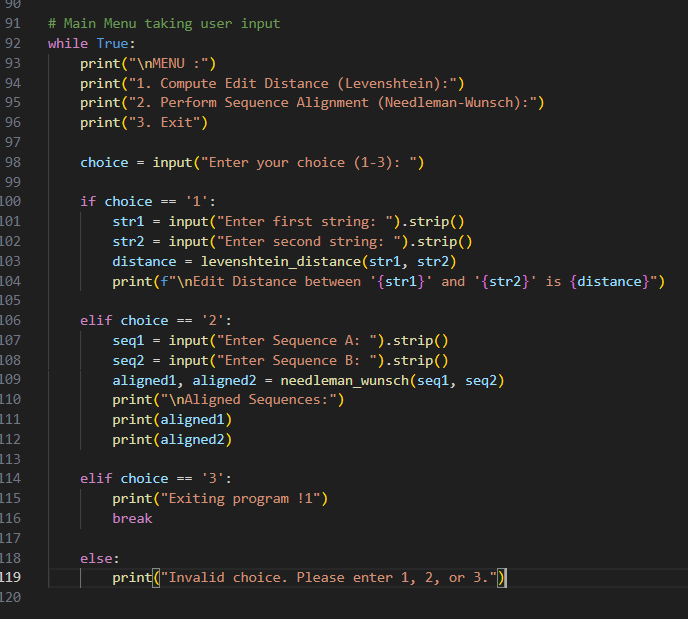
*)*

1. ***Data Types :*** str, int, list, tuples.

***Python Source Code :***

****

****

****

***Test Cases :***

|  |  |
| --- | --- |
| INPUT | OUTPUT |
| AGGCTATCACCTGACCTCCAGGCCGATGCCC  TAGCTATCACGACCGCGGTCGATTTGCCCGAC | -AGGCTATCACCTGACCTCCAGGCCGA--TGCCC---  TAG-CTATCAC--GACCGC--GGTCGATTTGCCCGAC |
| |  | | --- | |  |   GATTACA  GCATGCU | G-ATTACA  GCA-TGCU |
| ACT  ACT | ACT  ACT |
| ACGT  ACCT | ACGT  ACCT |

***Evaluation Comments :***