Carleton University

Department of Systems and Computer Engineering

SYSC 2004 – Object Oriented Software Development

Creating A DNA Sequencer Using Object-Oriented Programming in Java

Team Members:

Megan McEwen
Emma Maddock

Student IDs:

101003509
100996472

Professor: Moayad Aloquaily

TA: Charles Bergeron

Date: 27 March 2017

1.0 - PROBLEM IDENTIFICATION AND STATEMENT

This project's objective is to create a program which takes in two user-inputted DNA sequences and performs a number of functions, such as finding optimal alignments of the strands or printing the inputted strands, as selected from a menu outputs several different solutions for the user to view. DNA (deoxyribonucleic acid) is a double-stranded molecule, with forward and reverse strands. For the purposes of this program, only the forward strand is used. A user can determine if two forward strands from two different sections of DNA are similar by performing sequence analysis.

1.1 – The Problem

The problem presented in this project is to design and implement a program which performs various functions on two user-inputted strands of DNA, given parameters by the user, and using various algorithms. The goal of the program, is to produce a series of optimal alignments based on the user-inputted DNA strands, and user parameters.

This program takes in two user-inputted strands of DNA and outputs several options for the user to choose which analyze the strand in different ways. DNA is comprised of four "letters" – A, C, G, and T. A user can determine whether two strands are similar by analyzing the number of "matches" (same letters in the same position), "mismatches" (different letters in the same position) and "gaps" (a letter in one strand where there is no letter in another strand). Using various algorithms such as the Needleman-Wunsch (explained in Section 2), an optimal score can be produced and an optimal alignment created that aligns the letters and gaps in both sequences to ensure maximum similarity.

Solving the problem involves constructing a menu that accepts valid inputs and rejects invalid inputs, and having the menu options call methods from the appropriate classes to create DNA objects, and to perform the relevant algorithms.

1.2 – Program Execution

The program sees each "menu" option above as a block of code to be executed. All pathways start in a super class's main method, and depending on the option selected, may enter several sub-classes and abstract classes. The user can keep choosing options from the menu until they wish to exit the program. In aligning the strands, the following criteria will be implemented to ensure the best alignment possible:

- The order of letters in each sequence is always conserved.
- The number of matches is maximized using an optimal alignment score.
- The number of gaps in each sequence is conserved using a gap penalty.
- The number of mismatches is minimized using a mismatch penalty.
- All possible optimal alignments are discovered.

Using these criteria, the program will create optimal alignments for the strands inputted by the user. Each of these penalties and scores is explained further in Section 2.

2.0 – GATHERING OF INFORMATION AND INPUT/OUTPUT DESCRIPTION

This program can accomplish several actions, depending on what the user requires. These functions will be explained in the subsections below:

2.1 – Inputs and Outputs

There are multiple inputs and outputs for the program, as explained below.

2.1.1 - Inputs

The program has a total of five inputs: two sequences, the match and mismatch scores, and the gap penalty. The user can input up to two sequences at a time. These sequences must only be comprised of A, C, G, or T characters (as these are the letters used in DNA); otherwise, the program will prompt the user until a valid strand is entered. As well, the user can input the following parameters, all of which can be of either int or double values:

- *Match score*: used in the Needleman-Wunch algorithm. The value is used every time two genetic letters "match" – for example, if two 'G' letters appeared at the same place and the user inputted the match score as 1, the score function would enter a +1.
- *Mismatch score*: same as match score, but used every time two letters do not match. For example, if a 'C' and a 'T' appeared in the same spot in both sequences, and the user inputted the mismatch score as -1, the mismatch penalty function would enter a -1.
- *Gap penalty*: if a character in one sequence is aligned with a gap in the second sequence, a gap penalty is applied. For example, if the first sequence has a 'C' and the second has nothing, and the user inputted the gap penalty as 0, the gap penalty function would record a 0 for that space.

The user starts the program by entering the two sequences (String), and the match score, mismatch score, and gap penalty (int or double). The user can input two new sequences, as well as new scores and penalties, by selecting the "enter new sequences" option on the menu.

2.1.2 - Outputs

The program has a total of five different outputs, all of which use various methods and classes to manipulate the input sequences and scores:

- *Print both sequences:* outputs both inputted sequences for the user to see on the screen.
- *Print an initialized matrix*: outputs a 2D zero matrix, with sides *i* and *j* (*i* = length of strand 1, *j* = length of strand 2). See Section 2.2 for more detail.
- *Print a Needleman-Wunsch matrix*: outputs a matrix filled using the Needleman-Wunch algorithm (explained in Section 2.2). The bottom-right cell of the matrix contains the optimal alignment score.
- *Print an optimal alignment*: implements a traceback procedure (explained in Section 2.3) through the Needleman-Wunsch matrix and outputs an optimal alignment for the DNA strands.

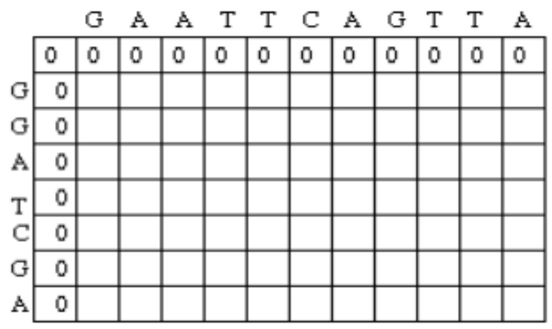
- *Print all optimal alignments:* Follows same procedure as print one optimal alignment, but outputs all possible pathways.
- *Enter new sequences*: allows the user to start over with two new strands of DNA to be sequenced.
- *Exit:* ends the program.

Each output option can be chosen from the menu, with the resulting sequence, matrix, or alignment outputted onto the screen for the user to view. These outputs are explained in more detail in subsequent sections.

2.2 – Dynamic Programming Matrix

The dynamic programming matrix uses the match score, mismatch score, and gap penalty, along with the lengths *i* and *j* of the sequences, to create an initial zero matrix and a Needleman-Wunsch matrix.

To start, a 2D matrix F(i, j) of type int is created and initialized with zeroes. An example of what might be outputted when a user asks for an initialized matrix can be seen in Figure 2.1 below:



		G	A	A	T	T	C	A	G	T	T	A
	0	0	0	0	0	0	0	0	0	0	0	0
G	0											
G	0											
A	0											
T	0											
C	0											
G	0											
A	0											

Figure 2.1 - Initialized Matrix [1]

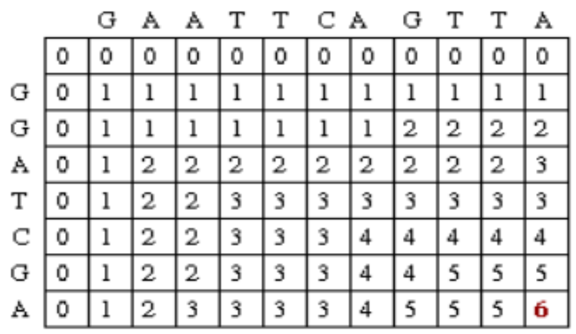
Once an initialized matrix has been created, it can be filled using the Needleman-Wunsch algorithm. Each square in the matrix can be filled using information from the cells to the left, right, and diagonal of it. The first row and first column are initialized as follows:

- F(0, 0) = 0
- F(i, 0) = (gap penalty * i)
- F(j, 0) = (gap penalty * j)

Once the first row and column are initialized, the rest of the matrix is filled using the equation below:

$$F(i, j) = \max[\, F(i-1, j-1) + \text{match/mismatch score}(\text{sequence1\_i}, \text{sequence2\_j}),\ F(i-1, j) - \text{gap penalty},\ F(i, j-1) - \text{gap penalty}]$$

Using this equation, the entire Needleman-Wunsch matrix can be filled in. Since it is possible for both ints and doubles to be inputted, all values first go through an overloaded method that allows for both types of values. The result is a matrix like the example shown in Figure 2.2 below:



		G	A	A	T	T	C	A	G	T	T	A
	0	0	0	0	0	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1	1	1	1	1	1
G	0	1	1	1	1	1	1	1	2	2	2	2
A	0	1	2	2	2	2	2	2	2	2	2	3
T	0	1	2	2	3	3	3	3	3	3	3	3
C	0	1	2	2	3	3	3	4	4	4	4	4
G	0	1	2	2	3	3	3	4	4	5	5	5
A	0	1	2	3	3	3	3	4	5	5	5	6

Figure 2.2 – Filled in Needleman-Wunsch Matrix [1]

The value in the bottom-right cell of the Needleman-Wunsch matrix is the optimal alignment score, and will be used in the traceback procedure.

2.3 – Traceback Procedure

The traceback procedure goes back through the Needleman-Wunsch matrix and outputs an optimal alignment for the sequences. To determine the best path to take through the matrix, it begins in the bottom-right cell and moves backwards to the one or more cells around it that the value inside the cell was derived from. A pathway created using the traceback procedure can be seen in Figure 2.3 below:

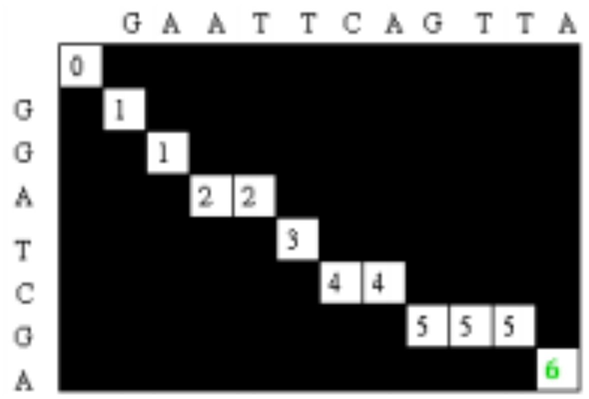


Figure 2.3 – Pathway Obtained Using Traceback [1]

Once a pathway is found, an optimal alignment can be printed. At each step in the traceback, a character is added to the alignment using the following rules:

- If the step was to (i – 1, j – 1), there was a match and the character at sequence_1[i] and sequence_2[j] is added.
- If the step was to (i – 1, j), there was a gap and a '_' representing a gap is added.
- If the step was to (i, j – 1), there was a mismatch and the character at sequence_2[j] is added.

Using these rules, an optimal alignment is outputted for the user.

2.4– Program Framework

The UML diagram for the program can be seen below in Figure 2.4:

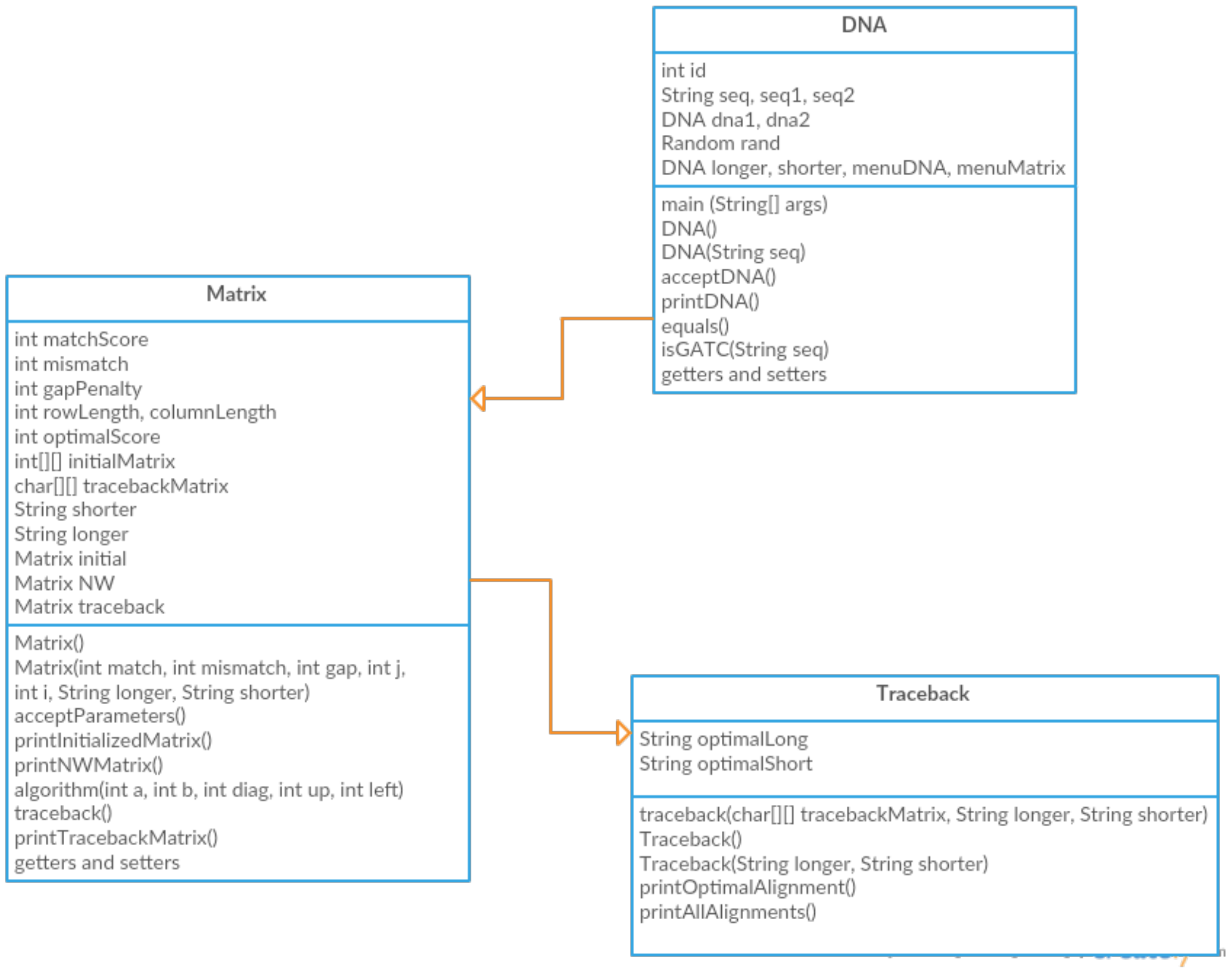


Figure 2.4: UML Diagram

This diagram gives an overview of the different functions of the program. As the code is developed, the UML diagram will be subject to change as variables and methods are added. Details on each method and class shown in the diagram will be explained in Section 3.

3.0 – TEST CASES AND SOFTWARE DESIGN

The program is divided into three classes. Class DNA is the main class, containing the main method. The Matrix and Traceback classes are both extended from DNA, and contain methods to implement the algorithms needed to find the initialized matrix, the Needleman-Wunsch Matrix, and the optimal alignment. Each method is described in detail below.

3.1 – Test Cases

There are a few places where test cases are needed to catch problematic inputs from the user:

- If the DNA input is wrong (contains letters other than G, A, T, or C) the program asks the user to input a valid sequence.
- If the user inputs incorrect parameters for the matrix (match score less than zero, mismatch score greater than zero, or gap penalty greater than zero) the program asks the user to input valid parameters.

3.2 – Design

3.2.1 - Introduction

The design of each method used in the three classes is described in detail below. The program runs entirely from the main method, located in the DNA class. Upon initialization, it creates a DNA and Matrix object, which are both used to call methods from their respective classes (given that main is static). The program has a layered structure, with lower layers (subclasses) returning values to upper layers (parent classes), which all return ultimately to the main method.

3.2.2 – Methods for Interacting with User

The following methods are used to interact with the user:

```
public static void main(String[] args)
```

Parameters: None (void method)
Return Value: None (void method)
Logic: contains the main code, which is the first thing executed by the program. All methods and classes used are first called or created in `main. main` contains the scanner that accepts strands of DNA, and the "menu" which the user chooses from when deciding which operations to use on the DNA strands. The menu is created inside a do-while loop, and uses `switch` to allow the user to select options using a console input. Main also creates two important objects used throughout the program – DNA and Matrix – which are used to manipulate user-inputted information since main is static.

```
public void acceptDNA()
```

Parameters: None (void)
Return Value: None (void)
Logic: uses `Scanner` to take in two user-inputted DNA strands, uses method `isGATC` to determine if the strands are valid. If they are not valid, an error message is outputted and the user is prompted to try again. Once both strands are valid, the method compares them using `.length` and assigns the longest to variable `longer` and the shortest to variable `shorter`.

```
public void acceptParameters(int j, int i, String longer, String
shorter)
```

Parameters: int j – shorter string length
int i – longer string length
String longer – longest string
String shorter – shortest string

Return Value: None (void)
Logic: Asks the user to input values for the match score, mismatch score, and gap penalty using a `Scanner` object. If the match score given is less than or equal to zero, an error message appears and the user is prompted to enter a valid match score. The same applies for a mismatch score greater than zero and a gap penalty greater than zero. The method then uses `System.out.println()` to create a matrix with size i*j that is populated by zeroes using for loops. Next, an NW matrix is created by populating the matrix using the Needleman-Wunsch algorithm. For the first row and column:

```
            NW.matrix[0][row + 1] = gap * (row +1)
            NW.matrix[col + 1][0] = gap * (col +1)
```

For the rest of the matrix:

```
for (int q = 0; q < i; q++) {
          for (int r = 0; r < j; r++) {
               NW.matrix[q+1][r+1] =
               algorithm(q,r,NW.matrix[q][r],NW.matrix[q][r+1],N
               W.matrix[q+1][r]);
          }
}
```

This creates a Needleman-Wunsch matrix. As well, every time the algorithm method is called, this populates the Traceback matrix.

3.2.3 – Functions for Calculation

The following methods are used to calculate values used in other parts of the program:

```
public int algorithm(int a, int b, int diag, int up, int left)
```

Parameters: int a - length of matrix
int b - width of matrix
int diag - diagonal move in traceback
int up - upwards move in traceback
int left - leftwards move in traceback
Return Value: returns maximum score
Logic: Defines a diagonal, up, and left score, as well as a max score. If when tracing back through the matrix a diagonal move is made, the diagonal score goes up and the diag location changes. The same applies to the up score and left score. Max is assigned the value of whichever score is the highest after the traceback has occurred. These scores are then used to populate a new matrix, with "D" representing a diagonal move, "L" representing a move to the left, and "U" representing a move upwards. Due to the nature of the algorithm, no moves to the right are possible. The max score is returned after this matrix is created.

```
public void traceback(char[][] tracebackMatrix, String longer,
String shorter)
```

Parameters: char[][] tracebackMatrix – matrix showing the result of the Needleman-Wunsch algorithm
String longer – longest strand
String shorter – shortest strand
Return Value: None (void)
Logic: Starting at the bottom right corner of the matrix, and using do-while loops and if statements, the tracebackMatrix method looks at each letter in the matrix and records the value before moving up, left, or diagonally depending on the pathway. As it travels it records the sequence letter at that location into a new string. By using two variables to keep track of location in the matrix, the need for a stack or queue to navigate the matrix is avoided.

3.2.4 – Other Functions

The following methods have other functions within the program:

`public DNA(String seq)`

Parameters: seq – a String variable that contains the DNA sequence inputted by the user.
Return Value: returns seq and id
Logic: Acts as a copy constructor, using `this` to assign value to String seq. Also uses `this` to assign a random value to id.

`public DNA()`

Parameters: None (void method)
Return Value: None (void method)
Logic: Acts as a default constructor for the id and seq variables.

`public Boolean isGATC(String seq)`

Parameters: seq – a String variable that contains the DNA sequence inputted by the user.
Return Value: True or False
Logic: Uses loops to check that every character entered by the user in the sequence is either “A”, “G”, “T”, or “C”. If the user entered the character as a lower-case, it is also converted to uppercase using the Java tool `toUpperCase`. The method returns false if any of the characters are not G, A, T, or C, and returns true otherwise.

`public void printDNA()`

Parameters: None (void)
Return Value: None (void)
Logic: prints both user-inputted sequences using `System.out.println()`

`public Matrix()`

Parameters: None (void)

Return Value: None (void)
Logic: Acts as a default constructor for the Matrix class and initializes all new variables to zero/null using `this.`

`public Matrix(int match, int mismatch, int gap, int j, int i,String longer,String shorter)`

Parameters: int match – match score
int mismatch – mismatch score
int gap – gap penalty
int j – shorter string length
int i – longer string length
String longer – longest string
String shorter – shortest string
Return Value: None
Logic: Acts as a copy constructor for all new variables defined in Matrix. The String variables `longer` and `shorter,` as well as `i` and `j,` are passed from main, while the remaining variables are created within Matrix.

`public void printInitializedMatrix()`

Parameters: None (void)
Return Value: None (void)
Logic: Uses `System.out.println()` as well as nested for loops to print the initialized (zero) matrix. It uses the "shorter" and "longer" variables to ensure that the longest sequence makes up the x axis of the matrix and the shorter sequence makes up the y axis.

`public void printNWMatrix()`

Parameters: None (void)
Return Value: None (void)
Logic: Uses the data created in acceptParameters along with nested for loops and `System.out.print()` to print out the matrix using the Needleman-Wunsch algorithm. The format of the matrix changes slightly depending on the gap penalty and length of sequences.

`public void traceback()`

Parameters: None (void)
Return Value: None (void)
Logic: Calls the Traceback class and creates a traceback object with all possible DNA alignments.

`public void printTracebackMatrix()`

Parameters: None (void)
Return Value: None (void)
Logic: Uses `System.out.print` and nested for loops to print the traceback matrix.

`public Traceback()`

Parameters: None (void)
Return Value: None (void)
Logic: acts as a default constructor for the Traceback class, initializing the new variables (optimalLong and optimalShort) to null.

`public Traceback(String longer, String shorter)`

Parameters: String longer, String shorter
Return Value: None (void)
Logic: Acts as a copy constructor for the Traceback class.

`public void printOptimalAlignment()`

Parameters: None (void)
Return Value: None (void)
Logic: Reverses the strings created in traceback, then prints them.

`public void printAllAlignments()`

Parameters: None (void)
Return Value: None (void)
Logic: prints all optimal alignments.

4.0 – IMPLEMENTATION

The solution was implemented using the Java language within Eclipse software. All of the code to run this project can be found in the attached files.

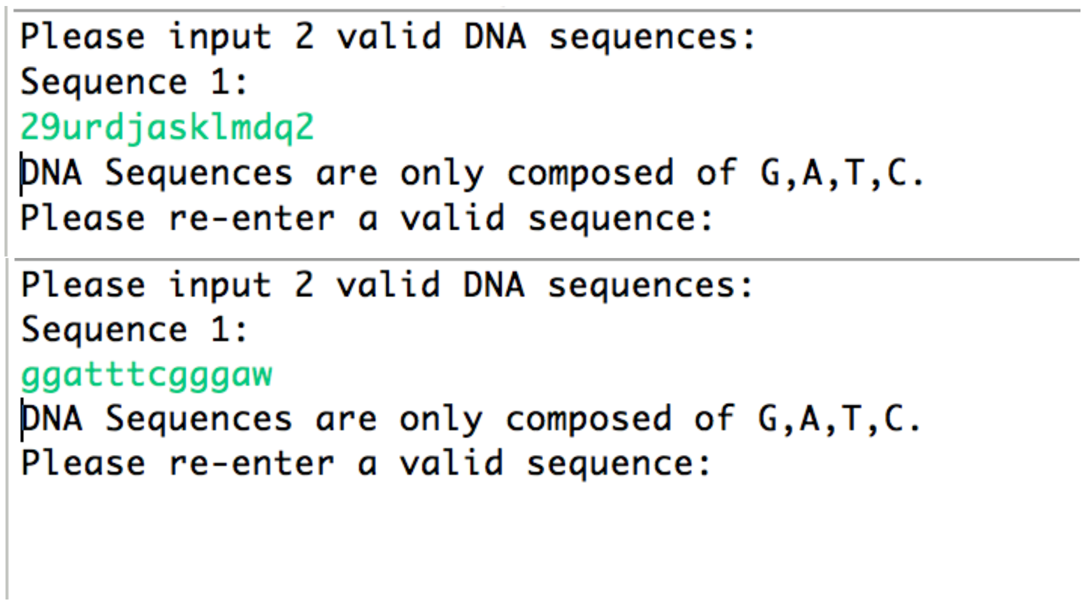
5.0 – SOFTWARE TESTING AND VERIFICATION

The software was tested using the various test cases presented in Section 3. The bugs have been worked out of the final code; however, the "output all algorithms" option was not successfully created or executed due to lack of time. The project work was divided evenly, with Emma Maddock typing the majority of the code, Megan McEwen typing the majority of the report, and both parties designing the code and solutions equally.

5.1 – Test Cases

Case 1: Incorrect Sequence Input
An incorrect sequence input by the user is a sequence including characters other than G,A,T, or C. When other characters were input by the user, the result in Figure 5.1 was generated. The case

is tested twice; once with a sequence made of entirely invalid characters, and once with a sequence that is made of all valid characters, except one. The test works for both cases because it loops through the sequence, checking that every character is either a G, A, T, or C.

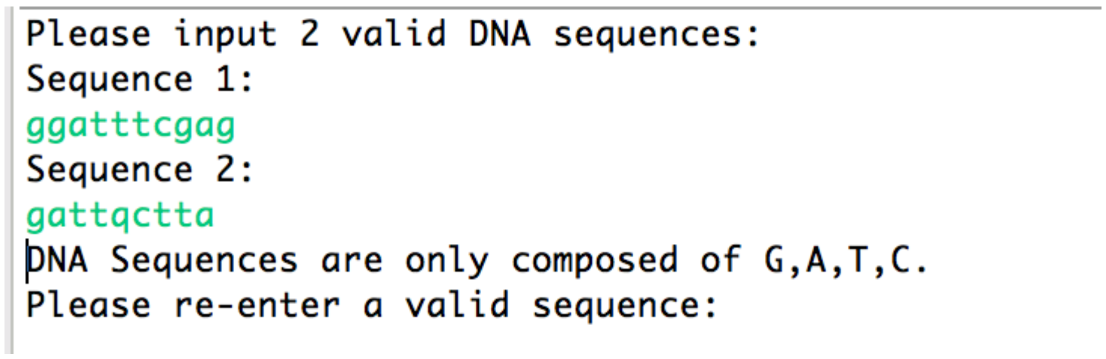


```
Please input 2 valid DNA sequences:
Sequence 1:
29urdjasklmdq2
DNA Sequences are only composed of G,A,T,C.
Please re-enter a valid sequence:
```

```
Please input 2 valid DNA sequences:
Sequence 1:
ggatttcgggaw
DNA Sequences are only composed of G,A,T,C.
Please re-enter a valid sequence:
```

Figure 5.1 – Testing for Invalid Sequences

The case also checks that both sequences are made of valid characters, informing the user they must try again even if the first sequence is valid but the second is not. This is seen in Figure 5.2:



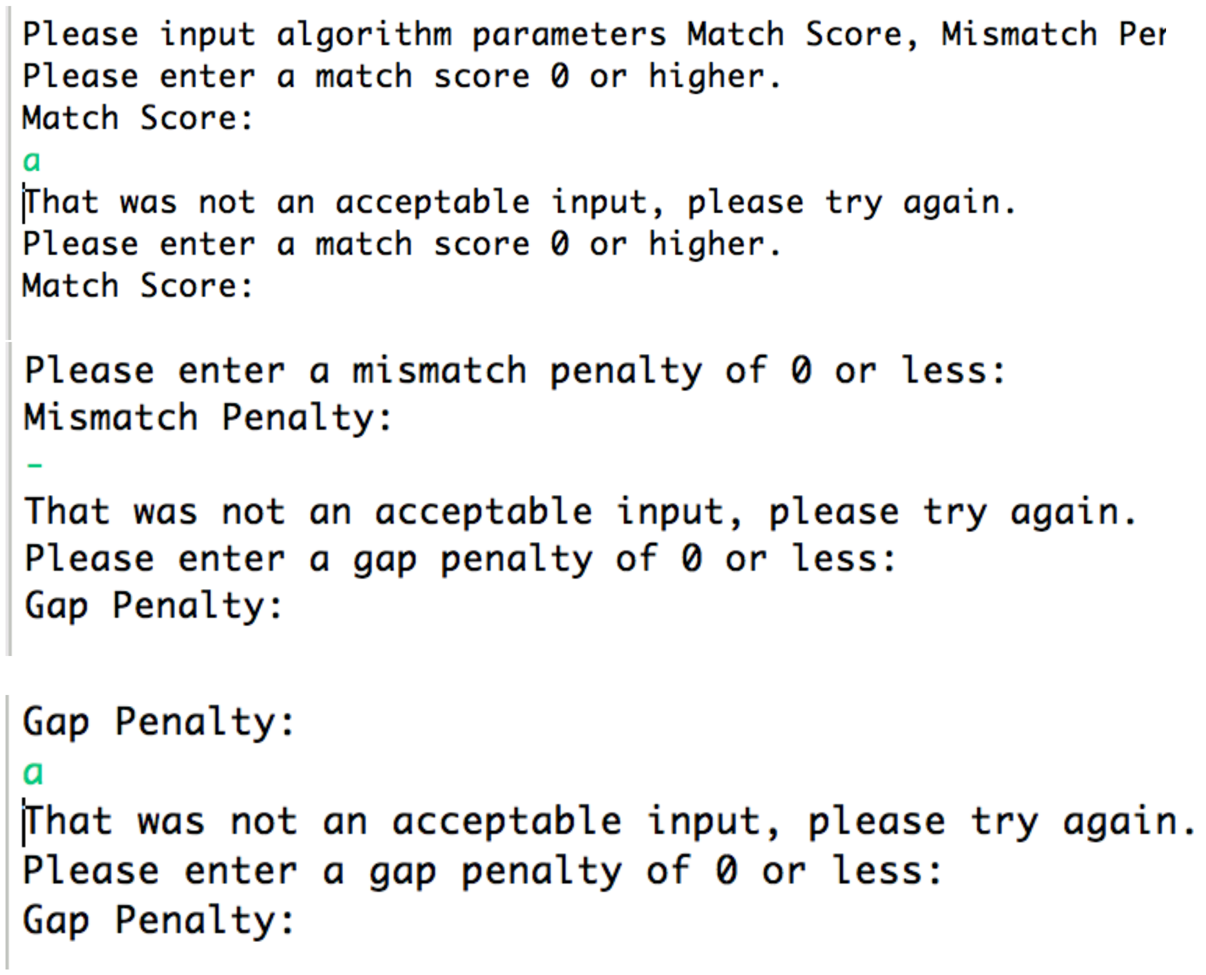
```
Please input 2 valid DNA sequences:
Sequence 1:
ggatttcgag
Sequence 2:
gattqctta
DNA Sequences are only composed of G,A,T,C.
Please re-enter a valid sequence:
```

Figure 5.2 – Testing Both Sequences to Ensure Validity

This works because the program does not exit the "sequence input" loop until both sequences are checked for validity.

Case 2: Testing Invalid Parameters

For the algorithm to run successfully, the match score must be greater than zero, the mismatch score must be less than zero, and the gap penalty must be lesser than or equal to zero. As well, the parameters must all be numbers (no characters). Figure 5.3 shows that when a user enters a character as a parameter, it is returned as invalid:



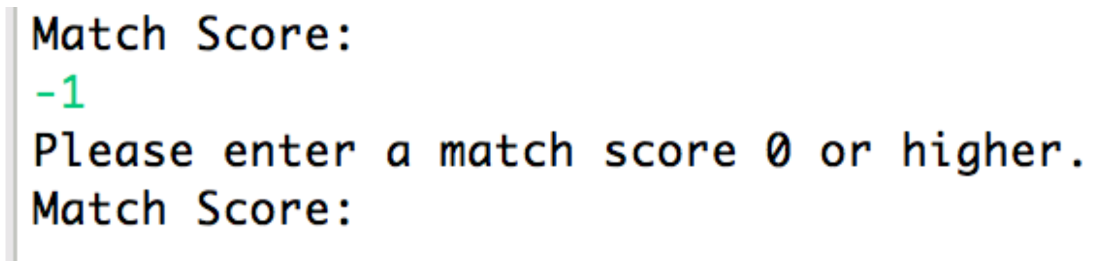
```
Please input algorithm parameters Match Score, Mismatch Per
Please enter a match score 0 or higher.
Match Score:
a
That was not an acceptable input, please try again.
Please enter a match score 0 or higher.
Match Score:
```

```
Please enter a mismatch penalty of 0 or less:
Mismatch Penalty:
-
That was not an acceptable input, please try again.
Please enter a gap penalty of 0 or less:
Gap Penalty:
```

```
Gap Penalty:
a
That was not an acceptable input, please try again.
Please enter a gap penalty of 0 or less:
Gap Penalty:
```

Figure 5.3 – Invalid Parameter Inputs (Characters)

The software also checks to make sure that the number that was inputted by the user falls within the acceptable range as described above. Figure 5.4 shows the result of inputting an integer out of these bounds:



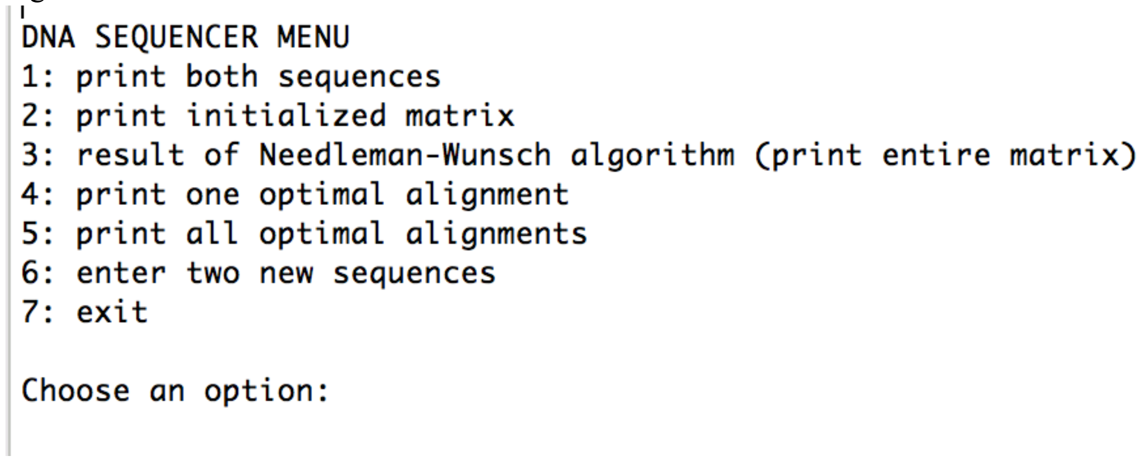
```
Match Score:
-1
Please enter a match score 0 or higher.
Match Score:
```

Figure 5.4 – Invalid Parameter Input (Integer Out-of-Bounds)

Once these two cases are checked, the program can enter the menu and begin to allow the user to choose what to do with their sequences.

5.2 – Outputs

Once the user has passed both test cases, the menu appears on their screen as seen in Figure 5.5:



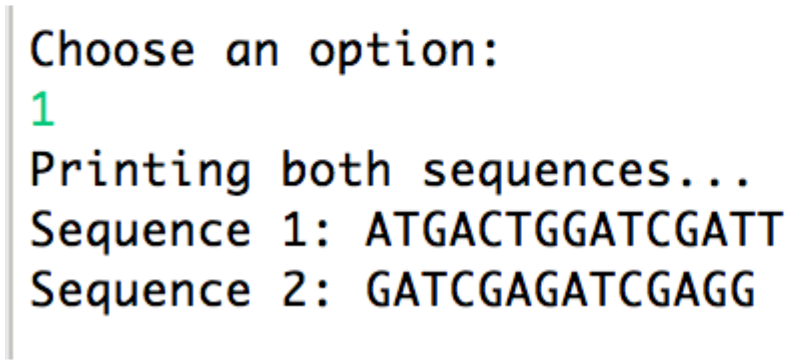
```
DNA SEQUENCER MENU
1: print both sequences
2: print initialized matrix
3: result of Needleman-Wunsch algorithm (print entire matrix)
4: print one optimal alignment
5: print all optimal alignments
6: enter two new sequences
7: exit

Choose an option:
```

Figure 5.5 – Menu

From this menu, there are seven possible outputs for the user to choose from. Each of these is shown in the sections below. For the purposes of this example, the gap penalty has been set to 0, the match score to 1, the mismatch score to -1, and the two sequences are ATGACTGGATCGATT and GATCGAGATCGAGG.

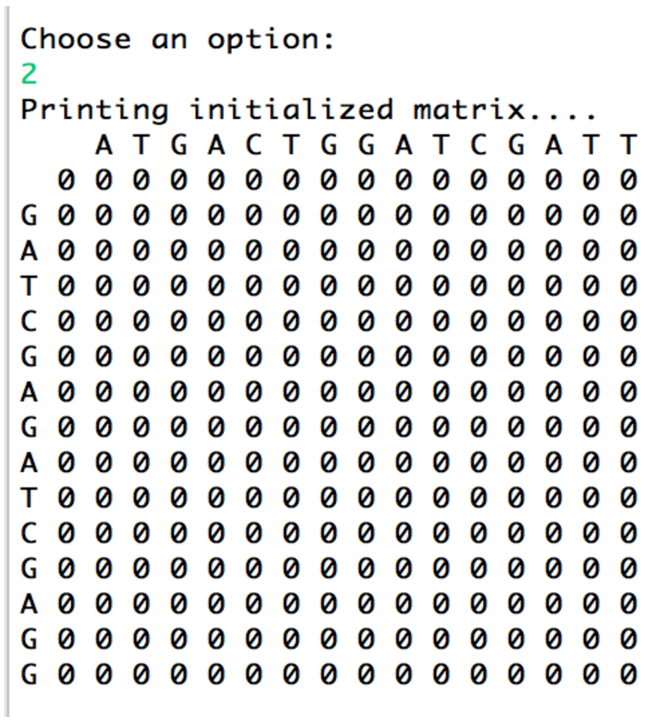
1. *Output Both Sequences*
   When Option 1 is selected, the sequences that the user inputted earlier are outputted to the screen as seen in Figure 5.6:



```
Choose an option:
1
Printing both sequences...
Sequence 1: ATGACTGGATCGATT
Sequence 2: GATCGAGATCGAGG
```

Figure 5.6 – Option 1 Output

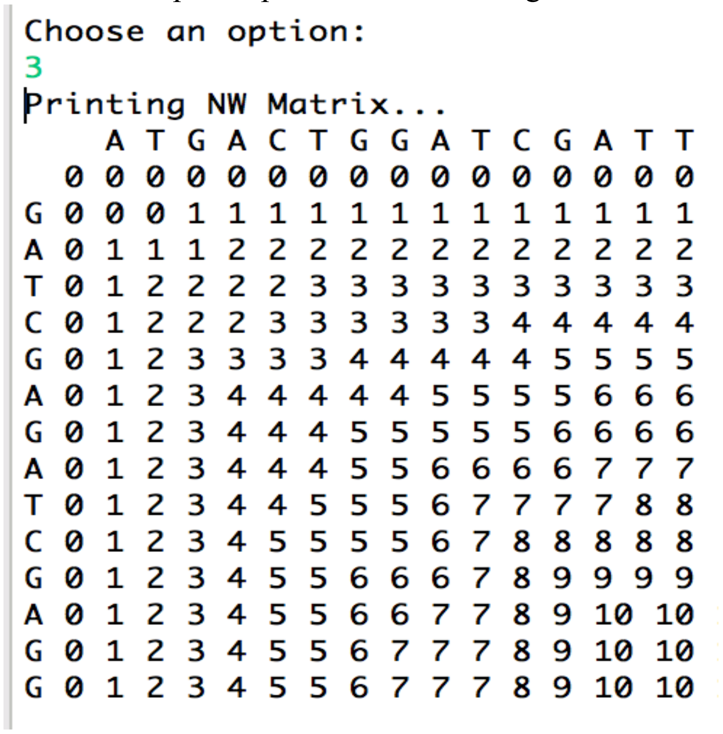
2. *Print Initialized Matrix*
   When Option 2 is selected, the sequences that the user inputted are converted into a zero matrix, with the first sequence forming the X axis and the second sequence forming the Y axis. This can be seen in Figure 5.7:



```
Choose an option:
2
Printing initialized matrix....
    A T G A C T G G A T C G A T T
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
G 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
A 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
T 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
C 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
G 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
A 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
G 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
A 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
T 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
C 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
G 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
A 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
G 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
G 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Figure 5.7 – Option 2 Output

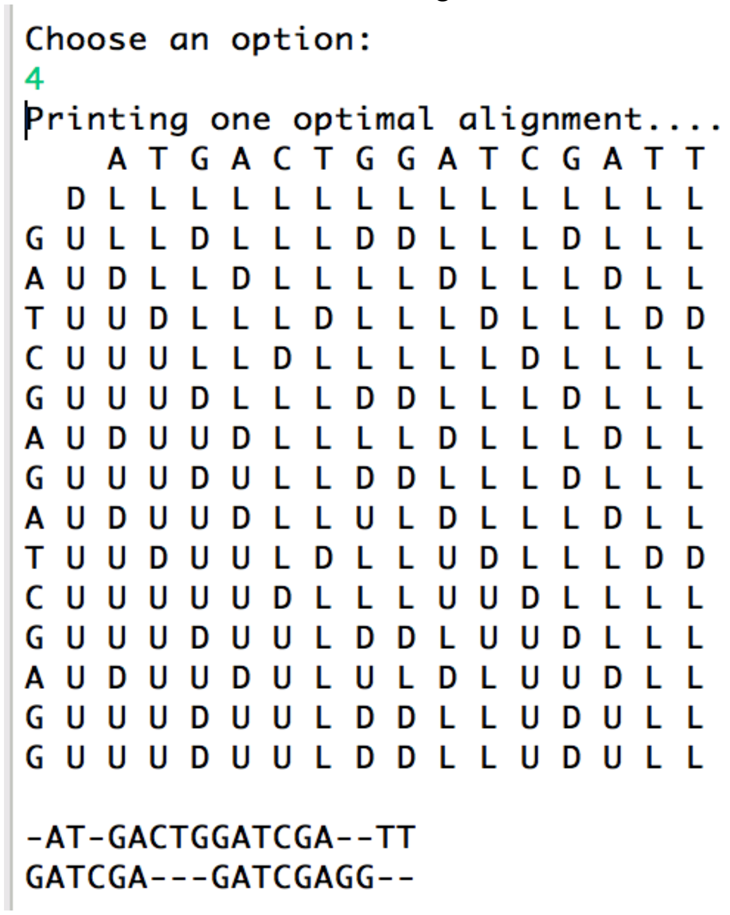
3. *Print Needleman-Wunsch Matrix*
   When this output is selected, the program uses the Needleman-Wunsch algorithm to analyze the sequences, the match score, the mismatch score, and the gap penalty to produce a filled matrix. A sample output can be seen in Figure 5.8:



```
Choose an option:
3
Printing NW Matrix...
    A T G A C T G G A T C G A T T
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
G 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1
A 0 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2
T 0 1 2 2 2 2 3 3 3 3 3 3 3 3 3 3
C 0 1 2 2 2 3 3 3 3 3 3 4 4 4 4 4
G 0 1 2 3 3 3 3 4 4 4 4 4 5 5 5 5
A 0 1 2 3 4 4 4 4 4 5 5 5 5 6 6 6
G 0 1 2 3 4 4 4 5 5 5 5 5 6 6 6 6
A 0 1 2 3 4 4 4 5 5 6 6 6 6 7 7 7
T 0 1 2 3 4 4 5 5 5 6 7 7 7 7 8 8
C 0 1 2 3 4 5 5 5 5 6 7 8 8 8 8 8
G 0 1 2 3 4 5 5 6 6 6 7 8 9 9 9 9
A 0 1 2 3 4 5 5 6 6 7 7 8 9 10 10
G 0 1 2 3 4 5 5 6 7 7 7 8 9 10 10
G 0 1 2 3 4 5 5 6 7 7 7 8 9 10 10
```

Figure 5.8 – Option 3 Output

4. *Print One Optimal Alignment*
   When this option is selected, several things happen. First, the program implements the traceback procedure, which goes through the Needleman-Wunsch-filled matrix starting from the bottom-right corner and determines the path that will produce an optimal alignment. Next, the program outputs the matrix showing which direction (starting from the bottom-right) that it took to determine the optimal alignment, with D representing a diagonal move, a U representing an upward move, and an L representing a left move. Finally, since this path determines the alignment backwards, the string is reversed and then outputted so that the user to view the final alignment.

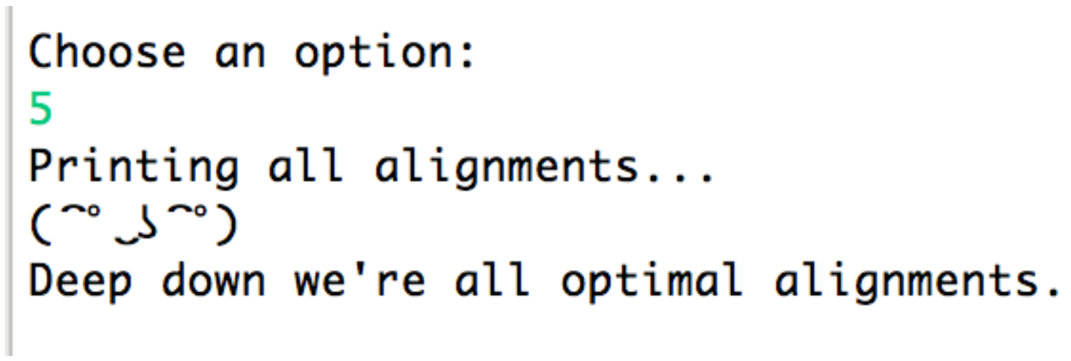


```
Choose an option:
4
Printing one optimal alignment....
    A T G A C T G G A T C G A T T
  D L L L L L L L L L L L L L L L
G U L L D L L L D D L L L D L L L
A U D L L D L L L L D L L L D L L
T U U D L L L D L L L D L L L D D
C U U U L L D L L L L L D L L L L
G U U U D L L L D D L L L D L L L
A U D U U D L L L L D L L L D L L
G U U U D U L L D D L L L D L L L
A U D U U D L L U L D L L L D L L
T U U D U U L D L L U D L L L D D
C U U U U U D L L L U U D L L L L
G U U U D U U L D D L U U D L L L
A U D U U D U L U L D L U U D L L
G U U U D U U L D D L L U D U L L
G U U U D U U L D D L L U D U L L

-AT-GACTGGATCGA--TT
GATCGA---GATCGAGG--
```

Figure 5.9 – Option 4 Output

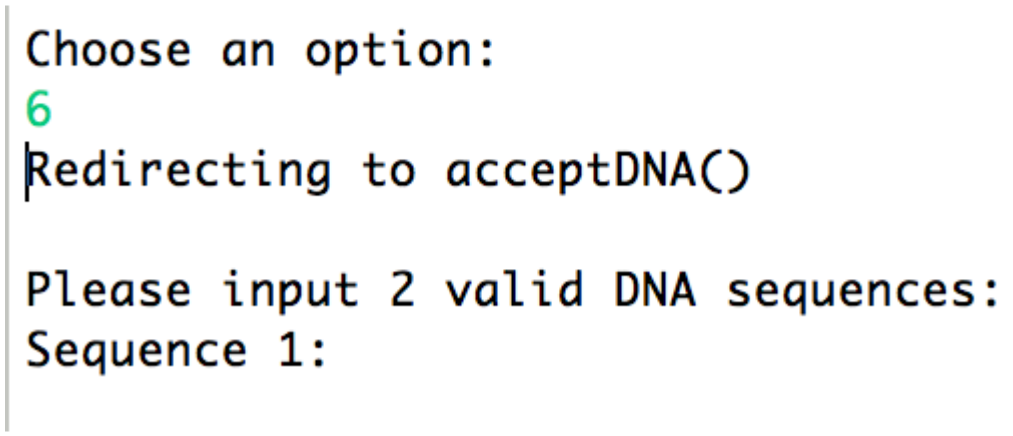
5. *Print all optimal alignments*
   This option, theoretically, would produce an output showing every optimal alignment possible. However, since this option was never fully implemented, a humorous message is outputted instead, as shown in Figure 5.10:



```
Choose an option:
5
Printing all alignments...
( ͡° ͜ʖ ͡°)
Deep down we're all optimal alignments.
```

Figure 5.10 – Option 5 Output

6. *Input New Sequences*
   When the user selects this option, the software redirects to the acceptDNA method and begins the program again, allowing the user to input new sequences and checks them for errors:

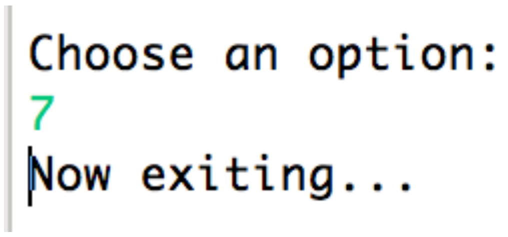


```
Choose an option:
6
Redirecting to acceptDNA()

Please input 2 valid DNA sequences:
Sequence 1:
```

Figure 5.11 – Option 6 Output

7. *Exit*
   When this option is selected, the exit command is implemented and the program exits. This is seen in Figure 5.12:



```
Choose an option:
7
Now exiting...
```

Figure 5.12 – Option 7 Output

These menu options can all be tested using different gap penalties, match scores, mismatch scores, and sequences. The program overall has only a few classes, but is very complex within those classes as it must implement several methods and operations in each class. This slows down the program run-time but makes the program straightforward for a user to understand.