Com S 229 Fall 2014

Project 2 (Part 1): Comparison and Compression of DNA Sequences

Due at 11:50 pm on Sunday, November 16

Introduction

This assignment gives you opportunities to use many features of C++ to develop a program for solving a real-world problem. These C++ features include input and output processing, object-oriented programming, class declaration and definition with variables and functions as class members, dynamic memory management, and pointers to arrays of one and two dimensions. The problem in Part 1 of Project 2 is similar to that in Part 1 of Project 1. However, Part 1 of Project 2 has an additional task, which is to compress a DNA sequence by saving another highly similar sequence as a reference and by encoding their differences. Solving this problem will help you understand what it takes to develop useful computer programs. Your task in this project is to implement the algorithm described below in C++. Please see the attached template code for the specification of each class and the main() function to be implemented.

1 Alignment-Based Compression

We describe an algorithm for compressing a DNA sequence in reference to another highly similar DNA sequence by computing an optimal global alignment of the two sequences and by encoding the differences on the alignment. See Part 1 of Project 1 for computation of an optimal alignment of two DNA sequences. Below we describe how to encode the differences on the alignment.

1.1 Encoding differences

Let $A = a_1a_2...a_m$ and $B = b_1b_2...b_n$ be two sequences of lengths m and n. The sequence A is stored on a computer with a **char** array AR of length $m + 2$ with a_i in $AR[i]$ for each $1 \leq i \leq m$ and with $AR[0]$ set to a space character and with $AR[m + 1]$ to the end-of-string character '
0'. The sequence B is stored with a **char** array of length $n + 2$. The sequence A is treated as a reference sequence, while B is to be encoded in reference of A. An optimal alignment of A and B is computed by the algorithm described in Part 1 of Project 1, with A being the top row and B the bottom row in the alignment along with a sequence of tags showing matches and differences as the middle row. The differences in the alignment are divided into three types: deletion gaps, insertion gaps and substitutions. Here a deletion gap consists only of letters from A, and an insertion gap consists only of letters from B.

The deletion and insertion gaps along with substitutions in the alignment are represented by using an array (called $operation$) of edit operations and a character array (called $subinsertion$) of letters of B in all substitutions and insertions. Both arrays are in the order of these differences in the alignment. Each element in the $operation$ array is a pair of integers $(position, indel)$, denoting a deletion, an insertion or a substitution. (Each element is implemented in C++ as a structure of two integers called $position$ and $indel$.) So the size of the $operation$ array is the total number of substitutions, deletion gaps and insertion gaps. A deletion gap of length $h > 0$ starting position i of A is represented by the pair $(i, -h)$ with i as the value of its $position$ and $-h$ as that of its $indel$. Similarly, an insertion gap of length $h > 0$ ending immediately before position i of A is represented by the pair (i, h). A substitution replacing a_i of A by b_j of B is represented by the pair $(i, 0)$. The size of the $subinsertion$ array is the total number of letters of B in all substitutions and insertions, with each element used to save a letter of B in a substitution or insertion. Below is part of an output produced on two short DNA sequences of lengths 11 and 10 included in the zip folder for this project.

```
Match Score  Mismatch Score  Gap-Open Penalty Gap-Extension Penalty
    10           -20            40                2
```

```
Sequence A: A
    Length: 11
Sequence B: B
    Length: 10
Alignment Score: -26
         Length: 12
Number of Edit Operations: 3
Length of Substitutions and Insertions: 2
        1     .    :
        1 AG TAACGACCT
          ||-|| ||--||
        1 AGCTATCG  CT
Name of the encoded sequence: B
Length of the encoded sequence: 10
Number of edit operations: 3
Length of substitutions and insertions: 2
Concatenation of subs and inserts in order: CT
Operation 0:: Position: 3  Indel: 1  Insertion: C
Operation 1:: Position: 5  Indel: 0  Substitution: T
Operation 2:: Position: 8  Indel: -2  Deletion
The original and derived sequences are identical.
Original:  AGCTATCGCT
Derived:  AGCTATCGCT
```

The size (called $editnum$) of the $operation$ array and the size (called $subinsertlen$) of the $subinsertion$ array can be computed in the traceback part of the optimal alignment computation. A revised version of the traceback part is given in the pseudocode in Figure 1. The two arrays are constructed by the pseudocode in Figure 2. The sequence B is generated from the sequence A by using the two arrays in Figure 3.

```
editnum = subinsertlen = intlen = 0;
let OA be empty;
i = 0; j = 0; mat = 'S';
while ( i ≤ m and j ≤ n )
   {
      if ( mat == 'S' )
         {
            if ( i == m and j == n )
               break;
            if ( j == n or S(i, j) == D(i, j) )
               { mat = 'D'; editnum++; continue; }
            if ( i == m or S(i, j) == I(i, j) )
               { mat = 'I'; editnum++; intlen = 0; continue; }
            append pair (a_{i+1}, b_{j+1}) to OA;
            i++; j++; if ( sub ) editnum++; subinsertlen++;
            continue;
         }
      if ( mat == 'D' )
         {
            append pair (a_{i+1}, −) to OA;
            if ( i == m − 1 or D(i, j) == S(i + 1, j) − q − r )
               mat = 'S';
            i++; continue;
         }
      if ( mat == 'I' )
         {
            append pair (−, b_{j+1}) to OA; intlen++;
            if ( j == n − 1 or I(i, j) == S(i, j + 1) − q − r )
               mat = 'S'; subinsertlen += intlen;
            j++; continue;
         }
   }
```

Figure 1. Generation of an optimal alignment.

```
i = j = 1;  opind = snind = 0;
for ( ind = 0; ind < length of the alignment; )
 { let top, mid and bot be the three alignment rows;
   take the triplet at top[ind], mid[ind] and bot[ind];
   if ( this triplet is a match )
    { i++; j++; ind++; continue; }
   if ( this triplet is a substitution )
    { operation[opind].position = i;
      operation[opind++].indel = 0;
      subinsertion[snind++] = bot[ind];
      i++; j++; ind++;
      continue;
    }
   let gaplen be the length of this gap;
   operation[opind].position = i;
   if ( this is a deletion gap )
    { i += gaplen;
      operation[opind++].indel = -gaplen;
      ind += gaplen;
    }
   else
    { j += gaplen;
      operation[opind++].indel = gaplen;
      for ( ; gaplen; gaplen-- )
       subinsertion[snind++] = bot[ind++];
    } // insertion gap
 }
```

Figure 2. Arrays operation and subinsertion.

```
  i = j = 1; B[0] = ' ';
  for ( snind = opind = 0; opind < editnum; opind++ )
   { oppos = operation[opind].position;
     indel = operation[opind].indel;

     if ( i < oppos ) // in concise code of two lines?
      { B[j+k] = A[i+k] for each k from 0 to oppos-1-i.
        i = oppos;
j += oppos-i;
      } // matches

     if ( indel < 0 ) // This part is an example of concise code.
        i -= indel;   // -indel is the deletion gap length.
     else
      {
        if ( ! indel ) i++; // just for substitution
        B[j++] = subinsertion[snind++]; // sub or insert letter
        for ( ; indel > 1; indel-- )
           B[j++] = subinsertion[snind++];
      }
   }
  if ( i <= M ) // Can be turned into concise code of two lines.
    { B[j+k] = A[i+k] for each k from 0 to M-i; } // matches
  B[N+1] = '\0';
```

Figure 3. Generation of B from A with the two arrays.

Requirements

Your task is to complete the four classes along with the main() function in the template file. The main() function creates class objects and produces results in the format of the

sample output files. The specifications including the prototypes of members in each class must not be changed so that your code can be tested by automated grading scripts. For dynamic memory management, the **new** and **delete** operators are used instead of the **malloc()**, **realloc()**, and **free()** functions. For processing input files, the **ifstream** class is used instead of the **fopen()** function. The **cout** standard output file stream is used to generate output. Except for each of the three dynamic programming matrices, output from your code is in agreement in format with the sample output files in the zip folder. It is not necessary to include output files in your submission. Your code should not have invalid memory writes or memory leaks. It is fine to use the **fatal()** in the template for generating error messages instead of using C++ exceptions.

Submission

You are required to include, in your submission, one C++ source code file including all class declarations and definitions along with the main() function. A well specified template is given in a file named project2.part1.cc. You should not change the file name. This template specifies what you need to do. You should make your code efficient and non-redundant and include as many checks as possible to catch errors and avoid segmentation faults in execution. Make sure that your source code compiles under the c++98 standard (g++ option -std=c++98) on pyrite.cs.iastate.edu. Note that the specification is subject to change. Please check on Bb for the latest clarifications.

There is no early submission credit for this project. Be sure to put down your name after the @author tag in the source file. Your zip file should be named First-name_Lastname_proj2.part1.zip. and each source file can be given any name. You may submit a draft version of your code early to see if you have any submission problem with Blackboard Learn. We will grade only your latest submission.