Com S 229 Fall 2014

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

Due at 11:50 pm on Saturday, December 6

No submission is accepted after this deadline

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, pointers to arrays of one and two dimensions, operator overloading, class and function templates, organization of header and code files, and construction of a makefile. Part 2 of Project 2 builds on Part 1 of Project 2. Your task in Part 2 is to overload an operator in a class, write class and function templates, and use the resulting code to compress a number of highly similar files of DNA sequences. In addition, you need to break a large file of code into several header and code files, and write a makefile. Doing this project will help you understand what it takes to develop useful computer programs.

1 Operator Overloading and Subclass

In the class Encoded, declare and define a public function with the following signature so that the operator <= can be used to compare two Encoded objects. In addition, add another public function named getNumDiff in this class.

```
bool operator<=(Encoded &rightobj) const;
int getNumDiff() const;
```

Consider comparing two Encoded objects $leftobj$ and $rightobj$. The comparison is performed by using the $operation$ and $subinsertion$ arrays in each object as follows. The

number of differences for an Encoded object is the total length of deletion gaps and insertion gaps plus the total number of substitutions, which is computed by using its $operation$ array. This computation is performed by the new function $getNumDiff()$, which returns the number of differences for the Encoded object. If the number of differences for $leftobj$ is less (greater) than that for $rightobj$, then the comparison $leftobj <= rightobj$ is true (false). If the two numbers are equal, then the $subinsertion$ array in each object is used to complete the comparison. If the length of the string in the $subinsertion$ array of $leftobj$ is less than or equal to than that for $rightobj$, then $leftobj <= rightobj$ is true. Otherwise, $leftobj <= rightobj$ is false.

Include a subclass named Compressed of the base class Encoded. The subclass has a constructor and a public function defining the operator $<=$; their signatures are given below. The public function overrides the function with the same signature in the base class. The subclass has no additional members.

Compressed(Alignment &obj);

bool operator<=(Encoded &rightobj) const;

Consider comparing a Compressed object $leftobj$ and an Encoded object $rightobj$. If the value returned by $getNumDiff()$ from $leftobj$ is less (greater) than that for $rightobj$, then the comparison $leftobj <= rightobj$ is true (false). If the two numbers are equal, then the $subinsertion$ array in each object is used to complete the comparison. If the length of the string in the $subinsertion$ array of $leftobj$ is less (greater) than that for $rightobj$, then $leftobj <= rightobj$ is true (false). Consider the case where the two numbers are equal. If the string returned by getSubInsertion() from $leftobj$ is less than or equal to that for $rightobj$, then $leftobj <= rightobj$ is true. Otherwise, it is false.

2 Class and Function Templates

Write a stack class template named Stack. Stack can hold pointers to objects of the same type T. Stack has four functions: $void\ push(T * obj)$, $T * pop()$, $T * peek()$, and $bool\ isEmpty()$.

Write a function template named $findMin()$ and a function template named $findMax()$.

Each function takes as an argument a stack reference of type Stack<T> and returns a pointer of type T. Note that each member in the stack is a pointer of type T. The function $findMin$ returns a pointer min of type T in the stack such that $*min <= *element$ is true for every $element$ in the stack. Similarly, the function $findMax$ returns a pointer max of type T in the stack such that $*element <= *max$ is true for every $element$ in the stack.

3 Processing Input Files

The main() function takes two or more input files, where the number of files can be any value greater than 1. Note that the number of input files is $argc - 1$. Each input file contains a DNA sequence in fasta format. These DNA sequences are highly similar. The first file is used to create a Direct object called origin of full scope, which plays the same role as the Direct object origin in Part 1. Two stacks of type Stack<Encoded> with full scope are created: one for holding Encoded objects and the other for holding Compressed objects. (Note that a pointer to a Compressed object is automatically upcast into the Encoded type when it is entered into the stack.) For each of the additional files, the file is used to create a Direct object named derived of block scope, origin and derived are used to create a Matrix object of block scope, the Matrix object is used to create an Alignment object of block scope, and the Alignment object is used to create an Encoded object with the new operator, and the pointer to the Encoded object is pushed onto its stack. Similarly, the Alignment object is used to create a Compressed object with the new operator, and the pointer to the Compressed object is pushed onto its stack.

After all the files are processed, for each of the two stacks, a smallest object and a largest object on the stack are computed by using the $findMin()$ and $findMax()$ functions with the stack as their argument. The sequence encoded in the smallest object and its name are obtained by calling the getDseq() and getDName() functions from the object, and are sent to cout along with the value reported by the function getNumDiff(). This step is repeated for the largest object.

Note that the scoring parameters are not provided on the command line. The members

in the struct scoretp variable are set to 10 for the match score, -20 for the mismatch score, 40 for the gap open penalty, and 2 for the gap extension penalty.

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.

Organization of Header and Code Files

You should produce code for Part 2 by generating a copy of the code for Part 1 and expand it with new functions and classes. You should organize your source code by creating the following header and source code files:

Alignment.hh, Alignment.cc,
Direct.hh, Direct.cc,
Encoded.hh, Encoded.cc,
Matrix.hh, Matrix.cc,
main.cc, Makefile,
prototype.hh,
template.stack.min.hh,
wrapper.hh, wrapper.cc.

Here the header file wrapper.hh contains all the wrapper function prototypes, and the code file wrapper.cc contains all the wrapper function definitions. The header file prototype.h contains all structure specifications. For each base class, there is a header file and a code file. The header file contains the specification of each member in the class, while the code file contains the definition of each function in the class. The class Compressed goes with its base class Encoded: its class specification is in the file Encoded.hh; its function definitions are in the file Encoded.cc. The header file template.stack.min.hh contains all class and function templates. The code file main.cc has only the main() function. Each header file should have a header guard. The Makefile file is used to produce an executable code file named cmps (compress) along with the clean function of removing all object and executable code files.

Requirements

You should not remove any member in any class from Part 1. You need to add new code and modify the main() function. The main() function can handle two or a higher number of files. 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. Output from your code is in agreement in format with the sample output included in this project description. 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, all the files mentioned above. These files should be put in a directory named project2.part2. You should not change any file name. 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.part2.zip. 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.