Swinburne University of Technology

Faculty of Science, Engineering and Technology

MIDTERM COVER SHEET

Subject Code: COS30008
Subject Title: Data Structures and Patterns
Assignment number and title: Midterm, Solution Design, Design Pattern, and Iterators
Due date: April 27, 2022, 23:59
Lecturer: Dr. Markus Lumpe

Your name:____________________________ **Your student ID:**_______________

Check Tutorial	Mon 10:30	Mon 14:30	Tues 08:30	Tues 10:30	Tues 12:30	Tues 14:30	Tues 16:30	Wed 08:30	Wed 10:30	Wed 12:30	Wed 14:30

Marker's comments:

Problem	Marks	Obtained
1	68	
2	120	
3	56	
4	70	
Total	314	

Midterm: Vigenère Cipher

Around 1550 Blaise de Vigenère, a French diplomat from the court of Henry III of France, developed a new scrambling technique that uses 26 alphabets to cipher a text. To be precise, de Vigenère modified a cipher invented earlier by the Italian Giovan Battista Bellaso, and turned it into a *autokey cipher* that incorporates the message into the key. The *Vigenère Cipher* is a polyalphabetic substitution technique based on a mapping table like the one shown below:

Key\Letter	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	**L**	M	N	O	P	Q	R
S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y
Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

The Vigenère cipher uses this table together with a keyword to encode a message.

To illustrate the use of this encryption method, suppose we wish to scramble the following message (Hamlet 3/1):

```
To be, or not to be: that is the question:
```

using the keyword `Relations`. First, we notice that the table provides only mappings for upper case characters. But this is not really a problem. The mapping is identical for upper and lower case characters. We just rewrite the keyword to consist of upper case characters only. When encoding a message, we convert each character to an upper case one, perform the corresponding encryption function, and output the result in either upper case or lower case depending on the original spelling. All characters not covered in the Vigenère cipher remain unchanged in the output. No keyword character is consumed in this case!

We begin by writing the keyword, followed by the message. To derive the encoded text using the mapping table, for each letter in the message, one has to find the intersection of the row given by the corresponding keyword letter and the column given by the message letter itself to pick out the encoded letter.

Keyword:	`RE LA  TI ONS TO BE  ORNO TT OBE THATISTH`
Message:	`To be, or not to be: that is the question:`
Scrambled Message:	`Lt nf, ia ccm nd dj: izoi cm ijj kcfmcbiv:`

Decoding of an encrypted message is equally straightforward. One writes the keyword plus the decoded message:

Keyword:	`RE LA  TI ONS TO BE  ORNO TT OBE THATISTH`
Scrambled Message:	`Lt nf, ia ccm nd dj: izoi cm ijj kcfmcbiv:`
Decoded Message:	`To be, or not to be: that is the question:`

This time one uses the keyword letter to pick a row of the table and then traces the row to the column containing the encoded letter. The index of that column is the decoded letter.

Problem 1 (68 marks)

To implement the Vigenère cipher, we need a data type, called `KeyProvider`, to represent the keyword. Initially, the keyword is populated with the keyword string that starts the scrambling process. For instance, in the above example the keyword string is "Relations" which is mapped to the initially keyword "RELATIONS". Every time a keyword character is consumed, either to encode or to decode a message, that character needs to be replaced with the "clear character" being processed. You can think of the keyword data type and implement it as a circular buffer. This buffer gets continuously updated with new keyword characters for the message, once the scrambling process in on the way. This behavior yields the required autokey cipher.

A suggested specification of class `KeyProvider` is shown below:

```
#pragma once

#include <string>

class KeyProvider
{
private:
  char * fKeyword;     // keyword
  size_t fSize;        // length of keyword
  size_t fIndex;       // index to current keyword character

public:

  // Initialize key provider. [10]
  // aKeyword is a string of letters.
  KeyProvider( const std::string& aKeyword );

  // Destructor, release resources. [4]
  ~KeyProvider();

  // Initialize (or reset) keyword [30]
  void initialize( const std::string& aKeyword );

  // Dereference, returns current keyword character. [4]
  char operator*() const;

  // Push new keyword character. [18]
  // aKeyCharacter is a letter (isalpha() is true).
  // aKeyCharacter replaces current keyword character.
  // Key provider advances to next keyword character.
  KeyProvider& operator<<( char aKeyCharacter );
};
```

Class `KeyProvider` maintains a keyword array whose length depends on the size of `aKeyword`. That is, the constructor for `KeyProvider` has to allocate the required heap memory for `fKeyword`, initialize its contents with the uppercase versions of the letters in `aKeyword`, and set `fIndex` to the start of the keyword. The method `initialize` performs those steps. To function properly, however, all instance variables have to be initialized with sensible values first, ideally using member initializers.

As we maintain resources in `KeyProvider`, we are required to define a destructor. The destructor has to properly release resources.

The method `initialize()` sets or resets the keyword to its initial value. This method has to release previously allocated heap memory, acquire fresh heap memory, and initialize the memory with the initial keyword `aKeyword`. After calling `initialize()`, the `KeyProvider` starts with the first keyword character.

The service interface of `KeyProvider` consists of two operators:

- `operator*()`: We use the dereference operator to access the current keyword character. If the keyword is not updated intermittently, the dereference operator has to return the same keyword character. As we use a circular buffer scheme, the current index into `fKeyword` is always valid.
- `operator<<()`: We use the shift-left operator to update the keyword with `aKeyCharacter`. That is, `aKeyCharacter` replaces the current keyword character with its uppercase variant and the keyword index advances to the next keyword character.

Class `KeyProvider` yields the fundamental abstraction for a Vigenère autokey cipher. Make sure that all necessary header files are explicitly included in `KeyProvider.cpp`.

You may test your implementation of `KeyProvider` using the following test driver (enable **`#define`** `P1` in main.cpp):

```
#include "KeyProvider.h"

int runP1( string argv[2] )
{
  cout << "Testing KeyProvider with \"" << argv[0]
       << "\" and \"" << argv[1] << "\"" << endl;

  KeyProvider lKeyWord( argv[0] );
  string& lMessage = argv[1];

  for ( char c : lMessage )
  {
    if ( isalpha( c ) )
    {
      cout << *lKeyWord;
      lKeyWord << c;
    }
    else
    {
      cout << ' ';
    }
  }

  cout << "\n";

  for ( char c : lMessage )
  {
    cout << (isalpha( c ) ? static_cast<char>(toupper( c )) : c);
  }

  cout << "\nCompleted" << endl;

  return 0;
}
```

Running the test driver should produce the following output:

```
Testing KeyProvider with "Relations" and "To be, or not to be: that is the question:"
RE LA  TI ONS TO BE  ORNO TT OBE THATISTH
TO BE, OR NOT TO BE: THAT IS THE QUESTION:
Completed
```

Problem 2 (120 marks)

Assume the implementation of `KeyProvider` is correct.

Using `KeyProvider`, define class `Vigenere` that satisfies the suggested specification as shown below:

```
#pragma once

#include "KeyProvider.h"

#define CHARACTERS 26

class Vigenere
{
private:
  char fMappingTable[CHARACTERS][CHARACTERS];
  const std::string fKeyword;
  KeyProvider fKeywordProvider;

  // Initialize the mapping table
  // Row 1:   B - A
  // Row 26:  A - Z
  void initializeTable();

public:

  // Initialize Vigenere scrambler [8]
  Vigenere( const std::string& aKeyword );

  // Return the current keyword. [22]
  // This method scans the keyword provider and copies the keyword characters
  // into a result string.
  std::string getCurrentKeyword();

  // Reset Vigenere scrambler. [6]
  // This method has to initialize the keyword provider.
  void reset();

  // Encode a character using the current keyword character and update keyword. [36]
  char encode( char aCharacter );

  // Decode a character using the current keyword character and update keyword. [46]
  char decode( char aCharacter );
};
```

Class `Vigenere` maintains a mapping table, the initial keyword string, and a keyword provider. The method `initializeTable()` sets up the mapping table. Somebody has already implemented this method:

```
void Vigenere::initializeTable()
{
  for ( char row = 0; row < CHARACTERS; row++ )
  {
    char lChar = 'B' + row;

    for ( char column = 0; column < CHARACTERS; column++ )
    {
      if ( lChar > 'Z' )
        lChar = 'A';

      fMappingTable[row][column] = lChar++;
    }
  }
}
```

The constructor has to initialize all member variables with sensible variables. Ideally, all member variables except `fMappingTable` should be initialized using member initializers. Please note that neither `KeyProvider` nor `Vigenere` define default constructors. We do not support empty keywords. Hence, you must use a member initializer for `fKeywordProvider` in order for the `Vigenere` constructor to work.

The method `getCurrentKeyword()` returns the current keyword stored in `fKeywordProvider`. There is no direct access to it. Instead, `getCurrentKeyword()` has to scan the keyword provider and assemble a result string that represents the current keyword. If the method works properly, then the keyword in `fKeywordProvider` is invariant.

The method `reset()` initializes the keyword provider with the initial keyword string. This means, we can use `reset()` to restart the Vigenère scrambler.

The methods `encode()` and `decode()` provide the scrambling operations. They implement the encoding and decoding process, respectively, as described above. When processing a character, these methods record whether the character is upper case or lower case, and each time a letter is being processed the current keyword character is updated as part of the autokey cipher process.

You can test your implementation of `Vigenere` using the following test driver (enable **#define** P2 in main.cpp):

```
#include "Vigenere.h"

int runP2( string argv[2] )
{
  Vigenere lSrambler( argv[0] );
  string lMessage = argv[1];

  // Test encoding
  cout << "Encoding \"" << lMessage
       << "\" using \"" << lSrambler.getCurrentKeyword() << "\"" << endl;

  for ( char c : lMessage )
  {
    cout << (isalpha( c ) ? static_cast<char>(toupper( c )) : c);
  }

  cout << "\n";

  string lEncodedMessage;

  for ( char c : lMessage )
  {
    lEncodedMessage += lSrambler.encode( c );
  }

  cout << lEncodedMessage << "\nCompleted" << endl;

  // Test decoding
  lSrambler.reset();

  cout << "Decoding \"" << lEncodedMessage
       << "\" using \"" << lSrambler.getCurrentKeyword() << "\"" << endl;

  for ( char c : lEncodedMessage )
  {
    cout << (isalpha( c ) ? static_cast<char>(toupper( c )) : c);
  }

  cout << "\n";
```

```
  string lDecodedMessage;

  for ( char c : lEncodedMessage )
  {
    lDecodedMessage += lSrambler.decode( c );
  }

  cout << lDecodedMessage << "\nCompleted" << endl;

  return 0;
}
```

Running the test driver should produce the following output:

```
Encoding "To be, or not to be: that is the question:" using "RELATIONS"
TO BE, OR NOT TO BE: THAT IS THE QUESTION:
Lt nf, ia ccm nd dj: izoi cm ijj kcfmcbiv:
Completed
Decoding "Lt nf, ia ccm nd dj: izoi cm ijj kcfmcbiv:" using "RELATIONS"
LT NF, IA CCM ND DJ: IZOI CM IJJ KCFMCBIV:
To be, or not to be: that is the question:
Completed
```

Problem 3 (56 marks)

Using the `Vigenere` scrambler data type, we can define an object adapter that provides us with an input file stream for Vigenère ciphers. That is, this object adapter performs encoding and decoding on-the-fly while reading the contents of a file.

Assume the implementation of class `Vigenere` is correct. Using `Vigenere`, define class `iVigenereStream` that satisfies the suggested specification as shown below:

```
#pragma once

#include <fstream>
#include <functional>

#include "Vigenere.h"

using Cipher = std::function<char ( Vigenere& aCipherProvider, char aCharacter )>;

class iVigenereStream
{
private:
  std::ifstream fIStream;
  Vigenere fCipherProvider;
  Cipher fCipher;

public:
  iVigenereStream( Cipher aCipher,
                   const std::string& aKeyword,
                   const char* aFileName = nullptr );      // [8]
  ~iVigenereStream();                                      // [2]

  void open( const char* aFileName );                      // [8]
  void close();                                            // [2]
  void reset();                                            // [10]

  // conversion operator to bool
  operator bool() { return !eof(); }

  // stream positioning
  uint64_t position() { return fIStream.tellg(); }
  void seekstart() { fIStream.clear(); fIStream.seekg( 0, std::ios_base::beg ); }

  bool good() const;                                       // [3]
  bool is_open() const;                                    // [3]
  bool eof() const;                                        // [3]

  iVigenereStream& operator>>( char& aCharacter );         // [17]
};
```

Class `iVigenereStream` is an object adapter for character file input streams. There is, however, one important characteristic of `iVigenereStream` that must be considered. Unlike standard character input file streams that ignore white space characters when performing formatted input by default, `iVigenereStream` must not skip white space characters. Hence, the underlying file stream must be processed in **binary** mode. In addition, we must access the characters of the underlying input file stream using `ifstream`'s `get()` method. This guarantees raw data access, which is required for the proper functioning of class `iVigenereStream`.

Class `iVigenereStream` does not define a default constructor. We always have to specify at least the scrambling mode `aCipher` and the keyword string `aKeyword`. The parameter `aCipher` expects a callable object. In this problem, we wish to decode a text input stream. Hence, we use the following lambda expression as parameter `aCipher`:

```
auto lCallable = []( Vigenere& aCipherProvider, char aCharacter )
                 {
                     return aCipherProvider.decode( aCharacter );
                 };
```

The lambda expression `lCallable` takes two arguments: a cipher provider and a character. It returns a decoded character as result.

The second argument to the constructor of `iVigenereStream` is `aKeyword` that we use to initialize the `fCipherProvider` member variable. (The cipher provider is a `Vigenere` object.)

The third constructor argument is a file name. If it is not `nullptr`, then the underlying fine stream must be opened.

File streams are value-based objects in C++. When these objects go out of scope, the corresponding destructor guarantees that the underlying file stream is properly closed.

The methods `open()` and `close()` have the expected semantics.

The method `reset()` has to restart an `iVigenereStream` stream. That is, this method resets the underlying cipher provider and positions the file pointer to the first character in the underlying file input stream. You can use the method `seekstart()` for this purpose.

The methods `good()`, `is_open()`, and `eof()` return the corresponding Boolean values from the underlying input file stream.

There is also a **`bool`**`()` method. This is a type conversion method that allows objects of type `iVigenereStream` to be used in a context where a Boolean value is expected. We use **`bool`**`()` to check if the underlying file input stream has reached end-of-file.

Finally, `iVigenereStream` provides formatted input for characters. This formatted input has to `get` a character from the underlying file input stream (note, raw input here). If we have not yet reached end-of-file, then the character must be processed by the cipher provider. That is, we call `fCipher` with the right arguments to obtain the result `aCharacter`.

You can test your implementation of `iVigenereStream` using the following test driver (enable **`#define`** `P3` in main.cpp):

```
#include "iVigenereStream.h"

int runP3( string argv[2] )
{
  iVigenereStream lInput( []( Vigenere& aCipherProvider, char aCharacter )
                              {
                                return aCipherProvider.decode( aCharacter );
                              } , argv[0], argv[1].c_str() );

  if ( !lInput.good() )
  {
    cerr << "Cannot open input file: " << argv[1] << endl;

    return 2;
  }

  cout << "Decoding \"" << argv[1] << "\" using \"" << argv[0] << "\"." << endl;

  char lCharacter;
  while ( lInput >> lCharacter )
  {
    cout << lCharacter;
  }
```

```
  lInput.close();

  cout << "Completed." << endl;

  return 0;
}
```

Running the test driver should produce the following output (Shakespeare's Richard III). The file `sample_3.txt` must be in the "Working Directory".

```
Decoding "sample_3.txt" using "Relations".

                    ACT I

                    SCENE I

               London. A Street.

               Enter Gloucester.

Gloucester. Now is the winter of our discontent
       Made glorious summer by this sun of York;
       And all the clouds that lour'd upon our house
       In the deep bosom of the ocean buried.
       Now are our brows bound with victorious wreaths;
       Our bruised arms hung up for monuments;
       Our stern alarums changed to merry meetings;
       Our dreadful marches to delightful measures.
       Grim-visag'd war hath smooth'd his wrinkled front;
       And now, - instead of mounting barbed steeds,
       To fright the souls of fearful adversaries, -
       He capers nimbly in a lady's chamber
       To the lascivious pleasing of a lute.
       But I, that am not shap'd for sportive tricks,
       Nor made to court an amorous looking-glass;
       I, that am rudely stamp'd, and want love's majesty
       To strut before a wanton ambling nymph;
       I, that am curtail'd of this fair proportion,
       Cheated of feature by dissembling nature,
       Deform'd, unfinish'd, sent before my time
       Into this breathing world, scarce half made up,
       And that so lamely and unfashionable
       That dogs bark at me, as I halt by them;
       Why, I, in this weak piping time of peace,
       Have no delight to pass away the time,
       Unless to see my shadow in the sun
       And descant on mine own deformity:
       And therefore, since I cannot prove a lover,
       To entertain these fair well-spoken days,
       I am determined to prove a villain,
       And hate the idle pleasures of these days.
       Plots have I laid, inductions dangerous,
       By drunken prophecies, libels, and dreams,
       To set my brother Clarence and the king
       In deadly hate the one against the other:
       And if King Edward be as true and just
       As I am subtle, false, and treacherous,
       This day should Clarence closely be mew'd up,
       About a prophecy, which says, that G
       Of Edward's heirs the murderer shall be.
       Dive, thoughts, down to my soul: here Clarence comes.

       Brother, good day: what means this armed guard
       That waits upon your Grace?
Completed.
```

Problem 4 (70 marks)

Assume the implementation of class `iVigenereStream` is correct. We can define a forward iterator that encapsulates an `iVigenereStream` stream and use this iterator to encode and decode characters in a for-each range loop, respectively. For example, we can read an encoded stream via a forward iterator and send each character to a target stream like below:

```
for ( char c : VigenereForwardIterator( lInput ) )
{
    cout << c;
}
```

Here, `lInput` is an `iVigenereStream` stream that performs on-the-fly encoding while reading a text input file. In the for-each-loop we only see the encoded characters.

A suggested specification of class `VigenereForwardIterator` is shown below:

```
#pragma once

#include "iVigenereStream.h"

class VigenereForwardIterator
{
private:
  iVigenereStream& fIStream;
  char fCurrentChar;
  bool fEOF;

public:

  VigenereForwardIterator( iVigenereStream& aIStream );               // [10]

  // forward iterator interface
  char operator*() const;                                             // [2]
  VigenereForwardIterator& operator++();      // prefix increment        [10]
  VigenereForwardIterator operator++( int );  // postfix increment       [10]

  bool operator==( const VigenereForwardIterator& aOther ) const;     // [8]
  bool operator!=( const VigenereForwardIterator& aOther) const;      // [4]

  VigenereForwardIterator begin() const;                              // [16]
  VigenereForwardIterator end() const;                                // [10]
};
```

Class `VigenereForwardIterator` defines a standard forward iterator. However, streams in C++ are generally not copiable. Hence, we need to maintain access to the input stream `fIStream` via a reference. This requires `fIStream` to be initialized via a member initializer.

The iterator has to always read one character. This can be achieved via the increment operators. When we construct a `VigenereForwardIterator` object, then its `fCurrentCharacter` has to be set to the first character in the stream. If the underlying stream runs in encode mode, then `fCurrentCharacter` is set to its encoded equivalent. Otherwise, it is set to the "clear value".

The iterator does not maintain a dedicated index. Instead, it uses a `fEOF` flag, which is true if and only if the underlying stream has reached end-of-file. Two iterators of type `VigenereForwardIterator` are equal (i.e., they are positioned on the same character), when their respective addresses are the same and if their `fEOF` flags are the same. The latter is used to define an end iterator.

The dereference operator returns the value of `fCurrentChar`. The increment operators advance the iterator. That is, increment means we input the next character (with on-the-fly scrambling) and test whether the iterator has reached end-of-file.

The auxiliary methods `begin()` and `end()` are required to compile the for-each range loop. The method `begin()` returns a copy of the current iterator whose stream has been reset and which is positioned on the first character, if such a character is available. The method `end()` also returns a copy of the current iterator, but with its `fEOF` flag set to true. An iterator positioned at the end has logically reached end-of-file.

You can test your implementation of `VigenereForwardIterator` using the following test driver (enable **`#define`** `P4` in main.cpp):

```
#include "VigenereForwardIterator.h"

int runP4( string argv[2] )
{
  iVigenereStream lInput( []( Vigenere& aCipherProvider, char aCharacter )
                              {
                                return aCipherProvider.encode( aCharacter );
                              } , argv[0], argv[1].c_str() );

  if ( !lInput.good() )
  {
    cerr << "Cannot open input file: " << argv[1] << endl;

    return 2;
  }

  cout << "Forward Iterator Decoding \"" << argv[1]
       << "' using \"" << argv[0] << "\"." << endl;

  for ( char c : VigenereForwardIterator( lInput ) )
  {
    cout << c;
  }

  lInput.close();

  cout << "Completed." << endl;

  return 0;
}
```

Running the test driver should produce the following output (Shakespeare's Richard III). The file `sample_4.txt` must be in the "Working Directory".

```
Forward Iterator Encoding "sample_4.txt' using "Relations".

                    SHF J

                    MLTBX J

               Oiwwrs. O Xcdtsx.

               Tbuxl Yqtohsmywy.

Sajxhxmywy. Zdr lx mbj owcqnk in tra rcxudtizfx
       Vtgt uftfcbvw xbybwa qt madf fzf qe Swad;
       Tir prk izp dzsvpe npfw xdpv'w oxph ajm zspit
       Wc ozm szxu kcmwr sk yxg dvtnc hozntg.
       Spk cmw xzv pglxk gdpff oxqa xxxhsorico fuyplqh;
       Jno twvcaxs vjok cdgl yq xbk ujbbhutik;
       Bjf ngjfh tavjngx uvbzhwy gh pmsff riyivsyk;
       Nhw ilnokyjg eejhiiy oa qfdlommzjp rqjzclkn.
       Senn-odkfz'k oje djmi zqlplp'e bql jgxhspmm yogwh;
       Lzi ruo, - xbmuser dc vcnhyjrv hngwsx bhlgek,
       Vt jkclmx mbt ygdsa iz njtgage pjbjskgmufw, -
       Dj utqwax gqremo nf t zjqa'e bqonnfv
       Sh wpf ycxuckcwze qehjorcb hv m qvmn.
```

```
        Pbi O, utvn fo iic mpbj'e scg miwsjmbt lkyrcm,
        Wkw gsmh eh qdmeu es ubrgjmm mcpxxfv-bemhh;
        T, cvha mn knmytz munek'h, fzc puog bsws'w jbxyeiu
        Yh fubzm vdzdky s rupyuc srcijba cmncj;
        U, cvhh zz scanijf'e bi ozct ommg vlwyhxurgd,
        Uwuplym dt imfuowi qe jntmzegnhrp gtyhtq,
        Msmcsg'y, msjntxku'h, nstc pnywvx rm nkrk
        Xfyb sbrf gasuibqwz ygwmx, aloyzt zmpy pbvh zx,
        Bzj giey ne moqytz ugs gosfegjcrvprf
        Mpji rpie guzl ux bl, tu J zlmn od uanu;
        Xts, K, hh bmvp edjt dcxrgd yjxu xv yshwn,
        Ufkk dt ehqqhdy hd tfeb heus ixf mbnb,
        Vmfmxm cb xzs yd laushb na sam tyc
        Xwr xmxvvbu cr qngh pkh sssxfrxqm:
        Esj izrayepfi, mqsuj O rssgxh swxyf o zdpuj,
        Ik joftnyscc yvyxw zbrf qmqe-xvptwk imkl,
        Y px isxfqfrorh yi ujbes f zcabsxj,
        Foz qmff cvf whtf jqyixdvqx er yixnw itny.
        Jttmx lbux Y xpcw, qozzlfjxrb reijyadil,
        Fz ryzfzzg rqshcsnnsi, dxrmqv, jsw pagfyl,
        Uc wil rz okiiajl Pkcjthkj sqp uzj ylsh
        Wr xmfoum ojhi yii amm bafcvxi hmf vuqsk:
        Uhl nu Eqsy Frajxo ks hx xovw epi knml
        Vx J oq cpunmx, obylz, chp yxfmvmffsok,
        Yila isn naicuw Gmzkmcxq goatwqm ej pql'w zb,
        Zdthy x tmeqjtxs, xxarx afbr, qpjw O
        Hg Dwqisx'z wknvp uzi fcwmwkyz xuvdp gw.
        Iaom, utawllco, iiec ov us lsji: vygr Bepmqvhw hryfk.

        Gfrykte, lhqv sug: bzhi bieor qpjm nwnsw acjke
        Lufx dvjlw oxph vpdl Zmqrs?
Completed.
```

The output should match the contents of `sample_3.txt`. It is the encoded text of Shakespeare's Richard III, Act I, Scene I.

Submission deadline: Wednesday, April 27, 2022, 23:59.
Submission procedure: PDF of printed code for code of **`KeyProvider`**, **`Vigenere(MT)`**, **`iVigenereStream`**, and **`VigenereForwardIterator`**.