Swinburne University of Technology

Faculty of Science, Engineering and Technology

ASSIGNMENT COVER SHEET

Subject Code:	COS30008
Subject Title:	Data Structures & Patterns
Assignment number and title:	3 - Iterators
Due date:	April 16, 2019, 10:30
Lecturer:	Dr. Markus Lumpe

Your name:____________________________ **Your student id:**________________

Check Tutorial	Wed 08:30	Wed 10:30	Wed 12:30	Wed 14:30	Thurs 08:30	Thurs 10:30	Thurs 12:30	Thurs 14:30	Fri 10:30

Marker's comments:

Problem	Marks	Obtained
1	50	
2	46	
3	10	
Total	106	

Extension certification:

This assignment has been given an extension and is now due on ____________

Signature of Convener:_________________________

Problem Set 4: Iterators


[1]

In mathematics, the *Fibonacci numbers* (or *Fibonacci sequence*) are positive numbers in the following sequence

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, …

For n ≥ 3, we can define this sequence recursively by

Fibonacci(n) = Fibonacci(n – 1) + Fibonacci(n – 2),

with seed values

Fibonacci(1) = 1 and Fibonacci(2) = 1.

Fibonacci numbers appear in numerous places, including computer science and biology. Unfortunately, evaluating a Fibonacci sequence for a given n in a recursive and bottom-up fashion is computationally expensive and may exceed available resources (in terms of both space and time). The recursive definition calculates the smaller values of Fibonacci(n) first and then builds larger values from them.

An alternative mathematical formulation of the Fibonacci sequence is due to *dynamic programming*, a technique developed by Richard E. Bellmann in the 1940s while working for the RAND Corporation. Dynamic programming uses memorization to save values that have already been calculated. This yields a top-down approach that allows Fibonacci(n) to be split into sub-problems and then calculate and store values. This method produces a very efficient iterative algorithm to generating the Fibonacci sequence.

The iterative formulation of the Fibonacci sequence uses two storage cells, `previous` and `current`, to keep track of the values computed so far:

```
Fibonacci( n ) =
  previous := 0;
  current := 1;
  for i := 1 to n do
   next := current + previous;
   previous := current;
   current := next;
   end;
```

For n ≥ 1, this algorithm produces the desired sequence in linear time, but only requiring constant space.

[1] Source: http://en.wikipedia.org/wiki/File:Fibonacci.png

Problem 1: Class FibonacciSequence

Using the dynamic programming solution, we can construct a C++ class, called `FibonacciSequence`, that produces the Fibonacci sequence up to a given n. Objects of this class *generate* the Fibonacci sequence via repeatedly calling the `advance` method. The Fibonacci sequence is infinite. We can model the same behavior. However, for practical purposes, we shall limit the sequence to the first 20 numbers. Otherwise, the corresponding iterator would not work in the expected way. The following class specification suggests a possible solution:

```
#pragma once

// forward declaration to break mutual recursion of
// FibonacciSequence and FibonacciSequence
class FibonacciSequenceIterator;

class FibonacciSequence
{
private:
  unsigned long fPrevious;    // previous Fibonacci number (initially 0)
  unsigned long fCurrent;     // current Fibonacci number (initially 1)
  unsigned long fPosition;    // position in the sequence (starts with 1)
  unsigned long fLimit;       // set limit for sequence (0 for no limit)

public:

  // Default constructor to set up a Fibonacci sequence (aLimit = 0 means infinite)
  FibonacciSequence( unsigned long aLimit = 20 );

  // get current Fibonacci number
  const unsigned long& current() const;

  // advance to next Fibonacci number, may throw out_of_range exception on
  // exceeding limit
  void advance();

  // extract sequence limit
  const unsigned long& getLimit() const;

  // return new iterator positioned at start
  FibonacciSequenceIterator begin();

  // return new iterator positioned at a desired limit
  FibonacciSequenceIterator end();
};
```

A `FibonacciSequence` object requires four member variables. The values `fPrevious` and `fCurrent` serve are the storage cells to compute the Fibonacci sequence. The values `fPosition` and `fLimit` denote the position in the sequence and the maximum position in the sequence, respectively. The constructor has to properly set up these variables.

The methods `current`, `advance`, and `getLimit` are the service functions of class `FibonacciSequence`. The function `current` returns the current Fibonacci number and the function `getLimit` the maximum position in the sequence. The method `advance` computes the next Fibonacci number according the dynamic programming scheme shown above. It must test, if the maximum position has been exceeded. In this case, an `out_of_range` exception must be thrown. Please note that a sequence can be infinite. In this case, the limit is 0 and we will never run out of numbers to generate. Design your tests accordingly.

The methods `begin` and `end` return corresponding iterators for `FibonacciSequence` objects. The method `begin` has to reset (i.e, restart) the Fibonacci sequence and return an iterator that is positioned at the first number. The method end has to return an iterator that is positioned just after the limit of the Fibonacci sequence. It must not change the `FibonacciSequence` object though.

Problem 2: Class FibonacciSequenceIterator

The class `FibonacciSequenceIterator` implements a standard forward iterator for `FibonacciSequence` objects. It maintains to instance variables: a pointer the a `FibonacciSequence` object and the iterator position. In this assignment, you will need to use pointer access to call methods. For example, to obtain the current Fibonacci number from the sequence object you need to write:

`fPtrToSequenceObject->current()`

where `fPtrToSequenceObject` is the pointer to a `FibonacciSequence` object.

The following class specification suggests a possible solution:

```
#pragma once

// forward declaration to break mutual recursion of
// FibonacciSequence and FibonacciSequence
class FibonacciSequence;

class FibonacciSequenceIterator
{
private:
  FibonacciSequence* fPtrToSequenceObject;    // pointer to sequence object
  unsigned long fIndex;                       // current iterator position

public:

  // iterator constructor, takes a pointer to a FibonacciSequence object
  // pointers can be compared
  FibonacciSequenceIterator( FibonacciSequence* aPtrToSequenceObject,
                             unsigned long aStart = 1 );

  // iterator methods

  const unsigned long& operator*() const;       // return current Fibonacci number
  FibonacciSequenceIterator& operator++();      // prefix, next Fibonacci number
  FibonacciSequenceIterator operator++( int ); // postfix (extra unused argument)
  bool operator==( const FibonacciSequenceIterator& aOther ) const;
  bool operator!=( const FibonacciSequenceIterator& aOther) const;

  // iterator methods

  // return new iterator positioned at start
  FibonacciSequenceIterator begin() const;

  // return new iterator positioned at limit
  FibonacciSequenceIterator end() const;
};
```

The implementation of `FibonacciSequenceIterator` follows standard practice and is similar to the `IntArrayIterator` and the `CharacterCounterIterator` studied in class and tutorials. This time you can and must compare the underlying collection via a simple pointer comparison. The methods `begin` and `end` should just forward the requests to the underlying `FibonacciSequence` object.

Build the C++ console application that takes one argument (i.e., a number string) and outputs the corresponding Fibonacci sequence to the console screen. Use the following main function in your application (see C++ reference for details on `atoi`):

```
#include <iostream>
#include <cstdlib>

#include "FibonacciSequence.h"
#include "FibonacciSequenceIterator.h"

using namespace std;

int main( int argc, char* argv[] )
{
    if ( argc < 2 )
    {
        cerr << "Missing argument!" << endl;
        cerr << "Usage: FibonacciIterator number" << endl;
        return 1;
    }

    cout << "Fibonacci sequence up to " << argv[1] << endl;

    FibonacciSequence lSequence( atoi( argv[1] ) );

    // C++11 range loop
    unsigned long i = 1;
    for ( const unsigned long& n : lSequence )
    {
        cout << i++ << ":\t" << n << endl;
    }

    // Old-style loops

    cout << "Old-style:" << endl;

    FibonacciSequenceIterator lIterator = lSequence.begin();
    unsigned long j = 1;

    for ( ; lIterator != lIterator.end(); lIterator++ )
    {
        cout << j++ << ":\t" << *lIterator << endl;
    }

    cout << "Once more, prints sequence 2..21:" << endl;

    FibonacciSequenceIterator lIterator2 = lIterator.begin();
    unsigned long k = 1;

    do
    {
        cout << ++k << ":\t" << *(++lIterator2) << endl;
    } while ( lIterator2 != lIterator2.end() );

    return 0;
}
```

Your program should produce output similar to the one shown below:

```
Fibonacci sequence up to 20
1:      1
2:      1
3:      2
4:      3
5:      5
6:      8
7:      13
8:      21
9:      34
10:     55
11:     89
12:     144
13:     233
14:     377
15:     610
16:     987
17:     1597
18:     2584
19:     4181
20:     6765
Old-style:
1:      1
2:      1
3:      2
4:      3
5:      5
6:      8
7:      13
8:      21
9:      34
10:     55
11:     89
12:     144
13:     233
14:     377
15:     610
16:     987
17:     1597
18:     2584
19:     4181
20:     6765
Once more, prints sequence 2..21:
2:      1
3:      2
4:      3
5:      5
6:      8
7:      13
8:      21
9:      34
10:     55
11:     89
12:     144
13:     233
14:     377
15:     610
16:     987
17:     1597
18:     2584
19:     4181
20:     6765
21:     10946
```

Problem 3:

Why does the last output run to 21 and prints the Fibonacci number 10946?

Submission deadline: Tuesday, April 16, 10:30

Submission procedure: on paper, no electronic submission, code of class `FibonacciSequence` and `FibonacciSequenceIterator`.