# Swinburne University of Technology

*School of Science, Computing and Emerging Technologies*

## ASSIGNMENT COVER SHEET

| | |
|---|---|
| **Subject Code:** | COS30008 |
| **Subject Title:** | Data Structures & Patterns |
| **Assignment number and title:** | 2 - Iterators |
| **Due date:** | Sunday, 13 April, 2025, 23:59 |
| **Lecturer:** | Dr. Markus Lumpe |

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

Marker's comments:

| Problem | Marks | Obtained |
|---|---|---|
| 1 | 44 | |
| 2 | 64 | |
| Total | 108 | |

**Extension certification:**

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

Signature of Convener:_________________________

# Problem Set 2: Iterators

[1]

In mathematics, the *Fibonacci numbers* (or the *Fibonacci sequence*) are an infinite series of positive numbers with the following pattern

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

For $n \geq 3$, we can define this sequence recursively by

$$\text{Fibonacci}( n ) = \text{Fibonacci}( n - 1 ) + \text{Fibonacci}( n - 2 ),$$

with seed values

$$\text{Fibonacci}( 1 ) = 1 \text{ and } \text{Fibonacci}( 2 ) = 1.$$

Fibonacci numbers appear in numerous fields, including computer science and biology. Unfortunately, evaluating the Fibonacci sequence for a given n in a recursive and bottom-up fashion is computationally expensive. It may exceed available resources (in terms of space and time). The recursive definition calculates the smaller values of Fibonacci( n ) first and then builds larger values from them. This process has $O(2^n)$ complexity (see lecture slides page 179).

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, process the sub-problems first, and store intermitted values. This method produces a very efficient iterative algorithm for 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
    current := current + previous;
    previous := current - previous;
  end;
```

For $n \geq 1$, this algorithm produces the desired sequence in linear time and constant space.

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

## Problem 1:

Using the dynamic programming approach, we can construct a C++ class, named `FibonacciSequence`, that produces the correct Fibonacci sequence. The sequence generated is infinite. Without a defined endpoint, a loop over a Fibonacci sequence would run forever. Eventually, the process ends due to the maximum integer value representable on a given computer architecture (e.g., $2^{64}$-1). Nevertheless, the `FibonacciSequence` class satisfies the concept `std::forward_iterator`. Objects of class `FibonacciSequence` can be used everywhere a standard forward iterator is expected.

```
#pragma once

#include <string>
#include <cstdint>
#include <concepts>

constexpr uint64_t MAX_FIBONACCI = 93;

class FibonacciSequence
{
private:

  uint64_t fPrevious;     // previous Fibonacci number (initially 0)
  uint64_t fCurrent;      // current Fibonacci number (initially 1)

public:

  using iterator = FibonacciSequence;
  using difference_type = std::ptrdiff_t;         // to satisfy concept weakly_incrementable
  using value_type = uint64_t;                    // to satisfy concept indirectly_readable

  FibonacciSequence() noexcept;

  // Retrieve the current Fibonacci number
  const uint64_t& operator*() const noexcept;

  FibonacciSequence& operator++() noexcept;       // prefix, next Fibonacci number
  FibonacciSequence operator++(int) noexcept;     // postfix (extra unused argument)

  bool operator==( const FibonacciSequence& aOther ) const noexcept;

  // Position iterator at start
  void begin() noexcept;

  // Position iterator at no limit
  void end() noexcept;
};

static_assert(std::forward_iterator<FibonacciSequence>);
```

Class `FibonacciSequence` defines two member variables. The values `fPrevious` and `fCurrent` are the storage cells to compute the Fibonacci sequence. The constructor has to initialize these variables using a member initializer list.

The dereference operator, `operator*()`, returns a constant reference to the current Fibonacci number (the number at which the iterator is positioned). There is no need to reveal the value of the previous Fibonacci number.

Objects of class `FibonacciSequence` *generate* the Fibonacci sequence by repeatedly calling one of the available increment operators. The increment operators compute the Fibonacci numbers using the approach shown in the pseudo-code (loop body). Only one increment operator performs the calculation. Please note that we can ignore overflow conditions due to computer architecture limits. Theoretically, we can fix this problem using a system with infinite precision. On a 64-bit architecture, the increment operators produce an incorrect result from the 93rd Fibonacci number onwards.

The auxiliary methods `begin()` and `end()` return a Fibonacci sequence iterator positioned at the first number and infinity, respectively. We use integer values, however, there is no integer value to represent infinity. Instead, we can use a combination of the values for `fPrevious` and `fCurrent` that never yields a valid Fibonacci number. The test driver in `Main.cpp` verifies the necessary setup. It uses a comma expression, `(` *`Statement`*`,` *`Expression`* `)`. The result of a comma expression arises from evaluating the *`Expression`* in a context where the *`Statement`* has been computed. There are four occurrences of comma expressions. Each comma expression sets the current iterator to either the start or infinity. The expression accesses either the current or next value.

The file `Main.cpp` contains a test function to check your implementation. Uncomment **`#define`** `P1` and compile your solution. Your program should produce the following output:

```
Fibonacci sequence for unsigned long long:
 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
…
81: 37889062373143906
82: 61305790721611591
83: 99194853094755497
84: 160500643816367088
85: 259695496911122585
86: 420196140727489673
87: 679891637638612258
88: 1100087778366101931
89: 1779979416004714189
90: 2880067194370816120
91: 4660046610375530309
92: 7540113804746346429
93: 12200160415121876738
FibonacciSequence is sound.
```

## Problem 2:

The class `FibonacciSequenceIterator` implements a standard forward iterator for `FibonacciSequence` objects. It maintains two instance variables: an object of class `FibonacciSequence` and the iterator position. The class `FibonacciSequenceIterator` defines a wrapper for `FibonacciSequence` objects to produce a defined and viable sequence over the representable Fibonacci numbers on a given architecture. In this problem, we consider a 64-bit computer system. The iterator must stop after the 92nd Fibonacci number has been computed.

```
#pragma once

#include "FibonacciSequence.h"

class FibonacciSequenceIterator
{
private:

  FibonacciSequence* fSequence;   // sequence object
  uint64_t fIndex;                // current iterator position

public:

  using iterator = FibonacciSequenceIterator;
  using difference_type = std::ptrdiff_t;         // to satisfy concept weakly_incrementable
  using value_type = uint64_t;                    // to satisfy concept indirectly_readable

  // iterator constructor
  FibonacciSequenceIterator( FibonacciSequence* aSequence = nullptr,
                             uint64_t aStart = 0 ) noexcept;

  // iterator methods
  const uint64_t& operator*() const noexcept;          // return current Fibonacci number
  FibonacciSequenceIterator& operator++() noexcept;    // prefix, next Fibonacci number
  FibonacciSequenceIterator operator++(int) noexcept; // postfix (extra unused argument)

  bool operator==( const FibonacciSequenceIterator& aOther ) const noexcept;

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

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

template<typename T>
concept BasicForwardIterator =
std::forward_iterator<T> &&                   // a basic forward iterator is a forward iterator
requires( const T i, const T j )
{// { expression } -> type_constraint;
    { i.begin() } -> std::same_as<T>;         // basic forward iterator provides begin()
    { i.end() } -> std::same_as<T>;           // basic forward iterator provides end()
};

static_assert(BasicForwardIterator<FibonacciSequenceIterator>);
```

The `FibonacciSequenceIterator` class satisfies `BasicForwardIterator`, a user-defined concept that extends the concept `std::forward_iterator` with the additional constraint that every type `T` used as a basic forward iterator must implement a suitable `begin()` and `end()`. This constraint stems from a requirement in the C++ standard that iterators used in a range loop must define a `begin()` and `end()` to mark the iterator range.

The definition of the class `FibonacciSequenceIterator` follows the approach we used in Tutorial 5. However, there are a few differences though. First, whenever the iterator advances using the increment operator, the underlying Fibonacci sequence must also advance. Second, the `begin()` method has to reset the underlying Fibonacci sequence to the start. Third, the `end()` method limits the position to the first non-representable Fibonacci number. The underlying Fibonacci sequence object remains unaffected by this boundary.

The file `Main.cpp` contains a test function to check your implementation. Uncomment **`#define`** `P2` and compile your solution. Your program should produce the following output:

```
Fibonacci sequence for 5 numbers:
 1: 1
 2: 1
 3: 2
 4: 3
 5: 5

Fibonacci sequence for unsigned long long:
 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
…
81: 37889062373143906
82: 61305790721611591
83: 99194853094755497
84: 160500643816367088
85: 259695496911122585
86: 420196140727489673
87: 679891637638612258
88: 1100087778366101931
89: 1779979416004714189
90: 2880067194370816120
91: 4660046610375530309
92: 7540113804746346429
93: 12200160415121876738
The Fibonacci sequence was generated successfully.
1 test(s) run.
```

**Submission deadline: Sunday, April 13, 2025, 23:59.**

**Submission procedure:**

Follow the instructions on Canvas. Submit electronically the PDF of the printed source files `FibonacciSequence.cpp` and `FibonacciSequenceIterator.cpp`. Upload the source files `FibonacciSequence.cpp` and `FibonacciSequenceIterator.cpp` to Canvas.

The sources will be assessed and compiled in the presence of the solution artifacts provided on Canvas.