Swinburne University of Technology

Faculty of Science, Engineering and Technology

LABORATORY COVER SHEET

Subject Code:	COS30008
Subject Title:	Data Structures and Patterns
Lab number and title:	12, Algorithm Analysis
Lecturer:	Dr. Markus Lumpe

Time cools, time clarifies; no mood can be maintained quite unaltered through the course of hours.

Mark Twain

Lab 12: Algorithm Analysis

In problem set 2, we used inheritance and method overriding in order to create a hierarchy of array sorters. In particular, we devised a solution to implement *Bubble Sort* and *Cocktail Shaker Sort*, a is bidirectional *Bubble Sort*.

In particular, we used two abstractions, `IntVector` and `SortableIntVector`, to model the sorting process and the underlying data collection. We are going to use both abstraction here again, but make one addition. We add an output operator to `IntVector`:

```
#pragma once

// include for size_t (unsigned integral type)
#include <cstddef>

#include <ostream>

class IntVector
{
private:
  int* fElements;
  size_t fNumberOfElements;

public:

  // Constructor: copy argument array
  IntVector( const int aArrayOfIntegers[], size_t aNumberOfElements );

  // Destructor: release memory
  // Destructor is virtual to allow inheritance
  virtual ~IntVector();

  // size getter
  size_t size() const;

  // element getter
  const int get( size_t aIndex ) const;

  // swap two elements within the vector
  void swap( size_t aSourceIndex, size_t aTargetIndex );

  // indexer
  const int operator[]( size_t aIndex ) const;

  // Lab 12 addition
  friend std::ostream& operator<<( std::ostream& aOStream,
                                   const IntVector& aObject );
};
```

Problem 1

Implement the output operator for `IntVector`. You can add it to `Main.cpp` or to `IntVector.cpp` as long as the C++ linker can find it.

You can use `#define P1` in `Main.cpp` to enable the corresponding test driver, which should produce the following output:

```
Test output operator:
Vector: 34 65 890 86 16 218 20 49 2 29
```

Problem 2

In this tutorial, we wish to compare two sorting algorithms with respect to their running time complexity. We choose *Bubble Sort* and *Sorting by Fisher & Yates* (better known as *BogoSort*). With respect to their running time, the former is $O(n^2)$, whereas the latter is O(n*n!). So, what does this mean in practice?

We start with the specification of `SortableIntVector` that we used in Problem Set 2, but make it an abstract class by declaring `sort()` it pure virtual. In addition, we provide an in-place definition for the constructor.

```
#pragma once

#include "IntVector.h"

#include <functional>

using Comparable = std::function<bool(int, int)>;

class SortableIntVector : public IntVector
{
public:

  SortableIntVector( const int aArrayOfIntegers[], size_t aNumberOfElements ) :
        IntVector(aArrayOfIntegers, aNumberOfElements)
  {}

  virtual void sort( Comparable aOrderFunction ) = 0;
};
```

In order to use this abstraction class, which defines a sorting interface, we need to define suitable subclasses. We start with *Bubble Sort*, that is, we define class `BubbleSort` as shown below:

```
#pragma once

#include " SortableIntVector.h"

class BubbleSort : public SortableIntVector
{
public:

  BubbleSort( const int aArrayOfIntegers[], size_t aNumberOfElements );

  void sort( Comparable aOrderFunction ) override;
};
```

In addition to sorting, we wish to monitor the progress and add an output statement at the end of the outer loop. This will allow us to observe how Bubble Sort rearranges elements.

You can use `#define P2` in `Main.cpp` to enable the corresponding test driver, which should produce the following output (sorting in increasing order and counting the number of exchanges):

```
Test bubble sort:
Before sorting: 34 65 890 86 16 218 20 49 2 29
      Stage i = 0: 2 34 65 890 86 16 218 20 49 29
      Stage i = 1: 2 16 34 65 890 86 20 218 29 49
      Stage i = 2: 2 16 20 34 65 890 86 29 218 49
      Stage i = 3: 2 16 20 29 34 65 890 86 49 218
      Stage i = 4: 2 16 20 29 34 49 65 890 86 218
      Stage i = 5: 2 16 20 29 34 49 65 86 890 218
      Stage i = 6: 2 16 20 29 34 49 65 86 218 890
      Stage i = 7: 2 16 20 29 34 49 65 86 218 890
      Stage i = 8: 2 16 20 29 34 49 65 86 218 890
      Stage i = 9: 2 16 20 29 34 49 65 86 218 890
After sorting: 2 16 20 29 34 49 65 86 218 890
Sorting required 29 exchanges.
```

Problem 3

Consider an array with 10 elements:

```
int lArray[] = { 34, 65, 890, 86, 16, 218, 20, 49, 2, 29 };
```

How can we sort this array, so that the elements are arranged in increasing order?

Sorting by Fisher&Yates:

This sorting method uses the Fisher&Yates shuffling. This technique exploits the fact that shuffling can produce a sorted array. For an array with 10 elements the odds of obtaining a sorted array through shuffling are 1 in 3,628,800 or 0.000000275573192. Even though these odds are not very good, they are still better than hitting the jackpot in the lottery. Moreover, the "*randomness of nature*" dictates that we may reach a sorted array much faster. But we also need to be prepared to wait longer than expected. Nevertheless, it is a fundamental property of randomness that a "possible" event, even a very improbable one, has to occur eventually. Fisher&Yates shuffling is defined as follows:

let n := N

while n > 1 **do**

  let k := *rand*() % n k is a random index between 0 and n-1

  n := n – 1

  A[n] :=: A[k]

A C++ solution for sorting by Fisher&Yates requires the proper initialization and use of a pseudo random number generator. We can use:

```
srand( (unsigned int)time( NULL ) );
```

to seed the C++ random number generator with the current time since January 1, 1970 in seconds. The function `srand` is defined in `cstdlib`, whereas `time` is defined in `ctime`. To obtain the next random number we call `rand()` that yields an integral number in the range between 0 and `RAND_MAX`. To limit this number to a specific range, we combine it with a modulus operation. For example, `rand() % n` yields a random number between 0 and n-1.

For the purpose for sorting, we define Fisher&Yates shuffling as a private member function. The Fisher&Yates sorter operates as follows:

do

  isSorted := isSorted(A)

  output "Stage: " + A

  if !isSorted

    shuffle(A)

while !isSorted

The class `FisherAndYatesSort` is defined as follows:

```
#pragma once

#include "SortableIntVector.h"

class FisherAndYatesSort : public SortableIntVector
{
private:
  void shuffle();

public:

  FisherAndYatesSort( const int aArrayOfIntegers[], size_t aNumberOfElements );

  void sort( Comparable aOrderFunction ) override;
};
```

You need to map the Fisher&Yates algorithm. It has an outer and an inner loop. This inner loop checks, if the array is sorted. The outer loop shuffles the array until it is sorted. We perform an output at the end of the inner loop, just before the array is shuffled again, to monitor the progress. In addition, we need to seed the random number generator in the constructor.

You can use `#define P3` in `Main.cpp` to enable the corresponding test driver, which should produce the following output (be patient, it may take a while; you may want to suppress stage output as it slows down the application further; the output sequence changes with time):

```
Test Fisher&Yates sort:
Before sorting: 34 65 890 86 16 218 20 49 2 29
      Stage: 34 65 890 86 16 218 20 49 2 29
      Stage: 65 34 49 16 2 86 890 218 29 20
      Stage: 65 34 890 29 49 16 218 2 20 86
      Stage: 890 2 20 86 49 34 16 65 29 218
      Stage: 86 34 890 49 2 65 20 29 218 16
      Stage: 2 29 218 890 86 65 20 49 34 16
      Stage: 29 65 86 20 34 49 218 890 16 2
      …
      Stage: 16 218 65 890 34 20 2 49 86 29
      Stage: 49 890 34 2 65 218 86 29 20 16
      Stage: 49 2 20 16 86 29 890 65 218 34
      Stage: 218 86 2 34 20 49 890 16 29 65
      Stage: 2 16 20 29 34 49 65 86 218 890
After sorting: 2 16 20 29 34 49 65 86 218 890
Sorting required 2747777 shuffle calls.
```

What is the result? Can you sort an array with 11, 12, or 13 elements?

Which sorting algorithm is better? Which one would you use in practice?

Please check with the tutor. You should complete this task as it may be a prerequisite for a later one.