Swinburne University of Technology

School of Science, Computing and Engineering Technologies

LABORATORY COVER SHEET

Subject Code:	COS30008
Subject Title:	Data Structures and Patterns
Lab number and title:	7, 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 7: Algorithm Analysis

In this tutorial, we are experimenting with two sorting algorithms and analyze their complexity in terms of the number of element exchanges these algorithms generate for a given input of size n. In particular, we use *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!). But what does this mean in practice? We already know that we should not use *Bubble Sort*. Recall, Donald Knuth's assessment: "*… bubble sort seems to have nothing to recommend it, except a catchy name…*". However, how does *Bubble Sort* compare to *Sorting by Fisher & Yates*? The *Sorting by Fisher & Yates* algorithm repeatedly shuffles a given collection until it is sorted. Basically, *Sorting by Fisher & Yates* seeks to find that permutation of a collection that is sorted and keeps shuffling the collection until it finds this permutation, possibly even checking a given permutation more than once. Suddenly, *Bubble Sort* looks appealing.

In order to run the experiments, we define three data types in this tutorial:

- a template class `BaseSorter` that defines the core infrastructure for a sorting algorithms,
- a template class `BubbleSorter` that is a subclass of `BaseSorter` implementing *Bubble Sort*, and
- a template class `FisherAndYatesSorter` that is also a subclass of `BaseSorter` implementing *Sorting by Fisher & Yates*.

The structure of these template classes follows that of template class `InsertionSorter` shown in class in week 1 (see *First Steps* page 18).

The test driver provided for this tutorial task makes use of conditional compilation via preprocessor directives. This allows you to focus only on the task you are working on.

The test driver (i.e., main.cpp) uses `P1`, `P2`, and `P3` as variables to enable/disable the test associated with a corresponding problem. To enable a test just uncomment the respective **`#define`** line. For example, to test problem 2 only, enable **`#define`** `P2`:

```
// #define P1
#define P2
// #define P3
```

In Visual Studio, the code blocks enclosed in **`#ifdef`** `PX` … **`#endif`** are grayed out, if the corresponding test is disabled. The preprocessor definition **`#ifdef`** `PX` … **`#endif`** enables conditional compilation. XCode does not use this color coding scheme.

Problem 1

We start with template class `BaseSorter`. Technically, this date type does not implement any sorting algorithm. It is a container type that maintains a copy of the array to be sorted and provides methods to access elements and swap two elements. In addition, it defines method `printStage()` that sends the content of the underlying array to the output. Finally, template class `BaseSorter` defines a template call operator to be invoked when the underlying array is to be sorted.

The template class BaseSorter is defined as follows:

```
template<typename T>
class BaseSorter
{
private:
  T* fCollection;
  size_t fSize;
  size_t fSwapCount;

public:

  BaseSorter( T* aCollection = nullptr, size_t aSize = 0 );
  virtual ~BaseSorter();

  size_t size() const noexcept;
  size_t getSwapCount() const noexcept;

  const T& operator[]( size_t aIndex ) const;
  void swap( size_t aLeftIndex, size_t aRightIndex );

  template<typename C = std::greater<T>>
  void operator()( bool aPrintStage = true, C aIsOutOfOrder = C{} ) noexcept
  {
    // intentionally empty
  }

  void printStage( size_t aIndent = 0 ) const noexcept;
};
```

Objects of template class `BaseSorter` maintain a private copy of the array to be sorted. The constructor allocates heap memory and copies the argument array. The destructor has to free the heap memory.

The getters `size()` and `getSwapCount()` return the size of the underlying array and the number of swap operations that have been performed on the underlying array, respectively.

Access to the elements is only via the public indexer or exchange elements via method `swap()`. The latter just takes two valid indices and exchanges the corresponding elements.

The call operator is a template function. The template argument is the ordering criterion. By default, we use `std::greater<T>` to trigger increasing order. We also include a flag to trigger whether the sorting process should print intermediate stages. By default, it is set to false, but we can use it, for example, to trace how *Bubble Sort* rearranges the elements in the underlying array in Problem 2.

Implement template class `BaseSorter`. Templates have no .cpp file. All code has to be defined in the header file `BaseSorter.h`.

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

```
Test BasicSorter:
[45,34,8,6,4,1,0,-2,-3,-100]
[45,34,8,6,4,1,0,-2,-3,-100]
Number of swaps: 0
Completed.
```

Problem 2

We now wish to implement with *Bubble Sort*, that is, we define template class `BubbleSorter` as a subclass of template class `BaseSorter` as shown below:

```
#pragma once

#include "BaseSorter.h"

template<typename T>
class BubbleSorter : public BaseSorter<T>
{
public:

  BubbleSorter( T* aCollection = nullptr, size_t aSize = 0 );

  template<typename C = std::greater<T>>
  void operator()( bool aPrintStage = true, C aIsOutOfOrder = C{} ) noexcept;
};
```

In template class `BubbleSorter`, the template call operator

```
  template<typename C = std::greater<T>>
  void operator()( bool aPrintStage = true, C aIsOutOfOrder = C{} ) noexcept;
```

implements *Bubble Sort*. We can use the implementation shown in class (see Tools for Algorithm Analysis, page 225). In addition to sorting, we wish to monitor the progress and add an output statement (i.e., `printStage()`) at the end of the outer loop. This will allow us to observe how *Bubble Sort* rearranges elements.

Templates combined with inheritance is a technique that results in some subtle issues in subclasses. In particular, we need to explicitly qualify the receiver object by the prefix **`this`**`->` to call base class methods. Otherwise, the compiler reports errors. In non-template classes a base class method named `f` can simply be called using the expression `f()`. However, in templates the expression `f()` does not depend on any template arguments. The compiler tries to resolve it without inspecting the base classes, which means that the method named `f` appears undefined. Qualifying the expression `f()` with **`this`**`->` makes the expression `f()` dependent on the template arguments. The compiler now defers the lookup of the method named `f` until the template is instantiated and only reports an error if expression `f()` is ill-formed under template specialization.

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 BubbleSorter:
[45,34,8,6,4,1,0,-2,-3,-100]
  [34,8,6,4,1,0,-2,-3,-100,45]
  [8,6,4,1,0,-2,-3,-100,34,45]
  [6,4,1,0,-2,-3,-100,8,34,45]
  [4,1,0,-2,-3,-100,6,8,34,45]
  [1,0,-2,-3,-100,4,6,8,34,45]
  [0,-2,-3,-100,1,4,6,8,34,45]
  [-2,-3,-100,0,1,4,6,8,34,45]
  [-3,-100,-2,0,1,4,6,8,34,45]
  [-100,-3,-2,0,1,4,6,8,34,45]
```

```
[-100,-3,-2,0,1,4,6,8,34,45]
Number of swaps: 45
Completed.
```

We use aIndent = 2 as argument to `printStage()` for the output occurring within *Bubble Sort*.

The 45 swaps are a consequence of sorting an array in reverse sorted order. This is the worst case.

Problem 3

Consider an array with 10 elements:

```
int lArray[] = { 45, 34, 8, 6, 4, 1, 0, -2, -3, -100};
```

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:

```
std::srand( static_cast<unsigned int>( std::time( NULL ) ) );
```

to seed the C++ random number generator with the current time since January 1, 1970 in seconds. The function `std::srand` is defined in `cstdlib`, whereas `std::time` is defined in `ctime`. To obtain the next random number we call `std::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, `std::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

We can define sorting by Fisher & Yates in a template class `FisherAndYatesSorter` that is a subclass of template class `BaseSorter` as shown below:

```
#pragma once

#include "BaseSorter.h"

#include <cstdlib>
#include <ctime>

template<typename T>
class FisherAndYatesSorter : public BaseSorter<T>
{
private:

  void shuffle() noexcept;

  template<typename C>
  bool isSorted( C aIsOutOfOrder ) const noexcept;

public:

  FisherAndYatesSorter( T* aCollection = nullptr, size_t aSize = 0 );

  template<typename C = std::greater<T>>
  void operator()( bool aPrintStage = true, C aIsOutOfOrder = C{} ) noexcept
};
```

In template class `FisherAndYatesSorter`, the template call operator

```
  template<typename C = std::greater<T>>
  void operator()( bool aPrintStage = true, C aIsOutOfOrder = C{} ) noexcept;
```

implements 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 FisherAndYatesSorter:
[45,34,8,6,4,1,0,-2,-3,-100]
  [-3,34,0,6,4,-2,-100,8,45,1]
  [0,8,34,1,-2,-100,45,6,-3,4]
  [34,1,0,4,8,6,-100,-3,-2,45]
  [0,-3,6,1,-100,8,4,34,-2,45]
  [8,1,6,-100,4,-3,-2,34,45,0]
  …
  [34,6,-100,1,8,-3,-2,4,0,45]
  [-100,34,-2,6,45,1,8,4,-3,0]
  [6,1,4,-100,-3,34,0,45,8,-2]
  [8,34,-2,4,1,-3,-100,45,6,0]
  [8,34,1,4,-2,6,0,45,-3,-100]
[-100,-3,-2,0,1,4,6,8,34,45]
```

```
Number of swaps: 18046340
Completed.
```

We use aIndent = 2 as argument to `printStage()` for the output occurring within *Bubble Sort*.

There are 18,046,340 swaps. 10*10! Is 36,288,000. The algorithm has found the sorted array, but inspected 1,804,634 permutations. But this is only one possible run.

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?