3 Assignment 1 [Assignment ID: cpp_basics]

3.1 Preamble (Please Read Carefully)

Before starting work on this assignment, it is **critically important** that you **carefully** read Section 1 (titled "General Information") which starts on page 1 of this document.

3.2 Topics Covered

This assignment covers material primarily related to the following: classes, templates.

3.3 Problems — Part A

- 8.8 a b [complexity of finding element in tree]
- 8.9 a b [complexity of summing lower triangular part of matrix]
- 8.10 a b c [complexity of array reversal]
- 8.12 [improvement possible by optimization]
- 8.13 a b c [complexity of computing Hamming weight]

3.4 Problems — Part B

1. *Linear congruential generator (LCG) class (*`linear_congruential_generator`*).* A linear congruential generator (LCG) is a type of pseudorandom number generator. A LCG is defined by a recurrence relation. In particular, the generated integer sequence x_n for $n \in \{1, 2, \ldots\}$ is given by

$$x_n = (ax_{n-1} + c) \mod m, \tag{1}$$

   where a, c, and m are integers such that $m > 0$, $0 < a < m$, and $0 \leq c < m$. The quantity x_0 is known as the seed and corresponds to the initial state of the LCG. The quantities a, c, and m are referred to as the multiplier, increment, and modulus of the LCG, respectively. For more information on LCGs, refer to:

   https://en.wikipedia.org/wiki/Linear_congruential_generator

   In this exercise, a LCG class will be developed. The class is to be called `linear_congruential_generator`. The source code for this class and its associated non-member functions should be placed in the following files:

   (a) `include/ra/random.hpp`. The code that specifies the API for the `linear_congruential_generator` class.
   (b) `lib/random.cpp`. Any non-API code.

   The `linear_congruential_generator` class should provide the following public members:

   (a) `int_type`. the unsigned integer type used by the `linear_congruential_generator` class; this type must be at least 64 bits in size
   (b) a constructor that takes the following arguments, each of type `int_type`: 1) multiplier, 2) increment, 3) modulus, and 4) seed. The fourth parameter (i.e., seed) should default to the value returned by the `default_seed` (static) member function.
   (c) move constructor and move assignment operator (which may be compiler-provided defaults if appropriate)
   (d) copy constructor and copy assignment operator (which may be compiler-provided defaults if appropriate)

(e) destructor (which may be the compiler-provided default if appropriate)
(f) `multiplier`. This (non-static) member function returns the multiplier value (i.e., a) for the generator. The return type is `int_type`.
(g) `increment`. This (non-static) member function returns the increment value (i.e., c) for the generator. The return type is `int_type`.
(h) `modulus`. This (non-static) member function returns the modulus value (i.e., m) for the generator. The return type is `int_type`.
(i) `default_seed`. This static member function returns the default seed value (i.e., x_0) for all generator objects. The value returned should always be 1. The return type is `int_type`.
(j) `seed`. This (non-static) member function restarts the sequence generation process with a new seed value (i.e., x_0). This function takes a single `int_type` argument specifying the desired seed value. The function does not return any value.
(k) **`operator`**`()`. This function advances the generator to the next position in the generated sequence and then returns the value corresponding to this new position. This function takes no arguments and has a return type of `int_type`.
(l) `discard`. This (non-static) member function discards the next n numbers in the generated sequence. The function takes a single argument of type **`unsigned long long`**, specifying the value of n. The function does not return any value.
(m) `min`. This (non-static) member function returns the smallest value that can potentially be output by the generator. This value is equal to 1 if the increment parameter (i.e., c) of the generator is 0 and is equal to 0 otherwise. The return type is `int_type`.
(n) `max`. This (non-static) member function returns the largest value that can potentially be output by the generator. This value is one less than the modulus parameter (i.e., m) of the generator. The return type is `int_type`.
(o) equality operator (i.e., **`operator`**`==`) and inequality operator (i.e., **`operator`**`!=`). These operators test two `linear_congruential_generator` objects for equality and inequality. Two `linear_congruential_generator` objects are deemed to be equal if and only if they have the same multiplier, increment, modulus, and have the same current state (i.e., the same x_i value).

Note that the `linear_congruential_generator` class is not default constructible. The following non-member function should also be provided:

(a) a stream inserter (i.e., **`operator`**`<<`). A stream inserter should be provided to allow `linear_congruential_generator` objects to be written to a stream. This function should output the following items (in order) separated by a single space (with no leading or trailing characters, including newlines):
    i. multiplier,
    ii. increment,
    iii. modulus, and
    iv. current state of generator (i.e., the x_i value).

As always, code must be const correct.

The `linear_congruential_generator` class and its associated non-member functions must be placed in a namespace called `ra::random`.

The C++ standard library provides the class template `std::linear_congruential_engine` for generating pseudorandom number sequences based on a LCG. It also provides two specific instances of this template called `minstd_rand0` and `minstd_rand`. This functionality of the standard library is likely to be quite helpful in testing the code developed in this exercise.

The code used to test the `linear_congruential_generator` class should be placed in a file called `app/test_random.cpp`.

2. *Rational number class template (`rational`).* A rational number is a number of the form x/y, where x and y are integers and $y \neq 0$ (i.e., a rational number is a ratio of integers). Rational numbers are sometimes useful for representing real numbers in contexts where the roundoff error associated with floating-point arithmetic would be problematic.

   In this exercise, a class template for representing rational numbers will be developed. The class template is to be called `rational`.

   The source code for this class and its associated non-member functions should be placed in the file `include/ra/rational.hpp`.

   The `rational` class template has a single template parameter `T`, which specifies the type to be used to represent each of the numerator and denominator of the rational number. The `rational` class should provide the following interface (i.e., public members):

   (a) `int_type`. This type member is the integral type used to represent each of the numerator and denominator of the rational number. That is, this type is simply an alias for the template parameter `T`.
   (b) default constructor. This constructor creates a rational number with the value 0.
   (c) two parameter constructor. This constructor creates a rational number with a particular numerator and denominator. This constructor has two parameters of type `int_type`, which (in order) correspond to the numerator and denominator of the rational number to be created. The second parameter has a default value of 1. The arguments to this constructor are not subject to any constraints except that the denominator must be nonzero. If the rational number to be constructed is specified as n/d and $d = 0$, the constructor must behave as if the rational number n/d were formed by dividing the rational number $n/1$ by the rational number $d/1 = 0/1$, which would result in division by zero. (See below for how division by zero must be handled.)
   (d) move constructor and move assignment operator (which may be compiler-provided defaults if appropriate)
   (e) copy constructor and copy assignment operator (which may be compiler-provided defaults if appropriate)
   (f) destructor (which may be the compiler-provided default if appropriate)
   (g) `numerator`. This (non-static) member function returns the numerator of the rational number. The return type is `int_type`.
   (h) `denominator`. This (non-static) member function returns the denominator of the rational number. The return type is `int_type`.
   (i) compound assignment operators for addition, subtraction, multiplication, and division (i.e., **`operator`**`+=`, **`operator`**`-=`, **`operator`**`*=`, and **`operator`**`/=`). These operators must be overloaded to allow a rational number to be added to, subtracted from, multiplied by, or divided by another rational number. The return type should follow the usual convention for compound assignment operators.
   (j) `truncate`. This (non-static) member function returns the integer obtained by rounding the rational number towards zero (i.e., the fractional part of the number is discarded). The return type is `int_type`.
   (k) `is_integer`. This (non-static) member function is a predicate that tests if a rational number is an integer. If the rational number is an integer, **`true`** is returned; otherwise, **`false`** is returned. (The

return type is **bool**.)

(l) **operator**!. This operator tests if a rational number is zero. The value **true** is returned if the number is zero; and **false** is returned otherwise. The return type is **bool**.

(m) equality and inequality operators (i.e., **operator**== and **operator**!=). These operators provide the usual tests for equality and inequality. For the purposes of comparison, two rational numbers are deemed to be equal if they represent the same real value.

(n) **operator**<, **operator**>, **operator**<=, and **operator**>=. These relational operators test the less-than, greater-than, less-than-or-equal, and greater-than-or-equal relations for rational numbers.

(o) prefix and postfix **operator**++. These operators provide the usual prefix and postfix increment operators. The increment operation adds 1 to the rational number.

(p) prefix and postfix **operator**--. These operators provide the usual prefix and postfix increment operators. The decrement operation subtracts 1 from the rational number.

The following non-member functions should also be provided:

(a) unary **operator**+

(b) negation operator (i.e., unary **operator**-). This operator should return the negative of a rational number.

(c) binary **operator**+, binary **operator**-, binary **operator***, and binary **operator**/. These operators provide addition, subtraction, multiplication, and division operations for two `rational<T>` objects (with `T` being the same for both objects).

(d) stream inserter (i.e., **operator**<<). A stream inserter must be provided for allowing a rational number to be written to a output stream (i.e., `std::ostream`). The output of a rational number with numerator n and denominator d (where n and d are coprime and $d > 0$), must generate:

  i. a minus character (i.e., "-") if $n < 0$;
  ii. one or more (decimal) digit characters corresponding to the value of $|n|$;
  iii. a slash character (i.e., "/"); and
  iv. one or more (decimal) digit characters corresponding to the value of d.

  The output must not contain any whitespace characters (e.g., spaces, tabs, newlines), including leading and trailing characters.

(e) stream extractor (i.e., **operator**>>). A stream extractor must be provided for allowing a rational number to be read from an input stream (i.e., `std::istream`). This function must read a rational number in the same format used by the stream inserter. If the stream data is incorrectly formatted, the stream inserter should indicate this by forcing the stream into an error state. (The `setstate` member function of the `std::istream` class can be used to set an error flag for a stream.)

The `rational` class must always represent a rational number in a manner such that

(a) the numerator and denominator are coprime (i.e., no common factors);

(b) the denominator is nonnegative.

This said, however, a user of the `rational` class can, for example, validly ask to construct a rational number with a numerator and denominator of 6 and -3, respectively, in which case these values would need to be converted by the constructor to a numerator and denominator of -2 and 1, respectively.

If an attempt is made to divide by zero, the division operation should yield the largest representable rational value (i.e., a rational number with numerator `std::numeric_limits<int_type>::max()` and denominator `int_type(1)`). (Note that this is true regardless of whether the number being divided by zero is positive, negative, or zero.) Construction of a rational number with a denominator equal to zero should also be treated as division by zero. Under no circumstances should the code actually divide by

zero. Furthermore, if division by zero is attempted, the code should not terminate the program (e.g., by calling `std::abort`). (Although division by zero is arguably better handled by throwing an exception, this is avoided since some students may not be familiar with the exception feature of the language.)

All identifiers for the `rational` class and its supporting code (i.e., non-member functions that provide support for the `rational` class) must be placed in the namespace `ra::math`. Again, it is extremely important that the code be const correct.

The code used to test the `rational` class should be placed in a file called `app/test_rational.cpp`.

With regard to the `truncate` member function, the following is helpful to know. The C++ programming language requires that integer division must round towards zero. So, for `x` and `y` of some integral type, the expression `x / y` is guaranteed to yield a result that is consistent with rounding towards zero (i.e., truncation).

3. *Optional — Rational number class template (*`rational`*).*

   (a) The `rational` class, as specified in Exercise 2, allows a rational object to be constructed from floating-point types. That is, code like the following will compile without error:

   ```
   rational<int> x(1.5, 3.2)
   rational<int> y(1.5f);
   x = rational<int>(1.5, 2.25);
   ```

   Code like this is extremely likely to correspond to a bug, as the floating-point values are implicitly converted (by truncation) to an integer type (namely, `rational<int>::int_type`) and then passed to the constructor of the `rational` class. Consequently, it would be desirable to prevent code like this from compiling (in order to prevent bugs of this nature). Use SFINAE to prevent code like that shown above from compiling. That is, use SFINAE to remove the relevant constructor from the overload set in the case that the arguments are not integral types.

   (b) If an attempt is made to divide by zero, throw an exception derived from `std::runtime_error` instead of returning a proxy value for infinity.