Note: These exercices will contribute to your final grade. Send the solutions to me by e-mail, `mateusz.skomra@ens-lyon.fr`, by March 19. You can work in pairs. General remarks:

- If you work in a pair, remember to write the names of both people that worked on the project in the e-mail that you send me.
- Put the class declaration in a .h file, the class definitions in a .cpp file, and the main function in another .cpp file.
- The main .cpp file that you send me should contain tests that show that the classes that you implemented work correctly. Test every method and function.
- Write comments in your code to make it clear for the reader.
- Your functions should check for errors whenever possible and return error messages. (For the moment, you do not need to worry about errors in the constructors that do not have a return value.)
- Make sure that constant methods have the *const* modifier, and the arguments that should not be modified by a given method are preceded by the *const* modifier. Also, for each method and attribute you should decide if it is *private* or *public*.

Exercise 1 (Rational numbers)**.** The aim of this exercise is to implement a class *Rational* that will allow us to perform exact operations on rational numbers. Here is the list of things that should consider/include (do not hesitate to add more methods and functions to your class if you want):

1. A rational number is of the form $\frac{p}{q}$. What should be the attributes of your class? What should be their type if we want to represent exactly every rational number from the interval $[-1, 1]$ whose denominator belongs to $\{1, 2, \ldots, 10^8\}$?
2. Implement the constructors for the class. In particular, provide the constructor without arguments and the copy constructor. Also, overload the assignment operator $=$.
3. Overload the operators $+$, $-$, $*$, $/$ that will perform the addition, subtraction, multiplication, and division of rational numbers. Note that it may be useful to perform these operations when one of the arguments is a rational number but other is of integer type (e.g., *int*). How to achieve that?
4. We would like every rational number $\frac{p}{q}$ to be stored in the memory as a reduced fraction. In other words, we want that $q > 0$ and that the greatest common divisor of p and q is equal to 1. Write the constructors and methods of your class in such a way that this is true. To do so, you may either write your own function that finds the gcd of two integers, or use the one provided in the standard library (*std::gcd* defined in header *<numeric>*).
5. Provide a method that displays a rational number exactly (as a fraction) and a second one that displays a floating-point approximation of the number. Provide the operators « and » so that the class *Rational* works correctly with the streams.
6. Overload the comparison operators ($<$, $>=$ etc.) to perform the comparisons between rational numbers.
7. Provide methods that output the floor, the ceiling, and the fractional part of the given number. Make sure that your methods work correctly with negative numbers.[1]

[1] See `https://en.wikipedia.org/wiki/Floor_and_ceiling_functions` for some examples.

8. If $\frac{a}{b}$ and $\frac{c}{d}$ are nonnegative rational numbers, then their *mediant* is the number given by $\frac{a+b}{c+d}$. Write a function that takes two nonnegative rational numbers as arguments and returns their mediant.[2]

9. Write a methods that counts the number of objects of the class *Rational* that are currently created.

10. Write a method that checks if the square root of the given rational number is again a rational number. If yes, it should compute this square root. If not, it should output -1.
Hint: You may want to start by writing a function that checks if a given natural number is a square. You can do this, for instance, by computing the floating-point approximation of the square root using the std::sqrt() function defined in the <cmath> header, and then performing a correct rounding. Another method (that does not involve floating-point numbers) is to use binary search on the natural numbers.

11. Write a function that solves quadratic equations with rational coefficients. It should take as an input three rational numbers a, b, c and output the following information: how many real roots does the polynomial $ax^2 + bx + c$ have, what are the *exact* roots of this polynomial, and what is their approximate form. For instance, given $(1, -1, -1)$, your function should say that the polynomial $x^2 - x - 1$ has two real roots, $\frac{1-\sqrt{5}}{2}$ and $\frac{1+\sqrt{5}}{2}$, and their approximate form is -0.618 and 1.618 respectively. Your function should output both the real and the complex roots of the polynomials.

12. Put the class declaration in Rational.h file, the class definitions in Rational.cpp file, and the tests that you performed in main.cpp file.

13.* (Additional exercice for those who want more) Write a function that takes as an input a rational number $\frac{p}{q}$ from the interval $[0, 1]$ and outputs the path that leads to $\frac{p}{q}$ in the Stern–Brocot tree.[3] Can you use your function to find good rational approximations of $\{\pi\}$, $\{e\}$, $\{\sqrt{7}\}$, and the Euler–Mascheroni constant? (Here, $\{x\}$ denotes the fractional part of x.) We would like to find approximations with denominators bounded by 2000.

Exercise 2 (Dynamic list of rationals)**.** We now want to implement the class *RationalList* that is a dynamic list of rational numbers. *(Note: in general, it is a much better idea to use the data structures provided in the standard library that write our own implementations. We do this only as an exercise.)*

1. Your class should have three attributes: the current size of the list (how many numbers does it contain), the maximal capacity of the list (how much memory space is reserved for the list), and a pointer to an array of rational numbers of size equal to the maximal capacity of the list. This array will be created by the constructor using the operator *new*.

2. Write a constructor for the list that takes as an argument the maximal capacity of a list. Provide a copy constructor and a constructor without arguments. Overload the assignment operator $=$. Do we need to provide a destructor?

3. Write methods that display the current size of the list and the current maximal capacity.

4. Write a methods that resizes the list (i.e., increases or decreases its maximal capacity). Be careful about the use of operators *new* and *delete*.

5. Create a method *push* that appends a given rational number to the end of the list.
Hint: If your list is full, a trivial solution would be to resize it by increasing its maximum capacity by 1 *(from* m *to* $m+1$*). However, this may lead to many resize operations. A better solution may be to increase the maximal capacity twice (from* m *to* $2m$*). Since we want to avoid doing many resize operations, we allow the list to have capacity that is bigger that the number of its elements.*

[2] See `https://en.wikipedia.org/wiki/Mediant_(mathematics)` for more information
[3] The algorithm is given in Wikipedia, see `https://en.wikipedia.org/wiki/Stern-Brocot_tree`

6. Create a method *pop* that outputs the last element of the list and deletes it from the list.

7. Overload the equality operators $==$ and $!=$, so that they check if the lists contain the same numbers.

8. Overload the subscript operator [] in such a way that *RationalList[i]* outputs the ith number of the list. Check that your operator works correctly with objects of type *RationalList* and *const RationalList*. Can you modify the numbers stored in a *const RationalList* from outside the class?

9. Create a methods that displays the current contents of the list. Overload the operators « and ».

10. Create a method *unique* that removes the duplicates from the list (this method should output a different list and do not modify the list on which it is called).

11. Create a method *sort* that sorts the numbers in the given list (this method should modify the list on which it is called). You may use any comparison-based sorting algorithm (such as bubble sort or insertion sort), but implement the merge sort if you want a somewhat more challenging task.

12. Put the class declaration in RationalList.h file, the class definitions in RationalList.cpp file, and the tests that you performed in main.cpp file (it may be the same main.cpp as for the previous exercise).