Programming Project 03

This assignment is worth 20 points (2% of the course grade) and must be **completed and turned in before 11:59 on Monday, February 5th .**

Assignment Overview
This assignment will give you more experience on the use of loops and conditionals, and introduce the use of functions.

Background
There are all sorts of special numbers. Take a look sometime at http://mathworld.wolfram.com/topics/SpecialNumbers.html for a big list. We are going to look at two classes of special numbers: friendly and solitary numbers.

Friendly vs Solitary
A **friendly** number is a number that has at least one other integer number that shares the same *abundancy index*. The abundancy index of a number is the ratio of `divisor_sum(n)/n`. If there is a least one other integer number that shares that abundancy index, the number is friendly and the two numbers together are known as a **friendly pair**. If the number shares its abundancy index with no other number, then it is **solitary**. See https://en.wikipedia.org/wiki/Friendly_number .

It turns out it is not easy to determine if a number is friendly or solitary. In most cases you have to iterate through a lot of numbers and just search. However, there are some rules. If a number is prime, or, more generally, if divisor_sum(n) and n are *relatively prime*, that is the greatest common divisor of both is 1, then it is a solitary number.

Sum of Divisors
The idea behind the sum of divisors is pretty simple. You take an integer and add up all the divisors of a number, including 1 (which is a divisor of every number) and the number itself. For example, the divisor sum of 13 is 14: 1 + 13. The divisor sum of 12 is: 1+2+3+4+6+12 -> 28

GCD and Abundancy Index
The abundancy index of a number is a ratio, divisor_sum(n)/n . However, it is more convenient to maintain that ratio than to reduce it to a floating point number, as that conversion can be approximate. If we are to maintain the abundancy index as a ratio, we have to reduce it, as a fraction, to its lowest terms. We can do that using the greatest common divisor. If we find the greatest common divisor of the numerator and denominator in a ratio, divide by the gcd, we get the ratio in its lowest terms. We can then directly compare two ratios without approximation.

Project Description / Specification
Two things:

1. In this and in all future projects we will provide *exactly* our function specifications: the function name, its return type, its arguments and each argument's type. The functions will be tested individually in Mimir using these exact function specifications. If you do not follow the function specifications, these independent tests of your functions will fail. Do not change the function declarations!
2. What you test on Mimir is a file that contains only the functions. You do not turn in a main program. We can test the functions individually on Mimir. However, you should write your own

main program to test your functions separate from Mimir. It is more flexible and you can debug more easily.

Functions

function: `divisor_sum`. Return is a long. Takes a single `long` argument, the number to calculate the divisor sum of. The divisor sum of 12 is: 1+2+3+4+6+12 -> 28

function: `gcd` return is long. Takes two `long` arguments, returns the greatest common divisor of the two input values. Rather than explain it I think you should look at this https://en.wikipedia.org/wiki/Euclidean_algorithm#Implementations wikipedia page implementation which explains it pretty well, even gives pseudo code. For example, gcd(240,180) is 60.

function: `is_solitary`: return is `bool`. Argument is a single `long` argument. Check if the argument and its divisor sum have as their gcd 1. If so, it is solitary and we return true. If not, well we really don't know but for this function we return false. is_solitary(64) -> true, is_solitary(28) -> false

function: `friendly_check`: return is a string (see last item). Takes two numbers: the first an integer to check and the second the upper limit of the other integers we are willing to check against the first number to see if they are a friendly pair (have the same abundancy index). We do so as follows:

- find the abundancy index of the integer to check. Reduce that ratio to lowest terms using gcd
- for every number from 2 up to the limit:
    - find the abundancy index of the range number. Reduce that ratio to its lowest terms using gcd
    - if the abundancy index of the integer to check matches the abundancy index of any of the numbers we are examining with the loop (the numerators are equal, the denominators are equal), then the two numbers are a friendly pair and we return:
        - numerator of the abundancy index
        - denominator of the abundancy index
        - the number friendly with the first number
- if we do not find a friend in the range from 2 to the limit, then we return:
        - numerator of the abundancy index
        - denominator of the abundancy index
        - -1
- because we would like to return more than a single element (the numerator and demoninator of the abundancy index as well as the friendly number (or -1)), we will return a string. Because we haven't worked with strings yet I provided a function in the skeleton.cpp called `abIndex_friend` that takes the three as arguments (numerator, denominator, friend or -1) and returns a string. Your function should return that string directly, i.e.
    - `return abIndex(numer, denom, friend);`

Input and Output

We provide a skeleton file to start with that contains a predefined main (see below). You will fill in the functions in the provided space and this is what you will turn into Mimir. **Do not change the main program!** It will mess with the tests and you won't get the credit you deserve! The TAs will check for this during grading. All you do is provide the required functions.

Deliverables

`proj03/proj03.cpp` -- the skeleton.cpp code with your functions added in. (*remember to include your section, the date, project number and comments in this file*).

1) In mimir this is Project 03 - friendly and solitary numbers
2) Mimir will start with the skeleton code in the IDE

General Hints:

1. You can write as many functions are you like over and above the ones I have specified.
    a. Make sure you write the requested functions exactly as specified. They will be tested individually according to that specification.
    b. We will only test the functions we indicated above.
2. The functions returns alone may not be very informative. Don't feel restricted to meeting the test criteria right away. Feel free to place lots of output statements in your functions so you can see what is going on.
3. Develop the functions one at a time and run the appropriate test case to see that the function works!
    a. Don't write everything all at once, write one function at a time, test it and make sure it works.

Specific Hints

If we were more knowledgeable about algorithms we could discuss how to be efficient about these checks, but for now we would just like them to work. Here are some pretty specific suggestions, but feel free to work it out for yourself

`divisor_sum`:

- You are going to check for each number from 1 to n to see if it divides evenly into n (obviously 1 and n will). If it does then you add it to the sum of divisors
- You can be more efficient than that, but it takes a little thought. That would matter for a big number.

`gcd`:

- The linked page gives a pretty good description in pseudo-code of what you need to do.
- it is essentially swapping (with t as a temp variable, bad naming!) the values:
    - a ← b
    - b ← a mod b
- gcd comes in handy to reduce your fractions and to determine (as best we can) whether a number is solitary

`is_solitary`:

- gcd is everything here. If the gcd of the divisor_sum(n) and n is 1, then it is solitary
    - does this capture a prime value??
- we assume it is not solitary (return false) if it doesn't meet the above conditions.

`friendly_check`

- remember to return the result of abIndex_friend as the return value

Testing

When we get to separate compilation of files, you will no longer submit a main program. We will test your programs independently. However, the approach taken here, a case statement with a case for each function is a pretty handy way to do your own testing. Pay attention and you might make use of it yourself later on.