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, September 25th .**

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: *k-hyperperfect* numbers and *smooth* numbers

Prime Factors
The prime factors of any integer should be familiar. You find all the prime integers that divide exactly (without remainder) into an integer. We exclude 1 (which is a factor of every integer) and the number itself (which, again, is a prime factor of every particular number).

- for example, the prime factors of 30 are 2,3,5 since 2*3*5=30. Note that 6, 15 and 10 are also factors but they are **not prime** factors.
- take a look at https://en.wikipedia.org/wiki/Table_of_prime_factors for a table of the prime factors of many numbers.

B-smooth Numbers
A number is B-smooth https://en.wikipedia.org/wiki/Smooth_number if none of its prime factors are greater than the value B. For example, the list of 5-smooth numbers (numbers whose prime factors are 5 or less) are listed in https://oeis.org/A051037 . 30, as seen above, is a 5-smooth number. It would also be 6-smooth, 7-smooth, etc.

k-hyperperfect numbers
We saw counting the divisors of a number in project 2. Here we are going to sum all the divisors of a number. A number *n* is k hyperperfect for some *k* if the following equality holds:

$$n = 1 + k*(\text{sum\_of\_divisors}(n) - n - 1)$$

- For example, 28 is 1-hyperperfect, meaning that the sum of its divisors (1+2+4+7+14+28) minus the 28 itself equals 28. These were traditionally called *perfect* numbers.
- 21 is 2-hyperperfect. Applying the formula:
  - sum_of_divisors(21) = (1+3+7+21) = 32
  - The value in parentheses is then (32 – 21 – 1) = 10
  - k =2, so 2 * 10 = 20
  - 20 + 1 = 21.

https://en.wikipedia.org/wiki/Hyperperfect_number gives a list of k-hyperperfect numbers.

Project Description / Specification
Warning
First, a warning. 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!

Functions
function: `biggest_prime`: return is long. Argument is a single `long n`. Returns the largest prime factor of `n`.

function: `sum_of_divisors`: return is `long`. Argument is a single `long n`. Sums up all the divisors of the argument `n` and returns that sum.

function: `k_hyperperfect`: return is long. Arguments are:
- `long n`. The number we are checking
- `long k_max`. The maximum `k` value considered.

The algorithm searches for a `k` from `1` up to and including `k_max` to test if the input number `n` is k-hyperperfect for that `k`. The first `k` in the range `1 to k_max` is returned where the input `n` proves to be k-hyperperfect. If `n` is not k-hyperperfect for any `k` from `1 to k_max`, the function returns 0.

`k_hyperperfect` should utilize `sum_of_divisors` in its calculations.

function: `b_smooth`: return is `bool`. Argument are:
- `long n`. The number being checked
- `long b`. The prime factor being checked.

Returns `true` if the input `n` is `b` smooth, `false` otherwise.

`b_smooth` should utilize `biggest_prime` in its calculations.

Input and Output and `main`
We have the ability in Mimir to test your functions individually, but for this project let's also do the following for the main program (which you will also write) which you can see tests for when you turn in the program. We will eventually the functions as specified **independent** of your main, but for this project we will test the output of main as follows:
- takes in 3 numbers from input: `n, k_max, b`
- prints out the following 4 numbers, all space separated, on a single line
    - the `biggest_prime` of `n`
    - the `sum_of_divisors` of `n`
    - the `k_hyperperfect` of `n` using `k_max`
    - the `b_smooth` of `n` using `b`
- as we are working only with longs, we do not need either `setprecision` or `fixed`
- as there is a Boolean result for `b_smooth`, we will use `cout << boolalpha;` for Boolean output.
- we guarantee that no number will exceed the size of a `long`

Deliverables
`proj03.cpp` -- your source code solution including the provided main (*remember to include your section, the date, project number and comments in this file*).
1) In mimir this is Project 03 -smooth and k-hyperperfect
2) You are provided with a proj03/proj03.cpp file in the directory above. The .cpp file will start blank.

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.
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 and then modify the main later to meet the specific test requirements.
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

`biggest_prime`:

The simplest, but not very efficient, algorithm for listing all the prime factors of a number is called "prime factorization by division" method. You do something like the following (let's use 300 as an example):

- The start. prime_list: `empty` number:`300`
- start with a divisor of 2 (a prime). If the number divides evenly by 2, do so, making 2 a prime factor: prime_list:`2` number:`150`
- can you divide by 2 again? If so keep doing so until you no longer can. We can one more time: prime_list: `2,2` number: `75`
- OK, now try 3 (a prime number). Does it divide evenly into 75? It does, so do the work again prime_list: `2,2,3` number:`25`
- Any more 3's? Go fish (wait, wrong game). OK, no more 3's
- On to 4. 4 you say? Yes, for simplicity's sake (that is, how you write your loop) you could now check 4. However, two things:
    - 4 isn't prime but
    - the number we have remaining cannot possible be evenly divisible by 4, since it is no longer evenly divisible by 2 (we checked, remember).
    - So we check 4, doesn't work, on to 5
- Can we divide evenly by 5? We can prime_list: `2,2,3,5` number:`5`
- Can we do 5 again? Yes. prime_list: `2,2,3,5,5` number:`1`
- We hit 1, we are done and we have our list of prime factors.

You need to pull out the largest prime factor along the way and return it.

`sum_of_divisors`:

- 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.

`k_hyperperfect` and `b_smooth` are straightforward given the other two functions