CMPUT 274/5 - Tangible Computing **Fall 2019**

Weekly Exercise #7

Euler's Phi Function

In the lecture on RSA Key Generation, we discussed how there are "enough" 15-bit integers e with $\gcd(e, \phi(n)) = 1$. Recall, $\gcd(a, b)$ is the greatest common divisor of a and b.

But we offered no proof of this fact: getting good estimates is a challenging mathematical problem. In this exercise, you will at produce experiments to see evidence that backs up the claim there are many "good" e.

We defined $\phi(n)$ to be $(p-1) \cdot (q-1)$ if $n = p \cdot q$ (i.e. a number we see with RSA). But there is a general definition of $\phi(n)$ for any positive integer n.

For **any** integer $n \geq 1$, $\phi(n)$ is simply the number of integers k with $1 \leq k \leq n$ with $\gcd(k, n) = 1$. For example, $\phi(12) = 4$ because the only numbers k between 1 and 12 with $\gcd(k, 12) = 1$ are 1, 5, 7, and 11.

One can show the following for any integer $n \geq 1$:

$$\phi(n) = n \cdot \prod_{p|n} \frac{p-1}{p}$$

where the product is over all primes dividing n. For example, if n is itself a prime then this formula simplifies to $n-1$, which is also the number of k in the range $[1, n]$ with $\gcd(k, n) = 1$. One should ponder why this holds if n is the product of two distinct primes p, q as with RSA numbers.

Using the formula above, we compute

$$\phi(12) = 12 \cdot \frac{2-1}{2} \cdot \frac{3-1}{3} = 4$$

as $p = 2$ and $p = 3$ are the only primes dividing 12. So the formula agrees with the definition in this case.

Understanding why the formula is correct is not required for this exercise, but you will compute values of $\phi(n)$ based on the formula.

Your Task

1. Implement the function
   `unsigned int phi(unsigned int n)`
   that returns $\phi(n)$. Use exactly this name with exactly these arguments.

   Note, this weekly assignment will be done with desktop C++. So the numbers we give

can be as large as 32-bits. You can assume $n \geq 1$ (we will not call it with $n = 0$).

Since the numbers are big, running $\gcd(k, n)$ for each $1 \leq k \leq n$ is too slow. See a discussion below with tips on how compute this fast enough.

2. Print the fraction $\phi(n)/n$ rounded off to *exactly* 5 digits after the decimal. We discussed how to print a specific number of digits using `cout` in the lectures. You should cast values to the type `double` instead of `float` when you print the fraction as this offers greater accuracy for floating point numbers.

3. And then, if $n \geq 2^{15}$ you should also print the fraction $x/2^{14}$ rounded off to *exactly* 5 digits after the decimal where x is the number of k in the range $2^{14} \leq k < 2^{15}$ with $\gcd(n, k) = 1$ (like what we are interested in for the assignment).

   It is ok if you simply try all such k and compute $\gcd(n, k)$ using the fast gcd function discussed in the lectures. You may freely use the greatest common divisor code posted to eClass for this, just mention it in your `README` if you do. The point of this part is to see that a random 15-bit number e still has a very good chance of having $\gcd(e, n) = 1$ (in the assignment, n would itself be of the form $\phi(m)$ for some RSA modulus m).

   If $n < 2^{15}$ you do not print anything for this part.

See the examples below for how the output should be formatted. You should produce *exactly* this output. We will provide TestCenter files for you to check your output before you submit. Though, this will be graded like a normal weekly exercise: there will be a rubric and you **must** exercise good style.

Computing $\phi(n)$
As you can see from the fornula, it suffices to simply determine all primes dividing n. One can do this using ideas behind the primality testing algorithm.

To find all primes p dividing n, do the following:

- Find the **smallest** p between 2 and $\sqrt{n}$ that divides n. If one exists, it **must** be prime (ponder why). Record it.

  Let n' be the value obtained by repeatedly dividing p out of n until it is no longer divisible by n. If $n' = 1$ **stop**. Otherwise, repeat this step using n' instead of n.

  For example, it $n = 120$ then we find $p = 2$ and obtain $n' = 15$ by repeatedly dividing 2 from n until it is not divisible by 2. We record 2 and repeat this step with 15 instead.

- If no p between 2 and $\sqrt{n}$ was found, then n was itself a prime. Record it and **stop**.

Now that you have recorded all primes dividing n, you can use the formula above to compute $\phi(n)$. Be careful about overflow: divide by p before multiplying by $p - 1$ when applying the formula.

Notes
The running time of this procedure is $O(\sqrt{n} \log n)$ because each loop only goes up to $\sqrt{n}$,

and there can be at most $\log_2 n$ prime divisors of n (as each prime is ≥ 2)[1]. With a bit more scrutiny, you can get the running time down to $O(\sqrt{n})$ by not "restarting" the loop back at 2 after n' is computed.

You also do not need an array to record all prime divisors of n (you can construct $\phi(n)$ on the fly as you find prime divisors). It is up to you to decide if you want to implement it this way or not.

Finally, you can avoid using the `sqrt()` function from `<cmath>` by iterating until `p*p > n`. Again, this is up to you.

Compiling

- The Arduino will NOT be used for this exercise. This is purely a C++ terminal exercise, run and compiled in the same way as a morning problem.

  You may compile and run your solution on the VM but outside of the TestCenter using:

  ```
  g++ eulerphi.cpp -o eulerphi
  ./eulerphi
  ```

- To test your solution in the TestCenter, download the TestCenter files for this exercise on eClass and put your implementation of `eulerphi.cpp` solution in the `soln` directory. You can now test it like a morning problem.

Sample Input 1

```
12
```

Sample Output 1

```
n           = 12
phi(n)      = 4
phi(n)/n    = 0.33333
```

Sample Input 2

```
101
```

Sample Output 2

```
n           = 101
phi(n)      = 100
phi(n)/n    = 0.99010
```

[1] In fact, one can show there are $O(\log n / \log \log n)$ primes dividing n but that's a lot harder to prove

Sample Input 3

```
34672912
```

Sample Output 3

```
n           = 34672912
phi(n)      = 17281440
phi(n)/n    = 0.49841
15-bit test = 0.49841
```

Sample Input 4

```
123989172
```

Sample Output 4

```
n           = 123989172
phi(n)      = 41329720
phi(n)/n    = 0.33333
15-bit test = 0.33337
```

Important Notes: Read This!

1. No template will be provided for this exercise, but your code must have the `phi` function as described above.
2. Absolutely no global variables. Also, you cannot use use `break`.
3. You will be provided with TestCenter files. If your code does not pass those tests, you will receive a deduction. But, unlike a morning problem, the TA will add more tests when they evaluate your submission. They will still be valid inputs: a single integer in the range $[1, 2^{32} - 1]$.
4. Style is still important, this is not a morning problem.
5. The running time of your implementation of `phi` should be no worse than $O(\sqrt{n} \cdot \log n)$ (a simple implementation of the idea from the description would satisfy this), better is also acceptable.
6. The formatting must be **exactly** correct for full marks (presentation errors are not permitted).

Submission Guidelines:

Submit all of the following files in a .tar.gz called `eulerphi.tar.gz` or a .zip called `eulerphi.zip`:

- `eulerphi.cpp`, containing your implementation of the weekly exercise.
- Your `README`, following the Code Submission Guidelines for C++ projects including instructions for how to compile and run your solution.

Note that your files must be named **exactly** as specified above. **Do Not** include the Test-Center files.

Style

Your submission must use good style. Generally speaking, good style rules from Python that are applicable to C++ still apply: comments must be placed appropriately, variable names must make sense, you should exercise good use of functions, etc.

With C++, good style dictates you **must use proper indenting**. There is no excuse for not using proper indentation and failure to do so will result in noticeable deductions.