Euler's φ Function

Lab Assignment 2 - Public Key Cryptography, UBB-CS Year 3

Popa Alex Ovidiu, group 936

Problem Statement: Write an algorithm for computing the value of Euler's function for natural numbers. For a given value v and a given bound b, list all natural numbers less than b which have v as the value of Euler's function.

Proposed solution:

Compute the Euler function for a value n using Euler's product formula.

Before getting into Euler's product formula, it is worth mentioning that one may think about a naive solution out of instinct before solving the problem.
The basic solution would take all natural numbers d less than a natural number n and check if they are coprimes with n, i.e.

$$gcd(n, d) = 1.$$

The count of all those numbers would be returned, and its equality checked with the given value v.
This would happen for each number up to the given bound b.
Implementing this was skipped as it's quite trivial to conclude its immense running time as v and b grow larger.

Euler's Product Formula

The first implemented function is nothing more than the theorem presented in the second lecture, the formula of which is:

$$\varphi(n) = n\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right)\cdots\left(1 - \frac{1}{p_j}\right)$$

where

$$p_1, p_2, \cdots p_j$$

are the prime factors of n. Or, equivalently,

$$\varphi(n) = n \cdot \prod_{p|n}\left(1 - \frac{1}{p}\right). \tag{1}$$

To prove (1), we will use two the other theorems stated in the lecture, namely the formula for $\varphi(p^k)$ (for p prime), as well as the multiplicative property of φ:

$$\begin{aligned}
\varphi(n) &= \varphi\left(p_1^{k_1}\right)\varphi\left(p_2^{k_2}\right)\cdots\varphi\left(p_j^{k_j}\right) \\
&= p_1^{k_1}\left(1 - \frac{1}{p_1}\right)p_2^{k_2}\left(1 - \frac{1}{p_2}\right)\cdots p_j^{k_j}\left(1 - \frac{1}{p_j}\right) \\
&= p_1^{k_1}p_2^{k_2}\cdots p_j^{k_j}\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right)\cdots\left(1 - \frac{1}{p_j}\right) \\
&= n\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right)\cdots\left(1 - \frac{1}{p_j}\right). \\
\implies &(1)
\end{aligned}$$

Running example for $\varphi(10)$:

1. n = 10 result = 10
2. 2 is a prime factor, so n = n/i = 5, result = 10 * (1.0 - 1.0/2.0) = 10 * (1.0 - 0.5) = 10 * 0.5 = 5.0
3. 3 is not a prime factor. The for loop stops after 3 as 4*4=16 is not less than or equal to 5.
4. After the for loop, result = 5.0, n = 5
5. n > 1 => result = 5.0 * (1.0 - 1.0/5.0) = 5.0 * 0.8 = 4.0
6. $\lfloor x \rfloor = 4$ is returned

```
<<EulersFunctionFractions>>=
def phiFractions(n):
    result = n
    p = 2
    while p * p <= n:
        if n % p == 0:
            while n % p == 0:
                n //= p
            result = result * (1.0 - (1.0 / float(p)))
        p += 1

    if n > 1:
        result = result * (1.0 - (1.0 / float(n)))
    return int(result)
@
```

As one can notice, some explicit type calls are made because of python implementation details, and this leads to having floating point errors, just take this example:

```
>>> 2**739 + 1
289179029371721471687588745441753893207178640573601538527580357720339848228986 7 \
263903614895099115516898199414270242812495998255990659472330369590362610632812985 1 \
351976783013074663752422325284123891275361063265590945125498 89
>>> 2.0 ** 739 + 1
2.891790293717215e+222
@
```

This algorithm is better than the one presented above because it basically finds all numbers that have gcd(n, d) $\neq$ 1, and when such a number is found, it is taken out of the result the along with all of its multiples below n.

Running example for $\varphi(15)$:

1. n = 15 result = 15
2. 2 is not a prime factor.
3. 3 is a prime factor, so n = n/i = 5, result = 5 The for loop stops after 3 as 4*4=16 is not less than or equal to 5.
4. After the for loop, result = 5, n = 5
5. n > 1 => result = result - result/n = 4
6. 4 is returned.

```
<<EulersFunction>>=
def phi(n):
    result = n
    p = 2
```

```
    while p * p <= n:
        if n % p == 0:
            while n % p == 0:
                n //= p
            result -= result//p
        p += 1

    if n > 1:
        result -= result // n
    return result
@
```

The actual algorithm concerning the value v and bound b is trivial, in the sense that it iterates until the bound and compares the result of the euler function with the value and adds the current number to a list if the two are equal. The function below does just that, namely prints out the values of the euler function for each bound which match the corresponding value, and the chosen implementation is given by the 'function' parameter. Furthermore, it returns the running time taken by the algorithm, to be used in the main function.

```
<<RunnerFunction>>=
<<EulersFunctionFractions>>
<<EulersFunction>>
from timeit import default_timer as time
def runner(values, bounds, function):
    start = time()
    for k in range(len(bounds)):
        result = []
        for i in range(1, bounds[k]):
            if function(i) == values[k]:
                result.append(i)
        print("(" + str(values[k]) + "," + str(bounds[k]) + ")->" + str(result))
    end = time()
    return end - start

@
```

In the main function some values and bounds were set, and some running time comparisons between the two algorithms were made. Although they perform quite equally running time-wise, the first one loses when dealing with big numbers, as mentioned before.

```
<<*>>=

<<RunnerFunction>>

def main():
    values = [12, 24, 36, 48, 128, 256, 1000]
```

```
    bounds = [468, 666, 41423, 12312, 50000, 80000, 1000000]

    print("Results for euler's formula with fractions: ")

    rt1 = runner(values, bounds, phiFractions)
    print("Running time for euler's product function: " + str(rt1))

    print("\nResults for euler's formula with fractions, without floating point issues: ")

    rt2 = runner(values, bounds, phi)
    print("Running time for euler's product function, without floating point issues: "
+ str(rt2))

    print("\nThe difference between the two times is: " + str(abs(rt1 - rt2)))
main()

@
```