Number Theory Assignment #4

21011712 박준영

1) lucas_lehmer_test(p)

```
def lucas_lehmer_test ( p ):
    if p == 2:
        return 1

    else:
        mersenne = (2 ** p) - 1
    r = 4

    for i in range(3, p + 1):
        r = (r ** 2 - 2) % mersenne

    if r == 0:
        return 1
    else :
        return 0
```

< the source code >

```
>>> lucas_lehmer_test(3)
1
>>> lucas_lehmer_test(17)
1
>>> lucas_lehmer_test(31)
1
>>> lucas_lehmer_test(521)
1
>>> lucas_lehmer_test(9689)
1
>>> lucas_lehmer_test(9697)
0
```

< results of given examples >

Explanation)

The Lucas Lehmer test tests whether a Mersenne number is a prime number or a composite number. And a Mersenne number is defined as

$$M_p = 2^p - 1$$

First, before I define the actual part for the algorithm, I define the base case; when *p == 2*. When *p == 2*, the Mersenne number is 3, which is a prime number. It should return *1*, indicating that it is indeed a prime number.

Second, when *p != 2*, I first set the Mersenne number according to the definition. Then, I set *r = 4* as the base case of the sequence of integers ($r_1 = 4$). After setting the Mersenne number and the base case, we get into a *for loop* that runs *(p + 1) − 3* times; the amount of times needed to find out the (*p − 1*)th integer in the sequence.

The *for loop* calculates and updates the integer r according to the definition

$$r_k \equiv r_{k-1}^2 - 2$$

until it reaches the *(p − 1)*th r (which is r_{p-1}). Because M_p is prime if and only if $r_{p-1} \equiv 0 \mod M_p$, if *r == 0*, then return the value that indicates it's a prime; *1*. If *r != 0*, then return *0*, showing that the Mersenne number is not a prime number.

The picture below shows the results of the given examples, and it seems the code is working properly.

---

2) find_mersenne_primes(max)

```
def is_prime(n):
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            return 0

    return 1

def generate_all_primes(n):
    a = []
    for i in range(2, n + 1):
        if is_prime(i):
            a.append(i)

    return a


def find_mersenne_primes(p) :
    prime = generate_all_primes(p)

    flag = -1
    k = 2
    while((2 ** k - 1) <= p ):
        num = (2 ** k) - 1

        for i in range(0, len(prime)):
            if num == prime[i]:
                flag = 1
                break

        if (lucas_lehmer_test(num) == 1 and flag == 1) :
            print(num, end = " " )

        k += 1
```

< the source code >

```
>>> find_mersenne_primes(5000)
3 7 31 127
```

< results of given examples >

Explanation)

First, I've taken the functions *is_prime(n)* and *generate_all_primes(n)* from the previous homework in order to use them in the new *find_mersenne_primes(p)* function. In the first line of the new function, I've made an array *prime* which holds all the primes generated by the *generate_all_primes(n).*

Then I've set the integer *k* to the value of *2*, which makes the initial Mersenne number automatically to *3*. While the Mersenne number value is smaller than the input value of *p*, the loop keeps on going to find all the Mersenne numbers. The variable *num* holds the value of $2^k - 1$, which is the Mersenne number value, and is used for later on comparisons.

In the *for loop* inside the while loop, it checks whether if the variable *num* matches with a value in the array *prime[].* It does this process until it reaches an end; when it has reached the end of the array. If it has found a match, or in other words knows it is indeed a prime number, it sets the variable *flag* to *1* and ends the loop. After the loop, it checks with the *lucas_lehmer_test* function to see if it has passed the test. If the integer in question has passed both tests, it prints the integer. After every process *k* is added by *1*, going back to the start of the *while loop.*

The picture below shows the results of the given examples, and it seems the code is working properly.
