Hensel (crypto 400)

###ENG PL

In the task we get parameters suggesting RSA:

```
n = 1581688906457476363395126526567273673701408932950303338239208333630259409060558913573169944824614765761181142076
e = 65537
c = 1408236251808595951375934941789684973143002666168694684085967418231651986982040655792497275368906494452408017292
```

But factorization of `n` reveals that it's a prime square.

It can actually still be solved via RSA, if we remember that in this case `phi = p*(p-1)` and simply run:

```
p = gmpy2.isqrt(n)
print(long_to_bytes(pow(c, gmpy2.invert(e, p*(p - 1)), p)))
```

But there is also a more generic approach useful when dealing with prime powers.

What we have is `m^e mod p^2` and we want to recover `m`. Recovering `m` for prime modulus is simple, since totien `phi = p-1`. But it's also possible for modulus which is a prime power, using Hensel Lifing lemma.

If we know solution to `f(x) = 0 mod p` we can use an iterative algorithm to compute solutions `mod p^k`.

```
import gmpy2
from src.crypto_commons.generic import long_to_bytes
from src.crypto_commons.rsa.rsa_commons import hensel_lifting


def main():
    n = 1581688906457476363395126526567273673701408932950303338239208333630259409060558913573169944824614765761181142
    e = 65537
    c = 1408236251808595951375934941789684973143002666168694684085967418231651986982040655792497275368906494452408017
    p = gmpy2.isqrt(n)
    k = 2
    base = pow(c, gmpy2.invert(e, p - 1), p)  # solution to pt^e mod p
    f = lambda x: pow(x, e, n) - c
    df = lambda x: e * x
    r = hensel_lifting(f, df, p, k, base)  # lift pt^e mod p to pt^e mod p^k
    for solution in r:
        print(long_to_bytes(solution))

    # print(long_to_bytes(pow(c, gmpy2.invert(e, p*(p - 1)), p)))

main()
```

Using code from our crypto-commons:

```
def hensel_lifting(f, df, p, k, base_solution):
    """
    Calculate solutions to f(x) = 0 mod p^k for prime p
    :param f: function
    :param df: derivative
    :param p: prime
    :param k: power
    :param base_solution: solution to return for p=1
    :return: possible solutions to f(x) = 0 mod p^k
    """
    previus_solution = [base_solution]
    for x in range(k-1):
        new_solution = []
        for i, n in enumerate(previus_solution):
            dfr = df(n)
            fr = f(n)
            if dfr % p != 0:
                t = (-(extended_gcd(dfr, p)[1]) * int(fr / p ** (k - 1))) % p
                new_solution.append(previus_solution[i] + t * p ** (k - 1))
            if dfr % p == 0:
                if fr % p ** k == 0:
                    for t in range(0, p):
                        new_solution.append(previus_solution[i] + t * p ** (k - 1))
        previus_solution = new_solution
    return previus_solution
```

And this gives us: `sponge_bob_square_roots` just as the simpler approach with RSA.

###PL version

W zadaniu dostajemy parametry sugerujace RSA:

```
n = 15816889064574763633951265265672736737014089329503033382392083336302594090605589135731699448246147657611811420768
e = 65537
c = 14082362518085959513759349417896849731430026661686946840859674182316519869820406557924972753689064944524080172929
```

Ale faktoryzacja `n` pokazuje że to potęga liczby pierwszej.

Nadal możemy rozwiązać to zadanie za pomocą RSA, jesli pamiętamy że w tym przypadku `phi = p*(p-1)` i uruchomimy:

```
p = gmpy2.isqrt(n)
print(long_to_bytes(pow(c, gmpy2.invert(e, p*(p - 1)), p)))
```

Ale jest też bardziej ogólne podejście do problemu potęg liczb pierwszych.

Mamy dane `m^e mod p^2` i chcemy poznać `m`. Odzyskanie `m` dla modulusa będącego liczbą pierwszą jest trywialne, bo totien `phi = p-1`. Odzyskanie `m` dla modulusa który jest potęgą liczby pierwszej jest także możliwe, za pomocą lematu Hensela.

Jeśli znamy rozwiązania `f(x) = 0 mod p` możemy użyć iteracyjnego algorytmu do policzenia rozwiązania `mod p^k`:

```
import gmpy2
from src.crypto_commons.generic import long_to_bytes
from src.crypto_commons.rsa.rsa_commons import hensel_lifting


def main():
    n = 158168890645747636339512652656727367370140893295030333823920833363025940906055891357316994482461476576118114 2
    e = 65537
    c = 140823625180859595137593494178968497314300266616869468408596741823165198698204065579249727536890649445240801 7
    p = gmpy2.isqrt(n)
    k = 2
    base = pow(c, gmpy2.invert(e, p - 1), p)  # solution to pt^e mod p
    f = lambda x: pow(x, e, n) - c
    df = lambda x: e * x
    r = hensel_lifting(f, df, p, k, base)  # lift pt^e mod p to pt^e mod p^k
    for solution in r:
        print(long_to_bytes(solution))
```

```
    # print(long_to_bytes(pow(c, gmpy2.invert(e, p*(p - 1)), p)))

main()
```

I używając kodu z naszego crypto-commons:

```
def hensel_lifting(f, df, p, k, base_solution):
    """
    Calculate solutions to f(x) = 0 mod p^k for prime p
    :param f: function
    :param df: derivative
    :param p: prime
    :param k: power
    :param base_solution: solution to return for p=1
    :return: possible solutions to f(x) = 0 mod p^k
    """
    previus_solution = [base_solution]
    for x in range(k-1):
        new_solution = []
        for i, n in enumerate(previus_solution):
            dfr = df(n)
            fr = f(n)
            if dfr % p != 0:
                t = (-(extended_gcd(dfr, p)[1]) * int(fr / p ** (k - 1))) % p
                new_solution.append(previus_solution[i] + t * p ** (k - 1))
            if dfr % p == 0:
                if fr % p ** k == 0:
                    for t in range(0, p):
                        new_solution.append(previus_solution[i] + t * p ** (k - 1))
        previus_solution = new_solution
    return previus_solution
```

Dostajemy: `sponge_bob_square_roots` tak samo jak dla podejścia z RSA.