Laboratory 4

Tiutiu Natan-Gabriel,937

Introduction

I will implement an algorithm for finding a non-trivial divisor using Pollard's p-1 Method. I also defined some functions for computing lcm for 2 numbers and also for a list of numbers. You can also find an implementaion for repeated squaring modular exponentiation algorithm (we will use rsme as a shortcut).

In order to generate the documentation written using markdown format , run **pandoc -t latex -o main.pdf main.nw** .

In order to generate the source code file, run **notangle main.nw>main.hs** .

Once you have the source code file,start the interpreter using command **ghci** ,then load the file using **:l main.hs** and finally run **pollard YOUR_NUMBER YOUR_BOUND** , for example **pollard 1725 7** in order to find a non-trivial divisor of 1725. You can also run **pollardWithIterations YOUR_NUMBER YOUR_BOUND ITERATIONS_NUMBER**, if you want to see whether the algorithm finds a non-trivial divisor in maximum ITERATIONS_NUMBER iterations.For example you can run **pollardWithIterations (2^31-1) 19 10** . I recommand you to use this function just for testing that in case of prime numbers our algorithm doesn't find any non-trivial divisor in any number of steps.

In the case you want to be asked whether you want to use the default bound or not, please take a look at functions **pollard_wrapper** and **pollard_with_iterations_wrapper**.

Repeated Squaring Modular Exponentiation & helper functions

Function generateBinary transforms a decimal number into a binary number. This is the mathematical model:

$$generate_binary_number(l, n) = \begin{cases} [0], if \ n = 0 and l = [] \\ l, if \ n = 0 \\ generate_binary_number([n\%2] \bigcup l, n/2), else \end{cases}$$

```
<<generate_binary_number>>=
def generate_binary_number(n):
    if n == 0:
        return [0]
    l = []
    while n > 0:
        l.append(n % 2)
        n = n // 2
    return l
@

<<rsme>>=
def rsme(b, k, n):
    a = 1
    if (k == 0):
        return a
    c = b
    l = generate_binary_number(k)
    # print(k,l)
    # print(l)
    if l[0] == 1:
        a = b
    for i in range(1, len(l)):
        c = c * c % n
        if l[i] == 1:
            a = c * a % n
    return a
@

<<rsme_wrapper>>=
def rsme_wrapper(b, k, n):
    if k >= 0:
        return rsme(b, k, n)
    k = (-1) * k
    res = rsme(b, k, n)
    return res
@

<<euclidean>>=
def euclidean(a, b):
    if (a == 0):
        return b
```

```
    return euclidean(b % a, a)
@

<<extended_euclidean>>=
def extended_euclidean(a, b):
    u2 = 1
    u1 = 0
    v2 = 0
    v1 = 1
    while b > 0:
        q = a // b
        r = a - q * b
        u = u2 - q * u1
        v = v2 - q * v1

        a = b
        b = r
        u2 = u1
        u1 = u
        v2 = v1
        v1 = v
    d = a
    u = u2
    v = v2
    return d, u, v
@

<<miller_rabin_test_wrapper>>=
def miller_rabin_test_wrapper(n, k):
    t = n - 1
    s = 0
    while t % 2 == 0:
        t = t // 2
        s += 1
    # print("aici",s,t)
    while k > 0:
        result = miller_rabin_test(n, s, t)
        k -= 1
        if not result:
            return False  # the result is composite
    return True  # the result may be prime
@

<<miller_rabin_test>>=
def miller_rabin_test(n, s, t):
    a = secrets.randbelow(n - 2) + 2
    # now let's compute the sequence
```

```
    sequence = []
    a_t = rsme(a, t, n)
    sequence.append(a_t)
    for i in range(1, s + 1):
        a_t = a_t * a_t % n
        sequence.append(a_t)

    if sequence[0] == 1:
        return True
    # print("sequence: ",sequence)
    for i in range(1, len(sequence)):
        if sequence[i] == 1:
            if sequence[i - 1] == n - 1:
                return True
            else:
                return False
    return False
@

<<trivial_primality_check>>=
def trivial_primality_check(number):
    for i in [2, 3, 5, 7, 11, 13, 17, 19]:
        if number % i == 0:
            return False
    return True
@

<<generate_large_prime_wrapper>>=
# generate_large_prime function will generate a random prime number between 2^order+1 and 2
# order represents the number of bits of the number
# secrets.randbelow(2**order - 1) generates a number from interval [0,2**order - 1) =>
# secrets.randbelow(2**order - 1) + 2**order + 1 generates a number from interval [2**order
# a number from interval [2**order + 1,2**(order+1) -1]
def generate_large_prime_wrapper(order):
    number = 2 ** order
    return generate_large_prime(number)
@

<<generate_large_prime>>=
def generate_large_prime(number):
    random_number = secrets.randbelow(number - 1) + number + 1
    if not trivial_primality_check(random_number):
        return generate_large_prime(number)
    if not miller_rabin_test_wrapper(random_number, 50):
        return generate_large_prime(number)
    # print("not trivial_primality_check(random_number): ",random_number,not trivial_primal
    return random_number
```

```
@

<<generate_key>>=
def generate_key(order):
    p = generate_large_prime_wrapper(order)
    print("p: ", p)
    q = generate_large_prime_wrapper(order)
    print("q: ", q)
    while p == q:
        q = generate_large_prime_wrapper(6)
        print("q: ", q)
    n = p * q
    phi_n = (p - 1) * (q - 1)
    # secrets.randbelow(phi_n-2) generates a random in range [0,phi_n-2),then
    # secrets.randbelow(phi_n-2) + 2 generates a random in range [2,phi_n) ,that is(1,phi_n,
    e = secrets.randbelow(phi_n - 2) + 2
    while euclidean(e, phi_n) != 1:
        e = secrets.randbelow(phi_n - 2) + 2
    _, d, _ = extended_euclidean(e, phi_n)
    d = (d + phi_n) % phi_n
    # (n,e) is public key and d is private
    print(n, e, d, p, q)
    return n, e, d
@

<<alphabet>>=

numbers = {0: " ", 1: "a", 2: "b", 3: "c", 4: "d", 5: "e", 6: "f", 7: "g", 8: "h", 9: "i", 1
           13: "m", 14: "n", 15: "o", 16: "p", 17: "q", 18: "r", 19: "s", 20: "t", 21: "u",
           25: "y", 26: "z"}
alphabet = {" ": 0, "a": 1, "b": 2, "c": 3, "d": 4, "e": 5, "f": 6, "g": 7, "h": 8, "i": 9,
            "m": 13, "n": 14, "o": 15, "p": 16, "q": 17, "r": 18, "s": 19, "t": 20, "u": 21,
            "y": 25, "z": 26}
@

<<compute_numerical_equivalent>>=
def compute_numerical_equivalent(text):
    numerical_equivalent = 0
    for i in text:
        numerical_equivalent = numerical_equivalent * 27 + alphabet[i]
    return numerical_equivalent
@

<<compute_literal_equivalent>>=
def compute_literal_equivalent(number, iterations):
    literal_equivalent = ""
    while iterations > 0:
        literal_equivalent = numbers[number % 27] + literal_equivalent
```

```
        number = number // 27
        iterations -= 1
    return literal_equivalent
@

<<encrypt>>=
# Plaintext message units are blocks of k letters, whereas
# ciphertext message units are blocks of l letters. The plaintext
# is completed with blanks, when necessary.
def encrypt(plaintext, n, e, k, l):
    if 27 ** k >= n or n >= 27 ** l:
        return "Please choose some appropriate values for k and l"

    ciphertext = ""
    while len(plaintext) % k != 0:
        plaintext += numbers[0]
    # Write the numerical equivalents
    for i in range(0, len(plaintext) // k):
        numerical_equivalent = compute_numerical_equivalent(
            plaintext[k * i:k * (i + 1)])  # alphabet[plaintext[2 * i]]*27+alphabet[plainte
        # print("numerical_equivalent: ",numerical_equivalent)
        encrypted_number = rsme_wrapper(numerical_equivalent, e, n)
        # print("encrypted_number: ", encrypted_number)
        literal_equivalent = compute_literal_equivalent(encrypted_number, l)
        ciphertext = ciphertext + literal_equivalent

    return ciphertext
@

<<decrypt>>=
# In the decrypt function we won't have any case in which we have to complete the ciphertex
# because the encrypt function always return a ciphertext of length multiple of "l"
def decrypt(ciphertext, n, d, k, l):
    # we MUST have 27^k < n < 27^l
    if 27 ** k >= n or n >= 27 ** l:
        return "Please choose some appropriate values for k and l"

    # print("ciphertext: ",ciphertext)
    plaintext = ""
    for i in range(0, len(ciphertext) // l):
        numerical_equivalent = compute_numerical_equivalent(ciphertext[l * i:l * (
                    i + 1)])  # alphabet[ciphertext[3 * i]] * (27**2) + alphabet[ciphertext
        # print("numerical_equivalent: ",numerical_equivalent)
        decrypted_number = rsme_wrapper(numerical_equivalent, d, n)
        # print("decrypted_number: ",decrypted_number)
        literal_equivalent = compute_literal_equivalent(decrypted_number, k)
        plaintext = plaintext + literal_equivalent
```

```
    for i in range(len(plaintext) - 1, -1, -1):
        if plaintext[i] == numbers[0]:
            plaintext = plaintext[:-1]
        else:
            break

    return plaintext
@

<<RSA>>=
def RSA(message, order=1024, k=-1, l=-1):
    n, e, d = generate_key(order)
    lower_bound = 0.21030991785714
    upper_bound = 0.21030991785716
    if k == -1 or l == -1:
        k = int(2 * order * lower_bound)
        l = int(2 * (order + 1) * upper_bound) + 1
    # print(n,e,d)
    print("Message to be encrypted: ", message)
    encrypted_message = encrypt(message, n, e, k, l)
    print("encrypted_message: ", encrypted_message)
    decrypted_message = decrypt(encrypted_message, n, d, k, l)
    print("decrypted_message: ", decrypted_message)
@

<<RSA_using_file>>=
def RSA_using_file(file_name, order=512, k=-1, l=-1):
    n, e, d = generate_key(order)
    lower_bound = 0.21030991785714
    upper_bound = 0.21030991785716
    if k == -1 or l == -1:
        k = int(2 * order * lower_bound)
        l = int(2 * (order + 1) * upper_bound) + 1
    # print(n,e,d)

    f = open(file_name, "r")
    message = f.read()
    f.close()

    print("Message to be encrypted: ", message)


    encrypted_message = encrypt(message, n, e, k, l)
    encrypted_file = file_name+".encrypted"
    # print(encrypted_file)
    f = open(encrypted_file, "w")
```

```
    f.write(encrypted_message)
    f.close()
    print("encrypted_message: ", encrypted_message)



    decrypted_message = decrypt(encrypted_message, n, d, k, l)
    decrypted_file = file_name + ".decrypted"
    f = open(decrypted_file, "w")
    f.write(decrypted_message)
    f.close()
    print("decrypted_message: ", decrypted_message)
@

<<main>>=
def main():
    message = "algebra"
    print("Message to be encrypted: ", message)
    encrypted_message = encrypt("algebra", 1643, 67, 2, 3)
    print("encrypted_message: ", encrypted_message)
    decrypted_message = decrypt(encrypted_message, 1643, 163, 2, 3)
    print("decrypted_message: ", decrypted_message)
@

<<tests>>=
def tests():
    assert miller_rabin_test_wrapper(101, 50)
    assert not miller_rabin_test_wrapper(123, 50)

    # testing miller_rabin_test_wrapper function (with 50 iterations)
    for i in [17, 19,31,61,89,107]:
        assert miller_rabin_test_wrapper(2**i - 1,50)
    for i in [21,29,49,80,99,123]:
        assert not miller_rabin_test_wrapper(2**i - 1, 50)

    # testing the extended_euclidean and euclidean functions
    for i in range(0, 20):
        a = random.randrange(10, 1000)
        b = random.randrange(10, 1000)
        l = extended_euclidean(a, b)
        assert a * l[1] + b * l[2] == euclidean(a, b)

    # testing the rsme_wrapper function, which computes a^b mod n using repeated squaring mo
    assert (rsme_wrapper(16, 10, 11) == pow(16, 10, 11)
    assert (rsme_wrapper(116, 107, 211) == pow(116, 107, 211)
    assert (rsme_wrapper(145, 129, 199) == pow(145, 129, 199)
    for i in range(0, 20):
```

```
        a = random.randrange(10, 1000)
        b = random.randrange(10, 1000)
        n = random.randrange(10, 1000)
        assert rsme_wrapper(a, b, n) == pow(a, b, n)

    # testing the encrypt and decrypt functions

    for i in range(0, 20):
        # 27^ k >= n or n >= 27^ l
        message_length = random.randrange(10, 1000)
        characters = list(alphabet.keys())
        message = ''.join([random.choice(characters) for n in range(message_length)])
        # remove trailing spaces
        for i in range(len(message) - 1, -1, -1):
            if message[i] == numbers[0]:
                message = message[:-1]
            else:
                break
        order = 128
        n, e, d = generate_key(order)
        #lower bound is a little bit SMALLER than log 27 2
        # 27 ** k <= n and n <= 27 ** l
        # =>k * log 2 27 <= log 2 n and n <= 27 ** l and as we choose
        # n in interval 2^(order)+1,2^(order+1)-1 => k * log 2 27 <= log 2 n <= order   =>
        # it's safe to take k = (log 2 27)^(-1) * log 2 n = log 27 2 * log 2 n
        lower_bound = 0.21030991785714
        # lower bound is a little bit LARGER than log 27 2
        upper_bound = 0.21030991785716

        k = random.randrange(1, int(2 * order * lower_bound)+1)
        # aux is lower bound for l
        aux = int(2 * (order + 1) * upper_bound) + 1
        l = random.randrange(aux, aux*4)

        # print("Message to be encrypted: ", message)
        encrypted_message = encrypt(message, n, e, k, l)
        # print("encrypted_message: ", encrypted_message)
        decrypted_message = decrypt(encrypted_message, n, d, k, l)
        print("message: ",message)
        print("decrypted_message: ", decrypted_message)
        assert message == decrypted_message
@

<<*>>=
import secrets
import sys
```

```
import random

print("sys.getrecursionlimit(): ",sys.getrecursionlimit())
sys.setrecursionlimit(2000)


<<generate_binary_number>>
<<rsme>>
<<rsme_wrapper>>

<<euclidean>>
<<extended_euclidean>>


<<miller_rabin_test_wrapper>>
<<miller_rabin_test>>
<<trivial_primality_check>>
<<generate_large_prime_wrapper>>
<<generate_large_prime>>
<<generate_key>>

<<alphabet>>
<<compute_numerical_equivalent>>
<<compute_literal_equivalent>>
<<encrypt>>
<<decrypt>>
<<RSA>>
<<RSA_using_file>>

<<main>>
<<tests>>

# RSA("the best time to visit cancun is from december to april during the peak season")

# RSA_using_file("message.txt")

tests()

# main()

@
```