Information Secutiry - Week 3

Arvid Lindstrom - s2740761, Nil Stolt Anso - s2705338,
Razvan Andrei Poinaru - s2914751

October 2, 2017

Abstract

Exercises submitted: 11, 12, 14, 15 and 17

Exercise 11

Program Output

```
formation Sec/infosecRepo/infosec2017/week3/ex11$ python repeatedSquares.py
Base: 43210 # <-- user input
Exponent: 23456
Modulus: 99987
Recursive solution = 82900
Iterative solution = 82900
```

Source of Program

```python
from timeit import default_timer as timer

# Function used for reference since pythons
# pow() is using some optimizations
def naiveEponentiation(base, exponent):
    for i in range(exponent-1):
        base *= exponent
    return base

def repeatedSquaresIter(base, exponent, modulus):
    # Reference: Stamp's Book, page 99

    #1. Find exponent in binary
    binStr = bin(exponent)[2::] #skip the '0b'-part

    #2. Initialize values
    exponent = 0
    output = 0

    #3. Loop through binStr from MSB to LSB
    for bit in binStr:
        output = pow(pow(base, exponent), 2)
        #4. Calculate new exponent
        exponent = (exponent * 2)

        if(bit == '1'):
            output *= base
            exponent += 1
        output = (output % modulus)

    return output
```

```
def repeatedSquaresRec(base, exponent, modulus):
    # base case
    if(exponent == 1):
        return base % modulus
    # recursive step
    else:
        returned = repeatedSquaresRec(base, exponent / 2, modulus)
        returned = returned * returned % modulus
        if(exponent % 2 != 0):
            returned = returned * base % modulus
    return returned % modulus

base = int(raw_input("Base: "))
exp = int(raw_input("Exponent: "))
mod = int(raw_input("Modulus: "))

print "Recursive solution = " + \
    str(repeatedSquaresRec(base, exp, mod))
print "Iterative solution = " + \
    str(repeatedSquaresIter(base, exp, mod))
```

Exercise 12

Answer to questions

```
arvid@arvid-Aspire-V3-371:~/Desktop/Arvids Stuff/University/Year Four/
Block 1/Information Sec/infosecRepo/infosec2017/week3/ex12$ python ex12.py
Prime: 7919
[2, 37, 107]
7917  # <-- modified to only show largest generator
arvid@arvid-Aspire-V3-371:~/Desktop/Arvids Stuff/University/Year Four/
Block 1/Information Sec/infosecRepo/infosec2017/week3/ex12$ python ex12.py
Prime: 23
[2, 11]
[5, 7, 10, 11, 14, 15, 17, 19, 20, 21]
```

Generators of 23 are $[5, 7, 10, 11, 14, 15, 17, 19, 20, 21]$.
Larget generator of 7919 is 7917.

Source of Program

```
def checkIfPresent(c, factors):
#Determine if number is already present
    if len(factors) != 0:
        for i in range(len(factors)):
            if c == factors[i]:
                return factors
    factors.append(c)
    return factors

def prime_factors(n):
#Determine prime factors
    number = n
    i = 2
    factors = []
    while i*i <=n:
        if n%i:
            i+=1
        else:
            n //= i
```

```
            factors = checkIfPresent(i,factors)
    if n > 1:
        if n != number:
            factors = checkIfPresent(n,factors)
    return factors

def repeatedSquaresRec(base, exponent, modulus):
    # base case
    if(exponent == 1):
        return base % modulus
    # recursive step
    else:
        returned = repeatedSquaresRec(base, exponent / 2, modulus)
        returned = returned * returned % modulus
        if(exponent % 2 != 0):
            returned = returned * base % modulus
    return returned % modulus

def generators(pf, p):
    #Determine generators
    gener = []
    for g in range(2,p):
        flag = 0
        for x in pf:
            if repeatedSquaresRec(g, (p-1)/x, p) == 1:
                flag = 1
                break
        if flag == 0:
            gener.append(g)
    return gener

def main():
    prime = input("Prime: ")
    factors = []
    factors = prime_factors(prime-1)
    print factors
    gener = generators(factors,prime)
    print gener[-1] #<-- remove [-1] to show all generators

if __name__ == "__main__":
    main()
```

Exercise 14

Answer to question regarding 57 * P1

The number 57 can be represented in binary as: 111001. Let the least most significant bit 2^0 : 1 represent the point P1. Every bit to the left (towards the most significant) will represent $P_i = P_{i-1} + P_{i-1}$. Thus we perform one addition per bit needed to represent the multiplier in binary. To yield the desired result we perform addition on all points P_i correlating with a high-bit. The amount of bits required to represent 57 is 6. The first value is simply P1 and thus requires no addition-operation. Therefore we have 5 additions for bits $2^1, 2^2, ...2^5$ and finally 3 additions since there are 4 high bits in the binary representation of 57 (and addition is a binary operation taking two arguments). This yields 5 + 3 = 8 steps.

Program output

```
Alice sends: (13, 16)
Bob sends: (7, 8)
--- Addition steps needed for 57 * P1: 8
Shared secret: (35, 20)
```

```
Alice and Bob will use the X-coord of (35, 20) : 35
Steps needed for: 209 * P(2, 7), a = 11, N = 167, -- 10
```

Source of Program

```python
from modinv import modinv

# Global used to track amount of addition-operations
numAdditions = 0

# Stamp's algorithm for adding two points
# P3(x3, y3) = P1(x1, y1) + P(x2, y2)
# -- Adding two points on an elliptic curve:

# Arguments: Two tuples p1 and p2 to be added
# Returns: One tuple p3 containing
# the new (x, y)-coordinates
def isEqual(p1, p2):
    if(p1[0] == p2[0] and p1[1] == p2[1]):
        return True
    return False

# Assuming a curve y^2 = x^3 + ax + b (% mod)
# p1 is the Pi with smallest Xi, p1[0] < p2[0]
def ellipticCurveAddition(p1, p2, a, mod):
    #0. Check the smallest Xi
    if (p1[0] > p2[0]):
        p1, p2 = p2, p1 # tuple swap (of tuples)

    #1. Calculate 'm'
    m = 0
    if(isEqual(p1, p2)):
        m = (3 * pow(p1[0], 2) + a) * \
            modinv(2 * p1[1], mod) % mod
    else:
        m = (p2[1] - p1[1]) * \
            modinv((p2[0] - p1[0]), mod) % mod

    x3 = 0
    y3 = 0
    # x3 = (m^2 - x1 - x2) % mod
    x3 = (pow(m, 2) - p1[0] - p2[0]) % mod
    # y3 = (m(x1 - x3) - y1) % mod
    y3 = ( m * (p1[0] - x3) - p1[1]) % mod

    return (x3, y3)

# "Naive multiplier"
def ellipticCurveMult(multiplier, point, a, modulus):
    increment = point
    for i in range(multiplier - 1):
        point = ellipticCurveAddition(point, increment, a, modulus)
    return point

# Find high bit points
# Given a multiplier and a point, compute all
# additions and return a list of the ones corresponding
# to the position with a high-bit in the multiplier
def findHighBitPoints(point, a, modulus, multiplier):
    global numAdditions
```

```
    binM = bin(multiplier)[2:]
    sumList = [0] * len(binM)
    sumList[0] = point
    orderedOutput = []

    # Starting with the least significant bit
    binM = list(reversed(binM))

    for i in range(1, len(binM)):
        prev = sumList[i-1]
        sumList[i] = ellipticCurveAddition(prev, prev, a, modulus)
        numAdditions += 1 #<-- track the amount of additions

    for i in range(len(binM)):
        if(binM[i] == '1'):
            orderedOutput.append(sumList[i])

    return orderedOutput

# Given a list of all points which had a high bit
# and recursively add them together like in hidden
# slide 51 from week 3
def recECCAdd(a, mod, list):
    global numAdditions
    if(len(list) == 2):
        numAdditions += 1 #<-- track the amount of additions
        return ellipticCurveAddition(list[0], list[1], a, mod)
    else:
        point = list[0]
        numAdditions += 1 #<-- track the amount of additions
        return ellipticCurveAddition(\
            recECCAdd(a, mod, list[1:]), point, a, mod)

# Fast multiplication using the bits of the multiplier
# Returns the point as a tuple
def fastECCMultiplier(point, a, modulus, multiplier):
    return recECCAdd(a, modulus, \
        findHighBitPoints(point, a, modulus, multiplier))

#### Answers to questions ####

a = 10
b = -21
p = 41 # modulus (prime)
P1 = (3, 6)

Alice_m = 44
Bob_m = 57

#Shared secret = 44 * (57 * P1), 57 * (44 * P1)

AliceMsg = fastECCMultiplier(P1, a, p, Alice_m)
print "Alice sends: " + str(AliceMsg)
numAdditions = 0    #<-- reset the counter to track
                    # necessary steps for 57 * P1
BobMsg = fastECCMultiplier(P1, a, p, Bob_m)
print "Bob sends: " + str(BobMsg)
print "--- Addition steps needed for 57 * P1: " + str(numAdditions)
sharedSecret = fastECCMultiplier(BobMsg, a, p, Alice_m)
print "Shared secret: " + str(sharedSecret)
```

```
print "Alice and Bob will use the X-coord of " + \
    str(sharedSecret) + " : " + str(sharedSecret[0])

# We test our counter against the given amount of
# steps from hidden slide 51
numAdditions = 0
fastECCMultiplier((2,7), 11, 167, 209)
print "Steps needed for: 209 * P(2, 7), a = 11, N = 167, -- " \
    + str(numAdditions)
# ..and receive 10 steps as expected
```

Exercise 15

This exercise uses the same code as exercise 14 with a slight output modification shown in the Source-code subsection.

Values sent to TA

```
3
5
157
4
9
19
135
```

Values received from TA

```
(79, 12)
```

Source of Program

```
#### Answers to questions ####

# Public Key
a = 3
b = 5
mod = 157
MyPoint = (4, 9)

# Private key
m = 24

# MyPoint_m = (19, 135)
MyPoint_m = fastECCMultiplier(MyPoint, a, mod, m)

# Point sent from TA
TAPoint = (79, 12) # = n * (4, 9)

SharedSecret = fastECCMultiplier(TAPoint, a, mod, m)
print SharedSecret
```

Program output

```
arvid@arvid-Aspire-V3-371:~/Desktop/Arvids Stuff/University/Year Four/
Block 1/Information Sec/infosecRepo/infosec2017/week3/ex15$ python ex15.py
(16, 99)
```

The final shared point on the elliptic curve is:
(16, 99)

Exercise 17

Here we once again use the code from exercise 14 with a modification shown below.
The final answer to k(reversed) * P =
$(36738533453554501594987308459541 0L, 171995391554293041834290054849881L)$

Program output

```
arvid@arvid-Aspire-V3-371:~/Desktop/Arvids Stuff/University/Year Four/
Block 1/Information Sec/infosecRepo/infosec2017/week3/ex17$ python ex17.py

First we verify that our code works by using original ’k’
Result is :
(44646769697405861057630861884284L, 522968098895785888047540374779097L)
 and required 164 steps to compute.
Result is :
(367385334535545015949873084595410L, 171995391554293041834290054849881L)
 and required 165 steps to compute.
```

Source of Program

```
numAdditions = 0

a = 321094768129147601892514872825668
b = 430782315140218274262276694323197
p = 564538252084441556247016902735257

Point = (97339010987059066523156133908935, \
        149670372846169285760682371978898)

k1 = 281183840311601949668207954530684
k2 = 486035459702866949106113048381182

print "First we verify that our code works by using original ’k’"
result = fastECCMultiplier(Point, a, p, k1)
print "Result is :\n" + str(result) + "\n and required " + \
    str(numAdditions) + " steps to compute."
numAdditions = 0
result = fastECCMultiplier(Point, a, p, k2)
print "Result is :\n" + str(result) + "\n and required " + \
    str(numAdditions) + " steps to compute."
```