MTH 329 - Cryptography - Assignment 6

Purva Parmar

20181081

RSA

Book: *Introduction to Cryptography with Coding Theory*, 2nd edition, Trappe and Washington

Section 6.9

Q5

> Factor 8834884587090814646372459890377418962766907 by the p - 1
>
> method.

We need to write code for p - 1 factorization algorithm as given in the book.

```
In [1]: from math import gcd
        def pm1(n, B):
            a = 2
            b = a % n

            for j in range(2, B + 1):
                # Compute b^j mod n
                b = pow(b, j, n)

            d = gcd(b - 1, n)
            if d == 1:
                print("No factor found. Increase bound.")
            else:
                return d
```

```
In [2]: n = 8834884587090814646372459890377418962766907
        factor = pm1(n, B=100)
        factor

Out[2]: 364438989216827965440001
```

```
In [3]: # Print remainder to check it is an actual factor
        print(f"{n % factor = }")
```

```
        # Divide to find other factor and print it
        print(f"{n // factor = }")

Out[3]: n % factor = 0
        n // factor = 242424242424268686907
```

So, the two factors are 364438989216827965440001 and 242424242424268686907.

Q6

> Let n = 537069139875071. Suppose you know that
>
> $$85975324443166^2 \equiv 462436106261^2 \pmod{n}.$$
>
> Factor n.

There isn't much to it. Following the Quadratic Sieve method, we see that

$$85975324443166 \not\equiv \pm 462436106261 \pmod{n}$$

So, a factors are easily found.

```
In [4]: n = 537069139875071
        a = 85975324443166
        b = 462436106261

        p = gcd(a-b, n)
        print(p)

        q = n // p
        print(q)

Out[4]: 9876469
        54378659
```

The two factors are 9876469 and 54378659. Multiplying them indeed does give n.

Q7

> Let $n = 985739879 \cdot 1388749507$. Find x and y with $x^2 \equiv y^2 \pmod{n}$ but $x \not\equiv \pm y \pmod{n}$.

Let $n = pq$. Let us fix y to be a random number, say y = 81. Then, we need to solve:

$$x \equiv 81 \pmod{p} \quad \text{and} \quad x \equiv -81 \pmod{q}$$

Because then, we get $x^2 \equiv 81^2 \pmod{p}$ and $x^2 \equiv 81^2 \pmod{q}$, resulting in $x^2 \equiv 81^2 \pmod{pq}$.

The two equations can be solved using the Chinese Remainder Theorem.

Here, we use SymPy, a python library for symbolic mathematics. It contains a function to use the theorem.

```
In [5]: from sympy.ntheory.modular import crt
        help (crt)

Out[5]: Help on function crt in module sympy.ntheory.modular:

        crt(m, v, symmetric=False, check=True)
        Chinese Remainder Theorem.

        The moduli in m are assumed to be pairwise coprime.  The output
        is then an integer f, such that f = v_i mod m_i for each pair out
        of v and m. If ``symmetric`` is False a positive integer will be
        returned, else |f| will be less than or equal to the LCM of the
        moduli, and thus f may be negative.

        If the moduli are not co-prime the correct result will be returned
        if/when the test of the result is found to be incorrect. This result
        will be None if there is no solution.

        The keyword ``check`` can be set to False if it is known that the moduli
        are coprime.
```

Some help text has been left out here for readability.

```
In [6]: p = 985739879
        q = 1388749507
        n = p * q

        # The first number returned is the solution, hence the [0]
        x = crt([p, q], [81, -81])[0]
        x

Out[6]: 1006856505147845013
```

So, we have a possible solution. x = 1006856505147845013, y = 81.

Q8

8a

> **(a)** Suppose you know that
>
> $$33335^2 \equiv 670705093^2 \pmod{670726081}.$$
>
> Use this information to factor 670726081.

This is the same situation as Q6. Using the Quadratic Sieve, the factors are found easily.

```
In [7]: n = 670726081
        a = 670705093
        b = 33335

        p = gcd(a-b, n)
        print(p)

        q = n // p
        print(q)

Out[7]: 54323
        12347
```

8b

> **(b)** Suppose you know that $3^2 \equiv 670726078 \pmod{670726081}$. Why won't this information help you to factor 670726081?

The Quadratic Sieve requires the condition that $x \not\equiv \pm y \pmod{n}$. Here, we have:

$$670726078 \equiv -3 \mod 670726081$$

So, this won't help factorize the number.

Q9

> Suppose you know that
>
> $$\begin{aligned} 2^{958230} &\equiv 1488665 \pmod{3837523} \\ 2^{1916460} &\equiv 1 \pmod{3837523} \end{aligned}$$
>
> How would you use this information to factor 3837523? Note that the exponent 1916460 is twice the exponent 958230.

We see that $2^{1916460} \equiv 1488665^2 \equiv 1^2 \pmod{n}$, and $1488665 \not\equiv \pm 1 \pmod{n}$. Using the Quadratic Sieve, we have:

```
In [8]: n = 3837523
        a = 1488665
        b = 1

        p = gcd(a-b, n)
        print(p)

        q = n // p
        print(q)

Out[8]: 3511
        1093
```

Q10

10a

> **(a)** Suppose the primes p and q used in the RSA algorithm are consecutive primes. How would you factor $n = pq$?

When the primes are consecutive, the Fermat Factorization method works well. We express n as a difference of two squares.

Let $n = x^2 - y^2$. Then $n = (x+y)(x-y)$ is the factorization of n.

To implement it, we calculate $n + 1^2,\ n + 2^2,\ n + 3^2,\ \ldots$ and so on until we find a square. Suppose $n + y^2$ is a square for some integer y. Then, put $x = \sqrt{n + y^2}$. Then, $(x+y)(x-y)$ is the factorization.

It can be implemented in code as follows.

```
In [9]: from sympy.ntheory.primetest import is_square
        import decimal

        def fermat(n):
            y = 1
            while not is_square(n + y*y):
                y += 1

            # Set decimal precision to 56 for large numbers
            decimal.getcontext().prec = 56

            x = int(decimal.Decimal(n + y*y).sqrt())

            return (x - y, x + y)
```

10b

> **(b)** The ciphertext 10787770728 was encrypted using e = 113 and n = 10993522499. The factors p and q of n where chosen so that $q - p = 2$. Decrypt the message.

In the Fermat factorization method above, since $p = x - y$ and $q = x + y$, we would get $x = \frac{p+q}{2}$ and $y = \frac{q-p}{2}$. This means y = 1. And so, $x = \sqrt{n + 1^2} =$ 104850. Thus, the factors are 104849 and 104851.

We could also have obtained the same with our function.

We define certain helpful functions first, as per the convention. `compute_d` computes the decryption key d. `text2int` converts text to integer. `int2text` converts integer to text. This is all based on the convention that a -> 1, b ->2, and so on. Spaces are encoded as 00. `rsa_decrypt` decrypts the ciphertext using the decryption key.

```
In [10]: def compute_d(p, q, e):
             phi = (p - 1) * (q - 1)
             d = pow(e, -1, phi)

             return d

In [11]: def text2int(message):
             # Convert to lowercase for uniformity
             message = message.lower()
             ciphertext = ""

             for character in message:
                 # Handle spaces specially
                 if character == " ":
                     ciphertext += "00"
```

```
                continue
            # Pad with "0" on left and take last two characters
            ciphertext += ("0" + str(ord(character) - 96))[-2:]

        return int(ciphertext)

In [12]: def int2text(number):
        number = str(number)

        # Append "0" on left if odd length string
        if len(number) % 2 != 0:
            number = "0" + number

        # Split into groups of two-digit numbers
        character_nums = [number[i:i+2] for i in range(0, len(number), 2)]

        # Convert each group into characters
        characters = [chr(int(i) + 96) for i in character_nums]

        message = ''.join(characters)

        # Replace the character ` (ASCII 96) to space
        message = message.replace('`', ' ')

        return message

In [13]: def rsa_decrypt(ciphertext, n, d):
        return int2text(pow(ciphertext, d, n))
```

Now, we can go about decrypting the message.

```
In [14]: n = 10993522499
         e = 113
         c = 10787770728

         p, q = fermat(n)
         d = compute_d(p, q, e)
         m = rsa_decrypt(c, n, d)

         print(f"{p = }")
         print(f"{q = }")
         print(f"{d = }")
         print(f"{m = }")

Out[14]: p = 104849
         q = 104851
         d = 5545299377
         m = 'easy'
```

The message was `easy`.

10c

> **(c)** The following ciphertext c was encrypted mod n using the exponent e:
>
> $$\begin{aligned} n &= 152415787501905985701881832150835089037858868621211004433 \\ e &= 9007 \\ c &= 141077461765569500241199505617854673388398574333341423525 \end{aligned}$$
>
> The prime factors p and q of n are consecutive primes. Decrypt the message.

Using the functions we developed, it goes like this.

```
In [15]: n = 152415787501905985701881832150835089037858868621211004433
         c = 141077461765569500241199505617854673388398574333341423525
         e = 9007

         p, q = fermat(n)
         d = compute_d(p, q, e)
         m = rsa_decrypt(c, n, d)

         print(f"{p = }")
         print(f"{q = }")
         print(f"{d = }")
         print(f"{m = }")

Out[15]: p = 1234567890000000314159265000031
         q = 1234567890000000314159265000143
         d = 66046277186625853468906938024685131899784936049279925683
         m = 'this number was not secure'
```

The message was `this number was not secure`.