Task: Given a unique set of data


```
{"srn": "555555",
"name": "Anastasia Rizzo",
"exercise1": {
"p": "2685735181983467",
"g": "2",
"a": "3628323521",
"b": "5915667893",
"cipherText": {
"encoded": "1580593562655238",
"base64": "BZ2KndYaBg=="
}
}
}
```


Question 4. Verify whether *g* is a generator for *p*. Provide a brief explanation and include the method from your code, as well as the runtime

Given the following:
Prime number **p** = 2685735181983467
Generator **g** = 2

Prime number **p** can be a special kind of prime called **"safe" prime**.
Number **p** is a safe prime if both **p** and **(p-1) /2** are prime numbers. (Wiener, 2003)

If **p** is a safe prime, it will be much easier to verify if **g** is a generator for **p**.

To check if number **p** is a safe prime, the following formulas have been applied:

To find q: **q = (p-1)/2**
q = (2685735181983467 – 1)/2 = 1342867590991733
Snippet from java code (presented in Appendix 4, page 13, 14):

```
q = ((p.subtract(one)).divide(two)); // q=(p-1)/2
```

To find d: **d = g^q mod p**
d = $2^{1342867590991733}$ mod 2685735181983467 = 2685735181983466
Snippet from java code (presented in Appendix 4, page 13, 14):

```
d = g.modPow(q,p); // d=g^q mod p or d=p-1
```

To find d: **f = g^d mod p**
f = $2^{2685735181983466}$ mod 2685735181983467 = 1
Snippet from java code (presented in Appendix 4, page 13, 14):

```
f = g.modPow(d,p); // check if g^d mod p is prime
```

Result: **q = (p-1)/2** is a prime number, therefore **p** is a **safe prime** number and **g** is a generator for **p.**

To calculate the **runtime:**

Snippet from java code (presented in Appendix 4, page 13, 14):

```
long current_local_time = System.currentTimeMillis(); //runtime
System.out.println("Runtime:  "  +  (System.currentTimeMillis()  -  current_local_time)  +  " millisecond/s"); // print runtime
```

Result: 2 millisecond/s

Question 5. Considering that *a* is Alice's private key and *b* is Bob's private key, compute their public keys and show how they can generate the same shared key. Include a brief explanation and the relevant code snippet.

Given the following:

Prime number **p** = 2685735181983467
Generator **g** = 2
Alice's private key **a** = 3628323521
Bob's private key **b** = 5915667893

To compute Alice's public key (x) and Bob's public key (y) the following formulas have been applied:
$\mathbf{x = g^a \bmod p}$ and $\mathbf{y = g^b \bmod p}$, where "p" is a prime number, "g" is a generator, "a" is a private key.

So, that $\mathbf{x} = 2^{3628323521} \bmod 2685735181983467 = 1268048862489310$
$\mathbf{y} = 2^{5915667893} \bmod 2685735181983467 = 1346574413247867$

Snippet from java code (presented in Appendix 5, page 14, 15):

```
x = g.modPow(a, p); // Alice's public key
y = g.modPow(b, p); // Bob's public key y
```

Result:

Alice's public key **x** = 1268048862489310
Bob's public key **y** = 1346574413247867

To compute Alice's shared key (kA) and Bob's shared key (kB) the following formulas have been applied:
$\mathbf{kA = y^a \bmod p}$ and $\mathbf{kB = x^b \bmod p}$, where "y" is Bob's public key, "x" is Alice's public key, "a" is a private key, "p" is a prime number.

So, that: $\mathbf{kA} = 1346574413247867^{3628323521} \bmod 2685735181983467 = 789708769021392$
$\mathbf{kB} = 1268048862489310^{5915667893} \bmod 2685735181983467 = 789708769021392$

Snippet from java code (presented in Appendix 5, page 14, 15):

```
kA = y.modPow(a, p); // Alice's shared key k
kB = x.modPow(b, p); // Bob's shared key k
```

Result:
Alice's shared key **k** = 789708769021392
Bob's shared key **k** = 789708769021392

Question 6. Decrypt the provided cipher text which has been encrypted with the shared key that you computed in Question 5. Include a brief explanation and the relevant code snippet.

Given the following:
Prime number **p** = 2685735181983467
Generator **g** = 2
Alice's private key **a** = 3628323521
Bob's private key **b** = 5915667893
Alice's public key **x** = 1268048862489310
Bob's public key **y** = 1346574413247867
Alice's shared key **k** = 789708769021392
Bob's shared key **k** = 789708769021392
Cypher text "encoded" **c** = 1580593562655238

To decrypt the provided cypherText "encoded" the following formulas have been applied:
xInv = k mod p, where "k" is a shared key, "p" is a prime number.
m = (c * k) mod p, where "c" is a provided cypher text, "k" is a shared key, "p" is a prime number.

So that:
xInv = 789708769021392 mod 2685735181983467 = 1139504415921727
m = (1580593562655238 * 789708769021392) mod 2685735181983467 = 495890948466

Snippet from java code (presented in Appendix 6, page 14, 15):

```
xInv = kA.modInverse(p); // Inverse of x in decryption part
m = c.multiply(xInv).mod(p); // plain text "encoded"
```

Result:
m = 495890948466

To decode plainText "encoded" from BigInteger to plainText "base64" this snippet of code has been applied:

```
Base64.Encoder encoder = Base64.getEncoder();
byte[] bigIntegerBytes = BigInteger.valueOf(495890948466L).toByteArray();
String base64EncodedBigIntegerBytes = encoder.encodeToString(bigIntegerBytes);
System.out.println("Plain text base64: "+ base64EncodedBigIntegerBytes);
```

To decode from plainText "base64" to plainText "text" this snippet of code has been applied:

```
Decoder decoder1 = Base64.getDecoder();
byte[] bytes = decoder1.decode(base64EncodedBigIntegerBytes);
String decodedString = new String(bytes, UTF_8);
```

Result:
The plainText “text” is: **sugar**

Question 7. Suppose Alice and Bob want to generate a new set of keys. They decide that they should use a 17-digit prime instead. How would they go on about generating a new *p* and a corresponding generator *g*? Provide a brief explanation and include the relevant code snippet, as well as its runtime.

Need to generate a new set of keys:
17-digit prime p and a corresponding **generator g.**

First, a new random **17-digit prime p** was generated.
Second, a new random **corresponding generator g** was generated.
Third, **if** number **p is a safe prime** was checked.
Fourth, whether **g is a generator for p** was verified.
Finally, **runtime** was calculated.

To generate new **17-digit prime p**:
A random number generator has been applied.
Snippet from java code (presented in Appendix 7, page 16, 17):

```
Random r = new Random(); // random number r
int numDigits = 17; // number of digits(17)
double LOG_2 = Math.log(10)/Math.log(2);
int numBits = (int) (numDigits * LOG_2);
p = new BigInteger (numBits, r); // prime number p
```

Result: new **17-digit prime p** = 7216142699972837

To generate a corresponding **generator g**:
Random number generator has been applied.
Snippet from java code (presented in Appendix 7, page 16, 17):

```
int low = 3; // low range
int high = 10; // high range
int e = r.nextInt((high - low) + 1) + low; // calculations of int e value in range between 3 and 10; int e = Integer i = BigInteger g
```

Result: a corresponding **generator g** = 3

To check if number **p** is a safe prime the following formulas have been applied:

To find q: **q = (p-1)/2**
q = (7216142699972837 – 1)/2 = 3608071349986418
Snippet from java code (presented in Appendix 7, page 16, 17):
```
q = ((p.subtract(one)).divide(two)); // q=(p-1)/2
```

To find d: **d = g^q mod p**
d = $3^{3608071349986418}$ mod 7216142699972837 = 7216142699972836
Snippet from java code (presented in Appendix 7, page 16, 17):
```
d = g.modPow(q,p); // d=g^q mod p or d=p-1
```

To find d: **f = g^d mod p**
f = $3^{7216142699972836}$mod 7216142699972837 = 1
Snippet from java code (presented in Appendix 7, page 16, 17):
```
f = g.modPow(d,p); // check if g^d mod p is prime
```

Result: **q = (p-1)/2** is a prime number, therefore **p** is a **safe prime** number and **g** is a generator for **p.**

To calculate the **runtime:**
Snippet from java code (presented in Appendix 7, page 16, 17):
```
long current_local_time = System.currentTimeMillis(); //runtime
System.out.println("Runtime: " + (System.currentTimeMillis() - current_local_time) + " millisecond/s"); // print runtime
```

Result: 2 millisecond/s.

Question 8. Generate a new set of private and public keys for Alice and Bob, using the *p* and *g* you generated in Question 7. Encrypt your SRN with the shared key. Include a brief explanation and the relevant code snippet.

Note:

The Question 8 was split on 2 parts.

Part 1 – to generate private keys (random) (code in Appendix 8 part 1, page 17, 18);

Part 2 – to generate public keys; to encrypt my SRN number with the shared key (code in Appendix 8 part 2, page 19, 20).

Part 1: To generate a new set of private keys (random).

Need to generate:
a new set of **private keys** for Alice (**a**) and Bob (**b**):

First, new **6-digit random numbers a** and **b** was generated (in this example, 6 digits were used, but it this can be changed to any number of digits).
Second, **if** number **a** and **b** are primes, has been checked by using Time Complexity: O(√N).
Third, **runtime** was calculated.

To generate new **6-digit random numbers a and b**:
A random number generator has been applied.
Snippet from java code (presented in Appendix 8 part 1, page 17, 18):

```
Random r = new Random(); // random number r
int numDigits = 6; // number of digits(6)
double LOG_2 = Math.log(10)/Math.log(2);
int numBits = (int) (numDigits * LOG_2);
p = new BigInteger (numBits, r); // random number p
```

Result: new **6-digit number a** = 300889 and new **6-digit number b** = 288293.

To check **if** number **a** and **b** are primes, Time Complexity O(√N) was applied:
Snippet from java code (presented in Appendix 8 part 1, page 17, 18):

```
OSQRTNMethod(a);
      }
    // Time Complexity: O(√N)
     static void OSQRTNMethod(long a){
       boolean isPrime = true; // boolean

          // if a is not a prime
          for (int i = 2; i <=Math.sqrt(a) ; i++) {
             if(a%i==0) {
                System.out.println("Number " + a +" is not a prime ");
                 isPrime = false;
                 break;
            }
          }
              // if a is a prime
              if(isPrime)
                 System.out.println("Number " + a +" is a prime ");
```

Result:
Alice's private key a = 300889 is prime
Bob's private key b = 288293 is prime

To calculate the **runtime:**
Snippet from java code (presented in Appendix 8 part 1, page 17, 18):

```
long current_local_time = System.currentTimeMillis(); // runtime
System.out.println("Runtime:  " + (System.currentTimeMillis() - current_local_time) + " millisecond/s"); // print runtime
```

Result: 1 millisecond/s.

Part 2: To generate Alice's (x) and Bob (y) public keys; to encrypt my SRN number with the shared key.

Given the following:
Prime number **p** = 7216142699972837
Generator **g** = 3
Alice's private key **a** = 300889
Bob's private key **b** = 288293
Original message **m** = 140359547 (*my SRN number*)

To compute Alice's public key (x) and Bob's public key (y) the following formula has been applied:
$\mathbf{x = g^a \bmod p}$ and $\mathbf{y = g^b \bmod p}$, where "p" is a prime number, "g" is a generator, "a" is a private key.

So, that $\mathbf{x} = 3^{300889} \bmod 7216142699972837 = 6321511077026144$
$\mathbf{y} = 3^{288293} \bmod 7216142699972837 = 2205468983963930$

Snippet from java code (presented in Appendix 8 part 2, page 19, 20):

```
x = g.modPow(a, p); // Alice's public key x calculations
y = g.modPow(b, p); // Bob's public key y calculations
```

To compute Alice's shared key (kA) and Bob's shared key (kB) the following formula has been applied:
$\mathbf{kA = y^a \bmod p}$ and $\mathbf{kB = x^b \bmod p}$, where "y" is Bob's public key, "x" is Alice's public key, "a" is a private key, "p" is a prime number.

So, that: $\mathbf{kA} = 6321511077026144^{300889} \bmod 7216142699972837 = 5704511815613684$
$\mathbf{kB} = 2205468983963930^{288293} \bmod 7216142699972837 = 5704511815613684$

Snippet from java code (presented in Appendix 8 part 2, page 19, 20):

```
kA = y.modPow(a, p); // Alice's shared key k calculations
kB = x.modPow(b, p); // Bob's shared key k calculations
```

To encrypt my SRN number with the shared key the following formulas have been applied:

$\mathbf{r = g^{kA} \bmod p}$, where "g" is a generator, "kA" is a shared key, "p" is a prime number.
$\mathbf{z = x^{kA} \bmod p}$, where "x" is a public key, "kA" is a shared key, "p" is a prime number.
$\mathbf{c = (m_i * x) \bmod p}$, where "$m_i$" is a message, "x" is a public key, "p" is a prime number.

So, that:
$\mathbf{r} = 3^{5704511815613684} \bmod 7216142699972837 = 654424626614820$

z = 6321511077026144 ^{5704511815613684} mod 7216142699972837 = 5566298450981226
c = (140359547 * 6321511077026144) mod 7216142699972837 = 3859422092333348

<u>Snippet from java code (presented in Appendix 8 part 2, page 19, 20):</u>

```
r = g.modPow(kA, p);// r=g^kA mod p calculations; for further calculations, Alice's shared key "kA" will be used as a shared key.

z = x.modPow(kA, p); // z=x^kA mod p calculations

c = m.multiply(z).mod(p); // c=(m*z)mod p; cypher text calculations
```

To decrypt the provided cypherText “encoded” the following formulas have been applied:

xd = r ^{a} mod p, where “r” is a cypher text pair, “a” is an Alice’s private key, “p” is a prime number.
xInverse = xd mod p

md = (xInverse * c) mod p, where “xInverse” is an inverse of x, “c” is a provided cypher text, “p” is a prime number.

So that:
xd = 654424626614820 ^{300889} mod 7216142699972837 = 5566298450981226
xInverse = 5566298450981226 mod 7216142699972837 = 6928872166590396
md = (1580593562655238 * 789708769021392) mod 2685735181983467 = 140359547

<u>Snippet from java code (presented in Appendix 8 part 2, page 19, 20):</u>

```
xd = r.modPow(a, p); // xd=r^a mod p calculations
xInverse = xd.modInverse(p); // inverse of x=x mod p=x^-1 calculations
md = xInverse.multiply(c).mod(p); // md=(x^-1 * c) mod p calculations
```

To encrypt and decrypt cypher text with Base64:
<u>Snippet from java code (presented in Appendix 8 part 2, page 19, 20):</u>

```
// Base64
    Base64.Encoder encoder = Base64.getEncoder();
    byte[] bigIntegerBytes = BigInteger.valueOf(2261795017164508L).toByteArray();
    String base64EncodedBigIntegerBytes = encoder.encodeToString(bigIntegerBytes);
    System.out.println("Plain text base64: "+ base64EncodedBigIntegerBytes);

    Decoder decoder1 = Base64.getDecoder();
    byte[] bytes = decoder1.decode(base64EncodedBigIntegerBytes);
    String decodedString = new String(bytes, UTF_8);
    // System.out.println("Plain text base64: "+ decodedString);
```

<u>Result:</u> CAkXMI2G3A==

References

Irvin, J., 2019. *BrutePrime.java.* [Online]
Available at: https://gist.github.com/jonathan-irvin/22f8339849d0e5b44b6a
[Accessed 03 April 2019].

Wiener, J. M., 2003. *Safe Prime Generation with a Combined Sieve,* s.l.: s.n.