1. Answers

   (a) No, BG does not have the $K_{C,TGS}$ to decrypt the necessary information.

   (b) If the lifetime of the ticket was too short, the client could not use the service for a meaningful amount of time. If the lifetime of the ticket was too long, the BG could get onto the clients machine and still use the same ticket with reprompting for the password.

   (c) Yes, $K_{C,V}$ is used to authenticate both the client and server in round (6). Without the key, this would not be possible with the same configuration.

2. Answers

   (a) Given the equation $s_i = K^{-1}(H(M_i) + xr) \pmod{q}$ and the variables $H(M_1) = 5$, $H(M_2) = 2$, $H(M_3) = 3$, $s_1 = 3$, $s_2 = 4$, and $q = 11$, s_3 is found to be the following:

   $$s_1 = 3 = K^{-1}(5 + xr) \pmod{11}$$
   $$s_2 = 4 = K^{-1}(2 + xr) \pmod{11}$$

   By elimination of a system of equations:

   $$-1 = 3K^{-1} \pmod{11}$$
   $$-1 \times 3^{-1} \pmod{11} = K^{-1}$$
   $$-1 \times 4 \pmod{11} = K^{-1}$$
   $$10 \times 4 \pmod{11} = K^{-1}$$
   $$40 \pmod{11} = K^{-1}$$
   $$7 \pmod{11} = K^{-1}$$

   Then the modular inverse of K^{-1} is found:

   $$K \times K^{-1} = 1 \pmod{11}$$
   $$K \times 7 = 1 \pmod{11}$$
   $$8 \times 7 = 1 \pmod{11}$$
   $$\therefore K = 8$$

   To find xr, K^{-1} is plugged into the original equation for s_1:

   $$s_1 = 3 = 7(5 + xr) \pmod{11}$$
   $$3 \times 7^{-1} = 5 + xr \pmod{11}$$
   $$3 \times 8 = 5 + xr \pmod{11}$$
   $$24 - 5 = xr \pmod{11}$$
   $$19 = xr \pmod{11}$$
   $$8 = xr$$

   Using the known value of xr, K^{-1}, and $H(M_3)$, s_3 can be found from the original equation:

   $$s_3 = 7(3 + 8) \pmod{11}$$
   $$s_3 = 7 \times 11 \pmod{11}$$
   $$s_3 = 77 \pmod{11}$$
   $$s_3 = 0$$

3. Answers

   (a) Timestamps need less rounds of communication to verify that the message is current. Nonces do not need to have synchronized clocks to work.

(b)
$$p = [2(2-1)]/[2 \times 2^{64}]$$
$$= 1/2^{64}$$

(c) PGP has the user create a password and prompts for it every time the private key needs to be used. The private key is encrypted with the SHA hash of the password and is decrypted the same way.

(d) No, once the SSL handshake finished, all data sent after the handshake is encrypted with the session key.

(e) Cryptographic keys are generated from a random number created each handshake. This number prevents replay attacks from happening.

4. Answers

(a) No, BG would calculate x (mod 73) for each number with a zero in the one's digit place, making the domain the set:

$$\{x | 0 \leq x < 910, 10 | x\} \tag{1}$$

This set has $\approx 10^2 = 100$ unique values compared to the original 1000, increasing the amount of possible collisions. This calculation works because M (mod 91) will always yield a result between 0 and 91. Multiplying that range by 10 makes the possible range for x the same as set 1. BG would then find an x where x (mod 73) is equal to $h(M)$ and find an M' that satisfies the equation:

$$M' = (x/10) \text{ (mod 91)}$$

BG would then replace M with M'. The following shows a collision for $M = 103$:

$$M = 103$$
$$h(M) = (103 \text{ (mod 91)} * 10) \text{ (mod 73)}$$
$$= (12 * 10) \text{ (mod 73)}$$
$$= 120 \text{ (mod 73)}$$
$$= 47$$

$$x = 850$$
$$h(M) \stackrel{?}{=} x \text{ (mod 73)} = 850 \text{ (mod 73)}$$
$$h(M) = 47$$
$$M' = (850/10) \text{ (mod 91)} = 85$$

Programming Problem

Self-critique

The program works as expected. I am curious if there are any restrictions for p, q, or h.

Program Output

```
    hoodz@DESKTOP-IOQ6GTP:/mnt/c/Users/hoodz/Desktop/Coding/CSCI-4534-Cryptography/
        Homework-3$ ./main
Running all unit tests for DSA class
------------------------------------------------------------------------
Running Unit Test - Input A...
g: 2
publicKey: 4
------------------------------------------------------------------------
```

```
Sign Hash:
r: 2
s: 1
--------------------------------------------------------------------------------
Verify Real Hash
w: 1
u1: 0
u2: 2
v: 2
v == r: true
--------------------------------------------------------------------------------
Verify Fake Hash
w: 1
u1: 1
u2: 2
v: 1
v == r: false
--------------------------------------------------------------------------------
Unit Test - Input A -> Passed Test
Running Unit Test - Input B...
g: 25
publicKey: 27
--------------------------------------------------------------------------------
Sign Hash:
r: 16
s: 9
--------------------------------------------------------------------------------
Verify Real Hash
w: 18
u1: 21
u2: 12
v: 16
v == r: true
--------------------------------------------------------------------------------
Verify Fake Hash
w: 18
u1: 3
u2: 12
v: 4
v == r: false
--------------------------------------------------------------------------------
Unit Test - Input B -> Passed Test
Running Unit Test - Input C...
g: 8
publicKey: 19
--------------------------------------------------------------------------------
Sign Hash:
r: 8
s: 23
--------------------------------------------------------------------------------
Verify Real Hash
w: 37
u1: 7
u2: 46
v: 8
v == r: true
--------------------------------------------------------------------------------
Verify Fake Hash
w: 37
u1: 21
```

```
u2: 46
v: 1
v == r: false
-------------------------------------------------------------------------------
Unit Test - Input C -> Passed Test
-------------------------------------------------------------------------------
Finished running all tests
 Passed Tests: 3
 Failed Tests: 0
hoodz@DESKTOP-IOQ6GTP:/mnt/c/Users/hoodz/Desktop/Coding/CSCI-4534-Cryptography/Homework-3
    $
```

Human-readable code

main.cpp

```cpp
#include "DSA.hpp"
#include "UnitTest-DSA.hpp"

using namespace std;

int main(){
    UnitTest_RunAll();
    return 0;
}
```

UnitTest-DSA.hpp

```cpp
#ifndef UNITTEST_DSA_HPP
#define UNITTEST_DSA_HPP

#include <iostream>
#include "DSA.hpp"

using namespace std;

/// ----------------------------------------------------------------------
/// @brief see if input 1 on homework works for DSA class
/// @return int 0 on pass, 1 on fail
/// ----------------------------------------------------------------------
int UnitTest_InputA();

/// ----------------------------------------------------------------------
/// @brief see if input 2 on homework works for DSA class
/// @return int 0 on pass, 1 on fail
/// ----------------------------------------------------------------------
int UnitTest_InputB();

/// ----------------------------------------------------------------------
/// @brief see if custom inputs work for DSA class
/// @return int 0 on pass, 1 on fail
/// ----------------------------------------------------------------------
int UnitTest_InputC();

/// ----------------------------------------------------------------------
/// @brief run all the unit tests for the DSA class
/// @return int 0 on pass, 1 on fail
/// ----------------------------------------------------------------------
int UnitTest_RunAll();

#endif
```

UnitTest-DSA.cpp

```cpp
#include "UnitTest-DSA.hpp"

/// ----------------------------------------------------------------------
/// @brief see if input 1 on homework works for DSA class
/// @return int 0 on pass, 1 on fail
/// ----------------------------------------------------------------------
int UnitTest_InputA(){
    try {
        int p = 7,
            q = 3,
            h = 3,
            x = 2,
            k = 1,
            HM1 = 3,
            HM2 = 4;

        DSA cipher(p, q, h, x, true, true);
        cout << "-----------------------------------------------------------------------------" << endl;
        cout << "Sign Hash:" << endl;
```

```cpp
20              const Signature sign = cipher.signHash(HM1, k);
21
22              if (sign.s != 1 || sign.r != 2) throw "Returned signature is wrong!";
23
24              cout << "-------------------------------------------------------------------------" << endl;
25              cout << "Verify Real Hash" << endl;
26              bool goodVerification = cipher.verifyHash(HM1, sign);
27
28              if (!goodVerification) throw "Good verification went wrong!";
29
30              cout << "-------------------------------------------------------------------------" << endl;
31              cout << "Verify Fake Hash" << endl;
32              bool badVerification = cipher.verifyHash(HM2, sign);
33
34              if (badVerification) throw "Bad verification went wrong!";
35              cout << "-------------------------------------------------------------------------" << endl;
36      } catch(const char* txt) {
37          cout << txt << endl; return 1;
38      } catch (...) { return 1; }
39
40      return 0;
41  }
42
43  /// -----------------------------------------------------------------------------
44  /// @brief see if input 2 on homework works for DSA class
45  /// @return int 0 on pass, 1 on fail
46  /// -----------------------------------------------------------------------------
47  int UnitTest_InputB(){
48      try {
49          int p = 47,
50              q = 23,
51              h = 5,
52              x = 7,
53              k = 13,
54              HM1 = 5,
55              HM2 = 4;
56
57          DSA cipher(p, q, h, x, true, true);
58              cout << "-------------------------------------------------------------------------" << endl;
59              cout << "Sign Hash:" << endl;
60              const Signature sign = cipher.signHash(HM1, k);
61
62              if (sign.s != 9 || sign.r != 16) throw "Returned signature is wrong!";
63
64              cout << "-------------------------------------------------------------------------" << endl;
65              cout << "Verify Real Hash" << endl;
66              bool goodVerification = cipher.verifyHash(HM1, sign);
67
68              if (!goodVerification) throw "Good verification went wrong!";
69
70              cout << "-------------------------------------------------------------------------" << endl;
71              cout << "Verify Fake Hash" << endl;
72              bool badVerification = cipher.verifyHash(HM2, sign);
73
74              if (badVerification) throw "Bad verification went wrong!";
75              cout << "-------------------------------------------------------------------------" << endl;
76      } catch(const char* txt) {
77          cout << txt << endl; return 1;
78      } catch (...) { return 1; }
79
80      return 0;
81  }
82
83  /// -----------------------------------------------------------------------------
84  /// @brief see if custom inputs work for DSA class
85  /// @return int 0 on pass, 1 on fail
86  /// -----------------------------------------------------------------------------
87  int UnitTest_InputC(){
88      try {
89          int p = 151,
90              q = 50,
91              h = 64,
92              x = 29,
93              k = 41,
94              HM1 = 11,
95              HM2 = 33;
```

```cpp
96
97              DSA cipher(p, q, h, x, true, true);
98              cout << "-------------------------------------------------------------------------------" << endl;
99              cout << "Sign Hash:" << endl;
100             const Signature sign = cipher.signHash(HM1, k);
101
102             if (sign.s != 23 || sign.r != 8) throw "Returned signature is wrong!";
103
104             cout << "-------------------------------------------------------------------------------" << endl;
105             cout << "Verify Real Hash" << endl;
106             bool goodVerification = cipher.verifyHash(HM1, sign);
107
108             if (!goodVerification) throw "Good verification went wrong!";
109
110             cout << "-------------------------------------------------------------------------------" << endl;
111             cout << "Verify Fake Hash" << endl;
112             bool badVerification = cipher.verifyHash(HM2, sign);
113
114             if (badVerification) throw "Bad verification went wrong!";
115             cout << "-------------------------------------------------------------------------------" << endl;
116      } catch(const char* txt) {
117          cout << txt << endl; return 1;
118      } catch (...) { return 1; }
119
120      return 0;
121 }
122
123 /// -----------------------------------------------------------------------------
124 /// @brief run all the unit tests for the DSA class
125 /// @return int 0 on pass, 1 on fail
126 /// -----------------------------------------------------------------------------
127 int UnitTest_RunAll(){
128      int passed = 0, failed = 0;
129
130      cout << "Running all unit tests for DSA class" << endl;
131      cout << "-------------------------------------------------------------------------------" << endl;
132
133      cout << "Running Unit Test - Input A..." << endl;
134      if (UnitTest_InputA()) { cout << "Unit Test - Input A -> Failed Test \u274c" << endl; failed++;}
135      else { cout << "Unit Test - Input A -> Passed Test \u2713" << endl; passed++; }
136
137      cout << "Running Unit Test - Input B..." << endl;
138      if (UnitTest_InputB()) { cout << "Unit Test - Input B -> Failed Test \u274c" << endl; failed++;}
139      else { cout << "Unit Test - Input B -> Passed Test \u2713" << endl; passed++; }
140
141      cout << "Running Unit Test - Input C..." << endl;
142      if (UnitTest_InputC()) { cout << "Unit Test - Input C -> Failed Test \u274c" << endl; failed++;}
143      else { cout << "Unit Test - Input C -> Passed Test \u2713" << endl; passed++; }
144
145
146      cout << "-------------------------------------------------------------------------------" << endl;
147      cout << "Finished running all tests" << endl;
148      cout << "\u2713 Passed Tests: " << passed << endl;
149      cout << "\u274c Failed Tests: " << failed << endl;
150
151      return 0;
152 }
```

DSA.hpp

```cpp
1  #ifndef DSA_HPP
2  #define DSA_HPP
3
4  #include <iostream>
5  #include "Cryptography.hpp"
6
7  using namespace std;
8
9  // see powerpoint for naming convention
10
11 struct Signature {
12     int r;
13     int s;
14 };
```

```
15
16  class DSA {
17  private:
18      // privateKey = x, publicKey = y
19      int p, q, h, privateKey;    // globally known
20      int publicKey, g;            // derived
21      bool verbose, output;        // output flags
22
23  public:
24      /// ---------------------------------------------------------------------
25      /// @brief constructor for the DSA class, generates g & public key
26      /// @param p (int) a prime number
27      /// @param q (int) a factor of (p-1)
28      /// @param h (int) a chosen number between 0 and p-1, and fastModExponentiation(h, (p-1)/q, p) > 1
29      /// @param privateKey (int) a number between 0 and q
30      /// @param verbose (bool) a flag to show debugging information
31      /// @param output (bool) a flag to show homework outputs
32      /// ---------------------------------------------------------------------
33      DSA(int p, int q, int h, int privateKey, bool verbose = false, bool output = false);
34
35      /// ---------------------------------------------------------------------
36      /// @brief calculates the signature values r & s for the given hash and random number
37      /// @param hash (int) the hash of a message
38      /// @param k (int) a random number < q
39      /// @return (Signature) the r & s values for hash and k
40      /// ---------------------------------------------------------------------
41      Signature signHash(const int hash, const int k) const;
42
43      /// ---------------------------------------------------------------------
44      /// @brief verifies that the signature matches the given hash
45      /// @param hash (int) the hash of a message
46      /// @param sig (Signature) the r & s values for the signature
47      /// @return (bool) true if the signature is paired with hash, otherwise false
48      /// ---------------------------------------------------------------------
49      bool verifyHash(const int hash, const Signature sig) const;
50
51      ~DSA()=default;
52  };
53
54  #endif
```

DSA.cpp

```
1   #include "DSA.hpp"
2
3   /// ---------------------------------------------------------------------
4   /// @brief constructor for the DSA class, generates g & public key
5   /// @param p (int) a prime number
6   /// @param q (int) a factor of (p-1)
7   /// @param h (int) a chosen number between 0 and p-1, and fastModExponentiation(h, (p-1)/q, p) > 1
8   /// @param privateKey (int) a number between 0 and q
9   /// @param verbose (bool) a flag to show debugging information
10  /// @param output (bool) a flag to show homework outputs
11  /// ---------------------------------------------------------------------
12  DSA::
13  DSA(int p, int q, int h, int privateKey, bool verbose, bool output) :
14  p(p), q(q), h(h), privateKey(privateKey), verbose(verbose), output(output) {
15      g = fastModExponentiation(h, (p-1)/q, p);              // h ^ ([p-1]/q) mod p
16      publicKey = fastModExponentiation(g, privateKey, p);   // g ^ x mod p
17
18      if (verbose || output){
19          cout << "g:             " << g << endl;
20          cout << "publicKey:     " << publicKey << endl;
21      }
22  }
23
24  /// ---------------------------------------------------------------------
25  /// @brief calcuates the signature values r & s for the given hash and random number
26  /// @param hash (int) the hash of a message
27  /// @param k (int) a random number < q
28  /// @return (Signature) the r & s values for hash and k
29  /// ---------------------------------------------------------------------
30  Signature DSA::
31  signHash(const int hash, const int k) const{
```

8

```cpp
    int r = fastModExponentiation(g, k, p) % q;              // (g ^ k mod p) mod q
    int inverseK = modInverse(k, q);                         // k^-1 mod q
    int s = (inverseK * (hash + (privateKey * r))) % q;      // [k^-1 (hash + privateKey * r)] mod q

    if (verbose || output){
        cout << "r:      " << r << endl;
        cout << "s:      " << s << endl;
    }

    return Signature {
        r,
        s
    };
}

/// ---------------------------------------------------------------------------
/// @brief verifies that the signature matches the given hash
/// @param hash (int) the hash of a message
/// @param sig (Signature) the r & s values for the signature
/// @return (bool) true if the signature is paired with hash, otherwise false
/// ---------------------------------------------------------------------------
bool DSA::
verifyHash(const int hash, const Signature sig) const{
    int w = modInverse(sig.s, q);
    int u1 = (hash * w) % q;
    int u2 = (sig.r * w) % q;
    int v = ((fastModExponentiation(g, u1, p) * fastModExponentiation(publicKey, u2, p)) % p)  % q;

    if (verbose || output){
        cout << "w:         " << w << endl;
        cout << "u1:        " << u1 << endl;
        cout << "u2:        " << u2 << endl;
        cout << "v:         " << v << endl;
        cout << "v == r:    " << boolalpha << (v == sig.r) << endl;
    }
    return v == sig.r;
}
```

Cryptography.hpp

```cpp
#ifndef CRYPTOGRAPHY_HPP
#define CRYPTOGRAPHY_HPP

using namespace std;

///
    ----------------------------------------------------------------------------------------------------------
/// @brief      helper function to calculate modular inverse, ax = 1 mod n from modLinearEquationSolver
/// @param a    (int) positive known term in the above equation
/// @param n    (int) modulus value of the equation
/// @return     (int) modular inverse of a mod n
/// @throw      if the modular inverse cannot be found, a const char* message notification is thrown
///
    ----------------------------------------------------------------------------------------------------------

int modInverse(int a, int n);

///
    ----------------------------------------------------------------------------------------------------------
/// @brief      uses the extended Euclidean algorithm to find the solution to ax = b mod n
/// @param a    (int) positive known term in the above equation
/// @param b    (int) positive congruent term in the above equation
/// @param n    (int) modulus value of the equation
/// @return     (int) the solution x to the equation ax = b mod n
/// @throw      if the modular inverse cannot be found, a const char* message notification is thrown
///
    ----------------------------------------------------------------------------------------------------------

int modLinearEquationSolver(int a, int b, int n);

///
    ----------------------------------------------------------------------------------------------------------
```

```
26  /// @brief      the extended Euclidean algorithm to find the greatest common denominator
27  ///             where remainder = ax + by = GCD(a, b)
28  /// @param a    (int) the first term to find the GCD with
29  /// @param b    (int) the second term to find the GCD with
30  /// @param x    (int*) return parameter to return Bezout coefficient x
31  /// @param y    (int*) return parameter to return Bezout coefficient y
32  /// @return     (int) the GCD of a & b, as well as Bezout coefficients in x & y
33  /// @cite       https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm#Pseudocode
34  /// @date       4 Mar 2024
35  ///
       -----------------------------------------------------------------------------------------------

36  int gcdExtended(int a, int b, int* x, int* y);
37
38  ///
       -----------------------------------------------------------------------------------------------

39  /// @brief      solve a ^ b mod n fast using less multiplications
40  /// @param a    (int) positive base of the equation
41  /// @param b    (int) positive exponent of the equation
42  /// @param n    (int) modulus value of the equation
43  /// @return     (int) result of a ^ b mod n
44  /// @note       relies on the idea that modding smaller parts gets same result
45  /// @example    a^5 mod n = a^2 mod n * a^2 mod n * a mod n
46  ///
       -----------------------------------------------------------------------------------------------

47  int fastModExponentiation(int a, int b, int n);
48
49  #endif
```

Cryptography.cpp

```
1   #include "Cryptography.hpp"
2
3   ///
        -----------------------------------------------------------------------------------------------

4   /// @brief      helper function to calculate modular inverse, ax = 1 mod n from modLinearEquationSolver
5   /// @param a    (int) positive known term in the above equation
6   /// @param n    (int) modulus value of the equation
7   /// @return     (int) modular inverse of a mod n
8   /// @throw      if the modular inverse cannot be found, a const char* message notification is thrown
9   ///
        -----------------------------------------------------------------------------------------------

10  int
11  modInverse(int a, int n) {
12      return modLinearEquationSolver(a, 1, n);
13  }
14
15  ///
        -----------------------------------------------------------------------------------------------

16  /// @brief      uses the extended Euclidean algorithm to find the solution to ax = b mod n
17  /// @param a    (int) positive known term in the above equation
18  /// @param b    (int) positive congruent term in the above equation
19  /// @param n    (int) modulus value of the equation
20  /// @return     (int) the solution x to the equation ax = b mod n
21  /// @throw      if the modular inverse cannot be found, a const char* message notification is thrown
22  ///
        -----------------------------------------------------------------------------------------------

23  int
24  modLinearEquationSolver(int a, int b, int n) {
25      int x,                  // first Bezout coefficient from gcdExtended
26          y;                  // second Bezout coefficient from gcdExtended
27
28      int gcd = gcdExtended(a, n, &x, &y);    // get the greatest common denominator and the Bezout coefficients
29      if (b % gcd == 0){
30          return ((x % n + n) % n);           // make sure solution x is within modulus range and return it
31      }
32
```

```cpp
33         throw "Modular Linear Equation Solver cannot return a value";
34   }
35
36   ///
        -------------------------------------------------------------------------------------------------------
37   /// @brief      the extended Euclidean algorithm to find the greatest common denominator
38   ///             where remainder = ax + by = GCD(a, b)
39   /// @param a    (int) the first term to find the GCD with
40   /// @param b    (int) the second term to find the GCD with
41   /// @param x    (int*) return parameter to return Bezout coefficient x
42   /// @param y    (int*) return parameter to return Bezout coefficient y
43   /// @return     (int) the GCD of a & b, as well as Bezout coefficients in x & y
44   /// @cite       https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm#Pseudocode
45   /// @date       4 Mar 2024
46   ///
        -------------------------------------------------------------------------------------------------------
47   int
48   gcdExtended(int a, int b, int* x, int* y){
49       int remainder   = a, newRemainder   = b;
50       int bezoutX     = 1, bezoutXCalc    = 0;
51       int bezoutY     = 0, bezoutYCalc    = 1;
52       int quotient    = 0, temp           = 0;
53
54       while (newRemainder != 0){                            // while there is something to divide
55           quotient = remainder / newRemainder;             // get the quotient result
56           temp = newRemainder;
57           newRemainder = remainder - (quotient * newRemainder);   // calculate next remainder
58           remainder = temp;
59
60           temp = bezoutXCalc;
61           bezoutXCalc = bezoutX - (quotient * bezoutXCalc);       // calculate next x Bezout coefficient
62           bezoutX = temp;
63
64           temp = bezoutYCalc;
65           bezoutYCalc = bezoutY - (quotient * bezoutYCalc);       // calculate next y Bezout coefficient
66           bezoutY = temp;
67       }
68
69       // cout << "Quotients of GCD: (" << bezoutYCalc << ", " << bezoutXCalc << ")" << endl;
70       // cout << "Final GCD: " << remainder << endl;
71       // cout << "Bezout coefficients: (" bezoutX << ", " << bezoutY << ")" << endl;
72
73       *x = bezoutX; *y = bezoutY;                          // assign return parameters and return GCD
74       return remainder;
75   }
76
77   ///
        -------------------------------------------------------------------------------------------------------
78   /// @brief      solve a ^ b mod n fast using less multiplications
79   /// @param a    (int) positive base of the equation
80   /// @param b    (int) positive exponent of the equation
81   /// @param n    (int) modulus value of the equation
82   /// @return     (int) result of a ^ b mod n
83   /// @note       relies on the idea that modding smaller parts gets same result
84   /// @example    a^5 mod n = a^2 mod n * a^2 mod n * a mod n
85   ///
        -------------------------------------------------------------------------------------------------------
86   int
87   fastModExponentiation(int a, int b, int n) {
88       int result = 1;
89       a = a % n;
90
91       while (b > 0){
92           if (b & 1) result = (result * a) % n;        // if b is odd, set the new result and mod it by n (result
            * [a mod n])
93           b >>= 1;                                      // another a^2 mod n is added, divide exponent b by 2
94           a = (a * a) % n;                              // a = a^2 mod n
95       }
96
97       return result;
98   }
```