Solutions to Homework Assignment # 6

1. (a) $e = 79 = 1001111_b$

1)	$b_5 = 0$	squaring	$z = 2^2 \bmod 101$	$= e^{10_2}$	$= 4$
2)	$b_4 = 0$	squaring	$z = 4^2 \bmod 101$	$= e^{100_2}$	$= 16$
3)	$b_3 = 1$	squaring	$z = 16^2 \bmod 101$	$= e^{1000_2}$	$= 54$
		multiplication	$z = 54 \cdot 2 \bmod 101$	$= e^{1001_2}$	$= 7$
4)	$b_2 = 1$	squaring	$z = 7^2 \bmod 101$	$= e^{10010_2}$	$= 49$
		multiplication	$z = 49 \cdot 2 \bmod 101$	$= e^{10011_2}$	$= 98$
5)	$b_1 = 1$	squaring	$z = 98^2 \bmod 101$	$= e^{100110_2}$	$= 9$
		multiplication	$z = 9 \cdot 2 \bmod 101$	$= e^{100111_2}$	$= 18$
6)	$b_0 = 1$	squaring	$z = 18^2 \bmod 101$	$= e^{1001110_2}$	$= 21$
		multiplication	$z = 21 \cdot 2 \bmod 101$	$= e^{1001111_2}$	$= \mathbf{42}$

(b) $e = 197 = 11000101_b$

1)	$b_6 = 1$	squaring	$z = 3^2 \bmod 101$	$= e^{10_2}$	$= 9$
		multiplication	$z = 9 \cdot 3 \bmod 101$	$= e^{11_2}$	$= 27$
2)	$b_5 = 0$	squaring	$z = 27^2 \bmod 101$	$= e^{110_2}$	$= 22$
3)	$b_4 = 0$	squaring	$z = 22^2 \bmod 101$	$= e^{1100_2}$	$= 80$
4)	$b_3 = 0$	squaring	$z = 80^2 \bmod 101$	$= e^{11000_2}$	$= 37$
5)	$b_2 = 1$	squaring	$z = 37^2 \bmod 101$	$= e^{110000_2}$	$= 56$
		multiplication	$z = 56 \cdot 3 \bmod 101$	$= e^{110001_2}$	$= 67$
6)	$b_1 = 0$	squaring	$z = 67^2 \bmod 101$	$= e^{1100010_2}$	$= 45$
7)	$b_0 = 1$	squaring	$z = 45^2 \bmod 101$	$= e^{11000100_2}$	$= 5$
		multiplication	$z = 5 \cdot 3 \bmod 101$	$= e^{11000101_2}$	$= \mathbf{15}$

2. (a) $b = 3$; $y = 26$

   (b) $a = 27$; $y = 14$

3. Alice — Bob

   setup: $K_{pr} = a$; $K_{pub} = b$
   publish b, n

   1) choose random session key K
   $y = e_{K_{pub}}(K) = K^b \bmod n$

   $\xrightarrow{y}$

   2) $K = d_{K_{pr}}(y) = y^a \bmod n$

   Alice completely determines the choice of the session key K.

   Note that in practice K might be much longer than needed in a private–key algorithm. E.g., K may have 1024 bits but only 56 actual key bits are needed. In this case just use the 56 MSB (or LSB). Often, it is safe practice to apply a cryptographic hash function first to K and then take the MSB or LSB bits.

4. Idea: Both sides choose a secret key and distribute it to the other party. Both parties combine the two secret keys then form one joint session key.

Alice		Bob
$K_{pr_A} = a_A$ $K_{pub_A} = b_A, n_A$		$K_{pr_B} = a_B$ $K_{pub_B} = b_B, n_B$
1) choose random K_A $y_A = e_{K_{pub_B}}(K_A) = K_A^{b_B} \bmod n_B$	$\xrightarrow{y_A}$	choose random K_B $y_B = e_{K_{pub_A}}(K_B) = K_B^{b_A} \bmod n_A$
	$\xleftarrow{y_B}$	
2) $K_B = d_{K_{pr_A}}(y_B) = y_B^{a_A} \bmod n_A$		$K_A = d_{K_{pr_B}}(y_A) = y_A^{a_B} \bmod n_B$
3) Combine K_A, K_B, e.g. by $K_{SES} = K_A \oplus K_B$		$K_{SES} = K_A \oplus K_B$

Note that there are other combining functions in step 3) than bit wise XOR, e.g., multiplication modulo n. Another possibility, which is cryptographically probably the safest, is to feed both K_A and K_B to a hash function and use the output of it as the session key.

5. (a) A message consists of, let's say, m pieces of cipher-text $y_0, y_1, \ldots, y_{m-1}$. However, the plain-text space is restricted to 26 possible values and the cipher-text space too. That means we only have to test 26 possible plain-text letters for each cipher-text letter:

   test: $\quad y_i \stackrel{?}{=} j^b \bmod n;\ j = 0, 1, \ldots, 25$

   (b) VANILLA

   (c) The attack can be prevented if we artificially enlarge the clear-text with random bits. For instance, in this example, the first 5 bits are reserved for the actual letter and the other 1019 bits would be random. The receiver simply ignores these random bits after decryption. Such padding with random numbers is part of many RSA standards. In practice, however, the padding is usually much shorter than the message, e.g., 100 random padding bits and 924 message bits.

6. (a) RSA encryption and decryption operation: exponentiation. For simplicity we assume $k \approx k - 1$ in this problem.

   Number of multiplications (and squarings) for one exponentiation with k–bit exponent:

   $$\#\otimes = 1.5 \cdot k$$

   complexity of one multiplication:

   $$t_\otimes = c \cdot k^2$$

   $\Rightarrow$ complexity for exponentiation:

   $$t_k = \#\otimes \cdot t_\otimes = 1.5 \cdot c \cdot k^3$$

   $$\frac{t_{1024}}{t_{512}} = \frac{1.5 \cdot c \cdot (2k)^3}{1.5 \cdot c \cdot k^3} = 8$$

   $\Rightarrow$ RSA with 1024 bits is eight times as slow as RSA with 512 bits.

   (b) Karatsuba algorithm:

   $$t'_\otimes = c \cdot k^{\log_2 3}$$

   $$t'_k = 1.5 \cdot k \cdot c \cdot k^{\log_2 3} = 1.5 \cdot c \cdot k^{1+\log_2 3}$$

   $$\frac{t_{1024}}{t_{512}} = \frac{1.5 \cdot c \cdot (2k)^{1+\log_2 3}}{1.5 \cdot c \cdot k^{1+\log_2 3}} = 2^{1+\log_2 3} = 2 \cdot 2^{\log_2 3} = 2 \cdot 3 = 6$$

   $\Rightarrow$ RSA with 1024 bits is only six times slower than RSA with 1024 bits if the Karatsuba algorithm is used for multiplication.

7. The basic idea is to represent the exponent in a radix 2^k representation. That means we group k bits of the exponent together. The first step of the algorithm is to precompute a look-up table with the values $A^0 = 1$, $A^1 = A$, A^2, ..., A^{2^k-1}. Note that the exponents of the look-up table values represent all possible bit patterns of length k.

   After the look-up table has been computed, the two elementary operations in the algorithm are now:

   - Shift intermediate exponent by k positions to the left by performing k subsequent squarings (Recall: The standard s-a-m algorithm shifts the exponent only by one position by performing one squaring per iteration.)

- The exponent has now k trailing zeros at the rightmost bit positions. Fill in the required bit pattern for the exponent by multiplying the corresponding value from the look-up table with the intermediate result.

This main loop is only performed l/k times, where $l+1$ is the bit length of the exponent. Hence, there are only l/k multiplications being performed in the main loop.

An exact description of the algorithm, which is often referred to as *k-ary exponentiation*, is given below[1]. Note that the bit length of the exponent in this description is $t\,k$ bits.

The complexity of the algorithm for an $l+1$ bit exponent is $2^k - 3$ multiplications in the precomputation phase, and about $l-1$ squarings and $l(2^k-1)/2^k$ multiplications in the main loop.

[1]The k-ary exponentiation description was taken from Menezes, van Oorschot, Vanstone: Handbook of Applied Cryptography, CRC Press, 1997.