ASSIGNMENT WEEK 9

All current cryptographic systems are based on certain problems from discrete mathematics being "hard". RSA is a widely-used cryptographic system, and at its core are calculations of the form

$$r = b^e \bmod M$$

where b is a base (integer number), e is an exponent (integer number), M is a modulus (integer number) and r is the remainder of b^e when reduced modulo M. For example, in $4^3 \bmod 17$ we have $b = 4$, $e = 3$, and $M = 17$. Then $4^3 = 64$, and its remainder when reduced by 17 is $r = 13$ since $64 = 3 \cdot 17 + 13$.

For this assignment, you will implement an efficient function **`exponentiationMod`** as outlined below. You may choose either Java or Python for your submission.

If you are choosing Java:

1. Write a function **`exponentiationMod(long base, long exponent, long modulus)`**. Your function will return the remainder when calculating $(base^{exponent} \, mod \, modulus)$. A straight-forward implementation without input checks may use a loop as follows:
```
    public static long exponentationMod(long base, long exponent, long
modulus) {
        long result = 1;
        while(exponent > 0) {
            result *= base;
            result %= modulus;
            exponent--;
        }
        return result;
    }
```
   The complexity of this approach is $O(\text{exponent})$. You will write a function with complexity $O(\log \text{exponent})$ by using the ***binary expansion of the exponent***. For example, to calculate 5^{12}, the intuitive implementation would use a loop of length 12, multiplying variable **`result`** twelve times by **`base`**. As a binary number, though, $12 = 0b1100$ $(12 = 8 + 4 = 2^3 + 2^2)$. We only need to calculate $5^8 \cdot 5^4$. Instead of looping over **`exponent`**, we could just loop over the bit representation of **`exponent`**. (Note that 5, 5^2, 5^4, and 5^8 all will be calculated to get to the two needed factors.)
   Use bit-shift or floor division by 2 to loop over the bits of **`exponent`**. Bit-test with bitwise and. Within the binary expansion loop, update and multiply variables as needed.
2. Use parameter checks to exit with an exception when **`exponent`** is not a non-negative integer. Negative exponents are beyond the scope of this assignment.
3. Test and debug your function. Provide test runs in form of a main file. Run some cases to show your function works. In particular, show the output for
   $\boldsymbol{M = 2^{62} - 1^{16} - 977}$
   $\boldsymbol{B = 2^{33} - 1301}$
   $\boldsymbol{e = 2^{17} + 2^{14} + 2^{8} - 7}$
   Exponents can be calculated with **`Math.pow`** and a type cast to integer.

If you are choosing Python:

1. Write a function **`exponentiationMod(base, exponent, modulus)`**. Your function will return the remainder when calculating $(base^{exponent} \, mod \, modulus)$. A straight-forward implementation without input checks may use a loop as follows:
```
def exponentiationMod(base, exponent, modulus):
```

```
result = 1
for(i in range(exponent)):
        result *= base
        result %= modulus
return result
```

The complexity of this approach is $O(\text{exponent})$. You will write a function with complexity $O(\log \text{exponent})$ by using the ***binary expansion of the exponent***. For example, to calculate 5^{12}, the intuitive implementation would use a loop of length 12, multiplying variable **`result`** twelve times by **`base`**. As a binary number, though, $12 = 0b1100$ ($12 = 8 + 4 = 2^3 + 2^2$). We only need to calculate $5^8 \cdot 5^4$. Instead of looping over **`exponent`**, we could just loop over the bit representation of **`exponent`**. (Note that 5, 5^2, 5^4, and 5^8 all will be calculated inside the loop to get to the two needed factors.)

Use bit-shift or floor division by 2 to loop over the bits of **`exponent`**. Bit-test with bitwise and. Within the binary expansion loop, update and multiply variables as needed.

2. Use parameter checks to exit with an exception when **`exponent`** is not a non-negative integer. Negative exponents are beyond the scope of this assignment.
3. Test and debug your function. Provide test runs in form of a main file. Run some cases to show your function works. In particular, show the output for

```
M = 2**63-2**16-977
b = 2**33-1301
e = 2**17+2**14+2**8-7
```

For reference, use Python's **`pow(base, exponent, modulus)`** (which does exactly what you are to implement) to verify your numerical result.

Each of the steps 1 – 3 will be graded according to the following rubric for a total of 12 points.

SCORE	4	3	2	1	0
SKILL LEVEL	Response gives evidence of a complete understanding of the problem; is fully developed; is clearly communicated.	Response gives the evidence of a clear understanding of the problem but contains minor errors or is not fully communicated.	Response gives evidence of a reasonable approach but indicates gaps in conceptual understanding. Explanations are incomplete, vague, or muddled.	Response gives some evidence of problem understanding but contains major math or reasoning errors.	No response or response is completely incorrect or irrelevant.