**Recipe name:** multiply\_by\_term

**Inputs:**

*input\_poly,* the polynomial that will be multiplied

*coefficient*, the coefficient of the term that *input\_poly* will be multiplied by

*power*, the power of the term that *input\_poly* will be multiplied by

**Outputs**:

*product*, the result of the polynomial multiplication

**Steps**:

1. *product* ← the 0 polynomial
2. For each term, term, in *input\_poly*
   1. *new\_coeff* ⇐ the coefficient of *term* multiplied by *coefficient* in Z256
   2. *new\_degree* ⇐ the degree of *term* + *power*
   3. Add a new term with coefficient of *new\_coeff* and a degree of *new\_degree* to *product*
3. Return *product*

**Recipe Name:** add\_polynomial

**Inputs:**

*poly1*, the first polynomial to be added

*poly2,* the second polynomial to be added

**Outputs:**

*sum*, the result of polynomial addition

**Steps:**

1. *sum* 🡨 a deep copy of *poly1*
2. *poly2\_terms* 🡨 get\_terms(*poly2*)
3. for each *key*, *value* in *poly2\_terms*, do the following
   1. *sum* = add\_term(*sum*, *value*, *key*)
4. Return *sum*

**Recipe Name:** multiply\_by\_polynomial

**Inputs:**

*poly1*, the first polynomial to be multiplied

*poly2,* the second polynomial to be multiplied

**Outputs:**

*product*, the result of polynomial multiplication

**Steps:**

1. *product* 🡨 the 0 polynomial
2. *poly2\_terms* 🡨 get\_terms(*poly2*)
3. for each *key*, *value* in *poly2\_terms*, do the following
   1. *product* 🡨 add\_polynomial(*product*, multiply\_by\_term(*poly1*, *value*, *key*))
4. Return *product*

**Recipe Name:** remainder

**Inputs:**

*numerator*, the numerator

*denom,* the denominator

**Outputs:**

the remainder of polynomial division of *numerator* and *denom*

**Steps:**

1. *quotient* 🡨 the 0 polynomial
2. *dividend* 🡨 *numerator*
3. *divisor* 🡨 *denominator*
4. While get\_degree(*divisor*) is less than or equal to get\_degree(*dividend*),
   1. *dividend\_coeff* 🡨 What get\_degree(*dividend*) maps to in get\_terms(*dividend*)
   2. *divisor\_coeff* 🡨 What get\_degree(*divisor*) maps to in get\_terms(*divisor*)
   3. *factor* 🡨 divide\_terms(*dividend\_coeff*, get\_degree(*dividend*), *divisor\_coeff,* get\_degree(*divisor*))
   4. *quotient* 🡨 add\_polynomial(*quotient*, *factor*)
   5. *dividend* 🡨 add\_polynomial(*dividend*, multiply\_by\_polynomial(subtract\_polynomial(the 0 polynomial, *factor*), *divisor*)
5. Return *dividend*

**Discussion**

1. If you are given a message that you want to encode and a value of k, which indicates how many error correction bytes you need, is it possible to guarantee that you will not have any coefficients that are equal to zero in the remainder from dividing the message polynomial by the generator polynomial?  If there were coefficients that are equal to zero in the encoded data, would it be a problem?  Why or why not?

No, it is not possible to guarantee that there will be no 0 coefficients in the remainder because a message containing only 0s will have 0s in the remainder no matter the generator polynomial.

Having 0s in the data is not a problem because 0 is a valid number in Z256 just like any of the other possible coefficients.

1. We have discussed the importance of modularity and writing your recipes/code in such a way that you can reuse them. If you needed a Polynomial class to represent polynomials with regular, real-number coefficients (as opposed to coefficients that are elements of Z256), how could you minimally change the code you have already written in order to reuse it for this purpose?

All we need to do is change any Z256 operations to regular python mathematical operations.