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*
    a. *new_coeff* ⇐ the coefficient of *term* multiplied by *coefficient* in Z_{256}
    b. *new_degree* ⇐ the degree of *term* + *power*
    c. 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
    a. *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
   a. *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*),
   a. *dividend_coeff* ← What get_degree(*dividend*) maps to in get_terms(*dividend*)
   b. *divisor_coeff* ← What get_degree(*divisor*) maps to in get_terms(*divisor*)
   c. *factor* ← divide_terms(*dividend_coeff*, get_degree(*dividend*), *divisor_coeff*, get_degree(*divisor*))
   d. *quotient* ← add_polynomial(*quotient*, *factor*)
   e. *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 Z_{256} just like any of the other possible coefficients.

2. 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 Z_{256}), 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 Z_{256} operations to regular python mathematical operations.