Project 1: Univariate Polynomials with Integer Coefficients

Josh Horswill - s47227156 - GitHub Repo: MATH2504 P1

Introduction & General Information:

This notebook is an extension to the responses to each part of Project 1. All of these functions, implementations and testing parameters are organised on the GitHubb repo under their respective sections. All of the testing for parts 1 to 6 are done in the section below, with all of their functions under the test folder on the repo. All of the polynomial implementations were moved under PolynomialTypes and their basic operations pushed under the basic_polynomial_operations files.

For convenience, the following is a collection of the types and terms that were used throughout the project to answer the individual tasks:

- PolynomialDense: Dense Polynomial Interpretation
- PolynomialSparse: Sparse Polynomial Interpretation
- PolynomialSparseBInt: BigInt Sparse representation
- PolynomialModP: Modulo prime sparse representation
- Term: General Polynomial term, used for Dense, Sparse, ModP
- TermBI: BigInt edition of Term, used for SparseBInt
- CRT: Chinese Remainder Theorem

Test Samples:

This section implements all of the testing that was implemented throughout the project. This first simply loads the files and runs the original PolynomialDense testing.

In [3]:
```
include("poly_factorization_project.jl")
include("test/runtests.jl")
```

```
done
  Activating project at `~/Desktop/MATH2504/P1/MATH2504_Project1_s4722715`
done
test_euclid_ints - PASSED
test_ext_euclid_ints - PASSED
prod_test_poly - PASSED
prod_derivative_test_poly - PASSED
ext_euclid_test_poly - PASSED
division_test_poly - PASSED

doing prime = 5          ..........
doing prime = 17         ..........
doing prime = 19         ..........
factor_test_poly - PASSED
```

Given the amount of testing that was conducted throughout the project, all of the relevent

tests that were made for each section are run and shown below. Generally, each type was tested for the four basic operations and then extra tests were ran for ModP implementation, CRT and the Squared Power method.

In [4]:
```
include("test/Sparse-BI-ModP_test.jl")
```

```
Polynomial Division - PASSED
Squared method test - PASSED
Product type testing:
Polynomial Product - PASSED
Polynomial BI Product - PASSED
Polynomial ModP Differentiation - PASSED

Derivative type testing:
Polynomial Differentiation - PASSED
Polynomial BI Differentiation - PASSED
Polynomial ModP Differentiation - PASSED

Division type testing:
Polynomial Division - PASSED
Polynomial ModP Division - PASSED
Polynomial BI Division - PASSED

Euclidean type testing:
Polynomial Euclidean - PASSED
Polynomial ModP Euclidean - PASSED

PolynomialModP Generality & Prime Tests:
PolynomialModP Generality - Passed
PolynomialModP Generality - Passed

CRT product output test:
CRT Test - Passed

Squared Power method test:
Squared method test - PASSED
```

Question 1: Pretty Printing & Understanding

The aim of this task was to set up the repository and get used to working with all of the functions and structs that were predefined in the given code. Along with this, we were asked to create a second example script that incorportates "pretty printing", below is its output, showing much of the important functionality that the types use.

In [5]:
```
include("example_script_2.jl")
```

```
Using x_poly, we construst two polynomial functions:
   1) 5·x^3 + 8·x^2 -9·x^1
   2) 1·x^2 + 2·x^1 + 3

These functions are operable:
Multiplication p1 × p2 =
5·x^5 + 18·x^4 + 22·x^3 + 6·x^2 -27·x^1

Division (w.r.t prime 101) p1 ÷ p2 = 5·x^1 + 99

Addition p1 + p2 =
5·x^3 + 9·x^2 -7·x^1 + 3
```

```
Subtraction p1 - p2 =
5·x^3 + 7·x^2 -11·x^1 -3

The polynomial functions are also able to be derived via the derivative implem
entation:
Polynomial Function: 5·x^3 + 8·x^2 -9·x^1
→ Derivative: 15·x^2 + 16·x^1 -9

Where the derivative function can be considered as an operator:
Operation considered: p1(x) × p2'(x)
Result = 10·x^4 + 26·x^3 -2·x^2 -18·x^1

We can find the mod result of a function to a given prime:
Prime = 19
Function = 5·x^3 + 8·x^2 -9·x^1
function mod prime = 5·x^3 + 8·x^2 + 10·x^1

As an extension to modulo, we can find the greatest common denominator of two
polynomial functions:
Function 1: 5·x^3 + 8·x^2 -9·x^1 & Function 2: 1·x^2 + 2·x^1 + 3
Greatest Common Denominator: 13
```

```
  Activating project at `~/Desktop/MATH2504/P1/MATH2504_Project1_s4722715`
```

In essence, the Polynomial types are a foundation for operating computer algebra. They allow the construction of polynomials in their own right respectively, which can be used to output products, addition, division and other functions, including those reliant on prime residues.

Question 2: PolynomialSparse Implementation

This part sees the implementation of PolynomialSparse. This type could he been implemented in several different ways, using heaps and other various tools, however it was decided that a good implementation would be to pack the polynomials in a sorted vector of terms, which if done correctly (in which it was), allows for quick computation and easier indexing when neccessary. This struct takes in a vector of terms and packs them into a vector, sorting them from highest degree to lowest using a defined sorting function. The struct does not store 0 terms and is a realisation of the given polynomial at each point. The structure of a PolynomialSparse is given as follows:

In [ ]:

```
struct PolynomialSparse

    terms::Vector{Term}

    #Inner constructor of 0 polynomial
    PolynomialSparse() = new([zero(Term)])

    #Inner constructor of polynomial based on arbitrary list of terms
    function PolynomialSparse(vt::Vector{Term})

        # Establish vt vector of 0s
        vt = filter((t)->!iszero(t), vt)
        if isempty(vt)
            vt = [zero(Term)]
        end

        # Take count of elements in vector of terms
        cnt = 0
```

```
        for t in vt
            cnt += 1
        end

        terms = [zero(Term) for i in 0:cnt-1] #Array of zeroes to the size of

        # For each term, if not a null item, commit to terms array and icreme
        cnter = 0
        inner_ct = 0
        for t in vt
            inner_ct += 1
            if t.coeff !=0 && t.degree != 0
                cnter += 1
                terms[cnter] = t #+1 accounts for 1-indexing
            elseif t.coeff != 0 && t.degree == 0
                cnter += 1
                terms[cnter] = t
            end
        end

        # Sort for degrees
        bubble_poly_sort!(terms)

        return new(terms)
    end
end
```

Changing the struct style lead to refactoring a variety of the functions that were provided with Polynomial Dense. Some functions only required changing over the input types to PolynomialSparse while others required more rigorous work. To lay out the formulation of each function and type this repository was organised into various directories associated to their usage. Polynomial structures are under PolynomialTypes, Term structures are under TermTypes with testing and operations under their own directories. These changes that you see below and for the following different types of Polynomials are all located in their respective files and directories, any changes to an addition function would be located in the addition basic_polynomial_operations funder in the addition file and so on for other operations.

For PolynomialSparse, the major changes that occured were to addtion, subtraction and division functionality, since no longer both polynomials would not be length of their degree, only now to the number of terms they have. The code below shows the changes to these functions, which are located in their relative files. While the heap structure could of saved some effort on implementation, using the organised vector method allows controllable and effective indexing when needed, making operations easier to implement and fairly efficient.

In [18]:

```
### ADDITION NEW IMPLEMENTATION
# Since we no longer have degree length polynomials, we have to check at each
# and then perform the + operation, pushing anything that is not in the polyn

function +(p::PolynomialSparse, t::Term)
    # Copy and count
    p = deepcopy(p)
    cnt = 0

    # If degrees match : addition of coefficients, else : continue
    for terms in p
        cnt += 1
```

```
        if terms.degree == t.degree
            p.terms[cnt] += t
        else
            p.terms[cnt] = terms
        end
    end

    # Any term with degree not in the original polynomial is pushed in
    if t.degree ∉ degrees(p)
        push!(p, t)
    end
    return p
end


function +(p1::PolynomialSparse, p2::PolynomialSparse)::PolynomialSparse
    # Polynomial realisation of the term function
    p = deepcopy(p1)
    for t in p2
        p += t
    end
    return p
end

### DIVISION CHANGES
# Due to the way the structure operated, after each iteration of h under the
# it was neccessary to repack f and q so avoid the occasional pushing of a 0
# Not an ideal fix however it was sufficient for this bug.

function divide(num::PolynomialSparse, den::PolynomialSparse)
    function division_function(p::Int)
        f, g = mod(num,p), mod(den,p)
        degree(f) < degree(num) && return nothing
        iszero(g) && throw(DivideError())
        q = Term(0,0)
        prev_degree = degree(f)
        while degree(f) ≥ degree(g)
            h = PolynomialSparse( (leading(f) ÷ leading(g))(p) )  #syzergy
            f = mod((f - h*g), p)
            fre = PolynomialSparse([i for i in f.terms])
            f = fre
            q = mod((q + h), p)
            qre = PolynomialSparse([i for i in q.terms])
            q = qre
            prev_degree == degree(f) && break
            prev_degree = degree(f)
        end
        @assert iszero( mod((num  - (q*g + f)),p))
        return q, f
    end
    return division_function
end


### SUBTRACTION NEW IMPLEMENTATION
# A new function was made for subtraction that was a bit more ellaborate than
# version. The terms of the polynomial and the subtraction term needed to be
# then pushed if the degree was not in the polynomial.
function -(p::PolynomialSparse, t::Term)::PolynomialSparse
    # Copy and Count
    pp = deepcopy(p)
    cnt = 0

    # If degrees match : subtract, else: continue
```

```
    for terms in pp
        cnt += 1
        if terms.degree == t.degree
            pp.terms[cnt] += (-t)
        else
            pp.terms[cnt] = terms
        end
    end

    # Push non-associated terms
    if t.degree ∉ degrees(p)
        push!(pp, (-t))
    end
    return pp
end


### EXTENDED_EUCLID_ALG CHANGES FOR POLYNOMIALSPARSE
# Similar to division, to avoid the pushing of a 0 term bug, the j, k and l v
# Not ideally efficient however a sutiabble fix to the problem, the assertion

function extended_euclid_alg(a::PolynomialSparse, b::PolynomialSparse, prime:
    # Initialisation
    old_j, j = mod(a, prime), mod(b, prime)
    old_k, k = one(PolynomialSparse), zero(PolynomialSparse)
    old_l, l = zero(PolynomialSparse), one(PolynomialSparse)

    # While mod condition isn't 0, re-define conditions and repack
    while !iszero(mod(j, prime))
        p = first(divide(old_j, j)(prime))
        old_j, j = j, mod(old_j - p*j, prime)
        jre = PolynomialSparse([i for i in j.terms])
        j = jre
        old_k, k = k, mod(old_k - p*k, prime)
        kre = PolynomialSparse([i for i in k.terms])
        k = kre
        old_l, l = l, mod(old_l - p*l, prime)
        lre = PolynomialSparse([i for i in l.terms])
        l = lre
    end
    g, s, t = old_j, old_k, old_l

    # Euclidean Assertion
    @assert mod(s*a + t*b - g, prime) == 0
    return g, s, t
end
```

Out[18]:
```
extended_euclid_alg (generic function with 4 methods)
```

PolynomialSparse as described, only packs in the Terms that are existing in the given polynomial, meaning that there is no spacing with 0 terms like in PolynomialDense. While this seems like a generally, more intuitive way to deal with Polynomials in computer algebra, there are many pros and cons when compared to the standard PolynomialDense version.

In terms of the four basic operations (addition, subtraction, multiplication and division), the sparse implementation outperforms its original implementation in most across the board with respect to time, allocations and size. With addition and subtraction, on average we see a time decrease of 20-30% with a decrease in allocations and output size of nearly a half. For multiplication and division, the increase in computation efficiency was less exhaggerated however still quite prominant. For multiplication of random polynomials of these two types,

we saw an average computation time decrease of 20-25% with a decrease in allocations and size of ~30%. Division was quite similar, however due to the repacking technique that was used, efficiency was slightly behind multiplication's improvemement, with a decrease in runtime of around 20% and storage reduction of 20%

Notably, while polynomial on polynomial operations were effectively improved, polynomial and integer operations suffered slightly, with increases in allocations and size for PolynomialSparse operations with integers when compared to PolynomialDense and integer operations. This mainly effected the exponent function ^ and multiplication between polynomials and integer factors however, this was not a strong enough difference to exclude the idea that PolynomialSparse was a more effecient method.

Overall, the four basic operations when using polynomial on polynomial operations were more efficient than its predeccessor in PolynomialDense, although did lose some efficiency for polynomial on integer cases. For non-operating functions (leading, zeros, first, etc.) the functionality remained the same and allocations were on average better in the PolynomialSparse case. The new type improved across most aspects, only noticeably yielding no improvement in these (polynomial, int) cases due to repeated complexity that the original struct does not implement.

Question 3: BigInt Implementation

In this section we look at the implementation of having a BigInt coefficient for terms instead of standard int (Int64). Much of the BigInt implementation was the same as PolynomialSparse, however now using a BigInt for polynomial coefficients. A way to implement this that seemed suitable was to create a new term, called TermBI, that operates the same way to Term, just taking in a BigInt for the polynomial coefficients, then have PolynomialSparse work for TermBI, thus creating PolynomialSparseBInt. Below is the structure of TermBI, which is located under TermTypes:

In [ ]:
```
### TERMBI
# Takes in BigInt coefficient and standard Int degree.
struct TermBI
    coeff::BigInt
    degree::Int
    function TermBI(coeff::BigInt, degree::Int)
        degree < 0 && error("Degree must be non-negative")
        coeff != 0 ? new(coeff,degree) : new(coeff,0)
    end
end
```

This was then used as the input term for the new PolynomialSparse iteration, which was aptly called PolynomialSparseBInt:

In [ ]:
```
struct PolynomialSparseBInt

    terms::Vector{TermBI}

    #Inner constructor of 0 (TermBI) polynomial
    PolynomialSparseBInt() = new([zero(TermBI)])
```

```
    #Inner constructor of polynomial based on arbitrary list of BigInt terms
    function PolynomialSparseBInt(vt::Vector{TermBI})

        # Vector of terms filled with a 0 Term Big Int
        vt = filter((t)->!iszero(t), vt)
        if isempty(vt)
            vt = [zero(TermBI)]
        end

        # Take count of elements in vector of terms
        cnt = 0
        for t in vt
            cnt += 1
        end

        terms = [zero(TermBI) for i in 0:cnt-1] #Array of zeroes to the size

        # For each term, if not a null item, commit to terms array and increm
        cnter = 0
        for t in vt
            if t.coeff !=0 && t.degree != 0
                cnter += 1
                terms[cnter] = t #+1 accounts for 1-indexing
            elseif t.coeff != 0 && t.degree == 0
                cnter += 1
                terms[cnter] = t
            end
        end

        # Sort from highest degree term to lowest
        bubble_poly_sort!(terms)
        return new(terms)
    end
end
```

Much of the operations that were defined for PolynomialSparse were the same for PolynomialSparseBInt, just with now respecting the coefficients are BigInt operators and use TermBI instead of Term. To construct a PolynomialSparseBInt, the kernel needs to know that the input is "big" and as an example, below compares the implementation of the same Polynomial using the two different struct types.

In [7]:

```
poly_sparse = PolynomialSparse([Term(1,2), Term(3,4)])
poly_bint = PolynomialSparseBInt([TermBI(big"1",2), TermBI(big"3",4)])
println("Sparse interpretation: $poly_sparse")
println("BigInt interpretation: $poly_bint")
```

```
Sparse interpretation: 3·x^4 + 1·x^2
BigInt interpretation: 3·x^4 + 1·x^2
```

The advantage, and reason for implememtation, of the BigInt polynomial type is its response to overflow. Using PolynomialSparse, we are limited to +-2^64 integers, which is a fairly practical range, but for larger computations induces overflow. Using BigInt, overflow does not occur however this benefit comes at the price of more expensive computation. Below shows examples of overflow for PolynomialSparse and how PolynomialSparseBInt handles this, while then showing comparative run times for various operations and compiling.

In [10]:

```
include("Testing/Comparison_Sparse_BI.jl");
```

```
Range limits for Int64:
```

```
    -> -9,223,372,036,854,775,808
    ->  9,223,372,036,854,775,807

Maximum Term coeff: 9223372036854775807·x^1

Overflow Example: Adding 1x^1 to the maximum representation:
    -> -9223372036854775808·x^1

If we use BigInt, we can represent the same term:
     -> 9223372036854775808·x^1

We can go even further and represent a much larger range of Integers:
    -> 92233720392233720368547758076854775807·x^1

And perform arithmatic:
    -> 200 * Term = 18446744078446744073709551615370955161400·x^2

################ TIMING ################
Comparision of overflow times

Addition of 9223372036854775807·x^2 and 25·x^2
PolynomialSparse:
  0.000016 seconds (12 allocations: 1.047 KiB)
PolynomialSparseBInt:
  0.000029 seconds (26 allocations: 1.453 KiB)

Product of 9223372036854775807·x^2 and 25·x^2
PolynomialSparse:
  0.000016 seconds (18 allocations: 1.562 KiB)
PolynomialSparseBInt:
  0.000032 seconds (33 allocations: 1.969 KiB)

Comparision of compiling 10x^1 + 20x^2 + 100x^10:
Polynomial
  0.000006 seconds (3 allocations: 464 bytes)
PolynomialSparse
  0.000004 seconds (4 allocations: 416 bytes)
PolynomialSparseBInt
  0.000008 seconds (4 allocations: 416 bytes)

Operation Timing: product of two random polynomials
Polynomial
  0.000059 seconds (247 allocations: 25.031 KiB)
PolynomialSparse
  0.000064 seconds (142 allocations: 14.219 KiB)
PolynomialSparseBInt
  0.001017 seconds (4.15 k allocations: 178.039 KiB)
```

As we can see, the new type handles overflow appropriately and data is not corrupted by exceeding range limits. However, as a result of this convenience we increase operation cost, with a noticeable increase to runtime, allocations and output size. From these examples we can conclude that PolynomialSparse is in general the faster method across the board, although results suffer from overflow when operating around the limits of the Int64 ranges, wherease the BigInt implementation handles this issue accordingly.

Question 4: PolynomialModP Implementation

A current hindurance to mod operations is that a prime needs to be provided for the function to return an answer. Currently, the division function returns a divisor function that relies on a prime to return the analytic answer. Wouldn't it be great if we had a polynomial type that

came with its own prime? Well thats what PolynomialModP is all about! As asked, this Polynomial type works as a conduit between PolynomialSparse and modulo operations by taking in a polynomial and a prime. The point of this is to simplfy the modulo operations so that they happen automatically instead of being prompted by the input of a prime number.

The polynomial stays stored as a polynomial however when operated on, showed or computed in anyway, is used as its Mod(Polynomial, P) form, where P is the given prime number. A key assumption that was made here however is that for the four major operations and for any of that matter that involve two PolynomialModP types, their primes need to be equal, otherwise more assumptions would have to be made over which prime to use which would yield incorrect outcomes down the line.

The implementation of this type was quite simple, all of the functionality that did not use modulo was nearly identicle to the PolynomialSparse implementation seeing that the polynomial was of this type. For the modulo cases (divide, gcd, euclid, etc) an assertion was made at the beginning of each function so that we could enforce the same prime number between polynomial pairings. The structure of this type is shown below.

In [ ]:
```
struct PolynomialModP

    # Take in prime number & PolynomialSparse
    p::PolynomialSparse
    prime::Int

    # The zero term just 101 as its prime
    PolynomialModP() = new(PolynomialSparse(zero(Term)), 101)


    function PolynomialModP(vt::PolynomialSparse, prime::Int)

        # If vector is empty, zero
        if isempty(vt)
            vt = [zero(Term)]
        end

        # Take count of elements in vector of terms
        cnt = 0
        for t in vt
            cnt += 1
        end

        terms = [zero(Term) for i in 0:cnt-1] #Array of zeroes to the size of

        # Count through and set terms
        cnter = 0
        inner_ct = 0
        for t in vt
            inner_ct += 1
            if t.coeff !=0 && t.degree != 0
                cnter += 1
                terms[cnter] = t
            end
        end

        # Sort Polynomial and set
        bubble_poly_sort!(terms)
        modp_sorted = PolynomialSparse(terms)
```

```
        # Return PolynomialModP
        return new(modp_sorted, prime)
    end
end
```

Given that the input polynomial p is a PolynomialSparse, the functionality translates across quite easily for the various operations and functions. The only difference is the declaration of the polynomial and prime at the beginning of each function call so that the operations can be conducted. A good example of this is in the multiplication function below, where it illustrates how the polynomials are extracted, made product, then returned modulo their asserted prime.

In [ ]:
```
function *(p1::PolynomialModP, p2::PolynomialModP)::PolynomialModP
    @assert p1.prime == p2.prime
    p_out = PolynomialSparse()
    for t in p1.p
        new_summand = (t * p2.p)
        p_out += new_summand
    end
    p_out = mod(p_out, p1.prime)
    return PolynomialModP(p_out, p1.prime)
end
```

This case applies to all other polynomial on polynomial operations and was fairly straightforward to replicate across. Given that we now output the results of operations modulo prime, the pow_mod and ^ functions effecitvely become the same function and we can just use pow_mod to now compute polynomial exponents, where we update the equivalent functions by:

In [ ]:
```
function pow_mod(p::PolynomialModP, n::Int)
    n < 0 && error("No negative power")
    out = one(p.p)
    for _ in 1:n
        out *= p.p
        out = mod(out, p.prime)
    end
    return out
end

function ^(p::PolynomialModP, n::Int)
    poly = p.p
    polysq = ^(poly, n)
    polysq = mod(polysq, p.prime)
    return PolynomialModP(polysq, p.prime)
end
```

And we can update the show function to show the polynomial saves to the type in its modulo form:

In [24]:
```
include("src/PolynomialTypes/PolynomialModP.jl")
x = rand(PolynomialModP)
println("Polynomial in its regular form:")
println(x.p)

println()
println("PolynomialModP form:")
println(x)
```

```
Polynomial in its regular form:
55·x^7 + 68·x^6 + 32·x^5 + 3·x^3 + 97·x^1

PolynomialModP form:
(  + 3·x^6 + 2·x^5 + 3·x^3 + 2x ) mod 5
```

In terms of testing, PolynomialModP builds off the back of PolynomialSparse and testing its derivative and product qualities was fairly straight forward. To then confirm that the type does as it was intended, the divide and euclidean testing was restructued slightly to ensure that the prime functions run through to completion and don't require a prime to prompt. For these sorts of functions, take the example below in defining division, the process is quite simple: the polynomial and prime are seperated, the PolynomialsSparse is passed to its designated function and its associated prime is then passed to its designated function, enforcing division and prime operations over one step.

In [ ]:

```
function divide(num::PolynomialModP, den::PolynomialModP)
    # Prime assertion
    @assert num.prime == den.prime
    if degree(den.p) > degree(num.p)
        return 0
    end

    # Polynomial & Prime Allocation
    num_val = num.p
    den_val = den.p
    p = num.prime

    # Process division for two polynomials
    function division_function(p::Int)
        f, g = mod(num_val,p), mod(den_val,p)
        degree(f) < degree(num_val) && return nothing
        iszero(g) && throw(DivideError())
        q = Term(0,0)
        prev_degree = degree(f)
        while degree(f) ≥ degree(g)
            h = PolynomialSparse( (leading(f) ÷ leading(g))(p) )  #syzergy
            f = mod((f - h*g), p)
            fre = PolynomialSparse([i for i in f.terms])
            f = fre
            q = mod((q + h), p)
            qre = PolynomialSparse([i for i in q.terms])
            q = qre
            prev_degree == degree(f) && break
            prev_degree = degree(f)
        end

        # Polynomial Assertion
        @assert iszero( mod((num_val  - (q*g + f)),p))
        return q, f
    end

    # Process returned function with the associated prime
    return division_function(p)
end
```

This implementation of taking the polynomial and the prime seperately is used across most of the functionality in PolynomialModP. As stated, its similarity to PolynomialSparse made lower level functionality quite simple to implement, the only main differences are in the

operations involving prime inputs. All testing done for PolynomialModP was done in the testing file and the results are shown at the top of this notebook.

Question 5: Chinese Remainder Theorem

One of the drawbacks from using the BigInt polynomial type was that computation for basic operations became more expensive, especially with computing products. An alternative to direct multiplication, which has been used throughout each of the polynomial types thus far, is to use the Chinese Remainder Theorem to reconstruct the product from two polynomials, minimising the amount of BigInt operations. The CRT involves returning a unique solution to a case of simultaneously linear congruences given a coprime moduli, which can be expanded to solving the coefficients of polynomials. The way the CRT was implemented for BigInts involved three stages of functionality which are shown in order below. Firstly, we begin with an initial function that computes the CRT for two vectors of integers. To induce simplicity, the input vectors are kept at length of 2 for all functions so that iteration and stepping through for the polynomial implementation becomes analogous.

In [ ]:
```
function iCRT_BI(um::Vector{BigInt}, p::Vector{BigInt})
    n = length(um)
    v = Vector{BigInt}(undef,n)

    v[1] = um[1]
    v[2] = (um[2] - v[1]) * Base.invmod(p[1], p[2]) % p[2]
    u = v[1] + v[2]*p[1]

    return u
end
```

By then taking this "internal" CRT function, we can use it to compute for two BigInt Polynomials:

In [ ]:
```
function my_getindex(idx::Int64, v1::PolynomialSparseBInt)::Int64
    v11 = degrees(v1)
    xarr = findall(x->x==idx, v11)
    return xarr[1]
end

function CRT_polyBI(p1::PolynomialSparseBInt, p2::PolynomialSparseBInt, n::Bi
    ak, bk = 0, 0
    c = PolynomialSparseBInt(TermBI(big"0",0))
    lenmax = max(degree(p1), degree(p2))
    for k in 1:lenmax
        if k ∉ degrees(p1)
            ak = 0
        else
            spot = my_getindex(k, p1)
            ak = coeffs(p1)[spot]
        end

        if k ∉ degrees(p2)
            bk = 0
        else
            spot = my_getindex(k, p2)
            bk = coeffs(p2)[spot]
        end
```

```
        ck = iCRT_BI([BigInt(ak), BigInt(bk)], [n,m])
        c = c + TermBI(BigInt(ck), k)
    end
    return c
end
```

The Polynomial CRT function takes the largest degree of the two input polynomials, iterates from 1 to that value and assigns the coefficient values at that degree index to either ak or bk. These are then used to compute the CRT coefficient of ck which latter becomes a PolynomialSparseBInt of its own. A difficulty that this function faces was the alignement of degrees, which was aided by the structure of the polynomial types.

Had we used the PolynomialDense type, matching the degrees would not have had been required and a less checking would have been needed. Fortunately though, as mentioned earlier in the explanation of the PolynomialSparse structure, the choice of using a strictly sorted vector of terms over a heap allowed indexing to be effective when it is needed. In this case, by finding the index that the degree occurs in a polynomial, which is found by the use of my_getindex(), we can associate the appropriate coefficent quite easily, or let 0 if the degree does not occur. With that function now in place, the final product computation function can be built from the following.

In [ ]:

```
function CRT_prodBI(p1::PolynomialSparseBInt, p2::PolynomialSparseBInt)
    height_a = maximum(coeffs(p1))
    height_b = maximum(coeffs(p2))

    bound = 2 * height_a * height_b * min(degree(p1)+1, degree(p2)+1)
    pri = big"3"
    M = pri

    C = mod(p1, BigInt(M))*mod(p2, BigInt(M))

    while M < bound
        pri = nextprime(pri+1)
        Cdash = mod(p1, BigInt(pri))*mod(p2, BigInt(pri))
        C = CRT_polyBI(C, Cdash, M, pri)
        M = M*pri
    end

    return C
end
```

By declaring a variety of bounds and initial conditions, we can iterate up the tree of primes and recursively compute the CRT_polyBI function and the result that is let to be C becomes the product between the two BigInt Polynomials. At a glance, it does not appear that the CRT method makes a significant difference however, there are cases where the implementation outperforms the original product method. The standard method of polynomial product computation is very effective for short length polynomials, even for larger coefficients however, as the number of thterms in the polynomials increases, we see that the CRT begins to be the more effective method. The standard method grows fast in complexity especially for 10,000+ term polynomials, and the time and allocation subsequently increases, while the CRT method remains consistent with processing through the primes and the end result is achieved quicker and with less expense for these larger cases.

The potential of this feature of CRT was not able to be tested fully due to resource limitations, computing the product of two polynomials of 10,000 terms length can take quite a long time! However, for computing the product of several hundred term length polynomials begins to close the gap in time between the standard method and CRT. Therefore, we can say that the CRT method demonstrates faster ability in caclulating the product for larger range polynomials, where the length of their terms span over 10,000 and for any range less than this, the original implementation has more merit in returning a quicker and less expensive result.

For testing, a suitable measure of whether the CRT outputs a correct product is to compare to the original product function. Given PolynomialSparseBInt has already passed its own testing parameters, CRT would still hold. So, we set up a testing funciton that generates two random BigInt polynomials and compares their CRT and product output, passing if all occurances are matching.

For "timing" the two operations, as mentioned there was some difficulty in processing products for larger length Polynomials due to hardware inability. The examples below show that the CRT is much slower for shorter cases bit begins to close the gap in timing as the polynomial lengths increase. I would have liked to have shown larger size polynomial products using both methods however the output through the notebook struggled greatly compared to the VSCode environemnt. Below is an example of a shorter length test, which shows us that typically the CRT method is slower for this case of product and though I was not able to clearly demonstrate it, as stated the CRT begins to gain effciciency for far larger terms.

In [8]:
```
p1 = rand(PolynomialSparseBInt)
p2 = rand(PolynomialSparseBInt)

using Primes
println("Short length polynomial product comparison")
println("Standard:")
@time p1*p2
println("CRT:")
@time CRT_prodBI(p1, p2);
```

```
Short length polynomial product comparison
Standard:
  0.000161 seconds (505 allocations: 23.617 KiB)
CRT:
  0.000679 seconds (2.83 k allocations: 133.797 KiB)
```

Question 6: Power & Factoring

As a way to improve the efficiency of computing powers of Polynomials, we can implement repeated squares method which works by taking a polynomial and a power term n, converting n into an 8Bit binary value (only did 8Bit, works for 16 to 64 but for simplicity 8 was sufficient), then instead of multiplying the polynomial by itself n times, we multiply it by itself to the power of either 1, 2, 4, ..., 128 depending on whether it is associated. The code to implement this for PolynomialBInt is shown below.

In [ ]:
```
function repsqr_exp(p::PolynomialSparseBInt, n::Int)
```

```
    @assert n < 2^8
    incr = [1, 2, 4, 8, 16, 32, 64, 128]
    bin_vals = digits(n, base=2, pad=8)
    return_val = TermBI(big"1",0)
    for i in 1:8
        if bin_vals[i] != 0
            return_val *= ^(p, incr[i])
        else
        end
    end
    return return_val
end
```

This method reduces computation time significanty since far less allocations and term produts need to be made to find an output. The larger the polynomial and the larger the exponent, the more efficient in comparison it becomes, which has been indicative of the processes we have seen in the previous questions. As for factoring however, the implementation was quite diffucult. While the factoring worked, a big pitfall to the function and its daughter functions is that computation time of the process was quite slow. For small polynomials, even for large prime numbers, there was not much of a notacibble drop in speed compared to PolynomialDense, although as the length of the polynomial increase, especially for the degree, the result began to take significantly more time, arguably exponentially more time. Below is the implementation of factor for PolynomialSparse, the other types are in the factor file with their sub-functions assigned also.

In [47]:

```
function factor(f::PolynomialSparse, prime::Int)::Vector{Tuple{PolynomialSpar
    #Cantor Zassenhaus factorization

    ### Initialisation and Packing
    f_modp = mod(f, prime)
    degree(f_modp) ≤ 1 && return [(f_modp,1)]

    ff = prim_part(f_modp)(prime)
    fff = PolynomialSparse([i for i in derivative(ff).terms])

    squares_poly = gcd(f, fff, prime)
    squares_poly = PolynomialSparse([i for i in squares_poly.terms])
    ff = (ff ÷ squares_poly)(prime)

    # make f monic
    old_coeff = leading(ff).coeff
    ff = (ff ÷ old_coeff)(prime)

    # Factor and set return value
    dds = dd_factor(ff, prime)
    ret_val = Tuple{PolynomialSparse,Int}[]

    # Iterate factor and push into storage variable
    for (k,dd) in enumerate(dds)
        sp = dd_split(dd, k, prime)
        sp = map((p)->(p ÷ leading(p).coeff)(prime),sp) #makes the polynomial
        for mp in sp
            push!(ret_val, (mp, multiplicity(f_modp,mp,prime)) )
        end
    end

    #Append the leading coefficient and return
    push!(ret_val, (leading(f_modp).coeff* one(PolynomialSparse), 1) )
```

```
    return ret_val
end
```

Out[47]: factor (generic function with 2 methods)

Most of the smaller functionality inside the factor function was easy to translate across for eachtime, very little change was required for the outputs to work. Given that the basic operations were working for each type that was constructed, each factorisation type generally worked quite well, occassionally throwing an error for having an insufficiently small prime number or a bug. With benchmarking however, stress testing the functions resulted in StackOverflowErrors, Julia continuously evaluating and other general errors. It appeared that the implementation, though effective for short length Polynomials, struggled for anything that began to exceed degrees of 8 and above. This could be due to a variety of factors, however with the limited time I had left to mitigate the issue, my best guess was that the main enumeration inside factor function began to increase in complexity and start to push corrupted terms. Unfortunately, due to these errors I was not able to perform sufficient benchmarking to show stress testing. Subsequently, I did not attempt the bonus question for factorizing with PolynomialSparseBInt.

Bonus Task: Perspective Summary

Dr. Mehta's relationship to STEM and Maths in general came from a bit of inheritance from her parents' careers and her own interest and passion. She realised from a young age (stating when she was as young as three) that she had a deep interest and engagement in probability, algebra and calculus. That naturally lead her to persue mathematics at university, eventually becoming a PhD graduate in mathematical phyics. Software in general didn't become a major aspect of her career until her first major role at Opcom. She currently works at You. Smart. Thing which is a personal journey planning software that is growing rapidly. She currently works in the engineering department and as an innovation/project lead which she's held for nearly 18 months since moving on from several distinguished firms. Initially Dr. Mehta started at Opcom as a software tester and transitioned to algorithm building. She was one of the first to begin commerically using C++ & C# and when she moved on to different companies (Deswik, Carmen, Jeppesen, SilverRail) she transitioned to using Java, Python, Perl and many more. Maithili's experience on the whole has seemed to encompass quite a lot of different aspects of software, with a focus more on data analytics, to which she has been able to apply a lot of her background in mathematics along with gaining new tools from the languages she has learned.

What interest me the most about Dr. Mehta's journey is how much programming she had done for a "mathematics" graduate. I was able to resonate with some of her points in that transition from language to language becomes easier when you learn how to implement algorithms correctly first. For me the steepest part of the coding learning curves ive faced (C, Python, Julia, Java) has come from lacking algorithm skills, but as ive used Julia for longer now I can see with her point that as time goes on, the hardest thing about transitioning languages ends up remembering to use the right base function (printf vs println). While I have been familiar with ML, data analytics and statistical methods, it was really interesting to find out the "value" of data on a whole, with regards to what consumers want and how its regulated. I was not aware of how strict/important data regulation is nor did

I know really what is valued at the moment i.e. cost algorithms. A final point I found interesting was Agile SCRUM and the modern culture of software engineering with sprints and other methodologies. Personally, I aim to go into Data Analytics after finisihng my degree and this talk has been quite helpful. A major point that I have taken away from Dr. Mehta is that data is valued quite differently and its value changes with contemporary demand, being able to pick out what is wanted vs what is needed will keep me at an advtange. Along with that point, I did like the different methodologies of thinking and how to tackle problems in DA and ML. Overall, I learned quite a lot from the talk and was especially interested in the data and software culture that Maithili covered, it was a good perspective into the next 12 months that await.