Contributing to SageMath

Minsun Kim
soon@haari.me
Korea University
Republic of Korea

ABSTRACT

This report presents my contributions to SageMath, a large-scale open-source mathematics software system. The project involved identifying areas for improvement within the mathematical infrastructure of SageMath and proposing changes to enhance usability, clarity, and functionality. Throughout the process, I engaged directly with the SageMath codebase, examined existing implementations, and submitted feedback and suggestions via the project's development channels. My work focused on Polynomial Rings, Linear Algebra, and more SageMath Wrapper gadgets aimed to support ongoing efforts to make SageMath more intuitive and accessible to its users. This report documents the contributions made, the motivations behind them, and the insights gained through active participation in an open-source scientific software community.

1 INTRODUCTION

SageMath (or simply Sage) is a powerful open-source mathematics software system that aims to provide a comprehensive and free alternative to commercial systems like Mathematica, Maple, and MATLAB. Built on top of Python and incorporating numerous open-source libraries such as Maxima, NumPy, SciPy, GAP, and FLINT, SageMath offers a unified interface for a wide range of mathematical computations—including algebra, calculus, number theory, linear algebra, and cryptography. It stands for System for Algebra and Geometry Experimentation.

As an open-source project, SageMath is collaboratively developed by a global community of researchers, educators, and contributors. Its development is openly managed on platforms like GitHub, allowing anyone to inspect, modify, and contribute to the codebase. This community-driven model not only promotes transparency and reproducibility in research but also encourages contributions that directly impact the software's capabilities and user experience.

In this project, I contributed to three distinct areas of SageMath's ecosystem:

- Polynomial Rings' quotient inversion algorithm, followed by polynomials' GCD and Extended GCD algorithm.
- Efficiently iterating all possible solutions of a linear system using matrices.
- Preparser's interface handling large decimal integers.

As an avid CTF(Capture The Flag) player focused on cryptography challenges, I use SageMath a lot, thus these contributions all had initiated from an inconvenience while solving challenges. These contributions were aimed at improving both the internal consistency of SageMath and the overall experience for its users. This report documents the work done, the motivation behind each change, and reflections on engaging with a complex open-source mathematical software system.

2 BACKGROUND

2.1 Brief information about SageMath

Figure 1: The official logo of SageMath.

As introduced in Introduction section, SageMath is a comprehensive free alternative to commercial systems such as Mathematica, Maple, Magma, and MATLAB. The fact the system is built on top of Python allows the user a lot of flexibility to import other projects or sources.

There exists several ways to use SageMath:

- Notebook graphical interface: The command `sage -n jupyter` allows the user to use jupyter.
- Interactive command line: Simply run `sage` to run an interactive shell.
- Programs: By writing interpreted and compiled programs in Sage
- Scripts: By writing stand-alone Python scripts that use the Sage library, note that the user is allowed to use both general Python scripts and SageMath scripts with slightly different rules.

The following is the long-term goals of SageMath:

- Useful: Sage's intended audience is mathematics students (from high school to graduate school), teachers, and research mathematicians. The aim is to provide software that can be used to explore and experiment with mathematical constructions in algebra, geometry, number theory, calculus, numerical computation, etc. Sage helps make it easier to interactively experiment with mathematical objects.
- Efficient: Be fast. Sage uses highly-optimized mature software like GMP, PARI, GAP, and NTL, and so is very fast at certain operations.
- Free and open source: The source code must be freely available and readable, so users can understand what the system is really doing and more easily extend it. Just as mathematicians gain a deeper understanding of a theorem by carefully reading or at least skimming the proof, people who do computations should be able to understand how the calculations work by reading documented source code. If you use Sage to do computations in a paper you publish, you can rest assured that your readers will always have free access to Sage and all its source code, and you are even allowed to archive and re-distribute the version of Sage you used.
- Easy to compile: Sage should be easy to compile from source for Linux, OS X and Windows users. This provides more flexibility for users to modify the system.
- Cooperation: Provide robust interfaces to most other computer algebra systems, including PARI, GAP, Singular, Maxima, KASH, Magma, Maple, and Mathematica. Sage is meant to unify and extend existing math software.
- Well documented: Tutorial, programming guide, reference manual, and how-to, with numerous examples and discussion of background mathematics.
- Extensible: Be able to define new data types or derive from built-in types, and use code written in a range of languages.
- User friendly: It should be easy to understand what functionality is provided for a given object and to view documentation and source code. Also attain a high level of user support.

2.2 CTF Usage of SageMath

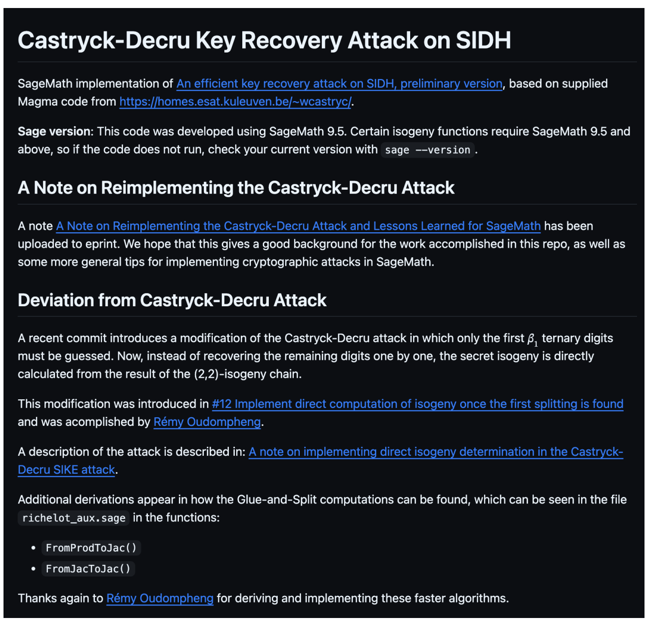
Python's default integer type `<class 'int'>` allows integers with unlimited size, unlike other languages like C/C++/Rust which need to import another implementation package such as GMP in order to use big integers. Python's default integer type is implemented with Pylong which is written in C behind Python wrapper.

SageMath, built on top of Python also include `<class 'int'>` of course, but it implements another basic integer type `<class 'sage.rings.integer.Integer'>`. It is implemented with GMP, and it handles big integers' operation with better complexity in some cases.

Since CTF(Capture The Flag) includes the category "Cryptography", it's unavoidable to use big integers outside the range of `uint64, int64` or even `uint128, int128`, which is why Python is preferred as a tool for solving Cryptography challenges.

Various Cryptography challenges are also related to Discrete Algebraical terms such as Rings and Fields, and Linear Algebraical terms such as Matrices and Vectors. SageMath allows the users to use them with ease, and defined well on many cases. For example, one can use matrices and vectors on Integer Ring, but on Integer Mod Rings and Finite Fields as well. The most basic yet very useful functionality of SageMath is solving linear equations, which is very helpful on terms of CTF challenges, even outside the category Cryptography.

It also implements relatively complex algorithms such as LLL(Lenstra-Lenstra-Lovasz) Algorithm, which is very helpful on attacking cryptographic implementations. For these reasons, SageMath is also actively being used for writing PoC(Proof of Concept) of cryptographic vulnerabilities. Figure 2 shows the Castryck-Decru attack's PoC written in SageMath.



Castryck-Decru Key Recovery Attack on SIDH

SageMath implementation of An efficient key recovery attack on SIDH, preliminary version, based on supplied Magma code from https://homes.esat.kuleuven.be/~wcastryc/.

Sage version: This code was developed using SageMath 9.5. Certain isogeny functions require SageMath 9.5 and above, so if the code does not run, check your current version with `sage --version`.

A Note on Reimplementing the Castryck-Decru Attack

A note A Note on Reimplementing the Castryck-Decru Attack and Lessons Learned for SageMath has been uploaded to eprint. We hope that this gives a good background for the work accomplished in this repo, as well as some more general tips for implementing cryptographic attacks in SageMath.

Deviation from Castryck-Decru Attack

A recent commit introduces a modification of the Castryck-Decru attack in which only the first β_1 ternary digits must be guessed. Now, instead of recovering the remaining digits one by one, the secret isogeny is directly calculated from the result of the (2,2)-isogeny chain.

This modification was introduced in #12 Implement direct computation of isogeny once the first splitting is found and was acomplished by Rémy Oudompheng.

A description of the attack is described in: A note on implementing direct isogeny determination in the Castryck-Decru SIKE attack.

Additional derivations appear in how the Glue-and-Split computations can be found, which can be seen in the file `richelot_aux.sage` in the functions:

- `FromProdToJac()`
- `FromJacToJac()`

Thanks again to Rémy Oudompheng for deriving and implementing these faster algorithms.

Figure 2: Giacomo Pope's PoC of Castryck-Decru attack on SIDH Isogeny Post-Quantum signature system.

2.3 Development for SageMath

For SageMath development, one has to set the environment so that one can update the source code easily, and rebuild a component if needed. It is highly recommended to manually build SageMath from source code, instead of downloading a prebuilt package for Linux and MacOS respectively. I used an Ubuntu 24.04 server for development instead of using my MacBook, for more compatibility. See Install from Source Code for more details on building SageMath from source code.

SageMath use various types of codes for it's functionality. The extension `.py` is mainly used, which is only run during the acutal runtime of the main code, like general Python files. Thus, if you edit a file with extension `.py`, there's no need to rebuild anything to see the impact.

However, there exists another type of extensions which are `.pxd`, `.pyx`. These files allows both Python grammar and C types. For example, you can define a C class using the `cdef` keyword, like `cdef class ComplexIntervalFieldElement(FieldElement):`. Becuase C is faster than Python in most of the cases, this allows the overall performance of SageMath to be faster. However, unlike `.py` extensions, editing these files cannot directly be applied to the main program because it includes C. Note that one has to rebuild the program for the affect to be applied. Gladly, it doesn't has to be rebuilt entirely, using the `sage -b` command, it is possible to only rebuild the changed files by passing the updated files as an argument.

For example, if the developer updated the source file `./src/sage/rings/polynomial/polynomi-al_modn_dense_ntl.pyx`, which extension is `.pyx` and needs to be rebuilt, running `sage -b ./src/sage/rings/polynomial/polynomial-_modn_dense_ntl.pyx` applies the change in a short time.

The compiler doesn't directly compile the `.pxd`, `.pyx` files, since it would require a whole new compiler development. Instead, it parses the `.pxd`, `.pyx` files into C at every segment when it's needed.

See Figure 3 for definition of class `ComplexIntervalFieldElement` using cdef, and see Figure 4 for definition of struct `_pyx_obj_4sage_5rings_16complex_interval...` which is actually compiled with the general GCC compiler, and be ready to called during running the main SageMath program.

Figure 3: Implementation of class ComplexIntervalFieldElement using cdef

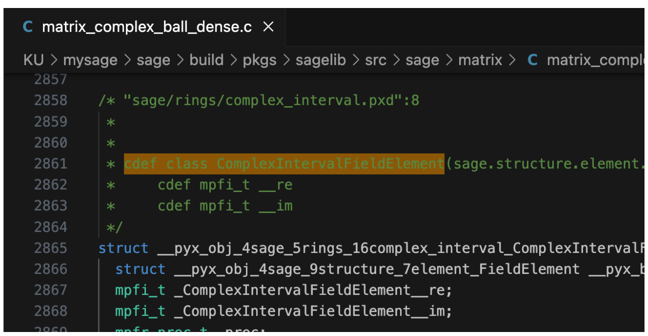


Figure 4: Implementation of struct ComplexIntervalFieldElement in C, deriven from .pyx

3 REIMPLEMENTING INVERT FUNCTION OF QUOTIENT POLYNOMIAL RINGS

3.1 Polynomial Rings

In mathematics, especially in the field of algebra, a polynomial ring or polynomial algebra is a ring formed from the set of polynomials in one or more indeterminates (traditionally also called variables) with coefficients in another ring, often a field. [1]

Often, the term "polynomial ring" refers implicitly to the special case of a polynomial ring in one indeterminate over a field. The importance of such polynomial rings relies on the high number of properties that they have in common with the ring of the integers.

[1]See https://en.wikipedia.org/wiki/Polynomial_ring

Polynomial rings occur and are often fundamental in many parts of mathematics such as number theory, commutative algebra, and algebraic geometry. In ring theory, many classes of rings, such as unique factorization domains, regular rings, group rings, rings of formal power series, Ore polynomials, graded rings, have been introduced for generalizing some properties of polynomial rings.

A closely related notion is that of the ring of polynomial functions on a vector space, and, more generally, ring of regular functions on an algebraic variety.

3.2 Univariate Polynomial Rings

Let K be a field or, more generally, a commutative ring.

The *polynomial ring* in X over K, denoted $K[X]$, can be defined in several equivalent ways. One such definition is to define $K[X]$ as the set of expressions, called *polynomials in* X, of the form

$$p = p_0 + p_1X + p_2X^2 + \cdots + p_{m-1}X^{m-1} + p_mX^m,$$

where $p_0, p_1, \ldots, p_m$—the coefficients of p—are elements of K, with $p_m \neq 0$ if $m > 0$. The symbols $X, X^2, \ldots$ are considered as "powers" of X, and follow the usual rules of exponentiation: $X^0 = 1$, $X^1 = X$, and

$$X^kX^\ell = X^{k+\ell}$$

for any nonnegative integers k and ℓ. The symbol X is called an *indeterminate* or a *variable*. (The term "variable" comes from the terminology of polynomial functions; however, here X has no value other than itself and cannot vary, being a constant within the polynomial ring.)

Two polynomials are equal if and only if the corresponding coefficients of each X^k are equal.

One can think of the ring $K[X]$ as arising from K by adjoining a new element X that is external to K, commutes with all elements of K, and has no other specific properties. This provides an equivalent construction of polynomial rings.

The polynomial ring $K[X]$ is equipped with addition, multiplication, and scalar multiplication operations, making it into a commutative algebra.

These operations are defined according to the standard rules for manipulating algebraic expressions.

3.3 Quotient Polynomial Rings

In the case of $K[X]$, the quotient ring by an ideal can be constructed, as in the general case, as a set of equivalence classes. However, since each equivalence class contains exactly one polynomial of minimal degree, another construction is often more convenient.

Given a polynomial p of degree d, the quotient ring of $K[X]$ by the ideal generated by p can be identified with the vector space of polynomials of degree less than d, with *multiplication modulo* p as the multiplication operation. Here, multiplication modulo p refers to taking the remainder upon division by p of the usual product of polynomials. This quotient ring is variously denoted as

$$K[X]/pK[X]$$

The ring $K[X]/(p)$ is a field if and only if p is an irreducible polynomial. In fact, if p is irreducible, every nonzero polynomial q of lower degree is coprime with p, and Bézout's identity provides polynomials r and s such that $sp + qr = 1$. Thus, r is the multiplicative inverse of q modulo p.

Conversely, if p is reducible, then there exist polynomials a, b of degrees less than $\deg(p)$ such that $ab = p$. Hence, a and b are nonzero zero divisors modulo p and cannot be invertible.

For example, the standard definition of the field of complex numbers can be summarized as the quotient ring

$$\mathbb{C} = \mathbb{R}[X]/(X^2 + 1),$$

where the image of X in $\mathbb{C}$ is denoted by i. By the above construction, this quotient consists of all polynomials of degree less than 2 in i, i.e., expressions of the form $a + bi$ with $a, b \in \mathbb{R}$. The remainder of the Euclidean division needed for multiplying two elements in the quotient ring corresponds to replacing i^2 with -1 in their polynomial product—precisely the usual definition of complex number multiplication.

I will be focusing on the specific cases where the ring of polynomial functions is a Finite Field, or an Integer Modulo Ring, with the univariate case.

3.4 Using Polynomial Rings on SageMath

To use Polynomial Ring in SageMath, the function `PolynomialRing` is helpful. For example, to define a Polynomial Ring on $\mathbb{F}_5$, `P.<x> = PolynomialRing(GF(5))` defined `P` the Polynomial Ring, and `x` the generator.

Evaluating `x^2 + 7*x + 9` output `x^2 + 2*x + 4`, because the base ring which is $\mathbb{F}_5$ has a property of integers moduluo 5, which is why the coefficients 7 and 9 became 2 and 4 respectively.

To use a Quotient Ring, a polynomial modulus has to be specified, note that the modulus doesn't necessarily have to be irreducible. The irreducibility is only defined on Polynomial Rings on Finite Field, thus not defined on Polynomial Rings on Integer Mod Rings. For an example, one can define the Quotient Polynomial Ring `Q` with generator `y`, with command `Q.<y> = P.quotient(x^3 + 3 * x^2 + 4)`.

Generally, the inverse of a polynomial element is not defined as a polynomial element, but for Quotient Polynomial Rings, the inverse is well defined, and can be determined if it exists or not. For example, using the Quotient Polynomial Ring from the upper paragraph, `1/y` gives the result of `y^2 + 3*y`. To check the correctness, we can evaluate `y * (y^2 + 3*y)`, and can see the result is 1, which proves the inverse is correctly computed.

However, the case for Quotient Polynomial Rings on non-Field base rings is quite different. Let us first define a composite number `N`, which is a product of two prime numbers.

```
sage: p = next_prime(2^64)
sage: q = next_prime(p)
sage: N = p * q
sage: N
340282366920938464385711811117245792737
```

Listing 1: Generating composite modulus N

Now, define a Polynomial Ring on the Integer Modulo Ring on `N`, using `Zmod`.

```
sage: P.<x> = PolynomialRing(Zmod(N))
sage: P
```

```
Univariate Polynomial Ring in x over Ring of integers modulo
      ↪ 340282366920938463463857118111117245792737 (using NTL)
```

Listing 2: Polynomial Ring definition

Define a random polynomial as modulus, and making a Quotient Polynomial Ring.

```
sage: quot = P.random_element(2).monic()
sage: quot
x^2 + 203923386675436777291950364769193178190*x +
      ↪ 175911500821280361038712429881418551097
sage: Q.<y> = P.quotient(quot)
sage: Q
Univariate Quotient Polynomial Ring in y over Ring of integers
      ↪ modulo 340282366920938463463857118111117245792737 with
      ↪ modulus x^2 + 203923386675436777291950364769193178190*x +
      ↪ 175911500821280361038712429881418551097
```

Listing 3: Quotient Polynomial Ring definition

However, in this case the inverse of polynomial element is not defined.

```
sage: 1/y
------------------------------------------...
NotImplementedError                         Traceback (most recent
      ↪ call last)
...
```

Listing 4: Failure of Inverse

3.5 Inverse on Composite Modulo Quotient Rings

This functionality is not implemented in SageMath for the latest version, however it is possible to compute the inverse. The most obvious approach to solving this problem is using linear equations. Let's first define the Quotient Ring.

$$
\begin{aligned}
Q &= x^d + q_{d-1}x^{d-1} + q_{d-2}x^{d-2} + \cdots + q_1x + q_0 \\
A &= a_{d-1}x^{d-1} + a_{d-2}x^{d-2} + \cdots + a_1x + a_0 \\
AB &= 1 \mod Q \\
B &= b_{d-1}x^{d-1} + b_{d-2}x^{d-2} + \cdots + b_1x + b_0
\end{aligned}
$$

d is the degree, and x is the generator of the Quotient Polynomial. Q is the quotient modulus, and A is the starting polynomial that we want to calculate the inverse. By the definition, $AB = 1 \mod Q$. The coefficient a_i, q_i are only known and the goal is to calculate b_i. Note that every values are defined over a Integer Modulo Ring, which allows modulus to be a composite number.

The expansion of AB is equal to the following.

$$
\begin{aligned}
AB &= b_{d-1}(Ax^{d-1}) + b_{d-2}(Ax^{d-2}) + \\
&\quad \cdots + b_1(Ax) + b_0(A) \\
&= b_{d-1}(Ax^{d-1} \bmod Q) + b_{d-2}(Ax^{d-2} \bmod Q) + \\
&\quad \cdots + b_1(Ax \bmod Q) + b_0(A \bmod Q) \\
&= 1 \mod Q
\end{aligned}
$$

Since A and Q are all known polynomials, it is possible to calculate the terms in form of $(Ax^i \bmod Q)$, which are multiplied to b_i respectively. This now becomes a set of linear equations on an Integer Modulo Ring.

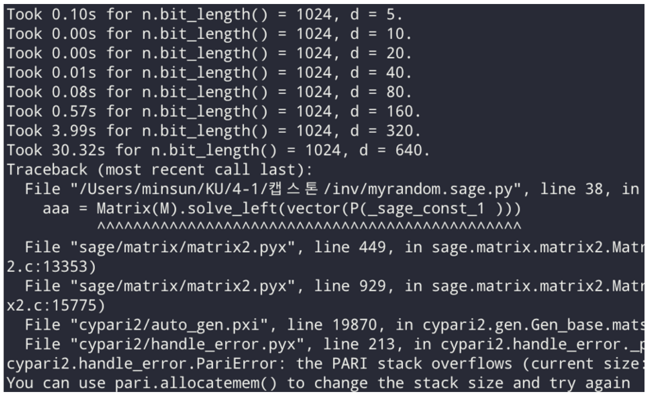
$$
\begin{aligned}
AB &= b_{d-1}(Ax^{d-1}) + b_{d-2}(Ax^{d-2}) + \\
&\quad \cdots + b_1(Ax) + b_0(A) \\
&= b_{d-1}(Ax^{d-1} \bmod Q) + b_{d-2}(Ax^{d-2} \bmod Q) + \\
&\quad \cdots + b_1(Ax \bmod Q) + b_0(A \bmod Q) \\
&= b_{d-1}(?x^{d-1} + ?x^{d-2} + \cdots + ?x + ?) \\
&\quad + b_{d-2}(?x^{d-1} + ?x^{d-2} + \cdots + ?x + ?) \\
&\qquad \vdots \\
&\quad + b_1(?x^{d-1} + ?x^{d-2} + \cdots + ?x + ?) \\
&\quad + b_0(?x^{d-1} + ?x^{d-2} + \cdots + ?x + ?) \\
&= (0x^{d-1} + 0x^{d-2} + \cdots + 0x + 1)
\end{aligned}
$$

There exists in total d linear equations with d variables. Because the ? marks in the above equations are all known values, it's possible to solve the equation.

Code 7.1 implements the above calculation using the method function `solve_left`. Figure 5 is the result of running the code. It can be seen that the calculations are correct, however the running time and memory usage is exponential to degree d and runs out of default stack size when $d = 1280$.

3.6 Using Extended GCD for inversion

In arithmetic and computer programming, the **Extended Euclidean algorithm** is an extension of the Euclidean algorithm. It computes, in addition to the greatest common divisor (gcd) of integers



```
Took 0.10s for n.bit_length() = 1024, d = 5.
Took 0.00s for n.bit_length() = 1024, d = 10.
Took 0.00s for n.bit_length() = 1024, d = 20.
Took 0.01s for n.bit_length() = 1024, d = 40.
Took 0.08s for n.bit_length() = 1024, d = 80.
Took 0.57s for n.bit_length() = 1024, d = 160.
Took 3.99s for n.bit_length() = 1024, d = 320.
Took 30.32s for n.bit_length() = 1024, d = 640.
Traceback (most recent call last):
  File "/Users/minsun/KU/4-1/캡스톤/inv/myrandom.sage.py", line 38, in
    aaa = Matrix(M).solve_left(vector(P(_sage_const_1 )))
          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "sage/matrix/matrix2.pyx", line 449, in sage.matrix.matrix2.Matr
2.c:13353)
  File "sage/matrix/matrix2.pyx", line 929, in sage.matrix.matrix2.Matr
x2.c:15775)
  File "cypari2/auto_gen.pxi", line 19870, in cypari2.gen.Gen_base.mats
  File "cypari2/handle_error.pyx", line 213, in cypari2.handle_error._p
cypari2.handle_error.PariError: the PARI stack overflows (current size:
You can use pari.allocatemem() to change the stack size and try again
```

Figure 5: Result of inversion using solve_left

a and b, also the coefficients of Bézout's identity, which are integers x and y such that

$$ax + by = \gcd(a, b).$$

This is a *certifying algorithm*, because the gcd is the only number that can simultaneously satisfy this equation and divide the inputs.[2]

It also allows one to compute, with almost no extra cost, the quotients of a and b by their greatest common divisor.

The term Extended Euclidean algorithm also refers to a very similar algorithm for computing the polynomial greatest common divisor and the coefficients of Bézout's identity for two univariate polynomials.

The extended Euclidean algorithm is particularly useful when a and b are coprime. In that case, x is the modular multiplicative inverse of a modulo b, and y is the modular multiplicative inverse of b modulo a. Similarly, the polynomial extended Euclidean algorithm allows one to compute the multiplicative inverse in algebraic field extensions, and in particular in finite fields of non-prime order.

It follows that both versions of the extended Euclidean algorithm are widely used in cryptography. In particular, the computation of the modular multiplicative inverse is an essential step in the derivation of key-pairs in the RSA public-key encryption method.

The Extended Euclidean algorithm is also called the Extended-GCD algorithm, or XGCD algorithm. SageMath implements the method function `gcd`, `xgcd` in many cases. As mentioned previously, XGCD also can be implemented on polynomials, especially on Finite Fields, or Integer Modulo Rings.

```
sage: F = GF(17)
sage: P.<x> = PolynomialRing(F)
sage: a = P.random_element(10)
sage: b = P.random_element(10)
sage: a
9*x^10 + 12*x^9 + 11*x^8 + 15*x^7 + 4*x^6 + 16*x^5 + 6*x^4 + 14*x^3
    ↪ + 11*x^2 + 6*x + 3
sage: b
11*x^10 + 6*x^9 + 16*x^8 + 8*x^7 + 13*x^6 + 11*x^5 + 15*x^4 + 7*x^3
    ↪ + 11*x^2 + 9*x + 6
sage: a.gcd(b)
1
sage: a.xgcd(b)
(1,
 x^9 + 16*x^8 + 7*x^7 + 4*x^6 + 6*x^5 + 13*x^4 + 16*x^3 + 2*x^2 +
    ↪ 4*x + 7,
 10*x^9 + 2*x^8 + 5*x^7 + 16*x^6 + 8*x^5 + 8*x^4 + 14*x^3 + 16*x^2
    ↪ + 13*x + 8)
```

Listing 5: XGCD on Polynomials defined on $\mathbb{F}_{17}$

However, unlike the example 5, the GCD and XGCD method is not implemented on Polynomial Rings defined on Composite Modulo Rings.

```
sage: F = Zmod(next_prime(2^64) * next_prime(2^65))
sage: P.<x> = PolynomialRing(F)
sage: a = P.random_element(10)
sage: b = P.random_element(10)
sage: a
9*x^10 + 12*x^9 + 11*x^8 + 15*x^7 + 4*x^6 + 16*x^5 + 6*x^4 + 14*x^3
    ↪ + 11*x^2 + 6*x + 3
sage: b
11*x^10 + 6*x^9 + 16*x^8 + 8*x^7 + 13*x^6 + 11*x^5 + 15*x^4 + 7*x^3
    ↪ + 11*x^2 + 9*x + 6
sage: a.gcd(b)
...
NotImplementedError: ...
```

Listing 6: XGCD on Polynomials defined on Zmod composite modulus

Ironically, the GCD and XGCD is also computable on Composite Modulo Rings as well. A trivial way to compute GCD, XGCD on Polynomial Ring is equal to the following example 7. The time complexity achieves $O(d^2)$, which is better than the previous Matrix equation solving which takes at least $O(d^{2.3})$. However, using the known algorithm: Half-GCD algorithm, we can achieve much better.

```
P.<x> = PolynomialRing(F)

a, b = P.random_element(10), P.random_element(10)

```

[2]See https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm

```
while b != 0:
    a, b = b, a % b
g = b.monic()
```

Listing 7: Simple implementation of GCD on Polynomial Rings

3.7 The Half-GCD algorithm

Given integers a and b with close to $2n$ bits each, the Half-GCD of a and b is a 2×2 matrix

$$\begin{bmatrix} u & v \\ u' & v' \end{bmatrix}$$

with determinant equal to ± 1, such that:

$$ua + vb = r \quad \text{and} \quad u'a + v'b = r',$$

where r and r' each have a number of bits close to n.

The Half-GCD is computed by performing roughly half of the Euclidean algorithm for computing the greatest common divisor $\gcd(a, b)$. There exists an efficient algorithm for computing the Half-GCD of two large integers which, when applied recursively, allows the greatest common divisor to be computed faster than using the classical Euclidean algorithm.

There also exists a work of optimizing Half-GCD algorithm. In the corresponding work, they propose a carefully optimized "half-gcd" algorithm for polynomials. They achieved a constant speed-up with respect to previous work for the asymptotic time complexity. They discussed special optimizations that are possible when polynomial multiplication is done using radix two FFT(Fast Fourier Transform)s.

The time complexity is roughly $O(d \log^2 d)$, which is much more useful compared to the naive Euclidean algorithm. The usage of Half-GCD algorithm shortens the runtime of GCD and XGCD of polynomials, and can be both applied to Finite Field Polynomial Rings, and Integer Modulo Ring Polynomial Rings.

3.8 Using PARI/GP for Half-GCD algorithm

PARI/GP is a cross platform and open-source computer algebra system designed for fast computations in number theory: factorizations, algebraic number theory, elliptic curves, modular forms, L functions... It also contains a wealth of functions to compute with mathematical entities such as matrices, polynomials, power series, algebraic numbers, etc., and a lot of transcendental functions as well as numerical summation and integration routines. PARI is also available as a C library to allow for faster computations. SageMath includes this system internally, and flexibly use the components of PARI/GP.

SageMath implements Half-GCD algorithm for specific cases, where FLINT is used as the implementation of the Polynomial Ring, and only the casd where base ring is a Finite Field. Gladly, PARI/GP implements the other cases, where NTL is used as implementation, and this includes the case where Composite Modulo Ring is used as the base ring.

Code 7.2 implements the inversion algorithm using PARI/GP which includes Half-GCD algorithm. In Figure 6, it can be seen that the running time is decreased by a surprising difference, and there is no more stack memory limit issues.

```
Took 0.00s for n.bit_length() = 1024, d = 5.
Took 0.00s for n.bit_length() = 1024, d = 10.
Took 0.00s for n.bit_length() = 1024, d = 20.
Took 0.00s for n.bit_length() = 1024, d = 40.
Took 0.01s for n.bit_length() = 1024, d = 80.
Took 0.28s for n.bit_length() = 1024, d = 1000.
Took 0.72s for n.bit_length() = 1024, d = 2000.
Took 1.86s for n.bit_length() = 1024, d = 4000.
Took 4.37s for n.bit_length() = 1024, d = 8000.
Took 10.80s for n.bit_length() = 1024, d = 16000.
```

Figure 6: Result of inversion using Half-GCD

3.9 Comparing with SageMath's runtime, dependent on implementation

The `inverse_mod` function in SageMath does not directly invoke the XGCD functions. However, the primary goal is to develop a properly working Half-GCD XGCD function.

```
p = next_prime(2^1024)
P.<x> = PolynomialRing(GF(p), implementation='FLINT')
$ sage xgcdtest.sage
Traceback (most recent call last):
...
ValueError: FLINT does not support modulus 1797...
```

Listing 8: Testing FLINT's modulus range

FLINT is fast, but only supports moduli up to 2^{63}. NTL, on the other hand, is slower but supports arbitrary-precision integers. Also it should be noted that FLINT only supports moduli in `range(2^63)`.

```
sage: PolynomialRing(Zmod(2^63 - 1), implementation='FLINT')
Univariate Polynomial Ring in x over Ring of integers modulo
    ↪ 9223372036854775807
sage: PolynomialRing(Zmod(2^63), implementation='FLINT')
ValueError: FLINT does not support modulus 9223372036854775808
```

Listing 9: Testing FLINT's modulus range 2

I used the following Code 10 for comparing runtime in several cases, with different implementation including FLINT and NTL.

```
import time

P.<x> = PolynomialRing(?)
deg = ?

a = P.random_element(deg)
b = P.random_element(deg)

st = time.time()
a.gcd(b)
en = time.time()
print(f"GCD took {(en - st):.2f}s.")

st = time.time()
a.xgcd(b)
en = time.time()
print(f"XGCD took {(en - st):.2f}s.")

st = time.time()
a._pari_with_name().gcd(b._pari_with_name())
en = time.time()
print(f"PARI GCD took {(en - st):.2f}s.")

st = time.time()
a._pari_with_name().gcdext(b._pari_with_name())
en = time.time()
print(f"PARI XGCD took {(en - st):.2f}s.")
```

Listing 10: The benchmark code for time comparison

1. Small Prime Modulus with FLINT:. `p = next_prime(2^16)`, `deg = 50000`

- GCD took 0.19s
- XGCD took 0.30s
- PARI GCD took 0.64s
- PARI XGCD took 0.68s

FLINT uses the half-GCD algorithm and outperforms PARI. Unimprovable with PARI.

2. Small Prime Modulus with NTL:. `p = next_prime(2^16)`, `deg = 10000`

- GCD took 0.12s
- XGCD took 11.95s
- PARI GCD took 0.08s
- PARI XGCD took 0.09s

NTL's GCD uses Half-GCD and is slightly slower than PARI. NTL's XGCD does not use half-GCD and is much slower.

More tests:

- Degree = 100000: GCD took 1.79s, PARI GCD took 1.24s
- Degree = 1000000: GCD took 20.62s, PARI GCD took 18.81s
- Degree = 100, 10000 iterations: GCD took 0.12s, PARI GCD took 0.50s

NTL XGCD is still faster than PARI for very low degree like 10.

3. Small Composite Modulus with FLINT:. `n = 1073741827 * 1073741831`, `deg = 100000`

- GCD took 1.66s
- XGCD took 2.57s
- PARI GCD took 3.58s
- PARI XGCD took 3.86s

FLINT is again better than PARI.

4. Multi-dimensional Fields with Small Modulus: `P.<x> = PolynomialRing(GF(256), impleme-ntation='FLINT')`

Error: FLINT doesn't support extension fields.

`P.<x> = PolynomialRing(GF(256), impleme-ntation='NTL')`, `deg = 30000`

- GCD took 27.59s
- XGCD took 31.42s
- PARI GCD took 7.29s
- PARI XGCD took 8.46s

For degree 60000:

- PARI GCD took 15.47s
- PARI XGCD took 18.73s

NTL does not use half-GCD in this case. PARI wins.

5. Big Prime Modulus with NTL:. `p = next_prime(2^1024)`, `deg = 5000`

- GCD took 0.79s
- XGCD took 32.75s
- PARI GCD took 2.12s

- PARI XGCD took 2.30s

NTL's GCD is competitive, but XGCD is still much slower.

6. Big Composite Modulus with NTL:. `p = next_prime(2^512), q = next_prime(p), P.<x> = PolynomialRing(Zmod(p*q), imple -mentation='NTL')`

Error: `NotImplementedError: Ring of integers modulo ... does not provide a gcd/xgcd implementation for univariate polynomials`

- PARI GCD took 2.12s
- PARI XGCD took 2.31s

NTL doesn't support GCD/XGCD for composite modulus, but PARI does, and its speed is consistent with large prime modulus cases.

3.10 Updating SageMath source code to use PARI/GP

I opened an issue, and made a pull request to the main develop branch of SageMath's GitHub repository. The following is the entire description of the issue I opened.

Problem Description. Polynomials defined on rings with FLINT have well - implemented gcd/xgcd methods, internally using the Half-GCD algorithm for fast computation.

However, if the modulus exceeds $2^{63} - 1$ and FLINT cannot be used (forcing use of NTL), there is room for improvement:

- For **composite moduli**, no proper gcd/xgcd is implemented in NTL.
- For **prime moduli**, gcd uses half-GCD, but xgcd does not and is slow for high-degree polynomials.

Proposed Solution. Use PARI's `gcd` and `gcdext` functions, which implement the half-GCD algorithm efficiently.

Example (assuming `a` and `b` are polynomials from the same ring):

```
P = a.parent()
g = P(a._pari_with_name().gcd(b._pari_with_name()))
s, t, r =
    ↪ a._pari_with_name().gcdext(b._pari_with_name())
s, t, r = map(P, [s, t, r])
```

Notes:

- The result `r = a*s + b*t` still holds, even though `gcdext` returns `(s, t, r)` in a different order.
- PARI's output is not automatically monic; leading coefficients are preserved.

Alternatives Considered. None.

Additional Information. Polynomial rings over extension fields have gcd/xgcd implemented, but not with Half-GCD. PARI does support Half-GCD here too.

The file `src/sage/rings/polynomial/ -polynomial_template.pxi` shows that `celement_gcd`/`celement_xgcd` are used internally. I wasn't exactly sure how to update these, and would welcome suggestions or a patch.

This is my first SageMath issue/PR—please let me know if I broke any unspoken rules!

In the pull request, two files were edited in total. The names are `sage/rings/polynomial /polynomial_modn_dense_ntl.pyx`, and `sage/ rings/polynomial/polynomial_element.pyx`.

See code 7.3 and code 7.4 for the difference between the original version and the revised version respectively.

3.11 Overall review from a senior contributor

There existed lot's of come-and-gos during the edit, since I was relatively less aware of the rule of SageMath's contributing, and it's environment. Unexpectedly, doctests were one of the most important features of SageMath because it's a system for educational purpose. The effort one has to make on doctests were nothing less than the main codes.

There was a lot of interactions between user202729, who is an active contributor of SageMath, and an avid Crypto CTF player himself. He spotted a lot of bugs of mine, and suggestions for a fix as well. The following is the latest review related to the GCD/XGCD update.

> This causes a lot of test failures, which I think is because of setting algorithm=pari by default, which causes issues because many base rings does not support algorithm=pari.
> My recommendation is to
> modify the generic _xgcd_univariate_poly-nomial implementation to add algorithm=pari, override _xgcd_univariate_polynomial of supported base classes to default to algorithm=pari by calling super().something, then modify gcd to pass the algorithm parameter down.

4 ENUMERATING ALL SOLUTIONS TO $Mx = v$ OVER FINITE FIELDS IN SAGEMATH

Many CTF(Capture The Flag) challenges reduce to solving a linear system $Mx = v$ over a finite field. `SageMath` offers solve_right to obtain a single solution, but when the right kernel of M is non-trivial the complete solution set forms an affine space. This section formalises that fact and shows a concise single-pass method that enumerates every solution while performing only one matrix reduction.

4.1 A quick recap of solve_right

Consider an example over $\mathbb{F}_7$:

```
M = random_matrix(GF(7), 10)
v = random_vector(GF(7), 10)
sol = M.solve_right(v)
assert M * sol == v
```

Listing 11: Unique solution example

Here `sol` is unique because M is a reversible matrix. The call internally performs a reduced row echelon form (RREF), which costs $O(n^3)$ field operations. There may exist faster algorithms implemented, that can be reduced down to $O(n^{2.3})$.

4.2 When the solution space is not unique

If M has a non-zero right kernel, multiple solutions exist. Let r be any particular solution and let $K := \ker_R(M)$. Then

$$\text{all solutions} = r + K := \{\, r + a \mid a \in K \,\}.$$

Sketch. For any $a \in K$ we have $Ma = 0$, hence $M(r + a) = Mr + Ma = v$. Conversely, if x is any solution, then $x - r \in K$. □

The right kernel is returned by `M.right_kernel_matrix()`:

```
K = M.right_kernel_matrix()
assert (M * K.T).is_zero()
```

Listing 12: Right kernel demonstration

4.3 Brute enumeration with two reductions

A direct approach calls `solve_right` (first reduction) to get r, then `right_kernel_matrix` (second reduction) to get K. Over $\mathbb{F}_q$ with $\dim K = \ell$ the total number of solutions is q^ℓ.

```
from itertools import product

def iterate_all(M, v):
    r = M.solve_right(v)
    K = M.right_kernel_matrix()
    P = r.base_ring()
    for coeffs in product(P, repeat=K.nrows()):
        yield r + sum(c * k for c, k in zip(coeffs, K))
```

Listing 13: Two-pass enumeration

This doubles the Gaussian elimination cost and requires exception handling to detect inconsistent systems.

4.4 The single-pass kernel trick

Attach v as an extra column and compute the right kernel once:

$$\begin{bmatrix} -v \mid M \end{bmatrix} x = 0.$$

The first row $(1 \mid r)$ encodes a particular solution; the remaining rows span the kernel. Enumeration therefore costs a single RREF and no `try` or `except` block for handling error raise in Python. This difference is significant because error raising and catching is very costly in Python. See Code 7.5 for the full implementation in SageMath.

4.5 Practical notes

The method requires the base ring to be a field. Over rings that are not fields, RREF is not well defined. The `solve_right` and `solve_left` algorithm is also differently implemented in different rings such as Integer Modulo Rings, and it is more complicated compared to Field operations.

In cryptanalytic workloads the single-pass method can almost halve the running time when q is small and the kernel dimension is large. The approach was discovered during a late-night CTF challenge and later refined for inclusion in SageMath documentation.

5 FIXING 4301+ DIGITS DECIMAL INTEGERS

5.1 Large Integers in Python

Python's answer to CVE-2020-10735 was to impose a hard cap on the number of decimal digits that the built-in conversions `int()` $\rightarrow$ `str()` will handle before they raise ValueError. Converting an enormous bignum to or from base 10 is nearly quadratic in time, so a few-megabyte string can pin a CPU core for seconds or minutes and yield an inexpensive denial-of-service. There exists a better time complexity than quadratic time using FFT(Fast Fourier Transform), however the DOS CVE was still assigned.

After several rounds of discussion on the Python Security Response Team mailing list, the Steering Council approved a default ceiling of 4300 digits. It was chosen to balance the following three goals:

(1) keep the worst-case conversion latency well under a few hundred milliseconds on commodity hardware;
(2) avoid breaking the bulk of existing scientific and financial code;
(3) preserve large open-source test suites (e.g. NumPy), which use integers up to roughly 14 k bits, i.e. 4300 digits.

Three runtime controls let applications raise, lower, or disable the limit:

- `sys.set_int_max_str_digits(n)` — programmatic API;
- `PYTHONINTMAXSTRDIGITS` — environment variable;
- `-X int_max_str_digits` — command-line switch.

Passing `0` removes the restriction entirely. Complementary getters `sys.get_int_max_str_digits()` and two fields in `sys.int_info`; `sys.default_max_str_digits`, `sys.str_digits_check_threshold` allow libraries to adapt at import time without hard-coding version checks.

The change landed abruptly and initially broke several number-crunching ecosystems (SymPy, SageMath, some blockchain clients). Projects that truly need unbounded conversions are expected either to increase the ceiling at start-up or to migrate hot paths to power of 2 encodings (hexadecimal or raw bytes, or even binary, octal) where no limit applies. For security-sensitive code that parses potentially hostile input, JSON deserialisers, logging sinks, RPC frameworks, the limit provides a simple, opt-out safety net, removing the burden of ad-hoc length checks.

5.2 CTF challenge regarding the mitigation

Using the fact that the int, str conversion is only allowed till 4300 digits, the allowed range for conversion is -10^{4300} to 10^{4300} exclusively. A CTF challenge named ZKPoF required an exploit using this bound, to binary search a value, using the error as an oracle. There exists some exclusive steps requiring Coppersmith's Attack in order to fully retreive the flag. See Code 7.6 for more details.

5.3 SageMath's mitigation to 4300 digit limit

SageMath always imports the module `all` which is written in `all.py`. In Figure 7, it can be seen that it internally always sets the limit to zero, which means completely removing the limit. It can be inferred that SageMath attempts to completely remove this mitigation of digit length limit, to give users ability to use any kind of numbers including integers without limitation.

However, when one tries to use an integer with 4301 or more digits with extension `*.sage`, running the code with `sage *.sage` raises the error in Figure 8. The specific error message is the following:

- SyntaxError: Exceeds the limit (4300 digits) for integer string conversion: value has 5000 digits; use sys.set_int_max_str_digits() to increase the limit - Consider hexadecimal for huge integer literals to avoid decimal conversion limits.

5.4 Preparser handling numbers

When one write `a = 11...11` (with 5000 digits of 1) into the file `long.sage`, running `sage long.sage` will first parse the file into `long.sage.py`, then it will be ran just like a normal Python code, but with additional SageMath modules imported with `from sage.all import *` or `from sage.all_cmdline import *`.

The content of the generated `long.sage.py` looks like the following according to the latest SageMath preparser.

```

# This file was *autogenerated* from the file long.sage
from sage.all_cmdline import *   # import sage library

_sage_const_11...11 = Integer(11...11)
a = _sage_const_11...11
```

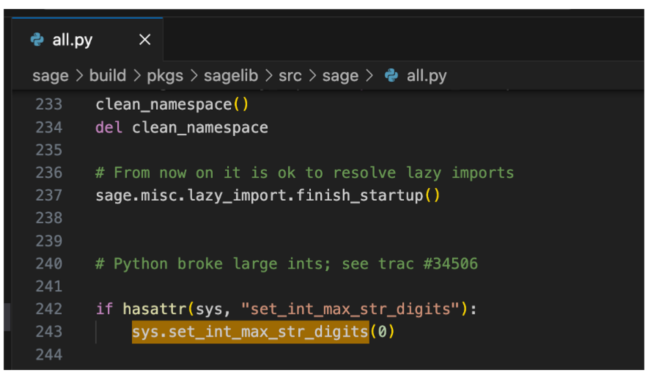


Figure 7: The partial source of all.py

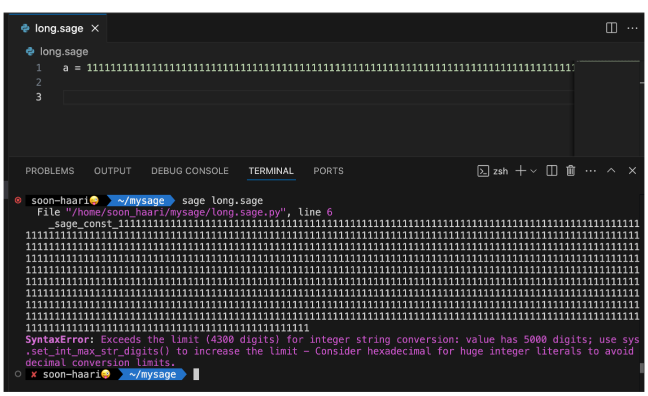


Figure 8: The error message after using long digits in *.sage

It can be seen that it's aiming to use the decimal integer of `<class 'int'>` with 5000 digits, and `all.py` is ran during the import `from sage.all_cmdline import *`, so the decimal limit is set to 0, thus unlimited. However that is not allowed even if the limit it set to infinity. The object that is generated by the 5000 decimal digits is run before the actual Python code is run, by the Python's compiler, and the limit is still set to 4300 at that time. So if one wants to use a large integer, it is recommended to use hex strings, or using the `int` keyword: `int("11...11")`.

Unlike Python, SageMath aims to allow users to use any kind of numbers, so it is a preparser's job to manage them. A working fix for this problem would be using the decimal strings as a Python `<class 'str'>`, so that calling class `Integer('11...11')` would be done without any problems. One can make preparser add quotes on Integers on default, or for the case where it's longer than 4300 digits only.

In the file `src/sage/repl/preparse.py`, the whole preparser is implemented here, and specifically the function `preparse_numeric_literals` handles the numeric literals including Integers for `*.sage` files and the shell command line. The following snippet is a summary of the fix. See Code 7.7 for the entire diff including the updated documentations.

```
def preparse_numeric_literals(code, extract=False, quotes="'"):
    ...
            if len(num) <= 4300:
                num_make = "Integer(%s)" % num
            elif quotes:
                num_make = "Integer(%s%s%s)" % (quotes, num,
    ↪ quotes)
            else:
                code_points = list(map(ord, list(num)))
                num_make = "Integer(str().join(map(chr, %s)))"
    ↪ % code_points
```

5.5 Worthy communications during the fix

My initial fix forced the preparser to make every decimal integers passed as a literal string, instead of making a condition if the length is over 4300.

The reasoning behind this was that every other types that are handled in the function preparse_numeric_literals, specifically Real numbers and Imaginary number, are always passed as <class 'str'> no matter what.

```
a = 1
b = 1.1
c = 1 + 1j
d = -1.1 + 1.1j
```

```


# This file was *autogenerated* from the file long.sage
from sage.all_cmdline import *   # import sage library

_sage_const_1 = Integer(1); _sage_const_1p1 = RealNumber('1.1');
    ↪ _sage_const_1j = ComplexNumber(0, '1'); _sage_const_1p1j =
    ↪ ComplexNumber(0, '1.1')
a = _sage_const_1
b = _sage_const_1p1
c = _sage_const_1  + _sage_const_1j
d = -_sage_const_1p1  + _sage_const_1p1j
```

However, an online contributer vincentmacri suggested the following comment:

> Have you tested the performance impact of this on integers under 4301 digits? When I test it, Integer('1234567890') is much slower than Integer(1234567890).
>
> Instead, I think you should only parse it as a string if it's over the limit that int can handle.
>
> Make sure to also test that this works with octal, hex, and binary values.

I did a quick experiment to compared the speed difference of passing the argument as <class 'int'> and <class 'str'>. Figure 9 is the code I used for testing, and in Figure 10, it can be seen that the speed difference is nearly 8 times slower using <class 'str'>.

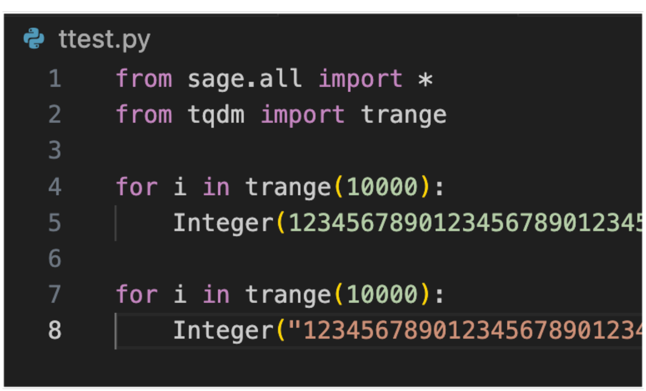


Figure 9: The test code for speed comparison.

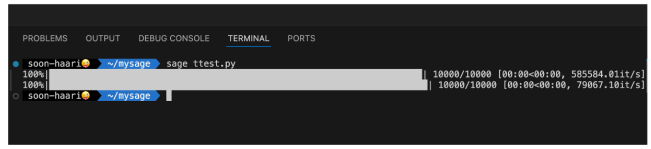


Figure 10: The test result for speed comparison.

Also he mentioned to check if the fix works with octal, hex, and binary values, so I replied to him including the reason why we can ignore the cases for base power of 2.

> Hello, thanks for the suggestion! I now made it to only add quotes when the length is longer than 4300.
>
> For octal, hex, and binary values, it is on a completely different elif branch:
>
> sage/src/sage/repl/preparse.py
>
> Lines 1315–1316 in aaf84e1:
>
> ```
> elif len(num) < 2 or num[1] in 'oObBxX':
>     num_make = "Integer(%s)" % num
> ```
>
> So it's kept used without quotes. Which works fine without the 4300-digit limit since the base is a power of 2 for all '0b...', '0o...', '0x...'.
>
> (python/cpython#95778: the original 4300-digit cap mitigates the O(bitlength²) cost of base-10 conversions, but the cost is only O(bitlength) for bases 2, 8, 16.)
>
> Edit: Apparently CPython now uses a better-than-quadratic algorithm even for base 10, but the limit still doesn't apply to power-of-two bases.

6 FUTURE WORK

The first issue and pull request related to the GCD/XGCD implementation broke a lot of other functionalities, and needs to be trimmed. I am planning to work on them for the second semester, while actively interacting with the other contributers including user202729.

In the other hand, the issue and pull request related to the 4300 digit limit, is almost ready to be merged and got a positive review. It will most likely be merged during the summer vacation.

I am planning to continue this topic for the second semester as well, from what I have experienced while using SageMath for playing CTFs and writing challenges.

I recently found a bug, while upsolving a CTF challenge named `Quo vadis?` which is related to the `Zp` ring, and my solution was to solve it manually with `Zmod` Polynomial Rings, by incrementing the degree one by one.

```
sage: P.<a, b> = PolynomialRing(Zmod(2^64))
sage: a * 2^63
---------------------------------------------------------
KeyError                  Traceback (most recent call last)
File /private/var/tmp/sage-10.7-current/local/var/lib/sage/venv
      ↪ -python3.13.3/lib/python3.13/site-packages/sage/structure
      ↪ /coerce.pyx:1223, in
      ↪ sage.structure.coerce.CoercionModel.bin_op
      ↪ (build/cythonized/sage/structure/coerce.c:15771)()
   1222 try:
-> 1223     action = self._action_maps.get(xp, yp, op)
   1224 except KeyError:

File /private/var/tmp/sage-10.7-current/local/var/lib/sage/venv
      ↪ -python3.13.3/lib/python3.13/site-packages/sage/structure
      ↪ /coerce_dict.pyx:1321, in
      ↪ sage.structure.coerce_dict.TripleDict.get
      ↪ (build/cythonized/sage/structure/coerce_dict.c:10828)()
   1320 if not valid(cursor.key_id1):
-> 1321     raise KeyError((k1, k2, k3))
   1322 value = <object>cursor.value

KeyError: (Multivariate Polynomial Ring in a, b over Ring of
      ↪ integers modulo 18446744073709551616, Integer Ring,
      ↪ <built-in function mul>)

During handling of the above exception, another exception occurred:

OverflowError             Traceback (most recent call last)
OverflowError: Python int too large to convert to C long
Exception ignored in: 'sage.libs.singular.singular.sa2si_ZZmod'
Traceback (most recent call last):
  File "<ipython-input-2-1a64d87be5c1>", line 1, in <module>
OverflowError: Python int too large to convert to C long
---------------------------------------------------------
OverflowError             Traceback (most recent call last)
OverflowError: Python int too large to convert to C long
Exception ignored in: 'sage.libs.singular.singular.sa2si_ZZmod'
Traceback (most recent call last):
  File "<ipython-input-2-1a64d87be5c1>", line 1, in <module>
OverflowError: Python int too large to convert to C long
0
```

This happens while using the Multivariate Polynomial Ring defined on `Zmod(2^64)`. The reasoning behind this is using a specific new implementation as default for performance improvements such as speed, however leading to an error. Using `P.<a, b> = PolynomialRing(Zmod(2^64), implementation='generic')` fixes the issue, but the bug still needs to be fixed. I am planning on fixing this issue if no one works on it during the summer vacation.

7 CODE

7.1 Inversion with linear equations

```
n = int("""
135066410865995223349603216278805969938881475605
66702752448514385152651060485953383394028715057 1
90944179820728216447155137368041970396419174304 6
49658927425623934102086438320211037295872576235 8
50964311056407350150818751067659462920556368552 9
47521350085287941637732853390610975054433499981 1
1500569772368909275 63
""".replace("\n", ""))

F = Zmod(n)

for d in [5, 10, 20, 40, 80, 160, 320, 640, 1280]:
    P.<x> = PolynomialRing(F)
    Q = P.random_element(d).monic()

    P.<x> = P.quotient(Q)

    A = P.random_element()

    import time

    st = time.time()
    d = P.modulus().degree()
    cur = A
    M = []
    for i in range(d):
        M.append(vector(cur))
        cur *= x
    M = Matrix(M)
    aaa = Matrix(M).solve_left(vector(P(1)))

    B = P(list(aaa))

    en = time.time()

    print(f"Took {en - st:.2f}s for {n.bit_length() = }, {d = }.")

    assert A * B == 1
```

7.2 Inversion with PARI/GP's Half-GCD algorithm

```
for d in [5, 10, 20, 40, 80, 1000, 2000, 4000, 8000, 16000]:

        n = int("""
13506641086599522334960321627880596993888147560 5
66702752448514385152651060485953383394028715057 1
90944179820728216447155137368041970396419174304 6
49658927425623934102086438320211037295872576235 8
50964311056407350150818751067659462920556368552 9
47521350085287941637732853390610975054433499981 1
1500569772368909275 63
""".replace("\n", ""))

        Zn = Zmod(n)
        P.<x> = PolynomialRing(Zn)

        quotient = P.random_element(d).monic()
        Q.<y> = P.quotient(quotient)
```

```

        target = Q.random_element()

        import time

        st = time.time()

        self = target

        parent = self.parent()
        _, a, g = parent.modulus()._pari_with_name().gcdext(self._polynomial._pari_with_name())
        if len(g) != 1:
                print("inverse doesnt exist")
                exit()
        res = parent(~g[0]*a)

        en = time.time()
        print(f"Took {en - st:.2f}s for {n.bit_length() = }, {d = }.")
        assert res * target == 1
```

7.3 Diff of file sage/rings/polynomial/polynomial_modn_dense_ntl.pyx

```
$ diff original-polynomial_modn_dense_ntl.pyx\ new-polynomial_modn_dense_ntl.pyx
1975c1975
<     def xgcd(self, other):
---
>     def xgcd(self, other, algorithm='pari'):
1981a1982
>         - ``algorithm='pari'`` -- the algorithm used for computing gcd of two polynomials, should be 'pari' or 'ntl'
1986a1988,1989
>
>         .. NOTE::
1987a1991,1997
>             Algorithm is set to 'pari' in default, but user may set it to 'ntl'
>             to use the algorithm using ``ntl_ZZ_pX.gcd``. Algorithm 'pari' implements
>             half-gcd algorithm in some cases, which is significantly faster
>             for high degree polynomials. Generally without half-gcd algorithm, it is
>             infeasible to calculate gcd/xgcd of two degree 50000 polynomials in a minute
>             but TEST shows it is doable with algorithm 'pari'.
>
2000a2011,2021
>
>         TESTS::
>
>             sage: P.<x> = PolynomialRing(GF(next_prime(2^512)), implementation='NTL')
>             sage: degree = 50000
>             sage: g_deg = 10000
>             sage: g = P.random_element(g_deg).monic()
>             sage: a, b = P.random_element(degree), P.random_element(degree)
>             sage: r, s, t = a.xgcd(b)
>             sage: (r == a.gcd(b)) and (r == s * a + t * b)
>             True
2002,2003c2023,2024
<         r, s, t = self.ntl_ZZ_pX().xgcd(other.ntl_ZZ_pX())
<         return self.parent()(r, construct=True), self.parent()(s, construct=True), self.parent()(t, construct=True)
---
>         if algorithm not in ('pari', 'ntl'):
>             raise ValueError(f"unknown implementation {algorithm!r} for xgcd function over {self.parent()}")
2004a2026,2041
>         if algorithm == 'pari':
>             P = self.parent()
>             s, t, r = self._pari_with_name().gcdext(other._pari_with_name())
>             s = P(s)
>             t = P(t)
>             r = P(r)
>             c = r.leading_coefficient()
>             if c != P(0):
>                 s /= c
>                 t /= c
>                 r /= c
>             return r, s, t
>         else:
>             r, s, t = self.ntl_ZZ_pX().xgcd(other.ntl_ZZ_pX())
>             return self.parent()(r, construct=True), self.parent()(s, construct=True), self.parent()(t, construct=True)
>
```

7.4 Diff of file sage/rings/polynomial/polynomial_element.pyx

```
$ diff original-polynomial_element.pyx new-polynomial_element.pyx
68a69
> from sage.cpython.wrapperdescr cimport wrapperdescr_fastcall
2836,2853d2836
<             except TypeError:
<                 pass
<
<         # Try to coerce denominator in numerator parent...
<         if isinstance(right, Polynomial):
<             R = (<Polynomial>right)._parent
<             try:
<                 x = R.coerce(left)
<                 return x.__truediv__(right)
<             except TypeError:
<                 pass
<
<         # ...and numerator in denominator parent
<         if isinstance(left, Polynomial):
<             R = (<Polynomial>left)._parent
<             try:
<                 x = R.coerce(right)
<                 return left.__truediv__(x)
2857c2840,2844
<         return NotImplemented
---
>         # Delegate to coercion model. The line below is basically
>         # RingElement.__truediv__(left, right), except that it also
>         # works if left is not of type RingElement.
>         return wrapperdescr_fastcall(RingElement.__truediv__,
>                 left, (right,), <object>NULL)
5463c5450
<     def gcd(self, other):
---
>     def gcd(self, other, algorithm='pari'):
5469a5457
```

```
>         - ``algorithm='pari'`` -- the algorithm used for computing gcd of two polynomials, should be 'pari' or 'generic'
5480,5482c5468,5476
<             on the base ring underlying the polynomial ring. If the base ring
<             defines a method :meth:`_gcd_univariate_polynomial`, then this method
<             will be called (see examples below).
---
>             on the base ring underlying the polynomial ring. If the algorithm is
>             set to default 'pari', it will try the PARI implementation. Otherwise
>             algorithm should be set to 'generic' and in this case, if will be checked
>             if the base ring defines a method :meth:`_gcd_univariate_polynomial`,
>             then this method will be called (see examples below). Algorithm 'pari' has
>             half-gcd algorithm implemented in some cases, which is significantly faster
>             for high degree polynomials. Generally without half-gcd algorithm, it is
>             infeasible to calculate gcd/xgcd of two degree 50000 polynomials in a minute
>             but TEST shows it is doable with algorithm 'pari'.
5538a5533,5540
>
>             sage: P.<x> = PolynomialRing(Zmod(4294967311 * 4294967357), implementation='NTL')
>             sage: degree = 50000
>             sage: g_deg = 10000
>             sage: g = P.random_element(g_deg).monic()
>             sage: a, b = P.random_element(degree), P.random_element(degree)
>             sage: (a * g).gcd(b * g) == g
>             True
5549a5552,5563
>
>         if algorithm not in ("pari", "generic"):
>             raise ValueError(f"unknown implementation {algorithm!r} for gcd function over {self.parent()}")
>         if algorithm == 'pari':
>             P = self.parent()
>             g = self._pari_with_name().gcd(other._pari_with_name())
>             g = P(g)
>             c = g.leading_coefficient()
>             if c != P(0):
>                 g /= c
>             return g
>
6288,6304d6301
<     def canonical_associate(self):
<         """
<         Return a canonical associate.
<
<         EXAMPLES::
<
<             sage: R.<x>=QQ[]
<             sage: (-2*x^2+3*x+5).canonical_associate()
<             (x^2 - 3/2*x - 5/2, -2)
<             sage: R.<x>=ZZ[]
<             sage: (-2*x^2+3*x+5).canonical_associate()
<             (2*x^2 - 3*x - 5, -1)
<         """
<         lc = self.leading_coefficient()
<         n, u = lc.canonical_associate()
<         return (u.inverse_of_unit() * self, u)
<
9704c9701
<     def xgcd(self, other):
---
>     def xgcd(self, other, algorithm='pari'):
9710a9708
>         - ``algorithm='pari'`` -- the algorithm used for computing xgcd of two polynomials, should be 'pari' or 'generic'
9720,9723c9718,9727
<             The actual algorithm for computing the extended gcd depends on the
<             base ring underlying the polynomial ring. If the base ring defines
<             a method :meth:`_xgcd_univariate_polynomial`, then this method will be
<             called (see examples below).
---
>             The actual algorithm for computing greatest common divisors depends
>             on the base ring underlying the polynomial ring. If the algorithm is
>             set to default 'pari', it will try the PARI implementation. Otherwise
>             algorithm should be set to 'generic' and in this case, if will be checked
>             if the base ring defines a method :meth:`_xgcd_univariate_polynomial`,
>             then this method will be called (see examples below). Algorithm 'pari' has
>             half-gcd algorithm implemented in some cases, which is significantly faster
>             for high degree polynomials. Generally without half-gcd algorithm, it is
>             infeasible to calculate gcd/xgcd of two degree 50000 polynomials in a minute
>             but TEST shows it is doable with algorithm 'pari'.
9757a9762,9772
>
>         TESTS::
>
>             sage: P.<x> = PolynomialRing(Zmod(4294967311 * 4294967357), implementation='NTL')
>             sage: degree = 50000
>             sage: g_deg = 10000
>             sage: g = P.random_element(g_deg).monic()
>             sage: a, b = P.random_element(degree), P.random_element(degree)
>             sage: r, s, t = a.xgcd(b)
>             sage: (r == a.gcd(b)) and (r == s * a + t * b)
>             True
9758a9774,9787
>         if algorithm not in ("pari", "generic"):
>             raise ValueError(f"unknown implementation {algorithm!r} for xgcd function over {self.parent()}")
>         if algorithm == 'pari':
>             P = self.parent()
>             s, t, r = self._pari_with_name().gcdext(other._pari_with_name())
>             s = P(s)
>             t = P(t)
>             r = P(r)
>             c = r.leading_coefficient()
>             if c != P(0):
>                 s /= c
>                 t /= c
>                 r /= c
>             return r, s, t
```

```
10996c11025
<     def has_cyclotomic_factor(self) -> bool:
---
>     def has_cyclotomic_factor(self):
```

7.5 One Gaussian elemination for iterating all roots

```python
M = random_matrix(GF(7), 10, 15)
v = random_vector(GF(7), 10)

from itertools import product

def iterate_all(r, ker):
    l = ker.nrows()
    P = r.base_ring()

    for it in product(P, repeat=l):
        yield r + sum(a * b for a, b in zip(ker, it))

def all_roots_with_total(M, v):
    P = v.base_ring()

    ker = block_matrix([[-v.column(), M]]).right_kernel_matrix()

    if ker[0, 0] == P(0):
        return iter([]), 0

    assert ker[0, 0] == P(1)
    r, ker = ker[0][1:], ker[1:, 1:]

    l = ker.nrows()

    tot = P.order()^l

    return iterate_all(r, ker), tot

it, total = all_roots_with_total(M, v)
for i in range(total):
    assert M * next(it) == v
```

7.6 ZKPoF, server.py

```python
#!/usr/bin/env python3
from Crypto.Util.number import getPrime, getRandomRange
from math import floor
import json, random, os

# https://www.di.ens.fr/~stern/data/St84.pdf
A = 2**1000
B = 2**80


def keygen():
    p = getPrime(512)
    q = getPrime(512)
    n = p * q
    phi = (p - 1) * (q - 1)
    return n, phi


def zkpof(z, n, phi):
    # I act as the prover
    r = getRandomRange(0, A)
    x = pow(z, r, n)
    e = int(input("e = "))
    if e >= B:
        raise ValueError("e too large")
    y = r + (n - phi) * e
    transcript = {"x": x, "e": e, "y": y}
    return json.dumps(transcript)


def zkpof_reverse(z, n):
    # You act as the prover
    x = int(input("x = "))
    e = getRandomRange(0, B)
    print(f"{e = }")
    y = int(input("y = "))
    transcript = {"x": x, "e": e, "y": y}
    return json.dumps(transcript)


def zkpof_verify(z, t, n):
    transcript = json.loads(t)
    x, e, y = [transcript[k] for k in ("x", "e", "y")]
    return 0 <= y < A and pow(z, y - n * e, n) == x


if __name__ == "__main__":
    n, phi = keygen()
    print(f"{n = }")

    rand = random.Random(1337)  # public, fixed generator for z
    for _ in range(0x137):
        try:
            z = rand.randrange(2, n)
            t = zkpof(z, n, phi)
            assert zkpof_verify(z, t, n)
            print(t)
            if input("Still not convined? [y/n] ").lower()[0] != "y":
                break
        except Exception as e:
            print(f"Error: {e}")
    print(
        "You should now be convinced that I know the factorization of n without revealing anything about it. Right?"
    )
    for _ in range(floor(13.37)):
        z = rand.randrange(2, n)
        t = zkpof_reverse(z, n)
        assert zkpof_verify(z, t, n)
        print(t)
    print(os.environ.get("FLAG", "flag{test}"))
```

7.7 Diff of file sage/repl/preparser.py

```

$ diff original-preparse.py new-preparse.py
1127,1128c1127,1129
<         arguments to RealNumber and ComplexNumber. If ``None``, will rebuild
<         the string using a list of its Unicode code-points.
---
>         arguments to RealNumber and ComplexNumber, and Integer when the
>         number is longer than 4300 digits. If ``None``, will rebuild the
>         string using a list of its Unicode code-points.
1190a1192,1200
>
>     Check that :issue:`40179` is fixed::
>
```

```
>         sage: preparse_numeric_literals("1" * 4300) == f"Integer({'1' * 4300})"
>         True
>         sage: preparse_numeric_literals("1" * 4301) == f"Integer('{'1' * 4301}')"
>         True
>         sage: preparse_numeric_literals("1" * 4301, quotes=None) == f'Integer(str().join(map(chr, {[49] * 4301})))'
>         True
1327c1337,1343
<                 num_make = "Integer(%s)" % num
---
>                 if len(num) <= 4300:
>                     num_make = "Integer(%s)" % num
>                 elif quotes:
>                     num_make = "Integer(%s%s%s)" % (quotes, num, quotes)
>                 else:
>                     code_points = list(map(ord, list(num)))
>                     num_make = "Integer(str().join(map(chr, %s)))" % code_points
2218c2234
<     return tmpfilename
\ No newline at end of file
---
>     return tmpfilename
```