# Quadratic Sieve Algorithm

a.k.a. the Second Fastest Integer Factoring Algorithm in the West

(and trying to get it to work)

Nils Olsson

Spring Semester, 2021

## Abstract

This document serves as a summary of the history and mathematical theory behind square factoring functions, as an loose introduction to the Python and Rust programming languages, and as a retrospective on implementing the quadratic sieve (QS) algorithm in these languages.

# Chapter 1

# Introduction

The factorization of integers has been an especially well studied problem, and not just for centuries but in fact millennia. Greek mathematicians studied the problem, even proving the fundamental theorem of arithmetic: that every integer has a unique prime factorization. As a corollary the Greeks had also extensively studied the prime integers, with the sieve of Eratosthenes being one of the most important algorithms for generating primes ever invented. The so called father of geometry Euclid of Alexandria (whose now self-titled "Euclid's lemma" was foundational for proving integer factorization uniqueness) described in his *Elements* the Euclidean algorithm for calculating the greatest common divisor of two integers. These ancient algorithms, and many others, comprise the fundamental building blocks for which all modern integer factorization algorithms are made.

# Chapter 2

# Square Factoring

# Chapter 3

# The Quadratic Sieve

# Chapter 4

# Implementation

## 4.1 Implementing in Python

## 4.2 Implementing in Rust

With the fundamental algorithm completed in Python, I made the move to re-implementing in the Rust language. This section is primarily a retrospective on my experience in implementing *(or at least attempting to implement)* the QS algorithm in Rust. I intend for it to be useful to anyone with an interest in Rust, as well as useful to myself in both reviewing my own knowledge of the language and providing a figurative map of my thoughts on the design of my project's implementation. I go over the Rust language in general, how its features have been very useful for implementing mathematical structures, and weave in how I used these features in my own QS implementation. If heavy use of programming jargon is not your thing then this section may not be very interesting.

### 4.2.1 Overview of Rust

As far as fundamental paradigms go, Rust as a programming language has much in common with more classically object oriented languages like C++ or Java, but also a lot in common with purely function languages like Haskell. Similarities with the later are in Rust's sophisticated *trait* feature, analogous to Haskell's *type classes*, with which Rust acomplishes polymorphism. In C++, polymorphism in achieved through hierarchies of `classes` and abstract `classes` (i.e. interfaces) (e.g. if a `Square` derives from `Shape`, then a function which accepts a `Shape` can accept a `Square`). Although Rust encapsulates data and functionality through `structs` (analogous to C++ `classes`) and their instantiations, polymorphism in Rust is achieved *only* through its traits (e.g. if `Shape` is a trait and `Square` implements the `Square` trait, then a function that requires its argument implement `Shape` can accept a `Square`).

The most substantial difference in polymorphism is that there is *no such thing* as inheritance for Rust `structs` like there is for C++ `classes`. (e.g. in C++, `Square` as a `class` is a subclass of an abstract class `Shape`; in Rust, `Square` *implements* `Shape`, where `Square` is a `struct` and `Shape` is a `trait`. Further more, *if* `Shape` *was a* `struct` *instead of a* `trait`, `Square` could not inherent any of `Shape`'s functionality and neither would share any functionality—as far as the compiler is concerned—without them both implementing some other trait.) This is a major paradigm shift for anyone moving from C++ to Rust, as inheritance based polymorphism is more-or-less the only flavor of polymorphism that students of computer science/programming are likely to have learned (as was the case for me especially), and one might believe on a first glance that if inheritance is not a feature then Rust `structs` cannot share functionality without copying entire swathes of code.

Instead, Rust `traits` fill this purpose: a `trait` can define default functionality for which any `struct` which implements the trait inherits. For example, the code in listing 1 defines a trait `Pow` representing the capability of the implementing type to be “raised to a power,” for which any type can implement. (A note on the language: an `impl` block *implements* functionality, i.e. one or several functions, onto an existing type. These can be *methods* for instances of the type, like `a.foo()`, or *functions* belonging to the type itself, like `Foo::foo()`.)

```
trait Pow {
    /// Raise this value to an exponent.
    /// (Must be implemented)
    fn pow(&self, e: u32) -> Self;

    /// Square this value.
    /// (Implemented by default, and does *not* need to be re-implemented)
    fn squared(&self) -> Self {
        self.pow(2)
    }
}

// Although we don't implement the `squared` method for `i32` here,
// it gets the default implementation from the `Pow` trait.
impl Pow for i32 {
    fn pow(&self, e: u32) -> i32 {
        <i32>::pow(e)
    }
    // fn squared... is inherited.
}
```

Listing 1: A trait with a default implementation.

The fundamental advantage of traits used in this way is that if we then write a function in which we need the functionality of `pow`, then we can require that whatever type(s) the function takes as parameters *implement the* `Pow` *trait*; furthermore, anyone is allowed to implement `Pow` trait, and thus use our function with their own `Pow` implementing type. In other words, polymorphism in Rust is achieved by abstracting over the functionality that types have, and not the types themselves. We call these functionality requirements on types *trait bounds*.

Another advantage of trait bounds is that they allow us to add functionality to a type *incrementally* by requiring increasingly stricter trait bounds for more functionality dependent operations. I bring up an example that I will use continuously from now on: my implementation of $M \times N$ matrices, `Matrix`. At its most generic, a matrix can be just a two-dimensional storage location in which we could potentially store anything. As such, in listing 2 I define `Matrix` without any bounds on what it is allowed to store aside from the type needing to be clonable.

```
#[derive(Clone)]
pub struct Matrix<T, const M: usize, const N: usize>
where
    T: Clone,
{
    pub rows: [[T; N]; M], // M arrays of T arrays of length N
}
```

Listing 2: The verbatim definition of my own matrix type.

In this snippet, `T` is called a generic parameter, and it is given the trait bound `Clone`; thus we are allowed to make a matrix of type `T` for any type `T` that is clonable. Personally, I find something inherently mathematical about the way in which we can restrict generic types in Rust. This leads me to a particular implementation (listing 3) for my `Matrix` `struct`: the zero matrix.

```
impl<T, const M: usize, const N: usize> Matrix<T, M, N>
where
    T: Zero<Element = T> + Clone,
{
    pub fn zeroes() → Self {
        // T::ZERO is the constant value that represents
        // zero for the generic type parameter T.
        let rows = array_init(|_| array_init(|_| T::ZERO));
        Self::from_array(rows)
    }
}
```

Listing 3: Implementation to construct a $M \times N$ all-zero matrix.

In this snippet we define the construction of a zero matrix of specified size $M \times N$ (but heed not the `const M` and `const N`, we cover that in a few paragraphs from here). This begs the question: *how do we know that the type* `T`*, for which the matrix is over, has a representation of zero at all?* For the generic type `T`, the only way then for the compiler to know that `T` has zero is to impose an additional trait bound, in this case a trait of my own called `Zero`. The only thing that this trait requires is that an implementing type provide a constant representation of zero, and with this we have a value with which to fill a zero matrix. Additionally, note the `Element = T` part of the trait bound: this requires that not only does `T` need to have a zero element, but that the zero element *must be of the same type as* `T` *itself*. This allows for a distinction when, for example, something like a set of numbers has a zero element, but the zero element is not itself a set (think of the quotient group $\mathbb{Z}_n$ having the congruence class $[0]_n$ as its zero element). (See listing 4 for the implementation of `Zero` on `i32` integers and on my own `CongruenceClass` `struct`; I cover the later in more detail next).

```
pub trait Zero {
    type Element;

    const ZERO: Self::Element;
}

pub trait One {
    type Element;

    const ONE: Self::Element;
}

impl Zero for i32 {
    type Element = i32;

    const ZERO: i32 = 0;
}

impl<const M: u32> Zero for QuotientGroup<M> {
    type Element = CongruenceClass<M>;

    const ZERO: CongruenceClass<M> = CongruenceClass<M>(0);
}
```

Listing 4: Definition of the zero and one traits, and the implementation of zero for the 32-bit integer type, and for my own quotient group type.

A highly experimental feature of Rust that I only discovered since having started this project is `const generics`. This feature has proven to be incredibly useful in the implementation of various mathematical structures, in particular congruence classes (see listing 5) and matrices.

```
/// Congruence class with modulo M.
///
/// # Examples
/// We can add congruence classes if they have the same modulus:
/// ```
/// let a = CongruenceClass::<5>(1); // 1 modulo 5
/// let b = CongruenceClass::<5>(4); // 4 modulo 5
/// 1 + 4 is congruent to 0 modulo 5:
/// assert_eq!(a + b, CongruenceClass::<5>(0));
/// ```
/// But not if they have different modulus:
/// ```skip
/// let a = CongruenceClass::<5>(1); // 1 modulo 5
/// let b = CongruenceClass::<6>(5); // 5 modulo 6
/// a + b; // fails to compile, since 5 ≠ 6
/// ```
pub struct CongruenceClass<const M: u32>(pub u32);

// `Add` implies `Add<CongruenceClass<M>>`, and
// `Self` refers to `CongruenceClass<M>`
impl<const M: u32> Add for CongruenceClass<M> {
    type Output = Self;

    fn add(self, rhs: Self) → Self {
        Self((self.0 + rhs.0) % M)
    }
}
```

Listing 5: Excerpt of my implementation of congruence classes in Rust, using traits and the const generic feature.

What const generics do is bake a constant value (though currently only primitive types like u32 are supported) directly into a `trait` or `struct`. This allows the compiler to know at compile time (or more specifically the type checker and type check time) whether or not, say, two instances of a `struct` are compatible with one-another based on the constants that are baked into their types; because although a may have to instances of `CongruenceClass`, their types are not fully qualified without the constant modulus.

Another instance in which this feature was useful was in the defining of my matrix `struct`. What is amazing about the const generic feature in this case is that the validity checking of operations like matrix addition and multiplication no longer need to take place a runtime, and instead are *completely validated at compile time*! (See the implementation of addition and multiplication in listing 6 below.)

```
// Implement same size matrix addition
impl<T, const M: usize, const N: usize> Add for Matrix<T, M, N> {
    type Output = Self;
```

```
    fn add(self, rhs: Self) → Self { /* ... */ }
}

// Implement MxN and NxP → MxP matrix multiplication
//
// With the use of const generics we can ensure at compile time that matrices
// that we multiply must have the same number of columns on the left as rows
// on the right.
impl<T, const M: usize, const N: usize, const P: usize> Mul for Matrix<T, M, N>
where
    T: Add<Output = T> + Mul<Output = T> + Copy,
{
    type Output = Matrix<T, M, P>;

    fn mul(self, rhs: Matrix<T, N, P>) → Matrix<T, M, P> { /* ... */ }
}
```

Listing 6: Excerpt of my implementation of matrices in Rust, using traits and the const generic feature.

Checking validity before the program is ever ran allows us to guarantee the success of such operations at runtime, eliminating the perhaps drastic amount of time spent checking whether matrices have compatible shapes at runtime otherwise.

Up to this point, with these tools (and with implementing a few other helper algorithms) I was able to implement the quadratic sieving algorithm itself (that is, finding $B + 1$ numbers that are smooth over the factor base, guaranteeing linear dependence with their binary exponent vectors), able to implement row-reduction to Echelon form with my own `Matrix`, and able to find all linearly dependent subsets of the exponent vectors using spanning vectors of the left nullspace. *However*, this is where my story for the time being turns sour, as I am currently *not* able to complete my implementation of the QS algorithm based on the fact that my matrix `struct` requires knowing the number of rows $M$ and columns $N$ *at compile time*. Depending on the number to factor $n$, there is no way for my program to deterministically know at compile time how large the factor base needs to be (controlling the number of columns of the matrix) nor how many smooth numbers it finds (controlling the number of rows).