Crowley-Kugler-Anderson Implementation of the Quadratic Sieve

Pete Crowley, Jack Kugler, William Anderson

Duke University

Math 404: Mathematical Cryptography

January 16, 2026

1 Introduction and Background

The difficulty of factoring underlies many modern cryptosystems. RSA is a prominent example. For numbers with more than 120 digits, factorization remains intractable for classical computers. However, algorithms have been developed that can drastically improve the efficiency of factoring over the naïve method of trial division. The subject of this project is the quadratic sieve, an example of such an algorithm over the integers.

As of 2025, the quadratic sieve is viewed as the second fastest way to factor, after the general number field sieve for values less than or equal to 10^{99}. It has also been stated to be the preferred algorithm for factoring composite numbers ranging from 10^{19} to 10^{119} that have no factors that can be extracted efficiently by trial division or using elliptic curves. It was discovered by Carl Pomerance in 1981. [1]

The quadratic sieve algorithm is based on the fundamental idea that when $x^2 \equiv y^2 \pmod{n}$, but x is not congruent to $\pm y \pmod{n}$, then $\gcd(x-y, n)$ is a non-trivial factor of n. This derives from rearranging the congruence relation and using difference of squares to write that $(x+y)(x-y) \equiv 0 \pmod{m}$, but since neither $x+y$ nor $x-y$ is a multiple of n, both these factors are neither n or 1. So one might believe that simply searching for numbers that are congruent to squares modulo n (also known as Fermat's Method) will suffice, but for large n, they are excruciatingly difficult to find, even for computers, so a finer approach is necessary, as will be discussed. [4]

The following description is the foundation of the quadratic sieve. One can begin by choosing a factor base, namely, the set of numbers less than n whose prime factorizations only contain primes up to a certain value. So if our base is 5099, for example, we only take numbers that factor as primes of 5099 or lower. This is also sometimes called the "smoothness bound," so this set of numbers in the example here is "5099-smooth." Our next step means taking $x \pmod{n}$, then squaring for $x^2 \pmod{n}$. As we run through x, for $1 \leq x \leq n$, we only keep the x such that x^2 has 5099-smooth factors. The process of determining adequate values of x that factor into values less than the smoothness bound is often achieved using a quadratic sieve, the namesake of the algorithm. This is discussed in more detail in our implementation of the algorithm. One examines the various quadratic residues of $x^2 \pmod{n}$, then searches for subsets of x, that when multiplied together, form a square modulo n. So if we have that $x^2 \equiv 2 \cdot 3 \cdot 5 \pmod{n}$, $y^2 \equiv 2 \cdot 3^2 \cdot 7 \pmod{n}$, and $z^2 \equiv 3 \cdot 5 \cdot 7 \cdot 11^2 \pmod{n}$, we observe that $(xyz)^2 \equiv 2 \cdot 3 \cdot 5 \cdot 2 \cdot 3^2 \cdot 7 \cdot 3 \cdot 5 \cdot 7 \cdot 11^2 \pmod{n}$, which by rearrangement, yields $(xyz)^2 \equiv 2^2 \cdot 3^4 \cdot 5^2 \cdot 7^2 \cdot 11^2 \pmod{n}$, or $(xyz)^2 \equiv (2 \cdot 9 \cdot 5 \cdot 7 \cdot 11)^2 \pmod{n}$. In the case that xyz is not congruent to $2 \cdot 9 \cdot 5 \cdot 7 \cdot 11 \equiv 6930 \pmod{n}$ or $-2 \cdot 9 \cdot 5 \cdot 7 \cdot 11 \equiv -6930 \pmod{n}$, then one has found what they sought. Note that this example might not hold for all n, because out of all numbers x where $0 \leq x \leq n-1 \pmod{n}$, not all of them will necessarily end up as quadratic residues modulo n. So the aforementioned example was theoretical. Of course, when we have a large number of different factorizations to deal with, a more efficient method of finding combinations that yield squares is necessary. Fortunately, this can be resolved relatively easily by representing the factorizations as vectors in a matrix, where the entries are the powers of each prime modulo 2. So, returning to the previous example, x^2 is represented as $(1, 1, 1, 0, 0, \ldots)$ where 1 represents the power of 2, 3, and 5, and 0 represents the power of all the other primes up to 5099. y^2 is represented as $(1, 0, 0, 1, 0, 0, \ldots)$ since this time there is no 5 factor and now 3 is squared, and $2 \equiv 0 \pmod{2}$. So we have a matrix where its row dimension is the number of x, $0 \leq x \leq n-1$ that are B-smooth, and its column dimension is the number of primes up to B. One must find a linear combination of the vectors that yields $\vec{0}$ (the 0 vector), namely, a linear dependence. This is trivially done via Gaussian elimination. If the number of vectors exceeds the number of primes up to B, one is guaranteed a linear dependence. So now, having found $(xyz)^2 \equiv 6930^2 \pmod{n}$, it holds that $n > \gcd(n, xyz - 6930) > 1$. From here, the Euclidean Algorithm can be used to factor n rapidly. Note that if xyz is actually congruent to $\pm 6930 \pmod{n}$, then we end up with a trivial factor, but even if this occurs, another linear dependence can be found. Now, with a probably significantly smaller number (at least an order of magnitude, or several, since presumably $2, 3, 5, 7$ can be easily ruled out), one can continue in the same fashion until only a prime number remains, thereby completely factoring n. This example outlines the fundamental steps guiding the quadratic sieve. [3]

2 Crowley-Kugler Anderson Implementation

2.1 Overview of the Implementation

We implemented the quadratic sieve in Python to take advantage of its numerous libraries and readable syntax. We take advantage of parallel computations running simultaneously on different CPUs of a multicore machine during the sieving process for large primes. The code for the algorithm is included in Appendix A and a sample output in Appendix B.

Consider attempting to factor a large integer N that is the product of two distinct primes. Our implementation can broadly be broken down into the following steps.

1. Choose a factor base bound B.
2. Find the primes p less than the bound such that N is a quadratic residue $\bmod p$ to use as the factor base.
3. Find squares that are B-smooth, meaning that their factorizations $\bmod N$ are products of primes in the factor base.
4. Create a matrix where each row consists of the exponents $\pmod 2$ of the primes in the factor base for one factorization of a B-smooth square.
5. Find a basis vector in the null space of the matrix and take the product of the specified factorizations.
6. Use corresponding modular equivalence of the form $x^2 \equiv y^2 \pmod N$. If $x \equiv \pm y \pmod N$, try another basis vector in the null space. If each basis vector in the null space is exhausted, increase the factor bound B by squaring and repeat from step 2.
7. If $x \not\equiv \pm y \pmod N$, then find $\gcd(N, x - y)$ to give a factor of N. Divide N by the factor to obtain the other factor.

2.2 Determining the Factor Base Bound

To determine a preliminary value for the factor base bound, we use a theoretical bound given by Crandall and Pomerance. [2] This bound is

$$B = \lceil \sqrt{e^{\sqrt{\ln N \cdot \ln \ln N}}} \rceil.$$

Experimental testing with various bounds yielded that this theoretical result yielded fast factorizations.

2.3 Find Primes Such that N is a Quadratic Residue

We wish to find primes $p \leq B$ such that N is a quadratic residue mod p. By Euler's Criterion, we know that $N^{\frac{p-1}{2}} \equiv 1 \pmod{p}$ if and only if N is a quadratic residue mod p. Therefore, for all primes less than the factor bound, we check if $N^{\frac{p-1}{2}} \equiv 1 \pmod{p}$. If so, we can include such a prime in the factor base.

2.4 Find B-smooth squares

We now desire to find values of x such that $x^2 \pmod{n}$ factors as a product of primes in the factor base. To do this, we employ the quadratic sieve that serves as the namesake of the algorithm. We wish to find values of x such that

$$x^2 - N \equiv 0 \pmod{p}.$$

This justifies why we selected p in the factor base such that N is a quadratic residue mod p. We sieve values around $\lceil \sqrt{N} \rceil$. We introduce a parameter sieve_interval_size. On each pass through the sieve, we sieve $2 *$ sieve_interval_size values, with half of the values being less than $\lceil \sqrt{N} \rceil$ and half of the values being greater than $\lceil \sqrt{N} \rceil$. We begin by sieving values close to $\lceil \sqrt{N} \rceil$ and move farther in either direction during subsequent passes in the sieve. We maintain an array with indices corresponding to all elements being sieved. For each p in the factor base, we find $\sqrt{N} \pmod{p}$, which exists because we selected p to have N as a quadratic residue. For the values of x being sieved such that $x \equiv \sqrt{N} \pmod{p}$, we add $\ln p$ to the index in the array corresponding to x. Once we have repeated this process for all primes in the factor base, we check if the value in the array corresponding to x is within a experimentally determined SIEVE_CUSHION of the natural logarithm for $x^2 - n$, the sieve polynomial. If so, we attempt to factor the value through trial division. Being within the sieve cushion of the sieve polynomial indicates that the product of found primes is close to $x^2 - n$.

Literature suggests that it is sufficient to only sieve for primes rather than prime powers and that one does not need to sieve for small primes. [2] These are two crucial optimization of our algorithm. We leave finding factorization by small primes (we skip the first 5 primes, a value captured in the parameter PRIMES_TO_SKIP) to the trial division component of the code. When performing the sieving process, we only require that the sum of the natural logarithm of the prime divisors that we sieved by to be within SIEVE_CUSHION of the natural logarithm of $x^2 - n$. This tradeoff means that we may trial divide some values that cannot be factored by numbers in the factor base, while capturing some values that our sieve misses.

2.5 Create Matrix of Factorizations

We perform the sieving process until we have obtained a minimum of EXTRA_VECTORS more factorizations than we have elements in our factorization base. If we exhaust all values $0 < x \leq 2\sqrt{N}$ to be sieved and still do not have EXTRA_VECTORS more factorizations, we simply return the found factorizations. EXTRA_VECTORS was experimentally optimized to be 10 in our code. We turn each factorization into a row vector where each component corresponds to the exponent on the power of a factor base element in the factorization mod 2. These row vectors are combined to form a matrix consisting of elements in $\mathbb{Z}/2\mathbb{Z}$.

2.6 Find Linear Combination

We now attempt to find the null space of the matrix. The null space is guaranteed to be non-empty if the number of factorizations exceeds the number of elements in the factor base. We select a basis vector of the null space and take the product of the congruences identified by the null space matrix. We thus obtain an equation of the form

$$x^2 \equiv y^2 \pmod{N}$$

where y is a product of primes in the factor base. We subsequently check if $x \equiv \pm y \pmod{N}$. If this is a case, we take another basis vector in the null space and try again. Once we find x and y such that $x^2 \equiv y^2 \pmod{N}$ and $x \not\equiv \pm y \pmod{N}$, we can take $\gcd(x - y, N)$ to find a nontrivial factor of N and thus factor N. If we exhaust all vectors in the null space without finding such a pair, we double the factor base and repeat the sieving process.

3 Potential Further Work

Our algorithm already implements numerous optimizations to give strong performance to the quadratic sieve algorithm. However, further tools could be used to further increase efficiency. One possible way to increase efficiency is to experiment with sieving prime powers in addition to primes and to further tune experimental parameters such as SIEVE_CUSHION, sieve_interval_size, EXTRA_VECTORS, B and PRIMES_TO_SKIP depending on the size of the value being factored.

4 Conclusion

We have implemented the quadratic sieve algorithm in Python and used optimizations in hardware and software to achieve high efficiency. From a hardware perspective, we utilize each CPU of a multicore machine to parallelize the process of the quadratic sieve. Each CPU checks 2*sieve_interval_size values at varying distance from the center value of $\lceil N \rceil$. This allows for one to attain many more factorizations in parallel than is possible on strictly one CPU. From a software perspective, we experimented with various parameters to maximize the efficiency of the program. Experimentation was conducted using a test suite that generated large values that were the product of two primes and measured the time necessary to factor. These parameters include the size of the base bound of the factor, the number of columns in excess of the number of rows in the factorization matrix, and the size of the interval to be sieved before checking the factorizations by trial division. Tuning these parameters represented significant advances in speed. Further optimizing the parameters of the algorithm to incorporate the size of N the number to be factored would enhance our approach.

References

[1] Wikipedia contributors. Quadratic sieve.

[2] Richard Crandall and Carl Pomerance. *Prime Numbers: A Computational Perspective*. Springer, 2 edition, 2005.

[3] Carl Pomerance. Smooth numbers and the quadratic sieve. *MSRI Publications*, 44:69–80, 2008.

[4] Wade Trappe and Lawrence Washington. *Introduction to Cryptography with Coding Theory*. Pearson Prentice Hall, Upper Saddle River, NJ 07458, 2 edition, 2006.

5 Appendix A

```
import numpy as np
from sympy.ntheory import factorint
from sympy import primerange, sqrt_mod
import sys
import time
from galois import GF2
import math
from concurrent.futures import ProcessPoolExecutor, as_completed
import os



EXTRA_VECTORS = 10
SIEVE_INTERVAL_SIZE = 100_000
PRIMES_TO_SKIP = 5
SIEVE_CUSHION = 20
PARALLEL = True
UPDATE_INTERVAL = 100




def choose_factor_base_bound(N) -> int:
    """Chooses the bound for the factor base using the formula
       from the book"""
    L = math.exp(math.sqrt(math.log(N) * math.log(math.log(N))))
    B = L ** (1/2)
    return math.ceil(B)
```

```
def is_quadratic_residue(N, p):
    """Returns True if N is a square mod p, False otherwise.


        :param N: The number to check
        :param p: The prime to check against
        :returns boolean: True if N is a square mod p, False
            otherwise
    """
    if pow(N, (p - 1) // 2, p) == 1:
        return True
    return False



def build_smart_factor_base(N, B) -> list:
    """Builds the factor base of primes less than B that are
       quadratic residues mod N


        :param N: The number to factor
        :param B: The bound for the factor base
        :returns factor_base: A list of primes that are
            quadratic residues mod N
    """
    factor_base = [-1]
    for p in primerange(2, B+1):
        if is_quadratic_residue(N, p):
            factor_base.append(p)
    return factor_base
```

```
def b_smooth_factor(x: int, factor_base: list) -> dict | None:
    """Trial division to check if x is B-Smooth and if so to
       find factorization


         :param x: The number to factor
         :param factor_base: The factor_base
         :returns powers: A dictionary of the form {p: exponent}
             if x is B-smooth, None otherwise
    """
    powers = dict()
    if x < 0:
        powers[-1] = 1
        x = -x
    for p in factor_base[1:]:
        while x % p == 0:
            if p not in powers:
                powers[p] = 1
            else:
                powers[p] += 1
            x //= p
            if x == 1:
                return powers
    if x != 1:
        return None
    return powers


def process_one_interval(N, B, factor_base, dict_factor_base,
   log_factor_base, iteration, sieve_interval_size):
```

```
"""The exact same functionality as the
   better_find_b_smooth_squares function, but parallelized.
   Check the docstring for that function for more
      information"""
def f(x):
    return x**2 - N

sieve_interval_center = math.ceil(math.sqrt(N))
small_start = sieve_interval_center - (iteration+1) *
   sieve_interval_size
large_start = sieve_interval_center + iteration *
   sieve_interval_size
smaller_registers = np.zeros(sieve_interval_size)
larger_registers = np.zeros(sieve_interval_size)

for p in factor_base[PRIMES_TO_SKIP:]:
    sols = sqrt_mod(N, p, all_roots=True)
    for sol in sols:
        for i in range((sol-large_start) % p,
           sieve_interval_size, p):
            larger_registers[i] +=
               log_factor_base[dict_factor_base[p]]
        for i in range((sol-small_start) % p,
           sieve_interval_size, p):
            smaller_registers[i] +=
               log_factor_base[dict_factor_base[p]]

results = []
for i in range(0, sieve_interval_size):
        for start, regs in [(small_start, smaller_registers),
            (large_start, larger_registers)]:
             x = i + start
             if regs[i] >= math.log(abs(f(x))) - SIEVE_CUSHION:
                 factors = b_smooth_factor(f(x), factor_base)
                 if factors is not None:
                     results.append((x, factors))
    return results

def parallel_find_b_smooth_squares(N, B, factor_base,
        dict_factor_base, verbose) -> list:
    """Just a parallel version of the
        better_find_b_smooth_squares function which
        processes each interval in parallel. Functionality is
            the same as better_find_b_smooth_squares.
        Check the docstring for that function for more
            information"""
    log_factor_base = [math.log(p) if p != -1 else 0 for p in
        factor_base]
    nums = []
    factorizations = []
    sieve_interval_size = min(SIEVE_INTERVAL_SIZE,
        int(math.sqrt(N)) - 2)

    max_iters = (int(math.sqrt(N)) // sieve_interval_size) - 1
    vectors_needed = len(factor_base) + 1 + EXTRA_VECTORS

    start_number = os.cpu_count()   # let's just start with the
        number of iterations our computer can do in parallel
```

```
with ProcessPoolExecutor() as executor:
    futures = []
    # submit the first start_number iterations
    for i in range(0, start_number):
        futures.append(executor.submit(process_one_interval,
           N, B, factor_base, dict_factor_base,
           log_factor_base, i, sieve_interval_size))
    iteration = start_number


    while futures and len(nums) < vectors_needed:
        for future in as_completed(futures):     # as we
            complete iterations
             result = future.result()
             # add the results from the interval to our list
             for x, factors in result:
                 nums.append(x)
                 factorizations.append(factors)
                 if verbose and len(nums) % UPDATE_INTERVAL
                   == 0:
                         print(f"Found {len(nums)} of
                           {len(factor_base) + 1 +
                           EXTRA_VECTORS} needed B-smooth
                           numbers so far")

             futures.remove(future)        # remove the
                iteration we just finished
            # if we have more work to do, submit the next
                iteration
            if iteration < max_iters and len(nums) <
                vectors_needed:
                futures.append(executor.submit(process_one_interval,
                    N, B, factor_base, dict_factor_base,
                    log_factor_base, iteration,
                    sieve_interval_size))
                iteration += 1

            # if we have enough numbers, break
            if len(nums) >= vectors_needed:
                break

    return nums, factorizations



def better_find_b_smooth_squares(N, B, factor_base,
        dict_factor_base, verbose) -> list:
    """Finds B-smooth squares mod N using the quadratic sieve
        method


        :param N: The number to factor
        :param B: The bound for the factor base
        :param factor_base: The factor base
        :param dict_factor_base: A dictionary of the form {p:
            index} for the factor base
        :returns nums: A list of numbers that are B-smooth
            squares mod N
    :returns factorizations: A list of dictionaries of the
        form {p: exponent} for each number in nums
"""
# the polynomial we will sieve with
def f(x):
    return x**2 - N

log_factor_base  = [math.log(p) if p != -1 else 0 for p in
    factor_base] # take the log of each of the primes except
    -1
sieve_interval_size = min(SIEVE_INTERVAL_SIZE,
    int(math.sqrt(N)) - 2)    # don't want to try and sieve
    any negative numbers
sieve_interval_center = math.ceil(math.sqrt(N))
iteration = 0

nums = []                   # the numbers we will return (note
    we return the numbers themselves, not their squares mod
    N)
factorizations = []      # the factorizations of the numbers
    squared mod N

# we want enough numbers to guarantee a linear dependence
while len(nums) < len(factor_base) + 1 + EXTRA_VECTORS:
    # the true x value of the first x we will check in both
        the small and large intervals
    small_start = sieve_interval_center - (iteration+1) *
        sieve_interval_size
    large_start = sieve_interval_center + iteration *
    sieve_interval_size

smaller_registers = np.zeros(sieve_interval_size)      #
    for numbers below sqrt(N)
larger_registers = np.zeros(sieve_interval_size)       #
    for numbers above sqrt(N)

# for each prime in the factor base (we can skip the
    first few since their log is small), we will find
    the two solutions to x^2 = N mod p
for p in factor_base[1+PRIMES_TO_SKIP:]:
    sols = sqrt_mod(N, p, all_roots=True)    # find the
        two solutions (if they exist)
    for sol in sols:
        for i in range((sol-large_start) % p,
            sieve_interval_size, p):    # all numbers
            congruent to sol mod p are also solutions
            larger_registers[i] +=
                log_factor_base[dict_factor_base[p]]
                # so we add the log of the prime to the
                register

        for i in range((sol-small_start) % p,
            sieve_interval_size, p):    # same as above
            for the smaller registers
            smaller_registers[i] +=
                log_factor_base[dict_factor_base[p]]

# for numbers x whose sum of logs of prime divisors is
```

```
    close to the number, we check if they are B-smooth
for i in range(0, sieve_interval_size):
    for start, regs in [(small_start,
        smaller_registers), (large_start,
        larger_registers)]:      # check both the smaller
        and larger registers
        x = i + start
        if regs[i] >= math.log(abs(f(x))) -
            SIEVE_CUSHION:          # this formula is what
            we define as "close enough" in the textbook
            factors = b_smooth_factor(f(x),
                factor_base)        # try to factor by
                trial division
            if factors is not None:
                # if the number is B smooth, we will
                    add it to our list
                nums.append(x)
                factorizations.append(factors)
                if verbose and len(nums) %
                    UPDATE_INTERVAL == 0:
                    print(f"Found {len(nums)} of
                        {len(factor_base) + 1 +
                        EXTRA_VECTORS} needed B-smooth
                        numbers so far")

                if len(nums) >= len(factor_base) + 1 +
                    EXTRA_VECTORS:      # we have found
                    enough factors
                    break
        # on the next iteration, we will search further away
            from sqrt(N)
        iteration += 1
        if (iteration+1) * sieve_interval_size > math.sqrt(N):
            # we've exhausted [0, 2sqrt(N)]
            break
    return nums, factorizations    # if we didn't find enough
        numbers, we will just return what we have and maybe get
        lucky


def vectorize_factorizations(factorizations, dict_factor_base)
    -> np.array:
    """Vectorizes the factorizations into a matrix mod 2

        :param factorizations: A list of dictionaries of the
            form {p: exponent} for each number in nums
        :param dict_factor_base: A dictionary of the form {p:
            index} for the factor base
        :returns matrix: A matrix of size (len(factorizations),
            len(dict_factor_base)) mod 2
    """
    # Build the B+1 by B matrix mod 2
    matrix = [[0 for _ in dict_factor_base] for _ in
        factorizations]
    # for each equation
    for i, factors in enumerate(factorizations):
        # for each prime in the factorization
        for p, exponent in factors.items():
            matrix[i][dict_factor_base[p]] = exponent % 2 # set
                it to the exponent mod 2
    return matrix


def quadratic_sieve(N, verbose=False):
    """Quadratic Sieve implementation to factor N


        :param N: The number to factor
        :param verbose: If True, print debug information
        :returns factors: A tuple of the two factors if found,
            None if it can't find any"""
    found_factors = False
    B = choose_factor_base_bound(N)
    while not found_factors and B < math.sqrt(N):

        factor_base = list(build_smart_factor_base(N, B))
        if verbose:
            print(f"=== Chose factor base of {B} consisting of
                {len(factor_base)} primes===")
        dict_factor_base = {p: i for i, p in
            enumerate(factor_base)}

        if verbose:
            print("=== Searching for B-smooth squares ===")

        if PARALLEL:
            nums, factorizations =
                parallel_find_b_smooth_squares(N, B,
                factor_base, dict_factor_base, verbose)
        else:
            nums, factorizations =
                better_find_b_smooth_squares(N, B, factor_base,
                dict_factor_base, verbose)

        if nums == []:
            B *= 2
            continue

        vectorized_factorizations =
            vectorize_factorizations(factorizations,
            dict_factor_base)
        # Find nontrival solution of Ax = 0 mod 2
        if verbose:
            print("=== Finding linear dependence===")
        A = GF2(np.matrix(vectorized_factorizations).T)
        # find the null space of A
        null_space = A.null_space()

        if verbose:
            print("=== Trying to factor n ===")
        # any row vector will work as a linear combo that
            yields 0 mod 2
        for row in null_space:
            linear_combo = row
            # each 1 in this row represents a congruence we
                should include
            exponents_squared = [0 for _ in factor_base]
```

```
sqrt_number = 1
for i, included in enumerate(linear_combo):
    if included == 1:
        sqrt_number = (nums[i] * sqrt_number) % N
            # we actually returned the sqrt of x
            from b_smooth method so this is fine
        # add the exponents of the factorization to
            our list we will take the product of
        for p, exponent in
            factorizations[i].items():
              exponents_squared[dict_factor_base[p]]
                 += exponent
        nums.append(nums[i])

exponents = [e // 2 for e in exponents_squared] #
    we want the square root of the product of the
    p^e's
# so we just divide all the exponents by 2 and take
    the product
exp_product = 1
for p, exponent in dict_factor_base.items():
    exp_product = (exp_product * pow(p,
        exponents[exponent], N)) % N

# works if x != +- y mod n, otherwise we keep going
    and try a different linear combo
if exp_product != sqrt_number and exp_product !=
    -sqrt_number % N:
     # then a common factor is gcd(sqrt_number -
                    exp_product, N)
                one_factor = math.gcd(sqrt_number -
                    exp_product, N)
                other_factor = N // one_factor        # and
                    another is just dividing N by the first
                if verbose:
                    print("=== Found factors ===")
                    print(f"{N} = {one_factor} * 
                        {other_factor}")
                return one_factor, other_factor

        # if we didn't find a solution, let's double B and try
            again
        if verbose:
            print("=== No factors found, trying again with 
                larger B ===")
        B *=2

    return None

if __name__ == '__main__':
    # Number to factor
    N = int(sys.argv[1])
    time_start = time.time()
    quadratic_sieve(N, verbose=True)
    time_end = time.time()
    print("Time elapsed: ", round(time_end - time_start, 2),
        "seconds")
```

6 Appendix B

```
python3 quadratic_sieve.py
   374484308052961590901918151033055420550092602l947
=== Chose factor base of 97206 consisting of 4611 primes===
=== Searching for B-smooth squares ===
Found 100 of 4622 needed B-smooth numbers so far
Found 200 of 4622 needed B-smooth numbers so far
Found 300 of 4622 needed B-smooth numbers so far
Found 400 of 4622 needed B-smooth numbers so far
Found 500 of 4622 needed B-smooth numbers so far
Found 600 of 4622 needed B-smooth numbers so far
Found 700 of 4622 needed B-smooth numbers so far
Found 800 of 4622 needed B-smooth numbers so far
Found 900 of 4622 needed B-smooth numbers so far
Found 1000 of 4622 needed B-smooth numbers so far
Found 1100 of 4622 needed B-smooth numbers so far
Found 1200 of 4622 needed B-smooth numbers so far
Found 1300 of 4622 needed B-smooth numbers so far
Found 1400 of 4622 needed B-smooth numbers so far
Found 1500 of 4622 needed B-smooth numbers so far
Found 1600 of 4622 needed B-smooth numbers so far
Found 1700 of 4622 needed B-smooth numbers so far
Found 1800 of 4622 needed B-smooth numbers so far
Found 1900 of 4622 needed B-smooth numbers so far
Found 2000 of 4622 needed B-smooth numbers so far
Found 2100 of 4622 needed B-smooth numbers so far
Found 2200 of 4622 needed B-smooth numbers so far
Found 2300 of 4622 needed B-smooth numbers so far
Found 2400 of 4622 needed B-smooth numbers so far
Found 2500 of 4622 needed B-smooth numbers so far
Found 2600 of 4622 needed B-smooth numbers so far
Found 2700 of 4622 needed B-smooth numbers so far
Found 2800 of 4622 needed B-smooth numbers so far
Found 2900 of 4622 needed B-smooth numbers so far
Found 3000 of 4622 needed B-smooth numbers so far
Found 3100 of 4622 needed B-smooth numbers so far
Found 3200 of 4622 needed B-smooth numbers so far
Found 3300 of 4622 needed B-smooth numbers so far
Found 3400 of 4622 needed B-smooth numbers so far
Found 3500 of 4622 needed B-smooth numbers so far
Found 3600 of 4622 needed B-smooth numbers so far
Found 3700 of 4622 needed B-smooth numbers so far
Found 3800 of 4622 needed B-smooth numbers so far
Found 3900 of 4622 needed B-smooth numbers so far
Found 4000 of 4622 needed B-smooth numbers so far
Found 4100 of 4622 needed B-smooth numbers so far
Found 4200 of 4622 needed B-smooth numbers so far
Found 4300 of 4622 needed B-smooth numbers so far
Found 4400 of 4622 needed B-smooth numbers so far
Found 4500 of 4622 needed B-smooth numbers so far
Found 4600 of 4622 needed B-smooth numbers so far
=== Finding linear dependence===
=== Trying to factor n ===
=== Found factors ===
3744843080529615909019181510330554205500926021947 =
    1123456667890987666543211 * 3333322225555555777777777
Time elapsed:  491.74 seconds
```