Quadratic Sieve

Mark Liu, Shaw Phillips, Jean-Luc Rabideau

November 22, 2023

1 Introduction

The motivation behind the Quadratic Sieve came from the challenge of factoring composite numbers, which has been a long standing problem for mathematicians. Some early methods of prime factorization include trial division and later on, Fermat's difference of two squares factorization. Trial division, by far the simplest way of factoring composite numbers, involves sequentially testing the divisibility of n with increasing primes until the factor is found. While this is a viable method for small composite numbers, it becomes wildly inefficient when faced with larger numbers, becoming impossible to calculate for numbers with more than 30 digits [1]. Fermat's difference of squares method uses the relation $a^2 - b^2 = (a - b)(a + b)$, starting with the smallest square greater than n and checking if the difference is square. If it's a square, we can use the difference of squares to factor n. Fermat's method proves to be much quicker than trial division, but it runs into the same time complexity issue when faced with larger composite numbers.

While both these methods are viable for small numbers, when it comes to real-world applications of prime factorization, namely factoring an RSA modulus hundreds of digits long, these methods are far too inefficient to be usable. In order to tackle the issue of factoring increasingly large numbers, various methods were invented, the fastest of which being the Quadratic Sieve and Number Field Sieve. In this paper, we will describe the Quadratic Sieve Method and present our implementation of the algorithm.

2 The Quadratic Sieve

Invented by Carl Pomerance in 1981, the Quadratic Sieve was the fastest factoring algorithm until the discovery of the Number Field Sieve in 1993, though it remains faster for numbers less than 110 digits long. The basis of the Quadratic Sieve is to attempt to find two numbers x, y such that $x \not\equiv \pm y \pmod{n}$ and $x^2 \equiv y^2 \pmod{n}$. If these conditions are true, then we know that $gcd(x - y, n)$ gives a nontrivial factor of n. Hereafter we will detail the procedure of the Quadratic Sieve Method.

Let s be the smallest integer such that $s > \sqrt{n}$ and let

$$f(x) = (x + s)^2 - n$$

From here we will calculate $f(x_1), f(x_2), f(x_3), ..., f(x_k)$. Once we have our $f(x)$'s, we will choose a subset such that $f(x_{i_1})f(x_{i_2})...f(x_{i_j}) = y^2$ for some $0 \leq y < n$. We will detail how to choose such a subset of $f(x_i)$ in Section 2. From here we obtain the relationship

$$(x_{i_1}x_{i_2}...x_{i_j})^2 \equiv y^2 \pmod{n}.$$

Then, we can use the Euclidean Algorithm to calculate $gcd(x_{i_1}x_{i_2}...x_{i_j} - y,\ n)$ and check if it produces a non-trivial factor of n. If not, we continue to find more relations and try again.

2.1 Factor Base

First, given some bound B, we will define a number x to be B-smooth if we can write x as the product of primes less than B. The basic idea is that if we can find $f(x_i)$'s that are B-smooth, we will have access to their prime factorization. From there, we can multiply these B-smooth numbers together to get a square number (all powers are even). We will call the set of small primes less than B that we will use the **factor base**.

In order to create our factor base, we will generate all primes up to some chosen bound B. To do this, we can use the Sieve of Eratosthenes. The Sieve of Eratosthenes works as follows: Initialize an array representing the numbers 2 to n, with all numbers initially marked as prime. Now, initialize p as 2. Now, cross out all multiples of p, marking them as non-prime. Iterate until we find the next prime number, and cross out all of its multiples. We will repeat this until we reach $\sqrt{n}$. Finally, take all remaining numbers as prime.

Note that we can actually discard some of these primes, as shown from the following lemma.

Lemma 1. *Let p be a prime in B. If there exists an integer x with $f(x)$ divisible by p, then n is a square mod p.*

Proof. Suppose that there exists x where $p \mid f(x)$. Then, we must have

$$f(x) \equiv (x+s)^2 - n \equiv 0 \pmod{p} \implies n \equiv (x+s)^2 \pmod{p}.$$

Since $x + s \in \mathbb{Z}$, n must be a square mod p. □

Taking the contrapositive of Lemma 1, if n is not a square mod p for some prime p, we can discard p and remove it from our factor base. Next, we have the following lemma:

Lemma 2. *For any integer $a \not\equiv 0 \pmod{p}$ and odd prime p, $a^{\frac{p-1}{2}} \equiv 1 \pmod{p}$ if and only if a is a square mod p.*

Proof. Let $x \equiv a^{\frac{p-1}{2}}$.
$\boxed{\implies :}$ Suppose that $x \equiv 1$. First, let g be a primitive root mod p. Then, we must have $a \equiv g^j$ for some j. Then we must have

$$g^{j\frac{p-1}{2}} \equiv a^{\frac{p-1}{2}} \equiv x \equiv 1 \pmod{p}.$$

We then must have that $j\frac{p-1}{2} \equiv 0 \pmod{p-1}$, so j must be even. Thus, $j = 2k$ for some k and $a \equiv g^{2k} \pmod{p}$, so a is a square mod p.

$\boxed{\impliedby :}$ Suppose that a is a square mod p. Then, $a \equiv y^2$, so

$$a^{\frac{p-1}{2}} \equiv y^{p-1} \equiv 1 \pmod{p}$$

by Fermat's Little Theorem. □

Now, for every prime p generated by the Sieve of Erathosthenes, we can calculate $n^{\frac{p-1}{2}} \pmod{p}$. If it is not congruent to $1 \pmod{p}$, we can discard p.

```
def generate_primes(n):
    '''
    generates primes up to a bound n using the Sieve of Eratosthenes
    '''

    primes = []
    not_prime = [False] * (n + 1)

    for i in range(2, n + 1):
        if not_prime[i]:
            continue
        # if i is a prime, remove all factors of i starting at i^2
        for j in range(i * i, n + 1, i):
            not_prime[j] = True

        primes.append(i)

    return primes

def quadratic_res(a, p):
```

```
    '''
    calculates the Legendre symbol
    '''
    return pow(a, (p - 1) // 2, p)

def generate_factor_base(n, B):
    '''
    generates a factor base up to bound B
    '''

    factor_base = []
    primes = generate_primes(B)

    for p in primes:
        if quadratic_res(n, p) == 1:
            factor_base.append(p)

    return factor_base
```

2.2 Generating B-Smooth Numbers

After creating the factor base and choosing some sieving interval M, we can begin sieving for B-smooth numbers for x in $[-M, M]$. We will sieve in subintervals of maximum size 10000000 due to memory constraints. In each subinterval, we will first calculate all $f(x)$ for x in the subinterval and for each x_i, we will initialize a register $\log_2 f(x_i)$. Note that we use the number of bits in $f(x_i)$ to approximate the value since subtracting bits is considerably less expensive than dividing integers.

Next, for each prime p in our factor base, we calculate

$$(x + s)^2 \equiv n \pmod{p} \text{ for } x \in \mathbb{Z}_p$$

which we can solve using the Tonelli-Shanks Algorithm. This will give us two solutions, r_1 and $r_2 = p - r_1$. These are the two values of $x + s \pmod{p}$ such that p divides $f(x)$. Thus, for each root r_i, we will subtract the value of $\log_2 p$ from each register where $x \equiv r_i - s \pmod{p}$.

Finally, we will consider x_i as candidates, where after sieving, the register $f(x_i) < 20$. To make sure these candidates are B-smooth, we will then use trial division over our factor base. Note that the trial division here is considerably less expensive as we have reduced the number of candidates from the entire subinterval to only a small fraction through sieving.

The choice of 20 as a threshold was made after testing, where values higher than 20 resulted in a much larger runtime without many more B-smooth numbers being found, and values lower than 20 resulted in not enough B-smooth numbers being found.

Finally, in the program, we will alternate between searching in the positive x intervals and the negative until we have at least 100 more relations than the length of the factor base. This is to ensure we have at least 100 linear relations to attempt to factor n with.

```
def generate_pos_smooth_nums(n, factor_base, interval_start, interval_end):
    '''
    generates B-smooth numbers going forwards (positive x values)
    '''
    # sieve using f(x) = (x+s)^2 - n
    search_nums = []
    # contains the registers with log f(x)
```

```
    sieve_list = []

    # calculate f(x) values in the interval and populate sieve array
    s = math.ceil(math.sqrt(n))
    for x in range(interval_start + s, interval_end + s):
        num = (x * x) % n
        search_nums.append(num)
        sieve_list.append(round(math.log2(num)))


    # sieve with odd primes
    for p in factor_base[1:]:
        # find residues x1 and x2
        residues = tonelli_shanks(n, p)

        log_p = math.log2(p)
        # subtract from numbers with factor p
        for res in residues:
            idx = (res - s - interval_start) % p
            while idx < interval_end - interval_start:
                sieve_list[idx] = sieve_list[idx] - log_p
                idx += p

    threshold = 20

    # store candidates and
    bsmooth_candidates = []
    x_list = []


    # find registers that are less than the threshold
    for i, candidate in enumerate(sieve_list):
        if candidate < threshold:
            bsmooth_candidates.append(search_nums[i])
            x_list.append(i + interval_start + s)

    # verify that the candidates are B-smooth
    bsmooth_nums, x_pos = verify_candidates(factor_base, bsmooth_candidates, x_list)

    return bsmooth_nums, x_pos

def generate_neg_smooth_nums(n, factor_base, interval_start, interval_end):
    '''
    generates B-smooth numbers going backwards (negative x values)
    '''
    # sieve using f(x) = (x+s)^2 - n
    search_nums = []
    # contains the registers with log f(x)
    sieve_list = []

    # calculate f(x) values in the interval and populate sieve array
    s = math.ceil(math.sqrt(n))
    for x in range(interval_start + s, interval_end + s, -1):
        num = abs((x * x) - n)
        search_nums.append(num)
```

```
        sieve_list.append(round(math.log2(num)))


    # sieve with odd primes
    for p in factor_base[1:]:
        # find residues x1 and x2
        residues = tonelli_shanks(n, p)
        log_p = math.log2(p)
        # subtract from numbers with factor p

        for res in residues:
            idx = (res - s - interval_start) % p
            # account for negative x value
            if idx < interval_start - interval_end:
                sieve_list[idx] = sieve_list[idx] - log_p
                idx -= p
                idx = abs(idx)

                while idx < interval_start - interval_end:
                    sieve_list[idx] = sieve_list[idx] - log_p
                    idx += p

    threshold = 20
    bsmooth_candidates = []
    x_list = []

    # find registers that are less than the threshold
    for i, candidate in enumerate(sieve_list):
        if candidate < threshold:
            bsmooth_candidates.append(-1 * search_nums[i])
            x_list.append(interval_start + s - i)

    # verify that the candidates are B-smooth
    bsmooth_nums, x_pos = verify_candidates(factor_base, bsmooth_candidates, x_list)

    return bsmooth_nums, x_pos
```

2.3 Gaussian Elimination

Once B-smooth numbers have been found, the exponents of their factors are stored modulo 2 as rows into a matrix with entries 0 or 1. Then, Gaussian Elimination is performed modulo 2 to find linear dependencies among factors of the B-smooth numbers found. These correspond to products of B-smooth numbers that are themselves squares. That is, $x^2 \equiv y^2$ (mod n). It still remains to be checked that $x \not\equiv \pm y$ (mod n) to get a nontrivial factor $gcd(x-y, n)$. The standard Gaussian Elimination algorithm is used, with one of the only primary optimizations saving memory by storing each row as a integer whose bits correspond to the factors modulo 2.

```
def build_matrix(nums, base):
    matrix = [0]*len(base)
    for j in range(len(nums)):
        num_pos = 1 << j
        div = nums[j]
        for i in range(len(base)):
            p = base[i]
            while(div % p == 0):
```

```
                div = div // p
                matrix[i] ^= num_pos
    return matrix

def gauss_elim(m,size):
    # m is a list, each entry is an integer whose bits correspond to factor base.
    # size is the number of primes in the factor base i.e. len(factor_base)
    # 1 in least significant bit means smallest prime divides the num

    # returns the rref matrix in the same format, and a list of pivot columns

    num_pivots = 0
    pivots = []
    free = []
    for col in range(size):
        #print("looking at column", col)
        col_pos = 1 << col
        looking = True
        for i in range(num_pivots, len(m)):  # look at each row

            row = m[i]
            if row & col_pos:  # if row i contains a 1 in this column
                if looking:
                    looking = False  # we have found a pivot in this row
                    # swap so this is the next pivot row if not already in place
                    if i != num_pivots:
                        m[i] = m[num_pivots]
                        m[num_pivots] = row
                else:
                    # add the new pivot row to every other row
                    m[i] ^= m[num_pivots]


        if not looking:
            num_pivots += 1
            pivots.append(col)
        else:
            free.append(col)

    #the matrix is now in echelon form
    #the following makes pivot columns have zeros above pivots (rref)

    for j in range(num_pivots):  # go through each pivot column
        col = pivots[j]
        col_pos = 1 << col
        for i in range(0, j): # check each row above the row with the pivot
            #print(i,col)
            if m[i] & col_pos:
                # add row j to row i to get rid of a one above a pivot
                m[i] ^= m[j]

    #rref is now computed

    #now, compute nullspace
    null = [0]*len(free)
```

```
    for i in range(len(free)):

        free_var = free[i]
        free_var_pos = 1 << free_var
        for j in range(min(free_var, len(m))):
            row = m[j]

            if row & free_var_pos:

                null[i] ^= (1 <<pivots[j])

        null[i] ^= free_var_pos
    return m, null
```

2.4 Greatest Common Divisor

After numbers x, y are found such that $x^2 \equiv y^2$ (mod n) but $x \not\equiv \pm y$ (mod n), the greatest common divisor $gcd(x - y, n)$ is a nontrivial factor of n. This is computed using the Euclidean Algorithm.

```
def gcd(a, b):
  a = abs(a)
  b = abs(b)
  if a<b:
    #swap a and b so that a is larger
    temp = a
    a = b
    b = temp

  if b == 0:
    return a

  remainder = a % b
  while remainder:
    #loop until remainder is 0

    a = b
    b = remainder
    remainder = a % b

  return b
```

2.5 Bound B

One of the difficulties of the Quadratic Sieve is choosing an optimal bound B and optimal sieving interval M. If we choose B to be very small, then our factor base will be smaller. However, since B-smooth numbers are sparsely distributed for smaller values of B it will take longer to find B-smooth numbers. On the other hand, if we choose B to be large, B-smooth numbers become more common, but creating the factor base, factoring potential B-smooth numbers, and Gaussian elimination on the final matrix will all take longer.

We chose to use the B value of

$$B = (e^{\sqrt{ln(n)ln(ln(n))}})^{\frac{\sqrt{2}}{4}}$$

based on the calculations of both Pomerance and Landquist [2]. In our implementation, we reduced the calculated value provided by Landquist to the form above in order to decrease the number of B-smooth numbers we would need, thereby reducing the matrix size and optimize the processing time.

```
def calc_bound(n):
    L = pow(math.e, 0.5 * math.sqrt(math.log(n) * math.log(math.log(n))))
    return int(L)
```

2.6 Sieving Interval

In order to keep $f(x)$ small and hence have a larger chance of $f(x)$ being made of small primes, we want to consider values of x close to 0, so we calculate a sieving interval M. This allows us to only consider values of x over the interval $[-M, M]$. The optimal value for M we chose was

$$M = (e^{\sqrt{log(n)log(log(n))}})^{\frac{3\sqrt{2}}{4}}$$

based on the calculations of both Pomerance and Landquist [2]. In our implementation, we found that M was too small for smaller n, so we take the max of the calculated interval and 500000, to ensure a large enough sieving interval.

```
def calc_interval(n):
    L = pow(math.e, math.sqrt(math.log(n) * math.log(math.log(n))))
    return math.ceil(pow(L, 3 * math.sqrt(2) / 4))

sieve_interval = max(calc_interval(n), 500000)
```

3 Empirical Results

For test numbers:

1. 16921456439215439701 (20 digits)
2. 468395662999369192342467268 09 (29 digits)
3. 617283580864197520363830491969135846 9663 (40 digits)
4. 374484308052961590901918151033055420550092602 1947 (49 digits)

we have the following results:

3.1 Time

1. 2.1456 s
2. 3.2675 s
3. 772.7679 s
4. 11021.8873 s

3.2 Factors

1. $5915587277 \cdot 2860486313$
2. $100000000105583 \cdot 468395662504823$
3. $555555222777777773333 \cdot 11111111111111111011$
4. $1123456667890987666543211 \cdot 3333322225555555777777777$

References

[1] Pomerance, Carl. (2008). Smooth numbers and the quadratic sieve.

[2] Landquist, Eric. (2001). The Quadratic Sieve Factoring Algorithm.