FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

Assignment of master's thesis

Title:	Quadratic sieve factorization algorithm
Student:	Bc. Ondřej Vladyka
Supervisor:	doc. Ing. Ivan Šimeček, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Security
Department:	Department of Information Security
Validity:	until the end of summer semester 2021/2022

Instructions

1) Study the Dixon method for integer factorization [1]. Study its modified version, the quadratic sieve method (QS), and its variations, the multiple polynomial quadratic sieve method (MPQS) and the self-initializing quadratic sieve method (SIQS) (see [2], [3] and [4]).
2) Research some existing open-source implementations of these algorithms.
3) Implement these algorithms using C++ language and the GMP library. Discuss and implement their parallelization.
4) Measure the performance of these methods on faculty server STAR. Compare the measured results of all implemented methods.

_

[1] J. Dixon, "Asymptotically fast factorization of integers", Mathematics of Computation, vol. 36, no. 153, pp. 255-260, 1981.
[2] R. Crandall and C. Pomerance, "Prime numbers: A Computational Perspective". New York: Springer, 2010, pp. 261-278.
[3] V. Shoup., "A Computational Introduction to Number Theory and Algebra", 2nd ed. Cambridge University Press, 2008.
[4] P. Kovac, "Optimalizovaná faktorizace velkých čísel pomocí knihovny GMP", Master's thesis, Czech Technical University, 2009.

Electronically approved by prof. Ing. Róbert Lórencz, CSc. on 12 June 2020 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Quadratic sieve factorization algorithm

Bc. Ondřej Vladyka

Department of Information Security
Supervisor: doc. Ing. Ivan Šimeček, Ph.D.

May 6, 2021

Acknowledgements

I would like to thank my friends and family for their support through thick and thin. I would also like to thank my supervisor, doc. Ing. Ivan Šimeček, Ph.D., for his invaluable consultations and guidance that allowed me to finish this thesis.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 6, 2021 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Vladyka, Ondřej. *Quadratic sieve factorization algorithm.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

Diplomová práce se zabývá faktorizací čísel pomocí algoritmu kvadratického síta a některých jeho modifikacích. V teoretické části práce je detailní popis těchto algoritmů. V praktické části práce je popsaná jejich implementace v jazyce C++ a paralelizace pomocí openMP a MPI. Implementované metody jsou otestovány na fakultním serveru STAR a výsledky navzájem porovnány.

Klíčová slova Faktorizace, faktorizace čísel, kvadratické síto, QS, více polynomiální kvadratické síto, Dixonova metoda, Fermatova faktorizace, MPQS

Abstract

This thesis is focused on the quadratic sieve factorization algorithm and some of its modifications. The theoretical part of this thesis serves as a detailed description of these algorithms. The practical part of the thesis describes their implementation in C++ programming language and parallelization using openMP and MPI libraries. The implemented methods are tested on the faculty server STAR and results are compared together.

Keywords Factorization, integer factorization, quadratic sieve, QS, multiple polynomial quadratic sieve, Dixon's method, Fermat's factorization, MPQS

Contents

List of Figures

List of Tables

Introduction

Number theory is a branch of mathematics dealing with the analysis of integers and prime numbers, among other things. Mathematical problems in number theory are quite special, typical for their rather simple description, but often very complex solution. A perfect example of one of these problems is the integer factorization problem.

The description of the integer factorization problem is very simple indeed, it is also probably as old as the human civilization itself. It goes as follows: "given any non-prime number, figure out all of its non-trivial divisors." Even an ordinary layman could surely understand that, but solving this very problem can get extremely difficult. So difficult in fact, that modern day computer security is partially based on this seeming unsolvability.

The concept of integer factorization and its related prime numbers has fascinated mathematicians since the times of the ancient Greeks, but most of the big breakthroughs happened surprisingly recently – in the 20th century. This is probably thanks to the rapid development in the computer science and cryptography in that era, but most notably thanks to the the invention of the asymmetric public-key cryptosystem RSA in 1977.

The basic idea of the security behind RSA is that multiplying two numbers together is fairly easy and quick to accomplish no matter their size, but its reverse operation – finding two non-trivial divisors of a given number – is unachievable for large enough number ("unachievable" meaning there is no publicly known algorithm that could find the divisors in a polynomial time based on the bit-length of the given number[1]).

For a long time there have been many different ways and approaches to solve the integer factorization problem. The easiest approach to find factors of a composite number N is to just use a "brute-force" method and try divide N by all the numbers starting from 2 up to $\sqrt{N}$ (this method is also known

[1]The existence of Shor's algorithm with quantum computers slightly refutes this claim, but the state of quantum computing is unfortunately/thankfully not developed enough to be usable in any practical use-case yet.

as *trial division*). This algorithm could be theoretically used to factorize any number, but using it to factorize larger and larger integers (that are usually used in real-life computer security scenarios) quickly becomes unfeasible. It is no wonder that the wide-spread usage of RSA during the last century has motivated many mathematicians to figure out new ways to factorize integers in much more efficient and less time consuming manner.

In this master's thesis, I will be analyzing two such algorithms. The first one is called the *Dixon's factorization method* published by John D. Dixon in 1981. The second one is called the *quadratic sieve factorization method*, an improvement on the Dixon's method published in 1982 by Carl Pomerance. I will implement both methods in the C++ programming language using the *GNU Multiple Precision Arithmetic Library* to help with handling arbitrary large integers. I will explore some basic optimization techniques known as *multiple polynomial quadratic sieve* and *large prime optimization*. I will discuss and implement parallelization using the openMP and MPI libraries. Lastly, I will test my implementations on the faculty server STAR and compare the results.

Chapter 1

Basic terms and definitions

Before the analysis of more advanced factorization algorithms can begin, I need to establish some basic (but important) terms and definitions that I will be using throughout the rest of this thesis. Even just to be able to properly define the integer factorization problem, I will need to refresh on some parts of the discrete mathematics and number theory. In most cases I will be working with the set of natural numbers $\mathbb{N} = \{1, 2, 3, \ldots\}$ or integers $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ and with two binary operations on those sets, $+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, $\cdot : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, $+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ and $\cdot : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, representing classical addition and multiplication of natural numbers or integers respectively. All terms and definitions used in this chapter are taken from [1] and [2].

Definition 1.1. (Integer divisibility)
For any two integers $a, b \in \mathbb{Z}$ we say that **a divides b** if there exists some $z \in \mathbb{Z}$ such that $a \cdot z = b$. If a divides b, we can write that as $a \mid b$. This is the equivalent of saying that a is a **divisor** of b, b is a **multiple** of a, or that b is **divisible** by a. If b is not divisible by a, we write that as $a \nmid b$.

With the new-found knowledge of integer divisibility, we can define the major protagonists of integer factorization: primes and composite numbers.

Definition 1.2. (Prime number)
Prime number is a number $p \in \mathbb{N} \setminus \{1\}$ that is divisible only by itself and the number 1. Therefore, we can say that p has precisely two divisors.

Definition 1.3. (Composite number)
Composite number is a number $c \in \mathbb{N} \setminus \{1\}$ that is not a prime number. Composite number always has at least three divisors. It is trivial to see that c is a composite if and only if there are some numbers $a, b \in \mathbb{N}, 1 < a < c, 1 < b < c$ such that $c = a \cdot b$. The notion of composite numbers can be extended to the set of integers $\mathbb{Z}$ as well, but it is unnecessary for the purpose of this thesis, since the factorization problem is usually focused on factorizing mainly large natural numbers.

Definition 1.4. (Probabilistic primality test)
Probabilistic primality test are family of algorithms that examine some odd integer $n > 2$. If this integer is prime, they always return true. If integer n is a composite, they might return true with certain probability, otherwise they return false. The test parameters are usually chosen such that the probability of returning true for a composite number is so extremely small it can be dismissed for real-life applications.

Definition 1.5. (Probable prime number)
A number $p \in \mathbb{N} \setminus \{1\}$ is a **probable prime number**, if p satisfies some specific condition that holds true for all prime numbers, but holds false for most composite numbers. Probable primes have one big advantage over prime numbers: they can be produced very quickly using probabilistic primality tests in polynomial time. In some rare cases, probable prime number can be a composite (in which case it is also called a **pseudoprime**).

In the old days of mathematics, some mathematicians considered the number 1 to be prime as well. Alas, this would slightly complicate the formulations of other theorems and algorithms used later on in this thesis. Therefore, number 1 is a special number in the set of $\mathbb{N}$ in that it is neither prime nor composite.

With the above-mentioned information, we can finally define the *fundamental theorem of arithmetic* and the primal focus of this thesis, the *factorization problem* itself.

Theorem 1.6. (Fundamental theorem of arithmetic)
Every natural number $n \in \mathbb{N}$ can be represented in exactly one unique way as a product of one or more primes:

$$n = p_1^{e_1} \cdot p_2^{e_2} \cdot p_3^{e_3} \cdot \ldots \cdot p_k^{e_k} = \prod_{i=1}^{k} p_i^{e_i}$$

where $p_1 < p_2 < \ldots < p_k \in \mathbb{N}$ are distinct primes and $e_1, \ldots, e_k \in \mathbb{N}$ are their exponents. This product is also called the **prime number factorization** of n. The exponents $e_1, \ldots, e_k$ can be called **prime exponents**.

The theorem 1.6 seemingly does not apply for $n = 1$. Fortunately, this case is not very interesting (it is the cases where n is very big that are interesting) and can be solved by letting $k = 0$ and interpreting 1 as a product of zero terms.

Just like in the definition 1.3, the fundamental theorem of arithmetic can be easily extended to all non-zero integers. If we want to find the prime number factorization of a number $n \in \mathbb{Z} \setminus \{0\}$, we can simply find the prime number factorization of $|n|$ and then multiply one of the primes (usually p_1) by -1 if $n < 0$.

Definition 1.7. (The integer factorization problem)
The integer factorization problem is a mathematical problem defined as follows: given some number $n \in \mathbb{Z}$, find all the prime numbers $p_1, p_2, \ldots, p_k \in \mathbb{N}$ and their exponents $e_1, e_2, \ldots, e_k \in \mathbb{N}$ from the prime number factorization of n.

Although the definition 1.7 is asking for all prime numbers and their exponents while factoring some n, I will consider it a correct solution to find some two integers $a, b \in \mathbb{Z} \setminus \{1, n\}$ such that $n = a \cdot b$, even if a or b are composites. The reason is that if some algorithm can find a and b in a reasonable time, it can be used recursively on a and/or b (both of which are smaller than n) to find a proper solution to the factorization problem. Also, when dealing with real-life applications and challenges (like the RSA cryptosystem), n is usually a composite with only two prime factors.

Definition 1.8. (Greatest common divisor)
Given some numbers $a, b \in \mathbb{Z}$, we call $d \in \mathbb{Z}$ a **common divisor** of a and b, if and only if $d \mid a$ and $d \mid b$. If such d is a positive integer and all other common divisors of a and b also divide d, d is called the **greatest common divisor** of a and b. Usually, the extended Euclidean algorithm is used to find the greatest common divisor of n in polynomial time with respect to bit-size of n. We write the greatest common divisor of a and b as

$$\gcd(a, b)$$

Definition 1.9. (Congruence modulo n)
Given some numbers $n \in \mathbb{N}$ and $a, b \in \mathbb{Z}$, we say that a **is congruent to** b **moludo** n if $n \mid (a - b)$. This is the same as saying that there exists some $k \in \mathbb{Z}$, such that

$$a = b + k * n$$

We write the congruence modulo n as

$$a \equiv b \pmod{n}.$$

Definition 1.10. (B-smooth number)
A number n is called **B-smooth number**, if its prime number factorization

$$n = \prod_{i=1}^{k} p_i^{e_i}$$

satisfies $p_i \leq B$ for all $i = 1, 2, \ldots, k$.

Definition 1.11. (B-smooth relation)
Congruence of type

$$x^2 \equiv \prod_{i=1}^{k} p_i^{e_i} \pmod{n},$$

where all $p_i \leq B$, is called **B-smooth relation** or smooth relation with respect to smoothness bound B.

Definition 1.12. (Partial B-smooth relation)
Congruence of type

$$x^2 \equiv \prod_{i=1}^{k} p_i^{e_i} \cdot P \pmod{n},$$

where all $p_i \leq B$ and P is some prime number and $P > B$ is called **partial B-smooth relation** or 1-partial B-smooth relation. It is called 1-partial relation, because it is only one prime over the smoothness bound B. A 2-partial or 3-partial B-smooth relations are very similar, only they are two or three primes over the smoothness bound B.

Definition 1.13. (Quadratic residue)
If an integer n is congruent to a square modulo p, meaning

$$x^2 = n \pmod{p}$$

for some $x \in \mathbb{Z}_p$, this integer n is called **quadratic residue** modulo p. If this congruence does not have a solution (such x does not exist), n is said to be **quadratic nonresidue** modulo p.

Definition 1.14. (Legendre symbol)
Given an odd prime p and integer n, the **Legendre symbol** is a function of n and p and is defined as

$$\left(\frac{n}{p}\right) = \begin{cases} 0 & \text{if } p \mid n \\ 1 & \text{if } n \text{ is quadratic residue modulo } p \\ -1 & \text{if } n \text{ is quadratic nonresidue modulo } p \end{cases}$$

Definition 1.15. (Euler's criterion)
Euler's criterion is a formula used to calculate Legendre symbol. Let p be an odd prime and n a positive integer not divisible by p, then

$$\left(\frac{n}{p}\right) \equiv n^{\frac{p-1}{2}} \pmod{p}$$

1.1 Basic algorithms

1.1.1 Sieve of Eratosthenes

According to [3], one of the things that heavily inspired the creation of the quadratic sieve method was an algorithm already known by ancient Greeks, the sieve of Eratosthenes, used to find all primes up to some parameter n. It is also fairly important because it is an extremely fast way to generate all primes below n for relatively small n, which is going to be important for factor base generation in quadratic sieve.

Algorithm 1: Sieve of Eratosthenes

Input: positive integer n
Output: all primes up to n
Primes $\leftarrow \emptyset$
`/* initialization phase */`
for $i = 2, 3, \ldots, n$ **do**
    Primes$[i] \leftarrow$ *true*
`/* sieving phase */`
for $p = 2, 3, \ldots, \lfloor\sqrt{n}\rfloor$ **do**
    if prime$[p]$ = *true* **then**
        $c \leftarrow p^2$
        while $c \leq n$ **do**
            Primes$[c] \leftarrow$ *false*
            $c \leftarrow c + p$
return all integers $i = 2, 3, \ldots, n$ for which Primes$[i]$ is *True*

1.1.2 Trial division

Trial division is using the "brute force" approach to factorize some composite n and is therefore by far the simplest method of them all. Still, it is widely used in general factorization programs with some given cut-off parameter to factorize out all of the small prime factors of n up to the cut-off. Below is a slightly improved version, firstly removing all factors of 2 and then only checking for divisibility with odd integers.

Algorithm 2: Trial division

Input: positive integer n
Output: all prime factors of n
Factors $\leftarrow \emptyset$
`/* remove factors of 2 */`
while $2 \mid n$ **do**
    add 2 to Factors
    $n \leftarrow n/2$
`/* start at 3, only checking odd integers */`
$i \leftarrow 3$
while $i \leq \lfloor\sqrt{n}\rfloor$ **do**
    if $i \mid n$ **then**
        add i to Factors
        $n \leftarrow n/i$
    else
        $i \leftarrow i + 2$
return Factors

Chapter 2

Quadratic sieve factorization method

Instead of diving straight into the theory behind the quadratic sieve method, first I should go over the two of its direct predecessors: the Fermat's integer factorization and the Dixon's factorization method.

2.1 Fermat's integer factorization

There are many different approaches to try and factorize integers, one of them is called the **Fermat's integer factorization**. In fact, all of the factorization methods in this chapter are ultimately based on this method. It is named after its inventor, the famous french mathematician Pierre de Fermat. The basic idea is as follows: suppose some number n, a product of two primes p and q, is to be factorized. If one can find a way to write down n as difference of two squares, meaning

$$n = x^2 - y^2 \tag{2.1}$$

for some integers $x, y \in \mathbb{Z}$, it is then trivial to see that

$$n = (x - y) \cdot (x + y).$$

Furthermore, assuming n is an odd integer (and if not, just find and remove all powers of 2 by simple trial division), then n can always be written as a difference of two squares

$$n = p \cdot q$$
$$n = \left(\frac{p+q}{2}\right)^2 - \left(\frac{p-q}{2}\right)^2.$$

Because n is odd, p and q have to be odd as well. That means $x = \frac{p+q}{2}$ and $y = \frac{p-q}{2}$ are valid integers from $\mathbb{Z}$.

If the factors are $(x-y)=1$ and $(x+y)=n$, then we have found so-called **trivial factors** of n, meaning no progression in finding the actual p and q prime factors. If that happens, we have to continue searching for different values of x and y, hoping they are going to be **non-trivial**.

For the sake of convenience, throughout this section I will be referring to x^2 as X and to y^2 as Y.

2.1.1 Algorithm

The real question here is how to generate the x and y values. There are probably many different variations, but the frequently used one is the one originally devised by Fermat. Start by letting $x = \left\lceil \sqrt{n} \right\rceil$, then repeatedly calculate $Y = x^2 - n$ and check if $\sqrt{Y} \in \mathbb{Z}$ while incrementing x by one each time.

Algorithm 3: Fermat's integer factorization

Input: odd composite integer n
Output: two factors of n
for $i = 0, 1, 2, \ldots$ **do**
  $x \leftarrow \left\lceil \sqrt{n} \right\rceil + i$
  $Y \leftarrow x^2 - n$
  $y \leftarrow \sqrt{Y}$
  if y *is an integer from* $\mathbb{Z}$ $\wedge\, (x-y) \neq 1$ **then**
    return $(x-y)$, $(x+y)$

As per [1, p. 226], this always terminates with the non-trivial factors of n before x reaches $\lfloor (n+9)/6 \rfloor + 1$. This is because the search for x and y starts around $\sqrt{n}$ and slowly stretches out. Therefore, the worst case scenario is when the two factors of n are the furthest apart, which is the case for $n = 3 * p$ where p is some prime and in this case x will reach exactly $(n+9)/6$ before finding non-trivial factors of n.

It is apparent that the time complexity of this algorithm is not only dependent on n, but it is also heavily dependent on its factors p and q. If these factors are relatively close to each other (therefore close to $\sqrt{n}$), Fermat's factorization will find them very quickly. The further are p and q apart, the longer it will take. In the most extreme cases, it can take even longer than with trial division. It is interesting to note that the worst case scenario for Fermat's method is the best case scenario for trial division method (where we start from 2 and end at $\sqrt{n}$) and vice versa.

2.1.2 Example

Given the number $n = p \cdot q = 8509$, find the factors p and q using the Fermat's integer factorization. Firstly, calculate $\left\lceil \sqrt{n} \right\rceil = 93$. Using the sequence $93, 94, 95, 96, 97, \ldots$ as values for x, calculate $Y = x^2 - n$. The resulting sequence for Y would be $140, 327, 516, 707, 900, \ldots$. When the $Y = 900$ is hit, the algorithm can stop because $900 = 30^2$ and so

$$97^2 - 8509 = 900 = 30^2$$

$$8509 = 97^2 - 30^2 = (97 - 30) \cdot (97 + 30)$$

$$8509 = 67 \cdot 127$$

$$p = 67,\ q = 127$$

2.2 Dixon's factorization method

As stated above, if n gets large enough while using Fermat's method, it quickly becomes infeasible to individually check all possibilities for y and x, especially when the worst case scenario scales exponentially with the bit-size of n.

According to [4], Maurice Kraitchik, Belgian mathematician specializing in number theory, suggested that equation 2.1 from Fermat's method can be loosened up a little bit, replacing it with congruence

$$x^2 \equiv y^2 \pmod{n} \tag{2.2}$$

with the condition that

$$x \not\equiv \pm y \pmod{n}.$$

If one can find values x and y that satisfy this congruence, then the result will be

$$n \mid (x^2 - y^2)$$

$$n \mid (x - y) \cdot (x + y).$$

By using the extended Euclidean algorithm to find the $\gcd((x-y), n)$ there is a chance we find one of the non-trivial factors of n. According to [3], if n is odd integer with at least two different prime factors, then the probability of this happening is at least 50 %.

But there is still one big problem to solve: How to find the values x and y from the congruence 2.2? Just picking some pseudorandom numbers and seeing if they fit in the equation is not very effective nor elegant solution. In 1981, John D. Dixon published a method in [5], where he described a much better way of finding them (in fact, he was not the first person to discover this method, but he was the first one to publish it with a rigorous proof of its time complexity). Instead of searching straight for the answers for x and y

that satisfy congruence 2.2, it is better to create them from some other values of x_j and y_j that "almost" satisfy the congruence 2.2.

Similarly to the section 2.1, for convenience sake I will be referring to x^2 as X (this time meaning "the left side of the congruence 2.2") and to y^2 as Y (meaning "the right side of the congruence 2.2").

2.2.1 The main idea

First and foremost, some parameter B representing the smoothness bound must be chosen. Then a set called the **factor base** (or just FB for short) is created. It is just a set of k primes $\text{FB} = \{p_1, p_2, \ldots, p_k\}$ such that $p_i \leq B$ for all $i = 1, 2, \ldots, k$. Next, we search for some amount of B-smooth relations by calculating x_j^2 for some values x_j and checking if that square is B-smooth using the primes prepared in FB. When we get enough different relations, we can combine a specific subset of them together to create x and y satisfying the congruence 2.2.

There are three important questions that need answers here however. How do we generate values x_j, how many relations do we need and how do we combine them together? The answer to the first question is to let

$$x_j = \left\lceil \sqrt{n} \right\rceil + j \text{ for } j = 1, 2, \ldots$$

and then calculate

$$Y_j = |x_j^2|_n.$$

To answer the second question is more tricky. The usual approach is to get about as many relations as there are factors in the factor base (in fact, it should be at least one more than the size of factor base). The exact number and the reasoning behind it will be discussed in the section 3.4.3 dealing with the implementation details. For now just suppose that some m relations is enough. If Y_j is B-smooth, there exist the exponents $e_{j,i}$ from prime factorization of Y_j. If we can find m such relations, we get

$$\begin{aligned} x_1^2 &\equiv \prod_{i=1}^{k} p_i^{e_{1,i}} \pmod{n} \\ x_2^2 &\equiv \prod_{i=1}^{k} p_i^{e_{2,i}} \pmod{n} \\ &\vdots \\ x_m^2 &\equiv \prod_{i=1}^{m} p_i^{e_{m,i}} \pmod{n} \end{aligned}$$

Now to answer the third question. After we multiply the chosen relations together, we need to have all the exponents for all the primes on the right-hand side even. Without loss of generality, suppose we multiply together the

relations for x_1, x_2 and x_3 making exponents for all primes in factor base $e_{1,i} + e_{2,i} + e_{3,i}$ even for $i = 1, 2, \ldots, k$.

$$x_1^2 \cdot x_2^2 \cdot x_3^2 \equiv \prod_{i=1}^{k} p_i^{e_{1,i}+e_{2,i}+e_{3,i}} \pmod{n}$$

$$(x_1 \cdot x_2 \cdot x_3)^2 \equiv \left(\prod_{i=1}^{k} p_i^{\frac{e_{1,i}+e_{2,i}+e_{3,i}}{2}} \right)^2 \pmod{n},$$

which gives us the values satisfying congruence 2.2. It is apparent that this principle generalizes to any finite set of relations, but only if the sum of exponents is even for all of their primes. Therefore, in order to use this method, there needs to be a way to find such subset from all of the m B-smooth relations that are collected. This can be achieved using linear algebra, discussed in more detail in the section 3.4.3.

2.2.2 Example

Given the number $n = p \cdot q = 44377$ and the smoothness bound $B = 7$, find the factors p and q using the Dixon's integer factorization. Firstly, we generate the factor base $FB = \{2, 3, 5, 7\}$ and calculate $\left\lceil \sqrt{44377} \right\rceil = 211$. Then, we follow by generating numbers with

$$x_j = 211 + j \text{ for } j = 1, 2, \ldots$$

$$Y_j = |x_j^2|_{44377}$$

and checking if Y_j is B-smooth number. If it is, we collect the smooth relation $x_j^2 \equiv Y_j \pmod{44377}$. After collecting enough relations and using linear algebra, we find that we can multiply relations

$$x_8^2 = (219)^2 \equiv 3584 = 2^9 \cdot 7 \pmod{44377}$$

$$x_{18}^2 = (229)^2 \equiv 8064 = 2^7 \cdot 3^2 \cdot 7 \pmod{44377}$$

to get modular equivalence of two squares as follows:

$$219^2 \cdot 229^2 \equiv 2^{16} \cdot 3^2 \cdot 7^2 \pmod{44377}$$

$$(219 \cdot 229)^2 \equiv (2^8 \cdot 3 \cdot 7)^2 \pmod{44377}$$

$$(5774)^2 \equiv (5376)^2 \pmod{44377},$$

getting the congruence of type 2.2. Now with some algorithm for finding the greatest common divisor of two numbers, we can see that

$$p = \gcd(5774 - 5376, 44377) = 199$$

$$q = 44377/199 = 223.$$

2.2.3 Algorithm

The more concrete algorithm would get fairly long and complicated, so only very general pseudo-code follows.

Algorithm 4: Dixon's integer factorization

Input: odd composite integer n, positive integer B
Output: two factors of n
FB $\leftarrow$ create factor base for primes $\leq B$
Relations $\leftarrow \emptyset$
Exponents $\leftarrow \emptyset$
$j \leftarrow 1$
while not enough B-smooth relations in Relations **do**
  $x_j \leftarrow \left\lceil \sqrt{n} \right\rceil + j$
  $Y_j \leftarrow |x_j^2|_n$
  `/* B-smoothness is tested using primes stored in FB */`
  if Y_j is B-smooth **then**
    Add $x_j^2 \equiv Y_j \pmod{n}$ to Relations
    Add prime exponents from prime number factorization of Y_j to Exponents
  $j \leftarrow j + 1$
Solutions $\leftarrow$ Use GEM on Exponents to get lists of relations with their linear combination giving even sums of exponents for all primes
foreach *solution in Solutions* **do**
  $x \leftarrow$ multiply all x_i from list of relations in solution
  $Y \leftarrow$ multiply all Y_i from list of relations in solution
  $y \leftarrow \sqrt{Y}$
  $p \leftarrow \gcd(x - y, n)$
  if $p \neq 1 \wedge p \neq n$ **then**
    return p, n/p
return Unable to find non-trivial factors.

The time complexity of most methods based on Fermat's factorization is very problematic to prove. The difficulty lies in the uncertainty whether Y_j is going to be B-smooth or not. Dixon's method is one of the very few with a rigorous mathematical proof, published by Dixon in [5] and stating that the time complexity of factorizing integer n with this method is

$$\mathcal{O}(\exp(3(2 \ln n \ln \ln n)^{1/2}))$$

or in the L-notation

$$L_n \left[\frac{1}{2}, 3\sqrt{2} \right],$$

making Dixon's method a sub-exponential factorization algorithm.

2.3 Quadratic sieve factorization method

When it comes to the basic Dixon's method described above, there are a few problems that make it very slow, so much so that it becomes unusable in any sort of real world scenario. However, it is a great starting point for other factorization methods and the quadratic sieve is next in line in this evolutionary branch.

In this method, the 2.2 congruence is still ultimately used to find factors of n, but what has changed is the way B-smooth relations are generated. In order to generate potential B-smooth relations, a polynomial is now used instead. There are a few different approaches using slightly different polynomials, but the basic one is defined as

$$Q(x) = \left(\left\lceil \sqrt{n} \right\rceil + x\right)^2 - n.$$

The sequence of $Q(1), Q(2), Q(3), \ldots$ is then used to generate possible B-smooth relations. If the value of $Q(x_j)$ for some x_j is B-smooth, then we get the B-smooth relation

$$Q(x_j) = \prod_{i=1}^{k} p_i^{e_{j,i}} \equiv \left(\left\lceil \sqrt{n} \right\rceil + x_j\right)^2 \pmod{n}$$

and, analogically to the Dixon's method, if we get big enough set of such B-smooth relations, we can find (using the same linear algebra tools used in Dixon's method) some subset J for which the sum of exponents $e_{j,i}$ of all $Q(x_j)$ in the subset is an even number for all the primes p_i in factor base and we get

$$\prod_{x_j \in J} Q(x_j) = \prod_{x_j \in J} \prod_{i=1}^{k} p_i^{e_{j,i}} = \prod_{i=1}^{k} p_i^{\sum_{x_j \in J} e_{j,i}} = \left(\prod_{i=1}^{k} p_i^{\frac{\sum_{x_j \in J} e_{j,i}}{2}}\right)^2.$$

Therefore, by multiplying all B-smooth relations in the subset J, we get

$$\prod_{x_j \in J} Q(x_j) = \left(\prod_{i=1}^{k} p_i^{\frac{\sum_{x_j \in J} e_{j,i}}{2}}\right)^2 \equiv \prod_{x_j \in J} \left(\left\lceil \sqrt{n} \right\rceil + x_j\right)^2 \pmod{n}$$

$$\left(\prod_{i=1}^{k} p_i^{\frac{\sum_{x_j \in J} e_{j,i}}{2}}\right)^2 \equiv \left(\prod_{x_j \in J} \left(\left\lceil \sqrt{n} \right\rceil + x_j\right)\right)^2 \pmod{n},$$

which is an instance of the congruence 2.2. The next three sections look at the three main improvements quadratic sieve offers over the Dixon's method: better factor base generation, addition of the sieving process and the logarithmic approximation.

2.3.1 Improving the factor base

In Dixon's method described in section 2.2, the process of generating the factor base was very simple. Based on the smoothness bound B, simply add all primes that are less than or equal to B to the FB. This leads to a big problem: the bigger B is, the larger the size of FB is going to be and the more B-smooth relations we need to find. This problem is unfortunately inevitable, but it can be mitigated a little bit with smarter strategy of FB generation.

As a reminder, the only point of FB is to hold the primes that are used to test if $Q(x)$ for some x is a B-smooth number. In other words, all primes in FB that can divide $Q(x)$ are useful to have in FB, other primes that can not divide $Q(x)$ are never used (their prime exponent is always going to be 0 in all relations) and so they are not useful and only make the size of FB bigger, resulting in needless search for more B-smooth relations. Luckily, with the way the polynomial $Q(x)$ is defined, there is a way to check if certain prime can ever divide $Q(x)$ for any value of x.

Suppose we need to find prime factors of integer n. For all primes $p \leq B$ we have to figure out if p divides $Q(x)$ for any x. If yes, then add p to FB. If not, skip it. We can write

$$p \mid Q(x)$$

$$p \mid \left(\left(\left\lceil \sqrt{n} \right\rceil + x \right)^2 - n \right)$$

$$\left(\left\lceil \sqrt{n} \right\rceil + x \right)^2 - n \equiv 0 \pmod{p}$$

$$\left(\left\lceil \sqrt{n} \right\rceil + x \right)^2 \equiv n \pmod{p}.$$

Therefore, if n is congruent to a square modulo p for any value of x, then p will be useful prime and should be added to FB. In other words, we need to check if check if n is quadratic residue modulo p, or, using the Legendre symbol, if $\left(\frac{n}{p}\right) = 1$.

In conclusion, it is better to generate the factor base in following way: For all primes $p \leq B$, check if $\left(\frac{n}{p}\right) = 1$ and if so, add p to FB. According to [1, p. 265], this process usually reduces the size of FB by about a half. The calculation of Legendre symbol can be done very quickly using the Euler's criterion formula.

2.3.2 Sieving

The most important addition of the quadratic sieve method is the sieving process. Instead of going through the sequence of values $Q(x_1), Q(x_2), Q(x_3), \ldots$, calculating the value of each $Q(x_i)$ individually and then checking if it is B-smooth number, Carl Pomerance described much more efficient process in [6]. First off, start by establishing some integer $s \in \mathbb{N}$ representing the size of

the sieving interval. Now, instead of calculating and checking each value of $Q(x_i)$ individually, we will work with the entire sequence of $Q(x)$ for all x from the **sieving interval** $[x_1, x_2, \ldots, x_s]$ simultaneously. If we do not have enough B-smooth relations at the end of this process, we continue with the values of x from the next sieving interval $[x_{s+1}, x_{s+2}, \ldots, x_{2s}]$ analogously. Generally, this process can be done with however many intervals it takes to find the required amount of B-smooth relations. For the sake of brevity, the phrase "$Q(x)$ inside the sieving interval" means "the value of $Q(x)$ using some x from the sieving interval".

The sieving portion of this algorithm comes as a result of the following observation. Instead of checking whether $Q(x)$ is divisible only by the primes in FB, we turn this process upside-down and, for each prime $p \in \text{FB}$, we find which values of $Q(x)$ inside the sieving interval p divides. Because $Q(x) = (\lceil\sqrt{n}\rceil + x)^2 - n$, we need the roots of modular polynomial

$$Q(x) = \left(\left\lceil\sqrt{n}\right\rceil + x\right)^2 - n \equiv 0 \pmod{p}. \tag{2.3}$$

According to [7, p. 16], solving the equation 2.3 usually leads to two integer solutions in $\mathbb{Z}_p$, lets call them x_p and $\dot{x}_p = p - x_p$. They can be found using the Tonelli–Shanks algorithm, or even better yet, using the Cipolla's algorithm (possible implementation described in [1, p. 102]).

Knowing that $p \mid Q(x_p)$ and $p \mid Q(\dot{x}_p)$, it is very easy to find all the other values of $Q(x)$ divisible by p inside the sieving interval by using the following principle:

$$p \mid Q(x) \iff p \mid Q(x + kp),\ k \in \mathbb{Z}. \tag{2.4}$$

The sieving strategy is now clear: For each prime $p \in \text{FB}$, calculate the roots x_p and $\dot{x}_p$ of the equation 2.3. Then, go through the sieving interval $[x_1, \ldots, x_s]$ and mark all x_i for which $p \mid Q(x_i)$ using the principle 2.4.

The incredible speed-up using this method manifests when the size of the factor base is much smaller than the size of the sieving interval. In that case, instead of doing some sort of trial division by p on all $Q(x)$ in the interval individually, we calculate only one slightly more complicated equation (roots from 2.3) for each $p \in \text{FB}$ and then, using simple addition, we get all $Q(x)$ divisible by p at once. The problem is, this method does not tell us how many times is $Q(x)$ divisible by p. Luckily, this is not a very big issue at all and it will be addressed at the end of the following section.

2.3.3 Logarithmic approximation

One aspect of the sieving process above was just briefly mentioned, but not clearly described. What does it actually mean to "mark" that some value of $Q(x)$ is divisible by p? The simplest way is to precompute all values of

$Q(x_1), \ldots, Q(x_s)$ for all values of x in the sieving interval in advance. Afterwards, to mark that $p \mid Q(x_i)$ would mean to divide the precomputed $Q(x_i)$ by p. At the end of the sieving, we could just check which values of $Q(x_i)$ in the interval are equal to 1, those represent B-smooth values. This approach has several disadvantages. Firstly, for each prime, we would have to find not only which values of $Q(x)$ are divisible by it, but also how many times, requiring some extra work. Secondly, and more importantly, the time complexity of integer division is very high and it would be very advantageous to substitute it with simple addition. The logarithmic approximation can achieve just that.

To do so, the logarithmic approximation uses the simple fact that

$$Q(x) = \prod_{i=1}^{k} p_i^{e_i}$$

$$\log(Q(x)) = \sum_{i=1}^{k} \log(p_i^{e_i}).$$

Instead of starting with the value of $Q(x)$ for each x in sieving interval at the beginning, dividing it by all primes from FB $Q(x)$ is divisible by and checking if it is equal to 1 at the end, we can create a variable for each $Q(x)$ inside the interval representing the sum of logarithms of all the primes dividing this particular $Q(x)$. All of these sums would be initially set to 0, and to mark that $p \mid Q(x_i)$ would now mean to just add the value of $\log(p)$ to the sum for this $Q(x_i)$. At the end, we check if this value is equal to $\log(Q(x))$ and if it is, $Q(x)$ is B-smooth number. To make this slightly faster, we should precompute and store the values of $\log(p)$ for every p in the FB.

Because working with logarithms requires swapping from integers to real numbers, this could be a problem for computer programs if we needed precise calculations. Fortunately, approximations are all we need. Addressing the problem from previous section, we just have the information that $Q(x)$ is divisible by some p from FB, but not how many times (i.e. we are missing the exponent of the prime). The usual solution to this, recommended for example in [1, p. 267], is to just ignore this. At the end, instead of comparing the resulting sum of logarithms we get from the sieving process for each $Q(x)$ to the exact value of $\log(Q(x))$, we compare it to some estimate of $\log(Q(x))$ called the **sieving threshold**. If the final sum of logarithms for some x in the sieving interval is bigger than the sieving threshold, $Q(x)$ is probably a B-smooth number. Therefore, at the end of the sieving process, we can look through all the final sums for all the values of x from the sieving interval, check if they are bigger than the sieving threshold and if so, check the $Q(x)$ manually for B-smoothness. By making the sieving threshold slightly smaller, we make up for the fact that we add just $\log(p_i)$ and not $\log(p_i^{e_i})$.

Proper selection of the sieving threshold is essential for this approach to work. If the sieving threshold is too big (i.e. strict), a lot of the actual B-smooth values of $Q(x)$ will be missed. If it is too small (i.e. loose), we will end

up manually checking a lot of values $Q(x)$ that are not actually B-smooth, therefore wasting a lot of time and potentially removing all the benefits of sieving.

2.3.4 Additional optimizations

In the section 2.3.1, we have removed all unnecessary primes from the factor base. However, it could be beneficial to ensure that some small primes are in the FB. For example, when we are manually checking if some $Q(x)$ is B-smooth, we need to divide it by all primes in the FB. Because of the way computers are built, it is incredibly fast to divide by the number 2 using the bit shifting operator. Also, when considering whether to add 2 to the factor base or not, we cannot use the Euler's criterion, because 2 is not an odd prime. To solve both of these problems, we could force the prime 2 to always be in the FB by multiplying the n we are trying to factorize by some small multiplier k. In [7, p. 13], it is advised to do so by letting $k = |n|_8$.

Another optimization can be made during the sieving process. Because the logarithmic values of small primes are so small, they do not contribute very much to the final sum for the particular numbers $Q(x)$ they divide. On top of that, they contribute most of the work in the actual sieving process. For example, consider the smallest prime $p = 2$. Every other value of $Q(x)$ in the sieving interval is going to be divisible by 2 and thus we will need to process half of the interval by adding $log(2)$ to it. Therefore, it can be very advantageous to skip sieving with small primes altogether and then make the sieving threshold slightly less strict to compensate for the small loss in the final sum.

2.3.5 Algorithm

All the differences between quadratic sieve and Dixon's method are in the way B-smooth relations are found. Everything hereafter (the usage of linear algebra and the extraction of solution using the greatest common divisor) is exactly the same and thus it will be skipped in the following algorithm description.

Unlike the Dixon's method, the time complexity of quadratic sieve algorithm is only conjectured. According to [8] in the analysis done by Carl Pomerance, the conjectured time complexity of factorizing composite n with quadratic sieve is

$$\mathcal{O}(\exp((\ln n \ln \ln n)^{1/2}))$$

or in the L-notation

$$L_n\left[\frac{1}{2}, 1\right],$$

making it sub-exponential algorithm with better constant component in the exponent than the Dixon's method. Interestingly enough, even though there

Algorithm 5: Quadratic sieve factorization method

Input: odd composite integer n, positive integer B, size of the sieving interval s
Output: two factors of n
FB $\leftarrow$ create factor base for primes $p \leq B, \left(\frac{n}{p}\right) = 1$
Relations $\leftarrow \emptyset$
Exponents $\leftarrow \emptyset$
while not enough B-smooth relations in Relations **do**
    `/* Beginning of sieving process. */`
    $I \leftarrow$ array of size s filled with 0
    $t \leftarrow$ sieving threshold estimate
    foreach p *in* FB **do**
        calculate the roots x_p and $\dot{x}_p$
        while $x_p \leq s$ **do**
            $I[x_p] \leftarrow I[x_p] + \log(p)$
            $x_p \leftarrow x_p + p$
        while $\dot{x}_p \leq s$ **do**
            $I[\dot{x}_p] \leftarrow I[\dot{x}_p] + \log(p)$
            $\dot{x}_p \leftarrow \dot{x}_p + p$
    `/* End of sieving process. */`
    foreach i *in* I **do**
        if $i > t$ **then**
            if $Q(x_i)$ is B-smooth **then**
                Add $Q(x_i) \equiv (\sqrt{n} + x_i)^2 \pmod{n}$ to Relations
                Add exponents from prime number factorization of $Q(x_i)$ to Exponents
    Prepare the next sieving interval
Enough B-smooth relations obtained, continue same as in algorithm 4

are many variations of the quadratic sieve method that offer considerable improvements (some of which are discussed later on in this thesis), the asymptotic time complexity estimate for them remains unchanged (they only improve some constants discarded in the asymptotic analysis).

2.4 Multiple polynomial quadratic sieve

The polynomial $Q(x)$ used in the simple version of quadratic sieve described above comes with one great disadvantage. The rate at which we find B-smooth relations usually starts fairly high, but decreases rapidly over time. This happens because $Q(x) = (\lceil \sqrt{n} \rceil + x)^2 - n$ and we start sieving with $x_1 = 1, x_2 = 2, \ldots$, thus the values of $Q(x)$ keep increasing with increasing x. And since smaller numbers are more likely to be B-smooth than bigger

numbers for the same B that is given, the longer the sieving takes, the less likely we are to find additional B-smooth relations. We could make smoothness bound variable B bigger, but that would only expand the factor base and in turn increase the amount of B-smooth relations needed. The proper remedy to this problem is to use more than one polynomial to generate B-smooth relations.

The general idea is as follows: We start the sieving process with some polynomial $Q_1(x)$ over fixed sieving interval, collecting B-smooth relations, and when the values of $Q_1(x)$ get too high and the probability of finding B-smooth relations decreases too much, instead of moving to the next interval, we switch to a slightly different polynomial $Q_2(x)$. We continue sieving and switching polynomials until we have gathered enough B-smooth relations.

2.4.1 Generating multiple polynomials

There are couple of differences between normal quadratic sieve and multiple polynomial quadratic sieve methods. The main difference is, as the name suggests, instead of sieving over single polynomial $Q(x) = (\lceil\sqrt{n}\rceil + x)^2 - n$ to gather all B-smooth relations, we use a family of polynomials called $Q_j(x)$. There are multiple different versions of MPQS using slightly different ways to generate the polynomials, but the most common one was suggested by Peter Montgomery and further modified by Robert Silverman in [9]. I followed the method described by Carl Pomerance in [1, p. 273], which is a slight modification of the Silverman method. The new family of polynomials is defined as

$$Q_j(x) = a_j x^2 + 2b_j x + c_j$$

with some coefficients $a_j, b_j, c_j \in \mathbb{Z}$ satisfying certain conditions. First of all, it is very important for a_j to be square of a prime for reasons discussed in a moment. Let us call that prime d_j. It is also important that $0 \leq b_j < a_j$ and

$$b_j^2 - a_j c_j = n.$$

This can be rewritten as

$$b_j^2 \equiv n \pmod{a_j} \tag{2.5}$$

meaning b_j can exist only if $\left(\frac{n}{p}\right) = 1$ for every prime p that divides a_j. Since the only prime dividing a_j is d_j, it is only needed that $\left(\frac{n}{d_j}\right) = 1$.

With the condition for b_j satisfied, we can write

$$a_j Q_j(x) = a_j^2 x^2 + 2a_j b_j x + a_j c_j = (a_j x + b_j)^2 - n$$

and therefore

$$(a_j x + b_j)^2 \equiv a_j Q_j(x) \pmod{n}. \tag{2.6}$$

Assuming $d_j \leq B$, a_j is also a B-smooth number. Therefore, all we need is to find values of x for which $Q_j(x)$ is B-smooth number and we can use the congruence 2.6 to generate B-smooth relations. In order to get an instance of the congruence 2.2 to get the solution, we use these relations identically as we used them in quadratic sieve and Dixon's methods, but with one small exception. To get a square on the right-hand side of the congruence, we make use of the fact that because a_j is a square, the product of squares is a square as well. When we multiply multiple congruences of type 2.6 together, the product of all a_j on the right-hand side has to always be square. Therefore, in the linear algebra step, we do not need to work with the prime exponents of the entire right-hand side $a_j Q_j(x)$, but only with the prime exponents of $Q_j(x)$.

Now we know what the new polynomials generally look like, but we do not know how to actually create them, in other words, how to actually pick the coefficients a_j, b_j, c_j such that the values of $Q_j(x)$ are relatively small over the sieving interval. To achieve that, first we need to consider the minimum of the function $Q_j(x)$. Because $a_j > 0$, the function is a convex quadratic function and it has a minimum precisely where the first derivative of $Q_j(x)$ is equal to 0, which is at $x = -b_j/a_j$.

In the quadratic sieve method, only positive integers for x were used in the sieving intervals, starting at $x_1 = 1$ and increasing linearly. Considering the fact that

$$\lim_{x\to\infty} Q_j(x) = \infty \wedge \lim_{x\to-\infty} Q_j(x) = \infty,$$

and that the minimum of $Q_j(x)$ is a negative integer, it would be very advantageous to shift the beginning of the sieving interval to negative values as well. That way, the values of $Q_j(x_1)$ will start off large, slowly decrease as x approaches $-b_j/a_j$, where the $Q_j(-b_j/a_j)$ hits the minimum and starts increasing again. This would maximize the amount of small values of $Q_j(x)$ over the sieving interval, making it more likely to find B-smooth relations. However, using $Q_j(x)$ with negative values of x comes with one problem that needs to be fixed and is further addressed in the following section 2.4.2.

As a result, suppose the sieving interval is $[-M, M]$ for some variable M. It can be seen that for every one concrete polynomial $Q_j(x)$, we will sieve with precisely $2M + 1$ values of x, then switch to the next polynomial $Q_{j+1}(x)$. The goal is to choose the coefficient a_j and with it calculate b_j, c_j such that the values of $Q_j(x)$ are kept relatively small over $[-M, M]$. One possible approach, suggested in [1], is to let the minimum and the maximum of $Q_j(x)$ over $[-M, M]$ approximately equal each other in absolute value, in other words, let the minimum and the maximum of $Q_j(x)$ over $[-M, M]$ have approximately the same magnitude, but opposite sign.

Given that the largest value of $Q_j(x)$ over this interval is at the endpoints

$x = \pm M$, we estimate that

$$Q_j(\pm M) = a_j M^2 \pm 2b_j M + c_j = \frac{a_j^2 M^2 \pm 2a_j b_j M + b_j^2 - n}{a_j} \approx \frac{a_j^2 M^2 - n}{a_j}.$$

It was already established that the minimum is at $x = -b_j/a_j$ and thus

$$Q_j\left(-\frac{b_j}{a_j}\right) = \frac{a_j b_j^2}{a_j^2} - \frac{2b_j^2}{a_j} + c_j = \frac{-b_j^2 + a_j c_j}{a_j} = -\frac{n}{a_j}.$$

Therefore, we need to choose a_j such that

$$\frac{a_j^2 M^2 - n}{a_j} \approx \frac{n}{a_j}$$

which can be solved by letting

$$a_j \approx \frac{\sqrt{2n}}{M}.$$

Considering that a_j should be square of a prime d_j, we can roughly define d_j and a_j as

$$d_j \approx \sqrt{\frac{\sqrt{2n}}{M}}$$

$$a_j = d_j^2.$$

There is one more condition to fulfill, namely that d_j needs to be prime. For the sake of efficiency, this is usually achieved by generating probable primes around $(2n)^{1/4}/M^{1/2}$ using Miller-Rabin or similar primality test. With this value of d_j, we can calculate the value of a_j and in turn the values of b_j and c_j. Each time new polynomial is needed, a new d_j is chosen and the rest of coefficient calculated from it. To get the value of b_j from a_j, the congruence 2.5 needs to be solved. If the modulus a_j was prime, this congruence would be easy to solve. Unfortunately, a_j is not a prime, but a square of prime d_j. One method to solve the congruence 2.5, described in [7, p. 20], is to first solve the congruence

$$b_j^2 \equiv n \pmod{d_j}$$

using Tonelli-Shanks or Cipolla's algorithm, then lift the result by Hensel lemma to higher prime power (more on "Hensel lifting" can be found in [1, p. 105]. This leads to two solutions for b_j, both describing the same polynomial, so only one of them is used. The value of c_j is then calculated simply with $c_j = (b_j^2 - n)/(a_j)$.

2.4.2 Negative values

When trying to implement this version of multiple polynomial quadratic sieve, one could run into a problem with the values $Q_j(x)$ is returning. Because we are now not restricting the sieving interval to positive integers for x only, $Q_j(x)$ can return negative integers as well. We need to be able to check the B-smoothness of a negative number, as well as to process them for the linear algebra step. This is done by adding one special "prime" to our factor base, the $p_0 = -1$. Its prime exponent e_0 will be zero if the B-smooth number is positive or one if the B-smooth number is negative. It is also important not to forget to include the exponent in the linear algebra step, making sure that in the final multiplication of B-smooth relations, the sum of all exponents e_0 is even.

2.4.3 Potential for improvement

Ironically enough, big potential bottleneck of the MPQS method is the polynomial generation process itself, or rather what follows it. Every time, after some new polynomial is created, the roots x_p and $\dot{x}_p$ of $Q_j(x) \equiv 0 \pmod{p}$ for each prime p from the factor base have to be computed. This would take fairly long indeed, using quite complicated Cipolla's algorithm to perform root extraction for every prime after each polynomial change. Luckily, this process can be sped up by using a simple suggestion found in [7]. Before the any of the polynomial generation or sieving begins, calculate and save the values of $r_p = \sqrt{n} \pmod{p}$ for all primes p in factor base. There should be two solutions in $\mathbb{Z}_p$, r_p and $\dot{r}_p = p - r_p$, but saving only the first one will suffice. With these values precalculated, we do not need to perform any more root extractions when the polynomial switches. The roots of $Q_j(x)$ can be found simply using

$$x_p = \left| \frac{r_p - b_j}{a_j} \right|_p , \dot{x}_p = \left| \frac{p - r_p - b_j}{a_j} \right|_p .$$

By far the most time consuming segment of the calculation above is the modular inverse $a_j^{-1} \pmod{p}$. Currently, each time the polynomial switch occurs, different polynomial coefficient are generated, including the a_j. Should we wish to make this process more time efficient, we need to find a way to be able to reuse the same a_j for more than one polynomial, saving time on the modular inverse calculations. The self-initializing quadratic sieve method addresses just that.

2.4.4 Algorithm

The general description of the MPQS method in pseudo-code follows in the algorithm 6.

Algorithm 6: Multiple polynomial quadratic sieve method

Input: odd composite integer n, positive integer B, bounds of the sieving interval M
Output: two factors of n
FB $\leftarrow -1$
FB $\leftarrow$ add all primes $p \leq B, \left(\frac{n}{p}\right) = 1$
Relations $\leftarrow \emptyset$, Exponents $\leftarrow \emptyset$
$d_0 \leftarrow (2n)^{1/4}/M^{1/2}$
while not enough B-smooth relations in Relations **do**
    `/* Create new polynomial and start sieving. */`
    $d_j \leftarrow$ next probable prime from previous d_{j-1} such that $\left(\frac{n}{d_j}\right) = 1$
    $a_j, b_j, c_j \leftarrow$ calculate polynomial coefficients from d_j
    $I \leftarrow$ array of size $2M + 1$ filled with 0
    $t \leftarrow$ sieving threshold estimate
    foreach p *in FB,* $p \neq -1$ **do**
        calculate the roots x_p and $\dot{x}_p$ in $\mathbb{Z}_p$
        offset the roots x_p and $\dot{x}_p$ so they start at the beginning of the sieving interval in $[-M, -M + p]$
        while $x_p \leq M$ **do**
            $I[x_p] \leftarrow I[x_p] + \log(p)$
            $x_p \leftarrow x_p + p$
        while $\dot{x}_p \leq M$ **do**
            $I[\dot{x}_p] \leftarrow I[\dot{x}_p] + \log(p)$
            $\dot{x}_p \leftarrow \dot{x}_p + p$
    `/* End of sieving process. */`
    foreach i *in* I **do**
        if $i > t$ **then**
            if $Q_j(x_i)$ is B-smooth **then**
                Add $Q_j(x_i) \equiv (\sqrt{n} + x_i)^2 \pmod{n}$ to Relations
                Add exponents from prime number factorization of $Q_j(x_i)$ to Exponents
Enough B-smooth relations obtained, continue same as in algorithm 4

2.5 Self-initializing quadratic sieve

As was mentioned above, the main idea behind the self-initializing quadratic sieve method is to change the way coefficients a_j, b_j are calculated. The goal is to be able to use the same a_j for multiple different values of b_j, thus skipping the modular inverse calculation for that many polynomials. Of course, the conditions for coefficients from section 2.4.1 still need to be met for the rest of the method to work. Looking back at the congruence 2.5 used to calculate b_j from a_j, it only had two solutions, both describing the same polynomial. That is because we defined a_j as a square of another prime. But, if a_j is a product of multiple primes

$$a_j = \prod_{i=1}^{s} d_{j,i},$$

then, according to the Chinese remainder theorem, the congruence 2.5 has 2^{s-1} solutions for b_j describing different polynomials. Much like the prime d_j used before, all the primes $d_{j,i}$ have to be odd and satisfy $\left(\frac{n}{d_{j,i}}\right) = 1$.

The theory behind the SIQS method and its implementation proved to be much more complicated than initially expected when creating the thesis assignment. After a consultation with my thesis supervisor, we decided not to focus on SIQS and spend more time on MPQS, large prime optimization and parallelization instead.

2.6 Large prime optimization

The large prime optimization can be done for any version of the quadratic sieve method – the basic one, MPQS or SIQS. All of these methods have something in common. During the process of sieving the sieving interval, one way or another, the relations that are probably B-smooth are marked. After the sieving of this interval is finished, some method (usually trial division by all primes from factor base) is used to check which ones are actually B-smooth and which ones are not. The actual B-smooth relations are collected, but the "almost" B-smooth relations are discarded and the time spent processing them is essentially wasted. The large prime optimization remedies just that.

Suppose we find two partial B-smooth relations with the same prime P that is over the smoothness bound B

$$x^2 \equiv \prod_{i=1}^{k} p_i^{e_i} \cdot P \pmod{n}$$
$$\dot{x}^2 \equiv \prod_{i=1}^{k} p_i^{\dot{e}_i} \cdot P \pmod{n}.$$

If both of these partial B-smooth relations would be found/marked during sieving, they would be checked for B-smoothness and discarded afterwards, because they both contain factor $P > B$. However, by multiplying them together, we get

$$(x \cdot \dot{x})^2 \equiv \prod_{i=1}^{k} p_i^{e_i + \dot{e}_i} \cdot P^2 \pmod{n},$$

which can actually be used as a substitute for one legitimate B-smooth relation. The P prime factor is already square, so it will have to be square if it is used in the final multiplication. Although the probability of finding two partial B-smooth relations for a given shared prime factor P is fairly small, the probability of having some two partial B-smooth relations that have some shared prime factor P in a set of many partial B-smooth relations is much larger (given the principles behind the *birthday paradox*). In a very similar manner, one could attempt to use 2-partial or even 3-partial B-smooth relations for 2-large or 3-large prime optimization.

2.7 State-of-the-art

The goal of this section is to go over some of the already existing open-source implementations of quadratic sieve methods.

2.7.1 Msieve

Msieve is a popular project written mainly by Jason Papadopoulos in the C language, using the GMP library to handle operations with big integers. It contains a static library called `libmsieve.a`, implementing two different algorithms for large number factorization, and a `msieve` demo application that uses the library. The two algorithms implemented are the self-initializing quadratic sieve with 2-large prime optimization and the number field sieve, so far the most advanced method to factorize large numbers.

According to the documentation, the library is multithread aware and also contains support for computing in a cluster via MPI, although it seems that is only for the linear algebra calculations. The most up to date version and documentation can be found in [10].

2.7.2 YAFU

Yet Another Factorization Utility, more known as its abbreviation YAFU, is one of the most efficient open-source implementations of the quadratic sieve method. It contains an implementation of the simple QS, MPQS and SIQS methods, all with the single large prime optimizations. According to their own homepage, "YAFU's SIQS implementation is the fastest publicly available quadratic sieve implementation" [11]. It also contains an implementation of

the number field sieve method and very fast implementation of the sieve of Eratosthenes, all written in the C programming language. All major stages of the factorization in SIQS are multi-threaded. It does not support cluster computation however.

2.7.3 MPQS implementation by Peter Kováč

The master's thesis [12] by Peter Kováč looks at few different versions of the quadratic sieve method, the most advanced one being the MPQS with single large prime optimization. Implementation is written in C++ language, using GMP to handle large integer operations. Neither the thesis nor the implementation focus on any parallelization of the algorithm.

Chapter 3

Quadratic sieve implementation

3.1 General implementation details

The implementation is split into two separate programs. The first program, called `qsieve`, implements all of the following methods in one-threaded, non-parallel execution style:

- trial division,
- Dixon's method,
- quadratic sieve,
- multiple polynomial quadratic sieve,
- large prime multiple polynomial quadratic sieve.

The second program is called `qsievempi` and implements only the large prime multiple polynomial quadratic sieve method, but in parallelized execution suitable for distributed computing in a cluster with openMP and MPI libraries.

The two programs have slightly different usage. Since the `qsieve` program is expected to be run interactively from terminal, the usage can be printed simply by running just `./qsieve` or `./qsieve -h`. The `qsievempi` is not expected to be run interactively, but instead to be added to some computer cluster's job queue, hence why there is no option to just print usage. For `qsieve`, the required options depend on the method that has been selected. For obvious reasons, since the `qsievempi` only implements one method, there is no option for method selection and the rest of the required options are exactly the same as `qsieve` with the the large prime multiple polynomial quadratic sieve method selected. If something goes wrong during the execution, error messages are printed to the standard output or to the output file if it is provided as option.

3.2 Technology used

I chose the C++ programming language for the implementation. The main reasoning behind that decision was that the language is very efficient and fairly easy to use for this task, with a set of already existing libraries to help with parallelization and some other problematic parts. I used the C++11 standard, mostly because I wanted to take advantage of the move semantics this standard offers. With them I can make the implementation slightly more efficient by cutting down the time it takes to make copies of some temporary object in the program.

Because the entire point of factorization of large integers is to work with large numbers, I needed some way to store them in the program. The fundamental C++ data types would not do, because they have a fixed size and would overflow easily, causing unexpected results or errors. I opted to use the GNU Multiple Precision Arithmetic Library, also known as GMP, to handle the large integer operations. It is a free open-source library under the dual GNU LGPL v3 and GNU GPL v2 license, written in C programming language [13]. For the implementation, I used GMP version 6.2.1.

I also decided to use some of the Boost libraries to help with several parts of the implementation. Boost is a collection of various C++ libraries distributed under the Boost Software License [14], providing some additional utility that is missing from the standard libraries. Most of the libraries in Boost are header-only libraries, meaning they do not have to be built separately, instead they can be simply imported into the project's source code by the `#import` compiler preprocessor directive. The Boost version 1.75.0 was used during implementation.

One of the tasks for this thesis was to discuss and implement parallelism of the algorithms, then measure the results on the faculty server STAR. STAR is a hybrid distributed-shared memory computer cluster, meaning it contains multiple nodes (each node has its own memory – distributed memory architecture) and each node contains multiple processors with multiple cores per processor (shared memory architecture). To successfully implement parallelization for STAR, I had to address both of these architectures separately.

To help with the shared memory architecture, I used the API standard for shared memory programming called openMP [15]. By using openMP I was able to split some parts of the original sequential algorithm into multiple threads, causing it to be computed in parallel on multi-core processors.

For the parallelization with distributed memory architecture, I used the Message Parsing Interface standard (or MPI for short) [16]. This is widely used standard for distributed computing, defining a set of functions for interprocess communication via message passing and other tasks.

Some of the calculations during the sieving process require finding the roots of a modular polynomial. To achieve that, several different algorithms can be used. I chose to use the Cipolla's algorithm implementation from

the Project Wycheproof [17]. It is a project developed and maintained by the Google Security Team under the Apache 2.0 license. The algorithm is originally written in the Java programming language, so I rewrote it into the C++ language and modified it slightly to fit my needs.

3.3 Trial division implementation

The files `TrialDivision.h` and `TrialDivision.cpp` contain very simple implementation of trial division factorization algorithm shown in section 1.1.2. The main purpose of this implementation is be able to show the juxtaposition of this very basic/trivial method and some of the other, more elaborate and complicated methods. There are no required options for this method.

3.3.1 Trial division usage

The figure 3.1 is an example of how the trial division method can be run. The `-m` option selects the trial division method, all the numbers to be factorized are read from the input file specified by the `-i` option and then factorized one by one until end of file is encountered or until some error occurs. The entire output is printed out to the output file specified by the `-o` option. The program will read from standard input and print to standard output if no input/output files are provided.

Figure 3.1: Trial division usage example

```
./qsieve -m td -i inputfile.txt -o outputfile.txt
```

3.4 Dixon's method implementation

To factorize with any method discussed in this thesis (not counting the trial division), 4 different steps/stages need to be implemented. The stages are in the following order

1. initialization,
2. gathering relations / sieving,
3. linear algebra,
4. obtaining the results.

Most of the Dixon's method, which was described in section 2.2, is implemented in the files `DixonMethod.h` and `DixonMethod.cpp`. The abstract class

in file `ILinSolver.h` is used to solve the linear algebra stage, with the concrete implementation of simple GEM in the files `SimpleGEM.h` and `SimpleGEM.cpp`.

Because using the GMP library with the C++ language can be very cumbersome because of its C-like design, I have decided to use the *mpz_int* class from the Boost multiprecision library instead. The library still uses GMP in the backend, but the design of its frontend is more suitable for C++ language. Additionally, using the *mpz_int* objects makes them compatible with the rest of the useful Boost integer functions, functions to calculate the modular inverse, square root and others.

3.4.1 Initialization

The initialization of the Dixon's method is very simple. The only thing that needs to be done is to create the factor base. The factor base is simply a set of all primes that are less or equal to the smoothness bound B. Although there is one question remaining: How to generate the primes? In the current version of the implementation, the list of primes in a file has to be provided via the option `-p`. In that file, the first line needs to be some integer k followed by k lines with one prime on each line in ascending order. At the beginning of the program start-up, this file is loaded into memory and then used to generate the factor bases during factorization. The program also contains a static list of the first 1000 primes in memory, so for $B \leq 1000$, the file with primes does not have to be provided.

I used the sieve of Eratosthenes to generate 5 files in the `primes` directory containing primes in the correct format to be used according to need. This was meant as a more of a temporary solution and a possible point for future improvement is to integrate the sieve of Eratosthenes algorithm in the program to generate the required primes on the fly.

3.4.2 Gathering relations

The goal of this stage is to gather enough B-smooth relations for the next stage. What exactly constitutes enough relations is going to be discussed in the following section 3.4.3. Together with the values for the left-hand side and the right-hand side for each relation, it is also important to store its prime exponents for the linear algebra step. To store these three components together, the class *BSmoothRelation* in the files `BSmoothRelation.h` and `BSmoothRelation.cpp` was created. The left-hand side and the right-hand side values are stored as *mpz_int* objects because of the unlimited precision, the prime exponents are stored as *dynamic_bitset* objects from Boost dynamic bitset library for reasons that will be obvious shortly.

The process itself is straightforward. Starting at the value of $x_0 = \left\lceil \sqrt{n} \right\rceil$ and incrementing by one, calculate $Y_j = |x_j^2|_n$ and check if Y_j is B-smooth. The B-smooth check itself is just a slight modification of trial division by all

primes from the factor base. Check if the prime $p \in \text{FB}$ divides Y_j and if it does, divide Y_j by p until it is no longer divisible by it. Store how many times p divided Y_j to use it later as the prime exponent of the prime p. If Y_j is equal to 1 at the end of the process, the original Y_j was B-smooth.

3.4.3 Linear algebra

Just to refresh the memory, the main objective of this stage is this: Given a set of enough B-smooth relations, figure out its subset for which the multiplication of all relations yields an even value for all prime exponents (e_i from definition 1.11). This is the same as saying that for each prime $p \in \text{FB}$, the sum of its prime exponent from all relations in the subset needs to equal to zero in $\mathbb{Z}_2$.

In order to achieve that, I used the method suggested in [1, p. 263] and [18]. Suppose that the size of the factor base is k. The idea is to think of the prime exponents of each B-smooth relation as one vector from $\mathbb{Z}_2^k$, for example

$$\vec{v}_i = (e_{i,1}, e_{i,2}, \ldots, e_{i,k})$$

for the ith relation. If we could get a sum of such vectors that gives the zero vector $\vec{0}$ as a result, we would fulfill the objective stated above.

Because we are working in $\mathbb{Z}_2$, sum of some subset of vectors is the same as linear combination of all vectors, where the vectors in the subset are multiplied by the scalar 1 and the rest is multiplied by the only other scalar, 0. Therefore we need to get the zero vector $\vec{0}$ as some linear combination of the other vectors. In other words, supposing that the set of all vectors is of the size m, there need to be some scalars $a_1, a_2, \ldots a_m \in \mathbb{Z}_2$ such that

$$a_1\vec{v}_1 + a_2\vec{v}_2 + \cdots + a_m\vec{v}_m = \vec{0}$$

and at least one scalar is equal to 1. If this linear combination exists, all the vectors that have their corresponding scalars equal to 1 are representing precisely the B-smooth relations from the desired subset.

To ensure that the above-mentioned linear combination exists, the set of all vectors has to be linearly dependent. Thus the size of the set (meaning the amount of B-smooth relations gathered) needs to be at least one larger than the dimension of the vector space $\mathbb{Z}_2^k$. This finally answers the question of "how many B-smooth relations is enough", and the answer is "at least $k+1$". If we get exactly $k+1$ relations, theoretically we have at least one solution, but that solution can lead to trivial factorization. In practice however, this rarely happens and we usually get multiple solutions from just $k+1$ relations, one of which is bound to be non-trivial.

To implement this part, I used a slight variation of the extended Gaussian elimination method described by Shoup in [2, p. 391]. I started off by creating an **exponent matrix** $\boldsymbol{M} \in \mathbb{Z}_2^{k+1\times k}$ on the left side and identity matrix on

the right side

$$\left(\begin{array}{cccc|cccc} e_{1,1} & e_{1,2} & \cdots & e_{1,k} & 1 & 0 & \cdots & 0 \\ e_{2,1} & e_{2,2} & \cdots & e_{2,k} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ e_{k+1,1} & e_{k+1,2} & \cdots & e_{k+1,k} & 0 & 0 & \cdots & 1 \end{array}\right)$$

Each row in the exponent matrix represents the prime exponents of one B-smooth relation in $\mathbb{Z}_2$. If in the relation with x_i^2 the prime p_j had exponent with odd value, then $e_{i,j} = 1$, otherwise $e_{i,j} = 0$. The objective now is to find a combination of relations (rows) that yields $\vec{0}$ in $\mathbb{Z}_2$. The Gaussian elimination method can be used with some caveats to get the left side (the exponent matrix) into upper triangular matrix. Because the matrix calculations are in $\mathbb{Z}_2$, multiplying the rows by scalars (which is normally done in GEM) does not make any sense, because the only scalars are 0 and 1. The only GEM operations left are swapping rows and adding one row to a different one. Adding in $\mathbb{Z}_2$ can be done very efficiently by using the bitwise xor operator in the C++ language. Now it is apparent why I used the *dynamic_bitset* to store the prime exponents of B-smooth relations. It is because each exponent is stored in $\mathbb{Z}_2$ as one bit that is either turned on or off and the class *dynamic_bitset* supports fast xor operations with its objects.

After Gaussian elimination method done, all rows that are equal to zero on the left-hand side represent linear combinations of B-smooth rows with even exponents and their respective rows on the right side represent which individual relations were used to get that particular linear combination. This also means that the amount of different solutions is equal to the amount of zero vectors in the upper triangular exponent matrix.

The use of GEM can get very cumbersome when factorizing very large numbers. The reason is that with factorizing larger number comes bigger B parameter which in turn makes the size of the factor base k bigger, making the exponent matrix grow in both dimensions. Luckily, the use of GEM is only one of many options for the linear algebra stage. Other, much more advanced methods exist that take into consideration the sparseness of the exponent matrix. One of such methods can be found in [7].

3.4.4 Obtaining the results

Obtaining the results is the last stage in the entire factorization process. One valid solution is a subset of B-smooth relations obtained through the previous stage. Lets assume the subset has a size of l and the relation

$$x_i^2 \equiv Y_i \pmod{n}$$

is the ith relations in the subset. To get the final result, a couple of simple multiplications and a square root computation has to be used

$$x = \prod_{i=1}^{l} x_i,\ Y = \prod_{i=1}^{l} Y_i,\ y = \sqrt{Y}$$

and then one of the factors $p = \gcd(x - y, n)$. If $p = 1$ or $p = n$, the factor is trivial and the next solution obtained through the previous stage has to be used analogically, if not, the final result is p and $q = n/p$.

3.4.5 Dixon's method usage

One example of Dixon's method usage can be found in the figure 3.2. The required option `-b` specifies the smoothness bound B. If the value for B is larger than 7919, a text file with primes in the correct format must be provided via the option `-p`. These text files can be found in the directory `primes` in the source files on the enclosed CD.

Figure 3.2: Dixon's method usage example

```
./qsieve -m dm -b 50000 -p primes.txt -i inputfile.txt
```

3.5 Quadratic sieve implementation

The Dixon's method serves as a foundation upon which all of the other quadratic sieve methods build. Therefore, a large portions of the implementations of these methods are exactly the same as in the Dixon's method implementation. Because of that, in these next few sections I will only describe the parts that differ from the Dixon's method. More specifically, I will only describe the changes in the stages

1. initialization,
2. sieving.

The stages

3. linear algebra,
4. obtaining the results

remain exactly the same. This section focuses on the implementation of the quadratic sieve method described in 2.3.

3.5.1 Initialization

The initialization in the quadratic sieve is slightly more complicated. First, the number n that is to be factorized is multiplied by small constant $k = |n|_8$ for the reasons discussed in 2.3.4. The rest of this implementation then works with this new value of kn as if it was the original n. Then, the factor base is built, but the process is slightly different than in Dixon's method implementation. Not all primes $p \leq B$ are added, but only the primes that also satisfy $\left(\frac{n}{p}\right) = 1$. The Euler's criterion is used for this calculation.

Finally, the *buildLoops()* function is called. This function creates objects of the class *SievingLoop*, each of them representing one of the results of the congruence 2.3 for all primes p in the factor base. In order to get the results, some algorithm to calculate the modular square root has to be used. I used the implementation of the Cipolla's algorithm described in the section 3.2. The class *SievingLoop* has only one method called *sieveInterval(interval)*, which loops through the *interval* array parameter during sieving, adding $\log(p)$ to its appropriate elements.

3.5.2 Sieving

The sieving is the most important change from the Dixon's method implementation. In the very beginning, an array representing the sieving interval is created. Because the array will be storing results of logarithmic operations, it has to be an array of floating point data types. The size of the array is based on the interval size option given via `-s`, the default being 100000. The *intervalStart* variable represents x_1 in the current sieving interval $[x_1, x_2, \ldots, x_s]$. At the beginning of sieving it is given the initial value of $\left\lceil \sqrt{n} \right\rceil + 1$.

Then the actual sieving of the current sieving interval commences. For each *SievingLoop* object the *sieveInterval(interval)* gets called with the current interval array as parameter. Inside the function is a simple for loop going through the array adding precomputed $\log(p)$ to appropriate elements using the principle 2.4.

Afterwords, the *sievingThreshold* variable has to be calculated. It could be calculated for each x in the sieving interval individually, which would be much more precise but very inefficient. Because the *sievingThreshold* is supposed to be only a rough estimate of $\log(Q(x))$, it can be calculated only once per entire sieving interval. To calculate it I chose the following equation

$$sievingThreshold = \log(intervalStart^2 - n) - \log(maxFactor),$$

where *maxFactor* is the larges prime in the factor base. In the equation above, the first logarithm is just a calculation of the $log(Q(x_1))$ for the first x_1 inside the sieving interval (and approximation for all the other values of x in current interval), but what about the second logarithm and the subtraction? It is there simply to make the approximation less strict to compensate for the fact

that the sieving ignores higher powers of primes. The idea is that if we take the value of some $Q(x)$, divide it only once by all primes from FB that divide that number and end up with a number that is less or equal to the larges prime in the factor base, the original $Q(x)$ has to be B-smooth.

Lastly the sieving interval array is looped over and checked for elements that are greater than the*sievingThreshold*. Suppose the ith element in the array is greater than the*sievingThreshold*, that means the value of $Q(x_i)$ is probably B-smooth. The value is therefore checked for B-smoothness and the relation added to the collection of relations if it passes.

After the current sieving interval $[x_1, x_2, \ldots, x_s]$ is sieved over, but not enough B-smooth relations are found, the interval array is zeroed out and the *intervalStart* variable is increased by the interval size, signifying that the interval has shifted to $[x_{s+1}, x_{s+2}, \ldots, x_{2s}]$. The entire sieving process is repeated until enough B-smooth relations are found.

3.5.3 Quadratic sieve method usage

For the quadratic sieve usage, there is only one new option available. It is the option `-s`, used to specify the size of the sieving interval. If the option is left out, the default value of 100000 is used as the size of the sieving interval instead. On top of that, all of the other options from previous method also remain with the same functionality.

Figure 3.3: Quadratic sieve usage example

```
./qsieve -m qs -b 50000 -s 100000 -p primes.txt -i inputfile.txt
```

3.6 Multiple polynomial quadratic sieve implementation

Once again, the multiple polynomial quadratic sieve method builds upon the quadratic sieve method, moderately improving it. Because of that, some of the parts of these implementations are exactly the same. In this section, I will explain the implementation details of the MPQS method described in 2.4, focusing mainly on the parts that differ from the implementation of QS in the previous section.

3.6.1 Initialization

Because of the usage of multiple polynomials during sieving, the initialization stage had to be changed slightly. In quadratic sieve, the sieving is done only on single polynomial $Q(x)$, so the roots of congruence $Q(x) \equiv 0 \pmod{p}$

from 2.3 can be found for each $p \in$ FB during the initialization process. In contrast, the MPQS uses multiple polynomials $Q_j(x)$, periodically switching from old one to a new one, and therefore the roots have to be calculated each time after switching to new polynomial occurs. This includes rather lengthy algorithm to calculate the modular square root, making the process of switching polynomials time consuming. Luckily, as discussed in section 2.4.3, the modular square root in the calculation does not depend on the coefficients a_j, b_j or c_j and thus it can be precomputed and stored during the initialization stage and used later on, making the switch to new polynomial much faster.

Therefore, the biggest change is that the *buildLoops()* cannot be called during the initialization. Instead, after the factor base is built, the modular square roots $r_p = \sqrt{n} \pmod{p}$ are calculated and stored for each prime $p \in$ FB. Additionally, the values of $\log(p)$ are also stored and ready for further use during sieving. The last change made to the initialization stage was the addition of -1 into the factor base at the very first position.

3.6.2 Sieving

The meaning of the `-s` option is slightly different in MPQS. It represents the variable M from the section 2.4.1. In other words, instead of the option `-s` representing the size of the entire sieving interval, it represents only half of the interval (given that the size of interval $[-M, M]$ is $2M + 1$). The sieving itself starts with the computation of the polynomial coefficients a_j, b_j and c_j using the formulas described in 2.4.1. Then, the *buildLoops(a_j, b_j)* function is called, which prepares the *SievingLoop* objects. The rest of the sieving stage implementation is almost the same as in QS implementation, but with two major differences.

First difference in the sieving stage is that once the interval $[-M, M]$ is sieved through using the polynomial $Q_j(x)$, instead of continuing with the next interval on the same polynomial (which is what was essentially happening in QS), the program switches to new polynomial $Q_{j+1}(x)$ while keeping the sieving interval at $[-M, M]$. This ensures that the values of $Q_j(x)$ do not keep increasing indefinitely as the sieving continues on, which is the whole point of using multiple polynomials.

The second difference is the calculation of the *sievingThreshold* variable. As was mentioned before, proper estimate of the *sievingThreshold* is crucial for the performance of the algorithm. If the estimate is too low, many of the $Q_j(x)$ values will be marked as probably B-smooth without actually being B-smooth. All of these values will have to be checked for B-smoothness, wasting a lot of time in the process. On the other hand, if the estimate if too big, a lot of $Q_j(x)$ values that are actually B-smooth will be missed. The way the threshold is calculated largely depends on the way the polynomials were constructed. In [1, p. 274], Carl Pomerance writes that the values of $Q_j(x)$ on

the interval $[-M, M]$ are bounded by $(M/\sqrt{2})\sqrt{n}$. Because I used his method of generating polynomials, I estimated the values of $\log(Q_j(x))$ as

$$sievingThreshold = \log(M\sqrt{n}) - \log(maxFactor).$$

3.6.3 Multiple polynomial quadratic sieve method usage

The usage of the MPQS implementation is exactly the same as with the QS implementation, with the exception of the `-m` option specifying which method to run of course. A small difference is that instead of representing the size of the sieving interval, the option `-s` now represent the M variable from the definition of MPQS. The size of the entire sieving interval in the figure 3.4 is therefore 200001 instead of just 100000.

Figure 3.4: Multiple polynomial quadratic sieve usage example

```
./qsieve -m mpqs -b 50000 -s 100000 -p primes.txt \
    -i inputfile.txt
```

3.7 Large prime optimization of MPQS

The singe large prime optimization, described in section 2.6, can be implemented in any version of quadratic sieve methods with just a few changes. I chose to implement it for the more efficient MPQS method to see how much it can be further improved.

3.7.1 Initialization

The initialization stage is identical to the MPQS initialization, with one small exception. The bounds for the large prime P from the partial B-smooth relation need to be established. The lower bound $P > B$ is already given by the definition of a partial relation, but what should be the upper bound? Setting the upper bound to $P \leq B^2$ can be very advantageous, because any number P in the range $B < P \leq B^2$ that is not divisible by any prime $p \leq B$ has to automatically be a prime. Setting the upper bound this big is impractical, as "It is noticed in practice, and this is supported too by theory, that the larger the large prime, the less likely for it to be matched up" [1, p. 272]. In my implementation I used the suggestion from [7] to set the upper bound of P to $128 \cdot B$.

3.7.2 Sieving

The sieving stage also has only one small change. Before, when some number $Q_j(x)$ was being checked for B-smoothness, it got progressively divided by all primes in factor base until some remainder was left. If that remainder was anything other than 1, the original number was not B-smooth. With the large prime optimization, there is one addition at the end. If the remainder is bigger than 1, but smaller than the upper bound for P, the original $Q_j(x)$ is "almost B-smooth" and can be collected as partial B-smooth relation. If there is some other partial B-smooth relation with the same P already collected, they can be combined as described in 2.6 and used as one proper B-smooth relation.

For this optimization to work well, a correct choice for the data container used to store the partial relations is extremely important. The container needs to support very efficient search based on the values of P, insertion and deletion. I used the class called *unordered_map* from C++ standard library, which supports these operations with constant time complexity on average, utilizing some hashing algorithm underneath. The partial relations are stored as a key-value pair, where P is the key for each particular relation and value is the rest of the information about that relation (the left-hand side value, the right-hand side value and the prime exponents).

With the addition of partial relations, it can be beneficial to make the *sievingThreshold* less strict to allow more of the almost B-smooth numbers to "fall through the sieve", get marked and then get processed into partial relations. For this reason I added one new option `-t` for dynamic change of the threshold strictness. Supposing the value of the option `-t` is called T, the new *sievingThreshold* formula is

$$sievingThreshold = \log(M\sqrt{n}) - \log(maxFactor \cdot T),$$

3.7.3 Large prime MPQS method usage

As mentioned above, there is only one added option `-t` accepting some floating point value. The default value is 1, the higher the value the less strict the sieving threshold is, meaning more $Q_j(x)$ values get processed. The entered value cannot be smaller than the default value.

Figure 3.5: Multiple polynomial quadratic sieve usage example

```
./qsieve -m lpmpqs -b 100000 -s 100000 -t 200 \
    -p primes.txt -i inputfile.txt
```

3.8 Parallelization

Many of the modern computer clusters are using the hybrid distributed-shared memory architecture and the faculty server STAR is no exception. Thus, to properly leverage the parallelization, I had to combine two different parallelization approaches, one for shared memory architecture and one for distributed memory architecture.

To achieve that, I used the help of the openMP and MPI libraries. However, compiling projects that are using MPI is slightly more complicated process that requires installation of the library. This makes sense on a computing cluster, but not so much on a simple desktop/laptop computer. For this reason I decided to implement the parallelized version of the large prime MPQS algorithm as a separate program called `qsievempi` that is compiled separately, so that the sequential implementations can still be used without the need to install MPI.

The parallelization was done only for the sieving procedure (which is the hardest part of the problem and should theoretically take most of the time), the linear algebra stage of the program was left sequential as I felt it was not really the focus of this thesis. For potential future improvement of the program, the implementation of parallel version of this stage would be a good start.

3.8.1 Parallelization via openMP

OpenMP is a library that substantially simplifies parallelization on a system with shared memory architecture. This is achieved by separating some of the workload into different threads, which can run simultaneously on a multi-core processor. All of the threads have shared memory, which has some benefits (simple inter-thread communication), but it can also lead to a lot of problems due to race conditions.

A very general description of the library utilization is as follows. The program begins with a single thread, which initializes the workload. Then, when the parallel block is encountered, multiple threads spawn and start working separately on their own. At the end of the parallel block is a synchronization barrier, where all the finished threads wait until every single thread is done. Afterward, the program continues in a single thread, usually also consolidating the individual results of each thread.

At the very beginning of the file `LPMPQS_mpi.h`, there are three important `#define` directives used in the parallelization process. First one defines the THREADS_PER_SLAVE constant, which symbolizes the amount of threads that can run simultaneously on a particular processor. The second one is defined as POLS_PER_THREAD and symbolizes how many different polynomials should each thread sieve through before the synchronization barrier and consolidation of each thread's results occurs. And finally the

POLS_PER_SLAVE, which is defined as a simple multiplication of the previous two constants, symbolizes how many polynomials in total get sieved through by all threads before the consolidation occurs.

The actual implementation of parallelization with openMP is fairly simple, using the general description above. Instead of generating only single polynomial at the time, POLS_PER_SLAVE polynomials are generated at once. The coefficients a_j and b_j for each polynomial are stored in an instance of *PolData* class. Then, up to THREADS_PER_SLAVE threads are spawned (or reused from an already existing thread pool), each thread sieving through POLS_PER_THREAD polynomials in parallel, collecting B-smooth and partial B-smooth relations into their own separate containers. After the sieving is done for all threads, a single thread iterates through all the separate containers, moving the collected results into one global container. This repeats until there is enough B-smooth relations collected in the global container.

3.8.2 Parallelization via MPI

One of the disadvantages of the openMP parallelization approach is its scalability. Since the resulting program is designed to be run on single computer, its scalability depends on how many processors cores the computer has and adding more of them gets increasingly more difficult. MPI can address that by allowing us to split the workload not between different cores on single computer, but between different processes running on different computers connected through a network. This helps with the scalability issue, as when it becomes difficult to add more processor cores to a single computer, we can simply start using more computers instead. A slight disadvantage is that the processes do not share memory and so the inter-process communication with MPI is more difficult to achieve, done so by passing messages between processes over said network.

In my implementation, I used the master-slave model to accomplish parallelization with MPI. One of the processes is called the "master process", all the other are called the "slave processes". The master process is responsible for the generation and distribution of the workload, meaning the coefficients of different polynomials, to all slave processes. All slave processes have to do is to wait until they obtain the polynomial coefficients from master process, work on the assigned workload and sieve over the polynomials, then send the obtained B-smooth relations back to the master process. The master process collects these relations from all slaves in a single container, sending back more work until there is enough of them collected.

To be able to send polynomial coefficients and B-smooth relations over the network, the data needs to be serialized first. The coefficients, stored in *PolData* objects as *mpz_int* data types, can be easily serialized into string objects by *mpz_int* member functions. The B-smooth relations are stored in *BSmoothRelation* objects and have three member variables, the left-hand side

and right-hand side values as *mpz_int* and the prime exponents of the relation as *dynamic_bitset*. The *dynamic_bitset* class also has member functions allowing them to be turned into or constructed from a string object, but doing that would not be very efficient, because it turns every bit from the bitset into one character, either 0 or 1. That would generate a needlessly long string with as many characters as the size of the factor base for each B-smooth relation. Instead of that, I used different member function *to_block_range()* to turn the bits in the bitset into blocks of long integers, transformed them into strings and then reassembled back into *dynamic_bitset* with *from_block_range()* on the other side.

3.8.3 Combining the openMP and MPI approaches

Once both of the approaches are designed and implemented separately, their combination is fairly straightforward. The initialization stage is identical for all processes and is unchanged from how it was described in section 3.7.1. Then, during the sieving stage, the roles of the processes get separated. The master process starts by generating and sending POLS_PER_SLAVE polynomials to each slave process. Then the master process splits into two separate threads, one to handle communication with slaves and one for its own sieving computations. The communication thread waits until it receives a result from a slave, saving the newly obtained relations. If there are not enough relations collected, the thread sends POLS_PER_SLAVE new polynomials to the slave, otherwise it sends an instruction to terminate. The other thread of the master process generates its own POLS_PER_SLAVE $-$ 1 polynomials and sieves them with THREADS_PER_SLAVE$-$1 threads using openMP technique from 3.8.1. This way, all of the processor cores of the machine running the master process are fully utilized. For the second thread to be able to spawn its own threads, the openMP nested parallelism had to be explicitly enabled.

Each slave process waits until it receives a polynomial batch from master and then sieves it in parallel using openMP as was described in 3.8.1. Afterwards, it sends the B-smooth relations found back to master and awaits next message. If it receives the termination instruction, the slave process graciously quits. After all slave processes quit, the master continues with the last two stages on its own.

3.8.4 Parallel large prime MPQS method usage

Figure 3.6: Parallel large prime MPQS usage example

```
./qsievempi -b 100000 -s 100000 -t 200 -p primes.txt \
    -i inputfile.txt
```

Chapter 4

Quadratic sieve testing

The testing and measuring the performance of all methods implemented in this thesis was done on the faculty server STAR. STAR is a computer cluster composed of 8 computer nodes, each node has the following specifications:

- motherboard Supermicro Super X10DRFF with dual CPU sockets,
- CPU Intel® Xeon® CPU E5-2630 v4 @ 2.20GHz,
- Each CPU has 10 cores and supports hyper-threading,
- 64 GB of RAM.

Nevertheless, the measuring on STAR came with some unpleasant restrictions. The server is set-up in such a way that maximum of 4 nodes can be used for one parallel program execution and each execution is forcibly terminated after 10 minutes, limiting the size of the numbers I could test these factorizing algorithms on.

For the purpose of testing, many different input numbers were created and separated based on their factorization difficulty into 11 individual files, which can be found in the `input` directory. Each implemented method (with the exception of trial division) has various options that can severely change the outcome of the factorization speed, so before I could do the actual measurements, I needed to know how to properly set these options. And because the ideal values for options change based on how big the factorized number is, I needed to know proper options for each method and for each of the different input files. For this reason I split the input files into two categories: training and testing. The training input files were used to estimate the ideal option values and the testing were used in the actual performance measurement. Each version (training/testing) of every file contains 5 composite numbers with exactly the same number of digits and approximately the same bit length. Each composite was created by multiplying two random prime numbers of approximately half of the composite's bit length together. The list of files with their

Table 4.1: Input files description

input file name	approx. bit length	number of digits
`40bit.in`	40	12
`60bit.in`	60	18
`70bit.in`	70	21
`80bit.in`	80	24
`100bit.in`	100	30
`120bit.in`	120	36

input file name	approx. bit length	number of digits
`140bit.in`	140	42
`160bit.in`	160	48
`180bit.in`	180	54
`200bit.in`	200	60
`220bit.in`	220	66

composite numbers' respective approximate bit length and number of digits can be found in table 4.1. The higher the bit length, the more difficult file it is for factorization.

The time measurement was done using the *steady_clock* class from the *chrono* library, which is part of the C++ standard. I chose this class because it represent a "monotonic clock", meaning it is guaranteed its value constantly moves forward and never decreases (even when the system time is changed during the run of the program), and as such it is ideal for time interval measurements. Each individual number from the input file was measured separately, starting the measurement right after it was loaded into memory and ending right after two of its factors were found (but before they were printed out).

The testing methodology I used was as follows: For each implemented method I selected an appropriate set of input files from 4.1 to test the method on. The more advanced method, the more difficult input files were chosen. Afterward, I ran the method with the training version of the selected input files multiple times, each time with different option values. After each run was finished and all 5 composite numbers were factorized, I calculated the average time it took to factorize one number of the approximate bit length with given options by taking the arithmetic mean of the 5 time measurements. All of the training measurements can be found in the appendix B. Lastly, I ran the method one last time, this time with the testing version of the input files and the option values that gave the best performance during training, once again taking the average of the 5 time measurements for every file.

4.1 Environment set-up and compilation

Before the two programs can be run, they need to be compiled in a properly prepared environment. In order to compile the `qsieve` program, a C++ compiler that supports C++11 is needed, as well as the GMP and Boost libraries. The `qsievempi` additionally requires openMP and MPI libraries present. The

`make` program with the enclosed `Makefile` file can be used to simplify the compilation process, but before it can be used, the `Makefile` requires some minor changes described below. This section focuses on compilation for both of the programs on the STAR server.

The Boost collection of libraries is already present on the STAR server, but unfortunately in the form of an older version 1.53.0, which is too old and does not support some of the features that I used for the implementation. Newer version of the Boost libraries is on the enclosed CD, it just needs to be copied over to STAR and extracted. In the `Makefile`, the appropriate line should be changed to `CFLAGS += -I/path/to/extracted/boost`. The Boost libraries used are header-only libraries, so no further action is needed.

Additionally to Boost, the GMP library of version 6.2.0 is on the enclosed CD and needs to be copied over and extracted as well. But, unlike the Boost libraries, it also needs to be built and installed. This is done by changing into the directory with extracted GMP and running the following:

```
./configure --prefix=${HOME}/gmp
make
make install
```

This will install the GMP library to the `gmp` directory in the home directory of the user. Afterward, the `Makefile` needs to be changed one last time, this time on the appropriate lines `#CFLAGS += -I/path/to/gmp/include` and `#GMPLIB += -L/path/to/gmp/lib`.

If the program is being compiled on desktop/laptop, the GMP and Boost libraries can be installed as a root user and no change in the `Makefile` should be necessary. After all of this is done, everything is setup correctly and the programs can be compiled. To compile `qsieve`, `make` has to be run in the directory with the `Makefile`. In order to compile `qsievempi`, `make mpi` has to be run instead. To clean the working directory from generated object files and binaries, the `make clean` can be run. The compilation is done with the `-O3` and `-ffast-math` options for maximal optimization and efficiency.

4.2 Testing the trial division method

Trial division implementation is the only one with no additional options, thus it was the only implementation that could be tested straight away with no training required. The testing was done on the files `40bit.in`, `60bit.in` and `70bit.in`, the average time it took to factorize one number for each input file can be seen in table 4.2.

The implementation of trial division method was fairly basic and, as can be seen in table 4.2, it struggled even with relatively easy 60 bit composite numbers, which are just 18 digits long. As for the `70bit.in` input file, trial division could not find the factors of even single composite number before the

Table 4.2: Trial division testing results

input file name	average time (s)
`40bit.in`	0.066
`60bit.in`	32.598
`70bit.in`	DNF

10 minute deadline of the STAR server was imposed, forcefully terminating the computation.

4.3 Testing the Dixon's method

I tested my implementation of the Dixon's method with the files `60bit.in`, `70bit.in`, `80bit.in` and `100bit.in`. The only option available for this implementation is the `-b` option for the value of smoothness bound variable B. For the training of this option, I used the following values for the training version of every file: 2500, 5000, 10000, 15000, 25000 and 50000. The final results using the best performing option on the testing version of every file can be found in the table 4.3.

Table 4.3: Dixon's method testing results

input file name	option `-b`	average time (s)
`60bit.in`	2500	1.120
`70bit.in`	5000	9.245
`80bit.in`	10000	37.373
`100bit.in`	—	DNF

The Dixon's method offers marginal improvement over the trial division, being able to find factors of composite numbers up to 80 bits long in time. Nevertheless, during training it failed to factorize single composite number from the file `100bit.in` in less than 10 minutes, before it was interrupted by STAR, no matter what option `-b` was selected.

4.4 Testing the quadratic sieve method

The quadratic sieve method was more complicated to train, because it requires two options: `-b` for B variable and `-s` for the size of the sieving interval. The testing was done on the input files `80bit.in`, `100bit.in`, `120bit.in`, `140bit.in`, `160bit.in` and `180bit.in`. To get a good performance out of this method for each file, I had to run their training versions with multiple different options `-b` and `-s`. The option values tried for each training file and

their results can be found in the appendix B. The final measurements of the testing input files with the best performing options are in the table 4.4.

Table 4.4: Quadratic sieve testing results

input file name	option `-b`	option `-s`	average time (s)
`80bit.in`	5000	10000	0.031
`100bit.in`	10000	10000	0.112
`120bit.in`	30000	50000	0.507
`140bit.in`	75000	100000	2.445
`160bit.in`	150000	100000	16.996
`180bit.in`	300000	250000	78.335

Comparing the tables 4.4 and 4.3, it is obvious that all of the improvements implemented in the quadratic sieve method offer a substantial increase of factorization efficiency over the Dixon's method, allowing me to more than double the size of the composite numbers before hitting the STAR server deadline of 10 minutes.

4.5 Testing the multiple polynomial quadratic sieve method

The MPQS method requires the same option as the quadratic sieve method, but with one small difference. The option `-s` is not the size of the sieving interval per se, instead it is the value of the M variable (which is about a half of the size of the sieving interval). Because MPQS offers some major changes to the sieving stage, I could not use the same option values that performed the best for the quadratic sieve method found in previous section, instead I had to find the best performing options from scratch again. The files `140bit.in`, `160bit.in`, `180bit.in` and `200bit.in` were tested with the results in the table 4.5.

Table 4.5: Multiple polynomial quadratic sieve testing results

input file name	option `-b`	option `-s`	average time (s)
`140bit.in`	35000	1000000	0.853
`160bit.in`	75000	1000000	5.046
`180bit.in`	150000	1500000	21.656
`200bit.in`	250000	1500000	93.147

4.6 Testing the large prime MPQS method

Unlike the MPQS from previous section, the large prime optimization does not include any substantial changes to the sieving process, it only improves its speed. Because of that, I chose to test the files `140bit.in`, `160bit.in`, `180bit.in` and `200bit.in` with the same best performing options `-b` and `-s` from table 4.5. But there is one more option required, the `-t` option, symbolizing the "looseness" of the sieving threshold. Therefore, I used the training version of the input files with the options options `-b` and `-s` from table 4.5 and many different values for option `-t`. The testing results, as well as the best options found, can be seen in table 4.6.

Table 4.6: large prime MPQS testing results

input file name	option `-b`	option `-s`	option `-t`	average time (s)
`140bit.in`	35000	1000000	100	0.691
`160bit.in`	75000	1000000	80	3.973
`180bit.in`	150000	1500000	200	17.182
`200bit.in`	250000	1500000	1000	75.443

4.7 Testing the parallelized large prime MPQS method

The testing of the parallelized large prime MPQS method was done on the input files `140bit.in`, `160bit.in`, `180bit.in`, `200bit.in` and `220bit.in`. For the first four files, I used the already found options that performed best in the sequential implementation. However, for the `220bit.in`, I had to use the training input file to find the best option for `-b`, `-s` and `-t` options from scratch again. The results and the best options found are in the table 4.7.

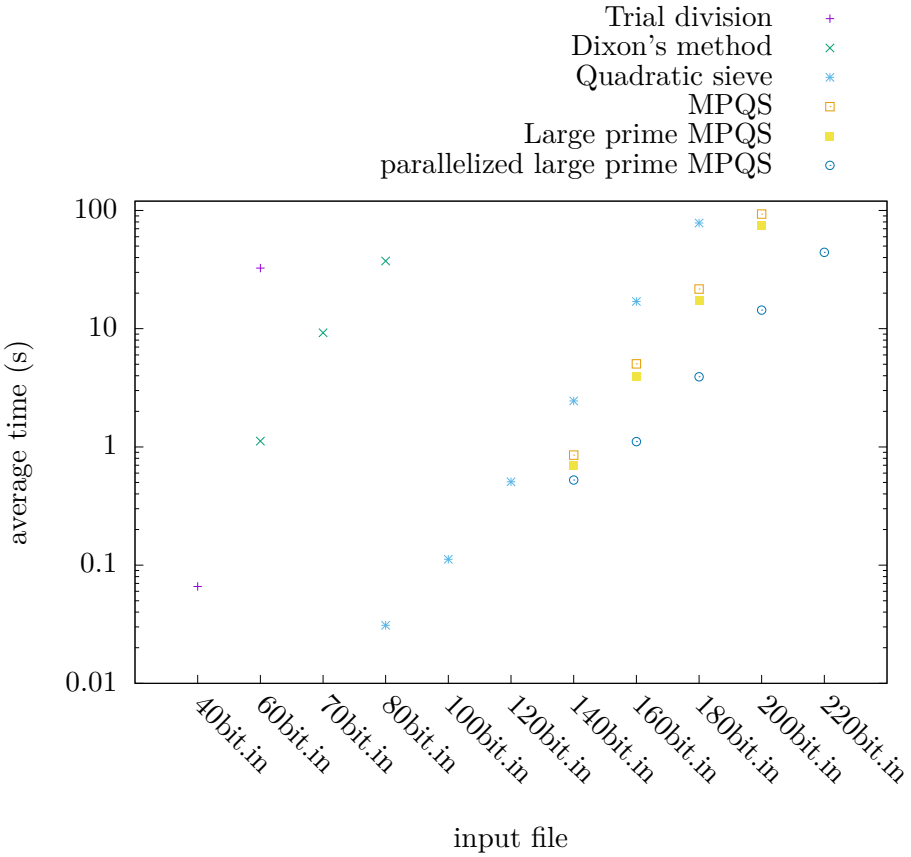
Table 4.7: Parallelized large prime MPQS testing results

input file name	option `-b`	option `-s`	option `-t`	average time (s)
`140bit.in`	35000	1000000	100	0.524
`160bit.in`	75000	1000000	80	1.108
`180bit.in`	150000	1500000	200	3.919
`200bit.in`	250000	1500000	1000	14.367
`220bit.in`	250000	1500000	30000	44.286

4.8 Comparing the results

The comparison of the average time results for all methods from the tables above can be seen in the figure 4.1 below. The y-axis is representing the average time spent to find factors on one composite number from the testing version of a given input file from the x-axis. The scale of the y-axis is logarithmic to better fit all of the results.

Figure 4.1: Average time to factorize one composite number from input files



While the trial division has fully exponential time complexity, all of the other algorithms implemented in this thesis are sub-exponential. This can be seen in the figure 4.1, as the results for trial division are increasing much more steeply than the others.

The asymptotic time complexity difference between the Dixon's method (described in section 2.2.3) and the quadratic sieve (section 2.3.5) is in the fairly large decrease for the constant component of the exponent from $3\sqrt{2}$ to 1. This is also reflected in the figure 4.1 in the huge increase in efficiency between Dixon's method and quadratic sieve.

In the section 2.3.5 it is also mentioned that the time complexity of the

quadratic sieve and all of its variations is asymptotically the same. This is apparent from the figure 4.1, where the increase in efficiency between quadratic sieve, MPQS and large prime MPQS is fairly small, asymptotically insignificant and much smaller than the above-mentioned increase between Dixon's method and quadratic sieve.

The increase in efficiency of the parallelized large prime MPQS method is somewhat dampened by the time costs of node communication overhead and the usage of synchronization primitives when accessing variables shared between multiple threads.

One more observation can be made in the table 4.7, where the best performing option `-b` for the files `200bit.in` and `220bit.in` is the same. I suspect the cause for this is the linear stage implemented sequentially via GEM. When factorizing relatively small composite numbers, the values for variable B can be fairly low and therefore the time spent in the linear algebra stage is insignificant compared to the sieving stage. But as the size of the composite number increases, so does the optimal value for B and thus the exponent matrix grows as well. As a result, the algorithm spends a substantial amount of time in the linear algebra stage for some of the more difficult input files. Thus, when factorizing numbers from the file `220bit.in` with smaller `-b`, the algorithm prefers spending more time in the parallelized sieving process and less time in the sequential GEM than the other way around. This could be rectified by implementing more sophisticated linear algebra stage with parallelization as mentioned in section 3.4.3. The average time spent in the linear algebra stage for one composite number from `220bit.in` input file with different values for option `-b` as well as the approximate size of the factor base (which implies the approximate size of the exponent matrix) can be found in the table 4.8 below.

Table 4.8: Time spent in the linear algebra stage for the input file `220bit.in`

option `-b`	approx. size of FB	average time (s)
200000	9000	3.5
250000	11000	7.5
300000	13000	14
350000	15000	22

When I tested the factorization of the most difficult input file used in this thesis, the `220bit.in` file, using one of the state-of-the-art programs called `msieve` (introduced in the section 2.7) on STAR, it was able to do so in the average time of just 15 seconds. Msieve is one of the most efficient implementations of quadratic sieve methods (based on the program's own documentation found in [10]), using the SIQS variation with the 2-large prime optimization and many other small optimizations to make the code as fast as possible, therefore it was to be expected it would perform the factorization task much faster than what my implementation was able to achieve.

Conclusion

In this master's thesis, I analyzed two factorization algorithms, the Dixon's method and the quadratic sieve. I also analyzed its modification known as the multiple polynomial quadratic sieve and the large prime optimization. I went over some of the existing open source implementations of quadratic sieve factorization. I implemented all of the methods above in the C++ programming language using the GNU Multiple Precision Arithmetic Library. I discussed and implemented parallelization using the openMP and MPI libraries, tested all implemented method on the faculty server STAR and compared all of the results.

The first chapter describes some of the basic terms, definitions and algorithms used throughout the rest of this thesis.

The second chapter focuses on the theoretical principles of the quadratic sieve factorization method. It goes over its predecessors, the Fermat's integer factorization and the Dixon's factorization method. Then the quadratic sieve itself is described in great detail, followed by the multiple polynomial quadratic sieve method. All of the above-mentioned methods are provided with algorithm description in pseudo-code. The main principle behind the self-initializing quadratic sieve variation is mentioned, followed by the explanation of the large prime optimization. The chapter is finished off with the rundown of some other open source implementations.

The third chapter deals with the implementation details. It goes over the general implementation idea as well as the technologies used. Then it explains the implementation of each method, providing an usage example at the end. The chapter ends with the description of the parallelization using openMP and MPI.

In the fourth and the last chapter, all of the implemented methods are tested on the faculty server STAR. The environment set-up and the testing methodology are described, then the results of testing each method follow. The chapter is closed off with the comparison of the results, as well as with some general observations encountered during testing.

Bibliography

[1] CRANDALL, Richard; POMERANCE, Carl. *Prime numbers: a computational persprective*. 2nd ed. Springer, 2005. ISBN 978-0387-25282-7.

[2] SHOUP, Victor. *A Computational introduction to number theory and algebra*. 2nd ed. Cambridge University Press, 2008.

[3] POMERANCE, Carl. A tale of two sieves. *Notice of the American Mathematical Society* [online]. 1996, vol. 43, pp. 1473–1485 [visited on 2021-04-03]. Available from: `https://www.ams.org/notices/199612/pomerance.pdf`.

[4] WILLIAMS, Hugh Cowie; SHALLIT, Jeffrey. Factoring integers before computers. *Mathematics of Computation 1943–1993: A Half-Century of Computational Mathematics*. 1994, vol. 48, pp. 481–531. ISBN 978-0-8218-0291-5.

[5] DIXON, John D. Asymptotically fast factorization of integers. *Mathematics of Computation*. 1981, vol. 36, no. 153, pp. 255–255. Available from DOI: `10.1090/s0025-5718-1981-0595059-1`.

[6] POMERANCE, Carl. The Quadratic Sieve Factoring Algorithm. In: *Advances in Cryptology: Proceedings of EUROCRYPT 84*. Springer-Verlag, 1985, pp. 169–182.

[7] KECHLIBAR, Marian. *The Quadratic Sieve - introduction to theory with regard to implementation issues* [online]. 2005 [visited on 2021-04-03]. Available from: `www.karlin.mff.cuni.cz/~krypto/Implementace_MPQS_SIQS_files/main_file.pdf`.

[8] POMERANCE, Carl. Smooth numbers and the quadratic sieve. *Algorithmic Number Theory*. 2008, vol. 44, pp. 69–81.

[9] SILVERMAN, Robert D. The multiple polynomial quadratic sieve. *Mathematics of Computation*. 1987, vol. 48, no. 177, pp. 329–329. Available from DOI: `10.1090/s0025-5718-1987-0866119-8`.

[10] PAPADOPOULOS, Jason. *Msieve* [online]. 2021 [visited on 2021-04-16]. Available from: `https://sourceforge.net/projects/msieve/`.

[11] *Yet Another Factorization Utility* [online]. 2020 [visited on 2021-04-16]. Available from: `https://sites.google.com/site/bbuhrow/home`.

[12] KOVÁČ, Peter. *Optimalizovaná faktorizace velkých čísel pomocí knihovny GMP*. 2009. MA thesis. Czech Technical University in Prague.

[13] *GNU Multiple Precision Arithmetics Library* [online]. 2020 [visited on 2021-04-17]. Available from: `https://gmplib.org/`.

[14] *Boost C++ Libraries* [online]. 2021 [visited on 2021-04-17]. Available from: `https://gmplib.org/`.

[15] *OpenMP* [online]. [N.d.] [visited on 2021-04-17]. Available from: `https://www.openmp.org/`.

[16] *Message Passing Interface* [online]. [N.d.] [visited on 2021-04-17]. Available from: `https://www.mpi-forum.org/`.

[17] *Project Wycheproof* [online]. 2019 [visited on 2021-04-17]. Available from: `https://github.com/google/wycheproof`.

[18] BÜTTCHER, Stefan. Factorization of Large Integers. *Ferienakademie 2001: Cryptography and Security of Open Systems* [online]. 2001 [visited on 2021-04-03]. Available from: `http://stefan.buettcher.org/cs/factorization/essay.pdf`.

Appendix A

Acronyms

CPU Central processing unit

DNF Did not finish

FB Factor base

GCD Greatest common divisor

GEM Gaussian elimination method

GMP GNU multiple precision

MPI Message parsing interface

MPQS Multiple polynomial quadratic sieve

QS Quadratic sieve

SIQS Self initializing quadratic sieve

RAM Random access memory

Appendix B

Training measurements

All the tables in this section contain the average time spent factorizing one composite number using implemented methods with different input files and various option values. The training versions of input files were used, the time is measured in seconds.

Table B.1: Training Dixon's method with option `-b`

	option `-b`					
input file name	2500	5000	10000	15000	25000	50000
`60bit.in`	**1.524**	1.827	3.030	4.559	7.920	18.263
`70bit.in`	11.280	**10.041**	12.715	16.913	26.221	65.532
`80bit.in`	213.330	82.116	**72.698**	80.992	107.515	207.350
`100bit.in`	DNF	DNF	DNF	DNF	DNF	DNF

Table B.2: Training quadratic sieve with options `-b` and `-s` for the input file `80bit.in`

	option `-s`		
option `-b`	10000	50000	100000
5000	**0.029**	0.065	0.111
10000	0.071	0.158	0.259
15000	0.125	0.271	0.441
20000	0.177	0.386	0.650

Table B.3: Training quadratic sieve with options `-b` and `-s` for the input file `100bit.in`

	option `-s`		
option `-b`	10000	50000	100000
10000	**0.147**	0.175	0.198
20000	0.170	0.218	0.278
25000	0.202	0.271	0.356
30000	0.251	0.340	0.467

Table B.4: Training quadratic sieve with options `-b` and `-s` for the input file `120bit.in`

	option `-s`		
option `-b`	50000	100000	200000
10000	2.179	2.372	2.463
20000	0.756	0.805	0.867
30000	**0.633**	0.693	0.756
40000	0.736	0.789	0.900

Table B.5: Training quadratic sieve with options `-b` and `-s` for the input file `140bit.in`

	option `-s`		
option `-b`	100000	250000	500000
25000	8.902	9.376	9.558
50000	3.580	3.785	3.913
75000	**3.290**	3.483	3.720
100000	3.859	4.129	4.487

Table B.6: Training quadratic sieve with options `-b` and `-s` for the input file `160bit.in`

	option `-s`		
option `-b`	100000	250000	500000
75000	24.856	25.942	26.434
100000	18.214	18.938	19.276
125000	15.750	16.224	16.568
150000	**15.197**	15.598	15.925

Table B.7: Training quadratic sieve with options `-b` and `-s` for the input file `180bit.in`

	option `-s`		
option `-b`	100000	250000	500000
150000	114.575	117.444	118.855
200000	85.950	86.493	87.086
250000	74.841	73.901	74.538
300000	71.976	**70.252**	70.694

Table B.8: Training MPQS with options `-b` and `-s` for the input file `140bit.in`

	option `-s`		
option `-b`	500000	1000000	1500000
35000	1.135	**1.095**	1.149
50000	1.345	1.263	1.280
75000	1.962	1.835	1.841

Table B.9: Training MPQS with options `-b` and `-s` for the input file `160bit.in`

	option `-s`		
option `-b`	500000	1000000	1500000
50000	4.982	4.666	4.790
75000	5.216	**4.629**	4.650
100000	6.039	5.250	5.154

Table B.10: Training MPQS with options `-b` and `-s` for the input file `180bit.in`

	option `-s`		
option `-b`	1000000	1500000	2000000
150000	20.474	**19.223**	22.458
200000	22.754	21.614	23.511
250000	27.227	25.069	26.950
300000	33.201	30.970	32.226

Table B.11: Training MPQS with options `-b` and `-s` for the input file `200bit.in`

	option `-s`		
option `-b`	1000000	1500000	2000000
150000	96.762	96.144	109.807
200000	91.765	85.638	96.529
250000	92.909	**82.596**	92.865
300000	97.855	86.977	94.062

Table B.12: Training large prime MPQS with option `-t` and the best performing options `-b` 35000 and `-s` 1000000 for the input file `140bit.in`

option `-t`	average time (s)
1	1.032
10	0.860
20	0.840
40	0.831
80	0.832
100	**0.828**
120	0.830
150	0.839

Table B.13: Training large prime MPQS with option `-t` and the best performing options `-b` 75000 and `-s` 1000000 for the input file `160bit.in`

option `-t`	average time (s)
10	3.885
20	3.761
40	3.726
80	**3.699**
150	3.716
200	3.738
400	3.834
600	3.921

Table B.14: Training large prime MPQS with option `-t` and the best performing options `-b` 150000 and `-s` 1500000 for the input file `180bit.in`

option `-t`	average time (s)
10	16.328
50	15.668
100	15.438
150	15.677
200	**15.644**
400	15.970
600	16.108
800	16.476

Table B.15: Training large prime MPQS with option `-t` and the best performing options `-b` 250000 and `-s` 1500000 for the input file `200bit.in`

option `-t`	average time (s)
100	70.289
500	69.557
1000	**69.347**
2000	70.440
4000	70.446
6000	71.366
8000	71.785
10000	73.455

Table B.16: Training parallelized large prime MPQS with options `-b`, `-s` and constant option `-t` 10000 for the input file `220bit.in`

	option `-s`		
option `-b`	1000000	1500000	2000000
200000	48.766	45.411	51.565
250000	46.274	44.406	48.507
300000	48.872	47.636	50.603
350000	54.175	54.201	55.559

Table B.17: Training parallelized large prime MPQS with options `-b`, `-s` and constant option `-t` 20000 for the input file `220bit.in`

	option `-s`		
option `-b`	1000000	1500000	2000000
200000	48.202	44.677	50.641
250000	45.805	43.772	47.931
300000	48.982	47.694	50.179
350000	54.608	54.562	55.856

Table B.18: Training parallelized large prime MPQS with options `-b`, `-s` and constant option `-t` 30000 for the input file `220bit.in`

	option `-s`		
option `-b`	1000000	1500000	2000000
200000	47.278	44.185	49.910
250000	45.562	**43.599**	47.841
300000	48.660	47.557	50.214
350000	55.169	54.687	56.386

Appendix C

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `libraries` ..... the GMP and Boost libraries used for implementation
- `qsieve` ..... the directory containing the `qsieve` program
  - `Makefile` ..... `Makefile` to easily compile the project with `make`
  - `readme.txt` ..... the file with `qsieve` program description
  - `input` ..... the directory containing training and testing input files
  - `primes` ..... the directory containing text files with primes
  - `src` ..... the directory containing source files
- `thesis` ..... the thesis text directory
  - `masters_thesis.pdf` ..... the thesis text in PDF format
  - `src` ..... the thesis source files in LaTeX format