Interplay of Number Theory and Fixed-Parameter Tractable Algorithms

Bachelor's Thesis Computing Science

University of Groningen

Author: Arnaud Van Hees
First Supervisor: Ivan Bliznets
Second Supervisor: Revantha Ramanayake

July 2025

Abstract

This thesis investigates the fixed-parameter tractability of NP-hard number-theoretic problems. While parameterized complexity has been studied extensively in the field of graph theory, number theory has been comparatively underexplored. Motivated by bridging this gap, we study five NP-hard problems from Garey and Johnson's 1979 book on *"Computer and Intractability"*, each of which are related to core number-theoretic topics such as modular arithmetic, Diophantine equations, divisibility, and multiplication.

The central research question guiding this thesis is *"What parameters are suitable for parameterization of NP-hard problems in number theory?"* To answer this question, we explore a variety of parameterizations for each problem. These include bounds on solution variables, input sizes, and number-theoretic properties, such as the number of distinct prime factors appearing in parts of the input.

Our contributions include nine fixed-parameter tractable (FPT) algorithms and one slice-wise polynomial (XP) algorithm. These algorithms leverage a variety of algorithmic techniques, including integer linear programming (ILP) with bounded constraints, dynamic programming over exponent vectors, brute-forcing over bounded search spaces, and classical number-theoretic techniques such as the Chinese Remainder Theorem (CRT) and Hensel's Lemma. We show that different parameterizations can lead to fundamentally different algorithms, even for problems under the same parameterization. Furthermore, we show that there is no single parameter that is optimal for each problem, and different parameterizations lend themselves better to different input instances.

Overall, this thesis provides strong evidence for the relevance of parameterized complexity in number theory. Our work contributes both a concrete toolkit of algorithmic techniques and outlines a framework for identifying other promising parameters.

Contents

1 Introduction & Motivation

Computational complexity theory traditionally classifies problems based on their worst-case behavior with respect to input size, distinguishing between tractable, polynomial-time solvable, and intractable, NP-hard, problems. However, this distinction often overlooks important structural, among other properties, that are present in the real-world instances of these problems. Many problems that are hard in general can be solved efficiently when certain parameters in the input are small. This insight forms the basis for parameterized complexity theory. A framework that analyzes computational problems using both input size and additional parameters [9].

Similarly to *polynomial time* being central to the classical formulation of computational complexity, central to parameterized complexity is the notion of *fixed-parameter tractability (FPT)*. Informally a problem is fixed-parameter tractable with respect to a parameter k if it can be solved in time $f(k) \cdot \text{poly}(n)$ where f is a computable function independent of the input size n. This makes FPT algorithms particularly useful when the parameter k is small in practice, even if the input n is large.

Formally, we define these key concepts as follows [11, 9, 12]:

Definition 1. *A* ***parameterized problem*** *is a language $L \subseteq \Sigma^* \times \mathbb{N}$, where Σ is a fixed, finite alphabet. For an instance $(x, k) \in \Sigma^* \times \mathbb{N}$, k is called the* ***parameter****.*

Definition 2. *A parameterized problem $L \subseteq \Sigma^* \times \mathbb{N}$ is called* ***fixed-parameter tractable*** *(FPT) if there exists an algorithm $\mathcal{A}$ (called a* ***fixed-parameter algorithm****), a computable function $f : \mathbb{N} \to \mathbb{N}$, and a constant c such that, given $(x, k) \in \Sigma^* \times \mathbb{N}$, the algorithm $\mathcal{A}$ correctly decides whether $(x, k) \in L$ in time bounded by $f(k) \cdot |(x, k)|^c$. The complexity class containing all fixed-parameter tractable problems is called FPT.*

With these definitions in mind, parameterized complexity as a field aims to address intractability questions for problems that are parameterized, and for which important applications are covered by small ranges of parameters, that classical complexity frameworks are not adequate to address [11].

As explained in more detail in Section 2, graph theory is a domain that has been extensively studied under the parameterized complexity and fixed-parameter tractability frameworks. In this thesis, we direct our attention to number theory. A domain that despite its importance, has received comparatively little attention from parameterized complexity and FPT approaches in literature.

Number theory has numerous practical applications across different fields, including integer optimization computational number theory, and most notably, cryptography [19]. Problems in number theory, particularly those involving divisibility, modular arithmetic and Diophantine equations, are often NP-hard [16]. As such, these lend themselves exceptionally well to cryptographic systems, where their computational hardness is central to the security of the system. One such example is public-key encryption and the widely used RSA protocol [10, pp. 15–18].

However, in practice, the instances of these problems may involve small or bounded quantities, such as a limited number of prime factors, upper bounds on solution variables, or structural properties. This raises an important question of whether these problems, though hard in general, can be efficiently solved when restricted to small parameter values?

This question inspires both the theoretical and practical contributions of this thesis. From a theoretical perspective, extending FPT techniques into number theory helps broaden the scope of parameterized complexity and deepen our understanding of which parameters impact number-theoretic problem's computational hardness. As mentioned by Misra et al. the potential of parameterized complexity in fields other than graph theory remains relatively underexplored [24]. This thesis aims to bridge that gap. From a practical standpoint, finding efficient FPT algorithms for number-theoretic problems could help cryptographers identify those parameter ranges that leave their systems vulnerable to FPT algorithms using these parameters as a basis for attacks. Conversely, these techniques could also be used in optimizing number-theoretic computations in areas like coding theory and integer optimization [19].

In this thesis, the focus is to identify natural parameterizations that make the chosen NP-hard number-theoretic problems tractable under these parameterizations. To this end, we analyze a variety of classical number-theoretic decision problems. The problems we have chosen are from the foundational text on the NP-hardness of decision problems written by Garey and Johnson in 1979 [16]. For each of these problems, we explore which parameters enable fixed-parameter tractable algorithms.

This leads us to the central research question of the thesis:

What parameters are suitable for parameterization of NP-hard problems in number theory?

To answer this question, each section of the thesis focuses on one number-theoretic problem. We formally define the problem, propose natural parameterizations, and construct FPT algorithms under those parameterizations. For each problem and parameterization, we aim to convey the underlying intuition before presenting the algorithm in full detail, followed by a formal proof of correctness and an analysis of its running time within the FPT framework.

Through this approach, we aim not only to provide concrete algorithmic results but also to offer an insight into what parameterizations are promising for the studying of parameterized complexity in the field of number theory. Furthermore, we hope this thesis serves as a foundation and reference point for future work, equipping other researchers with insights and techniques they can use when analyzing other problems in this area.

2 Related Work

2.1 Fixed-Parameter Tractability in Graph Problems

The field of parameterized complexity has seen extensive research in graph-related problems. Various well known NP-complete decision problems, such as VERTEX COVER, SATISFACTORY PARTITION, FEEDBACK VERTEX SET, and a variety of others have been proven to be fixed-parameter tractable with appropriate parametrization. Downey and Fellow were the first to show that the VERTEX COVER problem was FPT when parameterized by the size of the solution [15]. They also pioneered the research in the field of parameterized complexity. Many of the techniques they proposed in their initial papers, summarized in their *Fundamentals of Parameterized Complexity* book [12], are core to the research done nowadays. Using Downey and Fellow's techniques, together with other optimizations, Chen et al. shared the fastest known FPT algorithm for the VERTEX COVER problem in 2006, which runs in $\mathcal{O}^*(1.2738^k)$ time [5, 14].

2.2 Fixed-Parameter Tractability in Number Theory & Beyond

Contrary to graph problems, the application of FPT techniques in number theoretic problems is less developed. However, a motivating example that problems in number theory also lend themselves to FPT algorithms is the COVERING BY ARITHMETIC PROGRESSION (CAP) problem. CAP was proved to be NP-complete by Health in 1990 [18]. Recently, in 2024, Bliznets et al. presented a $2^{\mathcal{O}(k^2)}poly(n)$ time FPT algorithm to solve CAP. The CAP problem and its solutions have practical applications such as in VLSI chip manufacturing [4].

Randolph and Wegrzycki [27] recently studied the well known number theoretic problems of SUBSET SUM and K-SUM. The paper investigates FPT approaches to these problems and finds that they admit an FPT algorithm when the input set exhibits a small doubling property.

Randolph later expanded on this line of work in his thesis, where he presented a broader range of exact and parameterized algorithms for SUBSET SUM and related problems. Notable contributions include average case algorithms for GENERALIZED SUBSET SUM, algorithms that solve worst-case SUBSET SUM faster than $2^{0.5n}$ by a polynomial factor, and lastly insights into how SUBSET SUM is fixed-parameter tractable in the doubling constant depending on the time required to solve a type of Integer Linear Program [26].

The work by Misra et al. provides a State-of-the-Art survey of parameterized complexity across various domains. One of its key conclusions is the need to broaden the scope and impact of parameterized complexity, beyond its traditional focus on graph problems [24]. Exploring under-studied areas, such as number theory, offers a promising direction for extending the reach and applicability of FPT algorithms.

2.3 Fixed-Parameter Tractability Considerations for Selected Number-Theoretic Problems

In this thesis, we explore several classical NP-hard number-theoretic problems from the perspective of parameterized complexity. These problems, systematically categorized by Garey and Johnson [16], have, to the best of our knowledge, not yet been thoroughly examined within the framework of fixed-parameter tractability. We summarize below the existing literature on these problems and highlight the gaps we aim to address. Later on, in each problem's respective section, we state the formal problem definition.

2.3.1 Subset Product (SP14)[1]

The SUBSET PRODUCT PROBLEM (SPP) is analogous to the SUBSET SUM PROBLEM, with a slight variation. Instead of computing a target sum, the goal is to compute a target product. More specifically, given a set A of positive integers and a target integer B, the SPP asks whether there exists a subset $A' \subseteq A$ such that the product of its elements equals B [16, p. 224].

[1]Each code in parentheses (SP14, AN1,AN2,etc...) corresponds to the problem numbering in Garey and Johnson's appendix [16], indicating these are standard NP-complete problems from their book. We use these codes throughout for consistency and ease of cross referencing with their book.

Despite the extensive amount of research done on the closely related Subset Sum Problem, both from a classical and parameterized complexity perspecitve, the SPP has received comparitively little attention from either domain [13].

Nevertheless, a pseudo-polynomial time algorithm for the problem exists. The algorithm is derived by extending Richard Bellman's classic dynamic programming method designed for the Subset Sum Problem [3]. Dutta and Rajasree [13] describe the modifications to Bellman's algorithm more precisely. They use the prime factorization of B to define a dynamic programming table over B's exponent vectors and show that the time complexity of this algorithm is $\mathcal{O}(nB^{\mathcal{O}(1)})$.

Our approach to the SPP is largely inspired by this exponent-vector formulation presented by Dutta and Rajasree. However, our results differ in that we analyze the algorithm through the lens of parameterized complexity as opposed to classical computational complexity. In particular, we consider the SPP when parameterized by k, the number of distinct prime factors of the target B. To the best of our knowledge, no prior work has analyzed the SPP under this parameterization, nor studied its fixed-parameter tractability.

2.3.2 Quadratic Congruences (AN1)

The Quadratic Congruence Problem (QCP) involves determining whether a solution exist to $x^2 \equiv a \pmod{b}$ under the additional constraint that $x < c$ [16, p. 249]. Problems of this kind appear frequently in number theory, and closely related concepts such as quadratic residuosity, the Legendre symbol, and the Jacobi symbol are standard topics in number theory textbooks [6, 10, 8]. As discussed further in Section 4, these topics are not only of theoretical importance but also have many applications, including in primality testing, public key cryptography, and others [8].

Despite the the central role that these related topics play in number theory and their variety of practical applications, to the best of our knowledge, the QCP has not yet been studied from the perspective of parameterized complexity. In this thesis, we investigate whether natural parameterizations, such as bounding the number of distinct prime factors of b, or limiting the solution size of x, lead to fixed-parameter tractable algorithms for the QCP.

2.3.3 Simultaneous Incongruence (AN2)

The Simultaneous Incongruence Problem (SIP) is an NP-complete problem in modular arithmetic that seeks an integer x *violating* a system of congruences. While the closely related problem, of finding an integer x *satisfying* a system of congruences, solvable by the Chinese Remainder Theorem (CRT), has widespread applications in cryptography and other fields, the incongruence variant presents new computational challenges [10].

From a complexity perspective, the SIP shows how slight variations in problem definitions can drastically change the problem's tractability. Where the congruence problem admits polynomial-time solutions using the CRT, the incongruence variant remains NP-complete even if the same input were used [6, 16]. To the best of our knowledge, the SIP has not been studied in parameterized complexity literature. This gap motivates our work on exploring whether certain parameterizations lead to FPT algorithms for the problem. In particular we look at parameterizing by the number of input constraints.

2.3.4 Quadratic Diophantine Equations (AN8)

Diophantine equations, which require integer solutions to polynomial equations, are a classical topic in number theory with applications ranging from algebraic geometry to cryptography [29, 6]. The field of Diophantine equations is broad, and even within this class of equations different forms can have very different computational behavior. For example, linear Diophantine equations can be solved efficiently using standard number-theoretic techniques, like Euclid's algorithm, while more complex forms, such as quadratic or higher-degree equations are often NP-hard or even undecidable, by the Matiyasevich (MRDP) theorem [23, 29].

Quadratic Diophantine equations in particular have been studied from many mathematical perspectives, such as in the context of Pell's equation, $x^2 - ny^2 = 1$, or integer points on elliptic curves [29]. However, these problems have not been extensively researched from a parameterized complexity point of view. One recent exception is a result by Mandel and Ushakov, who showed that the general Diophantine Satisfiability problem is slice-wise polynomial (XP) when parameterized by the number of variables involved in the system

[22]. This result, though promising, does not directly address the individual problem like the QDEP we analyze.

To the best of our knowledge, no prior work has considered fixed-parameter tractability for the specific case of quadratic Diophantine equations involving both a quadratic term and a linear term, such as the form of our equation $ax^2 + by = c$. In this thesis, we analyze the QDEP under two different parameterizations and present FPT algorithms for both.

2.3.5 Comparative Divisibility (AN4)

The Comparative Divisibility Problem (CDP) looks for an integer c that divides more elements in one multiset than another. Beyond the inclusion by Garey and Johnson in their 1979 book [16], there is remarkably little published research on the CDP or related divisibility comparison problems, especially under any parameterized complexity perspective.

More broadly and from a classical computational complexity perspective, Plaisted's work showed that some problems involving polynomial and integer divisibility are NP-hard or NP-complete [25]. Additionally, the complexity of Presburger arithmetic with divisibility has been studied in the context of linear arithmetic. The work by Lechner et al. shows that Presburger formulas that integrate integer divisibility predicates are NP-hard, but decidable in NEXPTIME [20].

However, none of these results directly address the CDP or similar divisibility problems under parameterizations. To our knowledge, no prior work investigates the fixed-parameter tractability of the CDP. In light of this gap, our work investigates whether parameters like sizes of maximum values in the input sequence result in FPT algorithms.

3 Subset Product

Subset Product Problem (SPP) - (SP14) [16, p. 224]
Input: Finite set A, a size $s(a) \in \mathbb{Z}^+$ for each $a \in A$, and a positive integer B.
Question: Is there a subset $A' \subseteq A$ such that the product of the sizes of the elements A' is exactly B?

The algorithm we present in this section is largely inspired by the work of Dutta and Rajasree [13]. They describe a pseudo-polynomial time approach to the SPP using a dynamic programming table over the exponent vectors of the prime factorization of the target product B. We construct the same dynamic programming table, but formalize their description into a concrete algorithm and adapt its structure to make it suitable for parameterized complexity analysis. In particular, we parameterize the SPP by k, the number of distinct prime factors of B, and show that under this parameterization the problem admits an XP-time algorithm.

Informally, XP-time (or slice-wise polynomial time) means that for each fixed value of the parameter k, the problem can be solved in polynomial time, although the degree of the polynomial may depend on k.

Formally, we define this concept as follows [9]:

Definition 3. *A parameterized problem $L \subseteq \Sigma^* \times \mathbb{N}$ is called **slice-wise polynomial** (XP) if there exists an algorithm $\mathcal{A}$ and two computable functions $f, g : \mathbb{N} \to \mathbb{N}$ such that, given $(x, k) \in \Sigma^* \times \mathbb{N}$, the algorithm $\mathcal{A}$ correctly decides whether $(x, k) \in L$ in time bounded by $f(k) \cdot |(x, k)|^{g(k)}$. The complexity class containing all slice-wise polynomial problems is called XP.*

With this definition in mind, we proceed with the description of the XP algorithm used to solve the SPP.

3.1 Parameterization by Number of Distinct Prime Factors of Target Product

We consider the *Subset Product Problem (SPP)* under the parameterization by k, the number of distinct prime factors of the target product B. Throughout this subsection, we assume that the prime factorization of B is given as part of the input. That is, we write:

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

where each p_i is a distinct prime and $e_i \geq 1$.

Our goal is to determine whether there exists a subset $A' \subseteq A$ such that the product of its elements equals B, using an algorithm whose complexity is controlled by the parameter k. In what follows, we show that the problem is slice-wise polynomial (XP) with respect to k. This is achieved by reformulating the SPP as a vector subset sum problem over the exponent vectors of the prime factorizations of the elements in A, and solving the resulting problem using a dynamic programming approach.

To this end, we introduce and prove Theorem 3.1

Theorem 3.1. *The* SPP *is slice-wise polynomial (XP) when parameterized by the number k of distinct prime factors of the target product B. The algorithm runs in time $O(n \cdot (E+1)^k \cdot \mathrm{poly}(L))$, where $E = \max(e_i)$ for $1 \leq i \leq k$ and L is the total bit-length of the input.*

Proof. Let $B = \prod_{i=1}^{k} p_i^{e_i}$ be the prime factorization of the target product. For each element $a \in A$, we factor it with respect to the primes $\{p_1, \ldots, p_k\}$, writing:

$$a = \prod_{i=1}^{k} p_i^{v_i(a)} \cdot r_a,$$

where r_a is the residual factor of a not divisible by any p_i. We discard any $a \in A$ for which $r_a \neq 1$, since such an a cannot contribute to a subset product equal to B.

Remark 3.1. *Although the elements $a \in A$ are not given in factored form, we can efficiently factor each with respect to the primes $\{p_1, \ldots, p_k\}$ provided in the input. Since only elements with all their prime factors* *in this set and $a \leq B$ can contribute to the target product, we discard all others. For the other elements, calculating each exponent $v_i(a)$ takes time polynomial in the bit-length of a and B when using trial division over k primes. Hence, this factoring step is efficient for this algorithm.*

Next, we define $\vec{e} = (e_1, \ldots, e_k)$ to denote the exponent vector of B. For each filtered a, we define its exponent vector as: $\vec{v}(a) = (v_1(a), \ldots, v_k(a))$.

Observe that the subset product condition can now equivalently be stated as finding a subset $A' \subseteq A$ such that:

$$\sum_{a \in A'} \vec{v}(a) = \vec{e}.$$

We solve this newly described problem using dynamic programming.

To do so, we construct a dynamic programming table $\mathrm{DP}[\vec{x}]$, indexed by vectors $\vec{x} \in \mathbb{N}^k$ with $0 \leq x_i \leq e_i$, which records whether a subset of A with exponent sum $\vec{x}$ exists.

We initialize the table with $\mathrm{DP}[\vec{0}] =$ `true` to represent the empty subset. We then iterate over the exponents vectors of the input elements in A, and for each such vector $\vec{v}(a)$, we perform the addition of a to every reachable state $\vec{x}$ currently marked `true` in DP. If $\vec{x} + \vec{v}(a) \leq \vec{e}$ coordinate-wise, we mark $\mathrm{DP}[\vec{x} + \vec{v}(a)] =$ `true`. To avoid double counting, we ensure that new additions are only made from DP states that were marked `true` before processing the current element a.

If at the end of this process $\mathrm{DP}[\vec{e}] =$ `true`, we know that there must exist some subset $A' \subseteq A$ such that:

$$\prod_{a \in A'} a = B.$$

In pseudocode the algorithm works as follows:

Algorithm 1 Algorithm for Subset Product

```
1:  Algorithm SolveSubsetProduct($A, B = \prod_{i=1}^{k} p_i^{e_i}$)
2:  Initialize DP as dictionary indexed by vectors $\vec{x} \in \mathbb{N}^k$ with $0 \leq x_i \leq e_i$
3:  Set DP[$\vec{0}$] ← true
4:  for each $a \in A$ do
5:      Factor $a$ over $\{p_1, \ldots, p_k\}$ to get $\vec{v}(a)$          ▷ See Remark 3.1
6:      if $a$ contains any prime outside $\{p_1, \ldots, p_k\}$ then
7:          continue                              ▷ Skip $a$ as it cannot contribute to $B$
8:      end if
9:      Let NewDP ← DP.copy()                         ▷ Copy to avoid double counting
10:     for each $\vec{x}$ in DP.keys() do
11:         if $\vec{x} + \vec{v}(a) \leq \vec{e}$ coordinate-wise then
12:             NewDP[$\vec{x} + \vec{v}(a)$] ← true
13:         end if
14:     end for
15:     DP ← NewDP
16: end for
17: if DP[$\vec{e}$] = true then
18:     return YES
19: else
20:     return NO
21: end if
```

To prove the correctness of this algorithm, we introduce and prove Claim 3.1:

Claim 3.1. $\mathrm{DP}[\vec{e}] =$ *`true`, and thus we have a `YES` instance, if and only if there exists a subset $A' \subseteq A$ such that the product of elements in A' equals B.*

Proof. We prove Claim 3.1 in both directions.

Suppose DP$[\vec{e}]$ = `true`. Then by the dynamic programming table, there exists a sequence of elements $a_1, \dots, a_j \in A$ such that, for each i, the exponent vector $\vec{v}(a_i)$ was added to a previously reachable DP state. Furthermore, the total sum of these vectors is $\vec{e}$. That is,

$$\sum_{i=1}^{j} \vec{v}(a_i) = \vec{e}.$$

Now, let us define $A' = \{a_1, \dots, a_j\}$. Since each a_i for $1 \le i \le j$ consists of only primes $\{p_1 \dots, p_k\}$, and the sum of their exponent vectors equals $\vec{e}$, we conclude that:

$$\prod_{a \in A'} a = \prod_{i=1}^{k} p_i^{e_i} = B.$$

Hence, A' is a subset of A whose product equals B.

Suppose now that there exists a subset $A' \subseteq A$ such that $\prod_{a \in A'} a = B$. Since by assumption $B = \prod_{i=1}^{k} p_i^{e_i}$, and each $a \in A'$ is composed of only primes from $\{p_1, \dots, p_k\}$ it follows that:

$$\sum_{a \in A'} \vec{v}(a) = \vec{e}.$$

We now turn our attention to Algorithm 1 to verify the algorithm correctly detects this. Initially, DP$[\vec{0}]$ = `true` on Line 3. As the algorithm iterates over each $a \in A$ on Line 4, it computes the exponent vector $\vec{v}(a)$ on Line 5. If a contains any prime outside the given primes, $\{p_1, \dots, p_k\}$, it is discarded on Line 7, and as such does not contribute to a possible solution. For all other elements, the loop on line Line 10 checks for each reachable state $\vec{x}$ whether $\vec{x} + \vec{v}(a) \le \vec{e}$ coordinate-wise on Line 11. If so, this state is updated to `true` on Line 12.

Since $\sum_{a \in A'} \vec{v}(a) = \vec{e}$, and each intermediate partial sum of the exponent vectors is less then $\vec{e}$ coordinate-wise, it follows inductively, by iterating through the loop, that each partial state $\vec{x_i}$ remains valid and is updated during the iteration over A. As $A' \subseteq A$, by the time all elements of A' have been processed, the table entry DP$[\vec{e}]$ is set to `true` and the algorithm will correctly returns `YES` on Line 18.

Finally, suppose no subset $A' \subseteq A$ exists such that $\prod_{a \in A'} a = B$. In this case, the exponent vector $\vec{e}$ is unreachable via any valid subset, and so the entry DP$[\vec{e}]$ remains unset. As a result, the algorithm correctly returns `NO` on Line 20.

This completes the proof of correctness.

□

Having proven the correctness of Algorithm 1, through the proof of Claim 3.1, all that remains is to analyze the running time of the algorithm.

The dynamic programming table DP is indexed by vectors $\vec{x} \in \mathbb{N}^k$ with $0 \le x_i \le e_i$. Observe that the total number of such vectors is:

$$\prod_{i=1}^{k} (e_i + 1) \le (E + 1)^k,$$

where $E = \max(e_i)$ for $1 \le i \le k$. As such, the size of the dynamic programming table is $(E + 1)^k$.

Next, observe that since $\prod_{a \in A} a > B$, taking logarithms gives:

$$\log B < \sum_{a \in A} \log a \le L,$$

where L is the total bit-length of the input. As such, we have:

$$E + 1 \le \log B + 1 \le L + 1 \le 2L,$$

which implies:

$$(E + 1)^k \le (2L)^k.$$

Now consider that on Line 4, the algorithm iterates over all values in A. There is n such values to go through. Factoring each $a \in A$ over the primes $\{p_1, \ldots, p_k\}$ takes $\mathcal{O}(k \cdot \mathrm{poly}(L))$ time per element via trial division. In the worst case, the loop on Line 10 will go through all $(E+1)^k$ states of the DP and compute $\vec{x} + \vec{v}(a) \leq c$, which constitutes k coordinate-wise comparisons. These comparisons take $\mathrm{poly}(L)$ time.

As a result, the final running time of the algorithm is:

$$\mathcal{O}(n \cdot (E+1)^k \cdot \mathrm{poly}(L)) \leq \mathcal{O}(n \cdot (2L)^k \cdot \mathrm{poly}(L)).$$

Remark 3.2. *Algorithm 1 is not FPT in parameter k, because the term $(2L)^k$ is exponential in the parameter. However, for each fixed k, the algorithm runs in polynomial time in the input size. As such, the algorithm belongs to the XP, or slice-wise polynomial time class.*

□

4 Quadratic Congruences

Quadratic Congruence Problem (QCP) - (AN1) [16, p. 249]
Input: Positive integers a, b and c.
Question: Is there a positive integer $x < c$ such that $x^2 \equiv a \pmod{b}$?

Solving quadratic congruences has wide-ranging applications across number theory and computer science. For instance, it can play a role in integer factorization algorithms [8], in probabilistic primality tests such as the Solovay–Strassen test [30], and in cryptography through the Goldwasser–Micali encryption scheme [17]. These practical applications, along with theoretical interest in quadratic residues, motivate a detailed examination of the decision problem posed above under different parameterizations.

In the following subsections we analyze two parameterizations of the QCP. First, by c, leading to a simple brute-force FPT algorithm. Then, by k, the number of distinct prime factors of b.

4.1 Parameterization by c, Upper Bound on x

We begin by analyzing the QCP under the parameterization by c, which serves as an upper bound on the variable x. We show that this problem admits a fixed-parameter tractable (FPT) algorithm when parameterized by c.

To this end, we state and prove the following theorem.

Theorem 4.1. *The* QCP *is fixed-parameter tractable (FPT) when parameterized by c, the upper bound on x. The algorithm has running time:*

$$O(c \cdot \text{poly}(L)),$$

where L is the total bit-length of the input.

Proof. We prove this by explicitly constructing an algorithm that decides the problem and show that its running time is bounded by $f(c) \cdot \text{poly}(L)$, where $f(c) = c$.

Recall that we are given three positive integers a, b, c and are asked whether there exists a positive integer $x < c$ such that:

$$x^2 \equiv a \pmod{b}.$$

Since x is bounded by c, and c is the parameter, we can afford to brute-force over all $x \in \{1, 2, \ldots, c-1\}$. For each candidate value, we check whether $x^2 \equiv a \pmod{b}$. If this is the case, we have a `YES` instance. In case we have tried all $x \in \{1, \ldots, c-1\}$ and none hold, we have a `NO` instance. The algorithm proceeds as follows:

Algorithm 2 Algorithm for QCP
```
1: Algorithm SolveQCP$(a, b, c)$
2: for $x = 1$ to $c - 1$ do
3:     if $x^2 \bmod b = a \bmod b$ then
4:         return YES
5:     end if
6: end for
7: return NO
```

We now prove the correctness of Algorithm 2.

Firstly, suppose there exists an $x < c$ such that $x^2 \equiv a \pmod{b}$. By construction this value of x will eventually be considered in the loop on Line 2. Since $x^2 \equiv a \pmod{b}$, the condition on Line 3 is satisfied, and hence the algorithm correctly returns `YES` on Line 4.

Suppose now that the algorithm returns `YES`. Then there exists an $x \in \{1, \ldots, c-1\}$ such that $x^2 \bmod b = a \bmod b$. By simple arithmetic we see that:

$$x^2 \mod b = a \mod b \Leftrightarrow x^2 \equiv a \pmod{b}.$$

Since $x < c$ and x is a positive integer, we know that x satisfies the condition of the QCP, and therefore is a valid solution.

Lastly, we must consider the case when the algorithm returns `NO`. In this case, the loop on Line 2 finishes without finding any $x < c$ such that the condition on Line 3 holds. Therefore, we conclude there is no $x < c$ such that $x^2 \mod b = a \mod b$, and the algorithm correctly returns `NO` on Line 7.

This completes the proof of correctness.

We now analyze the runtime of the algorithm.

The loop on Line 2 runs for at most $c - 1$ times. In each iteration, we compute $x^2 \mod b$ and compare it to $a \mod b$. These operations involve standard arithmetic and modular reduction on integers of bit-length at most L, and can hence be done in poly(L) time.

Thus, the total running time of Algorithm 2 is

$$\mathcal{O}(c \cdot \text{poly}(L)).$$

□

4.2 Parameterization by Number of Distinct Prime Factors of b

While parameterizing the QCP by c, the upper bound on x, yields an FPT algorithm, its practicality is limited to instances where solutions are small. We now analyze the QCP under the parameterization by k, the number of distinct prime factors of b. This approach uses the algebraic properties of modular arithmetic allowing for efficient solutions even when c is large.

Throughout this subsection, we assume that b is odd and that we are given its prime factorization in the input. As such, we have:

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

where each p_i is a distinct odd prime, and each $e_i \geq 1$. Our goal is to determine whether there exists a positive integer $x < c$ such that:

$$x^2 \equiv a \pmod{b}.$$

We proceed by analyzing the structure of this congruence through a series of classical number theoretic observations and lemmas.

Firstly, we notice that we can break down the congruence modulo b into it's k distinct prime factors and use the Chinese Remainder Theorem (CRT) to find partial solutions for each of these local congruences. Formalizing this we have that:

Observation 4.1. *Let $b = \prod_{i=1}^{k} p_i^{e_i}$ be the factorization of b into distinct odd primes. The congruence $x^2 \equiv a \pmod{b}$ is solvable if and only if, for each $1 \leq i \leq k$, the broken down congruence*

$$x^2 \equiv a \pmod{p_i^{e_i}}$$

has a solution.

Proof. As all moduli, $p_i^{e_i}$ for $1 \leq i \leq k$, are pairwise coprime, this result follows directly from the CRT [10, pp. 41–43]. □

Next, we show that each of the local congruences $x^2 \equiv a \pmod{p_i^{e_i}}$ can be solved independently.

Lemma 4.1. *Let p_i be an odd prime, $e_i \geq 1$ and $a \in \mathbb{Z}$. Suppose a has the factorization $a = p_i^{\ell} \cdot a'$, where $\ell \geq 0$ and $p_i \nmid a'$. Then we have the following cases:*

$$\begin{cases} \textit{If } \ell \textit{ is odd and } \ell < e_i, & \textit{then } x^2 \equiv a \pmod{p_i^{e_i}} \textit{ has no solution.} \\ \textit{If } \ell \geq e_i, & \textit{then } x^2 \equiv a \pmod{p_i^{e_i}} \textit{ has a unique solution } x \equiv 0 \pmod{p_i^{\lceil \frac{e_i}{2} \rceil}}. \\ \textit{If } \ell \textit{is even and } \ell < e_i, & \textit{then } x^2 \equiv a \pmod{p_i^{e_i}} \textit{ has either no solution or has two distinct solutions modulo } p_i^{e_i - \frac{\ell}{2}}. \end{cases}$$

Furthermore, each of these cases can be solved in randomized polynomial time.

Proof. This result follows directly from finding the solutions, if any exist, to $x^2 \equiv a \mod p_i^{e_1}$ in randomized polynomial time using the Tonelli [31]-Shanks [28] algorithm and then applying Hensel's Lemma [10, pp. 44–45] to lift the found solutions to $x^2 \equiv a \pmod{p_i^{e_i}}$. □

Having shown how to compute the solutions to each local congruence $x^2 \equiv a \pmod{p_i^{e_i}}$, we now show how to combine these local solutions into a full global solution modulo b.

Lemma 4.2. *Suppose that for each $1 \leq i \leq k$, the local congruence:*

$$x^2 \equiv a \pmod{p_i^{e_i}},$$

has either one or two solutions. We partition the indices $\{1, \ldots, k\}$ into:

$$\begin{cases} S_1 = \{i \mid \textit{congruence has 1 solution}\} \\ S_2 = \{i \mid \textit{congruence has 2 solutions}\} \end{cases}$$

For each choice of local solution x_i, the system of congruences:

$$\begin{cases} x \equiv x_i \pmod{p_i^{e_i}} & \textit{for } i \in S_2 \\ x \equiv x_i \pmod{p_i^{\lceil e_i/2 \rceil}} & \textit{for } i \in S_1 \end{cases}$$

has a unique solution modulo $b' \mid b$, where:

$$b' = \prod_{i \in S_2} p_i^{e_i} \cdot \prod_{j \in S_1} p_j^{\lceil e_j/2 \rceil}$$

Moreover, this solution satisfies $x^2 \equiv a \pmod{b}$.

Proof. In the case that all local congruences have 2 solutions, the result of a unique solution modulo b follows directly from all moduli $p_i^{e_i}$ being coprime and a constructive proof of the CRT [10, pp. 41–43]. However, if a local congruence has only one solution, then x, our global solution, must satisfy a fixed condition modulo $p_i^{e_i}$, constraining the global solution more strictly.

As a result, when we apply the CRT to combine the local congruences, we know by Lemma 4.1 that we will be working modulo a smaller modulus $b' \mid b$, rather than the full modulo b. Nonetheless, by the CRT, we still find a unique global solution modulo b'.

In both cases, since $x \equiv x_i \pmod{p_i^{e_i}}$ and by basic modular arithmetic $x_i^2 \equiv a \pmod{p_i^{e_i}}$, we have that:

$$x^2 \equiv x_i^2 \equiv a \pmod{p_i^{e_i}}.$$

As this hold for all i, and since all $p_i^{e_i}$ are coprime, it follows that $x^2 \equiv a \pmod{b}$ [7, pp. 951–952]. □

Corollary 4.1. *Under the assumptions of Lemma 4.2, the congruence*

$$x^2 \equiv a \pmod{b}$$

has at most 2^k distinct solutions.

Proof. By Lemma 4.1, each local congruence $x^2 \equiv a \pmod{p_i^{e_i}}$ has at most 2 solutions. In the worst case all k congruences would have exactly two solutions. In this case there would be 2^k congruence systems for the CRT to solve in Lemma 4.2. As the CRT finds a unique solution to each system, we know there would be at most 2^k solutions to the congruence $x^2 \equiv a \pmod{b}$. □

With these structural insights in place, we are now ready to prove the main result of this section. We now present Theorem 4.2.

Theorem 4.2. *The* QCP *is fixed-parameter tractable (FPT) when parameterized by k, the number of distinct prime factors of b. The algorithm's running time is:*

$$\mathcal{O}(2^k \cdot poly(L)),$$

where L is the total bit-length of the input.

Proof. We prove this by using the structural insights presented in this subsection.

We start by looking at each $i \in \{1, \dots, k\}$ and attempt to solve the local congruence:

$$x^2 \equiv a \pmod{p_i^{e_i}}.$$

By Lemma 4.1, each local congruence either has no solution, one solution, or two solutions. If any local congruence has no solution, then by Observation 4.1, the global congruence $x^2 \equiv a \pmod{b}$ also has no solution. In this case we have a `NO` instance.

Assuming all local congruences have solutions, we proceed to construct all systems of congruences. By Corollary 4.1, there are at most 2^k such congruence systems.

For each of the at most 2^k combinations, we apply Lemma 4.2 to construct a unique solution x modulo b' for some $b' \mid b$. We know that each such x satisfies:

$$x^2 \equiv a \pmod{b}.$$

Lastly, for each constructed value $x \mod b$ or $x \mod b'$, we compute the minimal positive representative in the congruence class and check whether it satisfies $x < c$. If any such x satisfies this, we know we have a `YES` instance. Otherwise, if for none of the at most 2^k global solutions $x < c$, then we have a `NO` instance.

Finally, we analyze the running time of the algorithm outlined above.

Each local congruence is solved in randomized polynomial time [28, 31, 10, pp. 44–45]. There are k such congruences. Next, we consider that there are at most 2^k congruence systems that are combined in polynomial time using the CRT [6, pp. 12–22]. Lastly, checking whether $x < c$ is also polynomial in the input size.

As a result, the total running time of the algorithm is:

$$\mathcal{O}(2^k \cdot \text{poly}(L)).$$

□

5 Simultaneous Incongruences

Simultaneous Incongruence Problem (SIP) - (AN2) [16, p. 249]
Input: A collection $\{(a_1, b_1), \dots, (a_n, b_n)\}$ of ordered pairs of positive integers, with $a_i < b_i$ for $1 \le i \le n$.
Question: Does there exist an integer x such that $x \not\equiv a_i \pmod{b_i}$ for all $1 \le i \le n$?

Solving systems of modular congruences is a recurring theme in both number theory and constraint based scheduling. While the Chinese Remainder Theorem addresses the existence of an integer that satisfies a system of congruences, the SIP focuses on finding an integer that avoids a system of congruences. This formulation can model scenarios such as assigning time slots in scheduling where certain periods must be avoided.

In the following subsections, we present two distinct FPT approaches for the SIP, both parameterized by the number of input pairs n. One via an Integer Linear Programming (ILP) encoding, and another using a threshold function to partition the input instance into different cases.

5.1 Parameterization by n using ILP encoding

Theorem 5.1. *The* SIP *is fixed-parameter tractable (FPT) when parameterized by n, the number of input pairs. The FPT algorithm's running time is*

$$f(n) \cdot poly(L), \quad where\ f(n) = (3n+1)^{\mathcal{O}(3n+1)}$$

and L is the total bit-length of the input.

Proof. We prove this by reducing the SIP problem to an instance of Integer Linear Programming, constructing an Integer Linear Program (ILP) with a fixed number of variables, $3n+1$, and then applying Lenstra's algorithm [21] to solve the ILP. As such, we proceed with the encoding of the SIP into an ILP.

ILP solvers operate on systems of linear (in)equalities over integer variables [1]. However, as stated in the name, the constraints in the SIP are modular incongruences of the form $x \not\equiv a_i \pmod{b_i}$. As such, to apply ILP techniques, we must first express the problem's constraints using linear (in)equalities.

We observe that with the properties of the problem instance $0 < a_i < b_i$ for all $i \in \{1, \dots, n\}$, the following are equivalent:

$$x \not\equiv a_i \pmod{b_i} \iff x \bmod b_i \neq a_i \tag{1}$$

The central idea in the ILP modeling is to explicitly represent the remainder of $x \bmod b_i$. Instead of treating the modulo operator directly, which is non-linear and hence not allowed in ILPs, we model it by introducing auxiliary variables. For each i, we write $x = b_i \cdot q_i + r_i$, where q_i is the integer quotient of $\frac{x}{b_i}$ and r_i is the remainder. By additionally constraining r_i to lie in the interval $[0, b_i - 1]$, we ensure that $r_i = x \bmod b_i$.

The goal now is to enforce that for every i, $r_i \neq a_i$. As this inequality is not in the allowed ILP format yet, we encode this constraint as a disjunction of two linear inequalities [1]:

$$r_i - a_i \ge 1 \quad \vee \quad r_i - a_i \le -1$$

Disjunctions are not directly allowed in ILPs. As such, we introduce an auxiliary binary variable $z_i \in \{0,1\}$ and use a standard big-M trick to distinguish both cases of the disjunction [2]. As a result, the disjunction above is modeled as the following system of inequalities:

$$\begin{cases} r_i - a_i + Mz_i & \ge 1 \\ r_i - a_i - M(1 - z_i) & \le -1 \\ z_i & \ge 0 \\ z_i & \le 1 \end{cases}$$

where M is a sufficiently large constant (e.g. any $M > \max(b_i)$, $\forall i \in \{1, \dots, n\}$). Note that since $0 \le r_i \le b_i - 1$ and $0 < a_i < b_i$ by construction, the absolute difference $|r_i - a_i|$ is strictly less than b_i. Therefore, choosing $M > \max(b_i)$ ensures that the magnitude of M dominates any possible value of $r_i - a_i$, ensuring that the system behaves correctly in both cases.

Combining all the above components for each input pair (a_i, b_i), we obtain the full system of linear constraints that defines the ILP encoding of the SIP:

$$\begin{cases} x - b_i \cdot q_i - r_i = 0 & (2) \\ r_i \geq 0 & (3) \\ r_i \leq b_i - 1 & (4) \\ r_i - a_i + Mz_i \geq 1 & (5) \\ r_i - a_i - M(1 - z_i) \leq -1 & (6) \\ z_i \geq 0 & (7) \\ z_i \leq 1 & (8) \end{cases}$$

This ILP can now be solved to find the solution x. To ensure correctness, we must now prove that a valid solution to the ILP is also a valid solution to the original SIP. Furthermore, a solution to the original problem must also be a solution to the newly created ILP. To this end, we introduce and prove the following lemma.

Lemma 5.1. *The ILP has a feasible solution if and only if the original simultaneous incongruence problem has a solution.*

Proof. Assume that the constructed ILP has a feasible solution. Then there exists an integer assignment to the variables $x, q_1, \dots, q_n, r_1, \dots, r_n$, and, $z_1, \dots, z_n$ such that for each $i \in \{1, \dots, n\}$, the linear constraints (2)-(8) in the above ILP system hold.

From constraint (2) we have for each i:

$$x = b_i \cdot q_i + r_i$$

This is the standard integer division identity, where q_i is the quotient and r_i is the remainder. Constraints (3) and (4) ensure that $0 \leq r_i \leq b_i - 1$. Therefore, we know $r_i = x \bmod b_i$ for all i.

Constraints (5), (6), (7), and (8) ensure that $r_i \neq a_i$. We verify this by contradiction.

Suppose, for the sake of contradiction, that $r_i = a_i$. From constraints (7) and (8), we know that $z_i \in \{0, 1\}$. We consider both values of $z_i \in \{0, 1\}$:

If $z_i = 0$, then constraint (5) becomes: $r_i - a_i \geq 1$, which simplifies to $0 \geq 1$, a contradiction.

If $z_i = 1$, then constraint (6) becomes: $r_i - a_i \leq -1$, which simplifies to $0 \leq -1$, also a contradiction.

Thus, in both cases, one of the constraints fails to hold, contradicting the assumption that all constraints (2)-(8) are satisfied, an assumption that was made at the beginning of the proof of Lemma 5.1. As such, we conclude that any feasible solution to the system must satisfy $r_i \neq a_i$.

Since constraints (2)-(4) ensure that $r_i = x \bmod b_i$, and the above proof by contradiction shows that constraints (5)-(8) enforce that $r_i \neq a_i$, we conclude that:

$$x \bmod b_i \neq a_i$$

Using the equivalence (1) stated before, this leads to:

$$x \not\equiv a_i \pmod{b_i} \quad \text{for all } i \in \{1, \dots, n\}$$

Hence, we infer that if the ILP has a feasible solution, the original instance of the SIP also has a solution.

It remains to show that a solution to the original problem is also a solution to the constructed Integer Linear Program. We do this by defining variable assignments such that all constraints (2)-(8) of the ILP are satisfied.

Suppose there exists an integer x such that:

$$x \not\equiv a_i \pmod{b_i}, \forall i \in \{1, \dots, n\}$$

Or equivalently using (1):

$$x \bmod b_i \neq a_i$$

Then, we define for each i:

$$r_i := x \bmod b_i \in \{0, 1 \ldots, b_i - 1\}$$
$$q_i := \frac{x - r_i}{b_i} \in \mathbb{Z}$$

ensuring that $x = b_i \cdot q_i + r_i$, i.e. the standard division identity. Note that although we use the non-linear operations mod here, it is not part of the ILP formulation. We use it only to create witness assignments to variables such that they satisfy the ILP's linear constraints.

Since $r_i \neq a_i$, we know that $r_i < a_i$ or $r_i > a_i$. We define z_i as:

$$z_i := \begin{cases} 0 & \text{if} \;\; r_i > a_i \\ 1 & \text{if} \;\; r_i < a_i \end{cases}$$

This choice guarantees that either:

$$r_i - a_i \geq 1 \quad (\text{if} \;\; z_i = 0), \quad \text{or} \quad r_i - a_i \leq -1 \quad (\text{if} \;\; z_i = 1)$$

ensuring that constraints (5) and (6) are satisfied for a sufficiently large constant M.

We now show that with the variable assignments outlined above all constraints of the ILP system are satisfied. Constraint (2) holds by the definition of r_i and q_i, constraints (3) and (4) hold because $r_i \in [0, b_i - 1]$, constraints (5) and (6) hold by the choice of z_i and a sufficiently large constant M, and lastly, constraints (7) and (8) hold since $z_i \in \{0, 1\}$. Therefore, we know that there is a valid assignment to all variables $(x, q_1, \ldots, q_n, r_1, \ldots, r_n, z_1, \ldots, z_n)$ of the ILP. As such, we can conclude that if there exists a solution to the original SIP problem, then there is also a feasible solution to the constructed Integer Linear Program.

This completes the proof of Lemma 5.1, that the ILP has a feasible solution if and only if the original simultaneous incongruence problem has a solution.

□

By Lemma 5.1 we have shown that the original simultaneous incongruence problem (SIP) is equivalent to finding a solution to the constructed integer Linear Program. To conclude the proof of Theorem 5.1, we analyze the time complexity of solving the ILP.

Observe that the ILP has $3n + 1$ variables: the sought-after solution variable x, along with auxiliary integer variables $q_1, \ldots, q_n$ and $r_1, \ldots, r_n$ and binary variables $z_1, \ldots, z_n$. The dimension of this program is $d = 3n + 1$. Since the number of variables depends only on the parameter n, the dimension of the program is fixed for any given instance of the parameterized simultaneous incongruence problem. By Lenstra's algorithm [21], we know an Integer Linear Program in fixed dimension can be solved in time $f(d) \cdot poly(L)$, where $f(d) = d^{\mathcal{O}(d)}$ and L is the bit-length of the input. Therefore, the SIP is solvable in time $(3n + 1)^{\mathcal{O}(3n+1)} \cdot poly(L)$, which is fixed-parameter tractable with respect to the parameter n. □

5.2 Parameterization by n using Threshold Function

We now present an alternative fixed-parameter tractable algorithm for the SIP, based on analyzing the size of the input moduli. The general idea is as follows. If all moduli are large, the instance is trivially a `YES` instance by a simple pigeonhole argument. If all moduli are small, we can brute-force search for a solution over a bounded search space. In the remaining cases, we use a threshold function to split the moduli into groups of small and large moduli. We then show that a partial solution that avoids the small moduli constraints can be extended to a full solution that also avoids the large moduli constraints.

To formalize this strategy, we first define the threshold function that determines when a modulus is considered large or small. We then prove an upper bound on this function, which will allow us to bound the running time of the algorithm.

Definition 4. *We define a threshold function $T : \mathbb{N} \to \mathbb{N}$ recursively by:*

$$T(1) = 1 \quad T(k) = T(k-1) \cdot (k-1) + 1 \quad \textit{for } k > 1.$$

Before proceeding with the algorithm, we prove that this function grows at most exponentially in k^k, which will be crucial for bounding the runtime of the brute-force search.

Lemma 5.2. *For all $k \geq 1$, the threshold function stated in Definition 4 satisfies $T(k) \leq k^k$.*

Proof. We prove this by induction on k.

Base Case ($k = 1$): By definition, $T(1) = 1$. Since $1^1 = 1$, the base case, $T(1) = 1 \leq 1^1$, holds.

Inductive Step: Assume the claim holds for $k = m$. We now show that it holds for $k = m + 1$. By definition we have that:

$$T(m+1) = T(m) \cdot m + 1.$$

By the inductive hypothesis we know that $T(m) \leq m^m$. Thus:

$$\begin{aligned} T(m+1) &\leq m^m \cdot m + 1 \\ &= m^{m+1} + 1 \end{aligned}$$

Since $m \geq 1$, we have that $m^{m+1} + 1 \leq (m+1)^{m+1}$ by the exponential growth of $(m+1)^{m+1}$. Thus:

$$T(m+1) \leq (m+1)^{m+1},$$

and we conclude by induction that the claim holds for all $k \geq 1$. □

With the threshold function defined, we now address the simplest case, where all moduli are large. This case can be resolved immediately, as noted in the following observation.

Observation 5.1. *If every modulus satisfies $b_i > (n+1)$, then the instance is a* `YES` *instance. Specifically there exists an $x \in \{1, \dots, n+1\}$ such that:*

$$x \not\equiv a_i \pmod{b_i} \quad \textit{for all } i \leq n.$$

Proof. Each congruence $x \equiv a_i \pmod{b_i}$ excludes at most one value from $\{1, \dots, n+1\}$. With n constraints and $n+1$ candidate values, at least one x must remain by the pigeonhole principle. □

We now turn to the case where some moduli are small. In such cases, we can efficiently search for a partial solution x_0 to the small moduli constraints, as formalized in Observation 5.2 below.

Observation 5.2. *Let $\{(a_1, b_1), \dots, (a_n, b_n)\}$ be an instance of the* SIP. *Take the function $T : \mathbb{N} \to \mathbb{N}$ to be defined as in Definition 4. Let $b_1, \dots, b_k \leq (n+1)^{T(k)}$, and define $B_k := \prod_{i=1}^{k} b_i$. Then for any $k \leq n$, we can enumerate all $x_0 \in \{1, \dots, B_k - 1\}$ and check whether*

$$x_0 \not\equiv a_i \pmod{b_i} \quad \textit{for all } i \leq k$$

in fixed-parameter tractable time with respect to n.

Proof. In the worst case all $b_i = (n+1)^{T(k)}$. From this we conclude that $B_k \leq (n+1)^{k \cdot T(k)}$. Additionally, as $k \leq n$ and each incongruence check can be done in time polynomial in the total bit length of the input, L, we can conclude this brute-force search is FPT in n. □

Once we have found such a value x_0, we want to extend it to a full solution that is also incongruent to the large moduli constraints. The following lemma shows that this is always possible by checking only a bounded number of shifted values.

Lemma 5.3. *Let $\{(a_1, b_1), \dots, (a_n, b_n)\}$ be an instance of the* SIP, *and let $T : \mathbb{N} \to \mathbb{N}$ be defined as in Definition 4.*

Suppose that for some $k \in \{1, \dots, n\}$, the instance satisfies:

$$\begin{cases} b_1, \dots, b_k & \leq (n+1)^{T(k)} \\ b_{k+1}, \dots, b_n & \geq (n+1)^{T(k+1)} \end{cases}$$

Define $B_k := \prod_{i=1}^{k} b_i$. Then, for any $x_0 \in \{1, \ldots, B_k - 1\}$ such that:

$$x_0 \not\equiv a_i \pmod{b_i} \quad \textit{for all } i \leq k,$$

there exists $\ell \in \{0, \ldots, n\}$ such that the value

$$x = x_0 + \ell \cdot B_k$$

satisfies

$$x \not\equiv a_i \pmod{b_i} \quad \textit{for all } i \leq n.$$

Proof. By Observation 5.2, we can find, if it exists, an x_0 such that

$$x_0 \not\equiv a_i \pmod{b_i} \quad \text{for all } i \leq k$$

in FPT time. Assume such an x_0 exists.

Now consider the $n + 1$ values:

$$x_\ell = x_0 + \ell \cdot B_k \quad \text{for } \ell = 0, 1, \ldots, n.$$

Since all $b_1, \ldots, b_k$ divide B_k, each x_ℓ is congruent to x_0 modulo every b_i for $i \leq k$, and hence satisfies the first k constraints.

Consider now any fixed index $j > k$. Assume, for the sake of contradiction, that two distinct values x_{ℓ_1} and x_{ℓ_2} are congruent modulo b_j. That is:

$$x_{\ell_1} \equiv x_{\ell_2} \pmod{b_j}.$$

Using the definition of x_ℓ, we compute:

$$\begin{aligned} x_{\ell_1} &= x_0 + \ell_1 \cdot B_k \\ x_{\ell_2} &= x_0 + \ell_2 \cdot B_k. \end{aligned}$$

As such $x_{\ell_1} \equiv x_{\ell_2} \pmod{b_j}$ becomes:

$$\begin{aligned} x_{\ell_1} &\equiv x_{\ell_2} \pmod{b_j} \\ x_0 + \ell_1 \cdot B_k &\equiv x_0 + \ell_2 \cdot B_k \pmod{b_j} \\ (\ell_1 - \ell_2) \cdot B_k &\equiv 0 \pmod{b_j}. \end{aligned}$$

Hence, $b_j \mid (\ell_1 - \ell_2) \cdot B_k$. Since $\ell_1 \neq \ell_2$, we know $1 \leq |\ell_1 - \ell_2| \leq n$. Now, using the bound $B_k \leq (n+1)^{k \cdot T(k)}$, we get:

$$\begin{aligned} |\ell_1 - \ell_2| \cdot B_k &\leq n \cdot (n+1)^{k \cdot T(k)} \\ &< (n+1) \cdot (n+1)^{k \cdot T(k)} \\ &= (n+1)^{k \cdot T(k)+1} \\ &= (n+1)^{k \cdot T(k+1)} \\ &\leq b_j, \end{aligned}$$

where the last equality follows from re-writing Definition 4 as $T(k+1) = k \cdot T(k) + 1$.

Thus, the product $(\ell_1 - \ell_2) \cdot B_k < b_j$, and hence cannot be divisible by b_j unless $\ell_1 = \ell_2$. This contradicts our assumption that the values are different. Therefore, we can conclude that at most one value x_ℓ can be excluded per constraint j.

Since there are at most $n - k$ such constraints, and $n + 1$ partial solutions, at least one partial solution x_ℓ must remain such that:

$$x_\ell \not\equiv a_i \pmod{b_i} \quad \text{for all } i \leq n.$$

□

Combining Lemma 5.3 and Observation 5.1, we cover all possible cases defined by our threshold function. We are now fully equipped to state and prove the main result.

Theorem 5.2. *The* SIP *is fixed-parameter tractable (FPT) when parameterized by n, the number of input pairs. The FPT algorithm's running time is*

$$\mathcal{O}((n+1)^{n^{(n+1)}} \cdot \textit{poly}(L)),$$

where L is the total bit-length of the input and $T : \mathbb{N} \to \mathbb{N}$ is the threshold function defined as in Definition 4.

Proof. We first check if all $b_i > (n+1)$ for $1 \leq i \leq n$. If so, then by Observation 5.1, the instance is a `YES` instance.

Otherwise, there must exist a unique index $k \in \{1, \ldots, n\}$ such that the moduli satisfy:

$$b_1, \ldots, b_k \leq (n+1)^{T(k)} \quad \text{and} \quad b_{k+1}, \ldots, b_n \geq (n+1)^{T(k+1)},$$

where T is the threshold function from Definition 4.

Once this value of k is identified for the input instance, we define:

$$B_k := \prod_{i=1}^{k} b_i$$

and use Observation 5.2 to search for a value $x_0 \in \{1, \ldots, B_k - 1\}$ satisfying:

$$x_0 \not\equiv a_i \pmod{b_i} \quad \text{for all } i \leq k.$$

If such an x_0 exists, then by Lemma 5.3 there must exist some $\ell \in \{0, \ldots, n\}$ such that for $x = x_0 + \ell \cdot B_k$:

$$x \not\equiv a_i \pmod{b_i} \quad \text{for all } i \leq n.$$

In this case, the instance is a `YES` instance.

If no such x_0 exists, because every $x_0 \in \{1, \ldots, B_k - 1\}$ satisfies $x_0 \equiv a_i \pmod{b_i}$ for some $i \leq k$, then no valid partial solution exists, and we conclude that the instance is a `NO` instance.

It remains to analyze the runtime of the algorithm described above.

Each $b_i \leq (n+1)^{T(k)}$ for $i \leq k$. As such:

$$B_k \leq (n+1)^{k \cdot T(k)}.$$

In the worst case, $k = n$, and so:

$$B_n \leq (n+1)^{n \cdot T(n)}.$$

From Lemma 5.2 we know that the bound on $T(n)$ is $T(n) \leq n^n$, thus:

$$B_n \leq (n+1)^{n^{(n+1)}}.$$

Each check whether a candidate x_0 satisfies the k small moduli constraints can be done in time polynomial in the bit-length L of the input.

Therefore, the total running time of the algorithm is:

$$\mathcal{O}((n+1)^{n^{(n+1)}} \cdot \text{poly}(L)).$$

□

6 Quadratic Diophantine Equations

Quadratic Diophantine Equation Problem (QDEP) - (AN8) [16, p. 251]
Input: Positive integers a, b, and c.
Question: Are there positive integers x and y such that $ax^2 + by = c$?

In this section, we aim to bridge the gap between the extensive study of Diophantine equations in number theory and the limited research of these problems from a parameterized complexity perspective. Specifically, we consider the QDEP, an NP-complete decision problem involving both quadratic and linear terms. We present two parameterizations for the QDEP, both resulting in FPT algorithms. Firstly, we parameterize by an upper bound on y, and secondly we parameterize by the coefficient b of the linear term by.

6.1 Parameterization by Upper Bound on y

We begin by showing that the QDEP admits a fixed-parameter tractable algorithm when parameterizing the problem by the upper bound k on y. This approach uses a brute-force search over all possible values of y to efficiently solve the problem.

To this extent, we state and prove the following theorem:

Theorem 6.1. *The* QDEP *is fixed-parameter tractable (FPT) when parameterized by k, the upper bound on y. The FPT algorithm's running time is:*

$$\mathcal{O}(k \cdot poly(L)),$$

where L is the total bit-length of the input.

Proof. We prove this by explicitly constructing an algorithm that decides the problem and show that its running time is bounded by $f(k) \cdot poly(L)$, where k is the parameter and L is the total bit-length of the input.

Recall that we are given positive integers a, b, c, and a parameter k, and are asked whether there exists positive integers x and y such that:

$$ax^2 + by = c \quad \text{and} \quad y \leq k$$

The key idea is to brute-force all possible values $y \in \{1, 2, \ldots, k\}$ and check whether for any such y, the equation $ax^2 + by = c$ admits a solution for some $x \in \mathbb{Z}_{>0}$. In pseudocode, the algorithm works as follows:

Algorithm 3 Algorithm for QDEP

```
1: Algorithm SolveQDEP($a, b, c, k$)
2: for $y = 1 \ldots k$ do
3:     if $a \mid (c - b \cdot y)$ then
4:         $x^2 \leftarrow (c - b \cdot y)/a$
5:         if $x^2$ is a perfect square and $x^2 > 0$ then
6:             return YES
7:         end if
8:     end if
9: end for
10: return NO
```

We introduce and prove the following claim to ensure the correctness of Algorithm 3.

Claim 6.1. *There exist positive integers x and $y \leq k$ such that $ax^2 + by = c$ if and only if the algorithm returns* YES*. Otherwise, the algorithm returns* NO*.*

Proof. Assume that there exist positive integers x and y such that $y \leq k$ and $ax^2 + by = c$. Then we can rearrange this equation to get:

$$c - by = ax^2$$

As x is an integer this implies that $a \mid (c - b \cdot y)$, and that $\frac{(c-by)}{a} = x^2$ is a perfect square. Since x is a positive integer, $x^2 > 0$. As such, we know that when the algorithm checks this value of y, both conditions in Line: 3 and Line: 5 will be satisfied respectively, and the algorithm will return `YES`.

Conversely, suppose that the algorithm returns `YES`. Then the conditions in Line: 3 and Line: 5 must be satisfied. In other words, there exists some $y \in \{1, 2, \ldots, k\}$ such that $a \mid (c - b \cdot y)$, and $\frac{(c-b\cdot y)}{a}$ is a positive perfect square. Now, let $x^2 = \frac{(c-b\cdot y)}{a}$ and define x to be the positive square root of x^2. This step reconstructs the value of x such that the original equation $ax^2 + by = c$ holds. Since x is a positive integer by construction and y was selected from $\{1, 2 \ldots, k\}$, the solution pair (x, y) satisfies all constraints of the problem.

Noted, if no solution (x, y) exists with $y \leq k$, then the algorithm iterates through all possible $y \in \{1, \ldots, k\}$ and fails to find any y satisfying both conditions on Line: 3 and Line: 5. Thus, in Line 10, the algorithm correctly returns `NO` if and only if no solution exists after checking all possible values of y.

This completes the proof of Claim 6.1, that there exist positive integers x and $y \leq k$ such that $ax^2+by = c$ if and only if Algorithm 3 returns `YES`. □

To conclude the proof of Theorem 6.1, it remains to analyze the running time of Algorithm 3.

In Line 2, the algorithm iterates over at most k possible values of y in the range $\{1, 2, \ldots, k\}$. For each such y, it performs the following steps: Line 3 computes the value $c - b \cdot y$ and checks whether this value is divisible by a. If so, the quotient $\frac{(c-b\cdot y)}{a}$ is calculated and Line 5 checks if this is a positive perfect square.

All of these operations involve integers whose bit-length is at most L, where L is the total bit-length of the input. The arithmetic operations and the perfect square check can be performed in time polynomial in L. Therefore, each iteration of the loop runs in $\mathit{poly}(L)$ time.

Since the loop at Line 2 runs for k iterations, the total running time of the algorithm is $\mathcal{O}(k) \cdot \mathit{poly}(L)$. This is of the form $f(k) \cdot \mathit{poly}(L)$ where $f(k) = k$, which is fixed-parameter tractable with respect to the parameter k. □

6.2 Parameterization by Coefficient b

We now study the QDEP under a different parameterization. We parameterize by the coefficient b of the linear term by. Before proving Theorem 6.2, we outline a key structural property of the original equation that allows us to reduce the search space for x to a finite set.

The key observation is that any solution $x \in \mathbb{Z}_{>0}$ must satisfy the congruence $ax^2 \equiv c \pmod{b}$. This restricts the possible values of x to a subset of residue classes modulo b. For each such congruence class, the values of x are of the form $x = x_0 + tb$, where $x_0 \in \{1, 2, \ldots, b\}$ and $t \in \mathbb{Z}_{\geq 0}$.

The lemma below shows that within any such congruence class, it suffices to check only the smallest value x_0. If it does not give a valid solution, because $y \leq 0$, then all other larger values in that congruence class will also fail.

Lemma 6.1. *Let $x = x_0 + tb$ for some $x_0 \in \mathbb{Z}_{>0}$ such that $ax_0^2 \equiv c \pmod{b}$, and $t \in \mathbb{Z}_{\geq 0}$. If*

$$y = \frac{c - ax_0^2}{b} \leq 0,$$

then for all $t \geq 1$, we also have:

$$y' = \frac{c - a(x_0 + tb)^2}{b} \leq 0.$$

That is, if the smallest solution, x_0, in a congruence class does not make $y \geq 1$, then all larger congruent values, $x = x_0 + tb$ for $t \geq 1$, will also fail to do so.

Proof. We know that $x = x_0 + tb$ for some $t \in \mathbb{Z}_{\geq 0}$. We begin by expanding ax^2:

$$ax^2 = a(x_0 + tb)^2 = a(x_0^2 + 2tbx_0 + t^2b^2) = ax_0^2 + 2atbx_0 + at^2b^2.$$

Now we consider the numerator of y' with this expansion:

$$c - ax^2 = c - (ax_0^2 + 2atbx_0 + at^2b^2) = (c - ax_0^2) - 2atbx_0 - at^2b^2.$$

Since $a, b, x_0, t \geq 1$, the terms $2abtx_0$ and at^2b^2 are strictly positive. We know from our initial assumption that $c - ax_0^2 \leq 0$. As we are subtracting positive terms from $c - ax_0^2$, we know that:

$$c - ax^2 = (c - ax_0^2) - (2abtx_0 + at^2b^2) < c - ax_0^2 \leq 0.$$

Therefore:

$$y' = \frac{c - ax^2}{b} < \frac{c - ax_0^2}{b} \leq 0,$$

concluding that $y' \leq 0$ for all $t \geq 1$.

□

Equipped with Lemma 6.1 we can prove Theorem 6.2.

Theorem 6.2. *The* QDEP *is fixed-parameter tractable (FPT) when parameterized by* b*, the coefficient of* y*. The FPT algorithm's running time is:*

$$\mathcal{O}(b \cdot poly(L)),$$

where L *is the total bit-length of the input.*

Proof. We begin by rewriting the original equation $ax^2 + by = c$ to express y explicitly:

$$y = \frac{c - ax^2}{b}.$$

For this equation to have a positive integer solution (x, y), the following two conditions must be satisfied. Firstly, the numerator $c - ax^2$ must be divisible by b, so as to ensure y is an integer. Secondly, the numerator must be positive, to ensure $y > 0$. Formalized, these conditions translate to:

$$\begin{cases} ax^2 & \equiv c \pmod{b} \\ c - ax^2 & > 0 \end{cases}$$

Lemma 6.1 gives the algorithm's core idea. Since we only need to check the minimal value x_0 from each congruence class where $ax_0^2 \equiv c \pmod{b}$, we can restrict our search to the finite set $\mathcal{R} = \{x_0 \in \{1, \ldots, b\} \mid ax_0^2 \equiv c \pmod{b}\}$. For each $x_0 \in \mathcal{R}$, we compute $y = \frac{(c - ax_0^2)}{b}$ and return `YES` if y is positive. The lemma guarantees that we do not need to check a larger $x = x_0 + tb$ if x_0 witnesses a `NO` instance.

This algorithm is described in pseudo code below.

Algorithm 4 FPT Algorithm for QDEP parameterized by b

```
1: Algorithm SolveQDEPByB($a, b, c$)
2: for $x_0 = 1$ to $b$ do
3:     if $ax_0^2 \bmod b = c \bmod b$ then
4:         $y \leftarrow (c - ax_0^2)/b$
5:         if $y > 0$ then
6:             return YES
7:         end if
8:     end if
9: end for
10: return NO
```

We prove the correctness of Algorithm 4 in two directions.

First, suppose a solution $(x, y) \in \mathbb{Z}^2_{>0}$ exists. Let $x_0 = x \bmod b$ be the minimal value in its congruence class, such that $x = x_0 + tb$ for some $t \in \mathbb{Z}_{\geq 0}$. Since $ax^2 \equiv c \pmod{b}$, it follows by modular arithmetic that $ax_0^2 \equiv c \pmod{b}$ as well. Therefore, we know $x_0 \in \mathcal{R}$ and the algorithm will consider it.

We now argue that this x_0 must also witness a `YES` instance. Since x is a solution, we know that:

$$y = \frac{c - a(x_0 + tb)^2}{b} > 0.$$

Applying Lemma 6.1 in contrapositive form, we conclude that if $x = x_0 + tb$ yields $y > 0$, then x_0 must also satisfy $\frac{c-ax_0^2}{b} > 0$. As such, the algorithm will find that x_0 satisfies the modular condition and also that $y > 0$, and will correctly return `YES` on Line 6.

Conversely, suppose the algorithm returns `YES`. Then there exists some $x_0 \in \{1, \dots, b\}$ such that $ax_0^2 \equiv c \pmod{b}$ and $y = \frac{c-ax_0^2}{b} > 0$. It follows that the pair $(x, y) = (x_0, \frac{c-ax_0^2}{b})$ consists of positive integers and satisfies:

$$ax^2 + by = ax_0^2 + b \cdot \frac{c - ax_0^2}{b} = c.$$

Therefore, (x, y) is a valid solution to the equation, completing the proof of correctness.

It remains to analyze the runtime of this algorithm.

The main loop on Line 2 iterates over all values $x_0 \in \{1, 2, \dots, b\}$, performing at most b iterations. In each iteration, the algorithm computes $ax_0^2 \bmod b$ and compares it to $c \bmod b$ on Line 3 to check the modular condition. This comparison requires integer multiplication, squaring, and modular reduction, all of which can be performed in poly(L) time, where L is the total bit-length of the input.

If the condition on Line 3 holds, then $y = \frac{c-ax_0^2}{b}$ is calculated which also takes poly(L) time. On Line 5 we check if $y > 0$, which also takes poly(L) time.

Since each iteration takes poly(L) time and the loop runs at most b times, the total running time is:

$$\mathcal{O}(b \cdot \text{poly}(L))$$

□

7 Comparative Divisibility

Comparative Divisibility Problem (CDP) - (AN4) [16, p. 249]
Input: Sequences $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_m$ of positive integers.
Question: Is there a positive integer c such that the number of i for which c divides a_i more than the number of j for which c divides b_j?

The CDP asks a natural yet subtle question about the divisibility properties of two multisets of integers. While its formulation is simple, asking whether there exists an integer that divides more element in one set than the other, the underlying structure complicates things quickly.

In this section, we explore the fixed-parameter tractability of the problem under three different parameterizations, using the structure of the input sequences to our advantage.

7.1 Preliminaries and Notation

In the following results, we use the same notation and assumptions throughout to ensure consistency.

Let $A_s = (a_1, a_2, \ldots, a_n)$ and $B_s = (b_1, b_2, \ldots, b_m)$ denote the two input sequences of positive integers as stated in the description of the CDP. Let $A = \{a_1, a_2, \ldots, a_n\}$ and $B = \{b_1, b_2, \ldots, b_m\}$ be the corresponding multisets formed from these sequences.

Throughout the following proofs in this section we assume, without loss of generality, that A_s and B_s are sorted in non-decreasing order:

$$a_1 \leq a_2 \leq \cdots \leq a_n \quad \text{and} \quad b_1 \leq b_2 \leq \cdots \leq b_m$$

and therefore A and B are also sorted in non-decreasing order.

Recall that with this ordering:

$$a_n = \max(A), \quad b_m = \max(B).$$

We define the divisibility count functions $C_A(x)$ and $C_B(x)$ for any positive integer x as follows:

$$C_A(x) := |\{a \in A : x \mid a\}|, \quad C_B(x) := |\{b \in B : x \mid b\}|.$$

These functions count how many elements of A and B are divisible by x, respectively.

The key question of the CDP is whether there exists a positive integer c such that $C_A(c) > C_B(c)$. In the following subsections, we analyze the CDP under different parameterizations. All results in this section follow the setup and notation defined above.

7.2 Parameterization by Sequence Length of A_s (n)

Before proving Theorem 7.1, we establish a key structural lemma that will allow us to significantly reduce the search space of potential solutions. This lemma shows that we only need to consider the greatest common divisors of all non-empty submultisets of A as candidates for c, rather than examining all possible integers $c \in \{1, 2, \ldots, a_n\}$. However, it is important to note that a_n may be significantly smaller than n. In such cases, parameterizing by a_n can lead to a much more efficient algorithm than parameterizing by n. This motivates the approach taken in Theorem 7.2.

Lemma 7.1. *If there exists a positive integer c such that $C_A(c) > C_B(c)$, then the greatest common divisor, d, of the multiset $S = \{a \in A : c \mid a\}$ also satisfies this property.*

Proof. Let $S = \{a \in A : c \mid a\}$ and $d = \gcd(S)$, where S is a mutliset. Since c divides all $a \in S$ we have by construction that $c \mid d$. As such, for $C_A(d) = |\{a \in A : d \mid a\}|$ the following inequality holds for set A:

$$C_A(d) \geq C_A(c).$$

On the other hand, as $c \mid d$, any element of B divisible by d must also be divisible by c. This implies that the following inequality, with $C_B(d) = |\{b \in B : d \mid b\}|$, holds for B:

$$C_B(d) \leq C_B(c).$$

Combining both inequalities with the original assumption that c divides more elements of A than B, we obtain:

$$C_A(d) \geq C_A(c) > C_B(c) \geq C_B(d)$$

proving, after simplification, that d satisfies: $C_A(d) > C_B(d)$. This concludes the proof of Lemma 7.1, that when $c > 0$ exists satisfying the divisibility inequality, $d = \gcd(S)$ for $S = \{a \in A : c \mid a\}$ does as well. □

Equipped with Lemma 7.1 we are ready to prove Theorem 7.1.

Theorem 7.1. *The CDP is fixed-parameter tractable (FPT) when parameterized by n, the number of values in the first input sequence A_s. The FPT algorithm's running time is:*

$$\mathcal{O}(2^n \cdot ((n+m) \cdot poly(L)))$$

where m is the number of values in the second input sequence B_s and L is the total bit-length of the input.

Proof. Lemma 7.1 provides the foundational idea for our FPT approach. It shows that if there exists a positive integer c such that the number of elements in A divisible by c is strictly greater than the number of elements in B that are divisible by c, then the same property holds for $d = \gcd(S)$, where $S = \{a \in A : c \mid a\}$. This means that to decide whether such a value c exists, it suffices to consider the greatest common divisors of all non-empty submultisets of A.

The algorithm takes as input both multisets A and B of positive integers. It proceeds by iterating over all non-empty submultisets $S \subseteq A$ and computing $d = \gcd(S)$. For each such d, we compute $C_A(d)$ and $C_B(d)$. If for any such d the condition: $C_A(d) > C_B(d)$ holds, then we have a `YES` instance. If for all non-empty submultisets the condition $C_A(d) > C_B(d)$ does not hold, we have a `NO` instance. In pseudocode, the algorithm works as follows:

Algorithm 5 FPT Algorithm for CDP parameterized by n

```
1: Algorithm SolveCDP($A, B$)
2: for each nonempty submultiset $S \subseteq A$ do
3:     $d \leftarrow \gcd(S)$
4:     $C_A \leftarrow 0$
5:     for $a \in A$ do
6:         if $d \mid a$ then
7:             $C_A \leftarrow C_A + 1$
8:         end if
9:     end for
10:    $C_B \leftarrow 0$
11:    for $b \in B$ do
12:        if $d \mid b$ then
13:            $C_B \leftarrow C_B + 1$
14:        end if
15:    end for
16:    if $C_A > C_B$ then
17:        return YES
18:    end if
19: end for
20: return NO
```

We introduce and prove the following claim to ensure the correctness of Algorithm 5.

Claim 7.1. *A positive integer c satisfies $C_A(c) > C_B(c)$ if and only if Algorithm 5 returns `YES`.*

Proof. Suppose there exists a positive integer c such that the condition below holds:

$$C_A(c) > C_B(c).$$

Now, let $S = \{a \in A : c \mid a\}$ and let $d = \gcd(S)$, where S is a multiset. By construction, $c \mid a$ for all $a \in S$ and as such $c \mid d$. By Lemma 7.1 it follows that:

$$C_A(d) > C_B(d).$$

Notice that S is a non-empty submultiset of A. As such, the loop at Line 2 will eventually consider this S. At Line 3, $d = \gcd(S)$ is calculated. Then the nested loops in Lines 5-9 and Lines 11-15 calculate $C_A(d)$ and $C_B(d)$, respectively. As $C_A(d) > C_B(d)$, the condition on Line 16 holds and the algorithm will correctly return `YES` on Line 17. Hence, when such a c exists, Algorithm 5 correctly identifies this `YES` instance.

Conversely, suppose that Algorithm 5 returns `YES`. Then there exists a non-empty submultiset $S \subseteq A$ such that, with $d = \gcd(S)$, we have:

$$C_A(d) > C_B(d).$$

Let $c = d$. Since c is a positive integer, and satisfies the required inequality, it follows that a value c exists, precisely when $c = d$, for which the condition of the CDP is satisfied when Algorithm 5 returns `YES`.

This completes the proof of Claim 7.1, ensuring the correctness of Algorithm 5. □

To complete the proof of Theorem 7.1, it remains to analyze the runtime of Algorithm 5.

The main loop at Line 2 iterates over all non-empty submultisets $S \subseteq A$. Since A contains n elements, there are $2^n - 1$ such subsets.

For each submultiset S, the algorithm computes the greatest common divisor $d = \gcd(S)$. This computation can be performed in time polynomial in the bit-length of the elements in S, which is bounded by $\text{poly}(L)$, where L is the total bit-length of the input.

Next, the algorithm computes the counts $C_A(d)$ and $C_B(d)$ in the loops on Lines 5-9 and Lines 11-15, respectively. Each of these loops goes through all elements of A and B and performs the divisibility check with the candidate value d. The sizes of A and B are n and m, respectively. Each divisibility test, $d \mid a$ and $d \mid b$, takes $\text{poly}(L)$ time.

Thus, the total time spent per submultiset is $\mathcal{O}((n+m) \cdot poly(L))$. Since there are $2^n - 1$ subsets in total to consider, the overall running time of the algorithm is:

$$\mathcal{O}(2^n \cdot ((n+m) \cdot poly(L))).$$

□

7.3 Parameterization by Maximum Element of A (a_n)

We now consider the CDP under a different parameterization. Instead of parameterizing by n, the size of input sequence A_s, we parameterize by the largest value in A_s. Specifically, we let $a_n = \max(A)$ and consider all integers $c \in \{1, 2, \ldots, a_n\}$ as candidates. This parameterization can offer significant efficiency improvements over the algorithm discussed in Section 7.2, particularly when $a_n \ll n$. The following theorem establishes the fixed parameter tractability of the CDP under this parameterization.

Theorem 7.2. *The* CDP *is fixed-parameter tractable (FPT) when parameterized by $a_n = \max(A)$. The algorithm runs in time $\mathcal{O}(a_n \cdot (n+m) \cdot poly(L))$, where n and m are the lengths of the input sequences A_s and B_s, respectively, and L is the total bit length of the input.*

Proof. We observe that any integer $c > a_n$ cannot divide any value in A, because all values in A are at most a_n. As such, for any $c > a_n$ we have $C_A(c) = 0$ and therefore values of $c > a_n$ are guaranteed to result in a `NO` instance.

It follows that we only need to consider the values of $c \in \{1, 2, \ldots, a_n\}$. For each such c, we compute $C_A(c)$ and $C_B(c)$. If for any such c we find that $C_A(c) > C_B(c)$, then we have a `YES` instance. If no such c satisfies this, we have a `NO` instance. This algorithm is described in pseudocode below:

Algorithm 6 FPT Algorithm for CDP parameterized by $\max(A_s)$

```
Algorithm SolveCDPByMax$A_s(A, B)$
$a_n \leftarrow \max(A)$
for $c = 1$ to $a_n$ do
    $C_A \leftarrow 0$
    for $a \in A$ do
        if $c \mid a$ then
            $C_A \leftarrow C_A + 1$
        end if
    end for
    $C_B \leftarrow 0$
    for $b \in B$ do
        if $c \mid b$ then
            $C_B \leftarrow C_B + 1$
        end if
    end for
    if $C_A > C_B$ then
        return YES
    end if
end for
return NO
```

We proceed by establishing the correctness of Algorithm 6.

Suppose that there exists a positive integer c such that $C_A(c) > C_B(c)$. Since any valid c must satisfy $c \leq a_n$, it must be that $c \in \{1, 2, \ldots, a_n\}$. Thus, it is guaranteed that Algorithm 6 will eventually consider it in the loop on Line 3. As such, when the algorithm considers this c, the condition on Line 16 will hold and the algorithm will return `YES` on Line 17. In case such a c does not exist, the condition on Line 16 will never hold. As such, once all iterations of the loop on Line 3 have occurred, the algorithm will correctly return `NO` on Line 20.

We now analyze the running time of Algorithm 6. The loop in Line 3 runs at most a_n times. In each iteration the counts $C_A(c)$ and $C_B(c)$ are calculated on Lines 5-9 and Lines 11-15, respectively. This involves going through all n elements in A and all m elements in B. For each element a divisibility check is carried out, which can be performed in time polynomial in the bit-length of the number.

Thus, the total runtime of Algorithm 6 is:

$$\mathcal{O}(a_n \cdot (n + m) \cdot \text{poly}(L)).$$

□

7.4 Parameterization by Maximum Element of B (b_m)

We now consider the CDP under a third parameterization, this time from the perspective of the second input sequence. Rather than bounding the size or values of A_s, we investigate the case where the maximum value in sequence B_s is bounded.

Let $b_m = \max(B)$. The key insight here is that for any positive integer $c > b_m$, it is guaranteed that c cannot divide any element of B. As such, we have that $C_B(c) = 0$ for any $c > b_m$. We formalize this in the following observation:

Observation 7.1. *Let $B = \{b_1, b_2, \ldots, b_m\}$ be a multiset of positive integers, and let $b_m = \max(B)$. Then for any integer $c > b_m$, we have:*

$$\forall b \in B, \quad c \nmid b.$$

That is, no integer $c > b_m$ divides any element of B, and so $C_B(c) = 0$ for all $c > b_m$.

Remark 7.1. *As a consequence of Observation 7.1, if $a_n > b_m$, then setting $c = a_n$ yields:*

$$C_A(c) \geq 1 \quad \textit{and} \quad C_B(c) = 0,$$

and hence $C_A(c) > C_B(c)$, which guarantees a `YES` instance for the CDP.

Observation 7.1 and Remark 7.1 give a key insight into how we can significantly simplify our approach under this parameterization. More specifically, we can focus on two cases. Firstly, we iterate over all $c \in \{1, 2, \ldots, b_m\}$ and check if $C_A(c) > C_B(c)$ for any such c. If we have not witnessed a `YES` instance yet, we can then check if $a_n > b_m$. If this holds we can know that setting $c = a_n$ yields $C_B(c) = 0$ and $C_A(c) \geq 1$, which witnesses a `YES` instance.

The following theorem formalizes this approach and proves that the CDP is also fixed-parameter tractable under the $b_m = \max(B)$ parameterization.

Theorem 7.3. *The Comparative Divisibility Problem (CDP) is fixed-parameter tractable when parameterized by $b_m = \max(B_s)$. The algorithm runs in time $\mathcal{O}(b_m \cdot (n + m) \cdot poly(L))$, where n and m are the lengths of the input sequences A_s and B_s, and L is the total bit-length of the input.*

Proof. Take a_n and b_m to be defined as in Section 7.1.

We first iterate over all values $c \in \{1, \ldots, b_m\}$. For each such c, we compute $C_A(c)$ and $C_B(c)$. If at any point $C_A(c) > C_B(c)$, we return `YES`. If this is not the case, we check whether $a_n > b_m$. By Observation 7.1, we know that for any $c > b_m$, $C_B = 0$. By Remark 7.1, if $a_n > b_m$, then setting $c = a_n$ implies $C_A \geq 1$, since at least $a_n \mid a_n$, and hence $C_A > C_B$. Therefore, in this case, we also return `YES`.

In case we have still not witnessed a `YES` instance, then no valid c exists for which $C_A > C_B$, and we return `NO`. This process is described in the algorithm below:

Algorithm 7 FPT Algorithm for CDP parameterized by max(B)

```
1: Algorithm SolveCDPByMaxBs($A, B$)
2: $a_n \leftarrow \max(A)$
3: $b_m \leftarrow \max(B)$
4: for $c = 1$ to $b_m$ do
5:     $C_A \leftarrow 0$
6:     for $a \in A$ do
7:         if $c \mid a$ then
8:             $C_A \leftarrow C_A + 1$
9:         end if
10:    end for
11:    $C_B \leftarrow 0$
12:    for $b \in B$ do
13:        if $c \mid b$ then
14:            $C_B \leftarrow C_B + 1$
15:        end if
16:    end for
17:    if $C_A > C_B$ then
18:        return YES
19:    end if
20: end for
21: if $a_n > b_m$ then
22:     return YES
23: end if
24: return NO
```

It remains to prove the correctness and analyze the running time of Algorithm 7.

Suppose there exists a positive integer c such that $C_A(c) > C_B(c)$. If $c \leq b_m$, then the algorithm will consider this value in some iteration during the loop on Line 4. In this iteration $C_A(c)$ and $C_B(c)$ will be calculated and compared on Line 17. As $C_A(c) > C_B(c)$, the algorithm will return `YES` on Line 18.

If instead $c > b_m$, then by Observation 7.1 we know that $C_B(c) = 0$. In this can, we apply Remark 7.1, which tells us it suffices to check whether $a_n > b_m$. This condition is checked on Line 21. In case it holds, Algorithm 7 correctly returns `YES` on Line 22.

Finally, if no such c exists such that Algorithm 7 has returned `YES` in the cases outlined above, then for all c, we have $C_A(c) \leq C_B(c)$. In this case, the algorithm correctly returns `NO` on Line 24.

Now we analyze the running time of Algorithm 7. The loop on Line 4 iterates over all $c \in \{1, 2, \ldots, b_m\}$, performing b_m iterations in total. In each iteration, the algorithm performs $\mathcal{O}(n+m)$ divisibility checks to calculate $C_A(c)$ and $C_B(c)$. Each divisibility check can be performed in time polynomial in the bit-length of the dividend, which is bounded by the total bit-length L of the input.

Hence the total running time of the algorithm is:

$$\mathcal{O}(b_m \cdot (n+m) \cdot \text{poly}(L)).$$

□

The approach we took in this subsection can be seen as a combination of earlier insights. We know by Remark 7.1 that when $a_n > b_m$ we have a `YES` instance. In the remaining case where $a_n \leq b_m$, the algorithm from Theorem 7.2 is, in fact, sufficient to solve the problem. Since it only considers values up to a_n, this results in a runtime of $\mathcal{O}(a_n \cdot (n+m) \cdot \text{poly}(L))$. As $a_n \leq b_m$ this runtime is asymptotically not worse than $\mathcal{O}(b_m \cdot (n+m) \cdot \text{poly}(L))$, the runtime for Algorithm 6.

8 Conclusion

The guiding research question of this thesis was: *"What parameters are suitable for parameterization of NP-hard problems in number theory?"* To answer this question, we analyzed five different NP-hard number-theoretic decision problems from a parameterized complexity perspective. For each problem we explored a variety of parameterizations, identifying seven different types of parameters and constructing nine FPT algorithms and one XP algorithm by the end.

One key insight we found is that encoding decision problems as Integer Linear Programs (ILPs), and then bounding the number of constraints or variables, allows us to apply Lenstra's Algorithm [21]. Lenstra's Algorithm solves ILPs in a fixed number of constraints or variables in fixed-parameter tractable time. This approach was effective for our analysis of the Simultaneous incongruence Problem.

For other problems, such as the Quadratic Diophantine Equation Problem, we showed that bounding the solution variable, x in this case, or algebraic coefficients, such as b, reduced the problem to a brute-force search over a polynomially bounded space, with each candidate solution being verifiable in polynomial time.

For problems like the Quadratic Congruence Problem and the Subset Product Problem, parameterizing by the number of distinct prime factors of an input integer proved to be successful. Interestingly, this single parameterization led to two distinctly different algorithms for each problem. For the QCP, we applied standard number-theoretic techniques like the Chinese Remainder Theorem and Hensel's Lemma. On the other hand for the SPP, we transformed the problem into a vector subset sum and applied a dynamic programming approach over the target product's exponents vectors.

We also found that parameterizing by input size, such as the number of elements in the input sequence for the Comparative Divisibility Problem, can lead to efficient algorithms when combined with structural insights. For example, by iterating over all non-empty subsets of the input and checking the subsets' gcd values, rather than every possible value for c. Although this is an effective approach, in cases where the number of input values is significantly larger than the maximum value of the inputs, it is more effective to parameterize by the maximum value in the input.

A common theme throughout our analysis is that there is no ideal parameterization for all instances of a given problem. The structure of the input often determines which parameterization is most suitable. For instance, the Comparative Divisibility Problem admits multiple useful parameterizations, each leading to FPT algorithms for different input instances. Moreover, the same parameterization can lead to different approaches, as shown by our two fundamentally different FPT algorithms for the Simultaneous Incongruence Problem when parameterized by n, the number of inputs pairs.

However, directly answering the research question posed at the beginning of this thesis, we found that some parameter types proved effective across different problems. In particular, parameterizing by the absolute value of the largest element in the input, by the input length, and by the number of distinct prime factors in an input integer often lead to FPT or XP algorithms.

In summary, this thesis demonstrates the potential of parameterized complexity and fixed-parameter tractability in the field of number theory. We have shown that many number-theoretic NP-hard problems, which are intractable in general, admit efficient FPT or XP algorithms under the right parameterizations. These results extend the reach of fixed-parameter tractability in the domain of number theory and the algorithmic techniques we developed, particularly those based on ILP encoding and brute-force search over bounded solution spaces, together with the promising parameterization types we identified can serve as a reusable toolkit for future researchers in this field to use.

References

[1] Mokhtar S. Bazaraa, John J. Jarvis, and Hanif D. Sherali. *Introduction*, chapter 1, pages 1–43. John Wiley & Sons, Ltd, 2009.

[2] Mokhtar S. Bazaraa, John J. Jarvis, and Hanif D. Sherali. *Starting Solution and Convergence*, chapter 4, pages 151–199. John Wiley & Sons, Ltd, 2009.

[3] Richard Bellman. *Dynamic Programming.* Princeton University Press, USA, 2010.

[4] Ivan Bliznets, Jesper Nederlof, and Krisztina Szilágyi. Parameterized algorithms for covering by arithmetic progressions. In Henning Fernau, Serge Gaspers, and Ralf Klasing, editors, *SOFSEM 2024*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pages 125–138. Springer Science and Business Media Deutschland GmbH, 2024.

[5] Jianer Chen, Iyad A. Kanj, and Ge Xia. Improved parameterized upper bounds for vertex cover. In Rastislav Královič and Paweł Urzyczyn, editors, *Mathematical Foundations of Computer Science 2006*, pages 238–249, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[6] Henri Cohen. *A Course in Computational Algebraic Number Theory.* Springer Publishing Company, Incorporated, 2010.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[8] Karl-Dieter Crisman. *Number Theory: In Context and Interactive.* 2019.

[9] Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Daniel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms.* Springer Publishing Company, Incorporated, 1st edition, 2015.

[10] Abhijit Das. *Computational Number Theory.* Chapman & Hall/CRC, 1st edition, 2013.

[11] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity.* Monographs in Computer Science. Springer, 1999.

[12] Rodney G. Downey and Michael R. Fellows. *Fundamentals of Parameterized Complexity.* Springer Publishing Company, Incorporated, 2013.

[13] Pranjal Dutta and Mahesh Sreekumar Rajasree. Efficient reductions and algorithms for subset product. In *Algorithms and Discrete Applied Mathematics: 9th International Conference, CALDAM 2023, Gandhinagar, India, February 9–11, 2023, Proceedings*, page 3–14, Berlin, Heidelberg, 2023. Springer-Verlag.

[14] Henning Fernau, Fedor V. Fomin, Geevarghese Philip, and Saket Saurabh. On the parameterized complexity of vertex cover and edge cover with connectivity constraints. *Theoretical Computer Science*, 565:1–15, 2015.

[15] Rodney G. and Michael R. Fellows. Fixed-parameter tractability and completeness i: Basic results. *SIAM J. Comput.*, 24:873–921, 08 1995.

[16] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., 1990.

[17] Shafi Goldwasser and Silvio Micali. Probabilistic encryption & how to play mental poker keeping secret all partial information. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, page 365–377, New York, NY, USA, 1982. Association for Computing Machinery.

[18] Lenwood S. Health. Covering a set with arithmetic progressions is np-complete. *Information Processing Letters*, 34(6):293–298, 1990.

[19] Thomas Koshy. *Elementary number theory with applications*, chapter Fundamentals, pages 1–3. Academic Press, Amsterdam, 2nd edition, 2007.

[20] Antonia Lechner, Joel Ouaknine, and James Worrell. On the Complexity of Linear Arithmetic with Divisibility . In *2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 667–676, Los Alamitos, CA, USA, July 2015. IEEE Computer Society.

[21] H. W. Lenstra. Integer programming with a fixed number of variables. *Mathematics of Operations Research*, 8(4):538–548, 1983.

[22] Richard Mandel and Alexander Ushakov. The diophantine problem for systems of algebraic equations with exponents. *Journal of Algebra*, 636:779–803, 2023.

[23] Yu. V. Matiyasevich. The diophantineness of enumerable sets. *Doklady Akademii Nauk SSSR*, 191(2):279–282, 1970.

[24] Neeldhara Misra, Frances Rosamond, and Meirav Zehavi. Special issue "new frontiers in parameterized complexity and algorithms": Foreward by the guest editors. *Algorithms*, 13:260–263, 09 2020.

[25] David A. Plaisted. Some polynomial and integer divisibility problems are np-hard. In *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, pages 264–267, 1976.

[26] Tim Randolph. *Exact and Parameterized Algorithms for Subset Sum Problems.* PhD thesis, Columbia University, 2024.

[27] Timothy W. Randolph and Karol Wegrzycki. Parameterized algorithms on integer sets with small doubling: Integer programming, subset sum and k-sum. In *Embedded Systems and Applications*, 2024.

[28] Daniel Shanks. Five number theoretical algorithms. In *Proceedings of the 2nd Manitoba Conference on Numerical Mathematics*, pages 51–70, 1972.

[29] Nigel P. Smart. *Introduction*, page 1–14. London Mathematical Society Student Texts. Cambridge University Press, 1998.

[30] R. Solovay and V. Strassen. A fast monte-carlo test for primality. *SIAM Journal on Computing*, 6(1):84–85, 1977.

[31] Alberto Tonelli. Bemerkung über die Auflösung quadratischer Congruenzen. *Nachrichten von der Gesellschaft der Wissenschaften zu Göttingen, Mathematisch-Physikalische Klasse*, pages 344–346, 1891.