$i\mathbb{Z}$-$\mathbb{F}$ramework:

$\mathbb{P}$ragmatic $\mathbb{P}$rime $\mathbb{S}$ieve

Hazem Mounir
*zprime*137@*gmail.com*

March 2025

Abstract

Prime sieving has long been a cornerstone of number theory, yet modern cryptographic and computational applications demand methods that overcome significant space, time, and output constraints. This study introduces the iZ-Framework—a novel suite of algorithms that dramatically reduces the computational burden of prime sieving by continuously applying modular space reduction. The Classic Sieve-iZ algorithm reduces the candidate search space to $n/3$ by focusing on numbers that are co-prime with 2 and 3 (numbers of the form $6x \pm 1$). Building upon this, the Segmented Sieve-iZm algorithm utilizes bigger wheels and strategic segmentation to achieve constant space complexity $O(1)$, while simultaneously reducing the overall sieve workload and improving execution time. The VX6-Sieve algorithm further refines the process by attaining $O(n)$ bit complexity and introducing an efficient method for sieving large datasets via encoding prime gaps. Comprehensive benchmarks demonstrate that the iZ-Framework offers a more efficient and scalable solution compared to existing prime sieving methods. The paper culminates with a production-ready implementation of the Random-iZprime algorithm, which generates random primes of large bit-sizes with low latency and is particularly well-suited for cryptographic applications.

Introduction

Prime numbers have captivated mathematicians for centuries as the fundamental building blocks of number theory. Over time, many creative prime sieve algorithms—from the classic Sieve of Eratosthenes to more sophisticated methods like Euler's and Atkin's sieves—have been developed to identify primes within large numerical ranges. However, modern applications in cryptography and high-performance computing demand algorithms that can efficiently handle enormous input sizes, a task at which traditional methods falter due to their unbounded requirements for space, time, and output complexity.

Challenges in Prime Sieve Algorithms

Space Complexity: Traditional sieve algorithms require an array of size $n + 1$ to mark composites up to n, leading to linear space complexity $O(n)$. Segmented sieves significantly reduce the memory footprint to $O(\sqrt{n})$ by processing smaller sections sequentially. However, even this improvement becomes prohibitive for extremely large values of n.

Time Complexity: The classical Sieve of Eratosthenes performs approximately

$$n \sum_{2 \leq p \leq \sqrt{n}} \frac{1}{p}$$

composite-marking operations, which translates to a time complexity of $O(n \log \log n)$. Euler's sieve optimizes this by marking each composite only once at the cost of some bookkeeping, achieving linear bit complexity $O(n)$. Meanwhile, the Sieve of Atkin offers theoretical speed advantages. While artistic, it comes with a heavy preprocessing.

Output Complexity: Beyond sieve computation, storing large datasets of primes presents its own challenge. As primes grow larger, their bit-size increases, and storing each prime individually becomes increasingly inefficient. This issue necessitates a more scalable output format to manage the ever-expanding size of prime data.

The $i\mathbb{Z}$-Framework

The iZ-Framework re-imagines prime sieving by addressing these challenges head-on. It is based on the fundamental observation that all primes greater than 3 are of the form $6x \pm 1$. By restricting consideration to this subset of numbers, the framework initially reduces the search space for primes to roughly $\frac{1}{3}$ of the natural numbers and further refines it through continuous modular space reduction—ultimately narrowing the candidate set to as little as $\frac{4}{100}$ of the original space.

Two core structures form the backbone of the framework:

1. iZ Structure: The remaining candidates after excluding multiples of 2 and 3, denoted as the $i\mathbb{Z}$ set, are the numbers of the form $6x \pm 1$. By studying the multiplicative properties of this set, this structure initially reduces the search space to $n/3$ and serves as the foundation for further refinements.

2. iZ-Matrix Structure: The iZ-Matrix arranges the $i\mathbb{Z}$ set into two-dimensional grids of carefully chosen segment sizes, with each row representing a segment of candidate numbers. This segmentation strategy enables a standardized constant $O(1)$ space complexity instead of arbitrary $\sqrt{n}$ values, while also improving the bit complexity. Further optimizations—such as the VX6-Sieve, which encodes prime gaps rather than storing full prime numbers—address the output complexity challenge by significantly reducing the data footprint.

Key Contributions

While theoretically grounded, this work emphasizes algorithmic innovation over mere theoretical exposition. Its primary contributions are:

- **Sieve-iZ:** A simple yet powerful algorithm that leverages the iZ multiplicative properties to reduce the search space for primes from n to $n/3$, cutting a large portion of the redundancies that arise from considering all natural numbers. This optimization yields a faster execution time against other sieve algorithms.

- **Sieve-iZm:** By decoupling the segment size from the input n, and building on the structure of the iZ-Matrix, this algorithm relies on strategic segment sizes to achieve constant space complexity $O(1)$ using as little as 0.8 MB of memory, while also reducing the sieve workload by skipping composite marking for primes up to 23, offering a significant performance boost over existing methods.

- **VX6-Sieve:** This algorithm bounds the sieve workload per segment by processing the iZ-Matrix with a horizontal vector size defined as

$$vx6 = 5 \times 7 \times 11 \times 13 \times 17 \times 19 = 1,616,615 \text{ bits},$$

  which spans a numerical interval $S = 6 \times vx6 = 9,699,690$. It achieves a linear bit complexity $O(n)$ with exceptional constant factors—approximately $\frac{1}{2}S$ bitwise operations and $\frac{4}{100}S$ primality testing operations—and introduces an efficient method for encoding prime gaps, thereby enhancing primes dataset scalability.

- **Random-iZprime:** As a practical application of the iZ-Framework, this algorithm efficiently generates large, random primes with low latency. When compared with GMP's `mpz_nextprime` and OpenSSL's `BN_generate_prime`, Random-iZprime demonstrates competitive efficiency with better scalability via parallelism, making it particularly suitable for large-scale cryptographic applications.

Each algorithm is developed in a dedicated chapter, after exploring its theoretical underpinnings.

Benchmark Environment and Reproducibility

All benchmarks were conducted on an Apple MacBook Pro M1 with 8 GB of RAM and a 256 GB SSD, running macOS Sequoia 15.1.1. The algorithms are implemented in C and compiled with gcc to fully leverage the available hardware. Comprehensive benchmarking tools, provided as part of the iZ-library, ensure that our results are reproducible and allow for consistent performance comparisons across different systems.

Source Code Repository

The complete implementations, including test units, are encapsulated in the **Open-Source** *iZ-library*, available at: `https://github.com/Zprime137/iZ-lib`.

Key components of the iZ-library are included in the appendix of this paper.

To Aurel Stevens

Contents

Chapter 1

iZ-Structure: Theoretical Foundation

1.1 Introduction

In this chapter, we lay the theoretical groundwork for the *iZ-Framework*. Our starting point is the well-known optimization that, aside from the primes 2 and 3, every prime number falls into one of the two residue classes $\pm 1 \pmod 6$. This insight reveals that two-thirds of the natural numbers can be excluded from consideration when searching for primes.

The iZ-Framework capitalizes on this observation by partitioning the remaining candidates—the $i\mathbb{Z}$ set, which is defined as the union of two symmetric subsets $i\mathbb{Z}^-$ and $i\mathbb{Z}^+$—into a compact, index-based structure we call the **X-Arrays**. Within this framework, we introduce the **iZ Function**, a mapping that assigns each number in the $i\mathbb{Z}$ set a unique index within these arrays. This representation not only reduces space requirements but also simplifies subsequent operations.

Central to our approach is the *iZ Theorem*, which rigorously characterizes the multiplicative properties of the iZ function. By analyzing how indices behave under multiplication, the theorem establishes precise, index-based criteria for detecting composite numbers. Furthermore, the theorem naturally extends to exponentiation, unveiling elegant recursive patterns that further simplify the arithmetic of large numbers.

Finally, we present the **X*p*-Wheel**, a systematic and efficient mechanism for marking composite numbers within the X-Arrays. By exploiting the index-based equivalences derived from the iZ Theorem, the Xp-Wheel minimizes the operational overhead typically associated with prime sieving. Our analysis demonstrates that this method offers a scalable and space-efficient alternative to traditional sieves, enabling deterministic generation of all primes up to a given limit.

In what follows, we review key classical concepts that are essential in the context of prime sieve. We then introduce the theoretical constructs of the iZ-Framework in detail, setting the stage for the pragmatic, performance-oriented algorithms developed in subsequent chapters.

1.2 Preliminaries

A robust theoretical foundation is essential for developing efficient prime sieve algorithms. In this section, we briefly review key definitions, classical sieve principles, and fundamental concepts in modular arithmetic that underpin our approach.

1.2.1 Prime Numbers & The Fundamental Theorem of Arithmetic

A *prime number* p is a natural number greater than 1 with no integer divisors other than 1 and itself. Formally,

$$p \neq p_1 \cdot p_2 \quad \text{for any} \quad p_1, p_2 > 1 \in \mathbb{Z}.$$

This formulation emphasizes the indivisibility of primes. By the Fundamental Theorem of Arithmetic, every positive integer factors uniquely into primes, underscoring their role as the fundamental "building blocks" of the natural numbers.

Fermat's Little Theorem.

A cornerstone of number theory is Fermat's Little Theorem, which asserts that for any prime p and any integer a coprime with p,

$$a^p \equiv a \pmod{p}.$$

This result implies that p divides $a^p - a$ and forms the theoretical basis for many primality tests.

Modern primality tests build on Fermat's Little Theorem and other insights to efficiently determine primality in polynomial or near-polynomial time. The *AKS Primality Test* and the *Miller–Rabin Test* are two such examples, forming the backbone of large-scale prime-based applications in cryptography and beyond.

- **AKS Primality Test:** A deterministic polynomial-time algorithm for primality testing, developed by Manindra Agrawal, Neeraj Kayal, and Nitin Saxena in 2002.
- **Miller–Rabin Test:** A highly reliable probabilistic test developed by Miller and Rabin in the 1980s, widely used in cryptographic applications.

The Prime Set $\mathbb{Z}_{prime}$

The set of primes $\mathbb{Z}_{prime}$ starts with

$$\{2,\ 3,\ 5,\ 7,\ 11,\ 13,\ 17,\ \ldots\},$$

and extends infinitely—a fact famously proven by Euclid circa 300 BCE. The irregular spacing of these primes has long fascinated mathematicians. The Prime Number Theorem provides an asymptotic description of their distribution via the prime-counting function $\pi(n)$:

$$\pi(n) \sim \frac{n}{\log n}.$$

While there is no simple closed-form formula that generates all primes, sieve algorithms enable us to deterministically enumerate primes up to any given limit by methodically eliminating composites.

1.2.2 Prime Sieve: Core Concept

At its core, a prime sieve systematically removes composite numbers from the set of natural numbers:

$$\mathbb{Z}_{prime} \ = \ \mathbb{Z} \setminus \mathbb{Z}_{composite}, \tag{1.1}$$

where

- $\mathbb{Z}$ is the set of all numbers (Focusing on positive integers),
- $\mathbb{Z}_{composite}$ is the set of composite numbers.

The classical Sieve of Eratosthenes is one of the simplest and most elegant methods for achieving this:

Sieve of Eratosthenes

Attributed to Eratosthenes of Cyrene (circa 200 BCE), the algorithm operates as follows:

1. Begin with a list of integers up to n.
2. Let p initially be 2, the smallest prime number.
3. Mark all multiples of p within the range p^2 to n.
4. Move to the next unmarked number, which becomes the next prime p.
5. Repeat steps 3 and 4 until p^2 exceeds n.

6. All remaining unmarked numbers in the list are prime.

This sieve runs in $O(n \log \log n)$ time due to the summation of the reciprocals of primes, but its $O(n)$ space requirement limits its scalability for very large n.

1.2.3 Modular Arithmetic & Residue Classes

Modular arithmetic is a powerful tool for understanding divisibility and is central to many prime sieve algorithms. For a given integer n and modulus m, the residue of n modulo m is defined as

$$r \;=\; n \pmod m \;=\; n - m \cdot \left\lfloor \frac{n}{m} \right\rfloor .$$

This operation partitions the integers into m equivalence classes, each representing a distinct remainder when n divided by m, known as *residue classes*, denoted by $[r]_m$, where $r \in \{0, 1, 2, \ldots, m-1\}$. And we say n is congruent to r modulo m, denoted as $n \equiv r \pmod m$.

1.2.4 Residue Classes Modulo 6: Modular Space Reduction

In searching for primes, rather than considering all residues modulo 6, we need to leverage the fact that primes greater than 3 must satisfy

$$p \equiv \pm 1 \pmod 6.$$

Hence, beyond 2 and 3, only numbers of the form $6x \pm 1$ need to be checked for primality and marked as composites. This observation alone cuts down the candidates by two-thirds, an immediate 66% improvement.

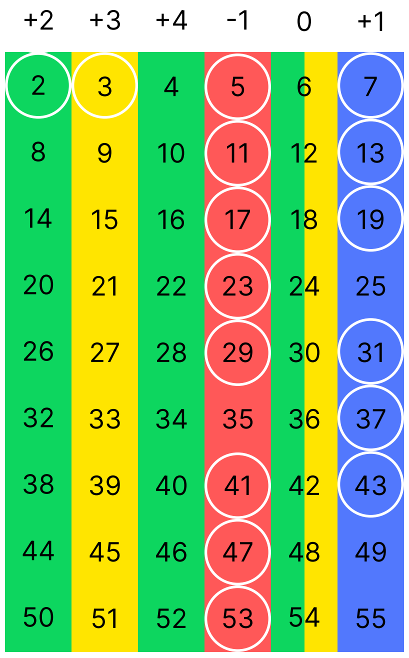

Figure 1.1: shows that beyond 2 and 3, all primes (highlighted with rings), lie in the residue classes $-1 \pmod 6$ (colored red), and $1 \pmod 6$ (colored blue).

Note 1.2.1. As illustrated in Figure 1.1, arranging numbers in rows of 6 reveals that multiples of 2 and 3 (and hence 6) align in predictable vertical patterns. This geometric insight is further exploited in later chapters with the introduction of the iZ-Matrix, which systematically applies more modular reductions to streamline the prime search.

1.3 X-Arrays: Computationally Efficient Representation

Directly operating on the numeric values of $6x \pm 1$, i,e.,

$$\{6x \pm 1 \mid x \in \mathbb{Z}^+\} = \{5, 7, 11, 13, 17, \ldots\},$$

poses two main challenges:

1. Storing these numbers requires a full integer representation (e.g., words of sizes 32-bit or 64-bit), which is not ideal in comparison to a bit per element representation.
2. The progression alternates between differences of 2 and 4, complicating the design of factorization wheels used for marking composites.

To address these issues, we partition the set by normalizing the arithmetic progression. This is achieved by classifying numbers based on their residues modulo 6 and indexing them by the variable x. Specifically, we define two arrays:

- `X5`: Represents numbers of the form $6x - 1$ via the index set $\{1, 2, 3, \dots\}$
- `X7`: Represents numbers of the form $6x + 1$ using the same index set $\{1, 2, 3, \dots\}$.

This normalization enables us to adopt a bit-level representation where each index corresponds to a single bit. The advantages of this approach are clear:

1. **Space Efficiency**: Each element is represented by just one bit rather than a integer word, significantly reducing memory footprint.
2. **Time Efficiency**: Bitwise operations are inherently fast, allowing rapid marking and checking of prime and composite states.

1.4 The *iZ-Theorem*

Having established that all primes greater than 3 reside in the form $6x \pm 1$, we now formalize this observation through the iZ Function. The iZ-Theorem characterizes how this function behaves under multiplication, thereby allowing us to map composite numbers precisely via their indices in the compact X-Arrays. We begin by introducing the key definitions and terminology that underpin the theorem.

1.4.1 Definitions & Terminology

Definition 1.4.1. Identity Set:

$$\mathbb{I} = \{\pm 1\} = \{-1, 1\}.$$

These two values encode the sign used by the identity parameter i in the iZ function; note that $\mathbb{I}$ is closed under multiplication and division.

Definition 1.4.2. iZ Function iZ_m is defined by

$$iZ_m(x,\, i) \;=\; m \cdot x \;+\; i \;=\; z,$$

where:

- $m \in \mathbb{Z}^+$ is the **modulus**, taken as a constant (defaulting to $m = 6$ if not specified).
- $x \in \mathbb{Z}^+$ is the **index** parameter, which maps to a pair of integers—one in each $i\mathbb{Z}$ subset.
- $i \in \mathbb{I}$ is the **identity** parameter, determining which subset the output z belongs to.
- **Output:** $z \in i\mathbb{Z}^i$ is a positive integer indivisible by any prime factor of the modulus m or the index x.

In our C implementation, the iZ Function is provided in both fixed-precision and arbitrary-precision versions:

- `uint64_t iZ(uint64_t x, int i)`: For 64-bit unsigned integers (suitable for numbers below 2^{64}).
- `void iZ_gmp(mpz_t z, mpz_t x, int i)`: Using the GMP library for arbitrary-precision arithmetic.

A simple Python implementation might be:

```python
def iZ(x: int, i: int) -> int:
    assert i in [-1, 1]  # i must be -1 or 1
    assert x > 0         # x must be positive
    return 6 * x + i     # computes the x-th number of the form i mod 6
```

Definition 1.4.3. The $i\mathbb{Z}$ Set is the union of two disjoint subsets generated by the iZ function:

$$i\mathbb{Z} = i\mathbb{Z}^- \cup i\mathbb{Z}^+, \quad \text{and } \emptyset = i\mathbb{Z}^- \cap i\mathbb{Z}^+,$$

where:

- $i\mathbb{Z}^- = \{iZ(x,-1) \mid x \in \mathbb{Z}^+\}$ corresponds to the numbers of the form $mx-1$,
- $i\mathbb{Z}^+ = \{iZ(x,1) \mid x \in \mathbb{Z}^+\}$ corresponds to the numbers of the form $mx+1$.

Properties of the $i\mathbb{Z}$ Set

Detailed proofs via the *iZ-Theorem* and its supporting lemmas establish that the $i\mathbb{Z}$ set has several important properties:

- **Density**: Exactly $\frac{2}{m}$ of the integers $\mathbb{Z}$ lie in $i\mathbb{Z}$. For $m=6$, this fraction is $\frac{1}{3}$.
$$\frac{|i\mathbb{Z}|}{|\mathbb{Z}|} = \frac{2}{m}$$
- **Exclusion of m's Factors**: No element in $i\mathbb{Z}$ is divisible by any prime factor of m by construction.
$$\gcd(mx \pm 1,\ m) = 1$$
- **Closure under Multiplication**: The $i\mathbb{Z}$ set is closed under multiplication. In other words, for any two elements
$$z_1 = iZ(x_1,\ i_1) \quad \text{and} \quad z_2 = iZ(x_2,\ i_2),$$
their product is also in $i\mathbb{Z}$:
$$iZ(x_1,\ i_1) \cdot iZ(x_2,\ i_2) = iZ(x,\ i),$$
where the resulting index x and identity i are determined by rules established in the theorem.
- **Recursive Indices under Exponentiation**: If $z = iZ(x,i)$, then its powers can be expressed as
$$z^k = iZ(x_k, i_k),$$
where the indices x_k and identities i_k follow predictable recursive patterns.

1.4.2 Statement of the *iZ-Theorem*

Lemma 1.4.1. *Let*
$$p = iZ(x_p,\ i_p) \quad \textit{and} \quad q = iZ(x_q,\ i_q),$$
be two numbers in $i\mathbb{Z}$. Then their product $z = pq$ also lies in $i\mathbb{Z}$, expressed as
$$z = iZ(x_z,\ i_z),$$
where the product index x_z could be expressed in terms of p or q as
$$x_z \ = \ p \cdot x_q \ + \ i_q \cdot x_p \ = \ q \cdot x_p \ + \ i_p \cdot x_q,$$
and the product identity i_z is determined by the identities of the factors p and q
$$i_z = i_p \cdot i_q \ \in \mathbb{I}.$$
From that, we derive this general equivalence
$$z \equiv 0 \pmod p \quad \iff \quad x_z \equiv i_q \cdot x_p \pmod p, \quad i_q = \frac{i_z}{i_p} \in \mathbb{I},$$
which shows the composites of a prime p within the $i\mathbb{Z}$ set have indices that fall into the residue classes $\pm x_p$ modulo p.

1.4.3 Proof of the *iZ-Theorem*

Construction & Expansion

We expand the product $(mx_p + i_p)(mx_q + i_q)$ and refactor, showing the resulting expression is still of the form $mx_z + i_z$, thereby confirming closure under multiplication.

Consider the product

$$z = pq = (m \cdot x_p + i_p)\,(m \cdot x_q + i_q).$$

Expanding the product, we obtain:

$$z = m^2 \cdot x_p \cdot x_q + m \cdot i_p \cdot x_q + m \cdot i_q \cdot x_p + i_p \cdot i_q.$$

Factoring out m where possible, we have:

$$z = m\big(m \cdot x_p \cdot x_q + i_p \cdot x_q + i_q \cdot x_p\big) \;+\; i_p \cdot i_q.$$

Since the identity set $\mathbb{I} = \{\pm 1\}$ is closed under multiplication and division, we can express z as

$$z = iZ(x_z,\, i_z),$$

where:

$$x_z = m\, x_p\, x_q + i_p\, x_q + i_q\, x_p \;\in\; \mathbb{Z}^+, \quad i_z = i_p \cdot i_q \;\in\; \mathbb{I}.$$

This confirms that the product of any two $i\mathbb{Z}$ numbers remains in $i\mathbb{Z}$, proving the closure under multiplication property.

Examples

- Let $p = iZ(2, 1) = 13$ and $q = iZ(3, -1) = 17$. Then

  $$z = pq = 13 \times 17 = 221.$$

  Here,

  $$x_z = 6 \times 2 \times 3 \,+\, 1 \times 3 \,+\, (-1) \times 2 \;=\; 37, \quad i_z = (1)\,(-1) \;=\; -1.$$

  Hence, $z = iZ(37,\, -1) = 6 \times 37 - 1 = 221$.

  In terms of a different modulus $m = 4$ (co-prime with p and q),

  we have $p = iZ_4(3, 1) = 13$, and $q = iZ_4(4, 1) = 17$, then:

  $$x_z = 4 \times 3 \times 4 \,+\, 1 \times 4 \,+\, 1 \times 3 \;=\; 55, \quad i_z = 1 \times 1 = 1.$$

  Hence, $z = iZ_4(55, 1) = 4 \times 55 + 1 = 221$.

- Let $p = iZ(5, 1) = 31$ and $q = iZ(10, 1) = 61$. Then

  $$z = pq = 31 \times 61 = 1891.$$

  Here,

  $$x_z = 6 \times 5 \times 10 \,+\, 1 \times 10 \,+\, 1 \times 5 \;=\; 315, \quad i_z = 1 \times 1 = 1.$$

  Hence, $z = iZ(315,\, 1) = 1891$.

The Product Identity i_z

Since $i_z = i_p \cdot i_q$, the identity of z follows from whether i_p and i_q match:

- If $i_p = i_q$, then $i_z = 1$ (since $1 \times 1 = 1$ or $-1 \times -1 = 1$), so $z \in i\mathbb{Z}^+$.
- If $i_p \neq i_q$, then $i_z = -1$ (since $1 \times (-1) = -1$), so $z \in i\mathbb{Z}^-$.

The Product Index x_z

Expressing the product index, we have:

$$x_z = m\,x_p\,x_q + i_p x_q + i_q x_p.$$

Rewriting, note that:

$$x_z = x_q(m\,x_p + i_p) + i_q\,x_p = p \cdot x_q + i_q\,x_p,$$

and similarly,

$$x_z = x_p(m\,x_q + i_q) + i_p\,x_q = q \cdot x_p + i_p\,x_q.$$

This shows that, modulo p, the index x_z satisfies:

$$x_z \equiv i_q \cdot x_p \pmod{p},$$

and, likewise, modulo q,

$$x_z \equiv i_p \cdot x_q \pmod{q}.$$

These equivalences provide the foundation for identifying the composite numbers within the $i\mathbb{Z}$ set based on the *Xp-Identity.*

We can confirm this from the examples above:

- In Example 1, where $p = 13$, $q = 17$, and $x_z = 37$, we have:

$$37 \equiv -2 \pmod{13}, \quad \text{and} \quad 37 \equiv 3 \pmod{17}.$$

- In Example 2, where $p = 31$, $q = 61$, and $x_z = 315$, we have:

$$315 \equiv 5 \pmod{31}, \quad \text{and} \quad 315 \equiv 10 \pmod{61}.$$

showing that the index x_z falls into the residue class of $i_q \cdot x_p$ modulo p.

This equivalence relationships is crucial for identifying the multiples of a given prime within $i\mathbb{Z}$ by their indices.

Completing the Proof

For a composite $z \in i\mathbb{Z}$ to be divisible by prime p, we have $z \equiv 0 \pmod{p}$. By construction and expansion, we established that in such cases, the index x_z falls into the residue class of $i_q \cdot x_p$ modulo p. Therefore, we conclude:

$$z \equiv 0 \pmod{p} \quad \Longleftrightarrow \quad x_z \equiv i_q \cdot x_p \pmod{p}, \quad \text{where } i_q = \frac{i_z}{i_p} \in \mathbb{I}.$$

This completes the proof of Lemma 1.4.1.

1.4.4 Wrapping Up

Multiplicative Properties

1. **Closure under multiplication**: The product z of two elements in $i\mathbb{Z}$ also belongs to $i\mathbb{Z}$.
2. **Product-Identity**: Determining which $i\mathbb{Z}$ subset the product belongs to depends on whether the identities of the factors match:

   Case 1: If $(i_p = i_q)$, the product resides in $i\mathbb{Z}^+$.

   Case 2: If $(i_p \neq i_q)$, it lands in $i\mathbb{Z}^-$.
3. **Product-Index**: The product index x_z could be directly computed as a function of p:

$$x_z = p \cdot x_q + i_q \cdot x_p,$$

   satisfying this modular equivalence for each factor p that divides z:

$$x_z \equiv \pm x_p \pmod{p}.$$

   This form is referred to as the **X*p*-Identity**, it implies the composition of z generally, and the divisibility by p specifically.

Divisibility Implications

The core equivalence from the iZ Theorem is:

$$z \equiv 0 \pmod{p} \quad \Longleftrightarrow \quad x_z \equiv \pm \cdot x_p \pmod{p}.$$

Depending on the parameters involved, this equivalence can be expanded into the following specific cases:

- For a composite $z \in i\mathbb{Z}^-$:

$$z \equiv 0 \pmod{p} \quad \Longrightarrow \begin{cases} x_z \equiv x_p \pmod{p} & \text{where } p \in i\mathbb{Z}^- \\ x_z \equiv -x_p \pmod{p} & \text{where } p \in i\mathbb{Z}^+ \end{cases}$$

- For a composite $z \in i\mathbb{Z}^+$:

$$z \equiv 0 \pmod{p} \quad \Longrightarrow \begin{cases} x_z \equiv x_p \pmod{p} & \text{where } p \in i\mathbb{Z}^+ \\ x_z \equiv -x_p \pmod{p} & \text{where } p \in i\mathbb{Z}^- \end{cases}$$

This symmetric, reflective behavior is the key insight that underlies the efficiency of the Xp-Wheel in marking composites.

Universality & Applications

The multiplicative structure established by the *iZ-Theorem* is independent of the specific modulus m ($m > 2$), underscoring the broad applicability of the framework. This universality enables a fine-grained analysis of composite relationships in various settings.

By translating multiplicative relationships into simple, index-based equivalences, the iZ-Theorem lays the groundwork for advanced sieving techniques. In subsequent sections, we will see how these relationships are harnessed by the Xp-Wheel to efficiently eliminate composite candidates—paving the way for efficient prime sieve algorithms.

Furthermore, the extension of the iZ-Theorem to define exponentiation rules further broadens its relevance in analytic number theory and cryptographic applications.

1.5 The Exponentiation of the iZ Function

Exponentiation plays a pivotal role in analytic number theory—it underpins both advanced primality tests and factorization techniques, and is critical to cryptographic protocols such as RSA and Diffie–Hellman. In this section, we explore how the iZ function evolves under exponentiation, revealing recursive relationships that express higher powers in the form

$$z^k = iZ(x_k, i_k),$$

where the sequences $\{x_k\}$ and $\{i_k\}$ follow predictable patterns.

1.5.1 Squaring the iZ Form

A key special case of the iZ-Theorem occurs when multiplying a number by itself. If we set $x_q = x_p$ and $i_q = i_p$ for a number $z = iZ(x_p, i_p)$, then:

$$z^2 = \big(iZ(x_p, i_p)\big)^2 = iZ\Big(p \cdot x_p + i_p \cdot x_p,\ i_p^2\Big).$$

Since $i_p^2 = 1$ (regardless of whether i_p is 1 or -1), and we denote $z = iZ(x,\ i)$, then:

$$z^2 = \begin{cases} iZ(z \cdot x - x,\ 1), & \text{if } z \in i\mathbb{Z}^- \\ iZ(z \cdot x + x,\ 1), & \text{if } z \in i\mathbb{Z}^+ \end{cases} \tag{1.2}$$

Key Observations:

1. **Identity Consistency**: Regardless of the initial identity i, the square $\big(m \cdot x + i\big)^2$ always has a positive identity ($i^2 = 1$). This holds true for all even powers since the product of identical identities is always 1.
2. **Index Evolution**: The index x_2 of the iZ form when squared is given by:

$$x_2 = z \cdot x + i \cdot x.$$

This shows that upon squaring, the index shifts linearly.

Examples:

- Let $z = iZ(17, -1) = 101$, where $x = 17$ and $i = -1$:

$$x_2 = 101 \cdot 17 - 17 = 1700, \quad z^2 = iZ(1700,\ 1) = 10201$$

- Let $z = iZ(17, 1) = 103$, where $x = 17$ and $i = 1$:

$$x_2 = 103 \cdot 17 + 17 = 1768, \quad z^2 = iZ(1768,\ 1) = 10609$$

To further illustrate the generality, consider the modulus $m = 4$. Applying the same rules to the above examples:

- For $z = iZ_4(25, 1) = 101$, we have

$$x_2 = 101 \cdot 25 - 25 = 2475, \quad z^2 = iZ_4(2475, 1) = 10201.$$

- For $z = iZ_4(26, -1) = 103$, we have

$$x_2 = 103 \cdot 26 + 26 = 2704, \quad z^2 = iZ_4(2704, 1) = 10609.$$

1.5.2 Higher Powers of the iZ Form

For higher powers $k > 2$, the iZ function obeys a recursive structure. Let $z = iZ(x, i)$. Then, we can express its kth power as:

$$z^k = iZ(x_k,\ i_k),$$

where:

$$x_k = z \cdot x_{k-1} + i^{k-1} \cdot x, \quad i_k = i^k. \tag{1.3}$$

Based on the initial identity i, two cases arise:

Case 1: For $i = 1$ (i.e, $z \in i\mathbb{Z}^+$)

The indices x_k evolve as follows:

- $x_2 = z \cdot x + x$
- $x_3 = z \cdot (z \cdot x + x) + x = z^2 \cdot x + z \cdot x + x$
- $x_4 = z \cdot (z \cdot (z \cdot x + x) + x) + x = z^3 \cdot x + z^2 \cdot x + z \cdot x + x$
- $x_5 = z \cdot (z \cdot (z \cdot (z \cdot x + x) + x) + x) + x = z^4 \cdot x + z^3 \cdot x + z^2 \cdot x + z \cdot x + x$
- $\cdots$

Example:

Let $z = iZ(2,\ 1) = 13$, we compute the index x_k as follows:

- $x_2 = 13 \cdot 2 + 2 = 28, \quad z^2 = iZ(28, 1) = 169$
- $x_3 = 13 \cdot 28 + 2 = 366, \quad z^3 = iZ(366, 1) = 2197$
- $x_4 = 13 \cdot 366 + 2 = 4760, \quad z^4 = iZ(4760, 1) = 28561$
- $x_5 = 13 \cdot 4760 + 2 = 61882, \quad z^5 = iZ(61882, 1) = 371293$
- $\cdots$

Case 2: For $i = -1$ (i.e, $z \in i\mathbb{Z}^-$)

Here the indices x_k alternate in sign:

- $x_2 = z \cdot x - x$
- $x_3 = z \cdot (z \cdot x - x) + x = z^2 \cdot x - z \cdot x + x$
- $x_4 = z \cdot (z \cdot (z \cdot x - x) + x) - x = z^3 \cdot x - z^2 \cdot x + z \cdot x - x$
- $x_5 = z \cdot (z \cdot (z \cdot (z \cdot x - x) + x) - x) + x = z^4 \cdot x - z^3 \cdot x + z^2 \cdot x - z \cdot x + x$
- $\cdots$

Example:

Let $p = iZ(2,\ -1) = 11$, we compute the index x_k as follows:

- $x_2 = 11 \cdot 2 - 2 = 20, \quad z^2 = iZ(20, 1) = 121$
- $x_3 = 11 \cdot 20 + 2 = 222, \quad z^3 = iZ(222, -1) = 1331$
- $x_4 = 11 \cdot 222 - 2 = 2440, \quad z^4 = iZ(2440, 1) = 14641$
- $x_5 = 11 \cdot 2440 + 2 = 26842, \quad z^5 = iZ(26842, -1) = 161051$
- $\cdots$

1.5.3 Recursive Expansion

To better understand the structure, x_k can be expanded explicitly as following:

- **Case 1:** For $z \in i\mathbb{Z}^+$, all the terms are consistently positive:

$$x_k = z^{k-1} \cdot x + z^{k-2} \cdot x + \cdots + z^{k-k} \cdot x.$$

- **Case 2:** For $z \in i\mathbb{Z}^-$, the signs alternate between positive and negative:

$$x_k = z^{k-1} \cdot x - z^{k-2} \cdot x + \cdots \pm z^{k-k} \cdot x.$$

In Summary:

- These recursive expansions illustrate the structured growth of the indices x_k as a function of z and k.
- In $i\mathbb{Z}^+$, terms always accumulate; in $i\mathbb{Z}^-$, they alternate, reflecting the underlying identity cycle.

1.5.4 Conclusions

Ultimately, the iZ Theorem's versatility goes beyond $m = 6$, which is advantageous in the context of prime sieve, offering a robust framework for analyzing an integer z in terms of its co-primes m for which ($z \equiv \pm 1 \pmod{m}$).

In the next section, we transition to the practical realm of prime sieving by introducing the **X*p*-Wheel**, a powerful tool that leverages these recursive and modular properties to pinpoint composite indices efficiently within the $i\mathbb{Z}$ set.

1.6 The X*p*-Wheel

Leveraging the modular equivalences established by the iZ Theorem, the *Xp-Wheel* provides a streamlined mechanism for mapping composite numbers within $i\mathbb{Z}$, isolating the primes. This section defines the rules of the X$p$wheel, discusses its optimization, and then compares its performance against the traditional Sieve of Eratosthenes.

1.6.1 Modular Rules

Recalling the core equivalence from the *iZ-Theorem*:

$$z \equiv 0 \pmod{p} \quad \Longleftrightarrow \quad x_z \equiv i_q \cdot x_p \pmod{p}, \quad i_q = \frac{i_z}{i_p} \in \mathbb{I}.$$

The Xp-Wheel translates these modular conditions into explicit index-generation functions. Depending on the identity of p and the targeted $i\mathbb{Z}^{\pm}$ subset, the Xp-Wheel operates on the simplified X-Arrays:

- `X5`, corresponding to $6x - 1$ (the $i\mathbb{Z}^-$ subset),
- `X7`, corresponding to $6x + 1$ (the $i\mathbb{Z}^+$ subset),

as follows:

Case 1: Prime $p \in i\mathbb{Z}^-$

- $i\mathbb{Z}^-$ Composites satisfy: $x_z \equiv x_p \pmod{p}$.

  $$x_z \in \{p \cdot x_q + x_p \mid x_q \in \mathbb{Z}^+\}$$

  This Xp-Form maps the indices of the modular structure $p \cdot i\mathbb{Z}^+$ in `X5`.

- $i\mathbb{Z}^+$ Composites satisfy: $x_z \equiv -x_p \pmod{p}$.

  $$x_z \in \{p \cdot x_q - x_p \mid x_q \in \mathbb{Z}^+\}$$

  This Xp-Form maps the indices of the modular structure $p \cdot i\mathbb{Z}^-$ in `X7`.

Case 2: Prime $p \in i\mathbb{Z}^+$:

- $i\mathbb{Z}^-$ Composites satisfy: $x_z \equiv x_p \pmod{p}$.

  $$x_z \in \{p \cdot x_q - x_p \mid x_q \in \mathbb{Z}^+\}$$

  This Xp-Form maps the indices of the modular structure $p \cdot i\mathbb{Z}^-$ in `X5`.

- $i\mathbb{Z}^+$ Composites satisfy: $x_z \equiv -x_p \pmod{p}$.

  $$x_z \in \{p \cdot x_q + x_p \mid x_q \in \mathbb{Z}^+\}$$

  This Xp-Form maps the indices of the modular structure $p \cdot i\mathbb{Z}^+$ in `X7`.

By marking all indices x_z corresponding to the composites of iZ root primes $p \leq \sqrt{n}$, the unmarked indices in the X-Arrays map to the primes in $i\mathbb{Z}$ up to n.

1.6.2 Optimizing the Starting Point of the Xp-Wheel

In practical sieving, primes are processed sequentially from smallest to largest, thus each composite is typically marked by its smallest factor first. Therefore, once a prime p is identified, one can start marking new composites from

$$x_q \;\geq\; x_p,$$

rather than the naive $x_q \geq 1$. This avoids redundancy because indices with $x_q < x_p$ have already been covered by smaller primes.

This optimization is particularly impactful for large primes. For example, if $p = iZ(10^9, 1) = 6000000001$, then we skip all $x_q < 10^9$, thereby saving 10^9 unneeded "mark composite" operations.

1.6.3 Applying the Xp-Wheel: A Practical Example

To illustrate the Xp-Wheel in action, let us compare it with the traditional Sieve of Eratosthenes for $n = 169$. We identify the root primes up to $\sqrt{169} = 13$, namely {2,3,5,7,11,13}.

Sieve of Eratosthenes

We maintain a boolean array of size $n + 1 = 170$. For each root prime p, we mark all multiples from p^2 up to n. The composites are:

- 2: {4, 6, 8, 10, 12, 14, . . . , 168} (total: 83)
- 3: {9, 12, 15, 18, 21, . . . , 168} (total: 54)
- 5: {25, 30, 35, 40, 45, . . . , 165} (total: 29)
- 7: {49, 56, 63, 70, 77, . . . , 168} (total: 18)
- 11: {121, 132, 143, 154} (total: 4)
- 13: {169} (total: 1)

The total number of "mark composite" operations is $83 + 54 + 29 + 18 + 4 + 1 = 189$.

After these markings, the unmarked positions in the boolean array correspond to the primes up to 169, which are:

{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167}

Total count of primes up to $n = 169$ is 39 primes.

Sieve with the Xp-Wheel

For the iZ-based approach:

- `X5` of size $x_n = n/6 + 1 = 29$ bits ranging from 0 to 28, representing integers $6x - 1$ up to n.
- `X7` of size x_n as well, representing $6x + 1$.

Hence, we work with a total of 58 bits instead of 170 bits.

Marking Composites of Root Primes Applying the optimized Xp-Wheel for each root prime:

$$x_z \in \{p \cdot x_q \pm x_p \mid x_p \leq x_q \leq x_n\}$$

- 2, 3: Have no multiples in the form $6x \pm 1$, no composites are marked.
- 5:

  `X5`: $x_z \in \{5x_q + 1 \mid 1 \leq x_q \leq 28\} = \{6, 11, 16, 21, 26\}$ (total: 5)

  `X7`: $x_z \in \{5x_q - 1 \mid 1 \leq x_q \leq 28\} = \{4, 9, 14, 19, 24\}$ (total: 5)

- 7:

  `X5`: $x_z \in \{7x_q - 1 \mid 1 \leq x_q \leq 28\} = \{6, 13, 20, 27\}$ (total: 4)

  `X7`: $x_z \in \{7x_q + 1 \mid 1 \leq x_q \leq 28\} = \{8, 15, 22\}$ (total: 3)

- 11:

  `X5`: $x_z \in \{11x_q + 2 \mid 2 \leq x_q \leq 28\} = \{24\}$ (total: 1)

  `X7`: $x_z \in \{11x_q - 2 \mid 2 \leq x_q \leq 28\} = \{20\}$ (total: 1)

- 13:

  `X5`: $x_z \in \{13x_q - 2 \mid 2 \leq x_q \leq 28\} = \{24\}$ (total: 1)

  `X7`: $x_z \in \{13x_q + 2 \mid 2 \leq x_q \leq 28\} = \{28\}$ (total: 1)

Summing these, the total "mark composite" operations is $5 + 5 + 4 + 3 + 1 + 1 = 19$—significantly fewer than the Eratosthenes count (189).

In absolute terms, 19 vs. 189 composite-marking steps is already a striking gap for $n = 169$. As n grows, this disparity becomes even more impactful, suggesting major efficiency gains for large prime-sieving tasks.

Extracting the Primes Marking the generates indices from the Xp-Wheel for each root prime in X5 and X7:

- `X5`: {0, 1, 2, 3, 4, 5, ~~6~~, 7, 8, 9, 10, ~~11~~, 12, ~~13~~, 14, 15, ~~16~~, 17, 18, 19, ~~20~~, ~~21~~, 22, 23, ~~24~~, 25, ~~26~~, ~~27~~, 28}.

  Constructing the $6x - 1$ values of the unmarked indices greater than 0 gives:

  {5, 11, 17, 23, 29, 41, 47, 53, 59, 71, 83, 89, 101, 107, 113, 131, 137, 149, 167}.

  Total count of primes in $i\mathbb{Z}^-$ up to 169 is 19 primes.

- `X7`: {0, 1, 2, 3, ~~4~~, 5, 6, 7, ~~8~~, ~~9~~, 10, 11, 12, 13, ~~14~~, ~~15~~, 16, 17, 18, ~~19~~, ~~20~~, 21, ~~22~~, 23, ~~24~~, 25, 26, 27, ~~28~~}

  Constructing the $6x + 1$ values of the unmarked indices gives:

  {7, 13, 19, 31, 37, 43, 61, 67, 73, 79, 97, 103, 109, 127, 139, 151, 157, 163}.

  Total count of primes in $i\mathbb{Z}^+$ up to 169 is 18 primes.

Total count of iZ primes up to $n = 169$ is 37 primes. Thus, the total primes after adding 2 and 3 is 39, consistent with the results of the Sieve of Eratosthenes.

1.6.4 Significance

By restricting attention to $i\mathbb{Z}$—namely, integers of the form $6x \pm 1$—we automatically skip multiples of 2 and 3, cutting the search space to $O(\frac{n}{3})$. Furthermore, the index-based equivalences which the optimized Xp-Wheel follows, allow us to mark composites in a targeted way, resulting in only

$$\sum_{5 \leq p \leq \sqrt{n}} \frac{2(x_n - p \cdot x_p)}{p}, \quad \text{instead of the traditional} \quad \sum_{2 \leq p \leq \sqrt{n}} \frac{n - p^2}{p}$$

marking operations. As illustrated by our example (19 vs. 189 markings), the Xp-Wheel can provide notable performance gains, underscoring its value as a fine-tuned and optimized approach to prime sieving beyond the classical Sieve of Eratosthenes.

1.7 Summary of Chapter 1

In this chapter, we established the theoretical foundation for the **iZ-Framework**. We began by revisiting classical definitions and methods, such as the Fundamental Theorem of Arithmetic and traditional sieve techniques, to remind the reader of the established landscape. This foundation naturally led us to the crucial observation that, aside from the primes 2 and 3, every prime number falls into one of the two residue classes $\pm 1 \pmod 6$. This underexploited insight in traditional sieves immediately reduces the candidate set by two-thirds, as only numbers of the form $6x \pm 1$ need to be examined.

Focusing on the remaining candidates (the $i\mathbb{Z}$ set), we partitioned these numbers into two subsets, $i\mathbb{Z}^-$ and $i\mathbb{Z}^+$, according to their residue class modulo m. We then introduced the **X-Arrays** (namely, `X5` and `X7`), a compact, computational efficient representation that maps the $i\mathbb{Z}$ subsets into bit-level arrays. This structure enables rapid bitwise operations, thereby optimizing both space and time efficiency.

To formalize this approach, we defined the **iZ Function**, which uniquely associates each element of $i\mathbb{Z}$ with an index in the X-Arrays. From there, we presented the **iZ-Theorem**, which rigorously characterizes the multiplicative behavior of the iZ function, and establishes key properties of the $i\mathbb{Z}$ set. By analyzing how indices evolve under multiplication, the theorem yields precise, index-based criteria—captured in what we term the Xp-Identity—for identifying composites, along with a recursive structure for higher powers.

Finally, we introduced the **X*p*-Wheel**, a specialized tool that leverages the index-based equivalences established by the iZ-Theorem to pinpoint composite numbers efficiently within the $i\mathbb{Z}$ set. This mechanism significantly reduces the marking overhead compared to traditional methods like the Sieve of Eratosthenes, as demonstrated by the dramatic reduction in composite-marking operations in our illustrative example.

In the next chapter, we transition from theory to practice by detailing the Classic Sieve-iZ algorithm, presenting its design and implementation, and demonstrating how these concepts are brought to life to achieve practical, high-performance prime sieving.

Chapter 2

The Classic Sieve-iZ Algorithm

2.1 Introduction

This chapter presents the *Classic Sieve-iZ Algorithm*, including its implementation details, benchmarking, and performance analysis.

In this introduction, we briefly review the theoretical basis from Chapter 1 and introduce the key data structure modules required for a clean C-language implementation.

2.1.1 Theoretical Basis

The Classic Sieve-iZ algorithm builds upon the following concepts from Chapter 1:

1. The $i\mathbb{Z}$ set, A Reduced Search Space for Primes:

Excluding multiples of 2 and 3 shrinks the search space from $O(n)$ to $O\left(\frac{n}{3}\right)$, yielding a 66% improvement over traditional sieves.

2. The X-Arrays, A Compact & Efficient Representation:

The $i\mathbb{Z}$ set is partitioned and structured into "X-Arrays" for efficient, compact bit-level representation:

- `X5`: Represents $i\mathbb{Z}^-$ (i.e., the $6x - 1$ integers),
- `X7`: Represents $i\mathbb{Z}^+$ (i.e, the $6x + 1$ integers).

Each array is $\lfloor \frac{n}{6} \rfloor + 1$ bits long, so collectively they require about $\frac{n}{3}$ bits. In the X-Arrays, the bit-index x maps to the integer $6x \pm 1$, and the bit-state (i.e., 0 or 1) is for storing primality state.

3. The Xp-Wheel:

Building on the iZ-Function's multiplicative behavior (established by iZ-Theorem 1.4.1), the *Optimized Xp-Wheel* prescribes how to mark composites of a prime p in the X-Arrays. Specifically:

$$\text{If} \quad p \in i\mathbb{Z}_- : \begin{cases} \texttt{X5} & : x_z \in \{\ p x_q + x_p\ \} \\ \texttt{X7} & : x_z \in \{\ p x_q - x_p\ \} \end{cases} \quad \text{where} \quad x_q \in (x_p : x_n),$$

$$\text{If} \quad p \in i\mathbb{Z}_+ : \begin{cases} \texttt{X5} & : x_z \in \{\ p x_q - x_p\ \} \\ \texttt{X7} & : x_z \in \{\ p x_q + x_p\ \} \end{cases} \quad \text{where} \quad x_q \in (x_p : x_n).$$

Next, we introduce the modules used to implement these ideas efficiently in C.

2.1.2 Data Structure Modules

Two primary modules, described in the Appendices, enable clean and modular implementations:

- `BITMAP` **Module**: Manages a dynamic array of bits (`BITMAP` structure) for memory allocation, bitwise operations (read, write), IO operations (read, write), and data validation. (See Appendix A.2.1)

- `PRIMES_OBJ` **Module**: Facilitates operations on prime arrays (`PRIMES_OBJ` structure), including allocation, insertion, resizing, IO operations (read, write), and validation. (See Appendix A.2.2)

These modules abstract away boilerplate tasks so we can focus on algorithmic logic.

2.2 Algorithmic Description & Implementation

In this section, we describe and implement the *Classic Sieve-iZ Algorithm* step by step.

2.2.1 Description

The Classic Sieve-iZ Algorithm generates the set of prime numbers up to a given limit n using the Xp-Wheel to mark composites in the $i\mathbb{Z}$ set. It operates following these steps:

1. Initialization:

- Initialize a `primes` array and insert the primes 2 and 3, since neither fits the $6x \pm 1$ form.
- Calculate $x_n = \lfloor \frac{n}{6} \rfloor + 1$ as the maximum x-index up to n.
- Create two bitmaps, `x5` and `x7`, each of length x_n. Set all bits 1 as "potentially prime".
- Calculate $n\_sqrt = \sqrt{n} + 1$ as the upper bound for root primes ($p \leq \sqrt{n}$).

2. Sieve Logic:

- For x in range $(1 : x_n)$:
  - If `x5`$[x]$ remains 1 (unmarked), then:
    - Append $z = iZ(x, -1)$ to `primes`.
    - If $z < n\_sqrt$: Mark its iZ composites in the X-Arrays using the Xp-Wheel.
      - `x5`$[z \cdot x + x : x_n : z] = 0$ (as composites of z in $i\mathbb{Z}^-$).
      - `x7`$[z \cdot x - x : x_n : z] = 0$ (as composites of z in $i\mathbb{Z}^+$).
  - If `x7`$[x]$ is 1, do the same with inverting the signs.
    - Append $z = iZ(x, 1)$ to `primes`.
    - If $z < n\_sqrt$: Mark its iZ composites.
      - `x5`$[z \cdot x - x : x_n : z] = 0$.
      - `x7`$[z \cdot x + x : x_n : z] = 0$.

3. Return Results:

- Return the *primes* array.

Note 2.2.1. The array slicing notation `x`$[a : b : s]$ denotes the range from a to b with step s in the `x` array. In other words:

```
for i in range(a, b):
    x[i] = 0
    i += s
```

2.2.2 Implementation

Translating the algorithmic description into code, below is the implementation of the Classic Sieve-iZ Algorithm in C language:

Listing 2.1: /src/sieve/sieve_iZ.c

```c
/**
 * @brief Generates a list of prime numbers up to a given limit using the
 * Xp Wheel to mark composites in the iZ set, the set of (6x +/- 1) numbers.
 *
 * @param n The upper limit for generating prime numbers.
 * @return A pointer to a PRIMES_OBJ structure containing the list of
 * primes up to n, or NULL if memory allocation fails.
 */
PRIMES_OBJ *sieve_iZ(uint64_t n)
{
    // Initialize primes object with approximate capacity
    PRIMES_OBJ *primes = primes_obj_init(n * .1 + 1000);

    // Memory allocation failed, check logs
    if (primes == NULL)
        return NULL;

    // Add 2, 3 to the primes object
    primes_obj_append(primes, 2);
    primes_obj_append(primes, 3);

    // Compute x_n, max x value < n
    uint64_t x_n = (n + 1) / 6 + 1;

    // Create bitmap X-Arrays x5, x7
    BITMAP *x5 = bitmap_create(x_n + 1);
    BITMAP *x7 = bitmap_create(x_n + 1);
    // Total size n/3 bits

    // Mark all bits as potential primes
    bitmap_set_all(x5);
    bitmap_set_all(x7);

    // Compute sqrt(n) as the upper bound for root primes
    uint64_t n_sqrt = sqrt(n) + 1;

    // Main loop ~ n/3 "is-prime" checks
    for (uint64_t x = 1; x < x_n; x++)
    {
        // if x5[x], implying it's iZ- prime
        if (bitmap_get_bit(x5, x))
        {
            uint64_t z = iZ(x, -1);
            primes_obj_append(primes, z); // add z to primes, increment p_count

            // if z is root prime, mark its multiples in x5, x7
            if (z < n_sqrt)
            {
                bitmap_clear_mod_p(x5, z, z * x + x, x_n);
                bitmap_clear_mod_p(x7, z, z * x - x, x_n);
            }
        }

        // Do the same if x7[x], inverting the signs
        if (bitmap_get_bit(x7, x))
        {
            uint64_t z = iZ(x, 1);
            primes_obj_append(primes, z);

            if (z < n_sqrt)
            {
                bitmap_clear_mod_p(x5, z, z * x - x, x_n);
                bitmap_clear_mod_p(x7, z, z * x + x, x_n);
            }
        }
    }

    // Cleanup: free memory of x5, x7
    bitmap_free(x5);
    bitmap_free(x7);

    // Handle edge case: if last prime > n, remove it
    if (primes->p_array[primes->p_count - 1] > n)
        primes->p_count--;

    // Trim unused memory in primes object
    if (primes_obj_resize_to_p_count(primes) != 0)
    {
        // Error already logged in primes_obj_resize
        primes_obj_free(primes);
        return NULL;
    }

    return primes;
}
```

2.3 Benchmarks

2.3.1 Sieve Algorithms in Comparison

We compare the **Classic Sieve-iZ** with:

- Sieve of Eratosthenes 'Optimized'
- Sieve of Euler
- Sieve of Atkin
- Segmented Sieve

All algorithms were implemented using the same data structures and modules, and are faithfully well optimized, ensuring a fair comparison. The exact implementations can be found in the Appendix A.3.

2.3.2 Execution Time Benchmarking

We use the `benchmark_sieve` (Appendix A.5) to measure each algorithm's runtime for various limits n. For instance, to benchmark from 10^4 to 10^9:

```c
// main.c
int main(void)
{
    SieveAlgorithm sieve_algorithms[] = {
        SieveOfEratosthenes,
        SieveOfEuler,
        SieveOfAtkin,
        Sieve_iZ,
        SegmentedSieve,
    };

    int models_count = sizeof(sieve_algorithms) / sizeof(SieveAlgorithm);
    SieveModels sieve_models = {sieve_algorithms, models_count};

    // benchmarking sieve algorithms
    benchmark_sieve(sieve_models, 10, 4, 9, 1); // testing from 10^4 to 10^9

    return 0;
}
```

Use the `make` command to compile and run *main.c*, then view results under `output/stdout.txt`, which contains the execution time for each algorithm at various limits, and the filename of the saved results in `output/` for visualization.

2.3.3 Test Results

Table 2.1 shows the runtime in microseconds (μ) for each algorithm at selected input scales.

Algorithm Name	10^4	10^5	10^6	10^7	10^8	10^9
Sieve of Eratosthenes	29	191	1746	16425	139982	2659127
Sieve of Euler	46	196	1559	14597	149964	1714330
Sieve of Atkin	52	311	2879	29412	319568	8088182
Sieve-iZ	46	134	1041	9552	93103	1457370
Segmented Sieve	57	380	3434	32300	312456	3083335

Table 2.1: Execution Time Benchmarking for Various Limits

Figure 2.1 visualizes *normalized* execution time ($(\mu/n) \times 1000$) for each algorithm across the range ($10^4 : 10^9$).

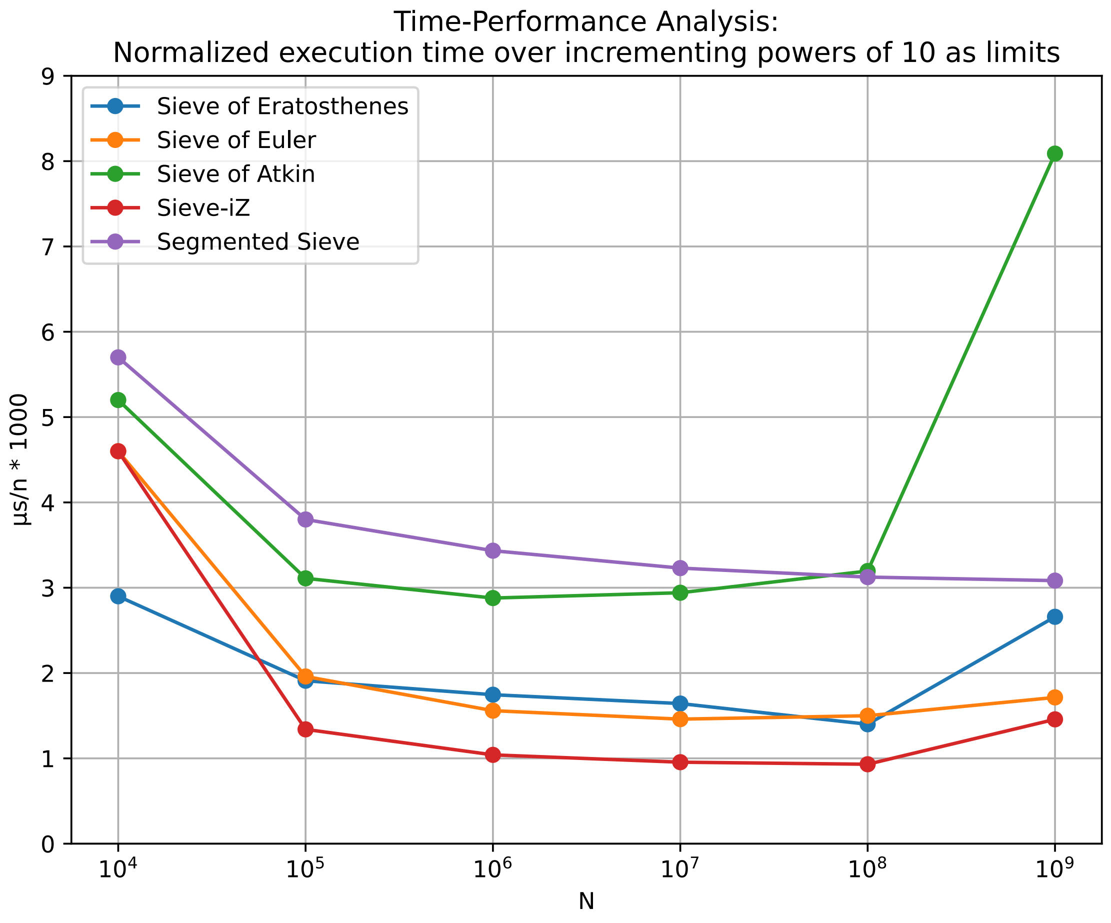

Figure 2.1: Time-per-input analysis for N in range ($10^4 : 10^9$)

While the exact results in microseconds depends on the computational model, the relative performance trends should remain consistent.

2.4 Performance Analysis

2.4.1 Time Complexity

Benchmark results show that the Classic Sieve-iZ consistently outperforms the other algorithms in terms of raw execution speed. Two primary factors account for this advantage:

1. Reduced Search Space

By operating exclusively on the integers of the form $6x \pm 1$, the algorithm avoids marking multiples of 2 and 3 entirely. In a naive setting, those multiples would require

$$\tfrac{n}{2} + \tfrac{n}{3} = \tfrac{5n}{6}$$

"mark-composite" operations. Skipping them thus confers a substantial efficiency boost. While all the algorithms are optimized to avoid checking the even numbers or marking them, the Classic Sieve-iZ goes a step further by bypassing multiples of 3 as well.

2. Xp-Wheel Efficiency

Composite marking proceeds via the Xp-Wheel, requiring about

$$\sum_{5 \le p \le \sqrt{n}} \frac{2\big(x_n - p \cdot x_p\big)}{p}$$

"mark-composite" operations. Asymptotically, this sum behaves like $O(n \log \log n)$, which doesn't reflect the applied speedups. The efficiency seen in the benchmarks, gained by the reduced search space and the selective marking of composites often performing less than n operations up to moderate scales.

2.4.2 Space Complexity

All the sieves in the comparison exhibit linear space complexity $O(n)$ with a couple of key exceptions merit attention:

Classic Sieve-iZ

Its two bit-arrays (`x5` and `x7`) occupy about $\frac{n}{3}$ bits total, giving $O\big(\frac{n}{3}\big)$ space—still linear, yet with a $3x$ better constant factor than traditional sieves, which makes it $3x$ better at scaling than traditional $O(n)$ sieves.

Segmented Sieve

Requiring $O\big(\sqrt{n}\big)$ space by processing consecutive segments, it circumvents the need to store information for all numbers up to n.

As n grows large (particularly at $n = 10^9$), all linear-space sieves experience performance decline due to higher memory usage and more frequent cache misses. Although the Segmented Sieve can be slower initially—owing to overhead for segment management—it delivers a stable performance for much higher limits due to its sub-linear memory requirements, ensuring locality of reference up to much higher limits than linear approaches.

2.5 Conclusions

Empirical benchmarks up to 10^9 confirm that *Classic Sieve-iZ* is highly efficient for quick prime generation up to moderate limits. It can be featured as:

Energy Efficient

By minimizing redundant markings, the algorithm reduces the total operations needed, thereby conserving energy and delivering faster execution times.

Simple Design

The Classic Sieve-iZ doesn't compromise on simplicity. Relying on the clear identity-based structure of $i\mathbb{Z}$ domain, it is easy to implement and understand, making it the natural extension of the Classical Sieve of Eratosthenes.

2.5.1 Limitations & Future Directions

Addressing the challenges that limit the scalability of the Classic Sieve-iZ, we identify the following areas for improvement:

Segmentation

While simple and efficient enough at small scales, the linear space complexity $O(\frac{n}{3})$ quickly becomes a bottleneck for larger limits, necessitating a more scalable approach. To overcome this, we have to divide large ranges into smaller, more manageable chunks that can fit into memory.

Skipping More Prime

While omitting of 2 and 3 is a major leap (saving $\frac{5n}{6}$ marking operations in naive settings), further optimization (i.e., skipping the due primes 5, 7, etc.) could further reduce the computational cost, especially for very large limits.

In a subsequent chapter, we introduce the *Segmented Sieve-iZm Algorithm*, a more advanced prime sieve that addresses these limitations and improves on the Classic Sieve-iZ Algorithm. But before that, we will delve into its theoretical underpinnings, introducing the **iZ-Matrix** structure.

Chapter 3

The iZ-Matrix Structure

3.1 Introduction

Modern prime-sieving algorithms face increasing demands for efficiency, especially as we scale up to handle ever-larger ranges in cryptography and number-theoretic research. In earlier chapters, we leveraged the fact that all primes above 3 lie in $\pm 1 \pmod 6$ to shrink the search space and reduce marking overhead. However, even these gains can be pushed further by adopting a more structured approach for composite detection, reducing the sieve's overhead and accelerating prime generation.

This chapter introduces the *iZ-Matrix (iZm)*—a two-dimensional representation of the bit arrays `X5` and `X7`. Mapping the sets $i\mathbb{Z}^-$ and $i\mathbb{Z}^+$ to a grid reveals geometric properties that streamline the sieve process and support advanced features like row-wise and column-wise scanning.

The Roadmap, for this chapter includes:

1. **Matrix Definition & Visualization**: We define *iZm5* and *iZm7* as two grids of size $(vx \times vy)$, highlighting how lattice structures emerge from the modular properties of the $i\mathbb{Z}$ set.
2. **Segment Construction**: We show how to build a pre-sieved "base" segment of size vx by multiplying small primes (starting from 5 and 7) and marking their composites only once.
3. ***VX6* Optimization**: By capping the prime factors at 19, we create a segment width 1,616,615 bits (≈ 0.2 MB) that comfortably fits into L2 cache, yielding significant speedups for large-scale sieving.
4. **Analytical Tools & Statistical Insights**: We illustrate how twin, cousin, and "sexy" prime pairs appear in these grids, offering an additional perspective on prime patterns restricted by each factor in vx.
5. **Algorithmic Solutions**: Finally, we present methods for finding the first composite of a prime p in any chosen row or column (**horizontal** vs. **vertical** sieve modes), leveraging simple modular equations to jump directly to the relevant composite indices.

Together, these ideas lay the groundwork for the upcoming algorithms—*Segmented Sieve-iZm*, *VX6-Sieve* and *Random-iZprime*— discussed in subsequent chapters, where the iZ-Matrix becomes the driving framework for fast, cache-friendly prime sieving over massive ranges and for generating random primes of large bit lengths.

3.2 The iZ-Matrix: Definition, Visualization & Properties

The iZ-Matrix (iZm) provides a structured, multi-dimensional representation of the X-Arrays `X5` and `X7`. By arranging these arrays into 2D grids, we can leverage the geometric patterns that emerge from modular arithmetic to streamline segmentation and enable more efficient prime-related computations.

3.2.1 Definition

To model the $i\mathbb{Z}$ subsets and their simplified X-Arrays, we define two matrices, each of size $(vx \times vy)$:

- **iZm5**: Represents $i\mathbb{Z}^-$, where a number z at coordinates $[x,\ y]$ is calculated as:

$$z = iZ(x + vx \times y,\ -1).$$

- **iZm7**: Represents $i\mathbb{Z}^+$, z at $[x,\ y]$ is calculated as:

$$z = iZ(x + vx \times y,\ 1).$$

Here, $x \leq vx \in \mathbb{Z}^+$, and $y \in \mathbb{N}$.

3.2.2 Visualization

For an intuitive understanding of the iZ-Matrix and its potential applications, consider Figures 3.1 and 3.2. These examples show grids of width $vx = 35$, where each prime in this list $\{5,\ 7,\ 11,\ 13,\ 17,\ 19\}$ uses a distinct color to highlight the cells it divides.

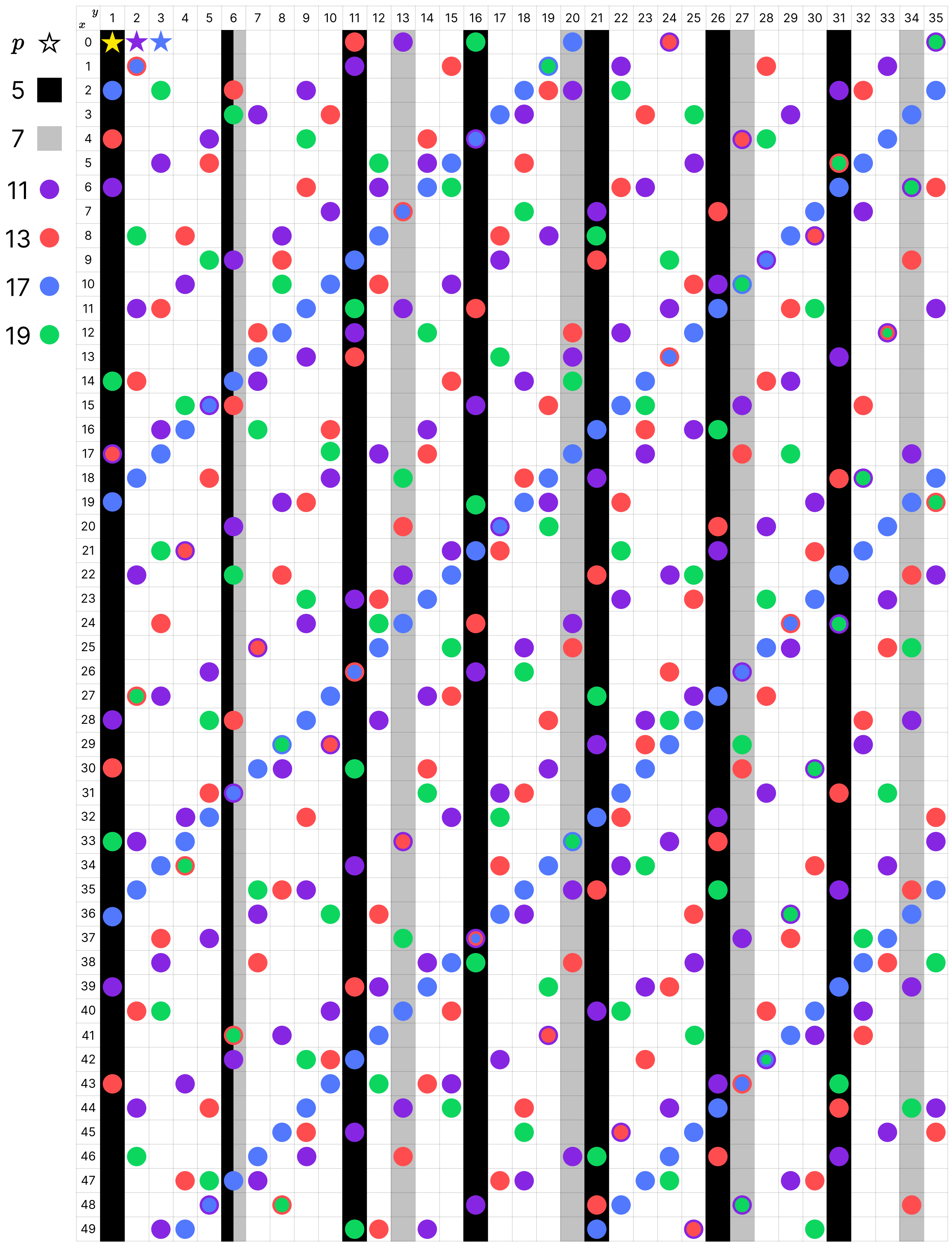

Figure 3.1: **iZm5** example (size 35 × 50). Each prime in {5, 7, 11, 13, 17, 19} marks its composite cells with a distinct color, defined at the top-left corner.

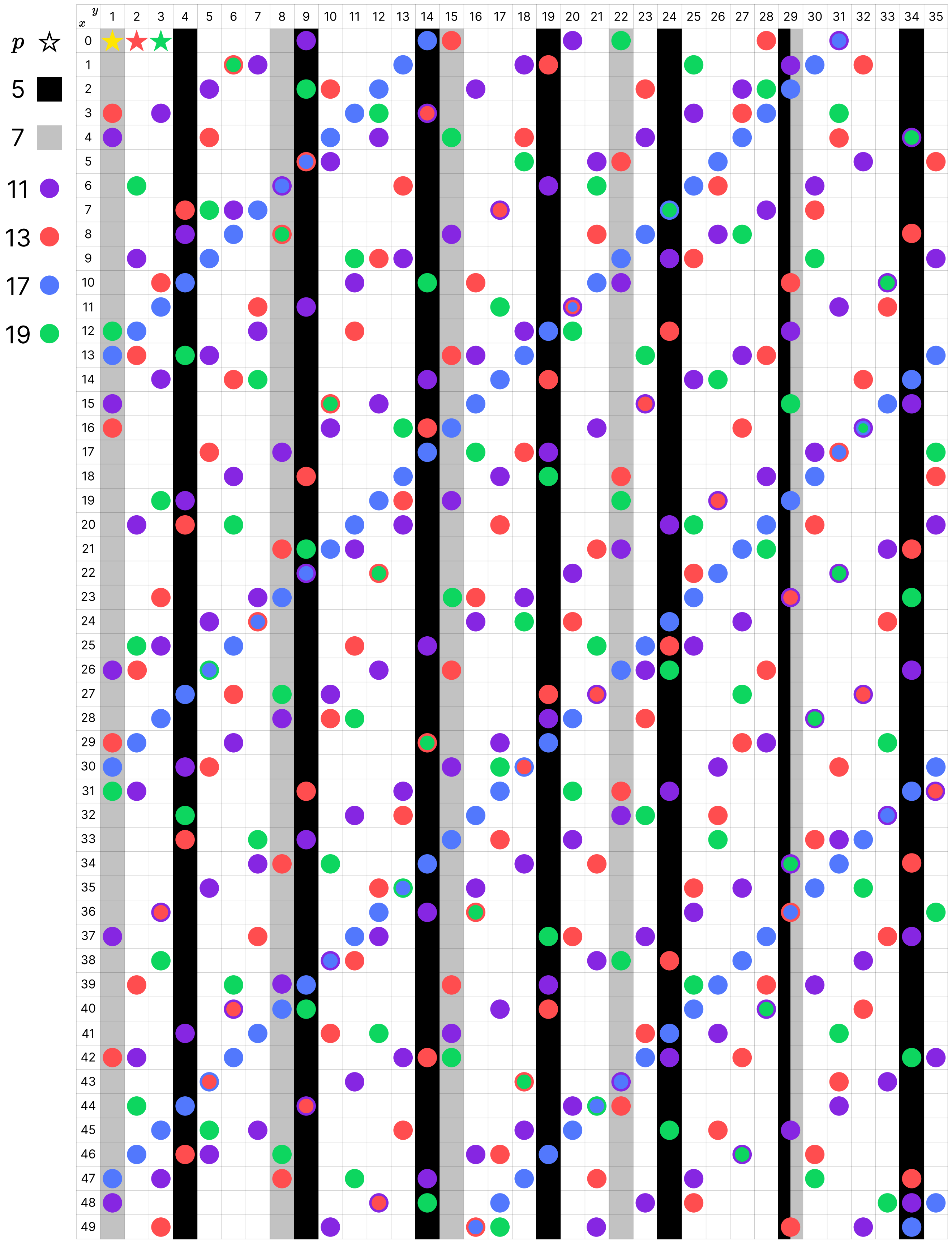

Figure 3.2: **iZm7** example (size 35 × 50). Each prime in {5, 7, 11, 13, 17, 19} marks its composite cells with a distinct color.

To validate these visualizations, select any colored cell at coordinates $[x,\ y]$ within iZm5 or iZm7. Then:

- In iZm5, calculate the corresponding iZ^{-} number as: $z = 6(x + 35 \times y) - 1$
- In iZm7, calculate the corresponding iZ^{+} number as: $z = 6(x + 35 \times y) + 1$

The resulting value z should be divisible by the prime(s) associated with the cell's color(s).

3.2.3 iZ-Lattice Structure

When focusing on the marks for specific primes, the highlighted cells often form a regular *lattice structure*. For example, Figure 3.3 isolates the marked primes in Figure 3.1 and demonstrates how each prime's composite cells manifest as repeated, orderly patterns.

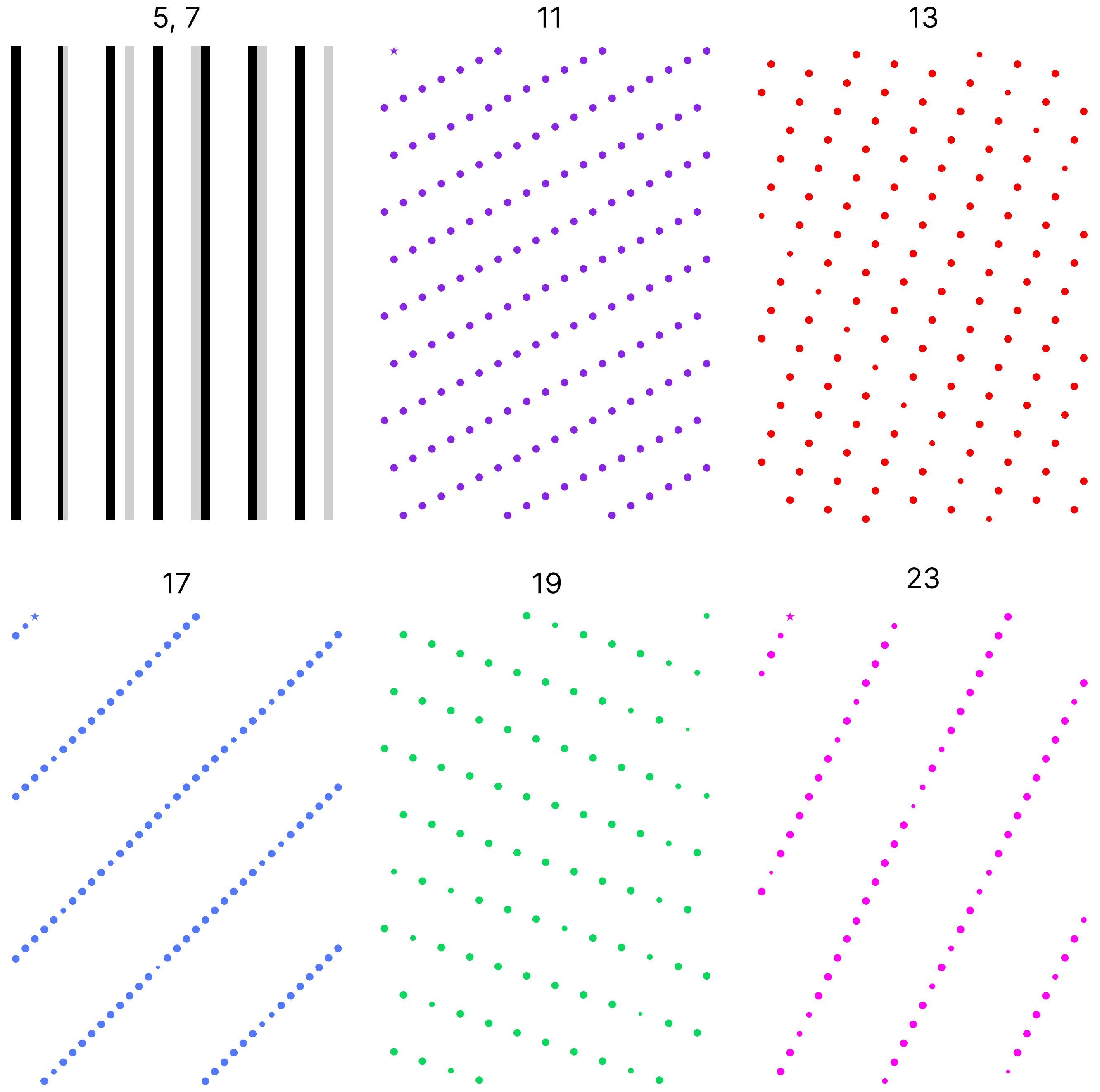

Figure 3.3: **iZ-Lattice** in iZm5 as an example. Each sub-figure represents a prime from $\{\{5, 7\}, 11, 13, 17, 19, 23\}$ marking its composite cells, forming distinct geometric patterns.

Such repetitive, "striped" patterns arise naturally from the iZ-Theorem's modular relationships and could be analyzed further through group–theoretic methods. In this work, we focus on leveraging them directly for segmentation in prime sieving.

3.3 Geometric Properties of iZm

The distinct visual patterns observed in the iZ-Matrix reflect core properties of modular arithmetic as applied to the $i\mathbb{Z}$ set. Recall from the iZ-Theorem, we know that for any prime $p \in i\mathbb{Z}$:

$$x_z \equiv \pm x_p \pmod p$$

mark the composites of p in the $i\mathbb{Z}$ subsets, where the sign ($\pm$) depends on both p's iZ-identity and whether we are in $i\mathbb{Z}^-$ or $i\mathbb{Z}^+$. Within an iZm of width vx, each iZ-prime displays one of two characteristic *marking patterns.*

3.3.1 iZm Pattern 1: Vertical Alignment

When

$$vx \equiv 0 \pmod p$$

and $[x,\ y]$ is a composite mark of p, then shifting horizontally by multiples of p leads to other composite columns of p at $[x \pm p \cdot m,\ y]$. In other words, if vx is a multiple of p, then p's composite marks align vertically in columns spaced exactly p units apart.

Visual Example

Just as multiples of 2 and 3 align vertically in a grid of width $vx = 6$ (see Figure 1.1 in Chapter 1), Figure 3.4 shows vertical alignment for primes 5 (black) and 7 (gray) in *iZm5* with $vx = 35$. Because 35 is divisible by both 5 and 7, their composite cells form straight vertical columns spaced by p.

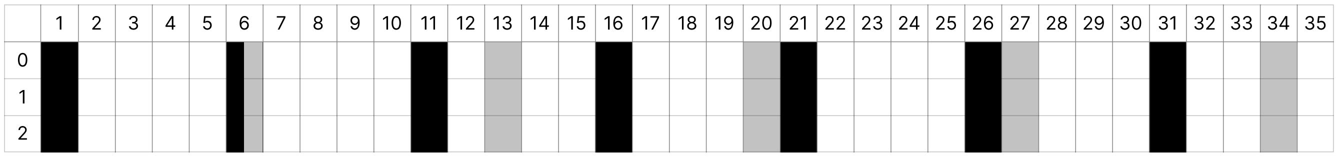

Figure 3.4: Vertical alignment of composites of 5 (black) and 7 (gray) in *iZm5* with $vx = 35$. Each prime's composite marks remain fixed on the x-axis across all rows.

Optimizing Composite Marking

By constructing a segment size vx to be divisible by chosen primes, one can mark those primes' composites just once in the first row—rather than in every row. Concretely, this reduces the total marking operations from

$$\frac{2(vx \times y)}{p} \quad \text{to} \quad \frac{2(vx)}{p},$$

which becomes especially advantageous as y (the number of segments) grows larger.

3.3.2 iZm Pattern 2: Inclined Alignment

When

$$vx \not\equiv 0 \pmod p$$

and $[x, y]$ is a composite mark of p, then shifting horizontally or vertically by multiples of p leads to other marks at $[x \pm p \cdot n,\ y \pm p \cdot m]$. Consequently, since vx is not a multiple of p, the composites of p appear in diagonal ("inclined") stripes, reflecting the fact that each row is offset by (vx mod p) to the left, relative to the previous row.

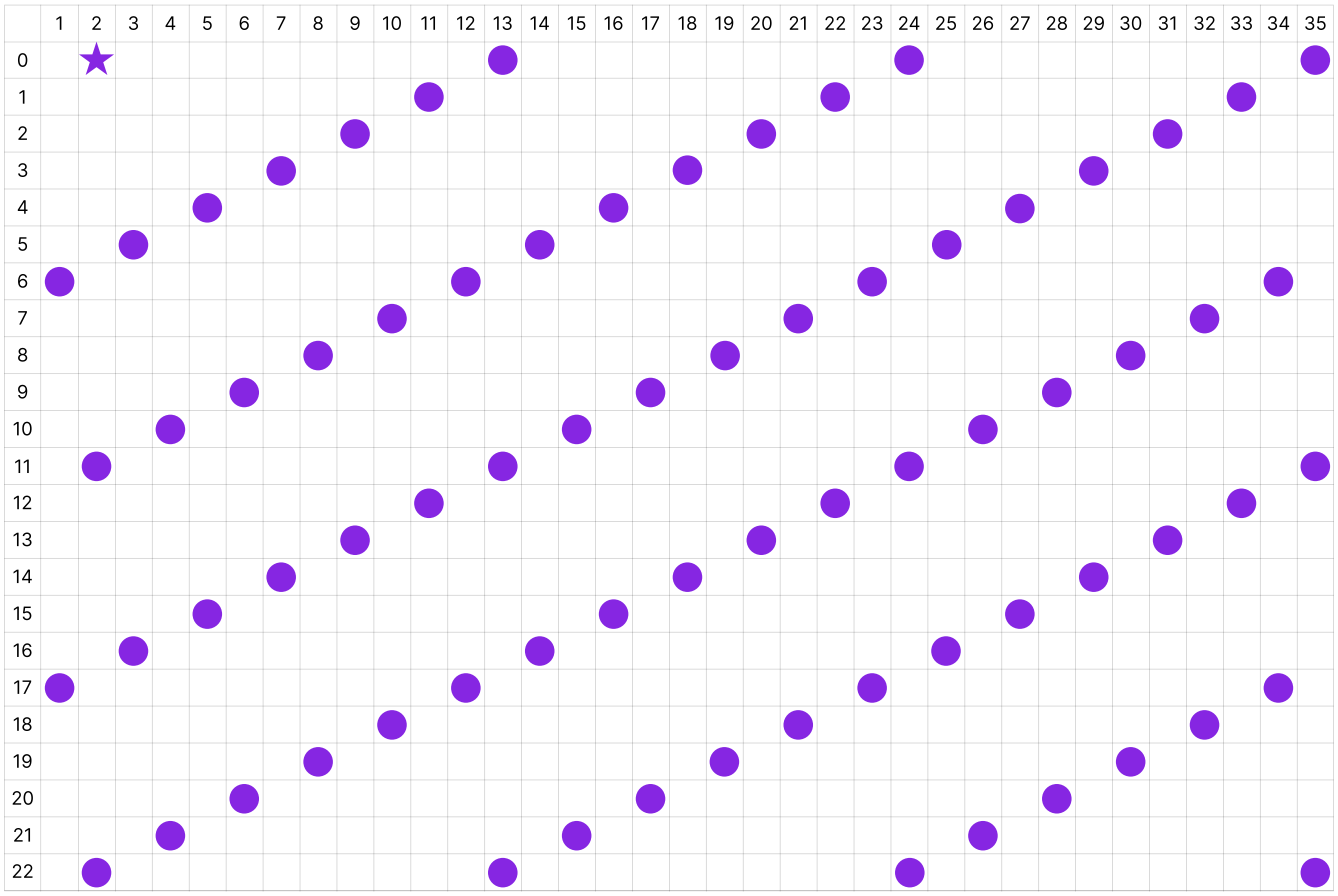

Figure 3.5: Inclined alignment of composites of 11 in *iZm5* with $vx = 35$. The marking pattern shifts each row, creating parallel diagonal stripes.

Visual Example

Figure 3.5 shows the composite marks of $p = 11$ in *iZm5* for $vx = 35$. Because $35 \equiv 2 \pmod{11}$, each successive row shifts 2 columns to the left (or -2 modulo 11), creating diagonal stripes. If we instead set $vx = 385$ (i.e., 35×11), these stripes would turn vertical, spaced 11 units apart along the x-axis.

3.3.3 In Summary

By representing $i\mathbb{Z}^-$ and $i\mathbb{Z}^+$ as two matrices—*iZm5* and *iZm7*—of strategic width vx, we gain powerful visual insights and computational framework for prime sieving. The geometric properties of the iZ-Matrix reveal two key marking patterns:

- **Vertical Alignment**, occurs when ($vx \equiv 0 \pmod{p}$): A prime p divides vx marks its composites in columns spaced by p, allowing the sieve to handle such primes with minimal overhead.

- **Inclined Alignment**, occurs when ($vx \not\equiv 0 \pmod{p}$): Other primes produce diagonal stripes repeating every p rows and columns.

These geometric insights underlie the upcoming *Segmented Sieve-iZm, Random-iZprime* algorithms. By carefully choosing the segment size vx, one can exploit these patterns to minimize marking operations and accelerate the sieving process—especially for large n. The next section introduces the tools for efficient iZm segmentation and processing.

3.4 Constructing iZm: Segmentation & Processing

In this section, we delve into the details of how to decide the segment size vx and constructing a pre-sieved base segment for the iZ-Matrix.

3.4.1 Calculate iZm Horizontal Vector Size VX

In traditional segmented sieves, the segment size is often tied to $\sqrt{n}$, the square root of the upper limit n. This unnecessary coupling can lead to suboptimal cache utilization and memory overhead, particularly for large n. By contrast, the *Segmented Sieve-iZm Algorithm* decouples the segment size from n and bases it instead on the *geometric properties* of the iZ-Matrix (*iZm*).

Key Idea:

Rather than using arbitrary $\sqrt{n}$ sizes, we construct the segment size vx by multiplying the first few primes in the $i\mathbb{Z}$ set (starting at 5) until $vx \times p \geq x_n$ or we exceed undesirable memory limit. This approach aims to:

- **Optimize Cache Utilization**: By using fixed segment sizes that fit comfortably into CPU caches, we can improve memory locality. This choice is hardware-dependent and can be tuned for specific architectures.
- **Minimize Marking Operations**: By ensuring vx is divisible by multiple small primes, we reduce the overhead of processing these primes to just one segment instead of every segment.

Function: `compute_limited_vx`

Below is a C function that computes vx given (1) the upper bound on the index range `x_n` and (2) a prime-aggregation limit `vx_limit`, which typically controls how many primes from `s_primes` we can multiply in before we risk exceeding cache-friendly sizes.

```c
// iZ.c
// List of small primes to be multiplied for vx
uint64_t s_primes[] = {5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47};
int s_count = sizeof(s_primes) / sizeof(s_primes[0]);

/**
 * @brief Calculate vx for a given range x_n
 *
 * @param x_n the range of bits to be vectorized
 * @param vx_limit the number of primes to be multiplied
 * @return size_t vx in range 35 to iZ primorial of limit primes
 */
size_t compute_limited_vx(size_t x_n, int vx_limit)
{
    size_t vx = 35;  // minimum vx size
    int i = 2;       // to skip 5 and 7

    // while vx * current prime doesn't exceed x_n & we haven't hit vx_limit:
    while (vx * s_primes[i] < (x_n / 2) && i < vx_limit)
    {
        // Multiply in the next prime
        vx *= s_primes[i];
        i++; // move to the next index
    }

    return vx;
}
```

The function initializes $vx = 35$ (the product of 5 and 7), then multiplies in additional small primes (e.g., 11, 13, 17, ...) until doing so either exceeds `x_n / 2` or surpasses `vx_limit`.

Usage: In the Segmented Sieve-iZm algorithm, we call `compute_limited_vx` to determine the optimal segment size vx for a given range n.

3.4.2 Construct Pre-Sieved iZm Segment

Once an optimal segment size vx is determined, the next step is to construct the base row in the matrices *iZm5* and *iZm7*. This row will serve as a pre-sieved segment to copy from, eliminating the need to repeatedly mark composites of the primes that divide vx, which their marks are fixed across all rows.

High-Level Procedure:

- **Initialization**:

  Start with `current_size` $= 35$ to account for primes $\{5, 7\}$.

  Sieve out 5 and 7 and their composites in `X5` and `X7` within the index range [1 : `current_size`].

- **For each prime p dividing vx** (from `s_primes` after 7):

  - **Duplicate** the existing segment pattern p times in both `X5` and `X7` to extend from `current_size` to `current_size` $\times p$.

  - **Update** `current_size` to `current_size` $\times p$.

  - **Mark** p and its composites in the newly extended region of both bitmaps.

This strategy effectively generates a fully sieved "base row" in `X5` and `X7` of size vx, ready for use in subsequent segments of the matrices *iZm5* and *iZm7*.

Function: `construct_iZm_segment`

Below is the C function that applies this logic to build a pre-sieved segment of length vx in the given `X5` and `X7` bitmaps:

```c
/**
* @brief Constructs a pre-sieved iZm segment of size vx.
*
* Marks all composites of small primes that divide vx in the bitmaps x5 (iZ-)
* and x7 (iZ+), so that subsequent segments can reuse this base segment.
*
* @param vx The total size of the segment.
* @param x5 A BITMAP as a segment of iZm5.
* @param x7 A BITMAP as a segment of iZm7.
*/
void construct_iZm_segment(size_t vx, BITMAP *x5, BITMAP *x7)
{
    size_t current_size = 35;  // Minimum starting size (5*7).

    // Step 1: Mark all bits in the range [1 : 35],
    // excluding 5 and 7 and their composites
    for (size_t i = 1; i <= current_size; i++)
    {
        // In x5, skip indices that are 1 mod 5 or -1 mod 7.
        if (((i - 1) % 5) != 0 && ((i + 1) % 7) != 0) {
            bitmap_set_bit(x5, i);
        }
        // In x7, skip indices that are -1 mod 5 or 1 mod 7.
        if (((i + 1) % 5) != 0 && ((i - 1) % 7) != 0) {
            bitmap_set_bit(x7, i);
        }
    }
    // Step 2: For each prime p beyond 7 that divides vx:
    int idx = 2;  // to skip 5, 7
    while (vx % s_primes[idx] == 0)
    {
        size_t p = s_primes[idx];
        idx++;

        // Convert prime p to its iZ index.
        // For p in iZ+, x = (p + 1)/6, etc.
        int x = (p + 1) / 6;

        // Step 2a: Duplicate the range [1:current_size] p.
        bitmap_duplicate_segment(x5, 1, current_size, p);
        bitmap_duplicate_segment(x7, 1, current_size, p);

        // Extend current_size by factor p.
        current_size *= p;

        // Step 2b: Mark p and its composites in the new region of x5, x7
        // using the Xp-Wheel.
        if ((p % 6) > 1) {
            // p is in iZ+ => mark its iZ^- composites, then iZ+.
            bitmap_clear_mod_p(x5, p, x, current_size + 1);
            bitmap_clear_mod_p(x7, p, p * x - x, current_size + 1);
        } else {
            // p is in iZ- => mark its iZ^+ composites, then iZ^-.
            bitmap_clear_mod_p(x5, p, p * x - x, current_size + 1);
            bitmap_clear_mod_p(x7, p, x, current_size + 1);
        }
    }
}
```

This function populates `X5` and `X7` with a fully pre-sieved segment of length vx. The Segmented Sieve-iZm Algorithm then reuses this "base segment" for covering the entire range of n in iZm5 and iZm7, thus eliminating the need of repeatedly processing small primes and accelerating prime generation across larger ranges.

3.5 *VX6* Vector for Efficient Segmentation

For best results in the Segmented Sieve-iZm algorithm, we set `vx_limit` = 6 in the function `compute_limited_vx`. Regardless of the input range n, this limits the segment size to

$$vx6 \;=\; 5 \times 7 \times 11 \times 13 \times 17 \times 19 \;=\; 1,616,615 \text{ bits} \quad \approx 0.2 \text{ MB}.$$

Once vx is determined, we invoke `construct_iZm_segment` to build the base row in the `X5` and `X7` bitmaps, pre-sieved from those primes.

3.5.1 Operational Cost of *VX6*

The breakdown below illustrates how many marking operations are required to construct the base segment of size $vx6$:

1. **Constructing VX2**:

    Processing the primes {5,7} in the range [1:35] costs constant 70 marking operations.

2. **Constructing VX3**:

Next prime: 11, in the extended range [1:385], costs

$$\frac{2 \times 385}{11} = 70.$$

3. **Constructing VX4**:

   Next prime: 13, in the extended range [1:5,005], costs

$$\frac{2 \times 5,005}{13} = 770.$$

4. **Constructing VX5**:

   Next prime: 17, in the extended range [1:85,085], costs

$$\frac{2 \times 85,085}{17} = 10,010.$$

5. **Constructing VX6**:

   Next prime: 19, in the extended range [1:1,616,615], costs

$$\frac{2 \times 1,616,615}{19} = 170,170.$$

Summing these yields a total marking cost of

$$70 + 70 + 770 + 10010 + 170170 = 181,090.$$

This is a **fixed** one-time cost for constructing *vx6* and is amortized over all subsequent segments in the iZ-matrix.

3.5.2 Efficiency of the Pre-Sieved *VX6*

By using this *vx6* vector as a pre-sieved base segment in the Segmented Sieve-iZm algorithm, we effectively remove those primes from the routine marking process across larger ranges. Consequently, we further optimize the time complexity of the Xp-Wheel from

$$O\left(\frac{n}{3} \sum_{5 \le p \le \sqrt{n}} \frac{1}{p}\right) \quad \text{to} \quad O\left(\frac{n}{3} \sum_{23 \le p \le \sqrt{n}} \frac{1}{p}\right)$$

because {5, 7, 11, 13, 17, 19} no longer contribute to the main sieving loop. If we had not pre-sieved them, their cumulative cost would be about

$$\frac{n}{3}\left(\frac{1}{5} + \frac{1}{7} + \frac{1}{11} + \frac{1}{13} + \frac{1}{17} + \frac{1}{19}\right) \quad \approx \quad \frac{n}{5} \text{ operations.}$$

This **constant** overhead $O(\approx 180,000)$ instead of $O(n/5)$ is a significant improvement that becomes more pronounced as n grows larger. These savings are crucial for justifying the incredible efficiency of the Segmented Sieve-iZm Algorithm.

3.5.3 Balancing Space & Time Complexities with *VX6*

One might ask why not include additional primes beyond 19, as we incrementally exclude more primes gives us sub-linear time complexity. However, two key factors limit the practical utility of extending the base segment beyond *vx6*:

1. **Hardware Constraints**: At segment size *vx6* (1,616,615 bits), we only need ($\approx$ 0.2 MB) for each `X5` and `X7`—comfortably fitting into the L2 cache of most modern CPUs. Including the next prime (23) extends this size to ($37,182,145$ bits $\approx$ 4.5 MB) for each X array, which will exceed cache capacities, and thus degrade performance due to more frequent cache misses.

2. **Diminishing Returns**: Each additional larger prime yields ever smaller gains (since $\frac{1}{p}$ diminishes as p grows). At some point, the extra overhead of a larger base segment outweighs the benefits of skipping those primes during the main sieving phase, especially if they come at the cost of losing cache privileges.

In practice, {5,7,11,13,17,19} prove sufficient to deliver near-optimal performance for the Segmented Sieve-iZm algorithm on typical contemporary hardware. Nevertheless, as hardware evolves—particularly with bigger caches or faster memory access—one could include more primes (e.g., 23, 29, ...) in the base segment to further reduce marking operations. Ultimately, selecting `vx_limit` is a trade-off between space and time, and can be tuned based on the application domain and available hardware resources.

3.5.4 *VX6*: Visualization

In the iZ-Matrices *iZm5* and *iZm7*, constructed by the *vx6* horizontal vector, derived from {5, 7, 11, 13, 17, 19}, all these primes mark their composites in vertical columns spaced p units apart. Figure 3.6 illustrates this effect, showing just the first 35 columns (out of the full 1,616,615) in both iZm5 (top) and iZm7 (bottom):

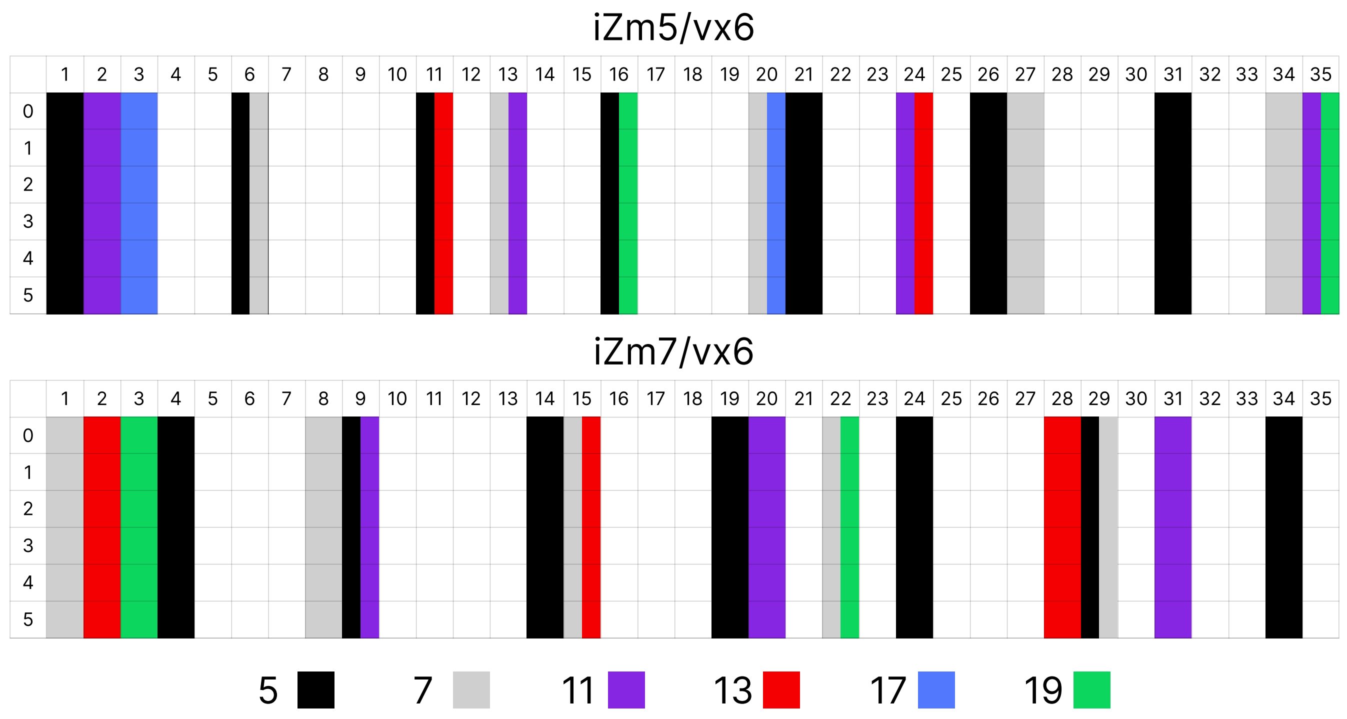

Figure 3.6: **The *vx6* Segment in iZm5 (top) and iZm7 (bottom).** Each subfigure displays the first 35 columns of the full 1,616,615-column vector. Because $vx = 1,616,615$ is divisible by {5, 7, 11, 13, 17, 19}, these prime factors' previously inclined stripes (seen in earlier examples) collapse into vertical lines spaced by p units.

Key Observation: In this small sample of the full *vx6* length (Figure 3.6), we can observe that these primes mark about half of the columns in both iZm5 and iZm7, reducing the search space within the $i\mathbb{Z}$ domain by half (from $n/3$ to $n/6$), as in this iZm structure with *vx6*, primes beyond 19 cannot be found in the columns marked (colored) by these primes.

3.6 iZm: Analytical Tools & Statistical Insights

Beyond powering efficient prime sieving, the *iZm* structure also reveals noteworthy patterns in prime distributions. This section examines how certain "special" prime pairs—twin primes, cousin primes, and sexy primes—manifest within progressively pre-sieved *vx* segments. By tracking how many of these prime pairs survive in each segment, we gain a clearer view of how *iZm* restricts potential residue classes *before* the broader sieve even runs.

3.6.1 Overview

In classical number theory, three prime pairs often draw special attention:

- **Twin Primes** are prime pairs $(p, p+2)$.
- **Cousin Primes** are prime pairs $(p, p+4)$.
- **Sexy Primes** are prime pairs $(p, p+6)$.

In the *iZm* grids, these pairs must align in adjacent (or nearly adjacent) columns based on their difference. Studying how these pairs appear (and vanish) as *vx* grows highlights which residue classes remain feasible for such small prime gaps.

3.6.2 Function: `print_vx_stats`

To gather these statistics, we implemented a helper function `print_vx_stats`. Given:

- A segment size *vx*,
- Two bitmaps `x5` and `x7` representing the $i\mathbb{Z}^-$ and $i\mathbb{Z}^+$ subsets,

this function iterates over the range [1 : `vx`] to count:

1. Total primes, the set bits in `x5` or `x7` after sieving out the primes dividing *vx*.
2. Twin primes (p and $p+2$), which in iZ terms means the same index x is prime in both `x5` and `x7`.
3. Cousin primes (p and $p+4$), occurs when x in `x5` and $(x-1)$ bit in `x7` are set.
4. Sexy primes (p and $p+6$), found when (x, $x-1$) are both set in either `x5` or `x7`.

In code:

```c
// iZ.c
/**
 * @brief Analyze and print prime-related statistics in a vx segment
 * of iZm5/iZm7.
 * @param vx  Upper limit of the range to analyze (1..vx).
 * @param x5  BITMAP pointer for iZm5 (iZ-).
 * @param x7  BITMAP pointer for iZm7 (iZ+).
 */
void print_vx_stats(size_t vx, BITMAP *x5, BITMAP *x7)
{
    // Initialize counters
    int primes_count = 0;
    int iZm5 = 0; // how many primes in iZm5
    int iZm7 = 0; // how many primes in iZm7
    int twin_count = 0;
    int cousins_count = 0;
    int sexy_count = 0;

    // Iterate over [1:vx] in both bitmaps x5 (iZ-) and x7 (iZ+)
    for (uint64_t x = 1; x <= vx; x++)
    {
        // Check if x is prime in iZ-
        if (bitmap_get_bit(x5, x))
        {
            primes_count++;
            iZm5++;
        }
        // Check if x is prime in iZ+
        if (bitmap_get_bit(x7, x))
        {
            primes_count++;
            iZm7++;
        }

        // Twin primes: check iZm5 & iZm7 at the same x
        if (bitmap_get_bit(x5, x) && bitmap_get_bit(x7, x))
            twin_count++;

        // Cousin primes: (p in x5 at x) and (p in x7 at x-1), etc.
        if (bitmap_get_bit(x5, x) && bitmap_get_bit(x7, x - 1))
            cousins_count++;

        // Sexy primes can occur in iZm5 at [x,x-1] or iZm7 at [x,x-1]
        if (bitmap_get_bit(x5, x) && bitmap_get_bit(x5, x - 1))
            sexy_count++;

        if (bitmap_get_bit(x7, x) && bitmap_get_bit(x7, x - 1))
            sexy_count++;
    }

    // Print results as a formatted row
    printf("| %-12zu", 2 * vx);        // total #bits in iZm5 + iZm7
    printf("| %-12d", primes_count);
    printf("| %-12d", twin_count);
    printf("| %-12d", cousins_count);
    printf("| %-12d\n", sexy_count);
}
```

Prime Parity in the iZ Context:

- Twin primes in this iZ context means $iZ(x,-1)$ and $iZ(x,1)$, thus differ by 2.
- Cousin primes happen when $iZ(x,-1)$ and $iZ(x,1)$ are both prime
- Sexy primes occur when $iZ(x,-1)$ and $iZ(x+1,-1)$ are both prime, or $iZ(x,1)$ and $iZ(x+1,1)$

In the next subsection, we apply this function to analyze the statistical distribution of primes in pre-sieved different segment sizes.

3.6.3 Function: `analyze_vx_potential_primes`

To gather stats at multiple segment sizes (*vx1*, *vx2*, . . .), we use an "umbrella" function that:

1. **Initializes** the bitmaps `x5` and `x7` for each tested segment size.
2. **Sieves out** the relevant prime p by marking its composites in `x5` and `x7` when vx is divisible by p.
3. **Invokes** `print_vx_stats` for each vx to retrieve counts of primes, twins, cousins, and sexy primes.
4. **Cleans up** all allocated data structures at the end.

```
// iZ.c
/**
 * @brief Analyzes the search space for potential primes in iZm5 and iZm7
 * for vx sizes vx1 to vx8.
 */
void analyze_vx_potential_primes(void)
{
    // Statistical analysis about the search space for primes
    print_line(92);
    printf("| %-12s", "2 * vx");
    printf("| %-8s|%-8s", "iZm5", "iZm7");
    printf("| %-12s", "Total PP");
    printf("| %-12s", "Twins");
    printf("| %-12s", "Cousins");
    printf("| %-12s", "Sexy");
    print_line(92);

    // Initialize x5, x7 bitmaps
    BITMAP *x5, *x7;
    size_t max_vx = 5 * 7 * 11 * 13 * 17 * 19 * 23 * 29;

    // Allocate memory for x5 and x7 with vx + 100 bits
    x5 = bitmap_create(max_vx + 100);
    x7 = bitmap_create(max_vx + 100);

    // Set all bits initially as candidates for primes
    bitmap_set_all(x5);
    bitmap_set_all(x7);

    bitmap_clear_bit(x5, 0);
    bitmap_clear_bit(x7, 0);

    // Mark columns of 5
    bitmap_clear_bit(x5, 1);
    bitmap_clear_bit(x7, 4);

    size_t current_size = 5;
    // this will print the space for potential primes
    // in iZm5 and iZm7 of segment size vx = 5
    print_vx_stats(current_size, x5, x7);

    int idx = 1; // to skip 5
    while (max_vx % s_primes[idx] == 0)
    {
        size_t p = s_primes[idx];
        idx++;

        // Convert prime p to its iZ index.
        // For p in iZ+, x = (p + 1)/6, etc.
        int x = (p + 1) / 6;

        // Duplicate the range [1:current_size] p.
        bitmap_duplicate_segment(x5, 1, current_size, p);
        bitmap_duplicate_segment(x7, 1, current_size, p);

        // Extend current_size by factor p.
        current_size *= p;

        // Mark p and its composites in the new region of x5, x7
        // using the Xp-Wheel.
        if ((p % 6) > 1)
        {
            // p is in iZ+ => mark its iZ^- composites, then iZ+.
            bitmap_clear_mod_p(x5, p, x, current_size + 1);
            bitmap_clear_mod_p(x7, p, p * x - x, current_size + 1);
        }
        else
        {
            // p is in iZ- => mark its iZ^+ composites, then iZ^-.
            bitmap_clear_mod_p(x5, p, p * x - x, current_size + 1);
            bitmap_clear_mod_p(x7, p, x, current_size + 1);
        }
        // for statistical analysis in this segment size
        print_vx_stats(current_size, x5, x7);
    }

    // Clean up bitmaps
    bitmap_free(x5);
    bitmap_free(x7);
}
```

Running this function (e.g., from `main`) prints row-by-row statistics of potential primes and prime pairs for *vx1* through *vx8*.

Output Summary

The output of `analyze_vx_potential_primes` can be found in `/output/stdout.txt`, summarized in Table 3.6.3 below:

vx	**vx1**	**vx2**	**vx3**	**vx4**	**vx5**	**vx6**	**vx7**	**vx8**
Total Bits	10	70	770	10,010	170,170	3,233,230	74,364,290	2,156,564,410
Primes	8	48	480	5760	92,160	1,658,880	36495360	1,021,870,080
Twins	3	15	135	1485	22,275	378,675	7,952,175	214,708,725
Cousins	3	15	135	1485	22,275	378,675	7,952,175	214,708,725
Sexy	5	30	270	2970	44,550	757,350	15,904,350	429,417,450

Table 3.1: Statistical Data for Potential Primes and Pairs in iZm5/iZm7.

Interpreting the Results (Example with *vx1* = *5*)

To illustrate how *vx1* = *5* reduces the search space for primes and prime pairs, consider the following statistics:

- **Total Bits**: $2 \times 5 = 10$ bits in `X5` and `X7`, covering a range of $10 \times 3 = 30$ natural numbers, 20 of which are multiples of 2 or 3, leaving 10 candidates of the iZ-form $6x \pm 1$.
- **Possible Primes**: 8 of those 10 positions remain unmarked after excluding multiples of 5 in `X5` and `X7`.
- **Twin Primes**: 3 valid twin pairs remain.
- **Cousin Primes**: 3 valid cousin pairs remain.
- **Sexy Primes**: 5 valid sexy pairs remain.

For larger *vx* values, this pattern generalizes: each time we multiply in a new iZ-prime p, we remove the composite residue classes modulo p. As a result, the overall search space for primes—and hence prime pairs—shrinks further.

Of particular interest is *vx6* = 1,616,615, the product of {5,7,11,13,17,19}. According to Table 3.6.3, *vx6* comprises 3,233,230 total bits across $i\mathbb{Z}^-$ and $i\mathbb{Z}^+$, which corresponds to a range of $3,233,230 \times 3 = 9,699,690$ natural numbers. Of these, 1,658,880 bits remain valid after sieving out composites from those six small primes, effectively cutting the $i\mathbb{Z}$-based search space in half (reducing from $\frac{n}{3}$ to about $\frac{n}{6}$).

In the next section, we build on these insights to present algorithmic solutions for processing *iZm*, demonstrating how focusing on primes already accounted for in the segment size vx helps us skip redundant marking steps. The result is a highly optimized sieve that combines theoretical efficiency with practical performance.

3.7 Algorithmic Solutions for Processing iZm

Thus far, we have focused on how to pre-sieve small primes within the iZ-Matrix. However, once we move beyond the primes that define the segment size vx, we move to processing the rest of root primes ($p \leq \sqrt{n}$). This section details how to do so in both **horizontal** and **vertical** modes, leveraging the structure and modular relationships of the iZ-Matrix.

3.7.1 Sieve Modes in iZm

The *iZm* 2D structure supports two primary approaches to composite marking:

1. **Horizontal Sieve**

   Processes a *row* of length vx with fixed y, where we process the *iZm* row by row incrementing the value of y until the entire range is covered.

2. **Vertical Sieve**

   Processes a column of length vy in *iZm* with fixed x. This mode is especially relevant in certain prime factorization tasks or advanced number-theoretic applications, where the goal might be to skip entire columns that can't produce primes.

In both modes, given a prime p, we need to locate its first composite mark in a specified row or column, then we can mark all subsequent composites of p by increments of p in either directions. This section presents efficient solutions for solving this problem in both horizontal and vertical sieving.

3.7.2 Solving for x in a vx Vector

When performing a horizontal sieve, marking the composites of a prime p in segment vx and fixed y, we seek the first composite mark of p, which is the smallest x such that

$$x + y \times vx \quad \equiv \pm x_p \pmod{p},$$

given the parameters:

- $matrix\_id \in \{-1, 1\}$: where -1 indicates *iZm5* and 1 for *iZm7*,
- p: the prime in question,
- vx: the segment (row) size,
- y: the current row index in the matrix.

Implementation

```
// Solve for the smallest x satisfying (x + vx*y) \equiv x_p (mod p),
// where matrix_id determines whether to use x_p or p - x_p based on iZ rules.
uint64_t solve_for_x(int matrix_id, uint64_t p, size_t vx, uint64_t y)
{
    // 1. Normalize x_p to either x_p or p - x_p, depending on matrix_id
    uint64_t x_p = normalized_x_p(matrix_id, p);

    // 2. Compute x
    // (vx*y - x_p) mod p gives how far x is from alignment;
    // subtracting from p yields the smallest positive solution.
    uint64_t x = p - ((vx * y - x_p) % p);
    return x;
}
```

Steps to Solve for x

1. **Normalization**: A helper function (`normalized_x_p`) applies the iZ-Theorem to determine whether to use x_p or $p - x_p$ based on *matrix_id* and whether p is in $i\mathbb{Z}^-$ or $i\mathbb{Z}^+$.
2. **Calculating x**: We rearrange

$$x + y \times vx \equiv x_p \pmod{p}$$

   into

$$x \equiv x_p - vx \times y \pmod{p}.$$

   The line `x = p - ((vx * y - x_p) % p);` ensures we get the smallest positive solution for x. This x value positions the first composite of p in the targeted vx vector, then the marking proceeds by increments of p.

3.7.3 Solving for y in a vy Vector

For vertical sieving, we fix x and seek the smallest y that satisfies

$$x + y \times vx \quad \equiv \pm x_p \pmod{p}.$$

Given parameters:

- $matrix\_id \in \{-1, 1\}$,
- vx: the segment size,
- p: the prime in question,
- x: the fixed column index,

Implementation

```
// Extended Euclidean Algorithm to find multiplicative inverse of 'a' mod 'm'
int modular_inverse(int a, int m);

/**
* @brief Solve for the smallest y in (x + vx*y) \equiv x_p (mod p).
* @return The computed y, or -1 if no solution exists.
*/
int solve_for_y(int matrix_id, int p, int vx, int x)
{
    // 1. Check gcd(vx,p)
    // If p divides vx, there's no modular inverse => no valid solution.
    if (vx % p == 0)
    {
        printf("No solution; vx and p are not co-prime.\n");
        return -1;
    }

    // 2. Normalize x_p based on matrix_id and p's identity (+/- 1).
    int x_p = (p + 1) / 6;
    int p_id = (p % 6 == 1) ? 1 : -1;

    if (matrix_id < 0) // iZm5
        x_p = (p_id < 0) ? x_p : p - x_p;
    else // iZm7
        x_p = (p_id < 0) ? p - x_p : x_p;

    // 3. Check for immediate solution (if x \equiv x_p mod p)
    if (x % p == x_p)
        return 0;

    // 4. Compute delta = x_p - x (mod p).
    int delta = (x_p - x) % p;
    if (delta < 0)
        delta += p;

    // 5. Find inverse of vx mod p using the Extended Euclidean Algorithm.
    int vx_inv = modular_inverse(vx, p);

    // 6. Compute y = (delta * vx_inv) mod p
    int y = (delta * vx_inv) % p;
    return y;
}
```

Steps to Solve for y

1. **Co-Primality Check**: If p divides vx, no modular inverse exists, so there is no solution. This is indicated by returning -1.
2. **Normalization**: Similar to the horizontal case, use the iZ-identity of p and `matrix_id` to choose x_p or $(p-x_p)$.
3. **Immediate Solution Check**: If $x \bmod p = x_p$, then $y = 0$ immediately solves the equation.
4. **Solve for y**:
   - Compute `delta` $= (x_p - x) \bmod p$.
   - Find the modular inverse vx_inv of $vx \bmod p$, using the Extended Euclidean Algorithm.
   - Multiply them: $y = (delta * vx\_inv) \bmod p$.

This result y marks the first composite of p in the targeted vy vector, and marking proceeds by increments of p.

3.7.4 Practical Takeaways

These two functions provide the core logic for locating a prime p's *first composite* in either a row or column of the iZ-Matrix. Once found, marking subsequent composites in increments of p is straightforward using the Xp-Wheel.

iZm Sieving Strategies:

- **Horizontal Sieve**:

  Ideal for segmented approaches, scanning consecutive rows. A prime p's composites in each row are quickly discovered and marked.

- **Vertical Sieve**:

  Useful in advanced applications (e.g., factorization) where skipping entire unproductive columns can save substantial work.

By leveraging these strategies, we can efficiently process the iZ-Matrix for a wide range of number-theoretic tasks, from prime enumeration to advanced factorization.

3.8 Summary & Directions

In this chapter, we shifted from the one-dimensional logic of the $i\mathbb{Z}$ sets to the more powerful, two-dimensional structure of the *iZ-Matrix (iZm)*. By arranging $i\mathbb{Z}^-$ and $i\mathbb{Z}^+$ into grids of size $vx \times vy$, we uncovered intrinsic geometric properties that enable systematic reduction of composite-marking overhead. In particular, we saw how primes dividing vx cause vertical alignments, while all other primes produce inclined (diagonal) stripes.

Key Contributions:

1. **Definition & Visualization**:

   We introduced *iZm5* and *iZm7*, demonstrating how to map the $i\mathbb{Z}^-$ and $i\mathbb{Z}^+$ sets into two-dimensional grids. Through figures and examples, we illustrated the lattice-like structures that emerge when primes mark their composites, highlighting the geometric patterns that guide efficient sieving.

2. **Base Segment Construction**:

   We presented two core functions, `compute_limited_vx` and `construct_iZm_segment`, for building a pre-sieved segment of fixed size vx. By factoring out small primes once, we amortize their marking cost and achieve constant overhead for those primes.

3. **Analytical & Statistical Tools**:

   The `print_vx_stats` and `analyze_vx_potential_primes` functions revealed how restricting residue classes (via vx) narrows the pool of candidate primes, highlighting how this affect the distribution of twin, cousin, and sexy primes within the iZ-Matrix.

4. **Horizontal & Vertical Sieving**:

   We concluded with practical algorithmic solutions for finding a prime p's first composite in either a row or a column of the matrix. The `solve_for_x` and `solve_for_y` routines leverage modular arithmetic to pinpoint these composites efficiently, paving the way for flexible sieving strategies.

Looking ahead, the next chapters will build on these concepts, introducing:

- **The Segmented Sieve-iZm Algorithm**:

  Employs maximum *vx6* as a pre-sieved base segment to achieve constant space complexity of $O(1)$, only requiring $4 \times vx6 \approx 0.8\,\text{MB}$ of memory, and optimizes the Xp-Wheel operations by starting from 23 instead of 5, at the cost of a fixed work *independent* from n. Together, these features deliver unmatched efficiency for large-scale prime sieving.

- **The VX6-Sieve Algorithm**:

  Standardizing the *vx6* segment as a cache-friendly pre-sieved base–covering fixed intervals $S = 6 \times vx6 = 9699690$ of natural numbers, this algorithm limits the deterministic sieve work to the primes less *vx6*, leaving only 4% of natural numbers that warrant further examination, using the Miller–Rabin primality test. This approach caps the bit-complexity at linear $O(n)$, with remarkable constant factors of $(\frac{S}{2})$ rapid bitwise operations and $(\frac{4S}{100})$ primality tests operations.

- **The Random-iZprime Algorithm**:

  Given a *bit-size*, and focusing only on generating one random prime for cryptographic applications, this algorithm uses maximum $vx < 2^{bit-size}$ sizes to align the most frequent composites vertically in the iZ-Matrix—a *purification technique* that yields unmarked columns with high prime density. Combined with the Miller–Rabin primality test on random unmarked columns, we can generate very large bit-size primes with great efficiency.

Chapter 4

The Segmented Sieve-iZm Algorithm

4.1 Introduction

This chapter presents the **Segmented Sieve-iZm Algorithm**, a highly efficient method for generating all primes up to a given limit n. We begin by outlining the algorithm's structure and describing how it builds on the iZ-Framework foundations. A detailed C implementation follows, highlighting how the base-segment concept and the Xp-Wheel seamlessly integrate. We then compare performance with other well-known sieve algorithms, and conclude with performance analysis.

4.2 *Sieve-iZm*: Description & Implementation

This section provides a detailed overview of the Segmented Sieve-iZm Algorithm. After describing its structure, we present a concrete C implementation.

4.2.1 Description

Building on the foundations of the iZ-Framework, the **Segmented Sieve-iZm Algorithm** integrates:

1. **X*p*-Wheel Concepts** (from Chapter 1): For efficient composite marking within the $i\mathbb{Z}$ space.
2. **iZ-Matrix & Its Toolset** (Chapter 3): For streamlined segmentation, base-segment construction, and locating each prime's first composite in a segment. In particular, it makes use of:
   - The `compute_limited_vx` function to determine the optimal segment size vx for a given range n.
   - `construct_iZm_segment` function to generate the pre-sieved base segment.
   - `solve_for_x` function to locate the first composite position of a prime p in the current segment.

The Segmented Sieve-iZm Algorithm proceeds in five primary stages: **(1) Initialization**, **(2) Preprocessing**, **(3) Processing the First Segment**, **(4) Processing Remaining Segments**, and **(5) Returning Results**.

1. Initialization

We begin by setting up all necessary data structures and parameters:

1. **Initialize** the `primes` array with enough capacity to store all primes up to n and insert the primes $\{2, 3\}$.
2. **Compute** `x_n` as $\lfloor \frac{n+1}{6} \rfloor + 1$ to represent the maximum iZ-index in the range of n.
3. **Initialize** `s_primes`, an array of small iZ-primes $\{5, 7, 11, 13, 17, 19, \ldots\}$.
4. **Initialize** `start_i` $= 2$ to track how many primes (pre-sieved) to skip from the marking composites routine.
5. **Calculate** the optimal segment size `vx` using `compute_limited_vx(x_n, vx_limit = 6)`.

6. **Append** to `primes` (From `s_primes`) any primes that divide `vx`, and increment `start_i` accordingly.
7. **Allocate** four bitmaps—`x5`, `x7`, `tmp5`, `tmp7`—each of length `vx`.

2. Preprocessing

Next, we construct the pre-sieved base segment:

- **Call** `construct_iZm_segment(vx, x5, x7)` to populate `x5` and `x7` with the first row of length `vx` in *iZm5/iZm7* where composite marks of primes that divide `vx` are fixed across all segments.

3. Process First Segment

We now process a copy of `x5` and `x7` as the first segment ($y = 0$):

1. **Clone** `x5` and `x7` to `tmp5` and `tmp7` for active sieving, preserving the original bitmaps for future resets.
2. **For** each x in range $\big[1 : vx\big]$.
   - If `tmp5`[x] is marked prime:
     - **Compute** prime $p = \mathrm{iZ}(x, -1)$, and append it to `primes`.
     - If $p \leq \sqrt{n}$, mark its composites in the current segment (`tmp5` and `tmp7`).
3. **Repeat** similarly for `tmp7`[x] (iZ+).
4. **Result**: By the end, we have gathered all primes necessary for processing subsequent segments.

4. Process Remaining Segments

For segments beyond the first one ($y > 0$):

1. **Compute** `max_y` $= \lfloor \frac{\texttt{x\_n}}{\texttt{vx}} \rfloor$, the number of remaining segments.
2. **Compute** `limit` = `vx` for all segments, except last segment, it's `x_n % vx`.
3. **For** each segment index `y` in range $\big[1 : \texttt{max\_y}\big]$:
   - **Reset** `tmp5` and `tmp7` by cloning the original `x5` and `x7`.
   - **For** each root prime $p \leq \sqrt{n}$ and after the pre-sieved ones in `primes`:
     - **Locate** the first composite position in *iZm5/iZm7* using the `solve_for_x` function.
     - **Mark** composites of p in the current segment (`tmp5` and `tmp7`) using the Xp–Wheel.
   - **Collect** unmarked indices as primes in the current segment, for each index x in range[1 : `limit`]
     - If `tmp5`[x] is prime, append $p = \mathrm{iZ}(x + \texttt{vx} \times y, -1)$ to `primes`.
     - If `tmp7`[x] is prime, append $p = \mathrm{iZ}(x + \texttt{vx} \times y, 1)$ to `primes`.

5. Clean Up & Return Results

1. **Free** the bitmaps `x5`, `x7`, `tmp5`, `tmp7`.
2. **Remove** any primes greater than n from the `primes` array.
3. **Trim** unused memory in the `primes` object.
4. **Return** the `primes` array.

Key Ideas

- **Constant Space Complexity** $O(1)$

  Apart from the output (the `primes` array), only a fixed auxiliary space of about 0.8 MB is required (four *vx6*-bitmaps, each ≈ 0.2 MB). This remains *constant*, independent of n, ensuring cache-friendly data-access.

- **Segment Reset**

  Each new segment, `tmp5` and `tmp7` clone from the original `x5`, `x7` for active sieving, automatically including composite marks for primes that divide vx.

- **Composite Marking**

  1. **Primes dividing *vx*** (up to 19) mark their composites *once* in the base segment, and require no further marking.
  2. **All other primes** $p \leq \sqrt{n}$ rely on `solve_for_x` to locate the first composite in each segment, then use the Xp-Wheel to mark subsequent composites.

Together, these design choices enable the Segmented Sieve-iZm Algorithm to handle large n with minimal overhead and constant auxiliary memory, delivering a highly scalable approach to prime sieving.

4.2.2 Implementation

Below is the C implementation demonstrating how the steps interlock:

```c
// sieve_iZ.c
// Segmented Sieve-iZm Algorithm
SieveAlgorithm Sieve_iZm = {sieve_iZm, "Sieve-iZm"};

PRIMES_OBJ *sieve_iZm(uint64_t n)
{
    // 1. Initialization
    size_t x_n = (n + 1) / 6 + 1;

    // Initialize primes array with enough capacity
    PRIMES_OBJ *primes = primes_obj_init(n * .1 + 1000);

    // Memory allocation failed, check logs
    if (primes == NULL)
        return NULL;

    // add 2, 3 to the primes array
    primes_obj_append(primes, 2);
    primes_obj_append(primes, 3);

    // list of small primes to be pre-sieved
    uint64_t s_primes[] = {5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47};

    // Calculate optimal segment size vx for x_n
    int vx_limit = 6; // max number of primes to be pre-sieved
    size_t vx = compute_limited_vx(x_n, vx_limit);
    int start_i = 2; // skip 2, 3

    // Add pre-sieved primes to primes array
    for (int i = 0; i < vx_limit; i++)
    {
        if (vx % s_primes[i] == 0)
        {
            primes_obj_append(primes, s_primes[i]);
            start_i++;
        }
        else
            break;
    }

    // Initialize x5, x7 bitmaps and their clones tmp5, tmp7
    BITMAP *x5, *x7, *tmp5, *tmp7;

    // Allocate memory for x5 and x7 with vx + 10 bits
    x5 = bitmap_create(vx + 10);
    x7 = bitmap_create(vx + 10);

    // 2. Preprocessing:
    // Generate pre-sieved segments of size vx in x5, x7
    construct_iZm_segment(vx, x5, x7);

    // Clone x5, x7 into tmp5, tmp7 for processing
    tmp5 = bitmap_clone(x5);
    tmp7 = bitmap_clone(x7);

    // 3. Process 1st segment to collect enough root primes
    for (uint64_t x = 2; x <= vx; x++)
    {
        if (bitmap_get_bit(tmp5, x)) // implying iZ-[x] is prime
        {
            size_t p = iZ(x, -1);
            primes_obj_append(primes, p);

            // Mark composites of z within this segment if any
            if ((p * p) / 6 < vx)
            {
                bitmap_clear_mod_p(tmp5, p, p * x + x, vx);
                bitmap_clear_mod_p(tmp7, p, p * x - x, vx);
            }
        }

        if (bitmap_get_bit(tmp7, x)) // implying iZ+[x] is prime
        {
            size_t p = iZ(x, 1);
            primes_obj_append(primes, p);

            if ((p * p) / 6 < vx)
            {
                bitmap_clear_mod_p(tmp5, p, p * x - x, vx);
                bitmap_clear_mod_p(tmp7, p, p * x + x, vx);
            }
        }
    }

    // 4. Processing remaining segments:
    int max_y = x_n / vx; // number of segments
    uint64_t limit = vx;  // upper bound for marking composites

    // Sieve and collect primes from the remaining segments
    for (int y = 1; y <= max_y; y++)
    {
        // Reset tmp data for each run
        tmp5 = bitmap_clone(x5);
        tmp7 = bitmap_clone(x7);

        // limit = x_n % vx in the last segment
        if (y == max_y)
            limit = x_n % vx;

        // Mark composites of the rest of root primes in current segment
        for (int i = start_i; i < primes->p_count; i++)
        {
            uint64_t p = primes->p_array[i];

            // Exit loop if p doesn't have composites in this range
            if ((p * p) / 6 > (y * vx + limit))
                break;

            // Locate the 1st composite of p in both iZm5 and iZm7
            uint64_t xp5 = solve_for_x(-1, p, vx, y);
            uint64_t xp7 = solve_for_x(1, p, vx, y);

            // Mark composites of p in the current segment using the Xp-Wheel
            bitmap_clear_mod_p(tmp5, p, xp5, limit);
            bitmap_clear_mod_p(tmp7, p, xp7, limit);
        }

        // Here we can offloaded the sifted BITMAPs to disk for saving results
        // Collect unmarked indices as primes
        for (uint64_t x = 1; x <= limit; x++)
        {
            if (bitmap_get_bit(tmp5, x)) // implying iZ- prime
                primes_obj_append(primes, iZ(x + vx * y, -1));

            if (bitmap_get_bit(tmp7, x)) // implying iZ+ prime
                primes_obj_append(primes, iZ(x + vx * y, 1));
        }
    }

    // 5. Clean up bitmaps and return results
    bitmap_free(x5);
    bitmap_free(x7);
    bitmap_free(tmp5);
    bitmap_free(tmp7);

    // Edge Case: If a prime or more > n got picked up, remove it
    while (primes->p_array[primes->p_count - 1] > n)
        primes->p_count--;

    // Trim unused memory in primes object
    if (primes_obj_resize_to_p_count(primes) < 0)
    {
        // Error: check logs
        primes_obj_free(primes);
        return NULL;
    }

    return primes;
}
```

4.3 Benchmarks

We use the function `benchmark_sieve` (see Appendix A.5) to measure performance at various input limits n. Below is an example invocation to benchmark sieve algorithms from 10^4 to 10^9:

```c
// main.c
int main(void)
{
    SieveAlgorithm sieve_algorithms[] = {
        SieveOfEratosthenes,
        Sieve_iZ,
        SegmentedSieve,
        Sieve_iZm,
    };

    int models_count = sizeof(sieve_algorithms) / sizeof(SieveAlgorithm);
    SieveModels sieve_models = {sieve_algorithms, models_count};

    // Benchmarking sieve algorithms from 10^4 to 10^9
    benchmark_sieve(sieve_models, 10, 4, 9, 1); // testing from 10^4 to 10^9

    return 0;
}
```

After building and running (using the `make` command), detailed results appear under `output/stdout.txt`. Table 4.1 lists the runtime results (in microseconds) of each algorithm for n ranging from 10^4 to 10^9.

4.3.1 Test Results

Algorithm Name	10^4	10^5	10^6	10^7	10^8	10^9
Sieve of Eratosthenes	27	180	1618	15446	138619	2629422
Sieve-iZ	37	124	1024	9595	93204	1460420
Segmented Sieve	58	395	3518	32445	314348	3102601
Sieve-iZm	25	144	1012	8585	83297	757213

Table 4.1: Execution Time Benchmarking for Various Limits

Figure 4.1 presents a visual comparison of these results, normalized to show time-per-input across the range 10^4 to 10^9.

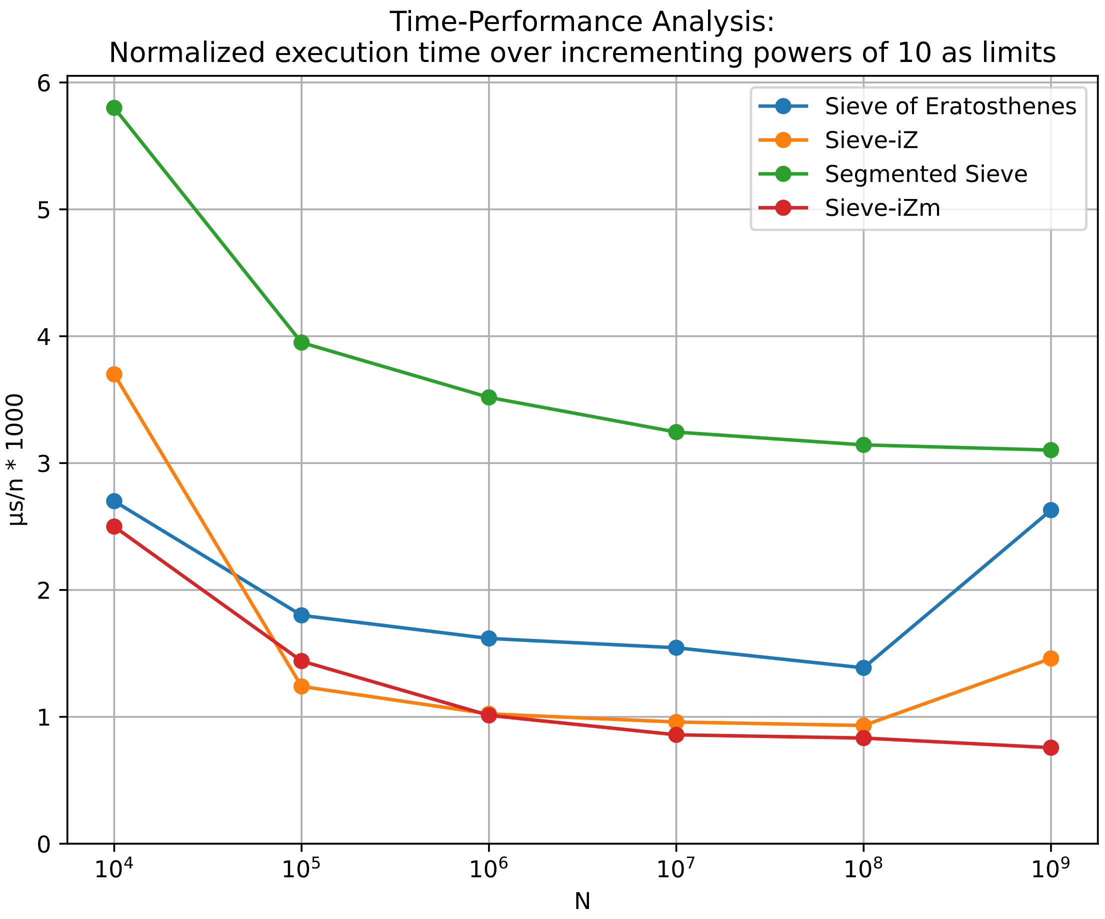

Figure 4.1: Time-per-input analysis for N in range $(10^4 : 10^9)$

4.3.2 More Tests

Limits 10^{10}, 2×10^{10} and 3×10^{10}

We test at larger limits using only Sieve-iZ, Segmented Sieve, and Sieve-iZm (for 10^{10}), then focusing on the two segmented approaches at 2×10^{10} and 3×10^{10}. By invoking the `measure_sieve_time` function as following:

```c
// main.c
int main(void)
{
    // Sieve testing for limit n = 10^10
    measure_sieve_time(Sieve_iZ, int_pow(10, 10));
    measure_sieve_time(SegmentedSieve, int_pow(10, 10));
    measure_sieve_time(Sieve_iZm, int_pow(10, 10));

    // Sieve testing for limit n = 2 * 10^10
    measure_sieve_time(SegmentedSieve, 2 * int_pow(10, 10));
    measure_sieve_time(Sieve_iZm, 2 * int_pow(10, 10));

    // Sieve testing for limit n = 3 * 10^10
    measure_sieve_time(SegmentedSieve, 3 * int_pow(10, 10));
    measure_sieve_time(Sieve_iZm, 3 * int_pow(10, 10));

    return 0;
}
```

Table 4.2 summarizes the results:

Algorithm	**Limit**	**Primes Count**	**Last Prime**	**Time (s)**
Sieve-iZ	10^{10}	455052511	9999999967	18.681585
Segmented Sieve	10^{10}	455052511	9999999967	30.712201
Sieve-iZm	10^{10}	455052511	9999999967	7.531324
Segmented Sieve	2×10^{10}	882206716	19999999967	61.320021
Sieve-iZm	2×10^{10}	882206716	19999999967	15.068850
Segmented Sieve	3×10^{10}	1300005926	29999999993	91.699019
Sieve-iZm	3×10^{10}	1300005926	29999999993	22.483192

Table 4.2: Benchmarking Results for $n = 10^{10}$

4.3.3 Key Observations

From the above runtime measurements, several important trends emerge:

1. **Early-Range Competitiveness**: Up to moderately large inputs (10^6–10^8), Sieve-iZm already competes favorably with standard sieves (e.g., Eratosthenes, Euler, or segmented variations).
2. **Strong Performance at Scale**: Once n surpasses 10^9, Sieve-iZm gains a pronounced lead over classical methods. This is most visible at $n = 10^{10}$ and beyond, where it runs significantly faster than both *Sieve-iZ* and *Segmented Sieve*.

Having outlined these observations, the next section focuses on analyzing why such performance gains materialize and how they compare with theoretical expectations.

4.4 Performance Analysis

4.4.1 Space Complexity

A standout feature of Sieve-iZm is its low, constant auxiliary space usage. Aside from storing final primes (which could be offloaded), working memory consists of just four bitmaps—`x5`, `x7`, `tmp5`, `tmp7`—each capped at $vx6 = 1,616,615$ bits (≈ 0.2 MB). Including small control arrays and metadata, total auxiliary memory is about ≈ 0.8 MB, which remains effectively unchanged as n grows large. This low, constant space complexity effectively addresses the longstanding space-efficiency challenge in prime sieving.

4.4.2 Time Complexity

From the benchmarks, two main design choices drive Sieve-iZm's speed:

1. **Cache-Friendly Segment Size**

   By capping the `BITMAP` objects at a modest size (max $vx6 \approx 0.2$ MB), nearly all marking operations occur within the CPU's cache, minimizing cache misses and ensuring consistently fast memory access.

2. **Skipping Small Primes**

   Any prime p that divides the segment size (vx) is handled once in the pre-sieved base segment. Ensuring vx is divisible by multiple small primes thus eliminates a large fraction of composite-marking overhead.

Beyond the fixed overhead of constructing the base segment, the time complexity is dominated by marking composites of the remaining primes ($p \geq 23$). Per the standard prime-sieve analysis, this leads to approximately

$$\frac{n}{3} \sum_{23 \leq p \leq \sqrt{n}} \frac{1}{p} \;\approx\; O\big(n \log\log n\big) \tag{4.1}$$

While it accurately reflects the overall growth trend ($\log\log n$), it does not fully convey the **practical** performance gains achieved by the lower constant and removing the significant terms of small primes (≤ 19). Theoretically, we can incrementally reduce the sieve work by including more primes in the base segment, however, this sub-linear theoretical complexity comes at the cost of losing cache privileges, which would likely offset any gains in practice.

4.4.3 Considerations for Large Scale

The implementation provided in this chapter is the general algorithm for benchmarking, and suitable for illustrative and educational purposes. For large-scale computations, there are several considerations to apply:

1. **Limiting the required root primes:**

   The required root primes for deterministic sieve grows with ($\log\log n$), which can be prohibitive for very large n.

2. **Offloading the output:**

   While the `primes` array grows slowly with ($\frac{n}{\log n}$), for very large n, we can't keep all primes in memory to be returned at the end. Instead, we can write the results to disk as they are generated or per segment to avoid memory bottlenecks.

In the next chapter, we introduce the *VX6-Sieve* algorithm, which combines deterministic and probabilistic methods to limit the number of root primes required, and provides a standardized output format for saving computational efforts.

Chapter 5

VX6-Sieve: Standardized Segmented Sieve

5.1 Introduction

In earlier chapter, we introduced the *vx6* segment—a cache-friendly, pre-sieved base segment within the iZ-Matrix (iZm)—which possesses the following key features:

- **Segment Composition:** The *vx6* segment is defined as the product of the six primes:

$$5 \times 7 \times 11 \times 13 \times 17 \times 19 = 1616615$$

- **Memory Efficiency:** It uses two bitmaps, each of size 1,616,615 bits, resulting in a total of 3,233,230 bits (approximately 0.4 MB).

- **Coverage:** The segment spans a space of

$$S = 6 \times 1,616,615 = 9,699,690$$

- **Pre-Sieving Benefit:** By pre-sieving out numbers divisible by the primes that constitute *vx6*, only about 17% (1,658,880 candidates out of 9,699,690) remain as potential primes.

In this chapter, we introduce the *VX6-Sieve Algorithm*, a dedicated sieve tool designed for processing an iZ-Matrix segment of horizontal length *vx6*. This algorithm incorporates an efficient encoding mechanism that stores prime gaps instead of full prime numbers, thereby saving on computational resources.

5.1.1 Remaining Challenges for Large-Scale Sieve

1. Unbounded Root Primes

While we have optimized the Xp-Wheel—a specialized tool that targets values congruent to $\pm x_p \pmod{p}$ within the $i\mathbb{Z}$ set and its corresponding X-Arrays—by starting from primes greater than 19, we have effectively reduced the sieve workload. This optimization brings the bit-complexity down to:

$$\frac{n}{3} \sum_{23 \leq p \leq \sqrt{n}} \frac{1}{p}.$$

However, since $\sqrt{n}$ grows unbounded, having to consider all root primes up to $\sqrt{n}$ eventually becomes a performance bottleneck for very large limits. To overcome this, the *VX6-Sieve Algorithm* restricts the deterministic sieving to only those primes up to *vx6* (1,616,615) using the same Xp-Wheel method as in the Sieve-iZm. The remaining candidates beyond this range are then filtered using probabilistic primality testing when necessary. By limiting the deterministic process to root primes satisfying $\frac{vx6}{p} \geq 1$, we ensure that only those primes contributing at least one composite per segment are considered, effectively mitigating the bottleneck.

2. Output Limits

Given the infinitude of primes, storing an exhaustive dataset becomes infeasible as limits increase. To address this challenge, we output the data in an efficient format by segmenting the list of primes and recording the gaps between consecutive primes rather than the primes themselves. This approach fixes the bit-size required per prime to a constant 2 bytes (using `uint_16`), resulting in a sub-linear output size of $O\left(\frac{N}{\log N}\right)$ as the primes become sparser. Consequently, this efficient encoding mechanism enables us to extend our limits further without a prohibitive increase in storage requirements.

5.2 VX Object Structure

We begin by defining the `VX_OBJ` structure and its associated functions for initializing and freeing the structure:

- The `VX_OBJ` structure represents a horizontal segment in the iZ-Matrix. It contains:
  - `vx`, the horizontal vector size,
  - `y`, a pointer to a numeric string, allowing for arbitrary values,
  - `x5` and `x7`, pointers to the bitmaps representing *iZm5* and *iZm7*,
  - a pointer to the `p_gaps` array,
  - `p_count` the number of primes in the segment,
  - and the `SHA256` hash of the `p_gaps` data for verification purposes.
- The `vx6_init` function initializes a new `VX_OBJ` structure using the provided y string.
- The `vx6_free` function frees all memory associated with the `VX_OBJ`.

```
#define VX6_EXT "vx6"                      ///< Extension for vx6 files.
#define P_GAPS_SIZE sizeof(uint16_t)       ///< Size of uint16_t in bytes.

/**
 * @brief Structure representing a horizontal segment in iZm.
 */
typedef struct
{
    size_t vx;                      ///< The horizontal vector size.
    char *y;                        ///< Pointer to y string.
    BITMAP *x5;                     ///< Pointer to x5 BITMAP.
    BITMAP *x7;                     ///< Pointer to x7 BITMAP.
    uint16_t *p_gaps;               ///< Pointer to p_gaps array.
    size_t p_count;                 ///< Number of elements in p_gaps array.
    unsigned char sha256[SHA256_DIGEST_LENGTH]; ///< hash of the p_gaps data.
} VX_OBJ;

/**
 * @brief Initializes a VX_OBJ structure.
 *
 * Allocates and initializes a new VX_OBJ structure using the provided y string
 *
 * @param y A pointer to string used for initialization.
 * @return Pointer to the newly created VX_OBJ, or NULL if allocation fails.
 */
VX_OBJ *vx6_init(char *y);

/**
 * @brief Frees a VX_OBJ structure.
 *
 * Frees all memory associated with the VX_OBJ.
 *
 * @param vx_obj Pointer to the VX_OBJ to free.
 */
void vx6_free(VX_OBJ *vx_obj);
```

5.3 VX6-Sieve Algorithm: Description & Implementation

The *VX6-Sieve* algorithm is a probabilistic sieve that

- Sieves the *vx6* segment in the *iZm* for a specific y.
- Stores prime gaps instead of primes to limit the output size.
- Provides a standardized output format for persisting computational efforts.

5.3.1 Description

It is a two-stages sieve algorithm that combines deterministic and probabilistic approaches to sieve the *vx6* segment.

+ **Parameters:**

Given the following parameters:

- `*vx_obj`, a pointer to the output `VX_OBJ` structure, the caller initializes it with the target y value, and frees it.
- `p_test_rounds`, the number of rounds for primality testing, defaulting to 25 if NULL or 0.
- `*filename`, a pointer to the output file name, or NULL.

The algorithm operates as follows:

1. Initialization:

- Initialize and cache `cached_vx6_primes`, primes up to *vx6* (1616615).
- Initialize and cache the pre-sieved bitmaps `vx_obj->x5` and `vx_obj->x7`.
- Initialize variables y and $yvx = y \times vx6$ as the base value in the segment.
- Compute `upper_limit` $= \sqrt{iZ(vx6 \times (y+1), 1)}$ for root primes that have composites in the segment.
- Initialize flag `is_large_limit = true` to determine if primality testing is required.

2. Deterministic Sieving:

- For each prime p: (skipping the pre-sieved primes less than 23)
  - Locate its first composite points in *iZm5* and *iZm7* using `solve_for_x` methods,
  - mark its composites in `vx_obj->x5` and `vx_obj->x7` using the Xp-Wheel.
  - If $p >$ `upper_limit`, turn off `is_large_limit`, and break the loop.

3. Collect Prime Gaps:

- Initialize a counter `gap = 18` to skip the first 18 numbers, since $x < 4$ are composite columns in *iZm/vx6* (See Figures 3.6).

- Iterate through values x from 4 to `vx6`, incrementing the `gap` counter by total 6 for each iteration:
    - Increment `gap` by 4 for the next prime candidate.
    - If the `vx_obj->x5[x]` bit is set:
        - Initialize a flag `is_prime`, initially set to true.
    - If `is_large_limit` flag is set:
        - Compute $p = iZ(yvx + x, i = -1)$,
        - Apply a Miller-Rabin primality test (with `p_test_rounds` iterations) to p,
        - If p doesn't pass the test, set `is_prime` to false.
    - If `is_prime` still set:
        - Record the current `gap` in `vx_obj->p_gaps[vx_obj->p_count++]`,
        - Reset `gap` to 0.
    - Else, clear the `vx_obj->x5[x]` bit.
    - Increment `gap` by 2.
    - Repeat the same process for `vx_obj->x7[x]` bitmap for the $(i = 1)$ case.

4. Finalization:

- Resize the `p_gaps` array to the count of detected gaps (`vx_obj->p_count`) to free extra space.

- Optionally, if a filename is provided, output the `vx_obj` to a file using `vx6_write_file` function.

- Clean up locally allocated variables.

5.3.2 Implementation

Below is the C implementation of the `vx6_sieve` function:

```c
// Global value defined in utils.h
#define vx6 5 * 7 * 11 * 13 * 17 * 19 // 1616615

/**
 * @brief Sieve all composites within the yth vx6 segment in iZm.
 *
 * Marks all composites within the x5, x7 bitmaps in the vx_obj,
 * uses probabilistic primality testing if needed,
 * and collects prime gaps instead of primes to limit the output size.
 *
 * @param vx_obj Pointer to the VX_OBJ structure.
 * @param p_test_rounds Number of rounds for primality testing.
 * @param filename A pointer to string representing the file name, or NULL.
 */
void vx6_sieve(VX_OBJ *vx_obj, int p_test_rounds, const char *filename)
{
    // caches primes < vx6 and the pre-sieved bitmaps
    cache_primes_obj();
    cache_vx6_bitmaps();
    // clone the pre-sieved cached bitmaps
    vx_obj->x5 = bitmap_clone(cached_x5);
    vx_obj->x7 = bitmap_clone(cached_x7);

    // count of operations, for performance analysis
    int mark_ops = 0;   // count of mark operations
    int p_test_ops = 0; // primality test operations

    // initialize mpz_t y from vx_obj->y
    mpz_t y;
    mpz_init(y);
    mpz_set_str(y, vx_obj->y, 10);

    // compute yvx = y * vx6
    mpz_t yvx;
    mpz_init(yvx);
    mpz_mul_ui(yvx, y, vx6);

    // compute upper_limit = sqrt(iZ(vx6 * (y+1), 1))
    mpz_t upper_limit;
    mpz_init(upper_limit);
    mpz_set(upper_limit, yvx);
    mpz_add_ui(upper_limit, upper_limit, vx6);
    iZ_gmp(upper_limit, upper_limit, 1);
    mpz_sqrt(upper_limit, upper_limit);

    // initialize GMP reusable variables p, x_p
    mpz_t p, x_p;
    mpz_init(p);
    mpz_init(x_p);

    // flag to determine if probabilistic primality test is needed
    int is_large_limit = 1;

    int start_i = 8; // to skip primes < 23
    // mark composites of cached primes skipping start_i
    for (int i = start_i; i < cached_vx6_primes->p_count; i++)
    {
        // break if p > upper_limit
        mpz_set_ui(p, cached_vx6_primes->p_array[i]);
        if (mpz_cmp(p, upper_limit) > 0)
        {
            // turn off is_large_limit, results are deterministic
            is_large_limit = 0;
            break;
        }

        int int_p = cached_vx6_primes->p_array[i];
        int int_xp;

        int_xp = solve_for_x_gmp(-1, int_p, vx6, y);
        bitmap_clear_mod_p(vx_obj->x5, int_p, int_xp, vx6);
        int_xp = solve_for_x_gmp(1, int_p, vx6, y);
        bitmap_clear_mod_p(vx_obj->x7, int_p, int_xp, vx6);

        mark_ops += 2 * vx6 / int_p;
    }

    if (!p_test_rounds)
        p_test_rounds = 25; // default value if null or 0

    int gap = 18; // skipping 3 * (4 + 2) = 18, as x < 3 can't be prime
    // perform probabilistic sieve for remaining candidates if is_large_limit
    for (int x = 4; x <= vx6; x++)
    {
        gap += 4;

        // check if iZ(x + vx6 * y, -1) is prime, if not clear x in x5
        if (bitmap_get_bit(vx_obj->x5, x))
        {
            int is_prime = 1;

            if (is_large_limit)
            {
                // compute x_p = x + vx6 * y
                mpz_add_ui(x_p, yvx, x);
                iZ_gmp(p, x_p, -1); // compute p = iZ(x_p, -1)
                is_prime = mpz_probab_prime_p(p, p_test_rounds);
                p_test_ops++;
            }

            if (is_prime)
            {
                vx_obj->p_gaps[vx_obj->p_count++] = gap;
                gap = 0; // reset gap
            }
            else
            {
                bitmap_clear_bit(vx_obj->x5, x);
            }
        }

        gap += 2;

        // same for iZ+
        if (bitmap_get_bit(vx_obj->x7, x))
        {
            int is_prime = 1;

            if (is_large_limit)
            {
                mpz_add_ui(x_p, yvx, x);
                iZ_gmp(p, x_p, 1); // compute p = iZ(x_p, 1)
                is_prime = mpz_probab_prime_p(p, p_test_rounds);
                p_test_ops++;
            }

            if (is_prime)
            {
                vx_obj->p_gaps[vx_obj->p_count++] = gap;
                gap = 0; // reset gap
            }
            else
            {
                bitmap_clear_bit(vx_obj->x7, x);
            }
        }
    }

    // resize p_gaps array to fit the actual count
    vx6_resize_p_gaps(vx_obj);

    // write p_gaps to a file if filename is provided
    if (filename)
        vx6_write_file(vx_obj, filename);

    // clear GMP variables
    mpz_clear(p);
    mpz_clear(x_p);
    mpz_clear(y);
    mpz_clear(yvx);
    mpz_clear(upper_limit);

    // print sieve statistics
    print_line(92);
    printf("Sieve Operation Statistics");
    print_line(92);
    printf("#marking operations: %d\n", mark_ops);
    printf("#primality testing operations: %d\n", p_test_ops);
    printf("#p_count: %d\n", vx_obj->p_count);
    print_line(92);
}
```

5.4 Testing & Performance Analysis

We evaluated the *VX6-Sieve Algorithm* using a range of y values and verified the results with the Miller–Rabin primality test. The algorithm successfully sieves the *vx6* segment and collects prime gaps for the remaining candidates. The resulting output files are saved in a standardized format that records prime gaps rather than the full prime numbers, thus greatly reducing storage requirements.

5.4.1 `test_vx6_sieve` Function

For testing the *VX6-Sieve Algorithm*, we implemented the `test_vx6_sieve` function. This function performs as documented below:

```
/**
 * test_vx6_sieve - Tests the sieving functionality of the vx6 algorithm.
 *
 * This function performs the following steps:
 *   - Initializes a VX_OBJ structure using the provided string 'y'.
 *   - Measures the execution time of the vx6_sieve function.
 *   - Prints sieve statistics including identifier, prime count,
 * and execution time.
 *   - Computes a base value using GMP arithmetic and verifies the
 * first 10 primes by incrementing the base with each prime gap and
 * validating primality with the Miller-Rabin test.
 *   - Outputs prime gap details and their validation status.
 *
 * Parameters:
 *   @y: A character pointer representing a numeric string used to
 * initialize the VX_OBJ.
 *
 * Returns:
 *   An integer value indicating the validity of the prime gaps:
 *     - 1 if all tested prime gaps are correct.
 *     - 0 if any tested prime gap fails the primality test.
 */
int test_vx6_sieve(char *y);
```

Usage

In the entry point *main.c*, the *VX6-Sieve Algorithm* is benchmarked as follows:

```
// main.c
/**
 * @brief Tests the vx6 sieve implementation.
 *
 * This function benchmarks the vx6 sieve algorithm by repeatedly testing it
 * with progressively larger input sizes.
 * It starts with an initial buffer value ("1000") and, in each iteration,
 * multiplies the buffer by 10^9 to increase the test size, invoking the
 * test_vx6_sieve function on each modified buffer.
 */
void testing_vx6_sieve(void)
{
    print_line(92);
    printf("Testing vx6 sieve");
    print_line(92);
    char buf[256] = "1000";
    for (int i = 0; i < 10; i++)
    {
        test_vx6_sieve(buf);
        strcat(buf, "000000000"); // multiply by 10^9
    }
}

int main(void)
{
    // Initialize the vx6 sieve testing
    testing_vx6_sieve();

    return 0;
}
```

This test, invokes the `test_vx6_sieve` function with y values constructed as follows:

Where $k = 10^3$, and $g = 10^9$, $y = k \times g^i$ for i in the range [0, 9].

The printed results are summarized in the tables below:

5.4.2 Sieve Workload & Execution Time Analysis

Table 5.1 shows the execution time in seconds, the number of bitwise operations, and primality testing operations for each y value:

Y	**Execution Time**	**#Bitwise Ops**	**#Primality Testing Ops**
k	0.032003	4031925	0
$k \cdot g$	1.059171	4683431	382128
$k \cdot g^2$	2.509662	4683431	380968
$k \cdot g^3$	2.811812	4683431	380293
$k \cdot g^4$	4.342190	4683431	380626
$k \cdot g^5$	4.592591	4683431	380530
$k \cdot g^6$	5.845369	4683431	380752
$k \cdot g^7$	6.255784	4683431	381379
$k \cdot g^8$	8.821161	4683431	380196
$k \cdot g^9$	9.408612	4683431	380530

Table 5.1: VX6-Sieve Algorithm Performance Analysis

Key Observations

While each $vx6$ segment covers a space ($S = 6 \times 1,616,615 = 9,699,690$) of natural numbers:

- The algorithm performs a maximum of 4,683,431 bitwise operations per segment, which corresponds to roughly $O(\frac{1}{2} \cdot S)$.
- The maximum number of primality testing operations is approximately 383,000, which is about $O(\frac{4}{100} \cdot S)$.
- Execution time increases with larger limits, primarily due to the growing bit-size of input candidates subjected to probabilistic primality testing.

These observations indicate that the sieve work over each *vx6* segment is bounded, leading to a linear bit-complexity of $O(S)$ per segment regardless of the overall limit. Notably, for $y = 1000$, the algorithm performs 4,031,925 bitwise operations in approximately 0.032 seconds without any primality testing, highlighting that the additional execution time for larger limits stems chiefly from the increased workload of primality testing.

5.4.3 Prime Gaps Arrays

Table 5.2 shows the first 10 prime gaps collected from the test above:

i	**First 10 Prime Gaps**
0	[28, 32, 6, 36, 4, 24, 6, 26, 16, 50, ...]
1	[28, 38, 42, 28, 30, 44, 22, 6, 18, 12, ...]
2	[112, 44, 42, 58, 92, 10, 164, 96, 48, 76, ...]
3	[30, 82, 54, 170, 60, 4, 276, 176, 156, 60, ...]
4	[78, 430, 98, 10, 60, 186, 74, 46, 66, 102, ...]
5	[88, 14, 6, 42, 82, 156, 68, 6, 28, 38, ...]
6	[30, 76, 24, 32, 100, 90, 104, 400, 162, 300, ...]
7	[46, 6, 18, 450, 86, 126, 166, 20, 114, 250, ...]
8	[310, 38, 130, 710, 130, 114, 66, 390, 284, 174, ...]
9	[196, 252, 294, 350, 330, 60, 108, 46, 102, 90, ...]

Table 5.2: First 10 Prime Gaps for $y = k \cdot g^i$ in *iZm/vx6*

To reconstruct the list of primes within a *vx6* segment, we first compute the base value:

$$\text{base} = iZ(y \times vx6, 1).$$

Then, starting with the first prime gap, we increment the base by each gap value sequentially to generate subsequent primes.

Examples

– For $y = 1000$:

`base = iZ(1000 * 1616615, 1) = 9699690001`

- The 1st prime is $9699690001 + 28 = 9699690029$.
- The 2nd prime is $9699690029 + 32 = 9699690061$.
- The 3rd prime is $9699690061 + 6 = 9699690067$.
- ...

– For $y = 1000 \times 10^9$:

`base = iZ(1000 *` 10^9 `* 1616615, 1) = 9699690000000000001`

- The 1st prime is $9699690000000000001 + 28 = 9699690000000000029$.
- The 2nd prime is $9699690000000000029 + 38 = 9699690000000000067$.
- The 3rd prime is $9699690000000000067 + 42 = 9699690000000000109$.
- ...

– This process is repeated for the remaining y values.

5.4.4 Prime Gaps Statistics

Table 5.3 shows the statistics for the prime gaps collected by the *VX6-Sieve Algorithm* for each $y = k \cdot g^i$:

i	**#(Primes)**	**#(Twins)**	**#(Cousins)**	**#(Sexy)**
0	422176	24220	24076	48817
1	221579	6720	6711	13233
2	150240	3086	3086	6232
3	113783	1781	1792	3538
4	91633	1139	1177	2354
5	76271	786	800	1533
6	65802	621	613	1136
7	58176	428	478	867
8	51527	358	348	730
9	46333	302	314	594

Table 5.3: Prime Gaps Statistics for $y = k \cdot g^i$ in *iZm/vx6*

Observations

- The number of *Twin Primes* and *Cousin Primes* are very close to each other, and even matched in the case $i = 2$, both are 3086.
- The number of *Sexy Primes* is close their sum (*Twins* + *Cousins*).

Note 5.4.1. These sample results not only validate the performance of the *VX6-Sieve* as a technical tool but also demonstrate its potential utility in prime research and statistical analysis. However, we are more focused on the technical aspects of the algorithm and its performance.

5.5 Writing & Reading VX6 Files

Efficient file I/O is crucial for persisting the computational output of the *VX6-Sieve Algorithm.* To facilitate data archiving and subsequent analysis, we implement functions that serialize and deserialize the `VX_OBJ` structure to and from binary files. These functions ensure that all essential data—such as the base string y, the number of prime gaps, the gap data itself, and a `SHA256` hash for integrity verification—are accurately stored and later retrieved.

5.5.1 Writing VX6 Files

The `vx6_write_file` function serializes a given `VX_OBJ` structure to a binary file. Specifically, it writes:

- The length of the `vx_obj->y` string (including the null terminator), followed by the string itself.
- The `p_count` value indicating the number of elements in the `vx_obj->p_gaps` array.
- The `vx_obj->p_gaps` array.
- The `SHA256` hash computed over the `p_gaps` array to ensure data integrity.

This standardized output format guarantees that the stored data remains compact and consistent, with the hash serving as a verification mechanism during file read operations.

```
/**
 * vx6_write_file - Write a VX_OBJ structure to a binary file.
 *
 * Parameters:
 * @vx_obj: Pointer to a VX_OBJ structure containing data to be written.
 * @filename: The path of the file to write to.
 *
 * Return: Returns 1 on successful write, and 0 if any error occurs.
 */
int vx6_write_file(VX_OBJ *vx_obj, const char *filename);
```

5.5.2 Reading VX6 Files

The complementary function, `vx6_read_file`, deserializes a `VX_OBJ` structure from a binary file. During this process, it:

- Reads the length of the y string, allocates the appropriate memory, and then reads the string.
- Retrieves the `p_count` value to determine the number of elements in the prime gaps array.
- Reads the `p_gaps` array into the allocated space.
- Reads the stored `SHA256` hash and recomputes a hash over the retrieved `p_gaps` array, comparing the two to validate data integrity.

This careful reconstruction ensures that the `VX_OBJ` structure is accurately restored and that the data has not been corrupted.

```
/**
 * vx6_read_file - Read a VX_OBJ structure from a binary file.
 *
 * @vx_obj: Pointer to a VX_OBJ structure where the read data will be stored.
 * @filename: The file path to read from.
 *
 * Return: Returns 1 on successful write, and 0 if any error occurs.
 */
int vx6_read_file(VX_OBJ *vx_obj, const char *filename);
```

5.5.3 Testing & Usage

To verify the correctness of the file I/O functions, we implemented `test_vx6_file_io`. This function performs the following steps:

- Initializes a `VX_OBJ` structure using a provided y string.
- Invokes the `vx6_sieve` function to populate the object with computed data.
- Writes the populated `VX_OBJ` to a binary file using `vx6_write_file`.
- Reads the data back into a new `VX_OBJ` structure using `vx6_read_file`.
- Prints basic information (such as the y string and prime gap count) to confirm the successful round-trip.

The testing is further integrated into the entry point (`main.c`) to run multiple test rounds with progressively larger y values.

```
/**
* test_write_read_vx6_file - Tests the process of writing and reading a vx6
* object to/from a file.
```

```
*
* Parameters:
*   @filename: A pointer to the file path where the vx6 object is
* stored and read from.
*   @y:         A character pointer representing a numeric y striking
*
* Returns:
*   An integer value to indicate success or failure:
*     - 1 if both the write and read operations are successful.
*     - 0 if either the write or read operation fails.
*/
int test_vx6_file_io(const char *filename, char *y);
```

Then, in the entry point *main.c*, we performed a test with the same values of y used in the previous test.

```
/**
 * @brief Tests the vx6 I/O operations.
 *
 * This function verifies the I/O operations related to the vx6 implementation.
 * It does so by testing the writing and reading of files using the
 * test_vx6_file_io function over increasingly larger file sizes. The file
 * size is adjusted by appending additional zeros to a buffer that starts with
 * the value "1000".
 */
void testing_vx6_io(int test_rounds)
{
    print_line(92);
    printf("Testing vx6 IO operations");
    print_line(92);

    char buf[256] = "1000";
    char filename[256];

    for (int i = 0; i < test_rounds; i++)
    {
        sprintf(filename, "%s/test_%d", DIR_iZm, i);
        test_vx6_file_io(filename, buf);
        strcat(buf, "000000000");
    }
}

// Entry point
int main(void)
{
    // Testing vx6 IO operations with 10 test rounds
    testing_vx6_io(10);

    return 0;
}
```

Results

The output files are stored in the default directory (`/output/iZm/`). Table 5.4 summarizes the file sizes for different test rounds:

i	**Filename**	**File Size**
0	test_0.vx6	844 KB
1	test_1.vx6	443 KB
2	test_2.vx6	301 KB
3	test_3.vx6	228 KB
4	test_4.vx6	183 KB
5	test_5.vx6	153 KB
6	test_6.vx6	132 KB
7	test_7.vx6	116 KB
8	test_8.vx6	103 KB
9	test_9.vx6	93 KB

Table 5.4: VX6 Files Size Analysis

Key Observation: By encoding prime gaps using `uint_16` (2 bytes per gap) rather than storing full primes, the file size decreases as the density of primes decreases. This efficient encoding facilitates scaling to higher limits.

5.6 Conclusions & Directions

The *VX6-Sieve Algorithm* offers a powerful, scalable method for sieving within a *vx6* segment of the iZ-Matrix for any given y. Its standardized output format—which encodes prime gaps instead of full primes—ensures that the data is both compact and easily shareable for further analysis.

Constant Space Complexity $O(1)$

The algorithm requires a fixed amount of memory ($2 \times \text{vx6} = 3,233,230$ bits) to sieve a segment covering

$$S = 6 \times \text{vx6} = 9,699,690 \text{ natural numbers},$$

resulting in a constant space complexity of $O(1)$.

Linear Bit-Complexity $O(n)$

Each *vx6* segment is processed with a bounded number of operations:

- Approximately $\frac{1}{2}S$ rapid bitwise operations, which takes about 30 milliseconds on humble hardware.
- Roughly $\frac{4}{100}S$ primality testing operations.

Thus, the overall bit-complexity per segment is linear $O(n)$, with highly favorable constant factors.

Standardized Output Format

The technique of encoding prime gaps fixes the output size per prime to 2 bytes (via `uint_16`), providing a sub-linear output size of

$$O\left(\frac{S}{\log S}\right)$$

as the primes become sparser. This efficient encoding is crucial for maintaining manageable output sizes even as the limits increase.

Uncertainty in Primality Testing

The algorithm employs the probabilistic Miller–Rabin test for prime verification, introducing a small error probability. This error rate can be minimized by increasing the number of testing rounds (with the default set to 25 rounds), albeit at the cost of increased execution time. The uncertainty for a prime p is estimated as:

$$1 - \left(1 - \frac{1}{4^{25}}\right)^p.$$

5.6.1 Directions

Based on our testing results (see Table 5.1), the primary factor affecting execution time is the cost of primality testing, which increases with the limit. In contrast, the bitwise operations, despite some redundancy, remain fast and bounded. Future improvements in the efficiency of primality testing or the strategic use of additional root primes could further reduce the number of necessary tests, thereby enhancing the overall time performance of the algorithm.

Chapter 6

Random-iZprime Algorithm

6.1 Introduction

In this chapter, we introduce the *Random-iZprime Algorithm*, a method designed to efficiently generate random, large primes. Building on the foundational concepts of the iZ-Framework presented in earlier chapters, Random-iZprime leverages the geometric properties of the iZ-Matrix to drastically narrow the search space by filtering out the frequent small primes. This focused approach allows the algorithm to quickly identify large primes, making it particularly well-suited for cryptographic applications where minimizing prime-generation latency is critical.

The algorithm is characterized by the following features:

1. **Probabilistic with High Assurance**

   It utilizes GMP's Miller–Rabin-based routines for primality testing, striking a practical balance between speed and accuracy for large numbers. For applications requiring absolute certainty, deterministic tests can be substituted.

2. **Efficient & Scalable**

   Designed with parallel execution in mind, the algorithm can run on multiple cores, achieving near-linear speedup by distributing independent prime searches.

3. **Flexible**

   Its modular codebase allows for straightforward integration and customization to meet the diverse requirements of various applications.

In the remainder of this chapter, we briefly review the iZ-based theory that underpins Random-iZprime, emphasizing how discarding composite residues effectively narrows the candidate search space. We then describe the construction of the iZ-Matrix using larger *vx* vectors to target higher bit lengths—thus further reducing the number of candidate columns. Finally, we present the C implementation of the algorithm and benchmark its performance against other standard prime-generation libraries.

6.2 Theoretical Basis

Review from Chapter 1: We established that all primes greater than 3 must be of the form $6x \pm 1$. As illustrated in Figure 1.1, when integers are arranged in rows of 6, multiples of 2, 3, and 6 align in predictable patterns, leaving only the positions corresponding to $6x + 1$ and $6x - 1$ as potential prime candidates. This observation immediately eliminates $\frac{2}{3}$ of all integers from consideration.

Refinement in Chapter 3: The iZ-Matrix further refines the $i\mathbb{Z}$ space by arranging each $i\mathbb{Z}$ subset into a two-dimensional grid. Here, the parameter vx—defined as the product of the most frequently occurring primes in the iZ-set—serves as the horizontal vector size. In this configuration, the composites associated with any prime that divides vx align vertically. Consequently, any prime in the iZ-Matrix (apart from those primes that divide vx) must have the form

$$iZ\big(x + vx \times y,\ \pm 1\big), \quad \text{where } \gcd(x, vx) = 1.$$

The iZ function inherently ensures that candidates are not divisible by 2 or 3, and the condition $\gcd(x, vx) = 1$ further excludes numbers divisible by any prime factor of vx. This filtering reduces the search space to only those columns corresponding to co-prime values.

For example, as shown in Figure 3.6, choosing

$$vx6 = 5 \times 7 \times 11 \times 13 \times 17 \times 19,$$

effectively halves the search space by eliminating all columns that share factors with $vx6$. More extensive constructions of vx can further diminish the composite columns, thereby increasing the density of primes among the remaining candidates.

Random-iZprime exploits these structural properties—specifically, the elimination of composite columns and the focus on co-prime "x" values—to efficiently select random prime candidates at large bit sizes. By starting from a randomly chosen base where $\gcd(x, vx) = 1$ and iterating over the parameter y, the algorithm navigates the iZ-Matrix in search of a prime.

In the following sections, we demonstrate how these theoretical insights culminate in a fast random-prime search engine, complete with parallel processing capabilities for enhanced speed and scalability.

6.3 Description & Implementation

6.3.1 Algorithm Overview

For *flexibility* and *modularity*, we break down the Random-iZprime algorithm into a set of functions, each handling a distinct part of the prime-generation process. The primary entry point, shown below, takes the target bit size, number of primality test rounds, and the number of cores to use:

```
// Random iZprime algorithm signature
void random_iZprime(
    mpz_t p,                    // Output: Random prime candidate
    int p_id,                   // Matrix ID -1 for (6x - 1) or 1 for (6x + 1)
    int bit_size,               // Bit-size of the prime
    int primality_check_rounds, // Number of Miller-Rabin testing rounds
    int cores_num               // Number of cores to use
);
```

Given these parameters, the algorithm generates random primes by performing the following steps:

1. **Compute vx**

   Based on the requested bit size, calculate an maximum vx for the given bit-size. This defines the "horizontal" dimension of the iZ-matrix and determines which residues are skipped.

2. **Set Random Base Value**

   Select a random column at x such that $\gcd(x,\ vx) = 1$, ensuring that $iZ(x + vx \times y,\ matrix\_id)$ can yield primes, rather than hitting composite columns.

3. **Search for a Prime**

   Increment y from a starting point (e.g., $y = 1$) while testing $iZ(x + vx \times y,\ matrix\_id)$ for primality. The process repeats until a prime is discovered.

4. **Parallelization**

   If multiple cores are available, each core (or child process) runs an independent prime search with a different seed for randomness. The first process to find a prime sends it back to the parent, which then terminates the remaining children.

6.3.2 Implementation Details

Dependencies

```
// random_iZprime.c
#include <iZ.h>        // For iZ framework and necessary includes
#include <sieve.h>     // For sieve_iZ
#include <signal.h>    // For kill
#include <sys/wait.h> // For waitpid

// Global PRIMES_OBJ cache
static PRIMES_OBJ *cached_primes = NULL;

// Cache primes for random_iZprime
static void cache_primes(void)
{
    if (cached_primes == NULL)
    {
        cached_primes = sieve_iZ(int_pow(10, 4));
    }
}
```

Here, we see:

- iZ.h and sieve.h are included for iZ-related functions.
- signal.h is used for process signaling.
- sys/wait.h is used for process management.
- cached_primes is a global pointer to store the prime cache.
- cache_primes initializes the prime cache using sieve_iZ.

GMP Random Seeding

```
// GMP random seeding
void gmp_seed_randstate(gmp_randstate_t state)
{
    unsigned long seed;

    // Open /dev/urandom for reading
    int random_fd = open("/dev/urandom", O_RDONLY);
    if (random_fd == -1)
    {
        // Fallback if /dev/urandom cannot be opened
        seed = (unsigned long)time(NULL);
    }
    else
    {
        // Read random bytes from /dev/urandom
        read(random_fd, &seed, sizeof(seed));
        close(random_fd);
    }

    gmp_randseed_ui(state, seed); // Seed the state with the random value
}
```

This function seeds GMP's random state with entropy from `/dev/urandom` (or a fallback if unavailable), ensuring genuine randomness for prime searches.

Constructing the Horizontal Vector Size *vx*

```c
// Compute closest vx to the given bit-size
void gmp_compute_vx(mpz_t vx, int bit_size)
{
    // Ensure cached_primes is set for the computation
    cache_primes();

    int i = 2;                                 // to skip 2, 3
    mpz_set_ui(vx, cached_primes->p_array[i]); // set vx = 5

    // Multiply vx by next prime p while vx * p < 2^bit_size
    while (mpz_sizeinbase(vx, 2) < bit_size)
    {
        i++;
        mpz_mul_ui(vx, vx, cached_primes->p_array[i]);
    }

    // Divide by the last prime that exceeded the bit size
    mpz_div_ui(vx, vx, cached_primes->p_array[i]);
}
```

We build *vx* by iteratively multiplying small primes until we approach $2^{\text{bit\_size}}$. Then, we undo the last factor if it crosses the desired threshold.

Setting Random Base Value

```c
// Generate random base value for p
// which by incrementing by vx can yield primes
void set_random_base(mpz_t p, int matrix_id, mpz_t vx)
{
    gmp_randstate_t state;
    gmp_randinit_default(state);
    gmp_seed_randstate(state); // seed random state

    // create mpz_t tmp variable to hold values
    mpz_t tmp;
    mpz_init(tmp);

    // tmp = random_x in vx range
    mpz_urandomm(tmp, state, vx);

    // initialize p with the random_x value tmp
    iZ_gmp(p, tmp, matrix_id);

    // find next x co-prime with vx
    for (int i = 1; i < 10000; i++)
    {
        // increment p by 6 to increment x by 1
        mpz_add_ui(p, p, 6);

        // Compute tmp = gcd(vx, p)
        mpz_gcd(tmp, vx, p);

        // break if gcd(vx, p) = 1, implying current x value can yield primes
        // of the form iZ(x + vx * y, matrix_id)
        if (mpz_cmp_ui(tmp, 1) == 0)
            break;
    }

    // set p = iZ(random_x + vx, matrix_id) to skip first row in iZm
    mpz_add(p, p, vx);

    // cleaning up
    mpz_clear(tmp);
    gmp_randclear(state);
}
```

We pick a random x value, ensure co-primality with vx, and shift by vx to position ourselves at [`random_x`, 1] in the matrix. This sets an initial candidate for prime testing.

Search for a Prime in iZ-Matrix

```
// Search for a random prime in the iZ-Matrix
void search_p_in_iZm(mpz_t p, int matrix_id, mpz_t vx, int primality_check_rounds)
{
    mpz_t tmp;
    mpz_init(tmp);

    // set random x and y values for p = iZ(x + vx * y),
    // such that p and vx are co-primes
    set_random_base(tmp, matrix_id, vx);
    mpz_set_ui(p, 0);

    int attempts_limit = 10000;

    for (int i = 0; i < attempts_limit; i++)
    {
        mpz_add(tmp, tmp, vx); // increment y

        // break if tmp is prime and set p = tmp
        if (mpz_probab_prime_p(tmp, primality_check_rounds))
        {
            mpz_set(p, tmp);
            break;
        }
    }

    if (mpz_cmp_ui(p, 0) == 0)
    {
        log_debug("No Prime Found :\\\n");
        search_p_in_iZm(p, matrix_id, vx, primality_check_rounds);
    }

    mpz_clear(tmp);
}
```

We increment y, testing each new candidate with the chosen primality routine, stopping upon success or if we hit the retry threshold.

Putting it all Together

```c
// Random iZprime algorithm: Main entry point
void random_iZprime(mpz_t p, int p_id, int bit_size,
    int primality_check_rounds, int cores_num)
{
    // 1. Compute vx for the given bit-size
    mpz_t vx;
    mpz_init(vx);
    gmp_compute_vx(vx, bit_size);

    // 2. If only one core, run the search in-process
    if (cores_num < 2)
    {
        search_p_in_iZm(p, p_id, vx, primality_check_rounds);
        return;
    }

    // 3. Else, fork multiple processes to search for a prime
    // Create a pipe for inter-process communication.
    int fd[2];
    if (pipe(fd) == -1)
    {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    pid_t pids[cores_num];

    // Fork child processes.
    for (int i = 0; i < cores_num; i++)
    {
        pid_t pid = fork();
        if (pid < 0)
        {
            perror("fork");
            exit(EXIT_FAILURE);
        }
        else if (pid == 0)
        {
            // Child process: close the read-end.
            mpz_t local_candidate;
            mpz_init(local_candidate);

            // Search for a candidate prime.
            search_p_in_iZm(local_candidate, p_id, vx, primality_check_rounds);

            // If a candidate is found, send it via the pipe.
            if (mpz_cmp_ui(local_candidate, 0) != 0)
            {
                char *candidate_str = mpz_get_str(NULL, 10, local_candidate);
                if (candidate_str != NULL)
                {
                    write(fd[1], candidate_str, strlen(candidate_str) + 1);
                    free(candidate_str);
                }
            }
            mpz_clear(local_candidate);
            close(fd[1]);
            exit(0);
        }
        else
        {
            // Parent process saves child's PID.
            pids[i] = pid;
        }
    }

    // 4. Parent reads first result from the pipe.
    close(fd[1]);
    char buf[bit_size / 3];
    memset(buf, 0, bit_size);

    ssize_t n = read(fd[0], buf, sizeof(buf));
    if (n > 0)
    {
        if (mpz_set_str(p, buf, 10) != 0)
            fprintf(stderr, "Error converting string to mpz_t\n");
    }
    close(fd[0]);

    // 5. Terminate all child processes
    for (int i = 0; i < cores_num; i++)
    {
        kill(pids[i], SIGTERM);     // Terminate child process
        waitpid(pids[i], NULL, 0); // Wait for child process to terminate
    }

    // Cleanup
    mpz_clear(vx);
}
```

- **Single-Core Mode**: Simply calls `search_p_in_iZm` in the main process.
- **Multi-Core Mode**: Forks child processes, each independently searching for a random prime. The first found prime is written to a pipe and read by the parent, which then kills the remaining children.

This parallel search approach often yields near-linear speedups if each child finds a prime candidate in roughly the same time frame.

In the next section, we benchmark the Random-iZprime algorithm against random prime generation functions in the GMP and OpenSSL libraries, highlighting how multi-core concurrency and iZ-based candidate reduction deliver efficient and scalable performance.

6.4 Benchmarks & Performance Analysis

This section evaluates the performance of **Random-iZprime** in generating large random primes, comparing it with two widely used alternatives. We consider both **single-core** and **multi-core** scenarios to highlight the parallel speedups achievable with Random-iZprime.

6.4.1 Benchmark Elements

We tested each algorithm by generating 10 random primes at various bit sizes (ranging from 1024 bits up to 8192 bits). To minimize one-off variations, each experiment was repeated multiple times, and we took the average run time. Within Random-iZprime, we considered three configurations:

1. **Single-Core** (comparable to the single-threaded runs in other libraries),
2. **4-Core**,
3. **8-Core**.

For the other libraries, we used:

- **GMP**'s `mpz_nextprime`, invoked on a random seed (version 6.3.0),
- **OpenSSL**'s `BN_generate_prime` (version 3.0.15).

All algorithms used the same **primality testing routine** (25 rounds of Miller–Rabin).

All benchmarks were conducted on the same hardware and software environment to ensure consistency. By varying the number of cores for Random-iZprime, we aim to quantify the performance gains from parallel prime search, particularly at higher bit lengths where prime density is lower.

Test Bit-Sizes

We chose **1024, 2048, 4096,** and **8192** bits, which are typical in cryptographic applications (e.g., RSA key generation). These sizes capture both smaller primes (1024 bits) and significantly larger primes (8192 bits) that push the limits of many libraries.

Results Reproducibility

Because prime generation is inherently random, the exact numeric results may vary upon re-running these tests. Nevertheless, our benchmarking tools—briefly documented in Appendix A.6—enable you to replicate these experiments on your own machine. While absolute times may differ due to hardware and random seeds, the relative performance trends across GMP, OpenSSL, and Random-iZprime should remain broadly consistent.

6.4.2 Benchmarking

Using the `benchmark_random_prime_algorithms` function (more details in the Appendix A.6), below is a sample main function illustrating how to run the tests for various bit lengths, saving the results to a file for further analysis.

```c
// main.c

/**
 * @brief Benchmarks random prime generation algorithms.
 *
 * This function benchmarks the performance of random prime generation
 * algorithms for various bit-sizes. It tests for 1 KB, 2 KB, 4 KB, and 8 KB
 * sizes (where 1 KB is defined to be 1024 bits) and performs a fixed
 * number of rounds for both primality checks and overall testing. The
 * results can be optionally saved.
 *
 * Local Parameters:
 * - Kb: Represents one kilobyte in terms of bit-size (1024 bits).
 * - primality_check_rounds: Number of rounds to perform for primality
 * testing (set to 25).
 * - test_rounds: Number of test iterations per bit-size (set to 5).
 * - save_results: Flag to determine if the results should be saved
 * (0 indicates they should not be saved).
 */
void testing_prime_gen_algorithms(void)
{
    print_line(92);
    printf("Testing random prime generation algorithms");
    print_line(92);
    // Random prime generation testing
    int Kb = 1024;
    int primality_check_rounds = 25;
    int test_rounds = 5;
    int save_results = 0;

    // Benchmark random prime generation algorithms for 1 KB bit-size
    benchmark_random_prime_algorithms(1 * Kb, primality_check_rounds,
        test_rounds, save_results);

    // Benchmark random prime generation algorithms for 2 KB bit-size
    benchmark_random_prime_algorithms(2 * Kb, primality_check_rounds,
        test_rounds, save_results);

    // Benchmark random prime generation algorithms for 4 KB bit-size
    benchmark_random_prime_algorithms(4 * Kb, primality_check_rounds,
        test_rounds, save_results);

    // Benchmark random prime generation algorithms for 8 KB bit-size
    benchmark_random_prime_algorithms(8 * Kb, primality_check_rounds,
        test_rounds, save_results);
}

// Entry point
int main(void) // Entry point
{
    testing_prime_gen_algorithms();
    return 0;
}
```

Running the above code automatically prints results to `output/stdout.txt` and saves detailed logs to a file named `random_prime_results_<timestamp>.txt` within the output directory.

Plotting the Results

For plotting the results saved in the output directory, we provide a Python script `plot_prime_gen_results.py` that reads the results from a given filename and generates visualizations using the `matplotlib` library (see Appendix A.6).

In the subsections below, we present summary tables of the collected data and accompanying figures for each bit length tested.

Test 1: 1024-bit Primes

#	GMP	OpenSSL	iZp (1 core)	iZp (4 cores)	iZp (8 cores)
1	0.008732	0.940930	0.010529	0.003381	0.005142
2	0.008613	1.422906	0.018641	0.004927	0.004274
3	0.015542	0.173003	0.011586	0.004721	0.004108
4	0.023269	2.988969	0.022492	0.003667	0.003117
5	0.013833	2.469772	0.008055	0.011109	0.007892
6	0.009113	0.165520	0.020031	0.004238	0.004447
7	0.013872	1.023229	0.003245	0.004430	0.004089
8	0.003612	1.597894	0.028956	0.013152	0.006587
9	0.003101	1.865864	0.004140	0.005523	0.011494
10	0.010164	1.956690	0.012144	0.004998	0.005950
Avg	**0.010985**	**1.460478**	**0.013982**	**0.006015**	**0.005710**

Table 6.1: Performance comparison for generating 1024-bit primes (time in seconds)

Figure 6.1 visualizes the results in the table above.

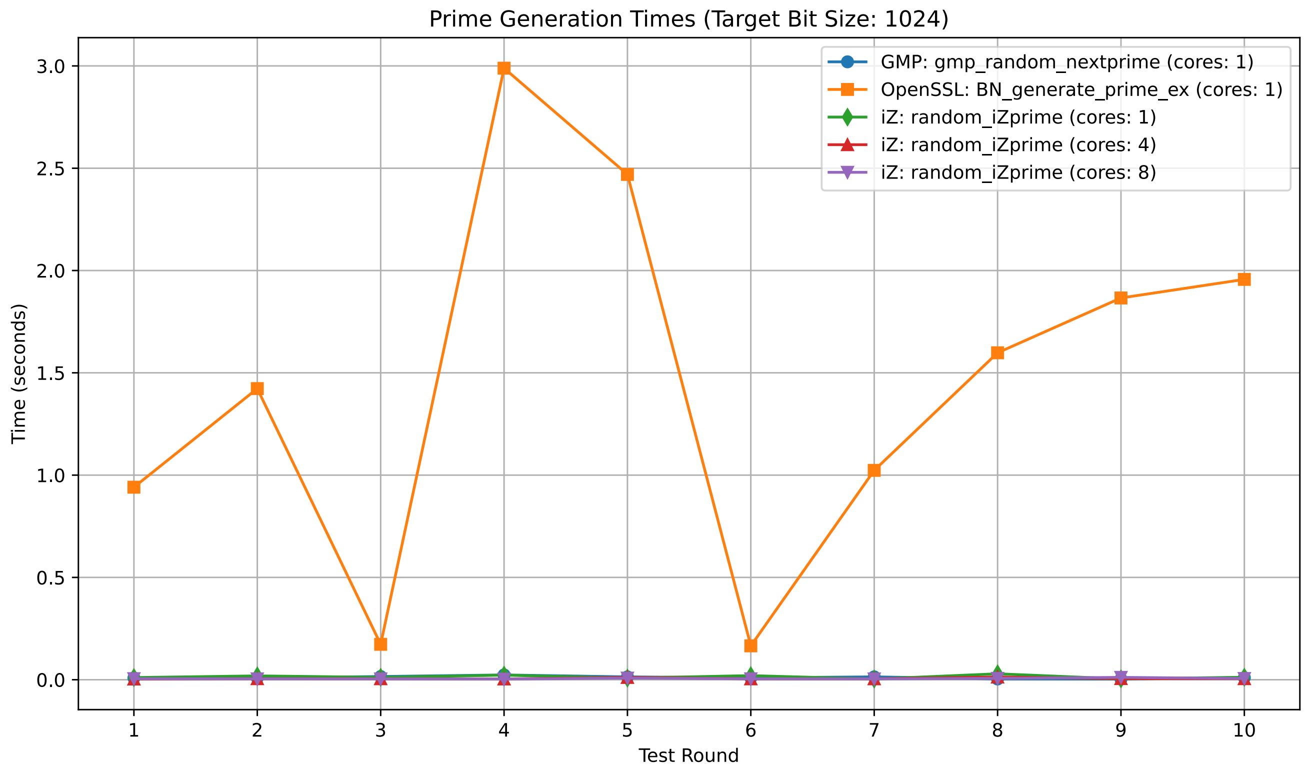

Figure 6.1: Performance comparison for generating 1024-bit primes (time in seconds)

Test 2: 2048-bit Primes

#	GMP	OpenSSL	iZp (1 core)	iZp (4 cores)	iZp (8 cores)
1	0.124402	24.395934	0.158063	0.089324	0.036901
2	0.196042	12.861515	0.182536	0.015903	0.108915
3	0.053405	11.401909	0.156033	0.017857	0.040922
4	0.197538	1.153450	0.040029	0.012705	0.028163
5	0.064923	3.326440	0.035204	0.033828	0.023575
6	0.016067	5.895255	0.056620	0.120726	0.012860
7	0.034456	0.649397	0.177153	0.080529	0.041932
8	0.019196	3.075183	0.032357	0.122815	0.041326
9	0.061010	10.628193	0.064861	0.022538	0.017737
10	0.035144	15.287856	0.104032	0.069838	0.085290
Avg	**0.080218**	**8.867513**	**0.100689**	**0.058606**	**0.043762**

Table 6.2: Performance comparison for generating 2048-bit primes (time in seconds)

Figure 6.2 visualizes the results in the table above.

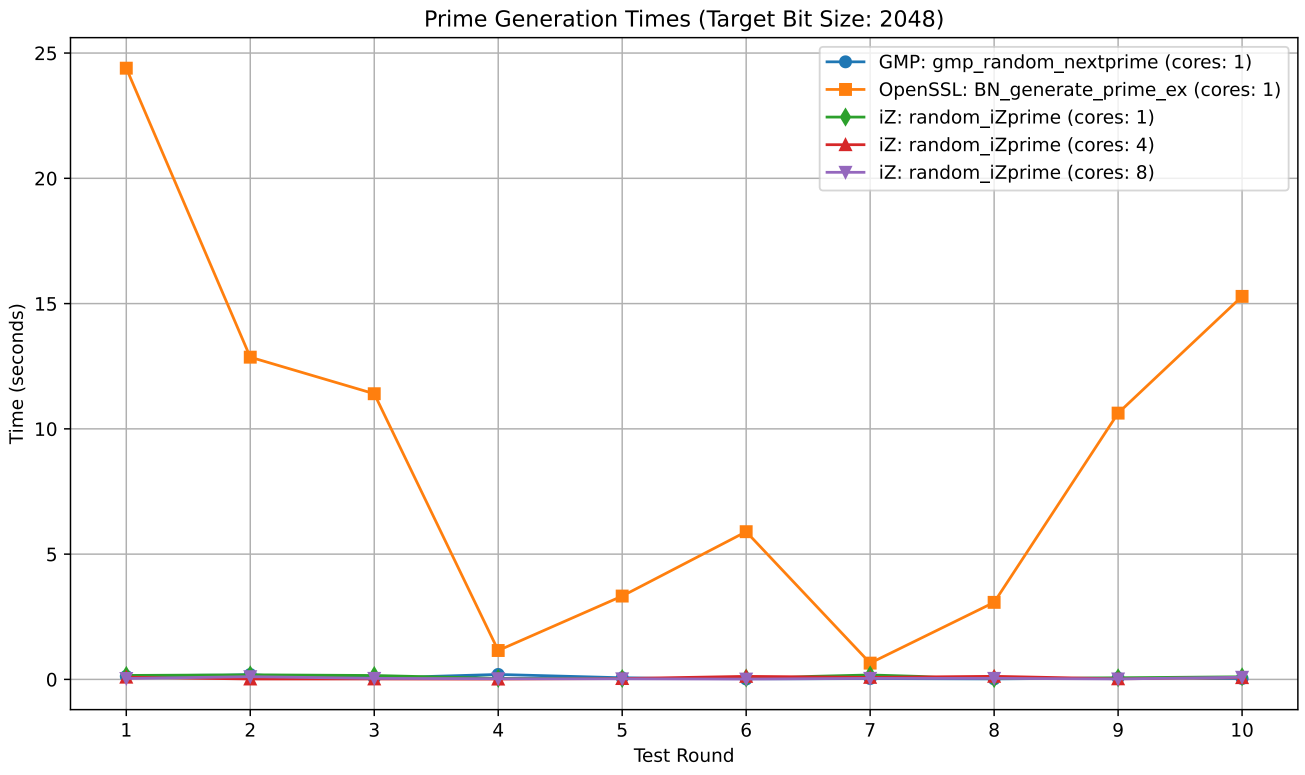

Figure 6.2: Performance comparison for generating 2048-bit primes (time in seconds)

Test 3: 4096-bit Primes

#	GMP	iZp (1 core)	iZp (4 cores)	iZp (8 cores)
1	2.143757	2.060247	0.265770	0.582672
2	0.427300	3.116232	0.666622	0.238619
3	0.344873	1.633365	0.449068	0.296668
4	1.954202	0.988343	0.415596	0.390948
5	0.278448	5.032147	1.198160	0.498869
6	2.083220	0.309284	1.347535	0.118937
7	2.219673	0.946282	1.370264	0.643910
8	3.506253	1.152812	0.528583	0.485366
9	0.517578	1.337444	0.280783	0.672737
10	0.665742	0.119402	0.590508	0.726553
Avg	**1.414105**	**1.669556**	**0.711289**	**0.465528**

Table 6.3: Performance comparison for generating 4096-bit primes (time in seconds)

Figure 6.3 visualizes the results in the table above.

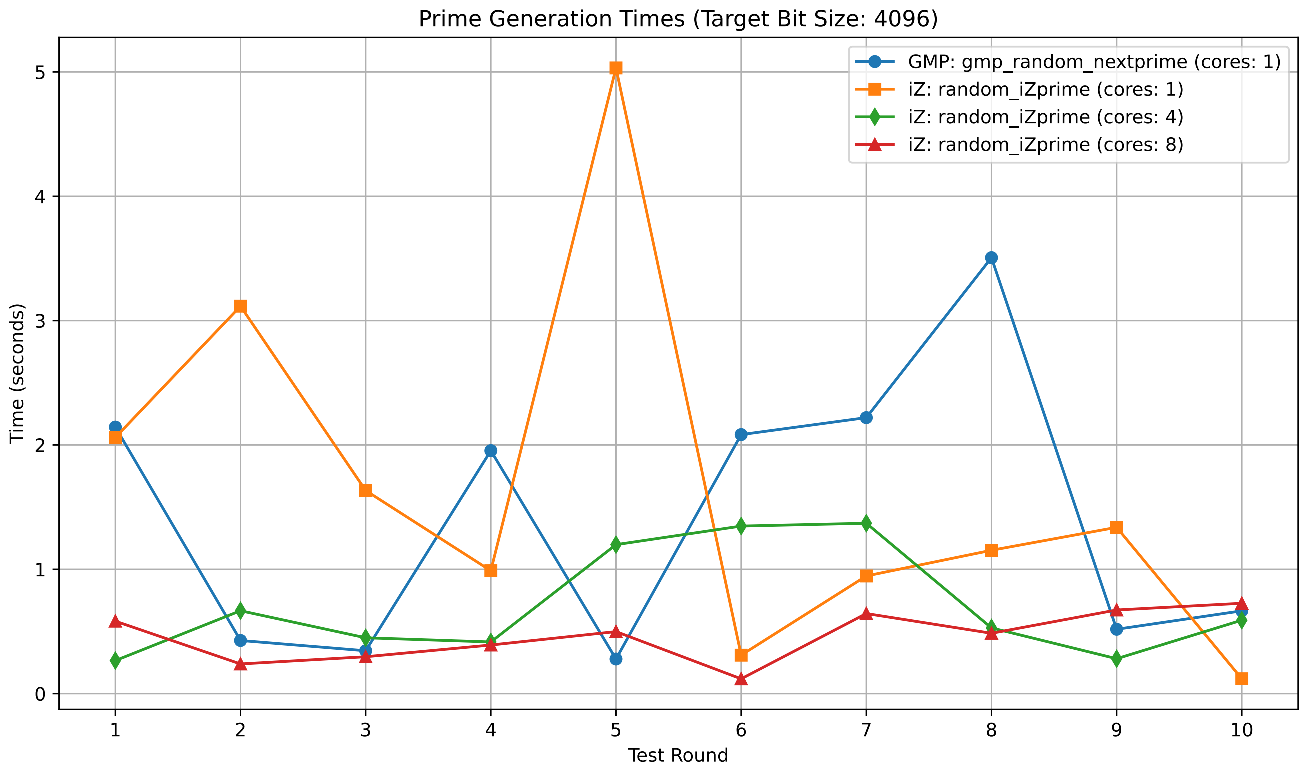

Figure 6.3: Performance comparison for generating 4096-bit primes (time in seconds)

Test 4: 8192-bit Primes

#	GMP	iZp (1 core)	iZp (4 cores)	iZp (8 cores)
1	13.033129	59.528888	5.023100	0.725199
2	3.430048	2.995323	4.117452	0.516154
3	26.200280	35.784237	9.846089	3.952084
4	10.326843	7.111855	1.008631	3.013857
5	17.455934	32.104637	51.327547	3.840639
6	1.204826	45.161471	2.638361	1.197998
7	24.725384	98.598841	2.820274	3.891904
8	23.570754	3.766899	6.087940	10.451171
9	18.153513	75.592195	25.153960	6.293954
10	4.048964	94.431717	15.093278	11.681656
Avg	**14.214968**	**45.507606**	**12.311663**	**4.556462**

Table 6.4: Performance comparison for generating 8192-bit primes (time in seconds)

Figure 6.4 visualizes the results in the table above.

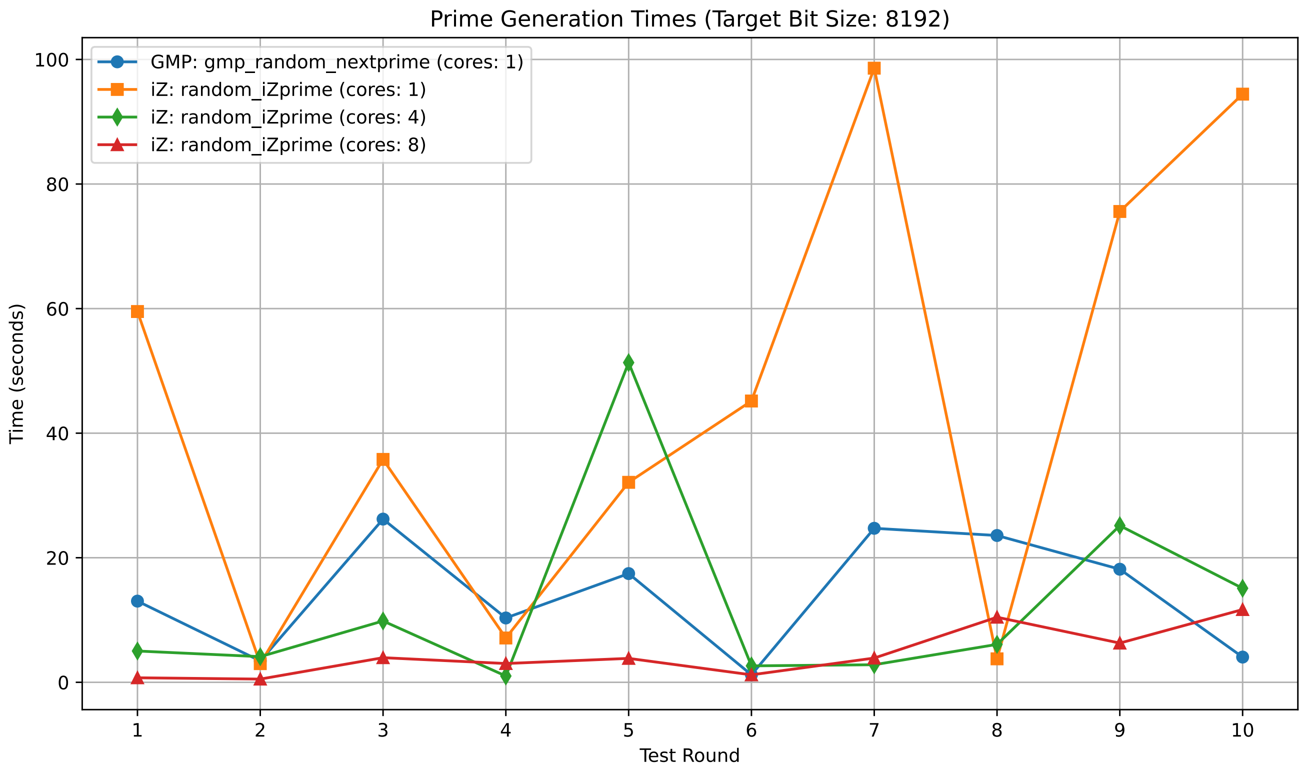

Figure 6.4: Performance comparison for generating 8192-bit primes (time in seconds)

6.4.3 Performance Analysis

Across all tested bit lengths, the results reveal several clear trends:

1. Single-Core Efficiency

Random-iZprime (1 core) is consistently faster than OpenSSL, and stands close to GMP's performance for moderate bit lengths. Although GMP can sometimes run slightly faster for certain bit sizes, Random-iZprime generally maintains a tight margin while exceeding OpenSSL's speed.

2. Parallel Speedups

With 4 or 8 cores, Random-iZprime shows a pronounced decrease in average prime generation time. This near-linear scaling is particularly evident at 2048 bits and above, where prime density is lower, leading to more extensive searches. In some cases (e.g., 4096 bits), the 8-core version achieves roughly **2–3**× the speed of the single-core version, significantly outpacing the other libraries.

3. High Bit-Length Behavior (8192 bits)

At 8192 bits, prime generation becomes quite time-consuming for all approaches. However, Random-iZprime with 4 or 8 cores continues to provide substantial improvements, often halving or quartering the single-core time. Such gains are essential for cryptographic applications that require large primes and must minimize key-generation latency.

4. Consistency in Results

While prime generation is random, Random-iZprime exhibits relatively stable run times across multiple trials, suggesting effective distribution of candidate searches among threads. By contrast, OpenSSL occasionally shows large swings in performance from one run to another, possibly due to differences in prime searching strategies or other overheads.

Overall, these benchmarks confirm that Random-iZprime offers both strong single-core performance and highly effective concurrency for large prime generation. In scenarios where cryptographic keys must be generated rapidly (e.g., ephemeral RSA keys, zero-knowledge proof parameters), harnessing multiple cores can drastically reduce wait times for prime discovery.

6.5 Conclusions

In this chapter, we presented the *Random-iZprime Algorithm*—a novel method for efficiently generating large, random primes. By leveraging the geometric structure of the iZ-Matrix and its inherent filtering of composite residues, the algorithm significantly narrows the candidate search space, enabling rapid identification of large primes even at high bit lengths. The key contributions and findings of this work are as follows:

- **Efficient Candidate Reduction:**

  Building on the observation that all primes greater than 3 have the form $6x \pm 1$, the algorithm discards $\frac{2}{3}$ of all integers at the outset. Moreover, the arrangement of numbers in the iZ-Matrix—coupled with the strategic selection of a horizontal vector (vx)—further reduces the search space by eliminating composite columns. This results in a more focused and efficient prime search.

- **Robust and Scalable Design:**

  The modular and flexible architecture of Random-iZprime allows it to efficiently generate random primes through probabilistic testing (via GMP's Miller–Rabin routines) while maintaining high assurance. The design also supports parallel processing, which translates into near-linear speedups as demonstrated by our benchmarks, particularly at bit sizes of 2048 bits and above.

- **Competitive Performance:**

  Benchmark comparisons with standard libraries such as

  - GMP's `mpz_nextprime` and
  - OpenSSL's `BN_generate_prime`,

  reveal that Random-iZprime achieves competitive single-core performance and, when scaled across multiple cores, consistently outperforms the alternatives. The efficiency improvements are most pronounced at higher bit lengths, where the density of primes is lower and the benefits of parallel processing become increasingly significant.

- **Practical Implications for Cryptography:**

  The ability to quickly generate large primes with low latency is critical for many cryptographic applications, including key generation for RSA and the construction of parameters for zero-knowledge proofs. Random-iZprime's blend of probabilistic testing, effective candidate filtering, and parallel search capabilities makes it a valuable tool in these domains.

In summary, the Random-iZprime Algorithm demonstrates that by integrating iZ-based theoretical insights with advanced parallel processing techniques, it is possible to achieve a fast and scalable prime-generation engine. Future work may focus on further optimizing the probabilistic testing component or exploring more sophisticated strategies for dynamic load balancing among cores. These enhancements could further reduce prime-generation latency, making Random-iZprime even more attractive for high-performance cryptographic systems and other applications requiring rapid prime discovery.

Appendix A

iZ-Framework: Structure & Components

This appendix provides the **iZ-Framework** implementations in C, capitalizing on hardware-level efficiency. We describe the core data structures and auxiliary functions that underpin the algorithms introduced in the main text.

A.1 iZ-Framework File Structure

Below is the overall file structure of the iZ-Framework, including source and header files along with directories for logs, output, and build artifacts:

```
iZ-lib/ "root directory"
    +-- build/ "build files"
    +-- include/ "header files"
    |   |   |-- bitmap.h
    |   |   |-- iZ.h
    |   |   |-- logger.h
    |   |   |-- primes_obj.h
    |   |   |-- random_iZprime.h
    |   |   |-- sieve.h
    |   |   |-- utils.h
    +-- logs/ "log files"
    |   |   |-- log.txt
    +-- output/ "output files and stdout"
    |   |   |-- stdout.txt
    +-- src/ "source code"
    |   |   +-- modules/ "data structures"
    |   |   |   |-- bitmap.c
    |   |   |   |-- logger.c
    |   |   |   |-- primes_obj.c
    |   |   +-- sieve/ "sieve algorithms and tests"
    |   |   |   |-- plot_sieve_results.py
    |   |   |   |-- sieve.c
    |   |   |   |-- sieve_iZ.c
    |   |   |   |-- test_sieve.c
    |   |   +-- random_prime/ "random prime generation algorithms and tests"
    |   |   |   |-- random_iZprime.c
    |   |   |   |-- test_iZprime.c
    |   |   |-- iZ.c
    |   |   |-- main.c "entry point"
    |   |   |-- utils.c
    |-- Makefile
    |-- README.md
```

A.2 Data Structures

A.2.1 BITMAP Module

The BITMAP structure (in src/modules/bitmap.c) is a dynamic bit-array, equipped with utility functions for creation, manipulation, and validation. It stores the array size (in bits), a pointer to the bit data, and a SHA-256 hash for integrity checks.

BITMAP Structure

```
// @brief Structure representing a dynamic array of bits.
typedef struct
{
    ///< Number of bits in the array.
    size_t size;
    ///< Pointer to unsigned char.
    unsigned char *data;
    ///< SHA-256 hash of the data for validation.
    unsigned char sha256[SHA256_DIGEST_LENGTH];
} BITMAP;
```

Functions

- bitmap_create: Creates a new bitmap with the specified size.
- bitmap_free: Frees the memory allocated for the bitmap.
- bitmap_set_all: Sets all bits in the bitmap to 1.
- bitmap_set_bit: Sets a specific bit in the bitmap to 1.
- bitmap_get_bit: Gets the value of a specific bit in the bitmap.
- bitmap_clear_bit: Clears a specific bit in the bitmap (sets it to 0).
- bitmap_clear_mod_p: Clears all bits from start_idx to limit with step p.
- bitmap_clone: Creates a clone of the given bitmap.
- bitmap_copy: Copies a segment of bits from the source bitmap to the destination bitmap.
- bitmap_duplicate_segment: Duplicates a segment of bits within the bitmap.
- bitmap_compute_hash: Computes the SHA-256 hash of the bitmap data.
- bitmap_validate_hash: Validates the SHA-256 hash of the bitmap data.

Additional functions for persisting/loading bitmaps from files reside in bitmap.c.

A.2.2 PRIMES_OBJ Module

The PRIMES_OBJ structure (in src/modules/primes_obj.c) stores an array of prime numbers, with creation, resizing, and validation utilities. It holds:

```c
/**
 * @brief Structure representing a collection of prime numbers.
 */
typedef struct
{
    ///< Current capacity of the primes array.
    int capacity;
    ///< Number of primes currently stored.
    int p_count;
    ///< Pointer to the dynamically allocated primes uint64_t array.
    uint64_t *p_array;
    ///< SHA-256 hash of p_array for validation.
    unsigned char sha256[SHA256_DIGEST_LENGTH];
} PRIMES_OBJ;
```

Key Functions

- primes_obj_init: Initializes a new PRIMES_OBJ structure with an initial estimate of capacity.
- primes_obj_append: Appends a prime number to the primes array.
- primes_obj_resize_to_p_count: Resizes the primes array to match the current number of primes stored.
- primes_obj_free: Frees the memory allocated for the PRIMES_OBJ structure.
- primes_obj_compute_hash: Computes the SHA-256 hash of the primes array.
- primes_obj_validate_hash: Validates the SHA-256 hash of the primes array.

Some I/O functions for persisting/loading this structure also appear in primes_obj.c.

A.2.3 Sieve Model

To encapsulate the sieve algorithms, we define a structure to associate the sieve function with its name, as well as a structure to hold multiple sieve algorithms.

```c
// Sieve function type definition,
// takes uint64_t limit and returns a PRIMES_OBJ structure
typedef PRIMES_OBJ *(*sieve_fn)(uint64_t);

// Structure to associate the sieve function with its name
typedef struct
{
    sieve_fn function;
    const char *name;
} SieveAlgorithm;

// Structure to hold multiple sieve algorithms
typedef struct
{
    SieveAlgorithm *models_list;
    int models_count;
} SieveModels;
```

A.3 Sieve Algorithms

The exact implementations of the sieve algorithms in the benchmarks are detailed below.

A.3.1 Sieve of Eratosthenes

```
/**
 * @brief Optimized Sieve of Eratosthenes algorithm to find all primes up to n.
 *
 * @param n The upper limit to find primes.
 * @return A pointer to a PRIMES_OBJ structure containing the list of prime
 * numbers up to n, or NULL if memory allocation fails.
 */
PRIMES_OBJ *sieve_eratosthenes(uint64_t n)
{
    // Initialize the primes object with an estimated capacity
    PRIMES_OBJ *primes = primes_obj_init(pi_n(n) * 1.5);

    // Create a bitmap to mark prime numbers
    BITMAP *n_bits = bitmap_create(n + 1);
    bitmap_set_all(n_bits);

    uint64_t n_sqrt = sqrt(n);

    // Add 2 as the first prime
    primes_obj_append(primes, 2);

    // Sieve algorithm to mark non-prime numbers, skipping even numbers
    for (uint64_t p = 3; p <= n; p += 2)
    {
        if (bitmap_get_bit(n_bits, p))
        {
            primes_obj_append(primes, p);
            if (p <= n_sqrt)
                bitmap_clear_mod_p(n_bits, 2 * p, p * p, n + 1);
        }
    }

    bitmap_free(n_bits);

    // Resize primes array to fit the exact number of primes found
    if (primes_obj_resize_to_p_count(primes) != 0)
    {
        // Error already logged in primes_obj_resize
        primes_obj_free(primes);
        return NULL;
    }

    return primes;
}
```

A.3.2 Segmented Sieve of Eratosthenes

```c
/**
 * @brief Segmented Sieve of Eratosthenes algorithm to find all primes up to n.
 *
 * @param n The upper limit to find primes.
 * @return A pointer to a PRIMES_OBJ structure containing the list of prime
 * numbers up to n, or NULL if memory allocation fails.
 */
PRIMES_OBJ *segmented_sieve(uint64_t n)
{
    // Initialize PRIMES_OBJ with an estimated capacity
    PRIMES_OBJ *primes = primes_obj_init(pi_n(n) * 1.5);
    if (primes == NULL)
    {
        // Error already logged in primes_obj_init
        return NULL;
    }

    // Define the segment size; can be tuned based on memory constraints
    uint64_t segment_size = (uint64_t)sqrt(n);

    // Step 1: Sieve small primes up to sqrt(n) using the traditional sieve
    BITMAP *n_bits = bitmap_create(segment_size + 1);
    if (n_bits == NULL)
    {
        primes_obj_free(primes);
        return NULL;
    }
    bitmap_set_all(n_bits);

    primes->p_array[primes->p_count++] = 2; // Add 2 as the first prime

    // Sieve odd numbers starting from 3 up to segment_size
    for (uint64_t p = 3; p <= segment_size; p += 2)
    {
        if (bitmap_get_bit(n_bits, p))
        {
            primes_obj_append(primes, p);

            // Start marking multiples of p from p*p within the small_bits
            for (uint64_t multiple = p * p; multiple <= segment_size;
                multiple += 2 * p)
                bitmap_clear_bit(n_bits, multiple);
        }
    }

    // Step 2: Segmented sieve
    uint64_t low = segment_size + 1;
    uint64_t high = low + segment_size - 1;
    if (high > n)
        high = n;

    // Iterate over segments
    while (low <= n)
    {
        bitmap_set_all(n_bits);
        // Sieve the current segment using the small primes
        for (int i = 0; i < primes->p_count; i++)
        {
            uint64_t p = primes->p_array[i];
            if (p * p > high)
                break;

            // Find the minimum number in [low, high] that is a multiple of p
            uint64_t start = (low / p) * p;
            if (start < low)
                start += p;
            if (start < p * p)
                start = p * p;

            // Mark multiples of p within the segment
            for (uint64_t j = start; j <= high; j += p)
            {
                // Skip even multiples
                if (j % 2 == 0)
                    continue;

                size_t index = j - low;
                bitmap_clear_bit(n_bits, index);
            }
        }

        // Collect primes from the current segment
        for (uint64_t i = low; i <= high; i++)
        {
            // Skip even numbers
            if (i % 2 == 0)
                continue;

            if (bitmap_get_bit(n_bits, i - low))
                primes_obj_append(primes, i);
        }

        // Move to the next segment
        low = high + 1;
        high = low + segment_size - 1;
        if (high > n)
            high = n;
    }

    // Step 3: Finalize
    // Trim the primes array to the exact number of primes found
    if (primes_obj_resize_to_p_count(primes) != 0)
    {
        // Error already logged in primes_obj_resize
        primes_obj_free(primes);
        return NULL;
    }

    return primes;
}
```

A.3.3 Sieve of Euler

```c
/**
 * @brief Generates a list of prime numbers up to a given limit using the Euler
 * Sieve algorithm.
 *
 * This function uses the Euler Sieve algorithm to find all prime numbers up to
 * a specified limit `n`. It initializes a bitmap to mark non-prime numbers and
 * an array to store the prime numbers found.
 *
 * @param n The upper limit up to which prime numbers are to be found.
 * @return A pointer to a PRIMES_OBJ structure containing the list of prime
 * numbers up to n, or NULL if memory allocation fails.
 */
PRIMES_OBJ *sieve_euler(uint64_t n)
{
    // initialization
    PRIMES_OBJ *primes = primes_obj_init(pi_n(n) * 1.5);
    BITMAP *n_bits = bitmap_create(n + 1);
    bitmap_set_all(n_bits);

    // starting the prime list with 2 to skip reading even numbers
    primes_obj_append(primes, 2);

    // sieve logic
    for (uint64_t i = 3; i <= n; i += 2)
    {
        if (bitmap_get_bit(n_bits, i))
            primes_obj_append(primes, i);

        for (int j = 1; j < primes->p_count; ++j)
        {
            uint64_t p = primes->p_array[j];

            if (p * i > n)
                break;

            bitmap_clear_bit(n_bits, p * i);

            if (i % p == 0)
                break;
        }
    }

    // cleanup
    bitmap_free(n_bits);

    // Resize primes array to fit the exact number of primes found
    if (primes_obj_resize_to_p_count(primes) != 0)
    {
        // Error already logged in primes_obj_resize
        primes_obj_free(primes);
        return NULL;
    }

    return primes;
}
```

A.3.4 Sieve of Atkin

```
/**
 * @brief Generates a list of prime numbers up to a given limit using the
 * Sieve of Atkin algorithm.
 *
 * This function implements the Sieve of Atkin, an efficient algorithm to find
 * all prime numbers up to a specified integer `n`. It initializes a bitmap to
 * mark potential primes and applies the Atkin conditions to identify primes.
 * The function also marks multiples of squares of primes as non-prime and
 * collects the primes into a dynamically allocated array.
 *
 * @param n The upper limit up to which prime numbers are to be found.
 * @return A pointer to a PRIMES_OBJ structure containing the list of
 * prime numbers up to n, or NULL if memory allocation fails.
 */
PRIMES_OBJ *sieve_atkin(uint64_t n)
{
    PRIMES_OBJ *primes = primes_obj_init(pi_n(n) * 1.5);
    BITMAP *n_bits = bitmap_create(n + 1);

    uint64_t n_sqrt = sqrt(n) + 1;

    // Initialize 2 and 3 as primes
    primes_obj_append(primes, 2);
    primes_obj_append(primes, 3);

    // 1. Mark potential primes in the bitmap using the Atkin conditions
    for (uint64_t x = 1; x < n_sqrt; ++x)
    {
        for (uint64_t y = 1; y < n_sqrt; ++y)
        {
            uint64_t num = 4 * x * x + y * y;
            if (num <= n && (num % 12 == 1 || num % 12 == 5))
                bitmap_set_bit(n_bits, num); // Toggle the bit

            num = 3 * x * x + y * y;
            if (num <= n && num % 12 == 7)
                bitmap_set_bit(n_bits, num); // Toggle the bit

            num = 3 * x * x - y * y;
            if (x > y && num <= n && num % 12 == 11)
                bitmap_set_bit(n_bits, num); // Toggle the bit
        }
    }

    // 2. Remove composites by sieving
    for (uint64_t i = 5; i <= n_sqrt; i++)
    {
        if (bitmap_get_bit(n_bits, i))
        {
            // Mark multiples of i^2 as non-prime
            for (uint64_t j = i * i; j <= n; j += i)
                bitmap_clear_bit(n_bits, j);
        }
    }
    // 3. Collect primes from the bitmap
    for (uint64_t i = 5; i <= n; i += 2)
    {
        if (bitmap_get_bit(n_bits, i))
            primes_obj_append(primes, i);
    }

    // cleanup
    bitmap_free(n_bits);

    // Resize primes array to fit the exact number of primes found
    if (primes_obj_resize_to_p_count(primes) != 0)
    {
        // Error already logged in primes_obj_resize
        primes_obj_free(primes);
        return NULL;
    }

    return primes;
}
```

A.4 Sieve Integrity Testing

To verify the correctness of the implemented sieve algorithms, the function `test_integrity` compares the outputs (by hashing prime lists) of multiple algorithms for a given limit n.

A.4.1 `test_integrity` Function

```c
/**
 * @brief Tests the integrity of different sieve models by comparing their
 * hash values.
 *
 * This function iterates through a list of sieve models, computes the primes
 * up to `n` using each model, and then compares the SHA-256 hash of the
 * resulting prime numbers. If all hashes match, the integrity is confirmed.
 *
 * @param sieve_models A structure containing the list of sieve models to
 * be tested.
 * @param n The upper limit for the prime number generation.
 * @return 0 if the integrity is confirmed, -1 if a hash mismatch is detected.
 */
int test_integrity(SieveModels sieve_models, uint64_t n);
```

A.4.2 `test_integrity` Usage Example

To perform the integrity test, in the *main.c* file, we call the `test_integrity` function as follows:

```c
int main(void)
{
    SieveAlgorithm sieve_algorithms[] = {
        SieveOfEratosthenes,
        SieveOfEuler,
        WheelSieve,
        SieveOfAtkin,
        Sieve_iZ,
        SegmentedSieve,
        Sieve_iZm,
    };

    int models_count = sizeof(sieve_algorithms) / sizeof(SieveAlgorithm);
    SieveModels sieve_models = {sieve_algorithms, models_count};

    // Performing integrity test for n = 10^3, 10^6, 10^9
    for (int i = 3; i < 10; i += 3)
        test_integrity(sieve_models, int_pow(10, i));

    return 0;
}
```

To execute the *main* function as the entry point, in the library directory, run the `make` command in the terminal:

```
    $ make
```

This will compile and run the *main* function, outputting the integrity test results in `/output/stdout.txt`

A.4.3 Test Results

The results of the integrity tests, from the code above, are summarized in the following tables for each of the limits $10^3, 10^6$ and 10^9, showing the number of primes generated, the last prime, and the hash of the generated primes list (first 16 chars):

Algorithm Name	**Primes Count**	**Last Prime**	**Hash**
Sieve of Eratosthenes	168	997	cbfc9e777d52ac21...
Segmented Sieve	168	997	cbfc9e777d52ac21...
Sieve of Euler	168	997	cbfc9e777d52ac21...
Wheel Sieve	168	997	cbfc9e777d52ac21...
Sieve of Atkin	168	997	cbfc9e777d52ac21...
Sieve-iZ	168	997	cbfc9e777d52ac21...
Sieve-iZm	168	997	cbfc9e777d52ac21...

Table A.1: Integrity Test Results for $n = 10^3$

Algorithm Name	**Primes Count**	**Last Prime**	**Hash**
Sieve of Eratosthenes	78498	999983	9a175956bcc0270c...
Segmented Sieve	78498	999983	9a175956bcc0270c...
Sieve of Euler	78498	999983	9a175956bcc0270c...
Wheel Sieve	78498	999983	9a175956bcc0270c...
Sieve of Atkin	78498	999983	9a175956bcc0270c...
Sieve-iZ	78498	999983	9a175956bcc0270c...
Sieve-iZm	78498	999983	9a175956bcc0270c...

Table A.2: Integrity Test Results for $n = 10^6$

Algorithm Name	**Primes Count**	**Last Prime**	**Hash**
Sieve of Eratosthenes	50847534	999999937	cab1dc967bd0e6ca...
Segmented Sieve	50847534	999999937	cab1dc967bd0e6ca...
Sieve of Euler	50847534	999999937	cab1dc967bd0e6ca...
Wheel Sieve	50847534	999999937	cab1dc967bd0e6ca...
Sieve of Atkin	50847534	999999937	cab1dc967bd0e6ca...
Sieve-iZ	50847534	999999937	cab1dc967bd0e6ca...
Sieve-iZm	50847534	999999937	cab1dc967bd0e6ca...

Table A.3: Integrity Test Results for $n = 10^9$

Conclusion: All algorithms generate consistent prime sets for each tested limit, confirming their correctness. If a discrepancy appears, it likely indicates an edge case that warrants further examination.

A.5 Sieve Benchmarking Tools

A.5.1 `measure_sieve_time` Function

For measuring the execution time of a given sieve algorithm, the `measure_sieve_time` function takes a sieve algorithm and an upper limit n, runs the algorithm, and measures the time taken to execute it. It prints the algorithm name, the value of n, the count of prime numbers found, the last prime number in the list, and returns the execution time in microseconds.

```c
// test_sieve.c

/**
 * @brief Measures the execution time of a given sieve algorithm.
 *
 * @param algorithm The sieve algorithm to be measured.
 * @param n The upper limit for the sieve algorithm.
 * @return The execution time in microseconds.
 */
size_t measure_sieve_time(SieveAlgorithm algorithm, uint64_t n);
```

A.5.2 `benchmark_sieve` Function

The `benchmark_sieve` function benchmarks the performance of different sieve algorithms over a range of exponents. It measures the execution time of various sieve algorithms using the `measure_sieve_time` function for a range of values determined by the base raised to the power of exponents from `min_exp` to `max_exp`. The results are printed and optionally saved to a file named by timestamp in the `output` directory.

```c
// test_sieve.c

/**
 * @brief Benchmarks the performance of different sieve algorithms over a range
 * of exponents.
 *
 * @param sieve_models A structure containing the list of sieve algorithms
 * to benchmark.
 * @param base The base value to be raised to the power of exponents.
 * @param min_exp The minimum exponent value.
 * @param max_exp The maximum exponent value.
 * @param save_results A flag indicating whether to save the results to a file.
 */
void benchmark_sieve(
    SieveModels sieve_models,
    int base,
    int min_exp,
    int max_exp,
    int save_results
);
```

A.5.3 benchmark_sieve Usage

To benchmark a list of sieve algorithms, in the *main.c* file, we call the `benchmark_sieve` function as follows:

```c
// main.c

int main(void)
{
    // List of sieve algorithms to benchmark
    SieveAlgorithm models_list[] = {
        SieveOfEratosthenes,
        SieveOfEuler,
        WheelSieve,
        SieveOfAtkin,
        Sieve_iZ,
        SegmentedSieve,
        Sieve_iZm,
    };

    int models_count = sizeof(models_list) / sizeof(SieveAlgorithm);
    SieveModels sieve_models = {models_list, models_count};

    // benchmarking sieve algorithms
    benchmark_sieve(sieve_models, 10, 4, 9, 1);

    return 0;
}
```

By running the `make` command, the *main* function will be compiled and executed, outputting the benchmark results in `/output/stdout.txt`.

If the `save_results` flag is set to 1, the results will also be saved in a file named by timestamp `/output/<timestamp>.txt`. The file name will also be printed in `stdout.txt`.

A.5.4 Plot Sieve Results

To visualize the saved benchmark results, we provide a Python script `src/sieve/plot_sieve_results.py`. Once the test results are saved in the `output` directory, take the timestamp of the results, also printed in `stdout.txt`, and pass it as an argument to the *plot_sieve_results* function, and optionally set the `save` flag to `True` to save the plot as `/output/<timestamp.svg>` file. Below is and example of how to use the *plot_sieve_results* function:

```python
# plot_sieve_results.py

# The main function "Entry Point"
if __name__ == '__main__':
    // Plot the benchmark results for the given timestamp
    plot_sieve_results(20250119183410, save=True)
```

The *plot_sieve_results* function reads the benchmark results from the `/output/<timestamp>.txt` file and generates a plot using the `matplotlib` library.

A.6 Random Prime Generation Benchmarking Tools

For benchmarking random prime generation algorithms, and formatted output, we implemented these structures and their functions in the `test_iZprime.c` file.

```c
// test_iZprime.c
#include <utils.h>
#include <random_iZprime.h>
#include <sys/time.h>

// List of prime generation algorithms: [iZp, gmp, ssl]
typedef enum
{
    iZp,      // random_iZprime
    GMP,      // gmp_nextprime
    OpenSSL   // BN_generate_prime_ex
} PRIME_GEN_ALGORITHM;

// RANDOM_PRIME_RESULT structure
typedef struct
{
    PRIME_GEN_ALGORITHM algorithm;
    int bit_size;
    int cores_num;
    char **primes_list; // Array of string primes
    double *time_array; // Array of execution times
    int results_count;  // Number of results stored in the arrays
} RANDOM_PRIME_RESULT;

// create results list structure
typedef struct
{
    RANDOM_PRIME_RESULT *results;
    int results_count;
} RESULTS_LIST;

/**
 * @brief Frees all memory associated with a RESULTS_LIST structure
 *
 * @param list Pointer to the RESULTS_LIST structure to be freed
 */
void free_results_list(RESULTS_LIST *list);

/**
 * @brief Prints the contents of a RESULTS_LIST structure
 *
 * @param list Pointer to the RESULTS_LIST structure to be printed
 */
void print_results_list(const RESULTS_LIST *list);
```

A.6.1 `measure_prime_gen_time` Function

This function performs multiple tests to measure the performance of prime number generation algorithms (iZp, GMP, or OpenSSL). For each test round, it:

- Generates a prime number using the specified algorithm
- Measures the time taken for generation
- Stores the generated prime and timing information

The function allocates memory for storing the results, which should be freed by the caller when no longer needed.

```c
/**
 * @brief Measures the time to generate random primes using different
 * algorithms.
 *
 * @param result Pointer to a RANDOM_PRIME_RESULT structure that will be
 * populated with the generated primes and timing data. The algorithm field
 * must be set before calling.
 * @param test_rounds Number of prime generations to perform
 * @param primality_check_rounds Number of rounds to use for primality testing.
 */
void measure_prime_gen_time(RANDOM_PRIME_RESULT *result, int test_rounds,
    int primality_check_rounds);
```

A.6.2 `benchmark_random_prime_algorithms` Function

This function evaluates the performance of multiple prime generation implementations:

- iZ's `random_iZprime` (custom implementation) with 1, 4, and 8 cores,
- GMP's `gmp_nextprime`,
- OpenSSL's `BN_generate_prime_ex` (if `bit_size` $\leq$ 2048).

The benchmark measures execution time for generating random primes of the specified bit size across multiple test rounds. Results are displayed to standard output and optionally saved to a timestamped file in the output directory.

```c
/**
 * @brief Benchmarks different random prime generation algorithms and
 * compares their performance.
 *
 * @param bit_size The number of bits for the generated prime numbers
 * @param primality_check_rounds Number of rounds for prime verification tests
 * @param test_rounds Number of times to repeat the prime generation for
 * each algorithm
 * @param save_results Non-zero to save results to a file, zero to only display
 * to stdout
 *
 * @note OpenSSL benchmarks are skipped for bit sizes > 2048 due to excessive
 * runtime
 * @note Results are saved in a directory specified by DIR_output global
 * variable
 * @note Memory allocated during the function is properly freed before
 * returning
 */
void benchmark_random_prime_algorithms(int bit_size,
    int primality_check_rounds, int test_rounds, int save_results);
```

A.6.3 Plot Random Prime Results

To visualize the saved random prime generation benchmark results, the Python script `plot_prime_gen_results.py` in the `src/random_prime` directory can be used. Once the test results are saved in the `output` directory, take the filename, also printed in `stdout.txt`, and pass it as an argument to the *plot_prime_gen_results* function, and optionally set the `save` flag to `True` to save the plot as `/output/<filename>.svg` file. Below is and example of how to use the *plot_prime_gen_results* function:

```python
# plot_prime_gen_results.py
if __name__ == '__main__':
    # plot_prime_gen_results("random_prime_results_20250226103248", save=True)
    # plot_prime_gen_results("random_prime_results_20250226112408", save=True)
    # plot_prime_gen_results("random_prime_results_20250226112841", save=True)
    plot_prime_gen_results("random_prime_results_20250226114627", save=True)
```