iZprime Framework: Pseudocode Documentation

`https://github.com/Zprime137/iZprime`

February 20, 2026

Introduction

The *iZprime* library is a performance-oriented C toolkit for prime sieving and prime generation. It includes classical sieve algorithms (Eratosthenes, segmented variants, Euler, Sundaram, and Atkin), as well as a family of algorithms—referred to here as the *SiZ* family—that apply effective wheel-based optimizations to reduce the constant factor in the $O(N \log \log N)$ cost model while minimizing memory usage and maximizing cache efficiency.

This document explains *how these algorithms work internally.* It is written for developers who want to understand the control flow and data structures used by the library, reason about performance trade-offs, modify or extend the algorithms safely, or build specialized prime-generation applications on top of the framework.

The emphasis here is on algorithmic design rather than formal proofs. All pseudocode presented corresponds directly to tested and working source code in the *iZprime* library.

The sieve algorithms in this document rely on two basic building blocks: an integer array (`INT_ARRAY`) (Appendix A.1) for storing the discovered primes, and a bitmap (`BITMAP`) (Appendix A.2) for representing candidates compactly and tracking which values survive as primes after marking.

Contents

1 Prime Sieve Algorithms

This section presents pseudocode for the prime sieve algorithms implemented in *src/prime_sieve.c*, together with a concise discussion of their complexity and practical behavior. Unless stated otherwise, all algorithms produce an ordered list of primes up to a given upper bound N.

In practice, prime values do not necessarily need to be stored in memory all at once: they may be counted, streamed out incrementally, or handled in fixed-size batches depending on the application. For clarity and consistency, however, the pseudocode in this section focuses on the core sieving logic and assumes that discovered primes are returned in an `INT_ARRAY`. This output container is not included in the complexity analysis, as it is not a required component of the sieving algorithms themselves.

Notations and Conventions.

- $\pi(n)$ denotes the prime counting function, which gives the number of prime numbers less than or equal to n, estimated asymptotically by $\pi(n) \sim n/\log n$.
- $range(start, end, step)$ denotes a sequence of numbers starting from *start* up to *end* (inclusive), incremented by *step* (defaulting to 1 if not specified).
- Inline conditionals are denoted as: $x \leftarrow a$ **if** *condition* **else** b, which assigns a to x if the condition is true, and b otherwise.

1.1 SoE (Sieve of Eratosthenes)

The Sieve of Eratosthenes, attributed to the ancient Greek mathematician Eratosthenes of Cyrene (c. 276–194 BCE), is one of the most fundamental algorithms for finding all prime numbers up to an upper bound N.

SoE Algorithm Description

1. **Initialization**:
   (a) Initialize an integer array *primes* to store the discovered primes, and add 2 to *primes* as the first prime to skip even candidates.
   (b) Initialize *sieve* bitmap of size $N + 1$ with all bits set to 1 (as prime candidates initially).
2. **Sieve Process**: Iterate over odd candidates starting from 3 up to N. For each candidate i that is still marked as prime:
   (a) Append i to *primes*.
   (b) If i is less than or equal to $\sqrt{N}$, mark all odd multiples of i starting from i^2 as composite in the *sieve* bitmap.
3. **Return Primes**: After processing all candidates, return the collected list of *primes*.

We encapsulate the sieve process step in a helper procedure `Process_N_Bitmap` for modularity.

```
1:  procedure PROCESS_N_BITMAP(primes, sieve_bitmap, n)
2:      INT_ARRAY_PUSH(primes, 2)          ▷ append 2 to primes, to focus on odd candidates
3:      n_sqrt ← ⌈√n⌉                      ▷ compute root limit
4:      for i ∈ range(3, n, 2) do          ▷ iterate over odd candidates
5:          if BITMAP_GET_BIT(sieve_bitmap, i) then   ▷ if i still marked prime
6:              INT_ARRAY_PUSH(primes, i)             ▷ append i to primes
7:              if i ≤ n_sqrt then                    ▷ marking loop for root primes only
8:                  BITMAP_CLEAR_STEPS(sieve_bitmap, 2i, i², n)   ▷ mark odd multiples of i starting from i²
9:              end if
10:         end if
11:     end for
12: end procedure
```

SoE Pseudocode

Algorithm 1 SoE — Sieve of Eratosthenes

Input: $n \in \mathbb{N}$ ▷ upper bound
Output: *primes* ▷ array of primes up to n
1: ▷ **1. Initialization:**
2: `INT_ARRAY:` $primes \leftarrow$ INT_ARRAY_INIT() ▷ initialize *primes* array
3: `BITMAP:` $sieve \leftarrow$ BITMAP_INIT$(n+1, 1)$ ▷ initialize *sieve* bitmap with all bits set to 1
4: PROCESS_N_BITMAP$(primes, sieve, n)$ ▷ **2. Sieve Process:** mark composites and collect primes
5: **return** *primes* ▷ **3. Output** *primes*

Asymptotic Analysis

Space Complexity: Since SoE maintains a single contiguous bitmap of size $N+1$, its auxiliary space requirement is **linear** $O(N)$. While acceptable for small to moderate values of N, this becomes prohibitive for large limits.

Speedups: This implementation applies several well-known optimizations to improve efficiency:

- **Skip even numbers**: The algorithm treats 2 as a special case and processes only odd candidates (incrementing by 2), immediately halving the search space.
- **Start marking from** p^2: For each prime p, marking begins at p^2 rather than $2p$, since all smaller multiples have already been marked by smaller primes.
- **Mark only odd multiples**: When marking multiples of odd primes, the step size is $2p$ rather than p, skipping even composites.

After applying these optimizations, the total bit-work is estimated as:

$$O\left(\frac{N}{2}\sum_{p=3}^{\sqrt{N}}\frac{1}{p}\right).$$

Time Complexity: Since the cost per prime for marking composites is proportional to $1/p$, and the sum $\sum_{p\le\sqrt{N}} 1/p$ grows as $O(\log\log N)$, the overall time complexity is $O(N \log\log N)$ with an improved constant factor (effectively $N/2$ instead of N for the naive implementation).

1.2 SSoE (Segmented Sieve of Eratosthenes)

The Segmented Sieve of Eratosthenes (SSoE) is a memory-efficient variant of the classical SoE algorithm. Rather than allocating a single monolithic bitmap of size N, SSoE divides the candidate range into smaller segments of size Δ (typically $\sqrt{N}$) and processes each segment sequentially. As a result, peak memory usage drops dramatically, allowing SSoE to handle limits N that would be infeasible with the standard SoE.

SSoE Algorithm Description

1. **Initialization**:
   (a) Initialize an empty integer array *primes* to store the discovered primes.
   (b) Compute the $segment\_size = \lceil\sqrt{N}\rceil$ to determine the size of each segment.
   (c) Create a *sieve* bitmap of size *segment_size* to represent candidates within each segment.
2. **Sieve First Segment**: Sieve the first segment $[1, segment\_size]$ using the standard SoE approach to collect all root primes up to $\sqrt{N}$ into the *primes* array.
3. **Sieve Subsequent Segments**: Define initial segment bounds $low = segment\_size + 1, high = 2 \cdot segment\_size$, then, for each segment $[low, high]$ until all segments up to N have been processed:
   (a) Reset the segment's *sieve* bitmap to all 1s.

(b) For each root prime $3 \leq p \leq \sqrt{high + 1}$, collected from the first segment, mark its odd multiples within the current segment's bounds as composite.
(c) Collect the unmarked candidates from the current segment and append them to *primes*.
(d) Advance the segment bounds *low* and *high* by *segment_size* and move to the next segment.

4. **Return Primes**: Return the collected list of *primes*.

SSoE Pseudocode

Algorithm 2 SSoE — Segmented Sieve of Eratosthenes

Input: $n \in \mathbb{N}$ ▷ upper bound
Output: *primes* ▷ array of primes up to n

▷ **1. Initialization:**
$segment\_size \leftarrow \lceil \sqrt{n} \rceil$ ▷ compute segment size
`BITMAP:` $sieve \leftarrow$ BITMAP_INIT$(segment_size + 1, 1)$ ▷ initialize *sieve* bitmap with all bits set to 1
`INT_ARRAY:` $primes \leftarrow$ INT_ARRAY_INIT$()$ ▷ initialize *primes* array
▷ **2. Sieve first segment:**
PROCESS_N_BITMAP$(primes, sieve, segment_size)$ ▷ collect root primes into *primes*
▷ **3. Sieve subsequent segments using root primes**
$low \leftarrow segment\_size + 1$ ▷ initialize segment bounds
$high \leftarrow low + segment\_size - 1$
while $low < n$ **do** ▷ iterate over segments
    BITMAP_SET_ALL$(sieve)$ ▷ reset the *sieve* bitmap
    ▷ a. Mark multiples of root primes in current segment
    for $p \geq 3 \in primes$ such that $p^2 \leq high$ **do** ▷ iterate over root primes
        $start \leftarrow \max(p^2, \lceil low/p \rceil \times p)$ ▷ find first multiple of p in current segment
        $start \leftarrow start$ **if** $start \bmod 2 \neq 0$ **else** $start + p$ ▷ ensure start is odd
        BITMAP_CLEAR_STEPS$(sieve, 2p, start - low, high - low)$ ▷ mark multiples
    end for
    ▷ b. Collect unmarked candidates from current segment
    $s \leftarrow low + (1$ if $low \bmod 2 = 0$ else $0)$ ▷ ensure s starts from the first odd candidate in the segment
    for $i \in range(s, high, 2)$ **do** ▷ iterate over odd candidates
        if BITMAP_GET_BIT$(sieve, i - low)$ **then** ▷ if unmarked, it's prime
            INT_ARRAY_PUSH$(primes, i)$ ▷ append i to *primes*
        end if
    end for
    $low \leftarrow low + segment\_size$ ▷ adjust bounds and move to next segment
    $high \leftarrow \min(high + segment\_size, n)$
end while
return *primes* ▷ **3. Output** *primes*

Asymptotic Analysis

Space Complexity: SSoE requires an `INT_ARRAY` of size $\pi(\sqrt{N})$ for storing the root primes, and a `BITMAP` of size $\sqrt{N}$ bits for the *sieve* bitmap. Thus, the overall auxiliary space complexity is:

$$O(\pi(\sqrt{N})) + O(\sqrt{N}).$$

This represents a significant improvement over SoE's $O(N)$ space requirement, enabling the algorithm to handle limits far beyond what would fit in main memory using a monolithic sieve.

Speedups: SSoE inherits all the optimizations from SoE. The key additional advantage lies in improved cache locality—by processing the sieve in small, cache-resident segments, the algorithm achieves dramatically faster memory access patterns compared to the monolithic SoE bitmap.

Time Complexity: The time complexity remains $O(N \log \log N)$, matching SoE asymptotically. However, in practice, SSoE often runs faster due to superior cache utilization, especially for large N where the monolithic bitmap would thrash the cache.

1.3 SoS (Sieve of Sundaram)

Recognizing that all primes other than 2 are odd, the Sieve of Sundaram (SoS) operates on a transformed index space that directly corresponds to odd candidates. Introduced by the Indian mathematician S. P. Sundaram in 1934, and could be considered the first wheel-based sieve.

The algorithm operates on a reduced space S of size approximately $N/2$ (Sundaram index space), and works by eliminating candidates of the form $i + j + 2ij$, where $1 \leq i \leq j$ and $i + j + 2ij \leq S$. After this elimination process, the remaining indices i in the range $[1, S]$ correspond to primes via the transformation $2i + 1$.

SoS Algorithm Description

1. **Initialization**:
   (a) Initialize an integer array *primes*, and add 2 to *primes* as the only even prime.
   (b) Compute $S = \lfloor (N-1)/2 \rfloor + 1$, the size of the Sundaram index space.
   (c) Initialize a *sieve* bitmap of size S with all bits set to 1, representing the odd candidates up to N.
2. **Sieve Process**: Iterate through indices i from 1 to S. If $sieve[i]$ is still set:
   (a) Compute the corresponding prime $p = 2i + 1$ and append it to *primes*.
   (b) If $p \leq \sqrt{N}$, mark composites of p in the Sundaram index space starting from $p \cdot i + i$ (corresponding to p^2) and incrementing by p (corresponding to marking odd multiples of p).
3. **Return Primes**: Return the collected list of *primes*.

SoS Pseudocode

Algorithm 3 SoS — Sieve of Sundaram

Input: $n \in \mathbb{N}$ ▷ upper bound
Output: *primes* ▷ array of primes up to n

▷ **1. Initialization:**
`INT_ARRAY:` $primes \leftarrow$ INT_ARRAY_INIT() ▷ initialize primes array
INT_ARRAY_PUSH($primes, 2$) ▷ add 2 (special case)
$S \leftarrow \lfloor \frac{n-1}{2} \rfloor + 1$ ▷ compute the size of the Sundaram index space
`BITMAP:` $sieve \leftarrow$ BITMAP_INIT($S, 1$) ▷ create sieve bitmap of size S with all bits set to 1

▷ **2. Sieve Process:** mark non-primes and collect primes
$n\_sqrt \leftarrow \lceil \sqrt{n} \rceil$ ▷ compute root limit
for $i \in range(1, S)$ **do** ▷ iterate through each candidate i in Sundaram space
  if BITMAP_GET_BIT($sieve, i$) **then** ▷ if i still marked, then $2i + 1$ is prime
    $p \leftarrow 2i + 1$ ▷ compute prime from index
    INT_ARRAY_PUSH($primes, p$) ▷ add p to *primes*
    if $p < n\_sqrt$ **then** ▷ if p is root prime
      $x_0 \leftarrow p \cdot i + i$ ▷ first composite index: corresponds to p^2
      BITMAP_CLEAR_STEPS($sieve, p, x_0, k$) ▷ mark composites additively with step p
    end if
  end if
end for
return *primes* ▷ **3. Output** *primes*

Asymptotic Analysis

Space Complexity: The algorithm requires **linear** $O(N/2)$ space for the sieve bitmap, which is approximately half the size of the classical SoE bitmap.

Speedups: The original Sieve of Sundaram uses a multiplicative approach to mark composites, targeting numbers of the form $i + j + 2ij$, which yields $O(N \log N)$ time complexity. This implementation instead leverages the Xp-Identity (Section 1.6) to turn marking into a simple additive progression with step size p.

This optimization, which recognizes that odd composites in the Sundaram space can be marked additively, reduces the bit-work to:

$$O\left(\frac{N}{2}\sum_{p=3}^{\sqrt{N}}\frac{1}{p}\right),$$

matching the performance of optimized SoE.

Time Complexity: With the additive marking optimization, the time complexity becomes $O(N \log \log N)$ with constant factors similar to the basic SoE implementation.

1.4 SoEu (Sieve of Euler)

The Sieve of Euler, also known as the Linear Sieve, was rediscovered by Gries and Misra in 1978. It is an elegant prime-finding algorithm that ensures each composite number is marked exactly once.

Unlike the Sieve of Eratosthenes, which may mark the same composite multiple times (for example, 15 is marked when sieving with both 3 and 5), the Sieve of Euler guarantees that each composite n is marked only by its smallest prime factor (SPF). This single-marking property makes the algorithm theoretically linear in the number of operations.

SoEu Algorithm Description

1. **Initialization**:
   (a) Initialize an `INT_ARRAY` *primes* to store the discovered primes, and add 2 to *primes* as the first prime to skip even candidates.
   (b) Create a *sieve* bitmap of size $N + 1$ with all bits set to 1 (as prime candidates).
2. **Sieve Process**: Iterate over odd candidates i from 3 to N:
   (a) If $sieve[i]$ is set, append i to *primes*.
   (b) For each collected prime $p > 2$, mark $p \cdot i$ as composite. Stop the inner loop when $p \cdot i > N$.
   (c) If $p \mid i$, stop the inner loop early; this ensures every composite is marked exactly once by its smallest prime factor.
3. **Return Primes**: Return the collected list of *primes*.

SoEu Pseudocode

Algorithm 4 SoEu — Sieve of Euler

Input: $n \in \mathbb{N}$ ▷ upper bound
Output: *primes* ▷ array of primes up to n

▷ **1. Initialization:**
`INT_ARRAY:` $primes \leftarrow$ INT_ARRAY_INIT() ▷ initialize primes array
INT_ARRAY_PUSH($primes, 2$) ▷ add 2 to primes
`BITMAP:` $sieve \leftarrow$ BITMAP_INIT($n + 1, 1$) ▷ create sieve bitmap
▷ **2. Sieve Process:** mark composites and collect primes
for $i \in range(3, n, 2)$ **do** ▷ iterate over odd candidates
  if BITMAP_GET_BIT($sieve, i$) **then** ▷ if i is prime
    INT_ARRAY_PUSH($primes, i$) ▷ add i to primes
  end if
  for $p > 2 \in primes$ **do** ▷ iterate over collected primes, and mark $p \cdot i$ as composite
    if $p \times i > n$ **then break** ▷ stop if product exceeds n
    end if
    BITMAP_CLEAR_BIT($sieve, p \times i$) ▷ mark $p \cdot i$ as composite

```
        if i mod p = 0 then break                ▷ crucial: stop at smallest prime factor
        end if
    end for
end for
return primes                                    ▷ 3. Output primes
```

Asymptotic Analysis

Space Complexity: The algorithm requires $O(N)$ space for the sieve bitmap and $O(\pi(N))$ space for storing all the progressively collected primes, which becomes a challenge for very large N.

Speedups: The key optimization is the inner loop termination condition ($i \bmod p = 0$), we know that p is the smallest prime factor of i. Therefore, any composite of the form $i \times q$ for primes $q > p$ will have a smaller prime factor (namely p), and will be marked later when that smaller factor is encountered. This crucial insight ensures each composite is marked exactly once.

The algorithm skips even numbers by starting at 3 and incrementing by 2, processing only odd candidates. The total bit-work for marking all odd composites exactly once is:

$$O\left(\frac{N}{2}\right).$$

Time Complexity: The algorithm achieves true $O(N)$ time complexity since each candidate number is visited at most once during the marking phase. However, the constant factors are typically higher than optimized SoE implementations due to the nested loop structure over all collected primes and less cache-friendly memory access patterns.

1.5 SoA (Sieve of Atkin)

The Sieve of Atkin is a modern algorithm for finding all prime numbers up to a specified upper bound N. Developed by A. O. L. Atkin and Daniel J. Bernstein in 2004, it employs a fundamentally different approach from the classical Eratosthenes sieve, based on deep results from algebraic number theory involving quadratic forms.

SoA Algorithm Description

1. **Initialization**:
   (a) Initialize an `INT_ARRAY` *primes*, and add 2 and 3 to *primes* as the first two primes.
   (b) Create a *sieve* bitmap of size $N + 1$ with all bits set to 0, representing the initial state where all numbers are considered composite until they pass the Atkin conditions.
2. **Preprocessing**: Flip *sieve*[i] for values of i generated by the three quadratic forms ($4x^2 + y^2$, $3x^2 + y^2$, and $3x^2 - y^2$), whenever the corresponding modular condition holds. Numbers satisfying these conditions an odd number of times are potentially prime.
3. **Sieving Primes Odd-Squares**: For each $p \geq 5$ with *sieve*[p] $= 1$ and $p^2 \leq N$, clear odd multiples of p^2.
4. **Collect primes**: Append all remaining $p \geq 5$ with *sieve*[p] $= 1$ to *primes*.
5. **Return Primes**: Return the collected list of *primes*.

SoA Pseudocode

Algorithm 5 SoA — Sieve of Atkin

```
Input: n ∈ ℕ                                     ▷ upper bound
Output: primes                                   ▷ array of primes up to n
                                                 ▷ 1. Initialization:
INT_ARRAY: primes ← INT_ARRAY_INIT()             ▷ initialize primes array
```

```
INT_ARRAY_PUSH(primes, 2)                                   ▷ add 2 and 3 as special cases
INT_ARRAY_PUSH(primes, 3)
BITMAP: sieve ← BITMAP_INIT(n + 1, 0)                       ▷ create sieve bitmap, initially all 0
                                                            ▷ 2. Preprocessing: Mark potential primes using Atkin conditions
for x ← 1 while 4x² < n do                                  ▷ Condition 1: 4x² + y² ≡ 1 (mod 12) or 5 (mod 12)
    for y ← 1 while 4x² + y² ≤ n do
        b ← 4x² + y²
        if b mod 12 = 1 or b mod 12 = 5 then
            BITMAP_FLIP_BIT(sieve, b)                       ▷ flip bit
        end if
    end for
end for
for x ← 1 while 3x² < n do                                  ▷ Condition 2: 3x² + y² ≡ 7 (mod 12)
    for y ← 1 while 3x² + y² ≤ n do
        b ← 3x² + y²
        if b mod 12 = 7 then
            BITMAP_FLIP_BIT(sieve, b)
        end if
    end for
end for
for x ← 1 while 2x² < n do                                  ▷ Condition 3: 3x² − y² ≡ 11 (mod 12) and x > y
    for y ← x − 1 down to 1 do
        b ← 3x² − y²
        if b > n then break
        end if
        if b mod 12 = 11 then
            BITMAP_FLIP_BIT(sieve, b)
        end if
    end for
end for
                                                            ▷ 3. Sieve: Eliminate composites by removing multiples of squares
n_sqrt ← ⌈√n⌉
for i ∈ range(5, n_sqrt, 2) do                              ▷ iterate over root primes ≥ 5
    if BITMAP_GET_BIT(sieve, i) then                        ▷ if i survived, it's prime
        BITMAP_CLEAR_STEPS(sieve, 2i², i², n)               ▷ clear odd multiples of i²
    end if
end for
                                                            ▷ 4. Collect remaining primes
for i ∈ range(5, n, 2) do
    if BITMAP_GET_BIT(sieve, i) then
        INT_ARRAY_PUSH(primes, i)
    end if
end for
return primes                                               ▷ 5. Output primes
```

Asymptotic Analysis

Space Complexity: The algorithm requires $O(N)$ space for the sieve bitmap.

Speedups: The Sieve of Atkin's primary advantage stems from its use of quadratic forms to identify prime candidates more directly than the Eratosthenes approach. By testing specific modular conditions based on well-chosen quadratic forms, many numbers are classified as prime or composite without the repeated marking inherent to the Eratosthenes sieve.

The bit-flipping mechanism elegantly handles numbers that satisfy multiple quadratic forms. The final

elimination phase is remarkably efficient, removing only odd multiples of squares of primes (rather than all multiples), resulting in a cost per prime of approximately $N/(2p^2)$.

The total bit-work can be approximated as:

$$O\left(N\left(\frac{1}{4}+\frac{1}{6}+\frac{1}{12}\right)+\frac{N}{2}\sum_{p=5}^{\sqrt{N}}\frac{1}{p^2}\right)=O(N),$$

where the first term represents the quadratic form tests and the second term (which converges) represents the square-multiple elimination.

Time Complexity: The theoretical time complexity is **linear** $O(N)$, which is asymptotically superior to SoE's $O(N \log \log N)$. However, in practice for typical ranges (up to 10^{12}), highly optimized implementations of SoE or its segmented variants often outperform SoA due to simpler operations, better cache utilization, and lower constant factors. The Sieve of Atkin's advantage becomes more pronounced for extremely large limits where the asymptotic improvement dominates.

1.6 SiZ (Sieve-iZ)

The Sieve-iZ (SiZ) algorithm serves as the baseline of the iZ family and applies a simple but effective wheel optimization based on modulus 6. By restricting the sieve to the reduced candidate space

$$i\mathbb{Z} = \{z \in \mathbb{N} : z \equiv \pm 1 \pmod 6\},$$

the algorithm automatically excludes all multiples of 2 and 3, eliminating roughly two-thirds of the natural numbers from consideration. We intentionally favor the modulus 6 wheel over larger wheels (such as modulus 30) as the base optimization. While larger wheels can further reduce the constant factor (e.g., to 4/15 for modulus 30), they require managing a significantly larger number of coprime residue classes. The modulus 6 wheel strikes a practical balance: it reduces the candidate space to 1/3 of the integers while requiring only two residue classes, keeping both the implementation and the sieving logic simple and efficient.

As a result, the required sieve space is reduced from N to approximately $N/3$, and the cost per prime for composite marking drops to about $N/(3p)$. The algorithm operates on two compact bitmaps—one for each residue class modulo 6—and uses the Xp-Identity to perform composite marking through simple additive steps in index space.

We begin with some formal definitions and notations that underpin the SiZ family of algorithms:

Definition 1.1 (The $i\mathbb{Z}$ set)**.** The set of all natural numbers coprime to 6, expressed as the union of two disjoint residue classes modulo 6:

$$i\mathbb{Z} \;=\; \{z \in \mathbb{N} : z \equiv \pm 1 \pmod 6\} \;=\; i\mathbb{Z}^- \dot\cup\, i\mathbb{Z}^+,$$

where

$$i\mathbb{Z}^- = \{6x-1 : x \in \mathbb{N}\} \quad \text{and} \quad i\mathbb{Z}^+ = \{6x+1 : x \in \mathbb{N}\},$$

with cardinality $|i\mathbb{Z}| = \frac{1}{3}|\mathbb{N}|$. Note that all primes other than 2 and 3 belong to $i\mathbb{Z}$.

To map between indices x and values of the form $6x \pm 1$, we define the iZ function:

Definition 1.2 (The iZ function)**.** Let $x \in \mathbb{N}$ be an index within a residue class, and $i \in \{\pm 1\}$ specify the residue class (-1 for $i\mathbb{Z}^-$ and 1 for $i\mathbb{Z}^+$). The iZ function maps (x, i) to the corresponding value in $i\mathbb{Z}$:

$$\boxed{iZ(x,\ i) = 6x + i}.$$

X-Arrays. When sieving $i\mathbb{Z}$ up to an upper bound N, we maintain two separate bitmaps, one per residue class, each of length $x_n = \lfloor (N+1)/6 \rfloor + 1$ bits:

- **X5**$[x]$ maps to the $6x-1$ candidate in the $i\mathbb{Z}^-$ lane,
- **X7**$[x]$ maps to the $6x+1$ candidate in the $i\mathbb{Z}^+$ lane.

The Xp-Identity (Composite Marking within $i\mathbb{Z}$). The Xp-Identity provides a modular, index-based formula for marking all composites of a given prime p within the $i\mathbb{Z}$ space using simple additive steps. Derived from Lemma B.1 (Appendix B).

Definition 1.3 (Xp-Identity)**.** Let $p = iZ(x_p, i_p) \in \mathbb{P}$ be a prime. The composite indices of p in the X-arrays are given by:

$$\texttt{X5:} \quad \left\{ \, p\,x_q - i_p\,x_p \;:\; x_q \geq x_p \in \mathbb{N} \, \right\} \quad \text{(composites of } p \text{ in } i\mathbb{Z}^-\text{)},$$
$$\texttt{X7:} \quad \left\{ \, p\,x_q + i_p\,x_p \;:\; x_q \geq x_p \in \mathbb{N} \, \right\} \quad \text{(composites of } p \text{ in } i\mathbb{Z}^+\text{)}.$$

While the Xp-Identity holds for all $x_q \geq 1$, we optimize the starting point ($x_q \geq x_p$) to start marking from p^2 in $i\mathbb{Z}^+$ and $p^2 \pm 2p$ in $i\mathbb{Z}^-$, avoiding duplicate marking of composites whose smaller cofactor q would have already been processed.

iZ Sieve Logic: To collect primes from the X-arrays ($x5$ and $x7$, where all bits are initially set) up to a given *x_limit*, iterate through x in 1..*x_limit* and then

1. if $x5[x]$ is set: compute $p = 6x - 1$ and collect as prime, then (if $p <$ *root_limit*) mark composites of p in both X-arrays (x5 and x7) using the Xp-Identity,
2. if $x7[x]$ is set, handle similarly for $p = 6x + 1$.

For modularity, we define the `Process_iZ_Bitmaps` procedure to encapsulate this process (implemented in *src/toolkit/iZ_toolkit.c*), which can be invoked by SiZ and its variants.

```
1: procedure PROCESS_IZ_BITMAPS(primes, x5, x7, x_limit)
2:     root_limit ← ⌈√(6 · x_limit + 1)⌉          ▷ only primes < root_limit need to mark composites
3:     for x ∈ range(1, x_limit) do                ▷ iterate over iZ candidates
4:         if BITMAP_GET_BIT(x5, x) then           ▷ if x survived in x5, then 6x − 1 is prime
5:             p ← IZ(x, −1)
6:             INT_ARRAY_PUSH(primes, p)           ▷ add p to primes
7:             if p < root_limit then              ▷ if p is root prime, mark its composites using Xp-Identity
8:                 BITMAP_CLEAR_STEPS(x5, p, p · x + x, x_limit)
9:                 BITMAP_CLEAR_STEPS(x7, p, p · x − x, x_limit)
10:            end if
11:        end if
12:        if BITMAP_GET_BIT(x7, x) then           ▷ if x survived in x7, then 6x + 1 is prime, handle similarly
13:            p ← IZ(x, 1)
14:            INT_ARRAY_PUSH(primes, p)
15:            if p < root_limit then
16:                BITMAP_CLEAR_STEPS(x5, p, p · x − x, x_limit)
17:                BITMAP_CLEAR_STEPS(x7, p, p · x + x, x_limit)
18:            end if
19:        end if
20:    end for
21: end procedure
```

SiZ Algorithm Description

1. **Initialization**:
   (a) Initialize the output array *primes*, and add 2 and 3 to focus on the $i\mathbb{Z}$ space.
   (b) Compute $x_n = \lfloor \frac{N+1}{6} \rfloor + 1$, the maximum iZ index needed to cover candidates $\leq N$.
   (c) Initialize two bitmaps $x5$ and $x7$ of length x_n, with all bits set to 1 as prime candidates.
2. **Sieve Process**: Invoke the `Process_iZ_Bitmaps` procedure to process the X-arrays up to x_n, and collect surviving primes into *primes*.
3. **Return Primes**: Output the collected list of *primes*.

SiZ Pseudocode

Algorithm 6 SiZ — Sieve-iZ

Input: $n \in \mathbb{N}$ ▷ upper bound
Output: *primes* ▷ array of primes up to n

▷ **1. Initialization:**
`INT_ARRAY:` $primes \leftarrow$ INT_ARRAY_INIT() ▷ initialize primes array and add 2 and 3
INT_ARRAY_PUSH($primes, 2$)
INT_ARRAY_PUSH($primes, 3$)
$x_n \leftarrow \left\lfloor \frac{n+1}{6} \right\rfloor + 1$ ▷ iZ upper bound
`BITMAP:` $x5 \leftarrow$ BITMAP_INIT($x_n, 1$) ▷ create bitmap $x5$ for representing $iZ-$ candidates
`BITMAP:` $x7 \leftarrow$ BITMAP_INIT($x_n, 1$) ▷ create bitmap $x7$ for representing $iZ+$ candidates
▷ **2. Sieve Process:** Mark composites and collect primes
PROCESS_IZ_BITMAPS($primes, x5, x7, x_n$) ▷ process both bitmaps and collect primes
return *primes* ▷ **3. Output** *primes*

Asymptotic Analysis

Space Complexity: SiZ maintains two bitmaps, each of size $N/6$ bits, for a total auxiliary space of **linear** $O(N/3)$. This represents a $3\times$ improvement over the classical SoE.

Speedups: The SiZ algorithm benefits from the reduced candidate space, which leads to fewer marking operations. The use of the Xp-Identity allows for efficient additive marking of composites within the $i\mathbb{Z}$ space. The total bit-work for marking all composites is estimated as:

$$O\left(\frac{N}{3}\sum_{p=5}^{\sqrt{N}}\frac{1}{p}\right).$$

Time Complexity: The applied optimizations come for free, yielding a time complexity of $O(N \log\log N)$ with an improved constant factor compared to classical SoE (effectively $N/3$ instead of $N/2$ for the optimized SoE).

1.7 SiZm (Sieve-iZm)

The Sieve-iZm (SiZm) algorithm extends the basic SiZ approach by introducing a cache-aware segmented layout denoted as the *iZ-Matrix* (iZm). Its central design goal is to perform expensive work only once, amortize that cost by reusing it across all segments, and keep active data structures small enough to reside comfortably in cache. The result is a highly scalable sieve with excellent cache locality, low memory overhead, and strong constant-factor performance. This separation between one-time preprocessing and per-segment work is a central design theme of the *iZprime* framework.

Key Ideas, Definitions, and Notations

Let $vx = VX_k$ be the product of the first k primes greater than 3. The iZm structure folds each $i\mathbb{Z}$ stream (i.e., $i\mathbb{Z}^-$ and $i\mathbb{Z}^+$) into a two-dimensional grid of fixed width vx and unbounded height vy. Each cell in the iZm corresponds to a candidate number in $i\mathbb{Z}$, mapped as follows:

- **iZm5**$[x, y]$: maps to $iZ(vx \cdot y + x,\ -1)$, representing $i\mathbb{Z}^-$ candidates.
- **iZm7**$[x, y]$: maps to $iZ(vx \cdot y + x,\ +1)$, representing $i\mathbb{Z}^+$ candidates.

where vx is fixed, $x \in [1,\ vx]$ is the index at the horizontal axis, and $y \in \mathbb{N}$ is the index at the vertical axis. This 2-D structure allows for two sieving strategies: horizontal (row-wise) or vertical (column-wise).

The key observation is that if a prime p divides the segment width vx, then all composites of p align in vertical columns spaced exactly p indices apart. When processing the iZm horizontally, as in the SiZm algorithm, this property allows all primes dividing vx to be handled once during initialization by pre-sieving them into base bitmap templates. These templates are then reused to initialize each segment's active bitmaps, eliminating the need to repeatedly mark these small but expensive primes during segment processing.

When processing the iZm vertically, as in the SiZm-vy algorithm, the same structural property allows entire columns corresponding to multiples of p to be skipped altogether. Each such skipped column reduces the candidate space by a factor of $(1 - \frac{1}{p})$ for every prime p dividing vx.

To maintain modularity, we define several reusable procedures that encapsulate the construction and processing of the iZm structure. These procedures are then composed to implement both the SiZm and SiZm-vy algorithms. (implemented in *src/toolkit/iZ_toolkit.c*)

Compute optimal segment size *vx*. In deciding the horizontal dimension vx of the iZm structure for row-wise sieving, we aim to balance cache efficiency with the need to cover a sufficient range of candidates. The vx size, using the `compute_l2_vx` procedure, is determined by the product of the smallest primes greater than 3, constrained by both the limit N and the CPU's L2 cache size.

```
INT_ARRAY: s_primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]   ▷ Global list of small primes
procedure COMPUTE_L2_VX(n)
    x_n ← ⌊(n+1)/6⌋ + 1                                              ▷ compute iZ bound
    L_2 ← L2_cache_size_bits                                         ▷ get L2 cache size in bits for the current architecture
    vx_limit ← min(x_n, L_2)                                         ▷ set vx limit to the minimum of L2 cache size or x_n
    vx ← 1                                                           ▷ initialize segment size
    for p > 3 ∈ s_primes do                                          ▷ iterate over primes greater than 3
        if vx × p > vx_limit then break                              ▷ stop if product exceeds limit
        end if
        vx ← vx × p
    end for
    return vx
end procedure
```

Here, L_2 represents the CPU's L2 cache size in bits, which varies by architecture. For example, on a typical machine with cache size 4 MB, $L_2 = 33,554,432$ bits. So in this case, this procedure would compute maximum segment size

$$VX_6 = 5 \times 7 \times 11 \times 13 \times 17 \times 19 = 1,616,615 \ ,$$

since including the next prime 23 (i.e., $VX_7 = VX_6 \times 23 = 37,181,145$) exceeds L_2. However, for more capable hardware with larger L2 cache sizes, this procedure will adaptively select larger segment sizes.

Construct pre-sieved base bitmaps. Once the segment size vx is determined, we sieve base X-arrays ($base\_x5$ and $base\_x7$) using all primes that divide vx. These base bitmaps serve as pre-sieved templates for initializing each segment's active bitmaps, eliminating the need to mark these small primes during segment processing. We encapsulate this step in the `iZm_construct_vx_base` procedure.

```
procedure IZM_CONSTRUCT_VX_BASE(vx, base_x5, base_x7)
    for p ∈ s_primes such that p | vx do                             ▷ iterate over primes dividing vx
        ip ← 1 if p mod 6 == 1 else −1
        x_p ← ⌊(p+1)/6⌋
        bitmap ← base_x5 if ip == −1 else base_x7                    ▷ select bitmap containing p
        BITMAP_CLEAR_BIT(bitmap, x_p)                                ▷ mark p itself as composite within the template
                                                                     ▷ Mark composites of p in base bitmaps using the Xp-Identity
        BITMAP_CLEAR_STEPS(base_x5, p, p · x_p − ip · x_p, vx)
        BITMAP_CLEAR_STEPS(base_x7, p, p · x_p + ip · x_p, vx)
    end for
end procedure
```

The X_0 solver. During segment processing in horizontal sieving (i.e., processing iZm's rows), for each root prime p, we need to locate p's first composite index to begin marking. The `iZm_solve_for_x0` procedure, based on the Xp-Identity, computes the first composite index x_0 of a prime p within a given matrix specified by m_id (-1 for $iZm5$, 1 for $iZm7$) at the yth segment of size vx, by solving the congruence:

$$vx \cdot y + x_0 \equiv x_p \pmod{p},$$

where x_p is normalized to x_p if $m\_id == ip$ or to $p - x_p$ otherwise.

procedure iZm_solve_for_x0(m_id, p, vx, y)
  $ip \leftarrow 1$ **if** $p \bmod 6 == 1$ **else** -1
  $xp \leftarrow \lfloor \frac{p+1}{6} \rfloor$
  if $y == 0$ **then**
    return $p \cdot xp + m\_id \cdot ip \cdot xp$ ▷ first composite index in segment 0 (optimized)
  end if
  $xp \leftarrow xp$ **if** $m\_id == ip$ **else** $p - xp$ ▷ normalize xp to be only positive
  return $p - ((vx \cdot y - xp) \bmod p)$ ▷ first composite hit of p in the y-th segment
end procedure

SiZm Algorithm Description

1. **Initialization**:
   (a) Compute the segment size vx (a product of small primes > 3) sized to fit in L2 cache.
   (b) Initialize an `INT_ARRAY` *primes* with the primes that divide the wheel $6 \cdot vx$.
   (c) Initialize and construct pre-sieved base bitmaps ($base\_x5$, $base\_x7$) of size vx.
   (d) Initialize active bitmaps ($x5$, $x7$) cloned from the pre-sieved base bitmaps for processing segments.
   (e) Compute $x_n = \lfloor \frac{N+1}{6} \rfloor + 1$, the maximum iZ index needed to cover candidates $\leq N$.
   (f) Compute $y\_limit = \lfloor x_n / vx \rfloor$, the number of full segments to process.
2. **Process First segment**: Using the cloned active bitmaps ($x5$, $x7$), sieve first segment using the `Process_iZ_Bitmaps` procedure, to collect all primes up to vx (enough to process the next segment).
3. **Process Remaining segments**: for each segment index y from 1 to y_limit (inclusive):
   (a) Reset active bitmaps ($x5$, $x7$) to the base state.
   (b) **Sieve Segment**: In the active bitmaps, for each root primes ($p \nmid (6 \cdot vx) \leq \sqrt{6(vx \cdot y + vx)}$), mark composites using the `iZm_solve_for_x0` procedure to find the first composite index for marking.
   (c) **Collect Primes**: Iterate through candidates x in $[1, vx]$ (or up to x_n mod vx for the last segment):
       if $x5[x]$ is set, compute $p = iZ(vx \cdot y + x, -1)$ and append to *primes*,
       if $x7[x]$ is set, compute $p = iZ(vx \cdot y + x, 1)$ and append to *primes*.
4. **Return Primes**: Return the collected list of *primes*.

SiZm Pseudocode

Algorithm 7 SiZm — Sieve-iZm

Input: $n \in \mathbb{N}$ ▷ upper bound
Output: *primes* ▷ array of primes up to n

▷ **1. Initialization:**
$vx \leftarrow$ COMPUTE_L2_VX(n) ▷ compute segment size that fits in L2 cache
`INT_ARRAY:` $primes \leftarrow$ INT_ARRAY_INIT() ▷ initialize *primes* array
for $p \in s\_primes$ such that $p \mid (6 \cdot vx)$ **do** ▷ add pre-sieved primes (those dividing $6 \cdot vx$)
  INT_ARRAY_PUSH($primes, p$)
end for

▷ Initialize base bitmaps, pre-sieved by primes dividing vx
`BITMAP:` $base\_x5 \leftarrow$ BITMAP_INIT($vx, 1$) ▷ create base bitmap for $iZm5$
`BITMAP:` $base\_x7 \leftarrow$ BITMAP_INIT($vx, 1$) ▷ create base bitmap for $iZm7$
iZm_construct_vx_base($vx, base\_x5, base\_x7$) ▷ pre-sieve base bitmaps

▷ **2. Process first segment:** process segment for $y = 0$
`BITMAP:` $x5 \leftarrow$ BITMAP_CLONE($base\_x5$) ▷ clone for active sieve in $iZm5$

```
BITMAP: x7 ← BITMAP_CLONE(base_x7)                          ▷ clone for active sieve in iZm7
PROCESS_IZ_BITMAPS(primes, x5, x7, vx + 1)                  ▷ process both bitmaps and collect primes
                                                            ▷ 3. Process remaining segments: for y in 1..y_limit
x_n ← ⌊(n+1)/6⌋ + 1                                         ▷ iZ upper bound
y_limit ← ⌊x_n/vx⌋                                          ▷ number of full segments
for y ∈ range(1, y_limit) do                                ▷ iterate over y up to y_limit inclusive
                                                            ▷ a. Reset active bitmaps to base state
    x5 ← BITMAP_CLONE(base_x5)
    x7 ← BITMAP_CLONE(base_x7)
                                                            ▷ b. Mark composites using root primes in current segment
    x_limit ← vx if y < y_limit else x_n mod vx             ▷ handle partial last segment
    root_limit ← √(6(vx · y + x_limit)) + 1                 ▷ local root limit
    for p ∈ primes such that p ∤ (6 · vx) do                ▷ iterate over root primes, skipping pre-sieved ones
        if p > root_limit then
            break                                           ▷ stop marking at local root limit
        end if
                                                            ▷ Mark composites of p in current segment using X_0 solver
        BITMAP_CLEAR_STEPS(x5, p, iZm_solve_for_x0(−1, p, vx, y), x_limit)
        BITMAP_CLEAR_STEPS(x7, p, iZm_solve_for_x0(1, p, vx, y), x_limit)
    end for
                                                            ▷ c. Collect primes from current segment
    for x ∈ range(1, x_limit) do                            ▷ iterate over indices in current segment
        if BITMAP_GET_BIT(x5, x) then                       ▷ if prime in iZm5
            INT_ARRAY_PUSH(primes, iZ(vx · y + x, −1))      ▷ compute corresponding p and add to primes
        end if
        if BITMAP_GET_BIT(x7, x) then                       ▷ if prime in iZm7
            INT_ARRAY_PUSH(primes, iZ(vx · y + x, 1))       ▷ compute corresponding p and add to primes
        end if
    end for
end for
return primes                                               ▷ 4. Output primes
```

Asymptotic Analysis

Space Complexity: During initialization, this algorithm requires an integer array of size $\pi(\sqrt{N})$ for storing the root primes, and four bitmaps (two base bitmaps and two active bitmaps), each with a fixed size of

$$VX_6 = 5 \times 7 \times 11 \times 13 \times 17 \times 19 = 1{,}616{,}615 \text{ bits } \approx 200 \text{ KB}$$

on typical hardware platforms to fit comfortably in L2 cache.

Overall, this yields a space complexity of $O(\pi(\sqrt{N}))+O(1)$, which practically behaves as $O(N^{\frac{1}{2}}/\log N^{\frac{1}{2}})$ for large N.

Speedups: Building upon the SiZ optimizations, SiZm achieves further improvements by pre-sieving the small primes that divide VX_6 (namely 5, 7, 11, 13, 17, and 19) in the wheel structure. Since these small primes contribute disproportionately to the total marking cost (due to their high frequency), eliminating them from the per-segment marking phase yields substantial performance gains.

After pre-sieving these primes, the remaining bit-work is estimated as:

$$O\left(\frac{N}{3}\sum_{p=23}^{\sqrt{N}}\frac{1}{p}\right) + O(1),$$

where the $O(1)$ term represents the one-time cost of wheel construction, which is negligible compared to the total work.

Time Complexity: The time complexity remains $O(N \log \log N)$, but with dramatically improved constant factors. In practice, SiZm often outperforms all other algorithms in this document for large limits, due to the combination of reduced search space, pre-sieving of expensive small primes, and excellent cache locality.

1.8 SiZm-vy (Sieve-iZm-vy)

The SiZm-vy variant processes the iZ-Matrix vertically, column by column, rather than horizontally by rows. It is especially attractive when the output order of primes is not critical. By skipping entire columns that correspond to candidates divisible by primes dividing vx, the algorithm can dramatically shrink the effective search space and reduce marking cost.

Under this strategy, the remaining candidates are reduced to approximately

$$N \prod_{p \mid 6 \cdot vx} \left(1 - \frac{1}{p}\right),$$

which represents a substantial constant-factor improvement before any per-prime marking even begins.

Key Ideas

- During initialization, the wheel width vx is chosen as $35 = 5 \cdot 7$ for moderate bounds (up to roughly one billion) and upgraded to $385 = 5 \cdot 7 \cdot 11$ for larger ranges. The choice can be tuned further as hardware capabilities and problem sizes grow.
- In the main sieving loop, we iterate over each column x in $2..vx$ and, for each matrix identifier $m\_id \in \{-1, +1\}$, skip columns that do not satisfy

$$\gcd(iZ(x, m\_id),\ vx) = 1.$$

The Y_0 solver. While processing a column x, each root prime p requires us to determine where its first composite appears within that column. The `iZm_solve_for_y0` procedure computes the smallest row index y_0 such that p divides the candidate at position $[x, y_0]$:

$$vx \cdot y_0 + x \equiv x_p \pmod{p},$$

where $x_p = \lfloor \frac{p+1}{6} \rfloor$ is normalized to be positive in the target matrix (same as in the Xp solver).

```
1: procedure IZM_SOLVE_FOR_Y0(m_id, p, vx, x)
2:     ip ← 1 if p mod 6 == 1 else −1
3:     xp ← ⌊(p+1)/6⌋
4:     xp ← xp if m_id == ip else p − xp                ▷ normalize x_p to be positive
5:     if gcd(vx, p) ≠ 1 then
6:         return −1                                    ▷ no modular inverse; no solution
7:     end if
8:     if x mod p == xp then
9:         return 0
10:    end if
11:    δ ← (xp − (x mod p)) mod p
12:    vx_inv ← MODULAR_INVERSE(vx mod p, p)
13:    return (δ · vx_inv) mod p
14: end procedure
```

SiZm-vy Algorithm Description

1. **Initialization**:
   (a) Compute $x_n = \lfloor \frac{N+1}{6} \rfloor + 1$, the iZ upper bound needed to cover candidates $\leq N$.

(b) Choose a small wheel width vx (35 or 385) based on N.
(c) Initialize *primes* with all root primes up to $root\_limit = \sqrt{N}$ using SiZ.
(d) Compute the number of rows $vy = \lfloor x_n / vx \rfloor$.
(e) Initialize *sieve* of size vy, with all bits set as potential primes in each column.

2. **Column-wise Sieving**: Iterate over columns $x \in \{2, \ldots, vx + 1\}$ in both lanes ($m\_id \in \{-1, +1\}$), skipping columns that are not coprime to the vx wheel ($\gcd(iZ(x, m\_id),\ vx) \neq 1$):
   (a) **Mark Composites**: Initialize a *sieve* for the current column with all bits set to 1. Then, for each (not pre-sieved) root prime $p \nmid 6 \cdot vx < root\_limit$, use the `iZm_solve_for_y0` routine to locate the first composite in the column and mark every pth position thereafter.
   (b) **Collect Primes**: Emit all unmarked candidates y in current column x as primes, using the iZm mapping: $iZ(y \cdot vx + x, m\_id)$.
3. **Return Primes**: Return the collected *primes* (unordered).

SiZm-vy Pseudocode

Algorithm 8 SiZm-vy — Sieve-iZm-vy

Input: $n \in \mathbb{N}$ ▷ upper bound
Output: *primes* ▷ array of primes up to n

▷ **1. Initialization**
$root\_limit \leftarrow \lfloor \sqrt{n} \rfloor + 1$
$primes \leftarrow \text{SiZ}(root\_limit)$ ▷ initialize with all primes $\leq \sqrt{n}$
$x_n \leftarrow \lfloor \frac{n}{6} \rfloor + 1$
$vx \leftarrow 35$ ▷ set vx wheel size, default $5 \cdot 7$
$k \leftarrow 4$ ▷ index of first root prime not dividing vx (points at 11)
if $n \geq 10^9$ **then**
    $vx \leftarrow 11 \cdot vx$ ▷ upgrade wheel: $5 \cdot 7 \cdot 11$
    $k \leftarrow k + 1$ ▷ skip 11 in the marking list
end if
$vy \leftarrow \lfloor x_n / vx \rfloor$ ▷ number of full rows

▷ **2. Column-wise sieving**
for $x \in range(2, vx)$ **do** ▷ iterate over columns in both lanes
    for $m\_id \in \{-1, +1\}$ **do**
        if $\gcd(iZ(x, m\_id),\ vx) \neq 1$ **then continue** ▷ skip unproductive columns
        end if
        $sieve \leftarrow \text{BITMAP\_INIT}(vy, 1)$ ▷ initialize *sieve* bitmap for current column
        for $p < root\_limit \in primes[k:]$ **do** ▷ a. Mark composites of root primes
            $y_0 \leftarrow \text{iZm\_solve\_for\_y0}(m\_id, p, vx, x)$ ▷ first composite of p in current column
            $\text{BITMAP\_CLEAR\_STEPS}(sieve,\ p,\ y_0,\ vy)$ ▷ mark composites of p in current column
        end for
        for $y \in range(0, vy - 1)$ **do** ▷ b. Collect primes from current column
            if $\text{BITMAP\_GET\_BIT}(sieve, y)$ **then**
                $\text{INT\_ARRAY\_PUSH}(primes,\ iZ(y \cdot vx + x,\ m\_id))$
            end if
        end for
        if $\text{BITMAP\_GET\_BIT}(sieve, vy)$ **then** ▷ handle partial last row
            $p \leftarrow iZ(vy \cdot vx + x,\ m\_id)$
            if $p < n$ **then**
                $\text{INT\_ARRAY\_PUSH}(primes, p)$
            end if
        end if
    end for
end for
return *primes* ▷ **3. Output** *primes* (unordered)

Asymptotic Analysis

Space Complexity: Beyond storing the root primes ($O(\pi(\sqrt{N}))$), SiZm-vy allocates a single bitmap of size $vy \approx x_n/vx$. Thus, the auxiliary space is

$$O(\pi(\sqrt{N})) + O\left(\frac{N}{210} \text{ or } \frac{N}{2310}\right) \text{ bits.}$$

For the fixed wheels used in this implementation ($vx \in \{35, 385\}$), the space remains linear in N but with a much smaller constant factor than classical $O(N)$ sieves.

Speedups: By filtering out all candidates divisible by small primes (below 13), the effective candidate space for sufficiently large N is reduced to

$$\frac{1}{2} \times \frac{2}{3} \times \frac{4}{5} \times \frac{6}{7} \times \frac{10}{11} = \frac{480}{2310} = \frac{16}{77} \approx \frac{1}{5}$$

of the original numbers up to N. Thus, the total bit-work for marking composites becomes:

$$O\left(\frac{N}{5} \sum_{p=13}^{\sqrt{N}} \frac{1}{p}\right),$$

improving the constant factor from $1/3$ to approximately $1/5$.

Time Complexity: The cost per prime is $1/p$, thus the overall time complexity is still $O(N \log \log N)$, but with improved constant factors due to the reduced candidate space.

2 Summary

Table 1 provides a consolidated technical comparison of all sieve algorithms described in this document. While many of these methods share identical asymptotic time complexity, their practical performance characteristics differ significantly due to variations in candidate density, cache locality, redundant marking behavior, memory layout, and wheel structure. The Bit-Work column offers a refined operational metric that approximates the dominant cost of composite marking and serves as a more informative indicator of real-world efficiency than asymptotic notation alone.

Algorithm	**Space Complexity**	**Time Complexity**	**Bit-Work**
SoE	$O_b(N)$	$O(N \log\log N)$	$O\left(\frac{N}{2}\sum_{p=3}^{\sqrt{N}}\frac{1}{p}\right)$
SSoE	$O_i(\pi(\sqrt{N})) + O_b(\sqrt{N})$	$O(N \log\log N)$	$O\left(\frac{N}{2}\sum_{p=3}^{\sqrt{N}}\frac{1}{p}\right)$
SoS	$O_b(N/2)$	$O(N \log\log N)$	$O\left(\frac{N}{2}\sum_{p=3}^{\sqrt{N}}\frac{1}{p}\right)$
SoEu	$O_i(\pi(N)) + O_b(N)$	$O(N)$	$O\left(\frac{N}{2}\right)$
SoA	$O_b(N)$	$O(N)$	$O\left(\frac{N}{2} + \frac{N}{2}\sum_{p=5}^{\sqrt{N}}\frac{1}{p^2}\right)$
SiZ	$O_b(N/3)$	$O(N \log\log N)$	$O\left(\frac{N}{3}\sum_{p=5}^{\sqrt{N}}\frac{1}{p}\right)$
SiZm	$O_i(\pi(\sqrt{N})) + O_b(1)$	$O(N \log\log N)$	$O\left(\frac{N}{3}\sum_{p=23}^{\sqrt{N}}\frac{1}{p}\right)$
SiZm-vy	$O_i(\pi(\sqrt{N})) + O_b\left(\frac{N}{6\cdot vx}\right)$	$O(N \log\log N)$	$O\left(\frac{N}{5}\sum_{p=13}^{\sqrt{N}}\frac{1}{p}\right)$

Table 1: Comprehensive comparison of prime sieve algorithms. Space complexity represents auxiliary space beyond the output. Bit-work estimates reflect the number of bit operations performed during sieving.

Space Complexity. The space complexity column distinguishes between integer storage and bitmap storage. The notation $O_i(x)$ denotes memory consumed by the `INT_ARRAY` component (measured in machine integers), while $O_b(x)$ denotes memory consumed by the `BITMAP` component (measured in bits). All reported values represent auxiliary space beyond the returned prime list and reflect the active working set required during sieving.

Time Complexity. The time complexity column reports asymptotic growth with respect to the upper bound N, abstracting away constant factors. Although several sieves share the same $O(N \log\log N)$ classification, their internal structure—particularly wheel reductions and pre-sieving strategies—can significantly alter the effective constant. The linear-time algorithms (SoEu and SoA) exhibit different asymptotic behavior but may incur larger constant overheads depending on implementation details and cache effects.

Bit-Work. The Bit-Work column estimates the number of composite-marking operations performed during sieving. Since marking dominates runtime in all bitmap-based sieves, this metric provides a practical approximation of computational effort. Algorithms that reduce candidate density (via wheel factorization), eliminate redundant markings, or pre-sieve small primes achieve measurable reductions in bit-work even when their asymptotic time complexity remains unchanged.

A Basic Data Structures

This section defines the basic data structures used throughout the algorithms in this document, including the integer array (`INT_ARRAY`) for storing primes and the bitmap (`BITMAP`) for representing candidate states during sieving.

A.1 Integer Array (`INT_ARRAY`)

To store the (growing) list of primes produced by a sieve, we use a dynamic integer array `INT_ARRAY`. It behaves as an append-only list with automatic resizing.

A minimal C definition of the `INT_ARRAY` structure is:

```
typedef struct
{
    size_t capacity;       ///< Total capacity (max count of elements) of the array
    int count;             ///< Number of elements currently stored
    int *array;            ///< Pointer to the dynamically allocated int array
} INT_ARRAY;
```

In this document, we use the following operations on `INT_ARRAY`:

- `int_array_init()`: Initializes and returns an `INT_ARRAY` object with a dynamically allocated array, with both *capacity* and *count* initialized to 0.
- `int_array_push(`*int_array*, *value*`)`: Appends the integer *value* to the end of *int_array*, expanding capacity as needed.

A.2 Bitmap (`BITMAP`)

For a memory-efficient representation of candidate ranges, we use a compact bit array `BITMAP`. Each *bit index* corresponds to a candidate, and each *bit state* indicates whether the candidate is still considered *potentially prime* (typically 1) or has been marked composite (0).

A minimal C definition of the `BITMAP` structure is:

```
typedef struct
{
    size_t size;        ///< Total number of bits in the bitmap
    uint8_t *data;      ///< A dynamically allocated array of bytes ((size + 7) / 8)
} BITMAP;
```

In this document, we use the following operations on `BITMAP`:

- `bitmap_init(`*size*, *initial_value*`)`: Initializes and returns a bitmap of given *size* with all bits set to *initial_value*.
- `bitmap_set_all(`*bitmap*`)`: Sets all bits in the *bitmap* to 1.
- `bitmap_get_bit(`*bitmap*, *index*`)`: Returns the bit-state at *index* in the *bitmap*.
- `bitmap_flip_bit(`*bitmap*, *index*`)`: Flips the bit-state at *index* in the *bitmap*.
- `bitmap_clear_bit(`*bitmap*, *index*`)`: Clears (sets to 0) the bit at *index* in the *bitmap*.
- `bitmap_clear_steps(`*bitmap*, *step*, *start*, *limit*`)`: Clears bits in the *bitmap* starting from *start* and incrementing by *step* until *limit* is reached.
- `bitmap_clone(`*bitmap*`)`: Creates and returns a deep copy of the given *bitmap*.

B Algebraic Study on the iZ Function

This appendix is included for completeness and for readers who are curious about the mathematical intuition behind the Xp-Identity. It is not required for understanding or using the algorithms in this document; rather, it explains why index-based additive marking works correctly within the $i\mathbb{Z}$ representation.

Lemma B.1. *Let $iZ(x \in \mathbb{N}, i \in \mathcal{I}) = mx + i$, where $m = 6$ and $\mathcal{I} = \{\pm 1\}$, be the function defining the $i\mathbb{Z}$ set of integers.*

For any $p = iZ(x_p, i_p)$ and $q = iZ(x_q, i_q)$ in $i\mathbb{Z}$, their product $z = pq$ also belongs to $i\mathbb{Z}$ and can be expressed as $z = iZ(x_z, i_z)$, where

$$x_z = mx_px_q + i_px_q + i_qx_p \quad \textit{and} \quad i_z = i_pi_q \in \mathcal{I}.$$

As a direct consequence, for each prime factor p dividing z, the following equivalence holds:

$$p \mid z \quad \Longleftrightarrow \quad x_z \equiv (\frac{i_z}{i_p})x_p \pmod{p}. \tag{1}$$

This equivalence is the core observation behind the Xp-Identity and explains why composite marking in the $i\mathbb{Z}$ space can be performed using simple additive steps on indices.

Proof. We expand the product $z = pq$:

$$z = pq = (mx_p + i_p)(mx_q + i_q) = m^2x_px_q + mi_px_q + mi_qx_p + i_pi_q,$$

Rewriting the expression by factoring out m, we obtain:

$$z = m(mx_px_q + i_px_q + i_qx_p) + i_pi_q = mx_z + i_z,$$

which confirms that $z \in i\mathbb{Z}$ with $z = iZ(x_z, i_z)$, where:

- The *product residue* $i_z \in \mathcal{I}$ is determined by:

$$i_z = i_pi_q = \begin{cases} +1 & \text{if } i_p = i_q, \\ -1 & \text{if } i_p \neq i_q. \end{cases}$$

- The *product index* x_z can be rewritten in terms of the factors p and q as:

$$\begin{aligned} x_z &= mx_px_q + i_px_q + i_qx_p \\ &= (mx_p + i_p)x_q + i_qx_p \\ &= px_q + i_qx_p, \text{ where } i_q = \frac{i_z}{i_p} \quad \#1 \\ &= (mx_q + i_q)x_p + i_px_q \\ &= qx_p + i_px_q, \text{ where } i_p = \frac{i_z}{i_q} \quad \#2 \end{aligned}$$

From forms (#1) and (#2), we obtain the following general equivalence:

$$x_z \equiv (\frac{i_z}{i_p})x_p \pmod{p},$$

for each prime factor p dividing z. □

The Xp-Identity. The equivalence established in Lemma B.1, which we refer to as the *Xp-Identity*, allows composites of a prime $p = iZ(x_p, i_p)$ to be located within the $i\mathbb{Z}$ index space using only additive steps. In practice, this means iterating over values of x_q using the following expressions:

$$\texttt{X5:} \quad \left\{ \, px_q - i_p x_p \;:\; x_q \geq 1 \in \mathbb{N} \, \right\} \quad \text{(composites of } p \text{ in } i\mathbb{Z}^-\text{)},$$
$$\texttt{X7:} \quad \left\{ \, px_q + i_p x_p \;:\; x_q \geq 1 \in \mathbb{N} \, \right\} \quad \text{(composites of } p \text{ in } i\mathbb{Z}^+\text{)}.$$

References

[1] Wikipedia contributors. Sieve of Eratosthenes. In *Wikipedia, The Free Encyclopedia*. Retrieved January 2026, from `https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes`

[2] Wikipedia contributors. Segmented sieve. In *Wikipedia, The Free Encyclopedia*. Retrieved January 2026, from `https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes#Segmented_sieve`

[3] Wikipedia contributors. Euler's sieve. In *Wikipedia, The Free Encyclopedia*. Retrieved January 2026, from `https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes#Euler's_sieve`

[4] Wikipedia contributors. Sieve of Sundaram. In *Wikipedia, The Free Encyclopedia*. Retrieved January 2026, from `https://en.wikipedia.org/wiki/Sieve_of_Sundaram`

[5] Wikipedia contributors. Sieve of Atkin. In *Wikipedia, The Free Encyclopedia*. Retrieved January 2026, from `https://en.wikipedia.org/wiki/Sieve_of_Atkin`