On Generating Prime Numbers Efficiently

Juhani Sipilä

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 2.6.2018

Supervisor

Assistant Prof. Pauliina Ilmonen

Advisor

Dr Vesa Kaarnioja

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Juhani Sipilä

Title On Generating Prime Numbers Efficiently

Degree programme Mathematics and Operations Research

Major Systems and Operations Research **Code of major** SCI3055

Supervisor Assistant Prof. Pauliina Ilmonen

Advisor Dr Vesa Kaarnioja

Date 2.6.2018 **Number of pages** 56+4 **Language** English

Abstract

The prime numbers can be considered as the building blocks of natural numbers, having innumerable applications in number theory and cryptography. There exist multiple different sieving algorithms for the generation of prime numbers.

In this thesis, an elementary modular result is utilized to construct an analytically useful generator function and its inverse function. The functions are used to generate a (log)log-linear time complexity prime sieving algorithm which is further optimized to be of linear time complexity. The constructed algorithms and their operation are studied and the linear implementations in JS, Python and C++ are compared to other prime sieves.

Keywords Prime number, prime sieve, algorithm, Myriad sieve

Aalto University
School of Science

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Diplomityön tiivistelmä

Tekijä Juhani Sipilä

Työn nimi On Generating Prime Numbers Efficiently

Koulutusohjelma Mathematics and Operations Research

Pääaine Systems and Operations Research **Pääaineen koodi** SCI3055

Työn valvoja Apulaisprof. Pauliina Ilmonen

Työn ohjaaja TkT Vesa Kaarnioja

Päivämäärä 2.6.2018 **Sivumäärä** 56+4 **Kieli** Englanti

Tiivistelmä

Alkulukuja voidaan pitää luonnollisten lukujen rakennuspalikoina joilla on lukemattomia sovelluksia lukuteoriassa ja kryptografiassa. Alkulukujen luomiseen on olemassa useita erilaisia seulonta-algoritmeja.

Tässä opinnäytetyössä käytetään modulaarista perustulosta analyyttisesti hyödyllisten kehitysfunktion ja sen käänteisfunktion luomiseen. Funktioiden avulla luodaan aikakompleksisuudeltaan (log)log-lineaarinen alkulukuseula, joka optimoidaan lineaariseksi. Rakennettuja algoritmeja ja niiden toimintaa tarkastellaan ja lineaarista implementaatiota JS, Python ja C++ ohjelmointikielillä verrataan toisiin alkulukuseuloihin.

Avainsanat Alkuluku, alkulukuseula, algoritmi, Myriad seula

"The problem of distinguishing prime numbers from composite numbers and of resolving the latter into their prime factors is known to be one of the most important and useful in arithmetic. It has engaged the industry and wisdom of ancient and modern geometers to such an extent that it would be superfluous to discuss the problem at length... Further, the dignity of the science itself seems to require that every possible means be explored for the solution of a problem so elegant and so celebrated."

C.F. Gauss, Disquisitiones Arithmeticae, article 329 (1801)

Contents

Table of notations

$\mathbb{N}$: The set of natural numbers, {1,2,3,...}.

$\mathbb{P}$: The set of prime numbers, {2,3,5,7,11,...}.

$\mathfrak{G}$: The set of numbers of the form $6m \pm 1$ ($m \in \mathbb{N}$). i.e. all natural numbers greater than 1 which do not have a factor of 2 or 3.

$\mathfrak{G}^*$: Composite numbers in $\mathfrak{G}$. The set of numbers which are expressible as a product of two (or more) elements from $\mathfrak{G}$.

p_n : The n:th prime number.

$\gcd(n, m)$: Greatest common divisor:

$$n = \prod_{p_i \in \mathbb{P}} {p_i}^{\alpha_i}, m = \prod_{p_i \in \mathbb{P}} {p_i}^{\beta_i}, \gcd(n, m) = \prod_{p_i \in \mathbb{P}} {p_i}^{\min\{\alpha_i, \beta_i\}} (\alpha_i, \beta_i \geq 0).$$

$\varphi(n)$: Euler's totient function. The number of integers m such that $1 \leq m \leq n$ for which $\gcd(n, m) = 1$.

$a \equiv n \bmod m$: Modular congruence: m divides a with a remainder of n.

$G(n)$: The generator function, producing all numbers of the form $6m \pm 1$ ($m \in \mathbb{N}$) in ascending order.

$G^{-1}(g)$: The inverse of the generator function, producing the index of $g \in \mathfrak{G}$.

1 Introduction

This thesis starts by introducing and defining the elementary definitions and necessary concepts for the presented prime generating algorithm. We start by discussing the basic concepts and terminology related to the prime numbers. We then explore and discuss certain important known results relating to the prime numbers. The applications of the prime numbers are also discussed briefly.

In chapter 2 we take a look at important results of prime generating algorithms. We introduce the oldest and probably the best known prime generating algorithm, the sieve of Eratosthenes, and analyze its behaviour. The sieve of Eratosthenes can be seen as the basis for improved and more advanced variations which are also discussed. The name sieve comes from the fact that these types of algorithms are sifting composite numbers (non prime numbers) out of an interval, in such a way that in the end only prime numbers remain in the sieve. The analogy comes from the process of sifting gold from less precious ores.

We also introduce various linear time complexity variations and introduce the concept of a factor wheel, which decreases the asymptotic time complexity to be sublinear. Probabilistic sieves are discussed briefly along with other probabilistic primality testing methods. These primality tests could also be used to generate prime numbers but the sieving methods are usually more appropriate for intervals as opposed to testing whether a given number is prime or not.

In chapter 3 we introduce the geometrical pattern observed in the squares of prime numbers, which was the original inspiration for the development of our prime sieve. This geometrical pattern gives rise to the ideas of a prime generator function $G(n)$, its range $\mathfrak{G}$ and an indexing function $I(x, y)$ which allows one to determine all composite numbers in $\mathfrak{G}$. These concepts are then used to construct a (log)log-linear prime sieve, similar to the sieve of Eratosthenes. This sieve is then optimized to be of linear time complexity. The pseudocode algorithms for these sieves are presented.

Chapter 4 is devoted to visualizing and analyzing the operation of the linear sieve. We also implement the linear sieve in JavaScript, Python and C++11 and measure the execution times of the implementations comparing them to other sieves. Lastly we discuss some potential further optimizations and improvements ending with conclusions. The implemented JavaScript and C++11 sieves can be found in the appendices or in GitHub (https://github.com/JuhaniSipila/MyriadPrimes).

1.1 Divisibility

The natural numbers form the primary subject matter of arithmetic. The process of dividing a natural number N into M equally sized parts has many simple but fascinating properties. For some integers N there is no way to divide it by M (without remainder) into smaller equally sized parts. For other N there are multiple distinct M such that the division is possible.

Definition 1.1 (Divisibility [2])**.** An integer a is said to be divisible by another integer b if there is a third integer c such that $a = b \cdot c$.

The integers b and c are called divisors or factors of a. For example the number 12 can be expressed as $12 = 1 \cdot 12 = 2 \cdot 6 = 3 \cdot 4$ and thus 12 is divisible by 1,2,3,4,6 and 12. The number 13 on the other hand doesn't divide into smaller equal integersized portions and is only trivially divisible by 1 and 13 itself. Divisibility is closely related to primality and next we shall see why numbers which are only divisible by 1 and themselves are considered so fundamental in mathematics.

1.2 Prime numbers and factoring

Definition 1.2 (Prime numbers [2])**.** A prime number (or simply a prime) is a natural number greater than 1, which has no positive divisors other than 1 and itself.

A natural number greater than 1 that is not a prime is called a composite number or simply composite. For example since 5 is only divisible by 1 and itself, it is prime. 6 on the other hand has two divisors, 2 and 3, thus 6 is composite.

Note that according to the modern definition, 1 is considered neither prime nor composite. During the history of number theory some have considered 1 as a prime while others considered it composite. Some have also disregarded 2 as a prime due to its eveness. After the modern definition of primality 2 is considered the smallest and only even prime. The modern definition of primality stems from one of the most important theorems in elementary number theory, the fundamental theorem of arithmetic.

Theorem 1.1 (Fundamental Theorem of Arithmetic [3])**.** *Every integer larger than 1 can be written as a product of one or more primes in a way that is unique except for the order of the prime factors.*

Primes can thus be considered as the basic building blocks of natural numbers. One way of determining whether or not a given number greater than 1 is prime or composite (i.e. checking for primality) is to determine all of its prime factors. Primes have only one prime factor, itself. Composites by their definition are comprised of multiple prime factors. Unfortunately this kind of method isn't very fast for sufficiently large numbers since the fastest known methods for factoring a number n take time exponential in the number of bits of n [1].

Primes have enumerous applications in mathematics, especially in number theory, but they also form the basis of modern methods of cryptography. In 1977 RSA public-key cryptosystem was published which revolutionized electronic commerce and allowed the exchange of encrypted messages through untrusted networks. More recently primes have been used in cryptocurrencies e.g. Primecoin. Next we will take a look at more involved results relating to primes.

1.3 Modular results related to this work

The initial idea for the prime sieve presented in this work originated from a geometrical pattern observed in the squares of primes, witnessed in a dream. The elementary modular congruence of $p_n^2 \equiv 1 \bmod 24$ is of ancient origin and not attributed to anyone. The corresponding geometrical pattern is presented in Figure 4 of chapter 3 along with related formulas.

Theorem 1.2. ${p_n}^2 \equiv 1 \bmod 24 \ \forall p_n > 3$.

Proof. For all $n \in \mathbb{N}$:

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} = \frac{n(n+\frac{1}{2})(n+1)}{3}.$$

Now if $p > 3$ is an odd prime, then $\frac{p-1}{2}$ is an integer and thus $\sum_{k=1}^{\frac{p-1}{2}} k^2 \in \mathbb{N}$. Also

$$\sum_{k=1}^{\frac{p-1}{2}} k^2 = \frac{(p-1)p(p+1)}{24} = \frac{(p^2-1)p}{24}$$

is an integer since $24 = 2^3 \cdot 3$ and $\gcd(p, 24) = 1, p > 3$. Now since p is a prime and gcd(p,24)=1 for all p>3, it follows that 24 has to divide $p^2 - 1$. i.e. $p^2 \equiv 1 \bmod 24$. □

An elementary yet fundamental property of primes is that all primes greater than 3 can be expressed as $6n \pm 1$. This is also the core property which is utilized in this work to generate a system to produce all composites of the same form, allowing one to determine the primes that are left over. The first known publication of this result is by Bungus in his book Numerorum Mysteria in 1599.

Theorem 1.3. *All primes $p_n > 3$ can be expressed in form $p_n = 6m \pm 1$.*

Proof. All natural numbers with remainders 0, 2, or 4 modulo 6 are divisible by 2 and numbers with remainders 0 or 3 modulo 6 are divisible by 3. Thus all primes greater than 3 must have a remainder of 1 or 5 modulo 6 i.e. they are of form $6n \pm 1$. □

1.4 Important results and history relating to primes

The Greeks are usually considered to have been the first to seriously study the nature of primes, although it is argued that Egyptian and Babylonian mathematicians had some degree of understanding of primes. Euclid's famous work Elements contains the first known definition for prime numbers and offers a proof for the infinitude of primes using a counterexample. He supposed that there is a largest prime p and constructed a number $q = (\prod_{p \geq p_i \in \mathbb{P}} p_i) + 1$ which clearly isn't divisible by any prime $\leq p$, thus producing a contradiction.

After Pythagoras and Euclid had made the study of the properties of numbers a subject worthy of the attention of Greek philosophers, an algorithmic method for the generation of primes was discovered by Eratosthenes. This algorithm forms the basis of modern prime sieves and is further discussed in chapter 4. Islamic mathematicians were the heirs of Greek wisdom throughout the Middle Ages, preserving and developing prime number theory slightly further. Prime related results were scarce in Europe until around 17th and 18th century when Fermat, Goldbach and others achieved groundbreaking results paving the way for further development. Goldbach conjectured that every even integer ≥ 2 can be expressed as the sum of two primes. Goldbach's conjecture remains still as one of the oldest unsolved problems in number theory. In 2013 Harald Helfgott proved the Goldbach's weak conjecture which states that every prime greater than 5 can be expressed as the sum of three primes.

One of Fermat's most influential achievements was his little theorem stated in 1640 that if p is a prime, then for any integer x, $x^p - x$ is a multiple of p i.e. $x^p \equiv x \mod p$. This is one of the fundamental theorems of number theory. The case $x = 2$ was known to the Chinese as early as 500 BCE. The first published proof was given by Euler in 1736.

Euler also proved an important result relating to primes in his thesis Variae Observationes Circa Series Infinitas in 1737, namely the product formula for, what later became known as the Riemann zeta function. He also proved that the sum of the reciprocals of primes diverges. The Riemann zeta function is considered one of the most important results in analytical number theory and the related unsolved hypothesis has perplexed mathematicians ever since it was introduced. Riemann introduced his influential ideas on the number of primes less than a given magnitude in 1859 by showing the connection between the distribution of the primes and the zeros of the analytically extended Riemann zeta function of a complex variable. The product formula for the zeta function gives insight to the relation of primes and the zeta function

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s} = \prod_{p_i \in \mathbb{P}} \frac{1}{1 - {p_i}^{-s}}.$$

For centuries mathematicians have been fascinated by the seemingly erratic distribution of primes. There doesn't seem to be an easy formula for the enumeration of primes. Considerable effort has been put forth to come up with approximative bounds and asymptotical formulas to shed light on the distribution of primes. One of the most important function in prime number theory is the prime counting function

$\pi(x)$, which counts the number of primes less than a given x. It is defined as $\pi(x) = \sum_{x \geq p_i \in \mathbb{P}} 1$. The π symbol of the function, as well as the big O notation, are due to Landau.

Gauss and Legendre conjectured in the 18th century that $\pi(n) \sim \frac{n}{\log(n)}$, which became known as the prime number theorem (PNT). Dirichlet communicated his slightly improved approximating function, the logarithmic integral Li(x) in a slightly different form compared to its modern definition, to Gauss. The PNT was proven independently by Hadamard and Vallée-Poussin in 1896 by extending the ideas presented by Riemann. Chebyshev tried to prove the PNT but managed to prove only a slightly weaker form, which was still strong enough to prove Bertrand's postulate, which states that there exists a prime between n and $2n$ for any integer $n \geq 2$. More formally $\pi(2n) - \pi(n) \geq 1, \forall n \geq 2$. The estimate for $\pi(n) \approx \int_2^n \frac{dt}{\log t}$ is suprisingly accurate. Staple calculated $\pi(10^{26}) = 1\,699\,246\,750\,872\,437\,141\,327\,603$. The estimate gives $\int_2^{10^{26}} \frac{dt}{\log t} \approx 1\,699\,246\,750\,872\,592\,073\,361\,408$. The error in the estimate is smaller than the square root of the actual prime count.

The frequent need of factors of numbers and the excessive labour required for their direct determination has inspired mathematicians off all ages to construct factor tables with continually increasing limit prior to the era of digital computers, e.g. Lehmer's Factor Table for the First Ten Millions and his list of prime numbers from 1 to 10 006 721. The reader interested in the history of (prime) number theory should refer to [4], which is an excellent book on the topic. In chapter XIII Dickson presents various papers enumerating primes of different forms in various intervals.

After Riemann's results and proof of the PNT mathematicians started looking at primes more in a probabilistic view rather than the old statistical view. The results of Hardy-Littlewood, Turán-Kubilius, Erdős-Wintner and Erdős-Kac among others are considered by some as the beginning of the era of probabilistic number theory around 1930s and 1940s. An interested reader may refer to [5] for some highlights on the history of probabilistic number theory.

2 Background on sieves

In this chapter we take a look at methods for generating prime numbers using sieving algorithms. A prime sieve is an algorithm for finding prime numbers by creating a list of all integers up to a desired limit and then progressively removing all the composite numbers in the interval until one is left with only prime numbers. The most well known prime number generating algorithm is the sieve of Eratosthenes, conceived in ancient Greek. It took almost two millennia until variations and faster extensions were crafted and published.

In 1934 Indian mathematician Sundaram published a simple deterministic sieve for prime number generation. After various breakthroughs in analytic number theory and the advent of computers and computational number theory, significant improvements have been achieved in reducing the time complexity and space complexity of prime sieves. The modern methods began to develop around 1970s and are still actively researched since many interesting problems in number theory and mathematics in general are directly related to the properties of prime numbers. Even after centuries of research, the prime numbers still mystify us and there are still many open questions related to them.

First we take a look at the classical sieve of Eratosthenes and analyze the algorithm, since it can be considered as the most fundamental algorithm in prime number generation and the basis for all efficient modern variations. Then we proceed to study how a sieve can be made linear in time complexity and further improve the results by introducing the theoretically important concept of wheel factorization. Wheels allow us to transform linear time sieves into sub-linear time complexity algorithms. For practical applications the concept of segmentation is also presented. Lastly we discuss modern developments such as the sieve of Atkin, probabilistic sieving and primality testing.

2.1 Sieve of Eratosthenes

The sieve of Eratosthenes is a simple algorithm for finding all prime numbers up to a given limit. It is still widely used today for finding relatively small primes, due the simplicity in its implementation. The algorithm is popularly used as a way to benchmark computer performance.

The sieve's conception is usually dated around 250 BCE and attributed to the Eratosthenes of Cyrene, a Greek mathematician and chief librarian at the library of Alexandria. He is also attributed to have invented the discipline of geography (including terminology used today) and being the first person to calculate accurately the circumference of Earth and the tilt of the Earth's axis. The earliest known reference to the algorithm is in Nicomachus of Gerasa's book Introduction to Arithmetic, dated around 100 AD.

The slightly improved "by hand" sieve of Eratosthenes [6] starts with a table of numbers from 2 to n and progressively crosses off numbers in the table until only prime numbers remain. Begin with the first number p in the table (i.e. $p = 2$) and

1. Declare p to be prime. Cross off all multiples of that number in the table starting from p^2.
2. Find the next uncrossed number in the table and set p to be that number. Repeat from step 1 until the end of the finite table is reached.

The starting point of p^2 is a minor optimization which can be made since the lower multiples are crossed off in earlier iterations. For finite tables of size n, once we have reached the $\sqrt{n}$:th index in the table, we can stop and declare the uncrossed values as prime numbers.

The algorithm can be expressed more formally by defining an array of bits of length n, presenting Boolean values indicating whether or not index i is prime or not. When the algorithm terminates, we have an array where the primality of $i \in [2, n]$ can be determined.

Algorithm 1 Sieve of Eratosthenes

```
function SieveOfEratosthenes(n)   ▷ n is the limit up to which primes are generated
    S[1] ← false                  ▷ 1 is neither prime nor composite, assign false
    i ← 2

    while i ≤ n do                ▷ Initialize array with true values
        S[i] ← true
        i ← i + 1
    end while

    p ← 2
    while p² ≤ n do               ▷ Main loop
        f ← p
        while f ≤ ⌊n/p⌋ do        ▷ Remove multiples of p starting from p²
            S[pf] ← false
            f ← f + 1
        end while

        p ← p + 1
        while S[p] == false do    ▷ Find the next prime
            p ← p + 1

        end while
    end while
end function
```

Sorenson [7] describes the algorithm in more abstract form and states that the more efficient variations of the algorithm differ from the sieve of Eratosthenes mainly in their implementations of the set S and in the way in which the removing of the composites happens. This is also the case for our linear algorithm presented in chapter 3.

By analyzing the algorithm for the sieve of Eratosthenes, we can see that it uses $O(n)$ bits of space for the array S. The initialization of the array takes $O(n)$ operations. The time spent removing the multiples is at most

$$\sum_{p \leq \sqrt{n}} \frac{n}{p} = O(n \log \log n).$$

This bound can be achieved by using the formula (cf. [2])

$$\sum_{(p \in \mathbb{P}) \leq n} \frac{1}{p} = \log \log n + O(1).$$

Note that the sum is taken over primes only. Finally, the time spent finding the next prime is the number of times we add one to p, which is at most $O(\sqrt{n})$.

The time spent removing the multiples clearly dominates in complexity and thus we can conclude that the asymptotic time complexity for the sieve of Eratosthenes is of order $O(n \log \log n)$. Note that some of the multiples may be crossed off more than once. This becomes a problem especially when n increases and we hit more and more composite numbers with multiple divisors. This problem can be remedied by finding ways in which each composite number is crossed off exactly once, thus producing a linear time complexity algorithm as we shall see next.

2.2 Linear sieve

In 1977 Mairson published theoretically significant improvements [8] to the sieve of Eratosthenes. He developed a linear time algorithm and used a wheel (using a different name for it) to improve the algorithm's time complexity to be sublinear $O(\frac{n}{\log \log n})$. Unfortunately he had to bring in multiplications to achieve these bounds, raising the space complexity to be $O(\frac{n \log n}{\log \log n})$ bits, instead of $O(n)$ bits required by the original sieve of Eratosthenes. The linear sieve algorithm presented by Mairson deletes all composite numbers whose lowest prime factor is p from a list S, which will contain only primes after the algorithm terminates. The algorithm first uses this set to compile a list of these composites, then proceeds to delete them from S using an auxiliary array.

Euler's proof of the product formula for the Riemann zeta function contains a version of the sieve of Eratosthenes in which each composite number is removed exactly once. This linear sieve was rediscovered by Gries and Misra in 1978 [9]. Since

then many researchers have crafted linear time algorithms and other improvements, including Pritchard in the early 1980s and Bengelloun in 1986. The latest one being the sieve of Atkin published in 2004.

Bengelloun designed an incremental algorithm using a data structure to store sufficient information so that, if the primes up to n are known, the primality of $n+1$ can be determined in constant time $O(1)$, requiring a total of $O(n \log n)$ bits of space. Such an algorithm is useful in applications where an upper bound on the primes needed is unknown beforehand.

The problem in the sieve of Eratosthenes is that a composite number c may be crossed off more than once, especially if c has multiple divisors. Ideally we would like to remove each composite number from the array exactly once. If each removal takes constant time $O(1)$, we can improve the asymptotical time complexity of the sieve of Eratosthenes to be $O(n)$.

Gries and Misra achieved this by assuming multiplications of integers not larger than n to be performable in constant time. The algorithm is similar to Mairson's but according to them "perhaps simpler and more elegant". Their version is also extendable to find the prime factorizations of all integers between 2 and n in $O(n)$ time complexity. Their paper demonstrates the elegant algorithm and shows its correctness and linearity.

The structural implementation of the array S varies between methods but the real differences are in the methods in which composites are removed. Mairson's method makes a pass through S to find all composite numbers and copies them in to an auxiliary array. Then it makes a pass through this array to generate values of f to remove all the composites $f \cdot p$ from S.

The solution proposed by Gries and Misra removes composites of the form $f \cdot p^k$, $k \in \mathbb{N}$ until $f \cdot p^k$ is larger than n, instead of just removing composites $f \cdot p$ from S. Pritchard's method finds the largest composite number first and works downwards.

These algorithms are based on a theorem which states that a composite number c can be uniquely written as $c = q \cdot p^k$, where

1. p is a prime and $p = \mathrm{lp}(c)$
2. $k \in \mathbb{N}$
3. $p = q$ or $p \leq \mathrm{lp}(q)$.

Here $\mathrm{lp}(i)$ denotes the lowest prime number that divides i. The proof can be found in their original paper. A similar idea is used in our linear algorithm.

Sorenson divides extensions and improvements to the sieve of Eratosthenes in to two broad categories:

1. Algorithms which disregard each composite number exactly once, thus using $O(n)$ arithmetic operations, and
2. Algorithms which use only $O(\sqrt{n})$ bits of memory.

Category 1 linear algorithms can be transformed in to sublinear time-complexity using wheels as we shall see next. It is an open problem whether or not one can create an algorithm with sub-linear time complexity using only $O(\sqrt{n})$ bits of memory, such an algorithm would belong to both categories [7].

2.3 Sub-linear sieve with wheels

Improvements to the sieve of Eratosthenes made by Mairson contained a method which allows one to transform linear time prime sieves into sub-linear time complexity using method which later conjured the concept of a wheel. The idea can be also used in other applications than prime number sieves. The formal definition and the naming is due to Pritchard in 1981, who introduced it [11].

Let us define

$$M_k = \prod_{i=1}^{k} p_i$$

$$W_k = \{x | 0 \leq x < M_k \wedge \gcd(x, M_k) = 1\}$$

$$W_k(n) = \{x \leq n | \gcd(x, M_k) = 1\}.$$

Here W_k denotes the k:th wheel and contains the integers between 0 and $M_k - 1$ which are relatively prime to the first k primes. $W_k(n)$ is the k:th wheel extended to n.

The data structure for wheels k allows one to determine in constant time whether or not an integer is in the extended wheel (i.e. relatively prime to M_k) and what the next largest element in the extended wheel is. Sorenson explains the process to compute the data structure:

1. Use trial division to find the first k primes. Compute M_k.
2. Sieve the wheel array $W[\cdot]$ so that $W[x] = 0$ or 1 depending on whether or not $\gcd(x, M_k) = 1$. The entries of 1s will be changed in the next step.
3. Set $W[M_k - 1] = 2$, and make a pass over the array starting from $M_k - 1$ going down. Save the previous x such that $W[x]$ was nonzero. When the next smallest nonzero element is found, store the difference. As a check, the value of $W[1]$ should be $p_{k+1} - 1$.

If one wants the previous element in the extended wheel, wheel symmetry can be used. The element previous to x is $x - W[M_k - (x \bmod M_k)]$. Using the data structure defined above, one can convert linear prime sieving algorithms into sub-linear

algorithms using the following steps, as explained by Sorenson

1. Choose k as large as possible such that $M_k \leq \sqrt{n}$. Find the first k primes and compute $W[x]$ for all $x < M_k$.

2. Initialize S to the kth wheel extended to n, $W_k(n)$. S can be implemented as a doubly linked list using arrays.

   $d := 1$
   do
   $\quad S := S \cup \{d\}$
   $\quad d := d + W[d \bmod M_k]$
   $while(d > n)$

3. Run the linear algorithm starting from $p := p_{k+1}$.

4. Output $S \setminus \{1\}$ and the first k primes.

The sublinear time complexity of this algorithm follows from the fact that S is initialized to contain only $O(\frac{n}{\log\log n})$ elements. Thus the linear sieving phase of the algorithm uses only $O(\frac{n}{\log\log n})$ arithmetic operations. Note that the constants in these asymptotical complexities are so large that these algorithms are mainly of theoretical interest for all practical values of n. For practical ranges of primes, the actual run-time of the sieve of Eratosthenes in a modern computer usually outperforms these methods due to significantly lower asymptotic constants and the simpler Boolean array used.

Pritchard also describes improvements to efficiency at a bit complexity level, how to reduce the operations to only additions and how to reduce the storage requirements. The space reductions however are mainly theoretical. He also presented the segmented wheel sieve in 1983. An interested reader can refer to his comprehensive paper [10] for various pseudocodes for linear sieves along with great analysis as well as a comprehensive family tree describing how each algorithm descends from the sieve of Eratosthenes and relates to one another.

2.4 Segmented sieve

According to Sorenson, the most practical improvement to the Sieve of Eratosthenes for large values of n is the idea of segmentation. This is largely due to the fact that RAM access typically becomes the speed bottleneck more than computational speed once the array size starts to grow beyond the size of the processor caches. The segmented sieve of Eratosthenes requires $O(\sqrt{n})$ space and has the same time complexity as the original algorithm. The time complexity can be improved to be linear (at the expense of space). Page segmented wheel sieves require significantly more space to store the required wheel presentations. For a detailed description on segmentation see Bays and Hudson [12].

The underlying idea of the segmented sieves is quite simple. Instead of sieving the desired interval from 2 to n at once, one can break the interval into segments of length Δ. After $\frac{n}{\Delta}$ intervals have been sieved, all primes up to n have been found. The space complexity is $O(\sqrt{n} + \Delta)$ suggesting $\Delta = \sqrt{n}$. Thus we get space complexity of order $O(\sqrt{n})$.

Unfortunately there is no apparent way to segment the linear and sub-linear sieving algorithms due to the fact that S uses a linked list presentation. However Pritchard's segmented wheel sieve runs in $O(n)$. In practice the fastest run-times for sufficiently large n are achieved using segmented wheels. For parallel implementations, Sorenson suggests to assign each processor intervals of length Δ.

2.5 Modern sieves

Legendre studied $\pi(x)$ using the sieve of Eratosthenes to find upper or lower bounds on the number of primes within a given set of integers. His extension to the sieve of Eratosthenes, named the Legendre sieve, is seen as the backbone to modern sieve theory which is considered to have begun around 1920s when Brun pioneered the field. Some noteworthy sieves in sieve theory are Brun's pure sieve, Selberg's sieve, the Large sieve and the Asymptotic sieve. Sieve theory is a rich and complex topic deemed to be outside the scope of this thesis. An interested reader may refer to e.g. [21] for further insight.

As the processing power of computers has increased, many alternative ways for the generation of primes have been suggested. These include the probabilistic Monte-Carlo variation of the sieve of Eratosthenes, also known as the Hawkin's random sieve [13] and probabilistic primality tests which are examined more in depth in the next section. In 2003 Atkin and Bernstein published a deterministic sieve of Atkin [14] which is of linear time complexity. After some preliminary work the sieve of Atkin crosses off the squares of primes.

With the increased computational power and the advent of internet, distributed computing came in to the picture. In 1996 the Great Internet Mersenne Prime Search GIMPS (www.mersenne.org) was founded to harness the spare computer cycles to search for the new Mersenne primes of form $2^n - 1$ using the Lucas-Lehmer test. In December 2017 the project found the 50th known Mersenne prime, which is $2^{77232917} - 1$. In recent years GPUs (Graphics Processing Units) have been used for scientific computing as well due to available parallel processing which speed up calculations considerably. CUDASieve (www.github.com/curtisseizert/CUDASieve) for example uses segmented sieve of Eratosthenes and seems to run significantly faster than the highly optimized parallel CPU version (www.primesieve.org). The CUDASieve reports that on a modern high-end GPU it can determine primes between 0 and 10^{12} in 12.5 seconds while the CPU version takes about 1 minute on a modern highend CPU running in 4 threads.

According to the prime number theorem $\pi(n) \sim \frac{n}{\log(n)}$, where $\pi(n)$ is the prime-counting function. From this theorem we can see that for sufficiently large n, the probability of a random integer $m \leq n$ being prime is close to $\frac{1}{\log(n)}$. The use of randomness can be thus leveraged to create a Monte Carlo variation of the sieve of

Eratosthenes, although this is more of a theoretical curiosity rather than a practical implementation.

2.6 Primality testing

In this subsection we present methods for testing primality. We start by introducing simple and slow methods and proceed to more complex and fast procedures. We start from deterministic methods by introducing the elementary trial division method and application of the Wilson's theorem. We then proceed to probabilistic primality tests, namely the Fermat's primality test and the probabilistic Miller-Rabin test. Lastly we discuss Miller-Rabin test's deterministic variant and the AKS primality test. The deterministic tests are guaranteed to prove or disprove primality whereas the probabilistic methods (in some intervals) can pass composites as primes. Usually the probabilistic methods are faster.

A naive way of checking for primality of n is to try to divide it with all integers from 2 to n. This is called the trial division method and it is very inefficient for large enough n, considered more as a baseline for faster methods. An obvious optimization is to limit the division up to $\sqrt{n}$ thus taking $O(\sqrt{n}(\log n \log\log n))$ operations, where the logarithmic part comes from the time complexity of the divisions (using Newton-Raphson division algorithm).

Wilson's theorem states that $n > 1$ is prime iff $(n-1)! \equiv -1 \bmod n$ [2]. Thus one can compute $(n-1)! \bmod n$ and determine wether n is prime or not. While being an interesting result, the computation of the factorial is infeasible for sufficiently large n and the method is even slower than the optimized trial divison.

Fermat's little theorem states that when $p \in \mathbb{P}$ and $\gcd(p, x) = 1$, $x^{p-1} \equiv 1 \bmod p$ [2]. Based on this congruence a probabilistic primality test can be crafted by randomly picking $1 < x < p-1$ not divisible by p and checking wether the congruence holds. The method is based on the fact that it is unlikely that the congruence holds for random x if p is composite. For higher confidence the test is repeated multiple times with different x.

Carmichael showed in 1910 that there are infinitely many (composite) Carmichael numbers for which any x satisfies Fermat's little theorem [15], the smallest one being 561. This means that there are infinitely many composites which the Fermat's primality test passes erroneously as prime. Carmichael numbers are however somewhat rare and despite its shortcomings the Fermat's primality test is fast (polylogarithmic).

To combat the problem of Carmichael numbers, variants of the Fermat's test were crafted. Solovay-Strassen primality test [16] uses a generalization of Fermat's theorem, namely Euler-Jacobi pseudoprimes. It is historically important as it showed the practical feasibility of the RSA cryptographic system.

Miller added a condition to an advanced variant of Fermat's test and presented a deterministic primality test [17] based on the validity of the extended Riemann Hypothesis. Rabin modified Miller's test [18] and made the test probabilistic, allowing to drop the assumption of the extended Riemann's Hypothesis. There still exists a small chance of erroneuosly classifying composites as prime. However it has been verified that the primality of $p < 341\,550\,071\,728\,321$ is guaranteed if the test is run

with $x = 2, 3, 5, 7, 11, 13, 17$.

Another probabilistic primality test worth mentioning is the Bailie-PSW primality test [19] which utilizes the strong Fermat probable primality test and the Lucas probable prime test. It has been verified not to pass any composite under 2^{64} as a prime.

In 2002 Indian computer scientist published the AKS primality test [20] named after the creators Agrawal, Kayal and Saxena. Miller's deterministic test runs in polynomial time but it is conditioned on the validity of the extended Riemann's Hypothesis. Adleman, Pomerance and Rumely improved the time complexity of deterministic primality tests out of exponential time and probabilistic Elliptic Curve methods could bring the expected time to be polynomial but the AKS primality test was the first deterministic test to unconditionally prove that primality can be solved in polynomial time. However the test is not practical on computers and currently the probabilistic tests are seen as best for practical primality testing purposes.

3 The Myriad sieving algorithm

This chapter presents the proposed system for the generation of primes. We start with the geometrical pattern observed in the squares of primes which ultimately led to the construction of our prime sieve variation. The geometrical pattern gives rise to the concept of a generator function and its inverse which can be used to generate a composite index generating function. This composite index generating function is the backbone of the presented algorithm. Then we take a look at the initial version of the prime sieving algorithm and its similarities to the sieve of Eratosthenes. Lastly the algorithm is optimized to be of linear time complexity.

3.1 The geometrical pattern

By considering the integer valued terms of the sequence $f_p(n) = \frac{{p_n}^2-1}{8}$ we can observe that there seems to exist an arithmetic progression in the sequence in the form of a cumulative sum. This is indeed true, since

$$f_p(n) = \sum_{k=1}^{\frac{p_n-1}{2}} k = (\frac{1}{2})(\frac{p_n-1}{2})(\frac{p_n+1}{2}) = \frac{{p_n}^2-1}{8}. \tag{1}$$

We can visualize the more general cumulative sum $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$ with unit squares as shape which we will refer to as a stepshape, starting with 1 unit square and keep growing the number of units to produce a shape of steps stopping when the height of the steps is n. Let us consider a stepshape which has an area mimicing the sequence $f_p(n)$. Firstly we notice that for all $p_n > 3$ the members of the sequence are divisible by 3 which indicates that stepshapes with sidelength of form $1+3k, k \in \mathbb{N} \cup \{0\}$, need to be skipped. In order to achieve this we can consider three types of elemental shapes which produce the interesting set of stepshapes as illustrated in Figure 1.

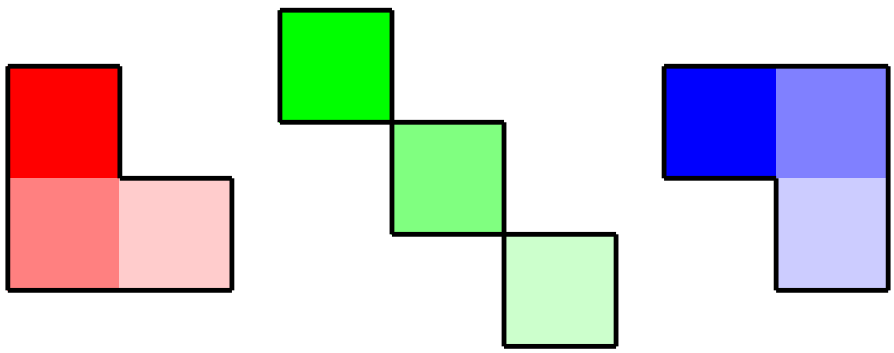

Figure 1: The three elementary shapes which make up the stepshapes.

Each elemental shape has an area of 3 and the diagonal part has a sidelength of 3 whereas the other have a sidelength of 2. The parts fit together in periods as indicated by the Figures 2 and 3 to grow the stepshape along the diagonal. Note that the possible areas of stepshapes of this form go through numbers which don't belong to the sequence $\frac{{p_n}^2-1}{8}, n > 3$ but this "problem" will turn out to reveal important properties of composites in the mimicking sequence. The skipping of certain indices results in a sidelength, as a function of n, of the form $3\lfloor\frac{n}{2}\rfloor + 2(n \bmod 2)$. Thus the area of the stepshape is $A(n) = \sum_{k=1}^{3\lfloor\frac{n}{2}\rfloor+2(n\, mod\, 2)} k$. Two such stepshapes are illustrated below with $n = 3$ in Figure 2 and $n = 8$ in Figure 3.

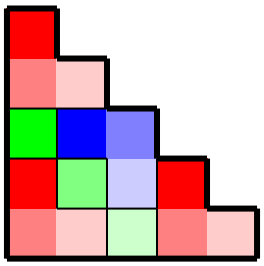

Figure 2: The stepshape corresponging to $n = 3$.

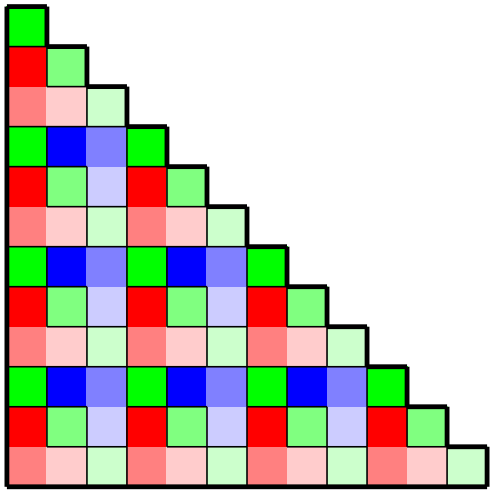

Figure 3: The stepshape corresponging to $n = 8$.

By considering the number of each elemental shape in the n:th stepshape we get Table 1 for the first such shapes. The sum of the count of each element is multiplied by 3 to get the total area as each elemental shape is of area 3. Notice that the last index $n = 8$ doesn't produce a value of the sequence $f_p(n)$.

Table 1: Amounts of elemental shapes in various stepshapes

n	1	2	3	4	5	6	7	8	
∟	1	1	3	3	6	6	10	10	$f_0(n)$
⋱	0	1	1	3	3	6	6	10	$f_0(n-1)$
¬	0	0	1	1	3	3	6	6	$f_0(n-2)$
Σ	1	2	5	7	12	15	22	26	
3Σ	3	6	15	21	36	45	66	78	

Notice how the number of each element grows as the cumulative sum $\sum_{k=1}^{n} k$ but each value is repeated twice due to phases in the periodicity in which they have been placed to the stepshape. We can present this by defining

$$f_0(n) = \sum_{k=1}^{\lfloor \frac{n+1}{2} \rfloor} k = \frac{1}{2}(\lfloor \frac{n+1}{2} \rfloor^2 + \lfloor \frac{n+1}{2} \rfloor).$$

Thus we can present the area of the stepshape with f_0 as

$$A(n) = 3(f_0(n) + f_0(n-1) + f_0(n-2)).$$

By solving for p_n in f_p we get $p_n = \sqrt{1 + 8f_p(n)}$. If we now consider $A(n)$ and our original sequence $f_p(n)$, we can notice that $A(n) = f_p(n)$ iff $n = \lfloor \frac{p_{k+2}}{3} \rfloor, k \in \mathbb{N}$ and thus one can divide the set of natural numbers to two types of indices, ones which produces primes and another which does not. This forms the basis for the indexing system for the prime generating function which are discussed in more detail in later sections.

By solving for p_n^2 in f_p and using $f_p = 3(f_0(n) + f_0(n-1) + f_0(n-2))$ (when n is of form $\lfloor \frac{p_{k+2}}{3} \rfloor$), one can obtain an interesting form for the squares of primes. This form of equation (2) can be used alternatively to prove the modular congruence $p_n^2 \equiv 1 \bmod 24$ and provides a way to graphically present this congruence. Figure 4 illustrates how 8 stepshapes wrapped around a central unit produce a prime square with stepshapes where $n = \lfloor \frac{p_i}{3} \rfloor$ for some i. Here $n = 7$ producing a square of sidelength 23. Note that when $n = \lfloor \frac{p_i}{3} \rfloor$, the sidelength of a single stepshape is $\frac{p_i - 1}{2}$ and thus the squares sidelength is $1 + 2\frac{p_i - 1}{2} = p_i$.

$$p_n^2 = 1 + 24(\sum_{k=1}^{\lfloor \frac{p_n+3}{6} \rfloor} k + \sum_{k=1}^{\lfloor \frac{p_n}{6} \rfloor} k + \sum_{k=1}^{\lfloor \frac{p_n-3}{6} \rfloor} k). \tag{2}$$

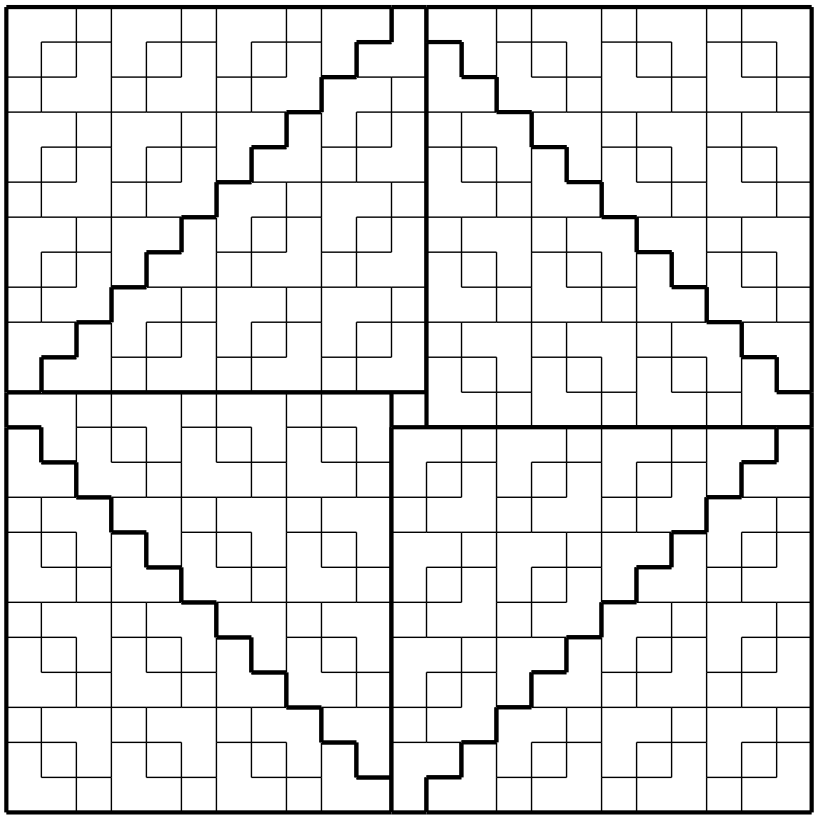

Figure 4: 8 stepshapes corresponging to $n = 7$ wrapped around a central unit produces a prime square.

3.2 The generator function

If we take the squareroot in equation (2) and open the sums, we are left with an expression

$$p_n = \sqrt{1 + 12(\lfloor \frac{p_n + 3}{6} \rfloor^2 + \lfloor \frac{p_n + 3}{6} \rfloor + \lfloor \frac{p_n}{6} \rfloor^2 + \lfloor \frac{p_n}{6} \rfloor + \lfloor \frac{p_n - 3}{6} \rfloor^2 + \lfloor \frac{p_n - 3}{6} \rfloor)}. \quad (3)$$

By allowing the cumulative sums in equation (2) to take also composite values as opposed to only primes, we can define a function $G(n) : \mathbb{N} \to \mathfrak{G}$ which goes through all the numbers greater than 1 which are 1 or 5 modulo 6, i.e. of form $6n \pm 1$. We will call this function the generator function. The generator function has a simpler form as shown in the following lemma. Based on the equation (3) we can extend it beyond the primes and obtain

$$G(n) = \sqrt{1 + 12(\lfloor \frac{n + 1}{2} \rfloor^2 + \lfloor \frac{n + 1}{2} \rfloor + \lfloor \frac{n}{2} \rfloor^2 + \lfloor \frac{n}{2} \rfloor + \lfloor \frac{n - 1}{2} \rfloor^2 + \lfloor \frac{n - 1}{2} \rfloor)}. \quad (4)$$

Lemma 3.1. $G(n)$ can be expressed as $G(n) = 6\lfloor \frac{n+1}{2} \rfloor + (-1)^n = 3n + \frac{3-(-1)^n}{2}$.

Proof. Consider n to be even i.e. $n = 2k$. Then $G(n) = \sqrt{1 + 12(k(3k+1))}$ and $G(n) = 6k+1$. Considering that both functions are positive and squaring them yields a tautology.

Similarily when n is odd i.e. $n = 2k-1$. Then $G(n) = \sqrt{1 + 12(k(3k-1))}$ and $G(n) = 6k-1$. Considering that both functions are positive and squaring them yields a tautology.

□

Obviously there are infinitude of forms expressing the same sequence but the form of G in Lemma 3.1 nicely illustrates the relation to numbers without factors of 2 or 3, as opposed to the form (4). Since G goes through all natural numbers greater than 1 without factors of 2 or 3, the function will generate all primes greater than 3 with certain set of indices but also all the composites, which are not multiples of 2 or 3, with another set of indices. From the perspective of sieving, this is nice because we don't have to worry about multiples of 2s or 3s and can concentrate on multiples of higher primes. There is a certain kind of similarity to the idea of wheel factorization although arguably these are different types of methods.

Based on the form $3n + \frac{3-(-1)^n}{2}$ it can be seen that G has an inverse function defined as $G^{-1}(n) = \lfloor \frac{n}{3} \rfloor$. This results in the fact that $G(\lfloor \frac{p_n}{3} \rfloor) = p_n$.

Lemma 3.2. $G(\lfloor \frac{g}{3} \rfloor) = g$, where $g \in \mathfrak{G}$. Thus $G^{-1}(n) = \lfloor \frac{n}{3} \rfloor$.

Proof. Suppose $g = 6k-1$. Then $G^{-1}(g) = \lfloor \frac{6k-1}{3} \rfloor = 2k-1$ and $G(2k-1) = 3(2k-1)+2 = 6k-1 = g$. Suppose $g = 6k+1$. Then $G^{-1}(g) = \lfloor \frac{6k+1}{3} \rfloor = 2k$ and $G(2k) = 3(2k)+1 = 6k+1 = g$. □

The first 25 values of $G(n)$ are evaluated in the Table 2 and the values from 26 to 50 are evaluated in the Table 3 to illustrate the density of primes when n is small. As n grows, $G(n)$ starts to hit more and more composites.

Table 2: Values of $G(n)$, for $n \in [1,25]$.

n	**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**
$G(n)$	5	7	11	13	17	19	23	25	29	31	35	37	41	43	47	49	53	55	59	61	65	67	71	73	77
Factorization	-	-	-	-	-	-	-	$5 \cdot 5$	-	-	$5 \cdot 7$	-	-	-	-	$7 \cdot 7$	-	$5 \cdot 11$	-	-	$5 \cdot 13$	-	-	-	$7 \cdot 11$

Table 3: Values of $G(n)$, for $n \in [26,50]$.

n	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**	**43**	**44**	**45**	**46**	**47**	**48**	**49**	**50**
$G(n)$	79	83	85	89	91	95	97	101	103	107	109	113	115	119	121	125	127	131	133	137	139	143	145	149	151
Factorization	-	-	$5 \cdot 17$	-	$7 \cdot 13$	$5 \cdot 19$	-	-	-	-	-	-	$5 \cdot 23$	$7 \cdot 17$	$11 \cdot 11$	5^3	-	-	$7 \cdot 19$	-	-	$11 \cdot 13$	$5 \cdot 29$	-	-

The relation of $G(n)$ to primes, or more precisely its range's ($\mathfrak{G}$) relation to primes, can be seen from the following Theorem 3.3.

Theorem 3.3. $\mathfrak{G} \setminus \mathfrak{G}^* = \mathbb{P} \setminus \{2,3\}$.

Proof. The numbers of form $6k \pm 1$ (i.e. $\mathfrak{G}$) minus the composites in $\mathfrak{G}$ (i.e. $\mathfrak{G}^*$) is the set of prime numbers in $\mathfrak{G}$ i.e. $\mathbb{P} \setminus \{2,3\}$. More formally, by the fundamental theorem of arithmetic each natural number can be (uniquely) expressed as a product of its prime factors. Using this theorem, we can present the sets $\mathfrak{G}$ and $\mathfrak{G}^*$ as

$$\mathfrak{G} = \{ \prod_{3<p_k\in\mathbb{P}}^{\infty} p_k^{s_k} | s_k \in \mathbb{N} \cup \{0\}, \sum s_k \geq 1\}.$$

$$\mathfrak{G}^* = \{ \prod_{3<p_k\in\mathbb{P}}^{\infty} p_k^{s_k} | s_k \in \mathbb{N} \cup \{0\}, \sum s_k \geq 2\}.$$

From these presentations it can be seen that the difference of these sets is

$$\mathfrak{G} \setminus \mathfrak{G}^* = \{ \prod_{3<p_k\in\mathbb{P}}^{\infty} p_k^{s_k} | s_k \in \mathbb{N} \cup \{0\}, \sum s_k = 1\}.$$

Since the set of numbers with exactly one prime factor is the set of primes and we have excluded the primes 2 and 3, we have

$$\mathfrak{G} \setminus \mathfrak{G}^* = \mathbb{P} \setminus \{2,3\}.$$

□

Benoit Cloitre in OEIS (www.oeis.org/A007310) notes a property which allows one to present the set using the Euler's totient function, $\mathfrak{G} = \{n > 1 \mid \varphi(4n) = \varphi(3n)\}$.

3.3 Composite generating indices

A nice property of $G(n)$ is the simplicity of its inverse function $G^{-1}(g) : \mathfrak{G} \to \mathbb{N}$ defined as $G^{-1}(g) = \lfloor \frac{g}{3} \rfloor$. This inverse function can be used to extract the indices for which $G(n)$ generates primes, but it can also be used to get a formula for the indices which generate all composites without factors 2 or 3 (if one is interested in these composites, they can be trivially calculated by $2n$ and $3n$, $n \in \mathbb{N}$). Consider the indices I for which $G(I)$ is a multiple of 5,7,11,13,...,$G(k)$.

For 5, we find indices (1) +7(=8) +3(=11) +7(=18) +3(=21) +...

For 7, we find indices (2) +9(=11) +5(=16) +9(=25) +5(=30) +...

For 11, we find indices (3) +15(=18) +7(=25) +15(=40) +7(=47) +...

For 13, we find indices (4) +17(=21) +9(=30) +17(=47) +9(=56) +...

The obvious pattern can now be exploited to obtain few generalizations of alternative forms, which are provable trough induction. Consider an indexing function $I(n,m) : \mathbb{N}^2 \to \mathbb{N}$ of the form:

$$I(n,m) = I(m,n) =$$
$$G(n+1) \cdot m + \lfloor \frac{G(n+1)}{2} \rfloor + (-1)^m \lfloor \frac{n+1}{2} \rfloor =$$
$$n + (4n + 2 + (-1)^{n+1}) \lfloor \frac{m+1}{2} \rfloor + (2n+1) \lfloor \frac{m}{2} \rfloor =$$
$$n + (3(2n+1) + (-1)^{n+1}) \lfloor \frac{m}{2} \rfloor + (2(2n+1) + (-1)^{n+1})(\frac{1 + (-1)^{m+1}}{2}).$$

It might be more obvious to see the symmetry and to obtain a simpler formula by simply using the fact that elements in $\mathfrak{G}^*$ are of form $G(k) = G(n)G(m)$. Using the inverse function G^{-1} on these elements to obtain

$$I(n,m) = \lfloor \frac{G(n)G(m)}{3} \rfloor =$$
$$12 \lfloor \frac{n+1}{2} \rfloor \lfloor \frac{m+1}{2} \rfloor + (-1)^n 2 \lfloor \frac{m+1}{2} \rfloor + (-1)^m 2 \lfloor \frac{n+1}{2} \rfloor - \frac{1 + (-1)^{n+m+1}}{2}.$$

Note also that $G(I(n,n)) = G(n)^2$. The form of the indexing function can be further simplified if we consider the even and odd indices separately.

$$I_{odd} = 12pq \pm 2p \mp 2q - 1.$$

$$I_{even} = 12pq \pm 2p \pm 2q.$$

Computationally the simples form might be to use $G(n) = 3n + \frac{3-(-1)^n}{2}$ which leads to

$$I(n,m) = \lfloor \frac{G(n)G(m)}{3} \rfloor = 3nm + ng + mk + s, \tag{5}$$

where

$$k = \begin{cases} 2 & n \text{ odd} \\ 1 & n \text{ even} \end{cases}$$

$$g = \begin{cases} 2 & m \text{ odd} \\ 1 & m \text{ even} \end{cases}$$

$$s = \begin{cases} 1 & n \text{ odd} \wedge m \text{ odd} \\ 0 & \text{otherwise.} \end{cases}$$

These are easily calculated by checking the last bits of n and m. Next we will present our initial sieving algorithm based on this idea of composite indices.

3.4 The initial sieving algorithm

Consider the 2-dimensional multiplication table 4 of elements from $\mathfrak{G}$, i.e. multiplying $G(n)$ by $G(m)$. Because multiplication is commutative, one needs to only consider the upper triangle of the multiplication matrix, illustrated in the below table.

Table 4: Values of $G(n)G(m)$. Elements of $\mathfrak{G}^*$.

$\mathfrak{G}$	n	1	2	3	4	5	6	7	8	9	10	11	12	13
m	$G_{n,m}$	5	7	11	13	17	19	23	25	29	31	35	37	41
1	5	25	35	55	65	85	95	115	125	145	155	175	185	205
2	7		49	77	91	119	133	161	175	203	217	245	259	287
3	11			121	143	187	209	253	275	319	341	385	407	451
4	13				169	221	247	299	325	377	403	455	481	533
5	17					289	323	391	425	493	527	595	629	697
6	19						361	437	475	551	589	665	703	779
7	23							529	575	667	713	805	851	943
8	25								625	725	775	875	925	1025
9	29									841	899	1015	1073	1189
10	31										961	1085	1147	1271
11	35											1225	1295	1435
12	37												1369	1517
13	41													1681

By considering only the non duplicate entries when the table is extended to infinity (the duplicates and corresponding indices have been underlined), one can construct the Table 5 of composite generating indices. The indices generating duplicate composites have been left out from the table below.

Table 5: Values of $I(n, m)$.

$I_{n,m}$	n	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
m	$G_{n,m}$	**5**	**7**	**11**	**13**	**17**	**19**	**23**	**25**	**29**	**31**	**35**	**37**	**41**
1	**5**	**8**	**11**	**18**	**21**	**28**	**31**	**38**	**41**	**48**	**51**	**58**	**61**	**68**
2	**7**		**16**	**25**	**30**	**39**	**44**	**53**		**67**	**72**		**86**	**95**
3	**11**			**40**	**47**	**62**	**69**	**84**		**106**	**113**		**135**	**150**
4	**13**				**56**	**73**	**82**	**99**		**125**	**134**		**160**	**177**
5	**17**					**96**	**107**	**130**		**164**	**175**		**209**	**232**
6	**19**						**120**	**145**		**183**	**196**		**234**	**259**
7	**23**							**176**		**222**	**237**		**283**	**314**
8	**25**													
9	**29**									**280**	**299**		**357**	**396**
10	**31**										**320**		**382**	**423**
11	**35**													
12	**37**												**456**	**505**
13	**41**													**560**

Naively one could go through all n and m ($n \geq m$) such that $G(n)G(m) \leq G_{max}$, where G_{max} indicates the limit to which we want to sieve. Note that this limit can be any natural number and not of form $G(I_{max})$ as the G^{-1} will round the limiting index accordingly i.e. $I_{max} = \lfloor \frac{G_{max}}{3} \rfloor$. However there are multiple optimizations to be made by skipping elements in the matrix which have already been determined to be composite.

In the naive method start by setting $x = y = 1$, calculating $G(k) = G(x)G(y)$ and determine it to be composite by crossing it off from our list of primes (setting a Boolean value false to the corresponding index). Keep increasing x and calculate the composites indices until we hit I_{max}. Increment y by 1, and initialize $x = y$ to the beginning of the diagonal and scan through the row, crossing off composite indices until we hit I_{max}, again. Once y reaches $\lfloor \frac{\sqrt{G_{max}}}{3} \rfloor$, we can scan the row for the last time and we have gone through the composites in the interval. The limit for y is simply obtained from the fact $G(I(n, n)) = G(n)^2$. This method allows one to determine all the composites of form $6n \pm 1$ in the sieving interval and leaves the primes of this form untouched.

An obvious optimization is to calculate only the rows with prime-producing indices by skipping all rows y which have been determined to be composite indices, since all the composites in this row have been crossed off at the level of the smallest prime factor of the corresponding composite. Instead of crossing off the composites themselves we can simply cross off the corresponding indices, since they are slightly smaller. The untouched prime indices can then be passed through the prime

generator function G to get all primes (greater than 3) in the interval. This gives us an algorithm quite similar to the sieve of Eratosthenes, at least in time complexity. Consider the pseudocode of algorithm 2.

Algorithm 2 Initial MyriadPrime

```
function InitialMyriadPrimeSieve(n)  ▷ n is the limit up to which primes are generated
    I_max ← ⌊n/3⌋
    i ← 1

    while i ≤ I_max do                          ▷ Initialize array with Boolean values
        primeIndices[i] ← true
        i ← i + 1
    end while

    y ← 0

    while y + 1 ≤ ⌊√n/3⌋ do                                          ▷ Main loop
        y ← y + 1
        x ← y
        if primeIndices[y] = true then
            I ← I(x, y)
            while I ≤ I_max do
                primeIndices[I] ← false
                x ← x + 1
                I ← I(x, y)
            end while
        end if
    end while

    i ← 1
    while i ≤ I_max do                  ▷ Generate the primes based on the indices
        if lowestPrimeIndice[i] > 0 then
            G(i) ∈ ℙ                                  ▷ Print or save the prime G(i)
        end if
        i ← i + 1
    end while
end function
```

Note that e.g. 175 can be expressed as $G(8)G(2)$ or $G(11)G(1)$ (this is the smallest composite appearing more than once) and the original algorithm will calculate and cross it off twice. This problem stems from the number of distinct divisors in a composite number. The asymptotic time complexity of this algorithm can be deduced from the fact that the number of composites crossed off is

proportional to the sum of the distinct divisors of these composites. One can use, as an upperbound, the summation formula of the distinct divisors function (cf. [2])

$$\sum_{G_{max} \geq g \in \mathfrak{G}^*}^{n} \omega(g) < \sum_{k=2}^{n} \omega(k) = n \log \log n + M \cdot n + O(\frac{n}{\log n}),$$

where M is Merten's constant. This gives us a similar asymptotic time complexity of $O(n \log \log n)$ as with the sieve of Eratosthenes. Since we are usign a Boolean array for the determination of prime indices, the space complexity is similar to the sieve of Eratosthenes, i.e. $O(n)$ bits. We can improve the algorithm by introducing a way in which each composite in the interval gets crossed off exactly once. A method for obtaining a linear sieving algorithm based on this framework is presented next.

3.5 Linear sieving algorithm

The original sieving algorithm can be improved to be of linear time complexity by crossing off each composite only once. We can achieve this by scanning the columns instead of rows and saving the number of the row y in which the corresponding composite index $I(x, y)$ was first discovered. Now for x which are prime indices, we have to scan the entire column (until we hit the limiting index or $y = x$) skipping only rows y which correspond to a composite indices. For x which are composite indices, we can stop when y exceeds the level indicated by the saved values or until we hit the limiting index.

We are saving the index of the lowest prime factor of the composite and can safely stop there, since eventually we are going to cross off the remaining composites at the lower level of the smallest prime factor. We can initialize the saved values as 0 and at the end, all indices which are still zero can be determined to be prime generating indices. Saving indice numbers instead of toggling Boolean values in an array is not optimal and grows the required space complexity, especially when the sieving interval grows large. The problem can be remedied by recursively crossing off composites, similar to the method of Gries and Misra, but it's not demonstrated in this work due its slightly more complex pseudocode.

Algorithm 3 Linear MyriadPrime

```
function LinearMyriadPrimeSieve(n)  ▷ n is the limit up to which primes are generated
    I_max ← ⌊n/3⌋
    i ← 1

    while i ≤ I_max do                                  ▷ Initialize array with values
        lowestPrimeIndice[i] ← 0
        i ← i + 1
    end while

    x ← 1                                               ▷ Initialize
    y ← 1
    I ← I(x, y)

    while I ≤ I_max do                                  ▷ Main loop
        if lowestPrimeIndice[x] > 0 then
            y_lim ← lowestPrimeIndice[x]
        else
            y_lim ← x
        end if
        while y ≤ y_lim do
            if lowestPrimeIndice[y] ≠ 0 then
                y ← y + 1
                continue
            end if
            I ← I(x, y)
            if I > I_max then
                break
            end if
            lowestPrimeIndice[I] ← y
            y ← y + 1
        end while
        x ← x + 1                                       ▷ Prepare for next iteration
        y ← 1
        I ← I(x, y)
    end while

    i ← 1
    while i ≤ I_max do                                  ▷ Generate the primes based on the indices
        if lowestPrimeIndice[i] > 0 then
            G(i) ∈ ℙ                                    ▷ Print or save the prime G(i)
        end if
        i ← i + 1
    end while
end function
```

3.6 Remarks on the algorithms

Based on Theorem 3.3 it is sufficient to sieve only $\{6k \pm 1\}$ instead of the entire $\mathbb{N}$ in order to find $\mathbb{P} \setminus \{2, 3\}$. This property allows one to skip the multiples of 2 and 3 in sieving. Furthermore the Theorem 1.3 allows one to construct the function G which is used to generate the primes, using the prime generating indices that the algorithm discovers. The function G^{-1} is used to construct the composite index generating function $I(n, m)$. One should also note that for every composite $g \in \mathfrak{G}$, the factors of g are also of form $6k \pm 1$ and thus belong in $\mathfrak{G}$.

The initial algorithm calculates $I(n, m)$ for all n, m such that $I \leq I_{max}$. In essence the algorithm is the sieve of Eratosthenes restricted to $\mathfrak{G}$. The linear version saves the lowest prime factor of the composites $G(I(n, m))$ and therefore makes it possible to skip duplicate composites. From the theoretical perspective it doesn't matter which form of $G(k)$ or $I(n, m)$ one uses. In practical testing it was however noticed that out of the presented forms, the equational form (5) for $I(n, m)$ worked best. Similarily the $G(n) = 3n + \frac{3-(-1)^n}{2}$ form seemed to be the fastest when the odd-even cyclic fractional part is presented using a ternary operator and checking the last bit of n's binary presentation. In C-notation $G(n)$ can be expressed as `3n+((n&1)==1?2:1)`.

3.7 Other potential uses of the framework

The indexing function $I(x, y)$ makes it possible to craft a quadratic Diophantine equation in which the purely quadratic coefficients are zero, reducing it to the hyperbolic special case. By solving the crafted Diophantine equation $i = G^{-1}(g) = \lfloor \frac{g}{3} \rfloor = I(x, y)$ for x and y one can determine the primality of $g = G(i) \in \mathfrak{G}$. If the equation has an integer solution (x, y) (or multiple solutions) g is composite and if no solutions exist g is prime. Unfortunately the solution of the Diophantine equation requires the factorization of g and is thus not feasible for practical primality testing.

The idea can however be extended to study twin primes. Primes p and $p + 2$ are called twin primes if both $p \in \mathbb{P}$ and $p + 2 \in \mathbb{P}$. The $6k \pm 1$ form is ideal for twin primes because if p and $p + 2$ are twin primes then $p = 6k - 1$ and $p + 2 = 6k + 1$ for some k. The unsolved twin prime conjecture states that there are infinitely many such primes. Since

$$g = G(i) \in \mathbb{P} \iff \nexists (x, y) \in \mathbb{N}^2 : i = G^{-1}(g) = \lfloor \frac{g}{3} \rfloor = I(x, y)$$

the conjecture can be stated using the framework as

$$\exists_\infty k \in \mathbb{N} : 6k - 1 \in \mathbb{P} \wedge 6k + 1 \in \mathbb{P}$$

$$\iff$$

$$\exists_\infty k \in \mathbb{N} : (\forall (a, b, c, d) \in \mathbb{N}^4 :$$

$$(2k - 1 \neq 3ab + 2a + 2b + 1) \wedge (2k - 1 \neq 3ab + 2a + b) \wedge (2k - 1 \neq 3ab + a + b)$$

$$\wedge (2k \neq 3cd + 2c + 2d + 1) \wedge (2k \neq 3cd + 2c + d) \wedge (2k \neq 3cd + c + d)).$$

That is, there are infinitely many twin primes iff there are infinitely many integers k such that for each k all natural number quadruplets (a, b, c, d) satisfy the six nonequalities. The framework can also be extended for sums of primes as long as only prime generating indices are used.

4 Analysis of the linear algorithm

In this chapter we focus on the analysis of the presented linear prime sieving algorithm. Since the linear sieve is of greater interest we present its source codes in C++11 and JavaScript in appendices A and B respectively. For testing purposes the algorithm was also implemented in Python.

We first visually analyze the operation of the algorithm, i.e. which numbers are crossed off and which can be skipped to produce linear runtime. We then proceed to analyze the actual execution times of these algorithms in modern laptop and desktop enviroments comparing it to the sieve of Atkin, the original sieve of Eratosthenes and the segmented wheel sieve. Lastly we present potential further optimizations which could speed up the algorithm and discuss the pros and cons of the presented framework.

4.1 Analysis of the search space of the linear algorithm

The name Myriad was chosen for the algorithm to reflect the infinitude of different methods in which primes can be sieved and particularily the infinitude of ways in which the presented framework can be used to achieve this. Consequently there are too many different ways to enumerate all the possible sequences of operations and thus we present only one as the linear method which was seen as the simplest. This simplicity is also reflected in the presented source codes where the relevant elements of the space are first looped through vertically and proceeded horizontally until every element in the space has been operated (or skipped).

Let us start by analyzing the operations which the linear sieve carries out. Initially we only know the limit of the sieving interval n. This allows us to calculate the maximum index of the generator function using $G^{-1}(n)$ and allows us to get an idea on the space we are about to operate on. Consider all elements $G(k) = G(n)G(m) \leq G_{max}$ (or equivalently their indices $I(n,m) \leq I_{max}$). By [2] we know that the amount of elements in this space is of order $O(n \log \log n)$.

The initial space, in which no composite has been crossed off or skipped, is visualized in Figure 5. Note that the rightside tail has been cut for clarity after index 50. The following Figures 5-16 present the space of composites < 2500 needed to crossoff. The elements in light blue will be calculated and crossed off unless they are deemed pink (later figures). The dark blue elements are greater than our maximum index and are skipped altogether. Red elements indicate that the corresponding index has been processed.

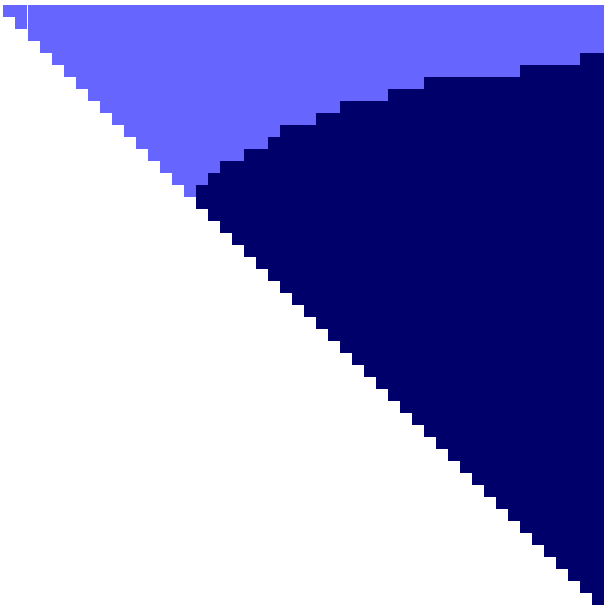

Figure 5: Initial search space.

The algorithm starts from $n = m = 1$, calculates the corresponding index $I(1,1) = 8$ and saves in to an array at position 8 the smaller of the indices (in this step 1), namely m since $m \leq n$ in this half of the space. After 8 has been deemed as (the lowest) composite generating index we can safely in the future skip all indices $I(8,k)$, where k is greater than the saved index 1. By symmetry and $m \leq n$, we can also skip all $I(n,8)$. The first skippable elements can be seen in Figure 6 colored in pink.

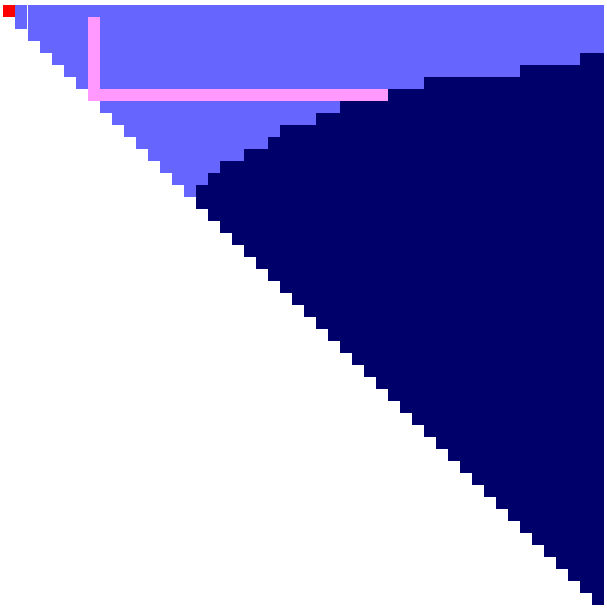

Figure 6: Search space after the first composite generating index has been determined.

Since the column at $n = 1$ contains only one element, we are done with it. Increment n and proceed to the column $n = 2$. Since $I(2,1) = 11$ we save its lower

generating index 1 to position 11. After this the space is as presented in Figure 7. Moving vertically in the column we calculate $I(2,2) = 16$ and save its lower generating index 2 to the corresponding index 16. After this the space is as presented in Figure 8.

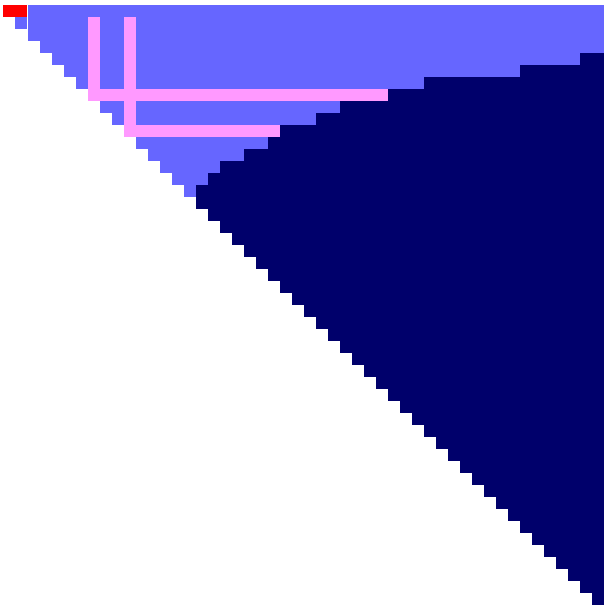

Figure 7: Search space after the second composite generating index has been determined.

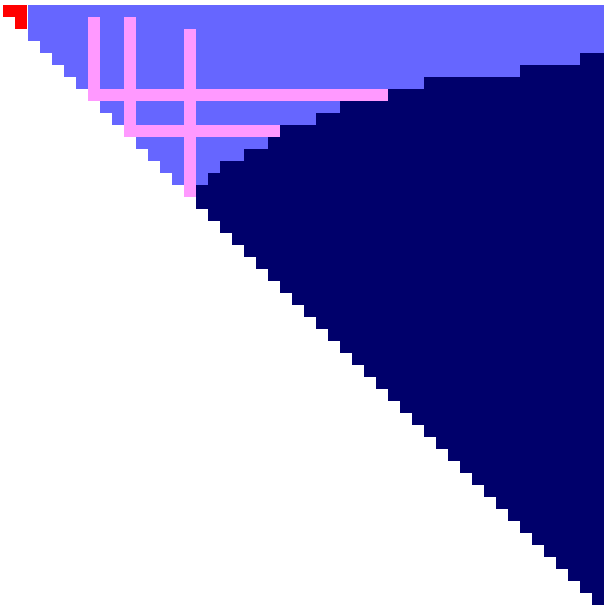

Figure 8: Search space after the third composite generating index has been determined.

After column $n = 2$ we move on to column $n = 3$. By calculating $I(3,1) = 18$ and saving the index, the space is as presented in Figure 9. We then calculate $I(3,2) = 25$ and save the index, producing a space in Figure 10. We then proceed to calculate $I(3,3) = 40$ and save the index, producing a space in Figure 11.

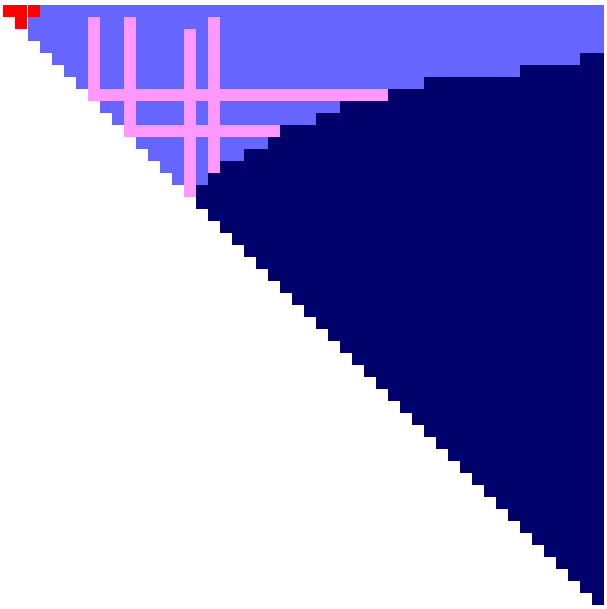

Figure 9: Search space after the fourth composite generating index has been determined.

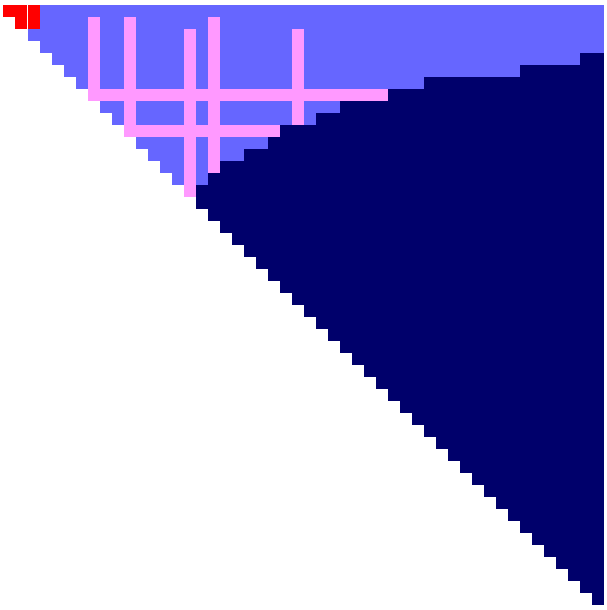

Figure 10: Search space after the fifth composite generating index has been determined.

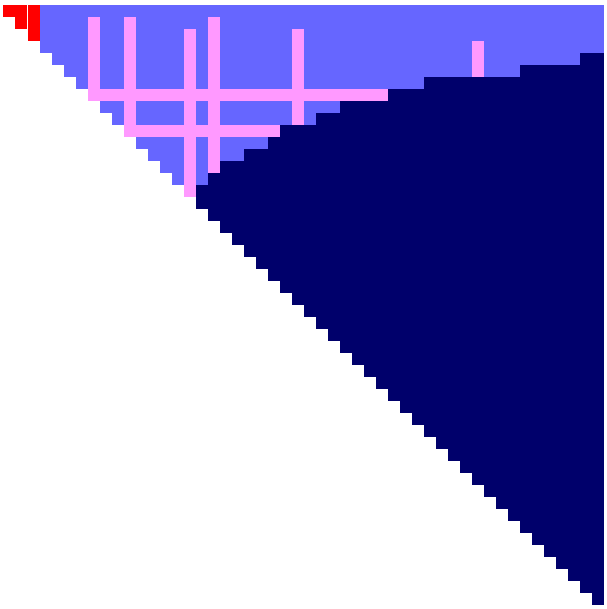

Figure 11: Search space after the sixth composite generating index has been determined.

Similarily we calculate the $n = 4$ column's indices saving the lower generating index. After calculating $I(4,1) = 21$, $I(4,2) = 30$ and $I(4,3) = 47$ the corresponding spaces are as presented in Figures 12, 13, 14 respectively. Note that the element $I(4,4) = 56$ is outside of the frame. In Figure 15 the space is shown after $I(5,1) = 28$ was calculated.

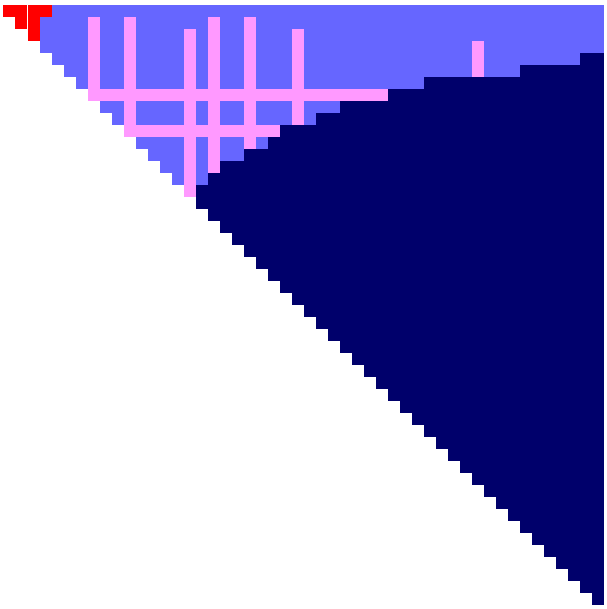

Figure 12: Search space after the 7th composite generating index has been determined.

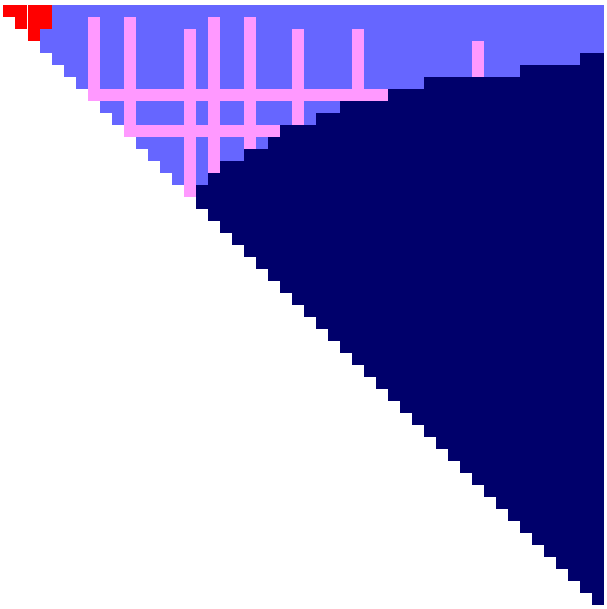

Figure 13: Search space after the 8th composite generating index has been determined.

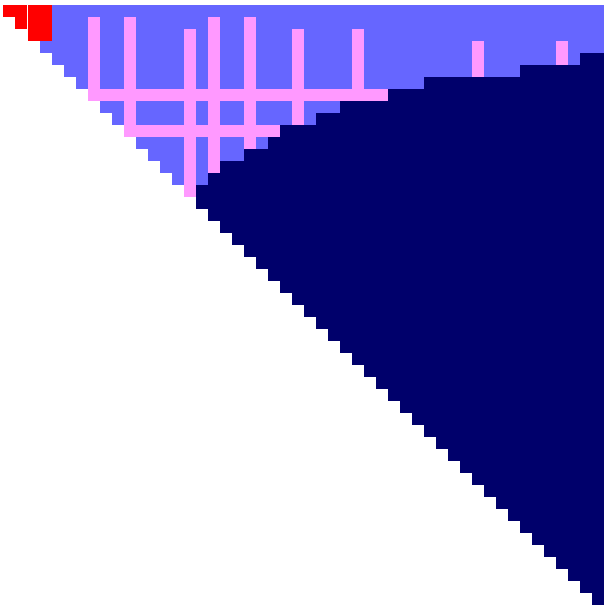

Figure 14: Search space after the 9th composite generating index has been determined.

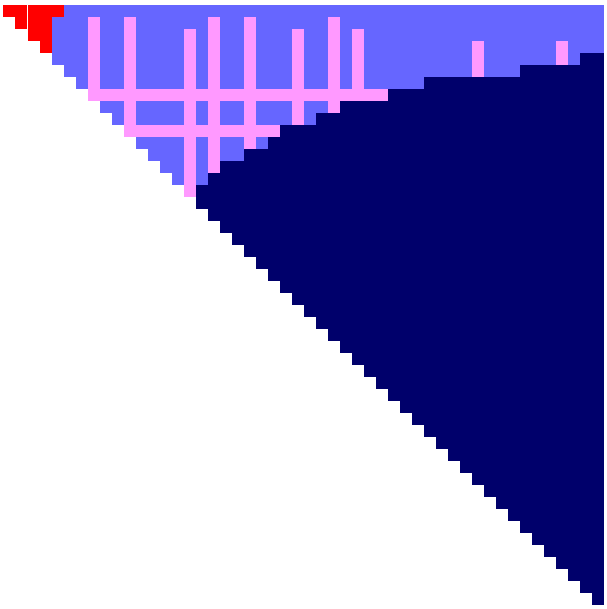

Figure 15: Search space after the 11th composite generating index has been determined.

In Figure 16 the studied space is presented after the largest index less than 50, $I(1,9) = 48$ has been crossed off. Notice that we need to only calculate the entire (skipping composite generating m) column n iff $G(n)$ is prime, stopping at the diagonal or until we hit the upper bound. For composite $G(n)$ we have to calculate elements only up to the saved index or until we hit the limiting index. When we have incremented n up to $\lfloor \frac{G_{max}}{15} \rfloor$ we can stop as it will be the last index generating multiples of $G(1) = 5$, since 5 is the lowest value of G.

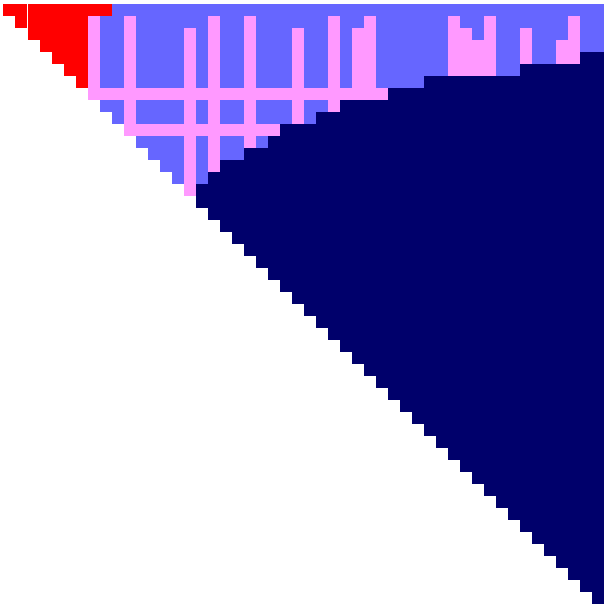

Figure 16: Search space after the 30th composite generating index has been determined and all skippable elements within visible region have been mapped.

To get a better idea of the distribution of the skippable elements, we present Figure 17 to show the composite elements that we need to calculate as well as the

skippable elements when a sieving limit of 50 000 is used. Note that again the rightside tail has been cut for better fit. Before we analyze the actual execution times of the different algorithms, let us discuss briefly about the actual implementations of these algorithms in the following section.

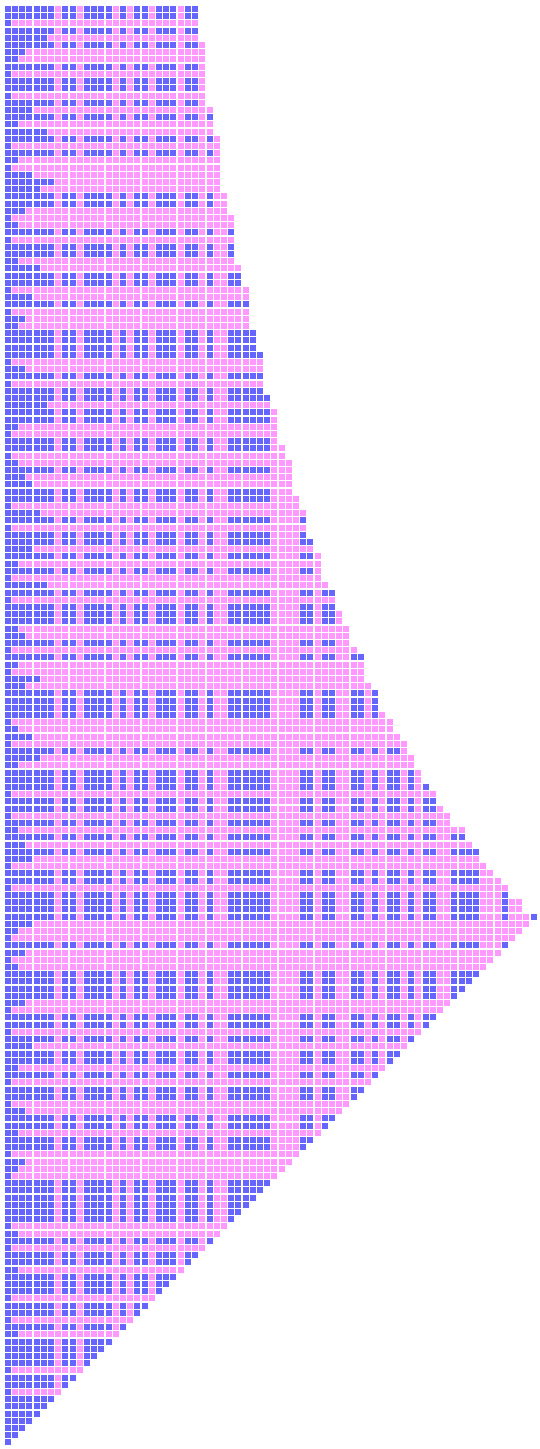

Figure 17: Search space when the sieving limit is 50 000. The figure has been rotated 90 degrees and the tail of the space is cut for visual convenience.

4.2 Implementations

The algorithm was initially crafted in JavaScript (ECMA6) as a proof of concept for the idea that one could sieve primes by using the function $G(n)$, generating the numbers of form $6k \pm 1$. Several optimization were noticed in the course of the programming and after the validity of the results of the original sieve (running in $O(n \log \log n)$) were confirmed the algorithm was modified to skip indices which were already crossed off or were about to be crossed off later in the execution of the algorithm. Thus the linear algorithm was conjured, initially written also in JavaScript. The JavaScript implementation was seen as a fast and debug friendly way to see if the framework could be leveraged to the purpose of sieving primes.

JavaScript allowed to work crossplatform in Linux/Windows/OSX, after all the initial need for primes rose up in a web-based project. As the idea span out to form the basis of this thesis, the algorithm was implemented in Python and C++11. The implementations were tested in modern desktop and laptop enviroments and the execution times were recorded. The desktop had a 3.7GHz i7-8700K CPU with 16Gb of DDR4 RAM running a 64-bit Ubuntu 16.04 (Linux). The laptop had a 1.9GHz i7-3517U CPU with 10Gb of DDR3 RAM running 64-bit Windows 10. The JavaScript and C++11 implementations were run on both machines but the Python implementation was only run on the laptop. Next we will take a look at the implementations, observe their execution times and compare the results to competing algorithms.

4.2.1 JavaScript

The JavaScript implementations were run on Chrome build 65.0.3325.181 64-bit with V8 build 6.5.254.41. The execution time of the algorithms were done by calculating the time difference using performance.now() -function, noting that the browser needed cache reboots between consecutive runs for reliable result in the smaller intervals. It is noteworthy to mention that web browser Chrome's highly optimized V8 engine optimizes ones JavaScript code on execution and handles multiple things in the background. This had its pros and cons. The primes were sieved relatively fast which was a good thing but on the other hand it jammed the Chrome once the sieving range grew to 10^9. However since the jamming happend similarily on the sieve of Eratosthenes and sieve of Atkin, it was deemed merely as a bug. Apparently if too much memory or CPU time is used, Chrome will output the "Oh snap.." error message.

The JavaScript linear Myriad sieve was compared to the sieve of Eratosthenes and the sieve of Atkin. For the JavaScript sieve of Atkin, an implementation by Mohammad Shahrizal Prabowo (https://gist.github.com/rizalp/5508670) was used. All different JavaScript sieves tested, sieved the desired interval and returned an array of the primes within it.

4.2.2 Python

The Python version of the linear Myriad sieve was crafted more as a practice but it turned out that most prime sieves had been implemented in Python 2.7 which eased the execution time comparisons. For some unknown reason Python was only able to use a maximum of 10% of CPU in Windows 10, which dramatically affected the speed of all sieves and made higher sieving intervals cumbersome to work with. The Python implementations were run in the laptop enviroment and the execution times were recorded using Python's -mtimeit flag.

The Python linear Myriad sieve was compared to the sieve of Eratosthenes, sieve of Atkin and a 2/3/5 wheel sieve. The Python implementation as well as the compared sieves, sieved the desired interval and returned an array of primes within it. The Python implementation of the sieve of Atkin is by Steven Krenzel, the implementation of the sieve of Eratosthenes is by M. Dickins and the 2/3/5 wheel sieve implementation is copyrighted to zerovolt.com, which unfortunately isn't a valid URL at the time of writing this thesis.

4.2.3 C++

Before the algorithm was written in C++, it was implemented and tested in C. In the C implementation it was noted that around the sieving limit of 10^7 the array used to store the lowest prime indices started to have memory leaks for an undetermined reason releted to memset() function. Thus it was implemented in C++11 using a vector to store the values. This was slightly slower compared to using arrays and uses too much space compared to what could be achieved trough better optimization. The C++ implementation was checked to contain no memory leaks (or other errors) using Valgrind up to 10^9.

The C++ linear Myriad sieve was compared to the sieve of Eratosthenes, sieve of Atkin and a highly optimized segmented wheel sieve (based on the sieve of Eratosthenes). The implementation of the sieve of Atkin is by Anuj Rathore and the segmented wheel sieve is maintained by Kim Walisch (www.github.com/kimwalisch). The C++11 implementations were run on the Ubuntu desktop and the execution times were recorded using the chrono library for nanosecond accuracy. The segmented wheel sieve had its own timer reporting in milliseconds. This was deemed acceptable as the implementation was multiple orders of magnitude faster than any other tested implementation.

The C++ implementations (excluding the wheel sieve) output the found primes in to stdout, which slows down the execution and isn't feasible for large sieving intervals past 10^8. Due to improper memory management the Myriad algorithm consumes over 16Gb of RAM when the sieving interval's upperbound was 10^{10} and thus 10^9 is the highest limit up to which primes were sieved in testing. The implementation of the sieve of Eratosthenes as well as the sieve of Atkin started to crash at the same limit. The highly optimized segmented wheel sieve had no problems going past 10^{12} but testing was concluded at this limit.

4.3 Results

The execution times of the JavaScript, Python and C++ implementations of the Myriad sieve and the competing sieves were recorded using consecutive powers of ten as an upper limit of sieving. The smallest upper limit used was 10^3. The testing was concluded at 10^9 after which the implementations started to consume excessive memory, with the exception of the highly optimized C++ wheel sieve. For each sieving limit, all the implementations were run 10 times and the lowest execution time was recorded. The Python's timing utility ran 100 times for intervals lower than 10^5. The recorded results for the execution times in milliseconds can be seen in Table 6. The log-log figures of the results for each platform are presented in the following subsections.

Table 6: Execution times in milliseconds with different sieving limits.

Method \ Limit	10^3	10^4	10^5	10^6	10^7	10^8	10^9	10^{10}	10^{11}
Myriad (JS)	0.2	0.8	1.2	6.5	56.7	503.8	-	-	-
Eratosthenes (JS)	0.1	0.4	0.9	9.6	183.6	4969.4	-	-	-
Atkin (JS)	0.2	3.6	9.9	73.6	842.3	-	-	-	-
Myriad (Py)	0.795	10.0	103	1000	7190	-	-	-	-
Eratosthenes (Py)	1.2	16.2	155	1930	-	-	-	-	-
Atkin (Py)	1.1	9.5	99.7	913	9720	-	-	-	-
Wheel (Py)	0.9	3.5	21.7	208	2250	-	-	-	-
Myriad (C++)	0.0238	0.0989	0.6331	7.6736	63.743	625.041	6142.61	-	-
Eratosthenes (C++)	0.1644	0.4441	4.7147	55.3728	522.543	5612.05	63456.9	-	-
Atkin (C++)	0.0514	0.33501	3.2857	34.4451	316.219	3206.48	42287.8	-	-
Segmented Wheel 1 thread (C++)	-	-	-	-	1	35	126	1584	20411
Segmented Wheel 12 threads (C++)	-	-	-	-	1	24	57	300	3406

4.3.1 JavaScript

The results for the execution times were quite interesting as can be seen from Figure 18. The first interesting observation is that the Myriad algorithm actually has a fighting chance and is not dominated by the other methods. Secondly the algorithm seems to run an order of magnitude faster than the competing implementation of the sieve of Atkin. This is probably due to the Atkin's algorithmic complexity and especially the fact that the used implementation utilizes the modulo operator % which can slow things down considerably when the sieving interval grows. The sieve of Atkin started to crash Chrome before the limit 10^8 and the other two crashed before 10^9.

The JavaScript implementation of the sieve of Eratosthenes seemed to be faster for smaller intervals but somewhere between the limits of 10^5 and 10^6 the Myriad sieve gains an advantage probably due to the duplicate composites that the sieve of Eratosthenes is crossing off. It would be interesting to implement other linear sieves or the wheel sieve in JavaScript and repeat the test. Note the behaviour around the sieving limits of 10^4 and 10^5 where the V8 apparently changes some optimization tactic. A similar effect was observed in the execution times on the laptop runs which were slightly slower but fairly similar to the execution times on the desktop enviroment.

4.3.2 Python

The recorded results can be seen in Figure 19. In Python the Myriad sieve and the sieve of Atkin were quite evenly matched and were both faster than the sieve of Eratosthenes. The wheel sieve implementation was an order of magnitude faster than the other implementations after the first limit of 10^3. Due to the excessive time it took to run the Python implementations, the testing was concluded at 10^7.

4.3.3 C++

The plotted results for the C++ implementations can be seen in Figure 20. Due to improper memory management the sieve of Eratosthenes, sieve of Atkin and the Myriad sieve could only sieve up to 10^9, after which excessive memory was used. The highly optimized segmented wheel sieve operated without problems on all tested intervals.

It is fairly surprising that the Myriad sieve was an order of magnitude faster than the sieve of Eratosthenes and the sieve of Atkin. The segmented wheel sieve was so fast that the first recorded 1ms execution time was with sieving limit 10^7. The segmented wheel sieve was run with a constraint of 1 thread and relaxing this condition, which led to the use of 12 threads. The speed increment can be seen in the higher intervals.

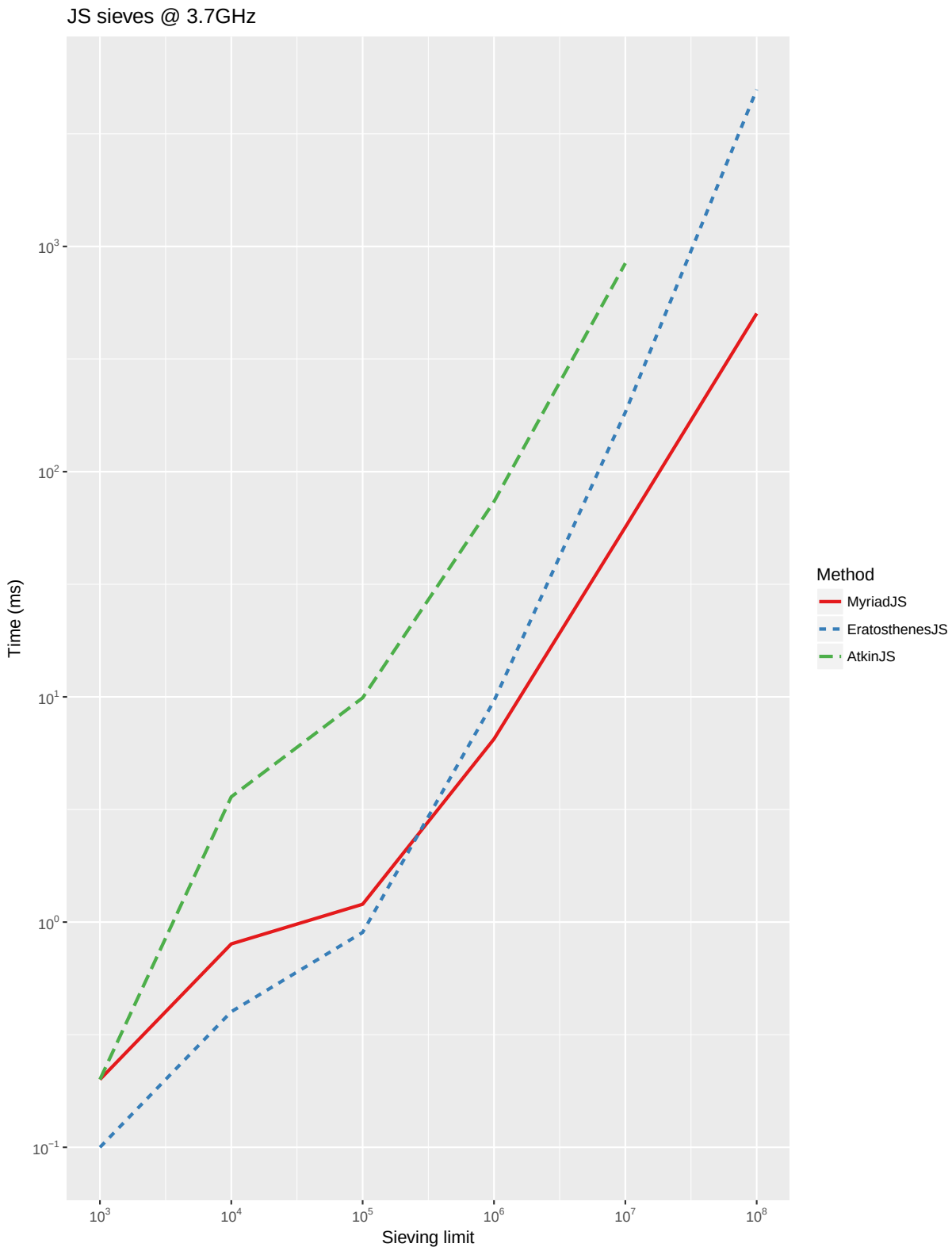

Figure 18: Execution times of various sieves in JavaScript.

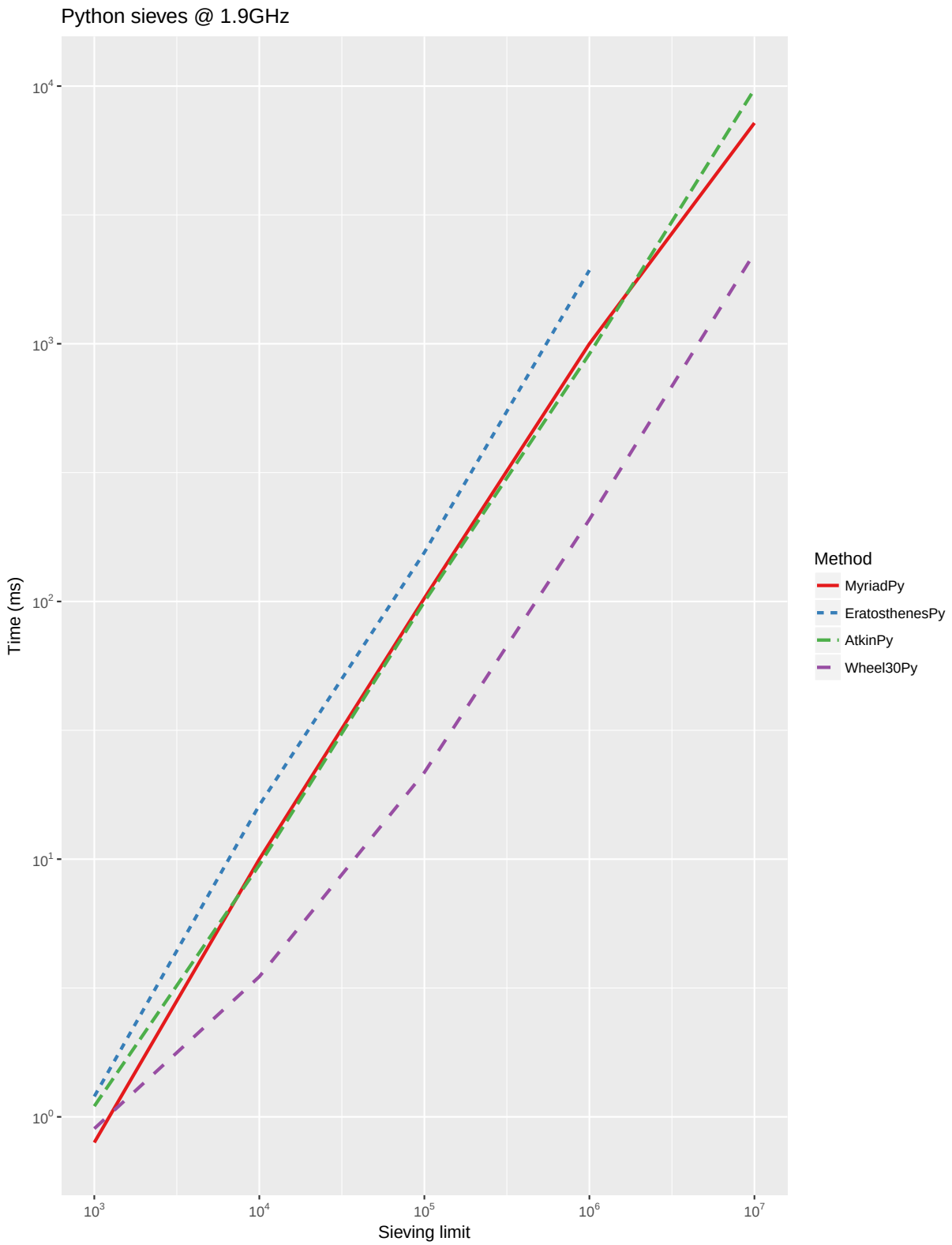

Figure 19: Execution times of various sieves in Python.

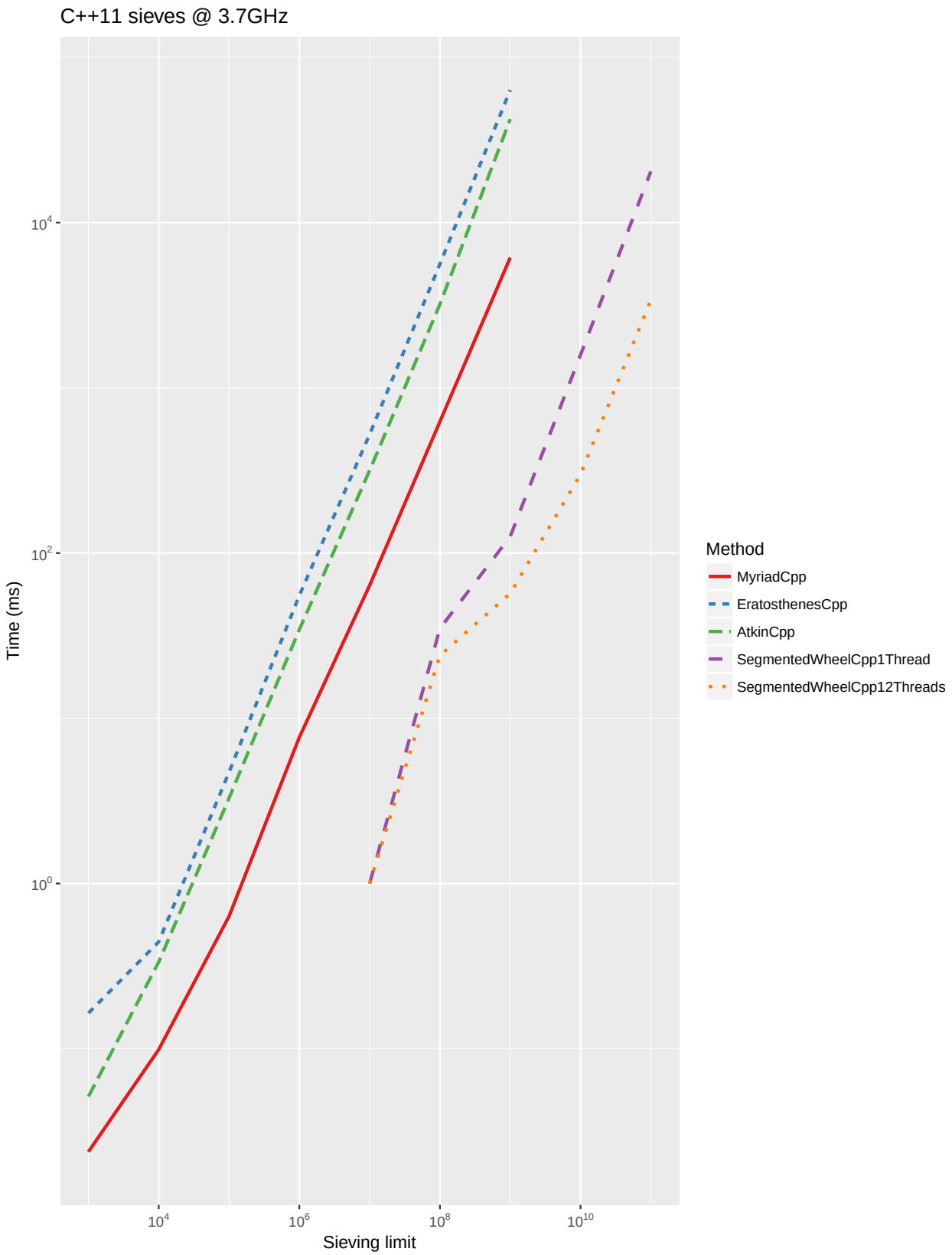

Figure 20: Execution times of various sieves in C++11.

4.4 Potential further optimizations

The biggest single problem in the current version of the C++11 implementation of the Myriad sieve is poor management of memory which seems to become a bottleneck at higher sieving intervals. There is also potential for parallel processing and perhaps the entire algorithm could be made parallel if the indices could be crossed off in monotonic order or the crossing off of duplicate composites turns out to not be too expensive speedwise. It would also be worth investigating how to segment the intervals or adapt wheels to make the algorithm faster.

Reasonable optimizations could be made by fine tuning the path of the algorithm in which indices are generated and crossed off as the search space can be crawled in many different fashions. Recursion could be used to loop through the lowest prime indices which should optimize the space complexity to be linear since then we can only crossoff indices in a Boolean array as opposed to saving the integer values of the lowest prime indices. This might help to increase the sieving interval as well.

Using more clever datastructures, such as heaps or buckets, one might scan the necessary indices in a monotonically increasing fashion and just save the primes (or the indices which generate them). A method in which the logarithmic part of the search space would be scanned horizontally would probably be worth investigating as it would reduce significantly the number of calculations of $I(n, m)$ which turn out to be greater than I_{max}.

The idea of having a secondary boolean array where composites would be also crossed off instead of just saving the lowest prime indices was tried out and seemed to make execution order(s) of magnitude slower. Multiple different variations for the calculations of the functions $G(n)$ and $I(n, m)$ were tried. The forms `G(n)=3*n+((n&1)==1?2:1)` and `I(n,m)=3*n*m+n*((m&1)==1?2:1)+m*((n&1)==1?2:1)+((((n&1)==1)&&((m&1)==1))?1:0)` seemed to be the fastest.

The summation methods for the calculation of indices were not tried as it was thought to be slower, as the cumulative sums would need to be adjusted on many skippable indices, but the results might be different. Another idea might be operate 2x2 or nxn blocks instead of singular elements as the block shares similar multiplications but makes the skipping somewhat more complex.

5 Conclusions

The entire project started from a geometric pattern observed in the squares of primes. It was fairly surprising how far the framework could be extended and that it revealed a prime sieve which turned in to a linear time complexity sieve. Furthermore it was surprising that the execution times in various programming languages were decent and competitive to moderately implemented sieves made by others.

Based on the empirical evidence as well as the literature it would seem that the idea of segmentation is crucial for the higher intervals as memory access will become a bottleneck faster than the computational speed. The use of a wheel seems to be useful in practice as well. The fact that there exists faster sieves was expected but it would be nice to see how fast the Myriad sieve could potentially be made. It would also be interesting to find faster competing JavaScript sieves.

The source codes of the Myriad sieve in JavaScript and C++ can be found in the appendices A and B but one can also find them online in the authors Github page https://github.com/JuhaniSipila/MyriadPrimes .

References

[1] S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani. *Algorithms.* McGraw-Hill, 2006.

[2] G. H. Hardy, and E. M. Wright. *An Introduction to the Theory of Numbers 4th Edition.* Oxford University Press 1960, Corrected in 1975.

[3] D. Underwood. *Elementary Number Theory.* W. H. Freeman and Co., 1978.

[4] L. E. Dickson. *History of the Theory of Numbers Vol.1 Divisibility and Primality.* Carnegie Institute of Washington, 1919.

[5] W. Schwarz. *Some highlights from the history of probabilistic number theory.* Advanced Studies in Pure Mathematics 49, 2007.

[6] M. E. O'Neill. *The Genuine Sieve of Eratosthenes.* Journal of Functional Programming 19, 2009.

[7] J. Sorenson. *An Introduction to Prime Number Sieves.* Computer Sciences Technical Report #909, 1990.

[8] H. G. Mairson. *Some new upper bounds on the generation of prime numbers.* Communications of ACM 20, 1977.

[9] D. Gries, and J. Misra. *A Linear Sieve Algorithm for Finding Prime Numbers.* Communications of ACM 21, 1978.

[10] P. Pritchard. *Linear Prime Number Sieves: a Family Tree.* Science of Computer Programming 9, 1987.

[11] P. Pritchard. *Explaining the Wheel Sieve.* Acta Informatica 17, 1982.

[12] C. Bays, and R. H. Hudson. *The Segmented Sieve of Eratosthenes and Primes in Arithmetic Progressions to* 10^{12}. BIT 17, 1977.

[13] J. Lorch, and G. Ökten. *Primes and Probability: The Hawkins Random Sieve.* Mathematics Magazine 80, 2006.

[14] A.O.L Atkin, and D.J. Bernstein. *Prime Sieves Using Binary Quadratic Forms.* Mathematics of Computation Vol. 73 Number 246, 2003.

[15] R. D. Carmichael. *Note on a new number theory function.* Bulletin 16 of the American Mathematical Society, 1910.

[16] R. Solovay, and V. Strassen. *A Fast Monte-Carlo Test for Primality.* SIAM Journal on Computing Vol.6 No.1, 1977.

[17] G. Miller. *Riemann's Hypothesis and Tests for Primality.* Journal of Computer and System Sciences 13, 1976.

[18] M. Rabin. *Probabilistic algorithm for testing primality.* Journal of Number Theory 12, 1980.

[19] C. Pomerance, J.L. Selfridge, and S.S. Wagstaff Jr. *The pseudoprimes to* $25 \cdot 10^9$. Mathematics of Computation Vol. 35, 1980.

[20] M. Agrawal, N. Kayal, and N. Saxena. *Primes in P.* Annals of Mathematics 160, 2004.

[21] D. X. Charles. *Sieve Methods* State University of New York at Buffalo, 2000.

A C++ implementation of the linear Myriad sieve

```
/*
Linear Myriad prime sieve v1.1 19.5.2018
Author: Juhani Sipilä
Aalto University - Systems Analysis Laboratory
*/
#include <iostream>
#include <vector>
#include <chrono>
using namespace std;

long long G(long long n){
  //All primes>5 can be expressed as 6a+-1 - Bungus 1599, Wells 1986
  return 3*n+((n&1)==1?2:1); //optimized 6*floor((n+1)/2)+(-1)^n
}

long long I(long long x,long long y){
  //optimized I=floor(G(x)*G(y)/3)
  return 3*x*y+x*((y&1)==1?2:1)+y*((x&1)==1?2:1)+((((x&1)==1)&&((y&1)
    ==1))?1:0);
}

void MyriadSieveLinear(long long limit){
  //init sieving range
  long long Imax = limit/3; //NOTE: floor(limit/3)

  vector<long long> lpi(1+Imax); //Lowest Prime Indices
  fill(lpi.begin(), lpi.end(), 0); //Set all elements to 0. >0
    indicates composite!

  //init variables for looping
  long long x = 1; //start from x=y=1 i.e. composite 25
  long long y = 1;
  long long Icur = 8; //init current index. floor(25/3)=8
  long long yLim = 1; //init yLim(x)! lpi[x], x or I_x <=Imax

  while(Icur<=Imax){ //horizontal loop
    //5's multiples are always crossable as it is the lowest prime in G
    ! floor(5/3)=1
    lpi[Icur] = 1;  //crossed off xth 5's multiple (y=1)
    y = 2;  //increment y ==> y=2
    yLim = (lpi[x]!=0?lpi[x]:x);
    while(y<=yLim){ //vertical loop
      if(lpi[y]!=0){y++; continue;} //skip composite rows
      Icur = I(x,y); //calculate next current index
      if(Icur>Imax){break;} //is the index out of search space?
      lpi[Icur] = y; //crossed off Icur and saved its lowest prime
    index
      y++;
    } //columns relevant elements crossed off
    x++; //move horizontally
```

```
    Icur = I(x,1); //prepare for next iteration
  } //all composites <=G_max=G(Imax) crossed off

  long long i = 1; //init output iterator
  long long g = 5; //init output to smallest prime in G
  long long pi = 2;
  cout << 2 << endl;   //manually output 2 & 3 since they aren't in space G
  cout << 3 << endl;   //assuming limit>3
  while(i<=Imax){
    if(lpi[i]==0){ //lpi=0 ==> G(i) is prime
      g = G(i);
      if(g>limit){break;} //stop output
      cout << g << endl;
      pi++;
    }
    i++;
  }
  cout << "PI(N): " << pi << endl;
}

int main(){
  auto t0 = chrono::steady_clock::now();
  long long lim = 1000000000;
  MyriadSieveLinear(lim);
  auto t1 = chrono::steady_clock::now();
  auto dt = t1-t0;
  cout << "Runtime " << chrono::duration <double, milli> (dt).count() << " ms" << endl;
  return 0;
}
```

B JavaScript implementation of the linear Myriad sieve

```javascript
/*
Linear Myriad prime sieve v1.1 19.5.2018
Author: Juhani Sipilä
Aalto University - Systems Analysis Laboratory
*/
function G(n){
  //all P>5 can be expressed as 6a+-1 - Bungus 1599, Wells 1986
  return 3*n+(n&1==1?2:1);
}

function I(n,m){
  return 3*n*m+n*(m&1==1?2:1)+m*(n&1==1?2:1)+((n&1==1)&&(m&1==1)?1:0);
}

function MyriadSieveLinear(limit){
  //maximum index of the interval
  var Imax = Math.floor(limit/3);
  var lpi = Array(1+Imax).fill(0); //lowest prime indices
  //interval = Array(1+limit);
  var x = 1;
  var y = 1;
  var Icur = 8;
  var yLim;

  while(Icur<=Imax){ //horizontal loop
    //5's multiples are always crossable as it is the lowest prime in G! floor(5/3)=1
    lpi[Icur] = 1;  //crossed off xth 5's multiple (y=1)
    y = 2;  //increment y ==> y=2
    yLim = (lpi[x]!=0?lpi[x]:x);
    while(y<=yLim){ //vertical loop
      if(lpi[y]!=0){y++; continue;} //skip composite rows
      Icur = I(x,y); //calculate next current index
      if(Icur>Imax){break;} //is the index out of search space?
      lpi[Icur] = y; //crossed off Icur and saved its lowest prime index
      y++;
    } //columns relevant elements crossed off
    x++; //move horizontally
    Icur = I(x,1); //prepare for next iteration
  } //all composites <=G_max=G(Imax) crossed off

  primes=[]; //push primes to an array
  primes.push(2);
  primes.push(3);
  var i=1;
  while(i<Imax){
    if(lpi[i]==0){
      primes.push(G(i));
    }
```

```
    i++;
  }
  g=G(i); //account for 'non-G' limits
  if(g<=limit && lpi[i]==0){primes.push(g)}

  return primes;
}
```