Comparison of Gaussian Prime Generation Algorithms

Wilhelm Carmevik, William Kraft

April 22, 2022

Abstract

We have analysed the time and space complexity and measured the runtime and memory usage of three different algorithms used for generating Gaussian primes. Considering complex numbers with integer components, i.e. Gaussian integers, one can extend the concept of primes. Furthermore, algorithms for generating primes like the sieve of Eratosthenes can be extended to the Gaussian integers, algorithm 1. The other algorithms tested include factoring primes in $\mathbb{Z}$ using Smith's algorithm, algorithm 2, and by checking if the norm is prime in $\mathbb{Z}$, algorithm 3. We have proved that the time complexity for the algorithms are $O(n\log\log n)$, $O(n^{1+\frac{1}{4\sqrt{e}}+\varepsilon}/\log^2 n)$ and $O(n\log\log n)$ respectively where $\varepsilon > 0$ and n is the norm up to which Gaussian primes are generated. The space complexity is $O(n)$ for all algorithms. The experimental data agreed with the derived complexities suggesting that they are correct. Our conclusion is that algorithm 3 is the most efficient, since it can use precalculated lists of primes in $\mathbb{Z}$ although algorithm 2 may conditionally have the same time complexity with primes in $\mathbb{Z}$ precalculated as our data also suggests.

Contents

1 Introduction **3**

1.1 Background . . . 3

1.2 Goal . . . 6

1.3 Research problem . . . 6

2 Method **6**

3 Results **9**

3.1 Theoretical analysis . . . 9

3.1.1 Analysis of algorithm 1 . . . 9

3.1.2 Analysis of algorithm 2 . . . 10

3.1.3 Analysis of algorithm 3 . . . 12

3.2 Experimental results . . . 13

3.3 Theoretical - Experimental Concurrence . . . 13

4 Discussion **16**

4.1 Evaluation of results . . . 16

4.2 Interpretation of results . . . 17

4.3 Evaluation of study . . . 18

4.3.1 Theoretical . . . 18

4.3.2 Experimental . . . 18

4.4 Future studies . . . 19

References **20**

A Introduction to Stieltjes integrals **21**

B Derivation of theoretical results **21**

C Code **24**

D Raw measurements **34**

1 Introduction

1.1 Background

Computational number theory has largely been concerned in researching primality tests for the integers, in part, because of the demand for primes in cryptography. Consequently there is a good understanding of how these algorithms perform from a computational perspective. However, there is a more limited understanding of how algorithms for generating primes in the Gaussian integers perform. In this paper we analysed the runtime and memory usage of three different algorithms for generating Gaussian primes.

The following definitions and theorems, taken from chapter 3 in Herstein (1975) unless stated otherwise, are used in this paper.

Definition 1.1. *The ring $\mathbb{Z}$ of integers is the ring formed from the set of integers $\mathbb{Z} = \{0, \pm 1, \pm 2, \pm 3, \pm 4, \ldots\}$ under the operations addition and multiplication.*

Furthermore, this ring is also commutative since the order of multiplication does not matter.

Definition 1.2. *The imaginary number i is defined by the property that its square, i^2, is equal to -1.*

Definition 1.3. *The ring $\mathbb{Z}[i]$ of Gaussian integers is the ring formed from the set $\{a + bi \mid a, b \in \mathbb{Z}\}$ under addition and multiplication of complex numbers.*

Addition intuitively works by adding the real and imaginary parts respectively. Multiplication works like multiplication with two binomials but definition 1.2 is used and occurrences of i^2 are replaced with -1.

Definition 1.4. *Let R be a commutative ring and let a be an element of R. An element $b \in R$ divides a if there exists an element $x \in R$ such that $a = bx$.*

In the ring $\mathbb{Z}$, 1 and -1 divides every element. The generalization of this property is captured in the following definition.

Definition 1.5. *Let R be a ring. A unit of the ring R is an element $u \in R$ that has a multiplicative inverse in R. That is, there exists an element $v \in R$ such that*

$$uv = vu = 1,$$

where 1 is the multiplicative identity.

The rings $\mathbb{Z}$ and $\mathbb{Z}[i]$ are not only commutative rings but also *integral domains*; there exists no two non-zero elements with a zero element product. Integral domains allows for the introduction of the concept of *irreducible elements.*

Definition 1.6. *A non-zero, non-unit element of an integral domain is said to be irreducible if every factoring of the element contains at least one unit.*

The concept of irreducible elements appears similar to the idea of prime numbers in $\mathbb{Z}$ and allows for a generalisation of primes to integral domains. While irreducible elements and prime elements are not generally equivalent, they are equivalent for all *unique factorization domains*, wherein the rings $\mathbb{Z}$ and $\mathbb{Z}[i]$ fall.

An example where a prime in $\mathbb{Z}$ is not a prime in $\mathbb{Z}[i]$ is the number 5, which can be factored into $(2+i)(2-i)$. The only units in $\mathbb{Z}[i]$ are $1, -1, i, -i$, neither of which occur in this factoring of 5. Therefore, 5 does not satisfy the definition of an irreducible element in $\mathbb{Z}[i]$ and thus 5 is not prime in the ring of Gaussian integers.

When generating primes in $\mathbb{Z}[i]$, the algorithms for finding primes in $\mathbb{Z}$ are relevant, as the latter are often used as subroutines of the former. The perhaps most famous algorithm for finding integer primes is the *sieve of Eratosthenes* which is described in many texts, for example in Bressoud and Wagon (2000). The algorithm consists of listing all integers from 2 to n in a table. Beginning with the first integer on the list, 2, which must be prime, one marks all multiples of it. The next non-marked integer must be prime, namely 3. The algorithm repeats, marking the multiples of the next non-marked integer. This is valid because the first non-marked integer on the list will always be prime since no smaller integer divides it. To find the primes under n one only has to continue until arriving at an integer greater than $\sqrt{n}$ because if n, which is the largest number on the list, is not prime at least one of the factors must be less than $\sqrt{n}$. The primes are simply the non-marked integers.

Stein (1976) proposes an algorithm extending the sieve of the Eratosthenes to the Gaussian integers. The algorithm is presented as a good exercise for understanding primes in $\mathbb{Z}[i]$. It is almost identical to the original sieve, but instead it traverses the Gaussian integers based on their norms.

Definition 1.7. *Let z be a Gaussian integer, then it can be written as $a + bi$ where a and b are integers. The norm $N(z)$ of z is defined as the whole number $a^2 + b^2$.*

Theorem 1.8. *An important property of the norm is that it is multiplicative. Let α and β be Gaussian integers then $N(\alpha\beta) = N(\alpha)N(\beta)$.*

Given theorem 1.8, and that only units in the Gaussian integers have norm 1, corollary 1.9 is trivial.

Corollary 1.9. *If the norm of a Gaussian integer is a prime in $\mathbb{Z}$ then the Gaussian integer is a prime in $\mathbb{Z}[i]$.*

However, the converse of this corollary is not true. As an example, 7 is a prime in $\mathbb{Z}[i]$ but has the norm 7^2 which is a composite number.

A theorem, first formulated by Fermat, states that any odd prime p can be written as the sum of two squares if, and only if, p is congruent to 1 modulo 4. Given this, the following theorem can be proven for Gaussian primes.

Theorem 1.10. *(Irving, 2004) All primes in $\mathbb{Z}[i]$ fall into the following three classes:*
(i) The primes in $\mathbb{Z}$ congruent to 3 *modulo* 4.
(ii) The unique Gaussian integers $a + bi$ and $a - bi$ where $(a + bi)(a - bi) = a^2 + b^2 = p$ and p

is two or a prime in $\mathbb{Z}$ congruent to 1 *modulo* 4.
(iii) The unit multiples of the preceding classes of Gaussian primes.

Smith's algorithm is an algorithm for constructing two squares that sum to a given prime p which is congruent to 1 modulo 4. The algorithm consists of finding an integer x such that $x^2 \equiv -1 \ (mod\, p)$ and applying the Euclidean algorithm to p and x, where the first two remainders less than $\sqrt{p}$, have squares that sum to p (Wagon, 1990). The first work on this algorithm was made by C. Hermite and J.A. Serret in 1848 and later improved upon by H.J.S. Smith in 1855 (Brillhart, 1972).

From theorem 1.10 it is possible to prove an algorithm for generating primes in $\mathbb{Z}[i]$ which proceeds by first finding the primes in $\mathbb{Z}$ and then writing the primes congruent to 1 modulo 4 as the sum of two squares. The primes in $\mathbb{Z}$ can be found by using the sieve of Eratosthenes, while Smith's algorithm can be used to write primes as the sum of two squares (Bressoud & Wagon, 2000).

Due to the many parameters that can affect the runtime and memory used when an algorithm executes, *time complexity* and *space complexity* is instead used as a metric when comparing algorithms. The time complexity of an algorithm is defined in a way such that it only depends on the algorithm itself and the input, namely the time complexity is the number of computational operations an algorithm needs to execute (Wegener, 2005). Likewise the space complexity is the number of bits of memory an algorithm needs. These metrics are given as functions of the input, whose format could for example be the input's binary size or the norm under which to generate primes as in the case of this study.

It is standard to measure the time complexity by considering the worst case scenario regarding the number of computational steps. The *unit cost model* is often used, meaning that every arithmetic operation takes a constant amount of time (Wegener, 2005). This is a simplification since addition and multiplication on larger numbers requires more bit operations than on small ones. In fact the time complexity for multiplication grows in proportion to the logarithm of the numbers multiplied, hence the name *logarithmic cost model* when this is taken into account. However, the unit cost model is used in this analysis.

It is seldom possible to find a function of the exact number of computational operations an algorithm requires, instead only the growth rate, or limiting behavior, of this function is found. The growth rate is denoted by the use of big O notation (Wegener, 2005).

Definition 1.11 (Big O-notation)**.** *(Wegener, 2005) For a function $f : \mathbb{N} \to \mathbb{R}^+$ and a function $g : \mathbb{N} \to \mathbb{R}^+$, $f = O(g)$ means that asymptotically f grows no faster than g which is defined by the condition that $\frac{f(n)}{g(n)}$ is bounded from above by a constant c as n tends to infinity.*

As an example, the sieve of Eratosthenes, introduced earlier, takes $O(n \log\log n)$ computational steps and $O(n)$ memory where n is the bound under which primes are found (Pritchard, 1981). The total time complexity of two subroutines that are run in series is the sum of the individual complexities of the subroutines, according to the following computational rule:

Theorem 1.12. *(Wegener, 2005) For any function* $f : \mathbb{N} \to \mathbb{R}^+$ *and any function* $g : \mathbb{N} \to \mathbb{R}^+$

$$O(f) + O(g) = O(f + g) = O(max\{f, g\})$$

1.2 Goal

The goal of this study is to develop the understanding of the computational efficiency of a selection of Gaussian prime generation algorithms. There is, to the authors' knowledge, no previous published research regarding the performance of algorithms for generating Gaussian primes. Prior literature regarding these algorithms has largely excluded discussions and analysis of their time and space complexities and their empirical performance. Since there is a collection of applications where great amounts of Gaussian primes are required, for example in investigating the Moat problem (Gethner et al., 1998), knowledge about the efficiency of the algorithms generating them is needed. A lack of information can lead to misinformed decisions which can impede progress and waste computational power. Therefore research on the efficiency and performance of these algorithms can contribute to the field of computational number theory.

1.3 Research problem

The specific type of algorithms studied were algorithms that take a norm as input and returns all Gaussian primes with norm less than it. The specific questions we wished to answer were:

- What time and space complexities could the algorithms theoretically achieve?
- How does the runtime and memory usage empirically increase for larger inputs and how well does this correspond to the time and space complexities?
- Using the answers to the previous two questions, how do these algorithms compare in time and space complexity?

2 Method

The algorithms were studied through both theoretical analysis and experimental data. This was accomplished through the use of the established rules of algorithm analysis and by implementing the algorithms in the Python programming language. To evaluate how well the theoretical analysis correlated with the experimental data, we used the definition of big O notation. The following are high-level descriptions of the algorithms. The code used to implement them in Python can be found in appendix C.

Algorithm 1

Algorithm 1 is the previously mentioned extension of the sieve of Eratosthenes. It proceeds by creating a list of all Gaussian integers that are in the first octant with norm less than n. It traverses the Gaussian integers in order of increasing norm. For each non-marked Gaussian integer encountered, the multiples of it, and its conjugate, are all marked on the list. When all numbers with norm less than or equal to $\sqrt{n}$ have been traversed, the non-marked Gaussian integers, and their conjugates, and unit multiples, are all the Gaussian primes with norm less than n by theorem 1.8.

The algorithm is presented in Stein (1976), however the description lacks details on its implementation. For example, traversing the Gaussian integers in order of their norm is difficult computationally, but has no description in the paper. To address this, our implementation traverses the list in order of Manhattan distance, the sum of their real and imaginary parts, which is computationally simpler. This is valid because when all elements with norm less than m have been traversed, the non-marked elements with norm less than m^2 are Gaussian primes by theorem 1.8. When the elements are traversed in order of their Manhattan distance, they are never chosen from outside the circle for which the non-marked elements are definitely prime. The exception is when the norm is less than 3, which can instead be handled manually in advance.

Algorithm 2

This algorithm is presented in Bressoud and Wagon (2000), with further explanation on Smith's algorithm in Wagon (1990). The algorithm proceeds by traversing all primes in $\mathbb{Z}$ less than n, applying different criteria to find the primes in $\mathbb{Z}[i]$. This is done through first using the sieve of Eratosthenes to generate the primes, and then traversing the list. If a number is congruent to 3 modulo 4, it is a Gaussian prime, and if it has a norm less than n it can be added to a list of Gaussian primes. Otherwise, if a prime is the number 2, or congruent to 1 modulo 4, the prime can be factored in $\mathbb{Z}[i]$ according to theorem 1.10. Using Smith's algorithm, one acquires the two unique integers α and β where $\alpha^2+\beta^2$ is equal to the prime. $\alpha+\beta i$ and $\alpha-\beta i$ are then Gaussian primes and can be added to the list of Gaussian primes. When all the primes in $\mathbb{Z}$ have been traversed, the unit multiples of all elements in the list need to be added to the list to obtain all Gaussian primes with norm less than n.

Smith's algorithm applied on a prime p finds the solution to the quadratic congruence $x^2 \equiv -1 \ (mod\ p)$ and applies Euclid's algorithm to p and x. The first two remainders that are less than $\sqrt{p}$ will have squares that sum to p. To solve the congruence equation $x^2 \equiv -1 \ (mod\ p)$ the algorithm uses Euler's criterion, stating that for an odd prime p and an integer a relatively prime to p,

$$a^{\frac{p-1}{2}} \equiv -1 \ (mod\ p) \tag{1}$$

if, and only, if there is no integer b such that $a \equiv b^2 \ (mod\ p)$, that is, a is a quadratic non-residue modulo p. Since p is congruent to 1 modulo 4 the number $\frac{p-1}{4}$ is an integer and (1) can be rewritten as

$$(a^{\frac{p-1}{4}})^2 \equiv -1 \ (mod\ p).$$

It follows that $x = a^{\frac{p-1}{4}}$ possibly is a solution to the congruence equation. The variable a traverse the primes until a solution to the equation is found.

Algorithm 3

Algorithm 3 consists of traversing the Gaussian integers with norm less than n and checking for each Gaussian integer if it is prime. A Gaussian integer is , according to theorem 1.10 and corollary 1.9, prime if its norm is prime in $\mathbb{Z}$ and the imaginary part is non-zero. When the imaginary part is zero, the Gaussian integer is prime if it is a prime in $\mathbb{Z}$ congruent to 3 modulo 4. To examine whether these conditions are satisfied, the integer, or norm, is compared with the contents of a hash set data structure containing the primes in $\mathbb{Z}$ generated by the sieve of Eratosthenes.

Algorithm 2 and 3 with primes precalculated

Since algorithm 2 and 3 share a common subroutine, the sieve of Eratosthenes, versions of algorithm 2 and 3 where the list of primes in $\mathbb{Z}$ was precalculated, have been analysed and measured as well. This was to enhance their differences and examine the likely case of lists of primes in $\mathbb{Z}$ being available.

Theoretical analysis

By analysing the number of elementary operations, as well as the greatest amount of computer memory simultaneously used by the respective algorithms, we have derived time and space complexities of the algorithms in terms of big O notation. When analysing the algorithms and when manipulating big O expressions we used methods presented in Greene and Knuth (1990) and Wegener (2005).

Experimental study

To experimentally examine the different algorithms, they have been implemented in the Python programming language. We chose Python as it is less dependent on the implementation. It lacks a compiler, cache efficiency and many optimisations which would have been found in most languages. While these would have decreased runtimes, they are also highly unpredictable and could have had unforeseen consequences.

We measured the algorithms' runtime and memory usage for inputs of every hundred thousandth integer less than or equal to five million. Their runtime was tracked with the help of the Python method time.process_time(), which was used to mitigate the effects of the operating system prioritising different threads. The memory usage was tracked using the tracemalloc library, where peak memory usage is the measured variable. As the tracemalloc library effects runtime performance the two tests were run separately.

To further reduce random errors we ran several measurements for each input, five for time measurements and three for memory measurements. The difference in number of iterations were chosen as memory usage was more deterministic than the more random runtime. The statistical method applied to this data was averaging the values for the same input. The standard deviation was also calculated for each input.

Compilation of data

To examine the validity of the theoretical analysis, the derived complexities were compared to the experimental data. To examine if they satisfied the definition of big O the quotients between the theoretical and experimental values were calculated and graphed. If these graphs were bounded, they would, by definition 1.11, satisfy the big O relation.

3 Results

3.1 Theoretical analysis

A summary of the results of the derivations can be found in table 1. The following are the derivations and deductions.

3.1.1 Analysis of algorithm 1

Algorithm 1 can be analysed in four separate parts: (1) listing all Gaussian integers with norm less than n, (2) finding the next Gaussian prime to act on, (3) marking the multiples of the prime and (4) listing the non-marked integers. The list including all integer with norm less than n has size proportional to n. Consequently both the time and space complexity of part (1) is $O(n)$. Finding the next Gaussian prime to act on (2), is done in constant time, and is repeated once for every Gaussian integer with norm less than $\sqrt{n}$. Since the number of Gaussian integers traversed is asymptotically proportional to $\sqrt{n}$, this is done in $O(\sqrt{n})$ computational steps and constant memory. Labelling all multiples, with norm less than n, of a Gaussian prime π requires a number of computational steps proportional to the amount of such multiples. There are less than $c \cdot n/N(\pi)$ such multiples where c is a constant. The time complexity of this part is the sum

$$\sum c \frac{n}{N(\pi)} \tag{2}$$

as π ranges over all Gaussian primes with $N(\pi) \leq \sqrt{n}$. According to theorem 1.10, the norm of a Gaussian prime is either a prime in $\mathbb{Z}$ or the square of such a prime. Furthermore for every prime p in $\mathbb{Z}$ there is only one Gaussian prime (up to complex conjugates and unit multiples) that has p as its norm (Irving, 2004). Thus the sum (2) is less than

$$cn \sum \frac{1}{p} \tag{3}$$

where p ranges over all primes in $\mathbb{Z}$ less or equal to $\sqrt{n}$. The reason that this sum is larger is that some of the reciprocals are squares of primes in (2). According to lemma B.1 (see appendix B), the sum $\sum_{p\leq\sqrt{n}} 1/p$ is $O(\log\log n)$, implying that the number of computational steps for part (3) is $O(n\log\log n)$. The memory used during this process is constant.

Part (4), listing all non-marked integers, will take $O(n)$ computational steps and the memory the new list will take up is $O(n)$ since the number of Gaussian primes will be less than the size of the first list.

The total time and space complexity for algorithm 1 is the sum of the time and space complexities respectively for the different parts. Using theorem 1.12,

$$O(n) + O(\sqrt{n}) + O(n\log\log n) + O(n) = O(n\log\log n) \tag{4}$$
$$O(n) + O(1) + O(1) + O(n) = O(n) \tag{5}$$

where (4) is the time complexity and (5) is the space complexity for algorithm 1.

3.1.2 Analysis of algorithm 2

Algorithm 2 is composed of two subroutines: generating the relevant primes in $\mathbb{Z}$, and generating its corresponding Gaussian prime. According to Pritchard (1981), the former subroutine has $O(n\log\log n)$ as time complexity and $O(n)$ as space complexity.

The number of computational steps of the latter subroutine is the sum of the number of computational steps of each instance of Smith's algorithm. Furthermore, Smith's algorithm consists of two subroutines: finding the solution x to the equation $x^2 = -1 \ (mod\ p)$ and Euclid's algorithm on p and x.

Since the congruence equation is solved by finding a quadratic non-residue by testing increasingly larger prime values, the size of the least quadratic non-residue is required to compute the computational complexity.

Burgess (1957) has proven that the least quadratic non-residue is $O_\varepsilon(p^{\frac{1}{4\sqrt{e}}+\varepsilon})$ for any $\varepsilon > 0$, which is the best unconditional bound that has been proven (Bober & Goldmakher, 2015).

To derive the time complexity for algorithm 2 one can proceed as follows:

- Substitute the bound for the least quadratic non-residue into the prime counting function $\pi(n)$. This provides the maximum number of trials required to find the solution to $x^2 \equiv -1 (mod\ p)$.

- Since Euclid's algorithm is executed on p and x, the solution to the former congruence equation, the number of operations needed to run it must be accounted for. This corresponds to $O(\log n)$ where n is the smaller of the two integers it is applied on (Uspensky & Heaslet, 1939). To ease the computations, one can use $O(\log p)$ as an upper bound for this time complexity.

- The total number of operations to find two squares which add up to p, is then the sum of the two former results.

- Since this process is executed for all primes congruent to 1 modulo 4 up to n, the time complexity is the sum of the former result as p traverses through such primes. This sum can be calculated with lemma B.2, for which there is a proof in appendix B.

- Since the primes up to n were obtained with the sieve of Eratosthenes, its time complexity must be added to the former result, providing the total time complexity for the algorithm.

Substituting Burgess' bound, $c_0(\varepsilon)p^{\frac{1}{4\sqrt{e}}+\varepsilon}$, into the prime counting function $\pi(x) = O(x/\log x)$ (Soprunov, 2010), one obtains that

$$O_\varepsilon\left(\frac{p^{\frac{1}{4\sqrt{e}}+\varepsilon}}{\log p}\right) \tag{6}$$

computational steps are required to solve the congruence equation. Adding $O(\log p)$ to this, using theorem 1.12, the computational complexity for Smith's algorithm for a given p becomes

$$O(\log p) + O_\varepsilon\left(\frac{p^{\frac{1}{4\sqrt{e}}+\varepsilon}}{\log p}\right) = O_\varepsilon\left(max\left\{\log p, \frac{p^{\frac{1}{4\sqrt{e}}+\varepsilon}}{\log p}\right\}\right) = O_\varepsilon\left(\frac{p^{\frac{1}{4\sqrt{e}}+\varepsilon}}{\log p}\right) \tag{7}$$

since any strictly positive power of p grows faster than any power of $\log p$ (see lemma B.3). The total complexity for all instances of Smith's algorithm is the sum,

$$\sum_{\substack{p \ prime \\ p\equiv 1 \ mod \ 4}}^{n} O_\varepsilon\left(\frac{p^{\frac{1}{4\sqrt{e}}+\varepsilon}}{\log p}\right). \tag{8}$$

Since the summed function is positive and monotone increasing for sufficiently large values of p, lemma B.2 implies that this sum is equal to

$$O_\varepsilon\left(\frac{n^{1+\frac{1}{4\sqrt{e}}+\varepsilon}}{\log^2 n}\right), \tag{9}$$

which is also the time complexity for algorithm 2 given that the primes are precalculated. To get the complexity of the whole algorithm, $O(n\log\log n)$ is added, giving

$$O(n\log\log n) + O_\varepsilon\left(\frac{n^{1+\frac{1}{4\sqrt{e}}+\varepsilon}}{\log^2 n}\right) = O\left(max\left\{n\log\log n, \frac{n^{1+\frac{1}{4\sqrt{e}}+\varepsilon}}{\log^2 n}\right\}\right) = O_\varepsilon\left(\frac{n^{1+\frac{1}{4\sqrt{e}}+\varepsilon}}{\log^2 n}\right) \tag{10}$$

since any strictly positive power of n grows faster than any power of $\log n$ (see lemma B.3). This is the total time complexity for algorithm 2.

Using different models for the least non-residue

The bound used for the least non-residue is likely an overestimate (Wagon, 1990). If the Extended Riemann hypothesis is true, as expected, Ankeny (1952) proved that the bound can be reduced to $O(\log^2 p)$. Using an analogous procedure one can derive that, using this bound, the time complexity for this algorithm is

$$O\left(\frac{n\log n}{\log\log n}\right). \tag{11}$$

If absolute upper bounds are used for how great the least non-residue is, the acquired estimate is most likely an overestimate, since it is summing over many different modulus p. A heuristic one can use to mitigate this problem is to use probability theory. Since half of all integers from 1 to $p-1$ are quadratic residues modulo p (Irving, 2004) the probability that a given integer is a quadratic non-residue is presumably $\frac{1}{2}$. Assuming that the probabilities are independent, which has not been proven, the number of trials needed to find a solution can be approximated to be constant. This heuristic would likely better represent the algorithm since the algorithm is executed for many different instances. It can be shown, using the same procedure as before, that from this heuristic the time complexity for all instances of Smith's algorithm together is $O(n)$ thus the time complexity for the whole of algorithm 2 is

$$O(n\log\log n). \tag{12}$$

The space complexity for Smith's algorithm does not increase for bigger inputs and when it is applied on a given p no numbers but the outputs are stored, giving that the space the whole algorithm needs is just $O(n)$.

3.1.3 Analysis of algorithm 3

Algorithm 3 is composed of two separated parts: creating a hash set data structure containing the primes in $\mathbb{Z}$ less than n and traversing the Gaussian integers with norm less than n while individually checking if they are Gaussian primes. The former, running the sieve of Eratosthenes, has a time complexity of $O(n\log\log n)$ and a space complexity of $O(n)$ (Pritchard, 1981).

The time complexity of checking if any Gaussian integer is prime is constant, as indexing into the hash set to check for primality, and calculating congruence modulo 4 all use constant time. Since this is done to all Gaussian integers with norm less than n, the time complexity is proportional to the number of such Gaussian integers i.e. $O(n)$.

If the primes are precalculated the time complexity for algorithm 3 is consequently $O(n)$, otherwise the time complexity for the whole algorithm is

$$O(n\log\log n)+O(n)=O(max\{n\log\log n,n\})=O(n\log\log n). \tag{13}$$

The space complexity for this algorithm is the sum of the memory used by the sieve of Eratosthenes and the memory used by the list of Gaussian primes. Since the list of Gaussian primes requires less memory than the number of Gaussian integers with norm less than n, its memory usage is $O(n)$. The total space complexity for algorithm 3 is therefore $O(n)$.

	Time complexity Unconditional	Time complexity ERH	Time complexity Probabilistic	Space Complexity
Alg. 1	$O(n \log\log n)$	-	-	$O(n)$
Alg. 2a	$O_\varepsilon\left(\frac{n^{1+\frac{1}{4\sqrt{e}}+\varepsilon}}{\log^2 n}\right)$	$O\left(\frac{n\log n}{\log\log n}\right)$	$O(n \log\log n)$	$O(n)$
Alg. 3a	$O(n \log\log n)$	-	-	$O(n)$
Alg. 2b	$O_\varepsilon\left(\frac{n^{1+\frac{1}{4\sqrt{e}}+\varepsilon}}{\log^2 n}\right)$	$O\left(\frac{n\log n}{\log\log n}\right)$	$O(n)$	$O(n)$
Alg. 3b	$O(n)$	-	-	$O(n)$

Table 1: A summary of the derived time and space complexities. Alg. 2b and Alg. 3b denotes the versions of algorithm 2 and 3 where the primes in $\mathbb{Z}$ are precalculated.

3.2 Experimental results

The raw measurements can be found in appendix D. The data collected for algorithm 1, 2 and 3 is graphed in figure 1. Note that this is the average of five or three tests depending on whether it is runtime or memory usage. Figure 2 gives the plot for the versions of algorithm 2 and 3 which already have a precalculated list of integer primes available.

For each respective input the standard deviation of the runtime and memory usage was calculated for all algorithms. The standard deviation was divided by the average runtime and memory respectively to measure the size of the standard deviation in relation to the measurements. Furthermore, the mean, median and max values of this quotient are presented in table 2.

	Mean:	Median:	Max:
Time:	$1.013789 \cdot 10^{-2}$	$5.623984 \cdot 10^{-3}$	$1.884223 \cdot 10^{-1}$
Memory:	$2.441367 \cdot 10^{-5}$	$1.676158 \cdot 10^{-6}$	$1.126621 \cdot 10^{-3}$

Table 2: For every input the standard deviation of the five runtime and three memory usage measurements were calculated. This was divided by the average value of the runtime and memory usage for that input. Furthermore, in this table the mean, median and maximum of those values for all algorithms are presented.

3.3 Theoretical - Experimental Concurrence

In figure 3 the averaged runtimes of algorithm 1 and 3 have been divided by the theoretically derived time complexity function evaluated at each respective input so that the big O relation can be examined. Furthermore in figure 4 the average runtime for algorithm 2 has been divided by the three different time complexity functions that we derived by considering different bounds for the least quadratic non-residue. In figure 5 the averaged memory for algorithm 1, 2 and 3 has been divided by the derived space complexity functions for those algorithms. Finally in figure 6 and 7, the same types of quotients have been plotted for the versions of algorithm 2 and 3 where the primes were precalculated. Note that these graphs do not necessarily have a y-axis that starts at 0.

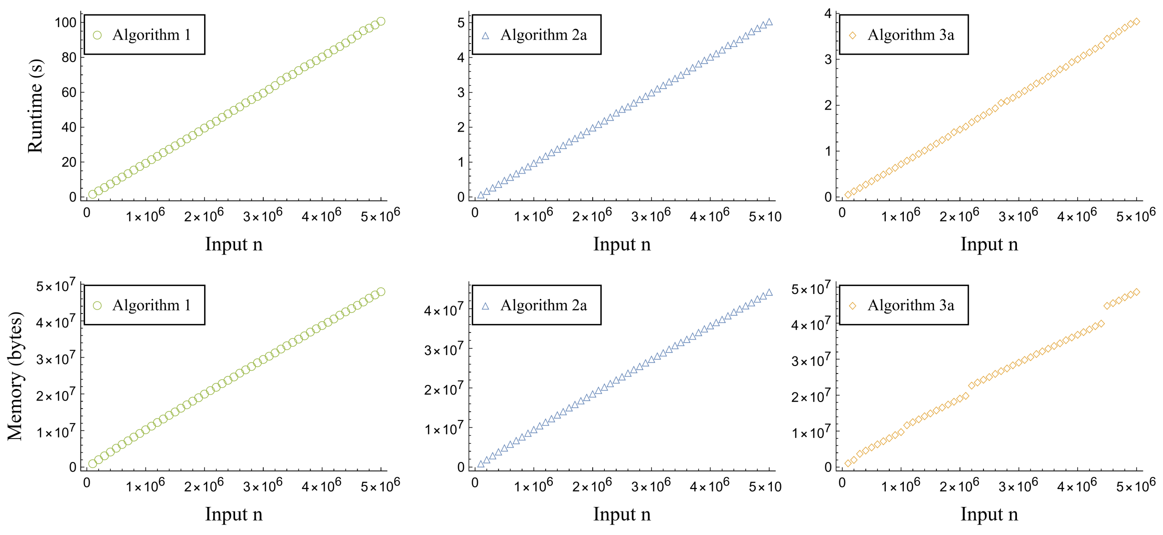

Figure 1: Plot of the runtime and memory measurements for all algorithms.

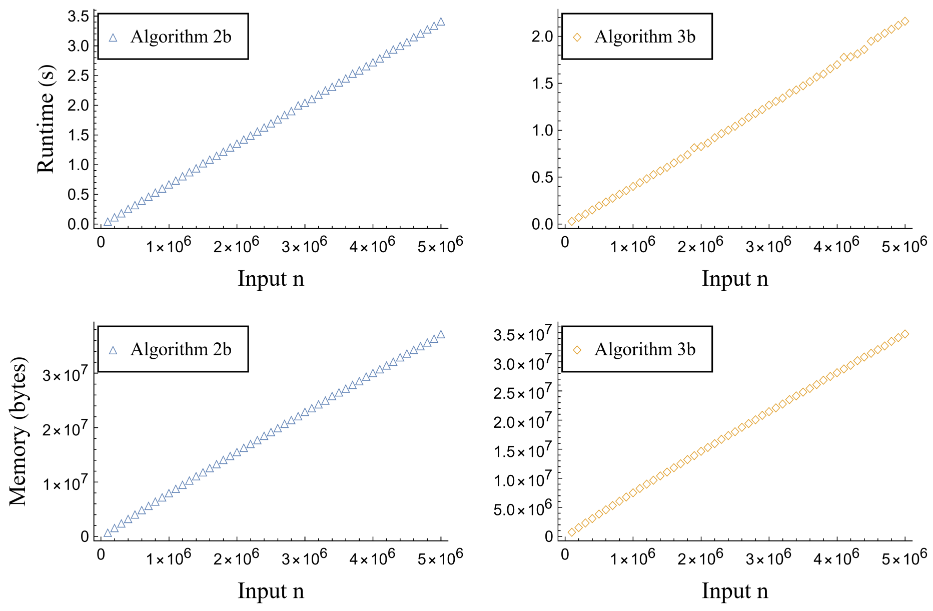

Figure 2: Plot of the runtime and memory measurements for algorithm 2 and 3 with primes precalculated.

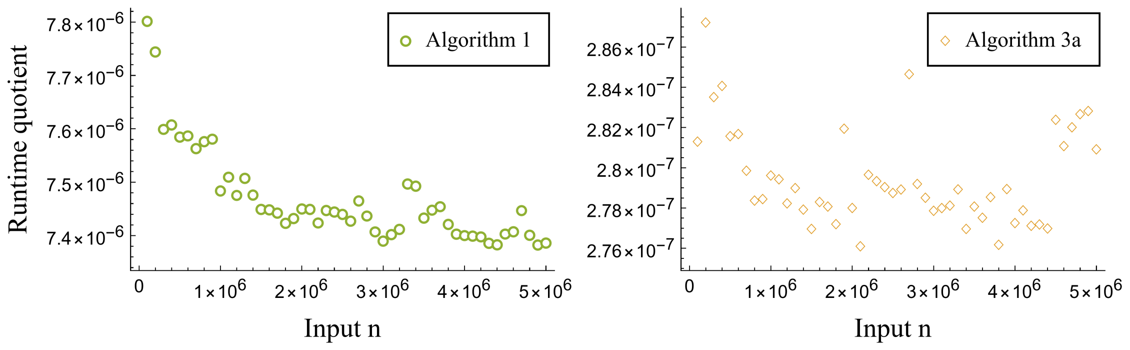

Figure 3: The quotient of the measured runtime of algorithm 1 and 3 with $n \log\log n$

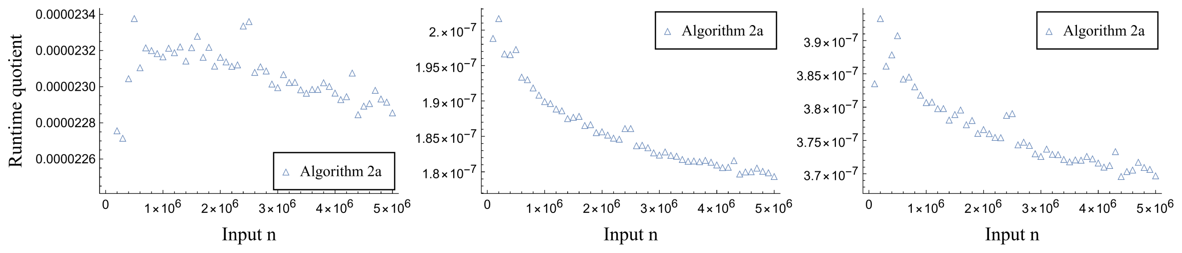

Figure 4: The quotient of the measured runtime of algorithm 2 with $n^{1+\frac{1}{4\sqrt{e}}+\varepsilon}/\log^2 n$ ($\varepsilon = 0.001$), $n\log n/\log\log n$ and $n\log\log n$ respectively.

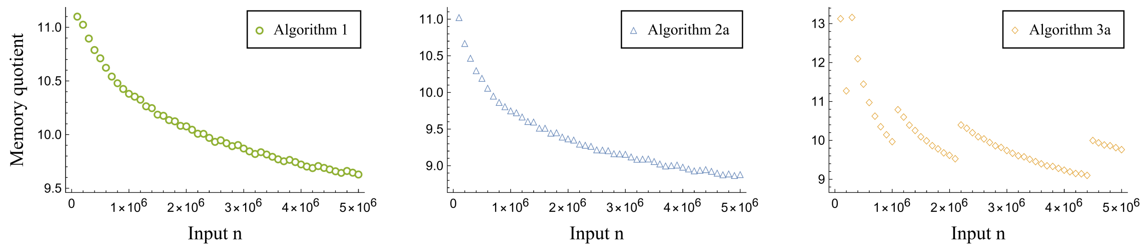

Figure 5: The quotient of the measured memory of algorithm 1, 2 and 3 with n.

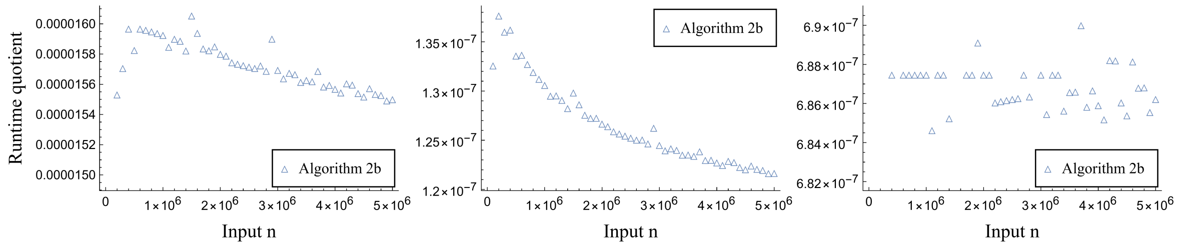

Figure 6: The quotient of the measured runtime of algorithm 2 (primes precalculated) with $n^{1+\frac{1}{4\sqrt{e}}+\varepsilon}/\log^2 n$ ($\varepsilon = 0.001$), $n\log n/\log\log n$ and n respectively.

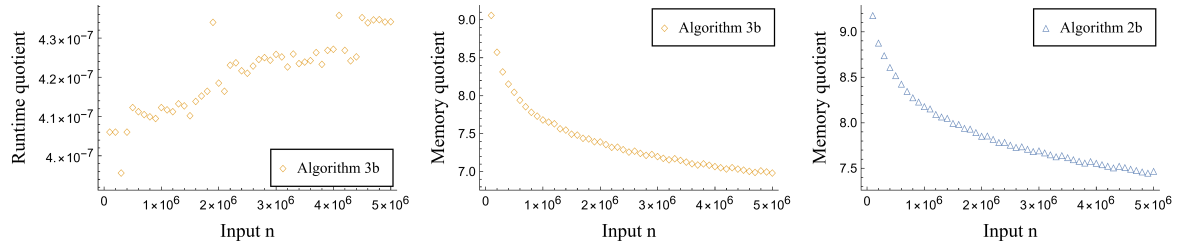

Figure 7: The quotient of the measured runtime of algorithm 3 (primes precalculated) n, the measured memory of algorithm 3 (primes precalculated) with n and the measured memory of algorithm 2 (primes precalculated) with n.

4 Discussion

4.1 Evaluation of results

We derived that the time complexity for algorithm 1 is $O(n\log\log n)$ and that the space complexity is $O(n)$, which appears to agree with the experimental data. Figure 3 (left) displays the quotient of the experimentally measured runtime at n and the theoretical runtime acquired by evaluating $n\log\log n$. The graph is clearly decreasing which suggests that the quotient stays bounded, thus satisfying definition 1.11 for the runtime to be $O(n\log\log n)$. Furthermore, the quotient of the measured memory usage at n and the theoretical memory usage $O(n)$ (figure 5, left) is decreasing suggesting it is bounded in its limiting behavior. This suggests that the theoretically derived time and space complexity is correct.

The same reasoning as above can be used to test the reliability of the different time complexities derived for algorithm 2. Three different answers were derived for the time complexities of algorithm 2 depending on which bound for the least quadratic non-residue is used. Unconditionally it can be shown with a bound due to Burgess (1957) that the time complexity for algorithm 2 is

$$O_\varepsilon\left(\frac{n^{1+\frac{1}{4\sqrt{e}}+\varepsilon}}{\log^2 n}\right). \tag{14}$$

However, assuming, as believed (Wagon, 1990), that the ERH is true, a bound for the least non-residue by Ankeny (1952), would imply that the time complexity is

$$O\left(\frac{n\log n}{\log\log n}\right). \tag{15}$$

Furthermore, a probabilistic argument would give that the time complexity is

$$O(n\log\log n). \tag{16}$$

However, this was based on the assumption that the probability of each number being a quadratic non-residue is $\frac{1}{2}$, and that it is independent of the probabilities for the neighboring numbers which might not be true. This result is thus merely a heuristic and not a fact.

Considering the quotient of the measured runtime for algorithm 2 and the three different theoretical time complexities (figure 4), all three graphs are decreasing which suggests that all quotients stay bounded. Since $n\log\log n$ is asymptotically the smallest of these three functions, the boundedness of the right graph of figure 4 would imply the boundedness of the other two graphs. One can also note that the quotient with $n^{1+1/4\sqrt{e}+\varepsilon}/\log^2 n$ seems to decrease the least despite the function asymptotically being the largest of the three. This is due to the function being smaller than the other two for small values (roughly up to $n = 10^{27}$). The theoretical data thus suggests that the time complexity for algorithm 2 is $O(n\log\log n)$ however this cannot be proven since one can only measure the runtime for finitely many values.

Likewise, the quotient of the memory usage for algorithm 2 (figure 5) suggests that the space complexity is $O(n)$ since it seems to be bounded, as it is decreasing. This is in accordance to what could unconditionally be proven and makes it more reliable than the time complexity.

The time and space complexity for algorithm 3, $O(n \log \log n)$ and $O(n)$, were the same as for algorithm 1. The runtime and memory quotient for algorithm 3 (figure 3, right and figure 5, right) is decreasing which suggest that the derived time and space complexities are correct. However, some jumps are present in the graphs of the quotients, which can, at least partly, be explained (see 4.3.2). However, since the time complexity was derived unconditionally it is still reliable.

Regarding the version of algorithm 2 and 3 in which a list of primes in $\mathbb{Z}$ was available, the space complexity is unchanged, but the time complexity differs. The time complexity of algorithm 3 is $O(n)$, however for algorithm 2 the time complexity only differs in the complexity derived using the probabilistic model for non-residues, which is $O(n)$, while the other complexities do not change as they are asymptotically larger than the time complexity for the sieve of Eratosthenes.

The graphs of the quotients for algorithm 2 and 3 with primes in $\mathbb{Z}$ precalculated, were all, except figure 6, right side, and figure 7, left side, decreasing. The time complexity quotient of algorithm 2 with primes precalculated and using the probabilistic model seems constant with some random error. It can be assumed that it is bounded. However, the quotient of algorithm 3 with primes precalculated is increasing which challenges our theoretical time complexity. It is possible that it is bounded if the growth decreases for larger values. Again, since we are only testing finitely many values, the conclusions that can be made are limited since experimental tests cannot prove or disprove any theoretical result.

4.2 Interpretation of results

Since the theoretical time and space complexities of the algorithms are known and supported by the data gathered, with some uncertainly regarding algorithm 3, it is possible to reason about how these algorithms compare to each other.

By our analysis all algorithms have the same space complexity and all but algorithm 2 have the same time complexity. Furthermore, if one assumes the probabilistic model for the least quadratic non-residue, they all have the same time complexity. This is also supported further by the measurements suggesting that the probabilistic model is valid for at least small values of n. Thus it seems like all algorithms have the same time and space complexity.

Instead examining the versions of algorithm 2 and 3 with the subroutine running the sieve of Eratosthenes omitted, as seen in table 1, algorithm 3 unconditionally has a time complexity of $O(n)$, while algorithm 2 only does assuming the probabilistic model to be true, and otherwise has a far worse time complexity.

If a list of already computed primes is accessible, algorithm 2 and 3 have the asymptotically lowest time complexities. However, we can not with certainty specify which of algorithm 2 and 3 has the lowest time complexity, and instead leave it to future studies to examine.

4.3 Evaluation of study

To properly evaluate the results of the study, we have used both theoretical and experimental methods, allowing them to verify each other. As mentioned previously, the theoretical and experimental agree to a high degree and the precision of the data is high (table 2), that is, the standard deviation is small for both the runtime and memory measurements. Despite this, there are some possible errors in the results and improvements in the method.

4.3.1 Theoretical

During the entire study, the uniform cost model was used, which assumes addition and multiplication of integers take constant time. While this is accurate for smaller integers, a more accurate model would use the logarithmic cost model. By using the logarithmic cost model other time and space complexities would possibly have been derived, however, we argue that these changes would be similar to a linear slowdown for all inputs of smaller size.

4.3.2 Experimental

During the study we used the Python 3 programming language to implement the algorithms. It was chosen due to its layer of abstraction that does, in part, limit the impact of the particular implementation. Furthermore it lacks a compiler, cache efficiency, and many optimisations, which is beneficial as such optimisations are often unpredictable. However, it also created an opaqueness in the processes executed and might have led to unknown changes in the data. We, the authors, do however consider the risks of this impacting the results in a meaningful way to be small.

However, what is most likely an artifact of the opaqeness of Python, were the jumps in the memory graph for algorithm 3 as shown in figure 1. These are suspected to have been caused when creating the hash set used to store the primes as these are absent in algorithm 3 without using the sieve of Eratosthenes (figure 2, bottom right) and in algorithm 2 when generating the primes (1, bottom center). This could have been caused by the hash set relocating when its reserved space is used, and then allocating more memory than required as a buffer. This limit would then be hit when executing for a certain n and cause it to use more memory than expected. For the next value of n, this limit would still be reached, but there would then be margin for some more primes.

In another vain, our ability to evaluate the accuracy of the algorithms was limited by the finite computation time. Some patterns might have emerged only at much larger values, and the differences between some time complexities might be difficult to discern with only the limited measurements which were available. We have relied on the theoretical results when comparing the algorithms and used the experimental data only to verify the results.

Similarly, a limited amount of benchmarks for each value of n was taken which might have introduced some elements of randomness. However, due to the low standard deviation of the results (see table 2) we doubt it considerably impacted our results or conclusion.

Due to background processes on the computer the tests were run on it is possible that the accuracy of some data points have been compromised. We attempted to mitigate this by using the time.process_time() method in Python which measures the time the program is active on the CPU, as well as by running with real time priority. The authors suggest that these measures have largely worked as in most graphs there are very few anomalies which suggest that if there where any background processes, the had minimal impact (figure 1). To lessen these risks further, a future study could run the benchmarks on a operating system with more manual control over data resources.

4.4 Future studies

While this study has focused entirely on the time and space complexity when executing the algorithms on a single threaded CPU, other factors can be relevant when discussing the efficiency of the algorithms as well.

As seen, the complexities of the three algorithms are similar, even a linear speedup could be relevant in the computational time for feasibly large values of n. A future study could therefore also evaluate the relative scaling factors of the complexities of the algorithms by implementing them in a low-level language and focusing on optimisations.

Another factor could be the feasibility of parallelizing the execution. If an algorithm can be executed over several cores and CPUs, possibly even making use of a GPU, it could improve its efficiency. We hypothesise that algorithm 2 and 3 would be the most apt for this, but leave it to a future study to examine these possibilities and provide a clearer insight.

References

Ankeny, N. C. (1952). The least quadratic non residue. *Annals of Mathematics*, *55*(1), 65–72. http://www.jstor.org/stable/1969420

Bober, J. W., & Goldmakher, L. (2015). Pólya–vinogradov and the least quadratic nonresidue. *Mathematische Annalen*, *366*(1-2), 853–863. https://doi.org/10.1007/s00208-015-1353-2

Bressoud, D., & Wagon, S. (2000). *A course in computational number theory*. Key College Publ.

Brillhart, J. (1972). Note on representing a prime as a sum of two squares. *Mathematics of Computation*, *26*(120), 1011–1013. http://www.jstor.org/stable/2005889

Burgess, D. A. (1957). The distribution of quadratic residues and non-residues. *Mathematika*, *4*(2), 106–112. https://doi.org/10.1112/S0025579300001157

Gethner, E., Wagon, S., & Wick, B. (1998). A stroll through the gaussian primes. *The American Mathematical Monthly*, *105*(4), 327–337. http://www.jstor.org/stable/2589708

Greene, D. H., & Knuth, D. E. (1990). *Mathematics for the analysis of algorithms* (3. ed.). Birkhäuser.

Herstein, I. N. (1975). *Topics in algebra* (2. ed.). John Wiley.

Irving, R. S. (2004). *Integers, polynomials, and rings : A course in algebra*. Springer.

Pritchard, P. (1981). A sublinear additive sieve for finding prime number. *Commun. ACM*, *24*(1), 18–23. https://doi.org/10.1145/358527.358540

Soprunov, I. (2010). A short proof of the prime number theorem for arithmetic progressions.

Stein, R. G. (1976). Exploring the gaussian integers. *The Two-Year College Mathematics Journal*, *7*(4), 4–10. http://www.jstor.org/stable/3027049

Uspensky, J. V., & Heaslet, M. A. (1939). *Elementary number theory*. McGraw-Hill.

Wagon, S. (1990). Editor's corner: The euclidean algorithm strikes again. *The American Mathematical Monthly*, *97*(2), 125–129. http://www.jstor.org/stable/2323912

Wegener, I. (2005). *Complexity theory : Exploring the limits of efficient algorithms*. Springer.

A Introduction to Stieltjes integrals

The Stieltjes integral is defined as follows

Definition A.1 (Greene and Knuth, 1990)**.** *1. Let f and g be real-valued functions on* $[a,b]$*.*

2. *Let P be a partition of* $[a,b]$ *into* $a = x_0 < x_1 < \ldots x_n = b$*.*

3. *Define a sum,*

$$S(P) = \sum_{0 \le k < n} f(t_k)(g(x_{k+1}) - g(x_k)), t_k \in [x_k, x_{k+1}]$$

4. *Then A is the value of the Stieltjes integral* $\int_a^b f(t)\,dg(t)$ *if and only if for all* $\varepsilon > 0$ *there exists a* P_ε *such that all refinements P of* P_ε *lead to sums near A, that is,* $\mid S(P) - A \mid < \varepsilon$*.*

The properties we will make use of are:

1. (Integration by parts) If $\int_a^b f(t)\,dg(t)$ exists then $\int_a^b g(t)\,df(t)$ exists and the sum of these two integrals is $f(t)g(t)\mid_a^b$.

2. If $\int_a^b f(t)\,dg(t)$ exists and $g'(t)$ is continuous on $[a,b]$, then

$$\int_a^b f(t)\,dg(t) = \int_a^b f(t)g'(t)\,dt.$$

3. If g is monotone increasing on $[a,b]$, f is positive on $[a,b]$, and both integrals exists, then

$$\int_a^b O(f(t))\,dg(t) = O\left(\int_a^b f(t)\,dg(t)\right).$$

4. If f and g are monotone increasing positive functions on $[a,b]$ and the integrals exist then

$$\int_a^b f(t)\,dO(g(t)) = O(f(a)g(a)) + O(f(b)g(b)) + O\left(\int_a^b f(t)\,dg(t)\right)$$

B Derivation of theoretical results

The purpose of this appendix is to fill in the details in how we derived our theoretical results. The most important and difficult of which is finding the big O of summations of variable length. The method we used to solve these have been inspired by Greene and Knuth (1990), particularly example 4.2.3 from that book. We use Stieltjes integrals, introduced in appendix A, to rewrite the sums as integrals with the prime counting function as the differential and then we approximate the integral by substituting the prime counting function for something that it is big O of. From here we use the theorems from Greene and Knuth (1990), which are restated in appendix A for the convenience of the reader, to manipulate the integral until we attain a big O expression for the sum.

In the analysis of algorithm 1 we used the following result. Note that the bound is n but because of the rules for logarithms it can be swapped to $\sqrt{n}$ as we did in the analysis. It is proved in Greene and Knuth (1990) although in a different way from what is given here.

Lemma B.1. *The sum* $\sum_{p \ prime}^{n} \frac{1}{p}$ *is* $O(\log\log n)$.

Proof. We write the sum as a Stieltjes integral

$$\sum_{p \ prime}^{n} \frac{1}{p} = \int_{c}^{n} \frac{1}{x} d\pi(x)$$

where c is a constant. The prime counting function $\pi(x)$ is $O(x/\log x)$ (Soprunov, 2010) so the above integral can be approximated as

$$\int_{c}^{n} \frac{1}{x} dO\left(\frac{x}{\log x}\right). \tag{17}$$

Using propery 4 from appendix A we get that (17) is

$$O\left(\frac{1}{\log n}\right) + O\left(\frac{1}{\log c}\right) + O\left(\int_{c}^{n} \frac{1}{x} d\frac{x}{\log x}\right). \tag{18}$$

Since c is constant and using property 2 from appendix A on the third term we get that (18) is

$$\begin{aligned} &O\left(\frac{1}{\log n}\right) + O(1) + O\left(\int_{c}^{n} \frac{1}{x}\left(\frac{1}{\log x} - \frac{1}{\log^2 x}\right) dx\right) \\ &= O\left(\frac{1}{\log n}\right) + O(1) + O\left(\int_{c}^{n} \frac{1}{x\log x} - \frac{1}{x\log^2 x} dx\right). \end{aligned} \tag{19}$$

Since $\int \frac{dx}{x\log x} = \log\log x + C$ and $\int \frac{dx}{x\log^2 x} = -\frac{1}{\log x} + C$ (19) is

$$O\left(\frac{1}{\log n}\right) + O(1) + O\left(\log\log n - \frac{1}{\log n}\right) = O(\log\log n)$$

which proves the lemma. $\square$

In the analysis of algorithm 2 the following lemma is paramount. Here a general version is proved since we used it in different derivations using different bounds on the least quadratic non-residue.

Lemma B.2. *For any function $f(x,\varepsilon)$ that is positive and monotone increasing for large arguments of x, that has a continuous derivative with respect to x and if a and b are integers* *such that* $\gcd(a,b)=1$*, then*

$$\sum_{\substack{p \text{ prime} \\ p\equiv a \bmod b}}^{n} O_\varepsilon\left(f(p,\varepsilon)\right) = O_\varepsilon\left(\frac{nf(n,\varepsilon)}{\log n}\right).$$

Proof. Let $g(x)$ be the function that is $O_\varepsilon\left(f(p,\varepsilon)\right)$. From the definition of big-O we know that $g(x) \le C(\varepsilon)f(p,\varepsilon)$ where $C(\varepsilon)$ is a constant dependent on ε. We get that

$$\sum_{\substack{p \text{ prime} \\ p\equiv a \bmod b}}^{n} g(x) \le \sum_{\substack{p \text{ prime} \\ p\equiv a \bmod b}}^{n} C(\varepsilon)f(p,\varepsilon) = C(\varepsilon)\left(\sum_{\substack{p \text{ prime} \\ p\equiv a \bmod b}}^{n} f(p,\varepsilon)\right). \tag{20}$$

From the definition of big-O we then get

$$\sum_{\substack{p \text{ prime} \\ p\equiv a \bmod b}}^{n} O_\varepsilon\left(f(p,\varepsilon)\right) = O_\varepsilon\left(\sum_{\substack{p \text{ prime} \\ p\equiv a \bmod b}}^{n} f(p,\varepsilon)\right). \tag{21}$$

Using Stieltjes integrals we can rewrite the discrete sum on the RHS to an integral with the prime counting function $\pi_{a,b}(n)$ for primes congruent to a modulo b. We get that

$$\sum_{\substack{p \text{ prime} \\ p\equiv a \bmod b}}^{n} f(p,\varepsilon) = \int_c^n f(x,\varepsilon)\ d\pi_{a,b}(x) \tag{22}$$

where c is a sufficiently large constant. We are going to use a theorem which states that $\pi_{a,b}(x)$ is $O(x/\log x)$ regardless of what a and b are, as long as they are relatively prime (Soprunov, 2010). By substituting $\pi_{a,b}(x)$ for this big O expression and using rules for Stieltjes integrals we get what the sum of the LHS in (22) is big O of. Simplifying, we get that

$$\int_c^n f(x,\varepsilon)\ dO\left(\frac{x}{\log x}\right) = O\left(f(n,\varepsilon)\frac{n}{\log n}\right) + O\left(f(c,\varepsilon)\frac{c}{\log c}\right) + O\left(\int_c^n f(x,\varepsilon)\ d\frac{x}{\log x}\right) \tag{23}$$

$$= O\left(\frac{nf(n,\varepsilon)}{\log n}\right). \tag{24}$$

The first equality simply comes from the application of property 4 from appendix A since $f(x,\varepsilon)$ is monotone increasing and positive and we are assuming that the integral exists. The term in the middle is constant since c is constant so it is just $O(1)$. Furthermore, $\int_c^n f(x,\varepsilon)\, d\frac{x}{\log x}$ is $O\left(nf(n,\varepsilon)/\log n\right)$ which yields the second equality. To see this we apply integration by parts (property 1) on this integral and get that

$$\int_c^n f(x,\varepsilon)\ d\frac{x}{\log x} = f(n,\varepsilon)\frac{n}{\log n} - f(c,\varepsilon)\frac{c}{\log c} - \int_c^n \frac{1}{\log x} - \frac{1}{\log^2 x}\ df(x,\varepsilon) \tag{25}$$

since the derivative of $x/\log x$ is $1/\log x - 1/\log^2 x$. Additionally, we can apply property 2 to the last term of the RHS since the derivative of $f(x,\varepsilon)$ with respect to x is continuous and get

$$\frac{nf(n,\varepsilon)}{\log n} - \frac{cf(c,\varepsilon)}{\log c} - \int_c^n \frac{df(x,\varepsilon)}{dx}\left(\frac{1}{\log x} - \frac{1}{\log^2 x}\right) dx. \tag{26}$$

Since $\frac{df(x,\varepsilon)}{dx} \geq 0$ and $1/\log x - 1/\log^2 x > 0$ for $x > c$ the last integral is positive and we get that

$$\frac{nf(n,\varepsilon)}{\log n} > \int_c^n f(x,\varepsilon)\, d\frac{x}{\log x} \tag{27}$$

which proves the equality (24). From (21) we further get that the constant factor, that is multiplied by $nf(n,\varepsilon)/\log n$ for it to be bigger than our original sum, is dependent on ε which finally gives us our desired result. □

Every time we add two big O expressions we need to know which one is bigger i.e. which one is big O of the other. In some cases, as in the analysis of algorithm 1, it is clear which function is bigger. However when comparing powers of logarithms and polynomials as we did in the analysis of algorithm 2 the following fact may be useful.

Lemma B.3 (Wegener, 2005)**.** *For all constants $k > 0$ and $\varepsilon > 0$ the following relations are true*

$$(\log\log n)^k = O(\log^\varepsilon n), \tag{28}$$

$$\log^k n = O(n^\varepsilon). \tag{29}$$

C Code

Source code to implementations can be found on https://github.com/ZekeWK/GyArbete, but is also appended here.

All files should be in a single folder and the file named Benchmarking.py should be run with Python 3 to generate the benchmarks.

Note that in the code for algorithm 1, the cash variable was supposed to be cashed_b. This bug was noticed after the code was run and probably had very minor impacts on results and thereby the conclusion.

Benchmarking.py:

```python
from Algorithm1 import Algorithm1
from Algorithm2 import Algorithm2
from Algorithm3 import Algorithm3

import Erastothenes

import tracemalloc
import time
from datetime import datetime
from tqdm import tqdm
from copy import deepcopy
from bisect import bisect_right
import statistics

def main():
    inputs = range(100000, 5000001, 100000)
    iterations_time = 5
    iterations_memory = 3

    benchmarks = benchmark_the_algorithms(inputs, iterations_time, iterations_memory)

    output_string = "Benchmark run at: " + str(datetime.now()) + " , with iterations_time: " + str(iterations_time) + " , and with iterations_memory: " + str(iterations_memory) + ".\n"
    output_string += benchmarks_to_readable(benchmarks)

    with open("results.csv", "a") as f:
        f.write(output_string)

    with open("last_results.csv", "w+") as f:
        f.write(output_string)

    print(output_string)

def benchmarks_to_readable(benchmarks):
    output_string = str()
    for benchmark_type, string in zip(benchmarks, ["Input, Time, Standard Deviation:\n", "Input, Memory, Standard Deviation:\n"]):
        output_string += string
        for benchmark in benchmark_type:
            output_string += str(benchmark[0]) + "\n"
            for test in benchmark[1]:
                output_string += str(test[0]) + "; " + str(test[1]) + "; " + str(test[2]) + "\n"
            output_string += "\n"
    return output_string

def benchmark_the_algorithms(inputs, iterations_time, iterations_memory):
    global natural_primes
    natural_primes = Erastothenes.ErastothenesSieve(max(inputs))

    time_benchmarks = []
    memory_benchmarks = []
    for algorithm in [run_Algorithm1, run_Algorithm2, run_Algorithm3, run_Algorithm2_Primes_Precalculated, run_Algorithm3_Primes_Precalculated]:
        time_benchmarks   .append(get_benchmark_of(get_time_used,   algorithm,
   tqdm(deepcopy(inputs)), iterations_time))
        memory_benchmarks .append(get_benchmark_of(get_memory_peak, algorithm,
   tqdm(deepcopy(inputs)), iterations_memory))
    return (time_benchmarks, memory_benchmarks)

def get_benchmark_of(test, algorithm, inputs, iterations_per):
    benchmarks = []
    for input in inputs:
        (mean, standard_deviation) = get_mean_usage(test, algorithm, input,
   iterations_per)
        benchmarks.append((input, mean, standard_deviation))
    return (algorithm, benchmarks)

def get_mean_usage(test, algorithm, input, iterations):
    global natural_primes

    if algorithm == run_Algorithm2_Primes_Precalculated:
        primes = natural_primes[0:bisect_right(natural_primes, input)]
    elif algorithm == run_Algorithm3_Primes_Precalculated:
        primes = set(natural_primes[0:bisect_right(natural_primes, input)].copy
   ())
    else:
        primes = None

    results = []
    for _i in range(iterations):
        results.append(test(algorithm, input, primes))

    mean = statistics.mean(results)
    standard_deviation = statistics.pstdev(results, mean)

    return (mean, standard_deviation)

def get_memory_peak(algorithm, input, primes):
    tracemalloc.start()
    algorithm(input, primes)
    memory_peak = tracemalloc.get_traced_memory()[1]
    tracemalloc.stop()
    return memory_peak

def get_time_used(algorithm, input, primes):
    start_time = time.process_time()
    algorithm(input, primes)
    end_time = time.process_time()
    return end_time - start_time

def run_Algorithm1(input, primes):
    Algorithm1(input)

def run_Algorithm2(input, primes):
```

```python
    Algorithm2(input, Erastothenes.ErastothenesSieve(input))

def run_Algorithm3(input, primes):
    Algorithm3(input, set(Erastothenes.ErastothenesSieve(input)))

def run_Algorithm2_Primes_Precalculated(input, primes):
    Algorithm2(input, primes)

def run_Algorithm3_Primes_Precalculated(input, primes):
    Algorithm3(input, primes)

if __name__ == "__main__":
    print("Press enter to activate")
    input()
    print("Activated")
    main()
    print("Done")
    input()
```

GaussianInteger.py

```python
import math

class GaussianInteger:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def new(tuple):
        return GaussianInteger(tuple[0], tuple[1])

    def __str__(self):
        if self.b >= 0:
            return str(self.a) + " + " + str(self.b) + "i"
        else:
            return str(self.a) + " - " + str(-self.b) + "i"

    def __abs__(self):
        return int(math.sqrt(self.a**2 + self.a**2))

    def norm(self):
        return self.a**2 + self.b**2

    def conjugate(self):
        return GaussianInteger(self.a, -self.b)

    def real(self):
        return self.a

    def imaginary(self):
        return self.b

    def __add__(self, other):
        return GaussianInteger(self.a + other.a, self.b + other.b)

    def __sub__(self, other):
        return GaussianInteger(self.a - other.a, self.b - other.b)

    def __mul__(self, other):
        if type(other) == GaussianInteger:
            return GaussianInteger(self.a * other.a - self.b * other.b, self.a * other.b + self.b * other.a)
        else:
            return GaussianInteger(self.a * other, self.b * other)
    __rmul__ = __mul__

    def __floordiv__(self, other):
        if type(other) == GaussianInteger:
            return (self * other.conjugate()) // other.norm()
        else:
            return GaussianInteger(self.a / other, self.b / other)

    def get_tuple(self):
        return (self.a, self.b)

    def __eq__(self, other) -> bool:
        return (self.a, self.b) == (other.a, other.b)

    def __ne__(self, other) -> bool:
        return (self.a, self.b) != (other.a, other.b)

    def __lt__(self, other) -> bool:
        return (self.norm() < other.norm() or (self.norm() == other.norm() and (self.a < self.b or (self.a == other.a and self.b < other.b))))

    def __key(self):
        return (self.a, self.b)

    def __hash__(self):
        return hash(self.__key())
```

Eratosthenes.py

```python
import math

def ErastothenesSieve(n):
    sqrt_n = math.floor(math.sqrt(n))

    possible_primes = [True for x in range(n + 1)]

    for number in range(2, sqrt_n + 1):
        if not possible_primes[number]:
            continue

        for non_prime in range(number * 2, n + 1, number):
            possible_primes[non_prime] = False

    primes = []

    for number in range(2, n + 1):
        if possible_primes[number]:
            primes.append(number)

    return primes
```

Algorithm1.py:

```python
import GaussianInteger as GI
import math

def Algorithm1(n):
    sqrt_n = math.isqrt(n)

    cashed_b = (None, None)
    def get_b(a):
        if cashed_b[0] == a:
            return cashed_b[1]
        else:
            val = min(a, math.isqrt(n-(a**2)))+1
            cash = (a, val)
            return val

    possible_gaussian_primes = [[True for b in range(0, get_b(a))] for a in range(1, sqrt_n + 1)]

    def in_bounds_possible_gaussian_primes(gaussian_integer):
        (a, b) = gaussian_integer.get_tuple()

        return 0 <= a-1 < len(possible_gaussian_primes) and 0 <= b < len(possible_gaussian_primes[a-1])

    def remove_non_prime(gaussian_non_prime):
        if in_bounds_possible_gaussian_primes(gaussian_non_prime):
            possible_gaussian_primes[gaussian_non_prime.real()-1][gaussian_non_prime.imaginary()] = False

    def readd_prime(gaussian_prime):
        if in_bounds_possible_gaussian_primes(gaussian_prime):
            possible_gaussian_primes[gaussian_prime.real()-1][gaussian_prime.imaginary()] = True
```

```python

def get_gaussian_prime(possible_gaussian_prime):
    if in_bounds_possible_gaussian_primes(possible_gaussian_prime):
        return possible_gaussian_primes[possible_gaussian_prime.real() -1][possible_gaussian_prime.imaginary()]
    return False

def remove_multiples(z):
    n_div_norm =  n // z.norm()
    sqrt_n_div_norm = math.isqrt(n_div_norm)

    for x in range(1, sqrt_n_div_norm + 1):
        for y in range(sqrt_n_div_norm + 1):
            w = GI.GaussianInteger(x, y)
            if w.norm() > n_div_norm:
                break

            product1 = z             * w
            product2 = z.conjugate() * w
            product3 = z             * w.conjugate()
            product4 = z.conjugate() * w.conjugate()

            if product1.real() < product1.imaginary() and product2.real() < product2.imaginary() and 0 > product3.imaginary() and 0 > product4.imaginary():
                break

            remove_non_prime(product1)
            remove_non_prime(product2)
            remove_non_prime(product3)
            remove_non_prime(product4)

    readd_prime(z)

remove_non_prime(GI.GaussianInteger(1, 0))

for gaussian_integer in (GI.GaussianInteger(1, 1), GI.GaussianInteger(2, 1), GI.GaussianInteger(3, 0)):
    remove_multiples(gaussian_integer)

sqrt_2_sqrt_n = math.isqrt(sqrt_n*2)
for manhattan_distance in range(2, sqrt_2_sqrt_n + 1):
    manhattan_distance_div_2 = manhattan_distance//2 + 1

    for a in range(manhattan_distance_div_2, manhattan_distance + 1):
        b = manhattan_distance - a

        z = GI.GaussianInteger(a, b)

        if get_gaussian_prime(z):
                remove_multiples(z)

    gaussian_primes = []
    for (a, values) in enumerate(possible_gaussian_primes):
        for (b, value) in enumerate(values):
            if value:
                if b ≠ 0:
                    gaussian_primes.append(GI.GaussianInteger(a + 1,b))
                    gaussian_primes.append(GI.GaussianInteger(a + 1,b).
   conjugate())
                else:
                    gaussian_primes.append(GI.GaussianInteger(a+1, 0))
    return gaussian_primes

if __name__ == "__main__":
    input = 100
    result = Algorithm1(input)
    result.sort()
    for i in result:
        print(i)
```

Algorithm2.py

```python
import Erastothenes
import GaussianInteger as GI
import math

def Algorithm2(n, natural_primes_list):
    global natural_primes

    natural_primes = natural_primes_list

    sqrt_n = math.sqrt(n)

    gaussian_primes = [GI.GaussianInteger(1, 1), GI.GaussianInteger(1, -1)]
    for natural_prime in natural_primes:
        if natural_prime % 4 == 1:
            gaussian_prime = find_two_squares_that_sum_to(natural_prime)

            gaussian_primes.append(gaussian_prime)
            gaussian_primes.append(gaussian_prime.conjugate())

        elif natural_prime % 4 == 3:
            if natural_prime ⩽ sqrt_n:
                gaussian_primes.append(GI.GaussianInteger(natural_prime, 0))

    return gaussian_primes

def eulers_criterion(prime):
    global natural_primes

    exponent = (prime - 1) // 4

    for natural_prime in natural_primes:
        possible_root = pow(natural_prime, exponent, prime)

        if pow(possible_root, 2, prime) == prime -1:
            return possible_root

def euclids_algorithm_stop_early(a, b, stop_size):
    if a < stop_size:
        return (a, b)
    return euclids_algorithm_stop_early(b, a%b, stop_size)

def find_two_squares_that_sum_to(prime):
    return GI.GaussianInteger.new(euclids_algorithm_stop_early(prime,
   eulers_criterion(prime), math.sqrt(prime)))

if __name__ == "__main__":
    input = 100
    result = Algorithm2(input, Erastothenes.ErastothenesSieve(input))
    result.sort()
    for i in result:
        print(i)
```

Algorithm3.py

```python
import GaussianInteger as GI
import Erastothenes
import math

def Algorithm3(n, natural_primes_set):
    natural_primes = natural_primes_set

    gaussian_primes = []

    sqrt_n = math.isqrt(n)
    for a in range(sqrt_n + 1):

        max_b = min(math.isqrt(n - a**2), a)
        for b in range(1, max_b + 1):

            norm = a**2 + b**2

            if norm in natural_primes:
                gaussian_primes.append(GI.GaussianInteger(a, b))
                gaussian_primes.append(GI.GaussianInteger(a, b).conjugate())

        if a % 4 == 3 and a in natural_primes:
            gaussian_primes.append(GI.GaussianInteger(a, 0))

    return gaussian_primes

if __name__ == "__main__":
    input = 100
    result = Algorithm3(input, set(Erastothenes.ErastothenesSieve(input)))
    result.sort()
    for i in result:
        print(i)
```

Verifier.py

```python
import Erastothenes

from Algorithm1 import Algorithm1
from Algorithm2 import Algorithm2
from Algorithm3 import Algorithm3

def run_Algorithm1(input):
    return Algorithm1(input)

def run_Algorithm2(input):
    return Algorithm2(input, Erastothenes.ErastothenesSieve(input))

def run_Algorithm3(input):
    return Algorithm3(input, set(Erastothenes.ErastothenesSieve(input)))

def main():
    inputs = range(10000, 10010)

    for n in inputs:
        result1 = sorted(run_Algorithm1(n))
        result2 = sorted(run_Algorithm2(n))
        result3 = sorted(run_Algorithm3(n))

        result1 = set(result1)
        result2 = set(result2)
        result3 = set(result3)

        differed = sorted(result1.symmetric_difference(result2).union(result1.symmetric_difference(result3)).union(result2.symmetric_difference(result3)))

        print("For n :" + str(n))

        for i in differed:

            sets = [' ', ' ', ' ']

            if i in result1:
                sets[0] = '1'

            if i in result2:
                sets[1] = '2'

            if i in result3:
                sets[2] = '3'

            print(str(i) + "      "+ "".join(sets))

    print("Done")



if __name__ == "__main__":
    main()
```

D Raw measurements

The following are the raw measurements of the experimental study. Time is measured in seconds, memory peak in bytes and StdDev is the standard deviation of the preceding column over the attempts. Also accessible at https://github.com/ZekeWK/GyArbete.

Algorithm 1

Input:	Time:	StdDev:	Memory:	StdDev:
100000	1.90625	0.009882	1109935	92.68345
200000	3.875	0.022097	2204641	156.9147
300000	5.778125	0.030298	3268325	112.7751
400000	7.78125	0.026146	4315215	103.9145
500000	9.7625	0.030298	5355455	25.3684
600000	11.78125	0.047393	6373705	20.7418
700000	13.7625	0.077434	7378163	20.7418
800000	15.81563	0.111541	8382763	25.3684
900000	17.8625	0.098027	9382094	28.28427
1000000	19.65	0.069597	10380029	11.46977
1100000	21.74688	0.103833	11388402	5.656854
1200000	23.67188	0.040745	12389637	11.46977
1300000	25.80938	0.0875	13343087	13.19933
1400000	27.73438	0.104115	14345317	11.46977
1500000	29.6625	0.049014	15278480	5.656854
1600000	31.69063	0.091856	16280709	11.46977
1700000	33.69688	0.098127	17230103	13.19933
1800000	35.64063	0.111366	18222491	11.46977
1900000	37.71875	0.218303	19154414	5.656854

2000000	39.85313	0.250702	20152021	11.46977
2100000	41.89375	0.169558	21094203	13.19933
2200000	43.79063	0.235642	22017663	11.46977
2300000	45.97813	0.223301	23016384	5.656854
2400000	48.00938	0.118421	23925407	13.19933
2500000	50.03125	0.28125	24829335	13.19933
2600000	51.99688	0.217496	25861939	11.46977
2700000	54.32188	0.2702	26781694	5.656854
2800000	56.17188	0.439171	27698286	144.702
2900000	57.99688	0.045715	28715205	11.46977
3000000	59.90625	0.069175	29610891	11.46977
3100000	62.05625	0.10203	30516324	5.656854
3200000	64.19375	0.188176	31422374	5.656854
3300000	67.00938	0.663914	32457969	1583.018
3400000	69.05313	0.564285	33369323	11.46977
3500000	70.56563	0.22716	34274776	18.18424
3600000	72.77813	0.1628	35170229	11.46977
3700000	74.91563	0.330217	36080782	5.656854
3800000	76.64688	0.099117	37105305	11.46977
3900000	78.51875	0.158546	37999226	5.656854
4000000	80.55313	0.210236	38885855	11.46977
4100000	82.60625	0.213646	39780659	13.19933
4200000	84.65313	0.299642	40690805	11.46977
4300000	86.57813	0.259206	41737717	11.46977
4400000	88.60625	0.202378	42632557	11.46977
4500000	90.91563	0.222863	43539459	13.19933
4600000	93.0375	0.24379	44427099	11.46977
4700000	95.61875	0.283257	45321128	152.8747
4800000	97.09688	0.253183	46370589	11.46977
4900000	98.92188	0.113966	47259137	11.46977
5000000	101.0406	0.15392	48139945	137.1844

Algorithm 2a

100000	0.09375	0.009882	1103521	26.39865
200000	0.196875	0.007655	2136415	1812.656
300000	0.29375	0.00625	3144130	1891.192
400000	0.396875	0.007655	4124231	2055.165
500000	0.503125	0.011693	5103145	34.92214
600000	0.596875	0.00625	6042725	26.39865
700000	0.7	0.00625	6974579	26.39865
800000	0.8	0.00625	7901793	190.4474

900000	0.9	0.007655	8836847	40.83571
1000000	1	0.009882	9762856	183.3903
1100000	1.103125	0.007655	10709646	271.2342
1200000	1.203125	0	11614399	145.1191
1300000	1.30625	0.007655	12502413	168.1375
1400000	1.403125	0.00625	13453337	143.4558
1500000	1.509375	0.007655	14288988	39.59798
1600000	1.615625	0.007655	15240483	160.4106
1700000	1.709375	0.007655	16079491	13.19933
1800000	1.815625	0.022964	17035197	146.5818
1900000	1.909375	0.00625	17868627	13.19933
2000000	2.015625	0.017116	18762528	0
2100000	2.115625	0.007655	19663969	156.5063
2200000	2.215625	0.00625	20479227	13.19933
2300000	2.31875	0.007655	21374323	7.542472
2400000	2.44375	0.023385	22267416	149.9867
2500000	2.55	0.022964	23075437	7.542472
2600000	2.621875	0.00625	23990627	277.6345
2700000	2.728125	0.007655	24891567	331.9331
2800000	2.828125	0.009882	25697801	269.4851
2900000	2.921875	0.013975	26609525	150.9908
3000000	3.021875	0.015934	27509547	13.19933
3100000	3.134375	0.021195	28315340	0
3200000	3.23125	0.011693	29125425	157.8635
3300000	3.334375	0.007655	30042562	164.0488
3400000	3.43125	0.007655	30959461	136.7951
3500000	3.53125	0.017116	31750899	13.19933
3600000	3.6375	0.011693	32535760	0
3700000	3.740625	0.007655	33339797	303.5845
3800000	3.85	0.02898	34260267	177.2481
3900000	3.95	0.011693	35189717	154.6207
4000000	4.046875	0.009882	35976753	13.19933
4100000	4.14375	0.015309	36774508	0
4200000	4.25	0.009882	37563616	0
4300000	4.378125	0.034799	38491550	164.0488
4400000	4.4375	0	39424919	13.19933
4500000	4.55	0.00625	40218670	147.0782
4600000	4.65625	0.013975	40996255	177.2481
4700000	4.775	0.035078	41782567	13.19933
4800000	4.86875	0.015934	42726097	160.2775
4900000	4.96875	0.013975	43511478	0

5000000	5.059375	0.00625	44463538	269.5774

Algorithm 3a

100000	0.06875	0.007655	1314829	26.39865
200000	0.14375	0.00625	2258900	28.28427
300000	0.215625	0.00625	3954371	41.09609
400000	0.290625	0.007655	4848451	42.12152
500000	0.3625	0.00625	5733873	24.51304
600000	0.4375	0	6597876	33.94113
700000	0.509375	0.007655	7452042	33.94113
800000	0.58125	0.00625	8298958	0
900000	0.65625	0.009882	9149869	26.39865
1000000	0.734375	0.009882	9988907	26.39865
1100000	0.809375	0.00625	11891711	26.39865
1200000	0.88125	0.007655	12746353	30.8689
1300000	0.959375	0.007655	13537892	33.94113
1400000	1.03125	0.009882	14386296	33.94113
1500000	1.103125	0.007655	15173237	34.92214
1600000	1.184375	0.00625	16016529	175.4638
1700000	1.259375	0.007655	16805912	33.94113
1800000	1.33125	0.011693	17646623	30.8689
1900000	1.43125	0.042619	18431433	30.8689
2000000	1.4875	0.011693	19277455	30.8689
2100000	1.553125	0.0125	20055189	7.542472
2200000	1.65	0.015934	22919460	33.94113
2300000	1.725	0.0125	23767303	175.4638
2400000	1.8	0.011693	24528157	7.542472
2500000	1.875	0.009882	25289002	33.94113
2600000	1.953125	0.009882	26156349	7.542472
2700000	2.071875	0.021195	26914833	7.542472
2800000	2.109375	0.009882	27673417	7.542472
2900000	2.18125	0.0125	28537829	7.542472
3000000	2.253125	0.00625	29284181	7.542472
3100000	2.33125	0.00625	30042757	7.542472
3200000	2.409375	0.011693	30805528	0
3300000	2.49375	0.015934	31675891	7.542472
3400000	2.553125	0.007655	32424986	0
3500000	2.640625	0.009882	33170319	7.542472
3600000	2.7125	0.015934	33909623	7.542472
3700000	2.8	0.011693	34666361	13.19933
3800000	2.853125	0.01875	35540700	0

3900000	2.959375	0.015934	36287545	154.6207
4000000	3.01875	0.011693	37028391	13.19933
4100000	3.103125	0.0125	37779501	7.542472
4200000	3.171875	0.009882	38522392	0
4300000	3.25	0.022097	39404833	7.542472
4400000	3.325	0.015309	40139350	0
4500000	3.46875	0.027951	45080947	158.4705
4600000	3.53125	0.013975	45813089	7.542472
4700000	3.621875	0.018222	46553211	7.542472
4800000	3.709375	0.030619	47450835	7.542472
4900000	3.790625	0.040263	48190389	158.4705
5000000	3.84375	0.022097	48924482	0

Algorithm 2b

100000	0.0625	0	918918	0
200000	0.134375	0.007655	1777462	2002.526
300000	0.203125	0	2624441	1940.722
400000	0.275	0.007655	3447556	1838.478
500000	0.340625	0.011693	4264259	2028.412
600000	0.4125	0.007655	5061442	328.0975
700000	0.48125	0.00625	5848987	9.42809
800000	0.55	0.00625	6629065	173.4769
900000	0.61875	0.007655	7414666	183.3903
1000000	0.6875	0	8188542	589.2391
1100000	0.753125	0.00625	8979374	149.9867
1200000	0.825	0.00625	9722924	130.1076
1300000	0.89375	0.011693	10497407	321.8999
1400000	0.959375	0.007655	11283279	173.4769
1500000	1.04375	0.026882	12006627	159.2427
1600000	1.10625	0.018222	12788528	33.94113
1700000	1.16875	0.00625	13514915	23.17086
1800000	1.2375	0.00625	14294597	30.8689
1900000	1.309375	0.00625	15016903	30.8689
2000000	1.375	0	15726924	299.92
2100000	1.44375	0.007655	16518559	24.51304
2200000	1.509375	0.007655	17223748	312.5124
2300000	1.578125	0.009882	17926907	30.8689
2400000	1.646875	0.007655	18711284	0
2500000	1.715625	0.011693	19411271	139.5357
2600000	1.784375	0.011693	20124764	33.94113
2700000	1.85625	0.00625	20917140	328.0975

2800000	1.921875	0	21615802	164.0488
2900000	2.01875	0.022964	22315062	0
3000000	2.0625	0.013975	23107753	156.5063
3100000	2.125	0	23806831	263.1974
3200000	2.2	0.00625	24509470	164.0488
3300000	2.26875	0.011693	25201854	164.0488
3400000	2.33125	0.00625	26012452	0
3500000	2.403125	0.00625	26697978	284.1408
3600000	2.471875	0.015309	27378184	164.0488
3700000	2.553125	0.007655	28075038	0
3800000	2.60625	0.011693	28757792	0
3900000	2.678125	0.007655	29581476	147.0782
4000000	2.74375	0.007655	30264070	147.0782
4100000	2.809375	0.00625	30955768	147.0782
4200000	2.890625	0.024206	31640345	150.9908
4300000	2.959375	0.0125	32315322	147.0782
4400000	3.01875	0.00625	33144304	0
4500000	3.084375	0.015934	33832238	147.0782
4600000	3.165625	0.0125	34505944	164.0488
4700000	3.228125	0.007655	35187888	0
4800000	3.296875	0.009882	35860320	0
4900000	3.359375	0.009882	36540938	0
5000000	3.43125	0.007655	37388667	276.0161

Algorithm 3b

100000	0.040625	0.007655	906866	0
200000	0.08125	0.00625	1716664	0
300000	0.11875	0.007655	2497470	11.31371
400000	0.1625	0.007655	3265744	0
500000	0.20625	0.00625	4028332	0
600000	0.246875	0.00625	4771352	0
700000	0.2875	0.007655	5506417	26.39865
800000	0.328125	0	6234810	0
900000	0.36875	0.007655	6968534	0
1000000	0.4125	0.007655	7692007	26.39865
1100000	0.453125	0.009882	8430760	0
1200000	0.49375	0.007655	9169922	181.0193
1300000	0.5375	0.007655	9848024	0
1400000	0.578125	0	10582767	26.39865
1500000	0.615625	0.007655	11257264	33.94113
1600000	0.6625	0.0125	11988846	0

1700000	0.70625	0.011693	12666148	33.94113
1800000	0.75	0.009882	13395944	33.94113
1900000	0.825	0.04239	14069442	33.94113
2000000	0.8375	0.0125	14805000	33.94113
2100000	0.875	0	15472734	0
2200000	0.93125	0.015934	16130160	0
2300000	0.975	0.0125	16868343	26.39865
2400000	1.0125	0.015309	17520745	169.3701
2500000	1.053125	0.015934	18173166	33.94113
2600000	1.1	0.007655	18931566	33.94113
2700000	1.146875	0.007655	19581810	33.94113
2800000	1.190625	0.00625	20232922	33.94113
2900000	1.23125	0.015309	20989510	166.6613
3000000	1.278125	0.00625	21628422	33.94113
3100000	1.31875	0.007655	22280182	33.94113
3200000	1.353125	0.0125	22935500	33.94113
3300000	1.40625	0.009882	23698788	33.94113
3400000	1.440625	0.011693	24341678	33.94113
3500000	1.484375	0.013975	24981216	33.94113
3600000	1.528125	0.025	25615328	33.94113
3700000	1.578125	0.009882	26265310	33.94113
3800000	1.609375	0.009882	27034219	30.8689
3900000	1.665625	0.0125	27674976	33.94113
4000000	1.709375	0.015934	28311484	166.6613
4100000	1.7875	0.040263	28956438	33.94113
4200000	1.79375	0.026882	29594660	33.94113
4300000	1.825	0.011693	30372965	30.8689
4400000	1.871875	0.011693	31003306	33.94113
4500000	1.959375	0.027243	31644697	30.8689
4600000	1.996875	0.018222	32273050	33.94113
4700000	2.04375	0.020729	32908804	33.94113
4800000	2.0875	0.015934	33702834	33.94113
4900000	2.128125	0.018222	34337544	166.6613
5000000	2.171875	0.027951	34967390	166.6613