Semester Project

Computing the prime-counting function

Jean-Luc Portner

Computation in Algebra and Number Theory
taught by
David Alexander Loeffler

Department of Mathematics
ETH Zürich
July 2022

1 Introduction

Prime numbers have fascinated mathematicians for centuries and counting them arises as a natural question. For the longest time, the only method to find the number of primes smaller than x, denoted by $\pi(x)$, was to calculate all primes up to x and count them. The fastest known method to calculate $\pi(x)$ was the sieve of Eratosthenes which will be described in section 2.

At the beginning of the 19th century, Legendre discovered a method for counting primes without having to list them all. His method will be described in section 3. This method, however, still had the downside of needing to calculate many additional terms making it unpractical.

The first efficient algorithm was described by the astronomer E. D. F. Meissel at the end of the 19th century. His method made the calculation more practical by drastically reducing the number of terms that needed calculating. Meissel managed to calculate $\pi(10^8)$ correctly and $\pi(10^9)$ with an error of 56. Subsequently many suggested improvements to Meissel's method although no one carried out the calculations.

In the middle of the 20th century, with the rise of digital computers, D. H. Lehmer extended Meissel's method and simplified it. He implemented it on an IBM 701 and managed to calculate $\pi(10^{10})$ (with an error of 1). His method was further drastically improved by Lagarias, Miller and Odlyzko and further improvements to this method were made by Deléglise, Rivat and Gourdon. Lehmer's original idea and further improvements will be described in section 4.

Gourdon's final method is to this day the best-known way to calculate $\pi(x)$. The leading implementation is the program "primecount" by Kim Walish and David Bough. All recent advancements in calculating $\pi(x)$ have been made with this implementation and the current record sits at 10^{29} giving a value for $\pi(x)$ of 1520698109714272166094258063.

Before we can discuss all these different methods however, we have to introduce a framework to analyze their asymptotic complexity.

1.1 Asymptotic complexity analysis and the RAM

To analyze the asymptotic complexity of an algorithm, we need to decide on a computational model on which the algorithm runs. The classical model of a Turing machine (TM) is not practical in our analysis as the time for a random read or write i.e. retrieving or writing values at random places on the tape scales with the total space used. That is the read/write head has to be moved from the current position to the position where the desired value should be read from/written to and back. Thus we consider the model of a *random access machine*, short RAM. The difference from a Turing machine is that a RAM does not have a sequential tape but uses an unlimited amount of registers that can be addressed by integers. Moreover, computations with these addresses are allowed. With this, every memory location can be accessed in constant time. This model also represents modern-day computers better as their memory supports near-constant read and write times.

In our discussions, the need for a RAM arises in the sieving procedures as sieving cannot be implemented efficiently on a Turing machine or even on a multitape Turing machine.

As is customary in analyzing the asymptotic complexity, we are not interested in the exact number of operations but just in their order. Therefore we use the big O-notation which determines the complexity up to a constant factor and is defined as follows:

Definition 1.1. Let f and g be two functions.

1. We say $f = O(g)$ if and only if

$$\exists M > 0, c > 0 \quad \text{such that} \quad |f(s)| \leq c \cdot |g(s)| \; \forall s \geq M.$$

2. Moreover we write $f = \Omega(g)$ if and only if $g = O(f)$.
3. If $f = O(g)$ and $g = O(f)$ then we write $f = \Theta(g)$.

With this defined, let n be the input size of an algorithm. Then, we can describe its asymptotic complexity depending on n. That is, we can say that the algorithm runs in time $O(n)$ if for an input of size n it needs $c \cdot n$ many operations to terminate (where c is a constant). The same can also be said about the space complexity i.e. the amount of memory the algorithm uses at any time during its computation.

2 The sieve of Eratosthenes

The sieve of Eratosthenes was first described in the work of Nicomedes (280-210 BC) entitled "Introduction to Arithmetic" and is probably the best-known method for finding primes. In this section, we follow the exposition from Nathanson in [8].

It is based on the following observation: Let $n \in \mathbb{N}$ be a composite number. Then there exist $d, d' \in \{1, \ldots, n\}$ with $d \leq d'$ such that $d \cdot d' = n$. Notice that, if $d > \sqrt{n}$ then

$$n = d \cdot d' > \sqrt{n} \cdot \sqrt{n} = n$$

which leads to a contradiction. Thus every composite number has a divisor $d \leq \sqrt{n}$ and in particular, every composite number is divisible by a prime $p \leq \sqrt{n}$.

To find all primes up to x we have the following algorithm:

Algorithm 1 Sieve of Eratosthenes

1: Write down all numbers from 1 to x.
2: Cross out 1.
3: Let d be the smallest number on the list whose multiples have not been eliminated already.
4: **if** $d > \sqrt{x}$ **then**
5: STOP
6: **else**
7: Cross out all multiples of $d \geq d^2$ and goto 3.
8: **end if**

The remaining non-crossed-out numbers are then the prime numbers up to x. Notice that it is enough to cross out all multiples $\geq d^2$ as all lower multiples have already been crossed out by a prior iteration.

Remark 2.1. If, however, we cross out all multiples $\geq d$ and keep track of how often a number has been crossed out, then we can also get the number of distinct prime factors every number has, which can be used to compute the Möbius function.

Trivially the algorithm terminates after maximally $x\sqrt{x}$ steps and thus lies in $O(x\sqrt{x})$. The complexity bound on the sieve of Eratosthenes can be improved as follows: We assume that crossing out a number can be done in $O(1)$.

Notice that the d's in our algorithm are exactly the prime numbers. Further note that crossing out all multiples of d then takes exactly $\frac{x}{d}$ many steps. If p is the biggest prime $\leq \sqrt{x}$ then the algorithm loops

$$\frac{x}{2} + \frac{x}{3} + \frac{x}{5} + \ldots + \frac{x}{p} = x \sum_{\substack{p' \leq \sqrt{x} \\ p' \text{ prime}}} \frac{1}{p'}$$

times.

Using the fact that the reciprocal sum of primes grows with $O(\log\log x)$, proved by Euler in 1737, and noticing that writing the numbers 1 to x can be done in time $O(x)$ we get the following theorem:

Theorem 2.2. *Finding all primes up to x can be done in time*

$$O(x \log\log x).$$

Remark 2.3. Importantly, the only arithmetic operation needed by the algorithm is addition. Hence, in reality the sieve is quicker than some other methods which have a better asymptotic complexity but also use multiplication which, at this time, has a bitwise complexity of at least $n\log(n)$ (see [4]) and can thus be slower.

Often we will also use the following result that $\log(x) \in O(x^\varepsilon)$ for $\varepsilon > 0$ which follows directly from l'Hopital's rule:

$$\lim_{x\to\infty} \frac{\log(x)}{x^\varepsilon} = \lim_{x\to\infty} \frac{x^{-1}}{\varepsilon x^{\varepsilon-1}} = \lim_{x\to\infty} \frac{1}{\varepsilon x^\varepsilon} = 0.$$

Thus sieving can be done in time $O(x^{1+\varepsilon})$.

More recently sieving algorithms have been found which run in sublinear time. More precisely Paul Pritchard found a sieving algorithm in [9] with asymptotic complexity of $\Theta(\frac{n}{\log\log n})$.

3 Legendre's method

In 1808, A. M. Legendre expanded on Eratosthenes' sieve and gave a more analytic method which we will describe in this section. We follow a presentation of this method from Giblin in [2].

The basic idea is to calculate the number of primes less than x by using the primes less than $\sqrt{x}$. Let us denote by X the set of integers 1 to $\lfloor x \rfloor$ and let $p_1, p_2, \ldots$ be the primes $\leq \lfloor\sqrt{x}\rfloor$. Moreover, let C_i be the set of multiples of $p_i \leq x$. Notice that the primes between $\sqrt{x}$ and x are exactly the numbers not belonging to any of the C_i. Furthermore, the sets C_i overlap and the intersection of k sets are exactly the numbers that have these k primes as prime factors.

We denote the cardinality of these intersections as follows

$$N(i_1, i_2, \ldots, i_k) = |(C_{i_1} \cap C_{i_2} \cap \ldots \cap C_{i_k}|.$$

The value of N is precisely the number of elements of X divisible by the product $p_{i_1}p_{i_2}\ldots p_{i_k}$ as the p_i are distinct primes. We therefore have

$$N(i_1, i_2, \ldots, i_k) = \left\lfloor \frac{x}{p_{i_1}p_{i_2}\ldots p_{i_k}} \right\rfloor.$$

For $k \geq 1$ we denote by S_k the sum $\sum N(i_1, \ldots, i_k)$ where the summation is over all tuples $i_1 < i_2 < \ldots < i_k$ for which the corresponding primes are ≥ 2 and $\leq \lfloor\sqrt{x}\rfloor$ Moreover, define $S_0 = |X|$.

Notice that N will be zero whenever the product in the denominator exceeds x i.e. whenever the product of the first k primes exceeds x. Thus all but finitely many N are zero. We denote the largest value of k for which N does not vanish by K.

The key observation is the following proposition:

Proposition 3.1 (Principle of inclusion-exclusion)**.** *The number of elements of X that do not lie in any of the C_i is*

$$S_0 - S_1 + S_2 - \ldots + (-1)^K S_K.$$

Proof. Let $z \in \mathbb{N}$ be such that z is contained in n sets C_i. Then z is counted once in S_0 and n times in S_1 - once for every C_i. In S_2 it is counted every time two of the C_i which contain z are chosen giving a total of $\binom{n}{2}$. In S_3 it is similarly counted $\binom{n}{3}$ times and so on. Thus in the total alternating sum of the S_i z occurs

$$1 - \binom{n}{1} + \binom{n}{2} - \ldots + (-1)^n \binom{n}{n} = \sum_{k=0}^{n} (-1)^k \binom{n}{k}$$

times. For $n \geq 1$ this evaluates to 0 as can be seen by using the binomial expansion

$$0 = (1-1)^n = \sum_{k=0}^{n} \binom{n}{k} 1^{n-k} (-1)^k.$$

If however z is contained in none of the C_i then it is counted only once, in S_0. Thus the result follows. □

As aforementioned the elements of X which lie in no C_i are the numbers not divisible by any prime $\leq \lfloor \sqrt{x} \rfloor$. That is the prime numbers between $\lfloor \sqrt{x} \rfloor$ and $\lfloor x \rfloor$. We, therefore, get Legendre's formula:

Theorem 3.2.

$$\pi(x) = \pi(\sqrt{x}) + \sum_{k=0}^{K} (-1)^k S_k.$$

The following example illustrates this method for $x = 28$:

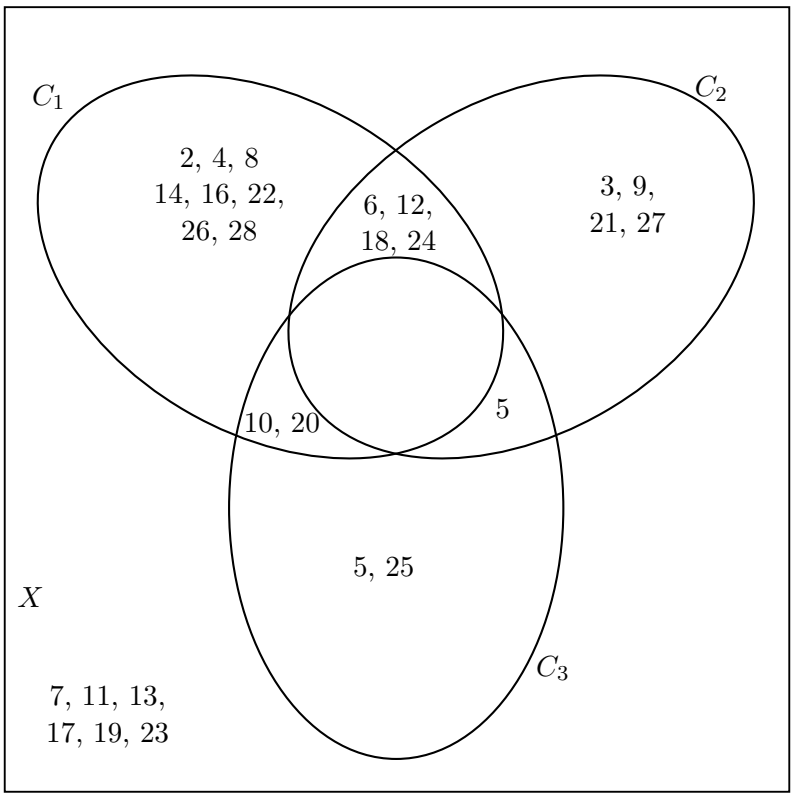

Figure 1: Illustration of the sets participating in Legendre's method.

Example 3.3. We show the workings of Legendre's method for $x = 28$. The primes $\leq \sqrt{x}$ are $2, 3, 5$. Thus the sets C_1, C_2, C_3 are given as shown in Figure 1. Notice that no number lies in the intersection of the three sets as $2 \cdot 3 \cdot 5$ is > 28. To find the amount of numbers lying outside

of $C_1 \cup C_2 \cup C_3$ we subtract their cardinalities from X. However as Figure 1 illustrates the sets $C_1 \cap C_2, C_1 \cap C_3, C_2 \cap C_3$ will be subtracted twice. Thus we have to add their cardinalities to get the desired result. If we express this in the terms given above it is exactly $S_0 - S_1 + S_2$. Carrying out the calculation we find that $\pi(28) = \pi(\sqrt{28}) + S_0 - S_1 + S_2 = 9$

4 The Meissel-Lehmer method

At the beginning of this chapter, we introduce general notation and formulas for algorithms of Meissel-Lehmer type. In the latter sections, we then present the different improvements made over time. We follow the presentations from Lagarias, Miller and Odlyzko in [5] as well as from Lehmer in [7].

4.1 General Meissel-Lehmer algorithms

Definition 4.1. Let us denote the primes $2, 3, 5, \ldots$ numbered in increasing order by $p_1, p_2, p_3, \ldots$.
For $a \geq 1$ let

$$\phi(x, a) = |\{n \leq x \mid p \mid n \Rightarrow p > p_a\}|$$

that is the *partial sieve function* counting the numbers $\leq x$ with no prime factors $\leq p_a$. Moreover, we define

$$P_k(x, a) = \left| \left\{ n \leq x \mid n = \prod_{j=1}^{k} p_{m_j}, m_j > a \text{ for } 1 \leq j \leq k \right\} \right|$$

that is the *k-th partial sieve function* counting the numbers $\leq x$ with exactly k prime factors $\geq p_a$. We extend this definition to $P_0(x, a) = 1$. Then, as $\phi(x, a)$ also counts 1 we get

$$\phi(x, a) = \sum_{k=0}^{\infty} P_k(x, a)$$

where the sum has only finitely many non-zero terms as every number $\leq x$ has a finite number of prime factors. Importantly, the numbers with only one prime factor are the primes. Thus $P_1(x, a) = \pi(x) - a$ and we get the following formula:

$$\pi(x) = \phi(x, a) + a - 1 - \sum_{k \geq 2} P_k(x, a).$$

If we take p to be the biggest prime $\leq x^{1/j}$ i.e. $p_{\pi(x^{1/j})}$. Then any $n \leq x$ can have at most j prime factors bigger than p. We conclude that $P_k(x, \pi(x^{1/j})) = 0$ for all $k \geq j$.
For $a = \pi(x^{1/j})$ we get the formulas of Meissel-Lehmer type:

$$\pi(x) = \phi(x, a) + a - 1 + \sum_{2 \leq k < j} P_k(x, a).$$

Different choices for j result in the different methods developed over time. A value of $j = 2$ for example yields Legendre's method:

$$\pi(x) = \phi(x, a) - a + 1.$$

For $j = 3$ we obtain Meissel's original formula which we will analyze in depth in 4.3.

A general method of Meissel-Lehmer type can thus be split into multiple parts: In the first step, one has to calculate the value of $\phi(x, a)$. For this the following recurrence is essential:

Lemma 4.2.

$$\phi(x,a) = \phi(x,a-1) - \phi\left(\frac{x}{p_a}, a-1\right).$$

Proof. We can rewrite the set $\{y \leq x \mid p \mid x \Rightarrow p > p_a\}$ whose cardinality equals $\phi(x,a)$ as follows:

$$\begin{aligned}\{y \leq x \mid p \mid x \Rightarrow p > p_a\} &= \{y \leq x \mid p \mid y \Rightarrow p > p_{a-1} \text{ and } p_a \nmid y\} \\ &= \{y \leq x \mid p \mid y \Rightarrow p > p_{a-1}\} \setminus \{p_a y \leq x \mid p \mid y \Rightarrow p > p_{a-1}\}.\end{aligned}$$

Notice that when taking absolutes the sets in the last expression equal $\phi(x,a-1)$ and $\phi(\frac{x}{p_a}, a-1)$. Thus we obtain the desired result. □

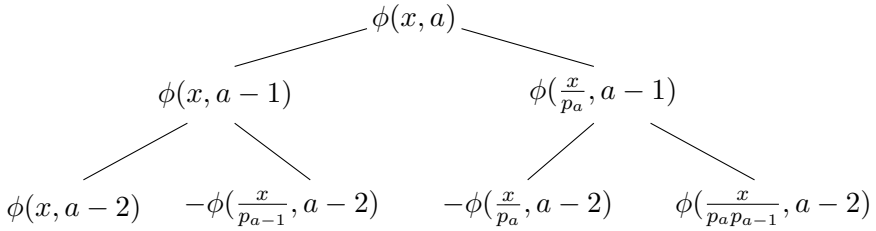

Figure 2: Binary tree from recursion of $\phi(x,a)$

Through the repeated application of this recurrence, one can build a structure similar to a binary tree as seen in Figure 2. Each node of the tree is represented by a term $\pm\phi(\frac{x}{n}, b)$ for some n and b. Each parent node has two children and the sum of each "level" of the tree is equal to $\phi(x,a)$. If some branches are cut early i.e. the recurrence is not applied anymore to that branch, then the sum over all the leaves is equal to $\phi(x,a)$. Notice also that every node in the tree can be uniquely described by the tuple (n,b) where $n = \prod_{k=1}^{r} p_{a_k}$ with $a \geq a_1 > \ldots > a_r \geq b+1$. That is the pair (n,b) is associated with the term $(-1)^r \phi(x/n, b)$.

The tricky part when building this tree is to decide when to stop applying the recurrence to a node and calculate its value. For this, different methods of Meissel-Lehmer type use different rules, called truncation rules, to optimize the number of leaves. This part of the calculation of $\pi(x)$ is by far the most complex and time-consuming. Thus nearly all advancements have been made here.

The second part of the calculation is to find the values of the $P_k(x,a)$. This is in general much simpler than the previous step as simple explicit formulas exist. We will show them for $k = 2$ and $k = 3$.

To calculate P_2 we use the following which holds whenever $a \leq \pi(x^{1/2})$:

$$\begin{aligned}P_2(x,a) &= |\{n \mid n \leq x, n = p_b p_c \text{ with } a < j \leq k\}| \\ &= \sum_{j=a+1}^{\pi(x^{1/2})} \left| \left\{ n \mid n \leq x, n = p_j p_k \text{ with } j \leq k \leq \pi\left(\frac{x}{p_j}\right) \right\} \right| \\ &= \sum_{j=a+1}^{\pi(x^{1/2})} \left(\pi\left(\frac{x}{p_j}\right) - j + 1 \right) = \binom{a}{2} - \binom{\pi(x^{1/2})}{2} + \sum_{j=a+1}^{\pi(x^{1/2})} \pi\left(\frac{x}{p_j}\right)\end{aligned}$$

where the second equality follows as for $n = p_j p_k$ we have $x \geq n = p_j p_k$ thus implying $p_k \leq \frac{x}{p_j}$ which translates to the inequality for the indices. The third inequality follow from enumerating and the fourth from Gauss' summation.

For P_3 if $a \leq \pi(x^{1/3})$ then the following holds:

$$\begin{aligned} P_3(x,a) &= |\{n \mid n \leq x, n = p_j p_k p_l \text{ with } a < j \leq k \leq l\}| \\ &= \sum_{j=a+1}^{\pi(x^{1/3})} \left| \left\{ n \mid n \leq \frac{x}{p_j}, n = p_k p_l \text{ with } j \leq k \leq l \right\} \right| \\ &= \sum_{j=a+1}^{\pi(x^{1/3})} P_2\left(\frac{x}{p_j}, a\right) = \sum_{j=a+1}^{\pi(x^{1/3})} \sum_{k=j}^{b_i} \left(\pi\left(\frac{x}{p_j p_k}\right) - k + 1 \right) \end{aligned}$$

with $b_i = \pi(\sqrt{x/p_i})$. The second equality follows as if $p_j p_k p_l \leq x$ and $p_j \leq p_k \leq p_l$ then clearly $p_j \leq x^{1/3}$ which translates to the inequalities for the indices. Moreover, we divided by p_j. The third is just using the definition of P_2 and the fourth uses the formula we deduced above for P_2.

We are now ready to present and compare different Meissel-Lehmer methods.

4.2 Lehmer's original method

Lehmer was the first to implement the method on a computer. In his method, he mostly chose a value of $a = \pi(x^{1/3})$ thus $P_k(x,a) = 0$ for $k \geq 3$ however some calculations were also carried out for $a = \pi(x^{1/4})$ which adds the term $P_3(x,a)$. For the computation of P_3 he used a short precalculated table of $\pi(y)$ for small values of y which he stored in memory. In contrast for P_2 no such table is viable as the values of y can get quite big. Thus a modified version of Eratosthenes' sieve was used whose values were stored on magnetic tape.

The mathematically interesting part of the calculation however lies in finding the value of $\phi(x,a)$. For this Lehmer suggested the following truncation rule:

Definition 4.3 (Truncation rule L)**.** A node $\pm\phi(x/n, b)$ will not be split if one of the following holds

(i) $x/n < p_b$
(ii) $b = c(x)$ for a very slowly growing function $c(x)$.

Lehmer originally chose $c = 5$ for his computations.

He computed the leaves using the following formulas:

Lemma 4.4. *When applying the truncation rule L the leaves can be calculated as follows:*

- *For leaves of type (i) it holds that $\phi(y,b) = 1$ if $y < p_b$.*
- *For leaves of type (ii) we have*

$$\phi(y,b) = \left\lfloor \frac{y}{Q} \right\rfloor \phi(Q,b) + \phi(y - \left\lfloor \frac{y}{Q} \right\rfloor Q, b)$$

where $b = c(x)$, $Q = \prod_{i \leq c(x)} p_i$ and the values of $\{\phi(y,b) \mid 1 \leq y \leq Q\}$ have been precomputed.

Proof. We start by proving the first formula: Let $1 < x \leq y$. Then as $y \leq p_b$ the prime factors of x are also smaller than p_b. Thus no x satisfies $p \mid x \Rightarrow p > p_b$ for a prime number p. Therefore $|\{x \leq y \mid p \mid x \Rightarrow p_b\}| = 1$ as only 1 is contained in this set. This yields the desired result of $\phi(y,b) = 1$.

The proof of the second formula goes as follows: Consider $x \in \mathbb{N}$ such that $Q\lambda \leq x \leq Q(\lambda+1)$. Then we can write x as $Q\lambda + r$ where r is not divisible by Q. Observe that as Q is divisible by p_i for $1 \leq i \leq b$ we have that $p_i \mid x$ if and only if p_i divides r.

Let us write $y = Q \cdot \lambda + r$. Then $\lambda = \left\lfloor \frac{y}{Q} \right\rfloor$ and we can compute the following:

$$\begin{aligned}
|\{x \leq y \mid p \mid x \Rightarrow p > p_b\}| &= \sum_{\mu=0}^{\lambda} |\{x \in \mathbb{N} \mid Q\mu \leq x < Q(\mu+1), p \mid x \Rightarrow p > p_b\}| \\
&\quad + \sum_{\mu=0}^{\lambda} |\{x \in \mathbb{N} \mid Q\mu \leq x \leq r, p \mid x \Rightarrow p > p_b\}| \\
&= \sum_{\mu=0}^{\lambda} |\{x < Q \mid p \mid x \Rightarrow p > p_b\}| + |\{x \leq r \mid p \mid x \Rightarrow p > p_b\}| \\
&= \lambda \cdot \phi(Q, b) + \phi(r, b).
\end{aligned}$$

where in the first equality we just dissected the set and used that the resulting sets are disjoint. In the second equality, we used the above observation. And in the final equality, we used the definition of ϕ as well as that Q is divisible by $p < p_b$ and therefore the strict inequality $x < Q$ can be changed to $x \leq Q$. By plugging in the values of λ and r in dependence of y and Q we get the desired result. □

For an exact description of the implementation of this truncation rule and the calculation, the reader can consult [7] or inspect the python implementation in section A.4 of the appendix. The big disadvantage of Lehmer's truncation rule is that it is rather space inefficient e.g. the calculation of $\phi(10^{10}, 65)$ leads to more than 3 million leaves that need to be calculated. Asymptotically the number of nodes in the tree of ϕ is roughly $\frac{1}{24}a^4 = \Omega(x/\log^4(x))$. Lehmer himself states in [7] that "this is a good example of how one can substitute time for space with a high-speed computer".

4.3 The extended Meissel-Lehmer method

This method was developed by Lagarias, Miller and Odlyzko. Its big advantage over Lehmer's method is its drastically improved time and space efficiency. To achieve this they chose a value of $a = \pi(x^{1/3})$ and split the computation of P_2 into batches of size $x^{2/3}$. The big change however was in the truncation rule they used for calculating $\phi(x, a)$ which significantly reduced the number of leaves:

Definition 4.5 (Truncation rule T)**.** Do not split a node labeled $\pm\phi(x/n, b)$ if either of the following holds:

(i) $b = 0$ and $n \leq x^{1/3}$ or
(ii) $n > x^{1/3}$.

Leaves of type (i) are called *ordinary leaves* and of (ii) are called *special leaves.*

The following lemma shows the great reduction in the number of leaves that can be achieved with this new rule:

Lemma 4.6. *When using truncation rule T to calculate $\phi(x, a)$ with $a = \pi(x^{1/3})$ the resulting binary tree has at most $x^{1/3}$ ordinary leaves and at most $x^{2/3}$ special leaves.*

Proof. We first prove the bound on the ordinary leaves. For this, we notice that no two leaves (n, b) have the same value of n. As else if (n, d) and (n, b) are two nodes with $d \geq b$. Then there exists a path through the tree given by $(n, d-1), \ldots, (n, b+1), (n, b)$. Thus (n, d) cannot be a leaf and the bound follows as $n \leq x^{1/3}$.

To bound the special leaves, observe that every special leaf (n, b) has a father node $(n^*, b+1)$ with $n^* = n^* p_{b+1}$. Moreover, by the definition of the special leaves, $n > x^{1/3} \geq n^*$ as $(n^*, b+1)$ is not a special leaf. We thus have at most $x^{1/3}$ many choices for n^* and maximally $a = \pi(x^{1/3}) \leq x^{1/3}$ many choices for p_{b+1}. Hence in total, there are at most $x^{2/3}$ possibilities for n. This gives the bound on the special leaves. □

To compute the contribution of the leaves a partial sieving process is applied to the interval $[1, \lfloor x^{2/3} \rfloor]$. This is done on successive subintervals of length $x^{1/3}$ to reduce space usage. An in-depth description of the precise algorithm can be found in [5] and a python implementation can be found in section A.5 of the appendix. In [5] it is also shown that the algorithm has the following asymptotic complexity:

Theorem 4.7. *The extended Meissel-Lehmer method can be implemented to calculate $\pi(x)$ using at most $O(x^{2/3+\varepsilon})$ arithmetic operations and using at most $O(x^{1/3+\varepsilon})$ storage locations on a RAM. All integers stored during the computation are of length at most $\lfloor \log_2(x) \rfloor + 1$ bits.*

Remark 4.8. Lagarias, Miller and Odlyzko also presented an algorithm that works on M parallel processors for $M < x^{1/3}$ and reduces the arithmetic operations per processor to $O(M^{-1}x^{2/3+\varepsilon})$ while using $O(x^{1/3+\varepsilon})$ storage locations per processor.

4.4 Further improvements

In 1994 Deléglise and Rivat improved on the previous method in [1]. They kept practically the same algorithm for calculating P_2 and used the same truncation rule as Lagarias, Miller and Odlyzko. However, they significantly improved the implementation of the latter by grouping together the leaves more efficiently and splitting the calculation of the most complicated parts into smaller subproblems. The improvements are summarized by the following theorem:

Theorem 4.9. *The algorithm described by Deléglise and Rivat takes $O(x^{1/3} \log^3(x) \log\log(x))$ space and has a time complexity of $O(\frac{x^{2/3}}{\log_2(x)})$.*

A proof of this theorem can be found in [1]. With their new algorithm, Deléglise and Rivat managed to compute $\pi(x)$ up to 10^{18}.

Finally, in 2001, Xavier Gourdon presented further improvements in [3]. Through optimizing the calculation of the subproblems used by Deléglise and Rivat he managed to reduce some of the constant factors as well as reduce the space complexity to $O((x/y)^{1/2})$ instead of the $O(y)$ achieved by Deléglise and Rivat where $y = x^{1/3} \log^3(x) \log\log(x)$. Arguably his most important contribution was that he managed to improve Lagarias, Miller and Odlyzko's parallel algorithm significantly. Through his advancements, only a small exchange of memory between the processes is needed after doing a relatively cheap precomputation. This allows for efficient distributed calculations.

5 Final remarks

In this final section, we present the different asymptotic complexities of the methods described as well as some computational results.

In Table 1 the time as well as space complexities of the discussed methods are shown and proofs for these can be found in the respective papers.

Method	Time	Space
Eratosthenes	$O(x \log\log x)$	$O(x)$
Legendre	$O(x)$	$O(x^{1/2})$
Lehmer	$O(x/\log^4 x)$	$O(x^{1/3}/\log x)$
Lagarias-Miller-Odlyzko	$O(x^{2/3+\varepsilon})$	$O(x^{1/3+\varepsilon})$
Deléglise-Rivat	$O(x^{2/3}/\log^2 x)$	$O(x^{1/3}\log^3 x \log\log x)$
Lagarias-Odlyzko	$O(x^{1/2+\varepsilon})$	$O(x^{1/4+\varepsilon})$

Table 1: Comparison of the asymptotic complexities

The method mentioned in the last row has been described by Lagarias and Odklyzko in 1987 in [6] and uses a completely different approach to the other versions. Here numerical integration of specific integral transforms of the Riemann ζ-function is being used to compute $\pi(x)$. Even though this method has superior asymptotic complexity it is not being used in practical applications as the implied constants are likely very large and therefore not competitive with the other methods.

In the appendix, a python implementation of the four described methods can be found. One should note however that these are quite slow due to their implementation in python and not in a fast language as C++. Therefore, for practical calculations, Kim Walish's primecount[1] should be used. Nonetheless, the implementation illustrates well the different workings of the methods and follows as close as possible the original descriptions of the different authors.

We conclude with Table 2 which displays the results of $\pi(x)$ as well as the functions $P_2(x,a)$ and $\phi(x,a)$ for the different powers of 10 and for $a = \pi(x^{1/3})$.

x	$P_2(x,a)$	$\phi(x,a)$	$\pi(x)$
10	1	5	4
10^2	9	33	25
10^3	63	228	168
10^4	489	1711	1229
10^5	4625	14204	9592
10^6	42286	120760	78498
10^7	374867	1039400	664579
10^8	3349453	9110819	5761455
10^9	30667735	81515102	50847534
10^{10}	279167372	734219559	455052511
10^{11}	2571194450	6689248638	4118054813
10^{12}	23729370364	61337281154	37607912018
10^{13}	566584397965	220518863542	346065536839

Table 2: Values of $\pi(x)$

[1] See `github.com/kimwalisch/primecount`.

References

[1] Marc Deléglise and Joël Rivat. "Computing $\pi(x)$: the Meissel, Lehmer, Lagarias, Miller, Odlyzko method". In: *Mathematics of Computation* 65.213 (1996), pp. 235–245.

[2] Peter Giblin. *Primes and programming: an introduction to number theory with computing.* Cambridge: Cambridge University Press, 1993.

[3] Xavier Gourdon. "Computing $\pi(x)$: improvements to the Meissel, Lehmer, Lagarias, Miller, Odlyzko, Deléglise and Rivat method". preprint. `numbers.computation.free.fr/Constants/Primes/Pix/piNalgorithm.ps`. 2001.

[4] David Harvey and Joris van der Hoeven. "Integer multiplication in time $O(n \log n)$". In: *Annals of Mathematics* 193.2 (2021), pp. 563–617.

[5] Jeffrey C Lagarias, Victor S Miller, and Andrew M Odlyzko. "Computing $\pi(x)$: the Meissel-Lehmer method". In: *Mathematics of Computation* 44.170 (1985), pp. 537–560.

[6] Jeffrey C Lagarias and Andrew M Odlyzko. "Computing $\pi(x)$: An analytic method". In: *Journal of Algorithms* 8.2 (1987), pp. 173–191.

[7] Derrick H Lehmer. "On the exact number of primes less than a given limit". In: *Illinois Journal of Mathematics* 3.3 (1959), pp. 381–388.

[8] Melvyn Bernard Nathanson. *Elementary methods in number theory.* Graduate texts in mathematics 195. New York: Springer, 2000.

[9] Paul Pritchard. "A sublinear additive sieve for finding prime number". In: *Communications of the ACM* 24.1 (1981), pp. 18–23.

A Python implementation

To run the ensuing code python 3 as well as the package `numpy` is needed. The code consists of five files and can also be found on GitHub under: `github.com/jlportner/CompProject/tree/main/PrimeCounting`.

In `eratosthenes.py` a version of Eratosthenes' sieve is implemented. This file is also necessary to run the other methods. In `legendre.py` Legendre's method has been implemented. The file `lehmer.py` contains an implementation of Lehmer's method. In the file `lmo.py` the extended Meissel-Lehmer method from Lagarias, Miller and Odlyzko has been implemented. For Lehmer's method as well as the extended Meissel-Lehmer method one can either call `lehmer` / `lmoMethod` to calculate $\pi(x)$ or one can also just compute P_2 or ϕ by calling the respective methods. Finally, in `main.py` a small working example that asks for an integer x and then computes $\pi(10^x)$ with all the different aforementioned methods has been implemented. This also measures the time each method took and prints it out.

A.1 main.py

```
from eratosthenes import eratosthenesSieve
from legendre import legendresMethod
from lmo import lmoMethod
from lehmer import lehmer
from timeit import default_timer as timer

while True:
    print("Enter x to calculate pi(10^x): ")
    a = int(input())
    n = 10**(a)

    print("Calculating number of primes up to " + str(n))

    if a >= 9:
        print("Running Erastothenes Sieve will take longer than
    1min, skipping it ...")
    else:
        print("Running Erastothenes Sieve ...")
        s = timer()
        print(len(eratosthenesSieve(n)))
        e = timer()
        print("Took: " + str(e-s) + "s")

    if a >= 8:
        print("Running Legendre will take longer than 1min, skipping
    it ...")
    else:
        print("Running Legendres Method ...")
        s = timer()
        print(legendresMethod(n))
        e = timer()
        print("Took: " + str(e-s) + "s")
```

```
    print("Running␣Lehmer-Method␣...")
    s = timer()
    print(lehmer(n))
    e = timer()
    print("Took:␣" + str(e - s) + "s")

    print("Running␣LMO-Method␣...")
    s = timer()
    print(lmoMethod(n))
    e = timer()
    print("Took:␣" + str(e - s) + "s")
```

A.2 eratosthenes.py

```
import numpy as np

def eratosthenesSieve(n):
    if n <= 1:
        return np.array([])
    n = int(np.floor(n))
    N = np.ones(n+1,dtype=bool)
    #Identify the elements of this array with the numbers where the
    ith cell matches with the ith number
    N[1] = N[0] = 0
    sqn = int(np.ceil(np.sqrt(n)))
    for p in range(2,sqn+1):
        if N[p] == 0:
            continue
        else:
            k = p**2
            while k <= n:
                N[k] = 0
                k += p

    return np.arange(n+1)[N]
```

A.3 legendre.py

```
import numpy as np
from eratosthenes import eratosthenesSieve

def legendresMethod(n):
    P = eratosthenesSieve(np.sqrt(n)).tolist()
    pi12 = len(P)
    pi = pi12 + int(n) - 1
    k = 1
    while k <= len(P) and np.prod(P[:k]) < n:
        Sk = 0

        def legendreRec(N,prod,length):
```

```
            nonlocal Sk
            if length == 1:
                for x in N:
                    if prod * x > n:
                        break
                    Sk += int(n / (prod*x))
            else:
                for i, x in enumerate(N):
                    if prod * x > n:
                        break
                    else:
                        legendreRec(N[(i+1):], prod * x, length-1)

        legendreRec(P,1,k)
        pi += (-1) ** k * Sk
        k += 1

    return pi
```

A.4 lehmer.py

```
import numpy as np
from eratosthenes import *

def preCalcPhi(n, a,P=None):
    if P is None:
        P = eratosthenesSieve(n).tolist()

    preCalc = np.vstack((np.arange(n+1),np.zeros((a,n+1))))
    for i in range(1,a+1):
        preCalc[i] = preCalc[i-1] - preCalc[i-1][(np.arange(n+1) /
   P[i-1]).astype("int")]

    return preCalc

epsilon = 10 ** -10
def lehmer(x, alternative=False):
    N = int(x ** (1 / 3) + epsilon)
    P = eratosthenesSieve(N)
    valPhi = phi(x,P)
    valP2 = P2(x)
    piVal = valPhi - valP2 - 1 + len(P)
    return int(piVal)

def phi(x,P=None):
    if P is None:
        N = int(x ** (1 / 3) + epsilon)
        P = eratosthenesSieve(N)
    k = int(np.min((len(P),5)))
    modVal = int(np.prod(P[:k]))
    phiCache = preCalcPhi(modVal, k,P)
```

```
    phiVal = phiRec(x, np.max((len(P),0)), P, k, phiCache,modVal)
    return phiVal

def phiRec(x, l, P, k, phiCache, modVal):
    if l <= k:
        q, r = np.divmod(x, modVal)
        return phiCache[l,modVal] * q + phiCache[l,int(r)]
    elif x < P[l-1]:
        return 1
    else:
        return phiRec(x, l - 1, P, k, phiCache, modVal) - phiRec(x /
   P[l-1], l - 1, P, k, phiCache, modVal)

def P2(x):
    n = x ** (1 / 3) + epsilon
    n12 = x ** (1 / 2) + epsilon
    P = eratosthenesSieve(x/n)
    b = np.searchsorted(P,n12,side="left")
    a = np.searchsorted(P,n,side="right")
    valP2 = 0
    curPi = b
    curP = P[np.searchsorted(P,n12,side="right"):]
    for i in range(b-1,a-1,-1):
        val = x/P[i]
        valIndex = np.searchsorted(curP,val,side="right")
        curPi += valIndex
        curP = curP[valIndex:]
        valP2 += curPi

    return valP2 - (b-a)*(b+a-1)/2
```

A.5 lmo.py

```
import numpy as np
from eratosthenes import *

def intersect2HalfOpen(I,J):
    a,b = I
    c,d = J
    if d <= a or b <= c:
        return -1
    else:
        return (max(a,c),min(b,d))

def intersect1HalfOpen(I,J):
    a,b = I
    c,d = J
    if d <= a or b < c:
        return -1
    else:
        return (max(a,c),min(b,d))
```

```
def sieveWithKnownPrimes(P,a,b):
    if b < a:
        return []
    a,b = int(a),int(b)
    length = b-a
    B = np.ones(b-a+1,dtype=bool)

    for p in P:
        start = int(p * (np.ceil(a/p))) - a
        k = start
        while k <= b-a:
            B[k] = 0
            k += p

    return np.arange(a,b+1)[B]

def calcPi(start,B,x):
    return start + len(B[np.argwhere(B <= x)])

def P2(n,P=None):
    n13 = n**(1/3)
    N = int(n13 + epsilon)
    n12 = n**(1/2)

    if P is None:
        P = eratosthenesSieve(N)
    P14 = P[np.argwhere(P <= n**(1/4))].flatten()
    pi13 = len(P)
    pi12 = 0
    pi = pi13
    S = 0
    for j in range(2,int(n**(2/3)/N)+2):
        a,b = (j-1)*N+1, min(j*N,int(n**(2/3)))

        B = sieveWithKnownPrimes(P,a,b)

        if a <= int(n12) <= b:
            pi12 = calcPi(pi,B,n12)

        I = intersect2HalfOpen((n/(b+1),n/a),(n13,n12))
        if I != -1:

            PI = sieveWithKnownPrimes(P14,int(I[0]+1),int(I[1]))
            if len(PI) > 0:
                PI = (n/PI).astype("int")
                L = np.vectorize(lambda x: calcPi(pi,B,x))
                S += np.sum(L(PI))
        pi += len(B)
```

```
    valP2 = pi13 * (pi13 -1)/2 - pi12 * (pi12 - 1)/2+ S
    return int(valP2)

def specialSieve(n):
    n = int(np.floor(n))
    N = np.ones(n+1,dtype=bool)
    f = np.zeros(n+1,dtype=int)
    mu = np.ones(n+1,dtype=int)

    #Identify the elements of this array with the numbers where the
    ith cell matches with the ith number
    N[1] = N[0] = 0
    sqn = int(np.ceil(np.sqrt(n)))
    for p in range(2,sqn+1):
        if N[p] == 0:
            continue
        else:
            mu[p] = -1
            f[p] = p
            k = 2*p
            while k <= n:
                N[k] = 0
                mu[k] *= -1
                if f[k] == 0:
                    f[k] = p
                k += p

    P = np.arange(n+1)[N]
    #primes bigger sqrt n
    Pover12 = P[P > sqn]
    for p in Pover12:
        mu[p] = -1
        f[p] = p
        k = p * 2
        while k <= n:
            mu[k] *= -1
            if f[k] == 0:
                f[k] = p
            k += p

    P12 = P[np.argwhere(P <= n ** (1 / 2))].flatten()
    for p in P12:
        k = p**2
        while k <= n:
            mu[k] = 0
            k += p**2

    return P, f[1:],mu[1:]

epsilon = 10**-10
```

```
def getBinaryExponents(x):
    exponents = []
    e = int(np.log2(x))
    while x != 0:
        q,r = np.divmod(x,2**e)
        if q == 1:
            exponents.append(e)
        x = r
        e -= 1
    return np.array(exponents)

def S2(x,F=None,P=None):
    N = int(x ** (1 / 3) + epsilon)
    if F is None or P is None:
        P, fSieve, muSieve = specialSieve(N)
        F = np.array((np.arange(1, N + 1), fSieve, muSieve)).T

    phi = np.zeros(len(P))
    exponentsN = getBinaryExponents(N)
    phiIndexN = []
    for e in exponentsN:
        be = 1 + np.sum(2 ** (exponentsN[exponentsN > e] - e))
        phiIndexN += [(e,be-1)]

    valS2 = 0
    for j in range(1, int(x ** (2 / 3) / N) + 2):
        c, d = (j-1) * N + 1, min(j * N, int(x ** (2 / 3)+epsilon))
        a = 2 ** (np.arange(int(np.log2(N))+1).reshape((-1, 1)) @
   np.ones(N+1).reshape((1, -1)))
        for i in range(1,int(np.log2(N))+1):
            jBound = int(N/2**i)
            a[i,jBound:] = -1
            a[i,jBound] = np.remainder(N,2**i)

        for b, p in enumerate(P):
            J = intersect1HalfOpen((x / ((d + 1) * p), x / (c * p)),
   (1, N))
            if J != -1:
                L, U = int(J[0]+1), int(J[1])
                FLU = F[L-1:U]
                for m, _, mu in FLU[(FLU[:, 1] > p) & (FLU[:, 2] !=
   0)]:
                    if m * p <= N:
                        continue
                    phiyb = phi[b]
                    y = int(x / (m * p) + epsilon)
                    l = y - (c - 1)
                    exponents = getBinaryExponents(l)
                    for e in exponents:
```

```
                        be = 1 + np.sum(2 ** (exponents[exponents >
    e] - e))
                        phiyb += a[e, be - 1]
                    valS2 -= mu * phiyb

            for index in phiIndexN:
                phi[b] += a[index]

            #k = l -1 to address indices properly
            start = int(p * (np.ceil(c / p))) - c
            k = start
            while k <= d - c:
                if a[0,k] == 1:
                    a[0, k] -= 1
                    for i in range(1, a.shape[0]):
                        lInd = ((k+1) + 2 ** i - 1) / (2 ** i) +
    epsilon
                        a[i, int(lInd)-1] -= 1
                k += p

    return int(valS2)

def S1(x,mu=None):
    N = int(x ** (1 / 3) + epsilon)
    if mu is None:
        _, _, muSieve = specialSieve(N)
    ks = np.arange(1, N + 1)
    du = (x / ks + epsilon).astype("int")
    valS1 = np.dot(mu, du)
    return int(valS1)

def phi(x):
    N = int(x**(1/3) + epsilon)
    P,fSieve,muSieve = specialSieve(N)
    F = np.array((np.arange(1, N + 1), fSieve, muSieve)).T
    valS1 = S1(x,muSieve)
    valS2 = S2(x,F,P)

    return valS1 + valS2

def lmoMethod(x):
    N = int(x ** (1 / 3) + epsilon)
    P, fSieve, muSieve = specialSieve(N)
    F = np.array((np.arange(1, N + 1), fSieve, muSieve)).T
    valS1 = S1(x, muSieve)
    valS2 = S2(x, F, P)
    valP2 = P2(x,P)
    pi13 = len(P)

    return valS1 + valS2 - valP2 + pi13 -1
```