Utrecht University

Faculty of Science
Department of Mathematics

MASTER THESIS

Efficient Computation of Chebyshev's Psi Function

Author:
Joël Ganesh

Studentnumber:
6524664

Supervisor:
Dr. Lola Thompson

Second reader:
Dr. Johan Commelin

March 21, 2024

Abstract

In number theory, Chebyshev's psi function (denoted by ψ) sums the logarithms of the largest prime powers below a given parameter. It is known that, asymptotically, $\psi(N)$ grows like N. In fact, this statement is equivalent to the Prime Number Theorem. However, precise estimates of values of ψ are relatively hard to obtain because of its relationship with primes, which are still somewhat mysterious. In this thesis, we present two algorithms for computing $\psi(N)$ using arguments based on recent approaches in [HT23] and [HKM23] for sums of related arithmetic functions. The former only considers elementary methods and sums certain arithmetic functions in roughly $O(N^{3/5})$ operations using roughly $O(N^{3/10})$ memory, while the latter also uses ideas stemming from basic Fourier theory and runs in roughly $O(N^{1/2})$ operations using roughly $O(N^{1/2})$ memory. We also discuss details regarding the implementation in C++. The implementation can be found on GitHub.

Contents

Chapter 1

Introduction

Number theoretic functions play a central role in analytic number theory. Here, we should note that a number theoretic function is a function from the positive integers (natural numbers) to the complex numbers whose values may depend on the prime factorization of its input. Some examples of number theoretic functions are:

- the prime indicator function,

$$\mathbb{1}_{\text{prime}}(n) = \begin{cases} 1 & \text{if } n \text{ is prime,} \\ 0 & \text{otherwise;} \end{cases}$$

- the *Möbius function*,

$$\mu(n) = \begin{cases} 0 & \text{if } p^2 \text{ divides } n \text{ for some prime } p, \\ (-1)^t & \text{otherwise, where } t \text{ is the number of distinct prime divisors of } n; \end{cases}$$

- the *Von Mangoldt function*,

$$\Lambda(n) = \begin{cases} \log p & \text{if } n \text{ is a proper power of a prime } p, \\ 0 & \text{otherwise;} \end{cases}$$

- *Euler's totient function*,

$$\varphi(n) = \#\big\{k \in \{1, 2, \ldots, n\} : \gcd(k, n) = 1\big\} = n \prod_{\substack{p \mid n \\ p \text{ prime}}} \left(1 - \frac{1}{p}\right).$$

In particular, partial sums of the previous examples have been studied in the past. For example, the Prime Number Theorem states that the prime counting function, defined via

$$\pi(N) = \sum_{n \le N} \mathbb{1}_{\text{prime}}(n) = \sum_{\substack{p \le N \\ p \text{ prime}}} 1,$$

is asymptotically equal to $N/\log N$ as $N \to \infty$, which is to say that

$$\lim_{N \to \infty} \frac{\pi(N)}{N/\log N} = 1.$$

In fact, this statement has been shown to be equivalent to other statements such as:

- $\psi(N) = \sum_{n\le N} \Lambda(n)$ being asymptotically equal to N, as $N \to \infty$;
- $M(N) = \sum_{n\le N} \mu(n)$ satisfying $M(N)/N \to 0$, as $N \to \infty$.

Proofs of these equivalences can be found in [Apo13, pp. 74–80, 91–101], for example. The function ψ is known as *Chebyshev's second function* or *Chebyshev's psi function* and M is known as *Mertens function*.

The main function appearing in this thesis will be Chebyshev's psi function. Chebyshev's psi function is named after the Russian mathematician Pafnuty Chebyshev, who lived between 1821 and 1894. As mentioned in [BJ99], some of Chebyshev's contributions were related to the distribution of prime numbers; in [Che52b], he showed that the Prime Number Theorem would follow if the limit

$$\lim_{N\to\infty} \frac{\pi(N)}{N/\log N}$$

exists. Moreover, in [Che52a], he proved a conjecture by Bertrand, currently known as Bertrand's postulate, stating that for any integer $n > 3$, there must be a prime strictly in between of n and $2n-2$. In this article, he also implicitly shows that for sufficiently large N,

$$0.92129 \cdot \frac{N}{\log N} < \pi(N) < 1.10556 \cdot \frac{N}{\log N},$$

as mentioned in [BJ99]. The Prime Number Theorem was later discovered independently by Hadamard and de la Vallée Poussin in 1896.

After the discovery of the Prime Number Theorem, the study of summatory functions like $\pi(N)$, $\psi(N)$ and $M(N)$ continued. In fact, in some sense these functions all have been related to the zeros of the Riemann zeta function, ζ, which is defined as the analytic continuation of the function

$$\zeta' : \{s \in \mathbb{C} \mid \mathfrak{Re}(s) > 1\} \to \mathbb{C}, s \mapsto \sum_{n=1}^{\infty} \frac{1}{n^s},$$

to $\mathbb{C}\backslash\{1\}$. The locations of the non-trivial zeros of the Riemann zeta function are still unknown. The Riemann hypothesis, originally formulated in 1859, does suggest a very restrictive pattern for these zeros, but it remains unsolved to this day. Because of its implications in (analytic) number theory, it is still a relevant open problem. To illustrate this, we note that the Riemann hypothesis implies sharper bounds on the above summatory functions than we have mentioned. For instance, it has been shown that the Riemann hypothesis is equivalent to the inequality

$$|\psi(N) - N| \le \frac{1}{8\pi}\sqrt{N}\log^2 N,$$

holding for $N \ge 74$, and likewise, it is also equivalent with the statement that for every $\varepsilon > 0$, there exists a constant $C(\varepsilon) > 0$ so that for sufficiently large N,

$$M(N) \le C(\varepsilon)N^{\frac{1}{2}+\varepsilon}. \tag{1.1}$$

Because of equivalences like these, there has been substantial interest in computing values of these summatory functions at large inputs. It also has led to conjectures on even sharper bounds on these functions. For instance, Mertens conjecture states that $|M(N)| \le \sqrt{N}$ for all $N \ge 1$. However, this conjecture has been disproven in 1985 by Odlyzko and te Riele in [OR85]. In fact, they mention that it is very probable that

$$\limsup_{N\to\infty} \frac{|M(N)|}{\sqrt{N}} = \infty,$$

implying that the inequality given in (1.1) is about the sharpest one could consider. We note that the proof by Odlyzko and te Riele is non-constructive in the sense that the proof does not give a particular counterexample, but rather shows one exists. Even today, there are no known counterexamples, although it has been shown in [KR06] that the first counterexample must be of size at most $10^{6.91\cdot 10^{39}}$.

The increased interest in computing values of these summatory functions has led to a search for efficient methods. A naive way to compute these sums would be to add all individual terms together. These implementations generally rely on some variant of the sieve of Eratosthenes to find the prime factorizations of integers, which generally suffices to compute individual terms. At a first thought, the naive approach may also seem to be reasonably efficient. After all, one might wonder how one would compute the number of prime numbers without listing them all, for instance. As a result, it may be somewhat surprising that in 1871, Meissel [Mei70] developed a method to compute the number of primes up to N essentially using only the primes up to $\sqrt{N}$. He used this method to compute values of $\pi(10^n)$ for $n = 7, 8, 9$ by hand, which were published in [Mei70; Mei71; Mei83; Mei85]. As mentioned in [Leh59; LMO85], the method was based on an observation by Legendre in [Leg08] stating that

$$\pi(N) - \pi\big(\lfloor\sqrt{N}\rfloor\big) + 1 = N - \sum_{\substack{p_1 \le \sqrt{N} \\ p_1 \text{ prime}}} \left\lfloor \frac{N}{p_1} \right\rfloor + \sum_{\substack{p_1 < p_2 \le \sqrt{N} \\ p_1, p_2 \text{ prime}}} \left\lfloor \frac{N}{p_1 p_2} \right\rfloor - \sum_{\substack{p_1 < p_2 < p_3 \le \sqrt{N} \\ p_1, p_2, p_3 \text{ prime}}} \left\lfloor \frac{N}{p_1 p_2 p_3} \right\rfloor + \cdots . \quad (1.2)$$

More specifically, as stated in [LMO85, p. 537], the method by Meissel "can be viewed as reducing the number of terms" present in the above identity by Legendre.

With the invention of computers, Meissel's method was later improved by several others. For instance, in [Leh59], Lehmer simplified and generalized the method by Meissel which made it possible to compute quantities such as the number of k-th prime powers below N. Lehmer also computed $\pi(10^9)$ and $\pi(10^{10})$, and noted that Meissel's calculation of $\pi(10^9)$ was off by 56. In 1985, Lagarias, Miller and Odlyzko [LMO85] improved on the method by Meissel-Lehmer and were able to compute $\pi(10^n)$ for $n = 12, \ldots, 16$.

In 1987, Lagarias and Odlyzko published a completely different method in [LO87] to compute $\pi(N)$ using integrals involving the Riemann zeta function. While this method had better theoretic bounds on the computation time, the method was not put into practice until 2013, where Platt used it to compute $\pi(10^{24})$ [Pla13]. One of the main reasons explaining the substantial time gap is the difficulty in manipulating analytic objects using computers.

Returning to the end of the 20th century, in 1996, Deléglise and Rivat [DR96a] further improved on the algorithm by [LMO85] to compute $\pi(10^n)$ for $n = 15, \ldots, 18$. Interestingly, Deléglise and Rivat have also considered methods to compute values of the Mertens function and Chebyshev's psi function in [DR96b] and [DR98], respectively. Specifically, they were able to compute $M(10^n)$ for $6 \le n \le 16$ and they compute approximations of $\psi(10^n)$ for $10 \le n \le 15$, where computations were reportedly done using 33 digits of precision.

At this point, we want to begin describing algorithms in terms of their *time* and *space complexity*. The time complexity of an algorithm represents the rough number of operations needed to complete it. In contrast, the space complexity of an algorithm represents the rough amount of memory required to complete it. To quantify the time and space complexity of an algorithm, it is conventional to use Big-O notation; it is said that the time (resp. space) complexity of an algorithm with input N is $O(g(N))$ if there exists a constant $C > 0$ such that the number of operations (resp. amount of memory) needed to complete the task with input N does not exceed $C \cdot |g(N)|$ for sufficiently large N.

Using the above notation, the method by Deléglise and Rivat to compute $\pi(N)$ has time complexity $O\big(N^{2/3}(\log N)^{-2}\big)$ and space complexity $O\big(N^{1/3}(\log N)^3 \log\log N\big)$. Their methods to compute $M(N)$ and $\psi(N)$ share the same time complexity, which is $O\big(N^{2/3}(\log\log N)^{1/3}\big)$, and the same space complexity, which is $O\big(N^{1/3}(\log\log N)^{2/3}\big)$.

As mentioned in [HT23], the methods by Deléglise and Rivat have been further improved by others. However, they also mention that as of 1996, these improvements were mainly focused on the actual implementation. As a result, the time complexities of these algorithms could only improve existing time complexities by logarithmic factors. Nevertheless, a noteworthy improvement is by Hurst in 2018, which was able to compute $M(2^n)$ for $n \leq 73$. To put the improvement in perspective, we note that $2^{73} \approx 9.4 \cdot 10^{21}$.

Only recently, there have been successful attempts to improve the time and/or space complexity by more than just a logarithmic factor. In 2023, Helfgott and Thompson [HT23] describe an algorithm to compute $M(N)$, sharing some similarities with [DR96b], with a time and space complexity of $O\big(N^{3/5}(\log N)^{3/5}(\log\log N)^{2/5}\big)$ and $O\big(N^{3/10}(\log N)^{13/10}(\log\log N)^{-3/10}\big)$ respectively. They were able to compute $M(2^n)$ for $n \leq 75$ and $M(10^n)$ for $n \leq 23$. Another successful attempt in 2023 was by Hirsch, Kessler and Mendlovic [HKM23] who describe a generic method to compute values of summatory functions such as the prime counting function and Mertens function with a time and space complexity of $O(N^{1/2+\varepsilon})$ for any $\varepsilon > 0$, where the implied constant may depend on ε.

The main focus of this thesis will be the computation of precise approximations of $\psi(N)$ using the recent techniques discussed in [HT23] and [HKM23]. We will compare the performance of both algorithms, also comparing it with the known algorithm discussed in [DR98]. We will start by introducing sieving techniques used to compute values of functions such as $\mu(n)$ and $\Lambda(n)$, in particular noting the sieve of Eratosthenes which we already briefly mentioned above. After this we will discuss how to use Vaughan's identity to obtain an identity for $\psi(N)$. This basically replaces Legendre's identity described in (1.2) used by Meissel, Lehmer and others to count primes. After this, we will get to the main part of the thesis, which compares the two approaches. We will end with details regarding the implementation in C++, where we will also go over some of the results.

1.1 Basic definitions and notation

1.1.1 Asymptotic analysis

Above we defined notions for the time and space complexity of an algorithm. We mentioned that the time (resp. space) complexity measured the number of operations (resp. amount of memory) necessary to complete a task. These complexities help to understand how quickly a computation can be done. However, we did not go over the definition of an operation or how we measure memory. Below, we will consider the computation of an individual addition, subtraction, multiplication or division to require a constant number of operations. In contrast, the amount of memory needed will be measured in the amount of bits that need to be stored as random-access memory (RAM), which is memory that is required to be retrievable within constant time. We note that in practice, operations such as additions on numbers are typically not evaluated in a constant number of operations, as they generally operate on the binary representation of integers, requiring a logarithmic amount of operations. Nevertheless, it is quite common to measure the time complexity this way and we will follow this convention.

In order to consider chains of asymptotic inequalities, we will use the notation $f(N) \ll g(N)$ to mean the same thing as $f(N) = O(g(N))$. We recall that the notation $f(N) = O(g(N))$ was used to say that there exists a constant $C > 0$ such that $f(N) \leq C \cdot |g(N)|$ for sufficiently large N. In the same vein, we will use the notation $f(N) \gg g(N)$ to say that there exists a constant $C > 0$ such that for sufficiently large N, $f(N) \geq C \cdot |g(N)|$.

We will say that $f(N)$ is of order $g(N)$ if both $f(N) \ll g(N)$ and $f(N) \gg g(N)$ hold. That is, $f(N)$ is of order $g(N)$ if there exist constants $0 < C_1 < C_2$ such that $C_1 \cdot |g(N)| \leq f(N) \leq C_2 \cdot |g(N)|$ for sufficiently large N.

We will also write $f(N) \sim g(N)$ to mean the same thing as $f(N)$ and $g(N)$ being asymptotically equal, i.e.,

$$\lim_{N \to \infty} \frac{f(N)}{g(N)} = 1.$$

Lastly, we will say that $f(n)$ is *roughly* equal to $g(n)$ if the quantity $f(n) - g(n)$ is bounded by a constant, for any $n \geq 1$.

1.1.2 Prime indexation

As seen in the introduction, prime numbers pop up frequently in text, but also in function definitions, sums etc. As a result, we will reserve the variable p to always denote a prime number. Here we potentially use subscripts when multiple primes are considered.

Chapter 2

Computational Sieves

To compute $\psi(N)$ naively, we need to store values of $\Lambda(n)$ for $n = 1, \ldots, N$. Similarly, for approaches to be considered in later chapters, we will need to store values of Λ and μ, which primarily depend on the prime factorization of the input. We will also see that it is desirable to compute the prime factorizations of integers in some range.

In this chapter, we will first focus on the sieve of Eratosthenes, which is used to generate all prime numbers up to some bound. There are different methods to compute the prime factorizations of integers, but the algorithm we consider can be easily adapted to compute the desired tables for Λ and μ as well. After discussing the sieve of Eratosthenes, we will discuss two segmented sieves which reduce the space complexity. One simply lowers the memory requirement, while the other lowers the space complexity even further, at the cost of additional time complexity. Afterwards, we extend the algorithm to compute the complete prime factorizations of integers and finally describe adaptations to the algorithm to find tables for Λ and μ.

2.1 Sieve of Eratosthenes

The sieve of Eratosthenes determines all primes up to a given bound, say N, by eliminating composite numbers. The sieve of Eratosthenes works as follows. We start with an array of the integers between 2 and N, where we keep track if integers are composite or not. At each iteration, we mark the first integer which has not been marked yet, say n, to be prime. Furthermore, we mark proper multiples of n to be composite. We then run these iterations until we have marked all integers to be either prime or composite.

To verify that the sieve of Eratosthenes works correctly, we note that the algorithm marks composites correctly. Thus, it is only possible that the algorithm incorrectly marks a composite number to be prime. Let us now suppose that q is a composite number which is incorrectly marked as a prime. Then q must be divisible by a prime $p < q$. As we established before, p cannot have been incorrectly marked to be composite, so p must have been marked to be prime. This in turn implies that q should have been marked to be composite, as multiples of p are marked to be prime in the same iteration where p is marked to be prime, leading to a contradiction. We note that, in essence, the idea behind the sieve of Eratosthenes is that an integer n is composite if and only if there is some prime $p < n$ dividing n.

Below we illustrate the sieve of Eratosthenes by considering the interval $[2, 10]$.

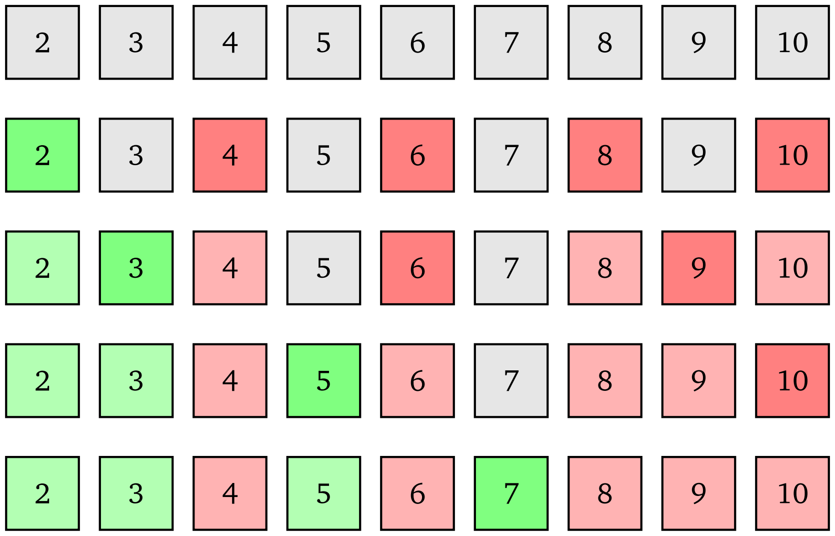


Figure 2.1: Visualization of the sieve of Eratosthenes on the integers between 2 and 10. The colors gray, green and red identify the unmarked, prime and composite numbers respectively.

Having discussed this example, we can make the observation that, when marking a prime p, it suffices to mark multiples of p of the form kp for $k \geq p$ to be composite. This can be explained by noting that smaller multiples of p are divisible by a prime $q < p$, which should therefore have been marked to be composite at an earlier stage. This observation also implies that if after some iteration all integers up to $\sqrt{N}$ are marked, then the remaining unmarked integers are prime.

Before discussing different optimizations regarding the space complexity, we briefly analyze the running time of the current algorithm. During the iteration where we mark the prime p, we mark $\lfloor N/p \rfloor$ integers in total, corresponding to the multiples of p. After having found all primes $p \leq \sqrt{N}$, we can determine the remaining primes by going over the array one last time. It follows that the time complexity of the program is of order

$$N + \sum_{p \leq \sqrt{N}} \frac{N}{p}.$$

Mertens' second theorem [Mer74; Vil05] states that $\sum_{p \leq K} p^{-1} \sim \log\log K$ as $K \to \infty$. As a result, we note that the time complexity of the current algorithm is $O(N \log\log N)$.

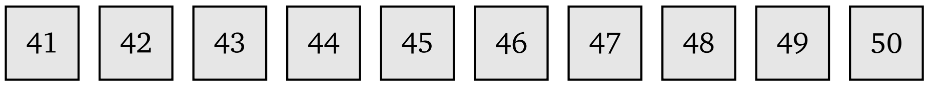
2.2 Segmented variants

Above we have illustrated the generic idea behind the sieve of Eratosthenes and we have discussed its time complexity. We will now focus on the space complexity instead. Note that in the implementation above, the memory required is proportional to the number of integers considered, as we are keeping track of a table with entries for each of the integers between 2 and N. If we are to store a complete table of all the primes up to N at some point, then there is not much memory to be saved as the number of primes less than N asymptotically grows like $N/\log N$ by the Prime Number Theorem. Therefore, in order to save memory, we want to process small segments of the table separately. We will refer to such algorithms as segmented sieves.

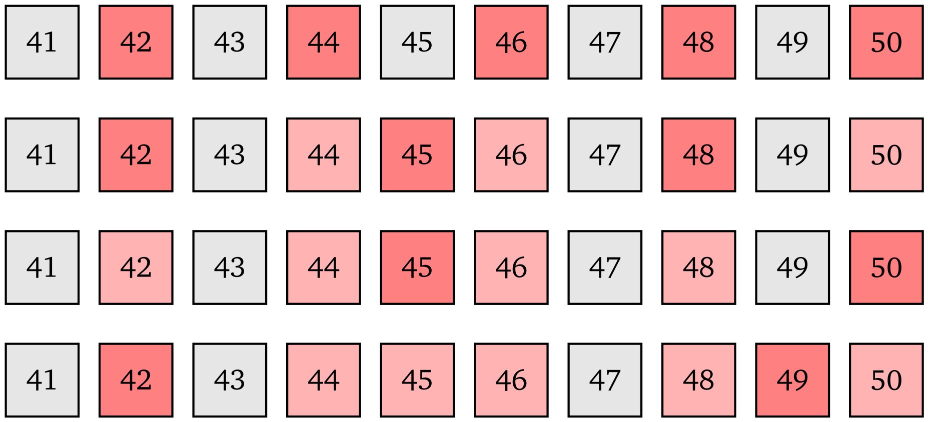
There have been several implementations of segmented sieves. However, some if not most of them can only be used to determine the primes up to some bound. While this is the essence of the sieve of Eratosthenes, we are instead interested in methods which can be generalized to compute values of functions which depend on the prime factorization of an integer such as μ. We will elaborate on a traditional description of a segmented sieve and also mention a variant by Helfgott.

2.2.1 Traditional segmented sieve

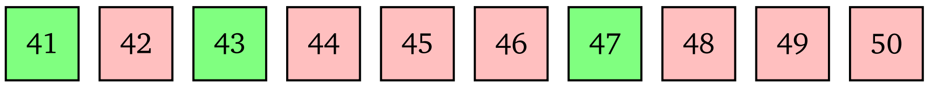
As we observed above, once we have determined all the primes up to $\sqrt{N}$ and marked all of their proper multiples to be composite, the remaining unmarked entries are prime. This motivates the idea of splitting the table into segments of length roughly $\sqrt{N}$. For a segment I, we then mark all multiples of primes $p \leq \sqrt{N}$ in I to be composite, leaving only the primes unmarked. Below we illustrate the concept with an example where we focus on a particular segment for $[2, 100]$. We assume that we have already computed the primes up to 10 using the generic version of the sieve of Eratosthenes as illustrated in Figure 2.1.



(a) We start with an array consisting of the integers between 41 and 50.



(b) We mark multiples of $2, 3, 5$ and 7 to be composite.



(c) We mark the remaining integers to be prime.

Figure 2.2: Visualization of the segmented sieve of Eratosthenes on $(40, 50] \subset [2, 100]$.
The colors gray, green and red identify the unmarked, prime and composite numbers respectively.

To implement this algorithm, we consider segments of the form $I_k = (k\sqrt{N}, (k+1)\sqrt{N}] \cap [2, N]$ for $k = 0, 1, \ldots, M = \lfloor \sqrt{N} \rfloor$. We then apply the above method on the segments $I_1, \ldots, I_M$ by first computing and storing the primes $p \leq \sqrt{N}$ by using the generic sieve of Eratosthenes on I_0. To process I_0, it takes $O(\sqrt{N} \log\log N)$ time and $O(\sqrt{N})$ memory. Similarly, to process a segment I_k for $1 \leq k \leq K$, analogous to the analysis of the generic sieve of Eratosthenes, the time and space complexity are $O(\sqrt{N} \log\log N)$ and $O(\sqrt{N})$ respectively, as the intervals are of length roughly $\sqrt{N}$.

In conclusion, by processing the segments I_k for $k \geq 1$ separately, we are able to obtain the same time complexity of $O(N \log\log N)$ to process the whole table. At the same time, we dramatically reduce the memory requirement from $O(N)$ to $O(\sqrt{N})$.

2.2.2 Helfgott's segmented sieve

In [Hel20], a segmented sieve is presented by Helfgott where segments of length roughly $\Delta = \sqrt[3]{N}(\log N)^{2/3}$ are considered. To sieve such a segment I for primes, the author asks for which integers $m \leq \sqrt{N}$ we can expect some multiple of m to be present in I. Namely, if we are given a list L of integers $2 \leq m \leq \sqrt{N}$ for which there is a multiple in I, we can essentially apply the same idea as in the traditional segmented version to obtain all the primes in I. That is, by marking the multiples of m in I for each $m \in L$, only the prime numbers in I remain unmarked.

As mentioned in [Hel20], heuristic arguments suggest that L is of size roughly $\Delta \log N$, which in turn suggests that this approach may be fruitful. Using local linear approximations and ideas stemming from Diophantine approximation, it was possible to find strict enough conditions on $m \in L$ to obtain an algorithm with time complexity $O(N \log N)$ and space complexity $O(\sqrt[3]{N}(\log N)^{2/3})$. While this algorithm has a slightly worse time complexity compared to the traditional segmented sieve, it improves the space complexity significantly. This makes this algorithm useful when memory requirements become problematic.

2.3 Prime factorization sieve

In the previous sections we have seen methods to sieve primes. We now adapt these methods to find the prime factorization of integers in $[2, N]$ for fixed N. For completeness, we consider a segmented variant to optimize the space complexity at no extra cost. We will also briefly mention Helfgott's adaptation to further optimize the space complexity at the cost of a slightly worse time complexity.

As before, we consider segments $I_k = (k\sqrt{N}, (k+1)\sqrt{N}] \cap [2, N]$, for $k = 0, 1, \ldots, \lfloor \sqrt{N} \rfloor$, of length roughly $\sqrt{N}$. We wish to compute the complete prime factorization of each integer in some segment. Let us first think of how to store the prime factorization of an integer n. A common way to store prime factorizations is to store prime divisors separately, together with their multiplicity. That is, we store the prime factorization of an integer n by storing the pairs $(p, \nu_p(n))$ for prime divisors p of n. Here, $\nu_p(n)$ denotes the multiplicity of p in the prime factorization of n.

Before computing the prime factorizations of integers inside any segment, we first compute the prime numbers up to $\sqrt{N}$ using the sieve of Eratosthenes. Let us now fix a segment I. For each prime $p \leq \sqrt{N}$ and for each integer $j = 1, \ldots, \lfloor \log_p(n) \rfloor$, in increasing order, we mark multiples of p^j in I to be divisible by p^j by adding a new pair (p, j) to the corresponding prime factorization list. After this process, for any integer $n \in I$, the pairs $(p, \nu_p(n))$ for primes $p \leq \sqrt{N}$ dividing n have been correctly established.

Therefore, it remains for us to find the pairs $(p, \nu_p(n))$ for primes $p > \sqrt{N}$ dividing n. Fortunately, any integer $n \leq N$ can have at most one prime divisor $p > \sqrt{N}$, and it appears with multiplicity 1. Hence, it suffices to compute the product

$$P_n = \prod_{p \leq \sqrt{N}} p^{\nu_p(n)}.$$

If $P_n = n$, then each prime divisor p of n satisfies $p \leq \sqrt{N}$ so that the prime factorization of n is complete. If instead $P_n \neq n$, then the remaining prime factor of n must be n/P_n, with multiplicity 1. Therefore, we add the pair $(n/P_n, 1)$ to complete the prime factorization of n.

It remains for us to analyze the time and space complexity of the described algorithm. To determine the time complexity, note that we first have to compute the primes up to $\sqrt{N}$. Using the sieve of Eratosthenes, this can be accomplished in time $O(\sqrt{N} \log \log N)$. Now, assuming we can add and update pairs to the container in $O(1)$ time, the time complexity to find all prime divisors, with multiplicity, of integers in I is majorized by

$$\sum_{p\leq\sqrt{N}}\sum_{j=1}^{\lfloor\log_p(n)\rfloor}\frac{\sqrt{N}}{p^j}\ll\sqrt{N}\cdot\sum_{p\leq\sqrt{N}}\sum_{j\geq1}\frac{1}{p^j}\ll\sqrt{N}\cdot\sum_{p\leq\sqrt{N}}\frac{1}{p}\ll\sqrt{N}\log\log N.$$

Here we use that $\sum_{j\geq1}p^{-j}=(p-1)^{-1}\ll p^{-1}$ using identities for geometric series, while for the last step we invoke Mertens' second theorem. Lastly, assuming products of two integers can be determined in $O(1)$ time, the computation time necessary to compute the prime factors $p>\sqrt{N}$ of integers $n\in I$, if they exist, is roughly

$$\sum_{n\in I}\sum_{\substack{p\leq\sqrt{N}\\p|n}}1=\sum_{p\leq\sqrt{N}}\sum_{\substack{n\in I\\p|n}}1\ll\sum_{p\leq\sqrt{N}}\frac{\sqrt{N}}{p}\ll\sqrt{N}\log\log N,$$

by again invoking Mertens' second theorem. In conclusion, the time complexity to determine the prime factorization of all integers $n\leq N$ is given by $O(N\log\log N)$, noting that there are roughly $\sqrt{N}$ segments in total.

To compute the space complexity, note that for each integer in a segment I, we need to store the integers themselves as well as their prime factorizations. We store the prime factorization by saving the pairs of prime divisors and their multiplicity. For a prime p dividing n, this takes $O(\log p+\log\nu_p(n))=O(\nu_p(n)\log p)$ space measured in bits. Noting that

$$\sum_{p|n}\nu_p(n)\log p=\log\prod_{p|n}p^{\nu_p(n)}=\log n,$$

we see that the space complexity to process a segment is of order $O(\sqrt{N}\log N)$.

2.3.1 Helfgott's implementation

Another method is described by Helfgott in [Hel20] using segments of size roughly $O(\sqrt[3]{N}(\log N)^{2/3})$. The same ideas are used as briefly mentioned in Section 2.2.2. Using the same reasoning as above, the space complexity becomes $O(\sqrt[3]{N}(\log N)^{5/3})$, as $O(\log n)$ bits are needed to store the prime factorization of n. On the other hand, the time complexity remains $O(N\log N)$. Again, while the algorithm has a slightly worse time complexity in comparison with the above implementation, it can become useful when memory requirements become an issue.

2.4 Computing tables for Λ and μ

To compute tables for Λ and μ, we could use the prime factorization sieve as before to generate the prime factorizations of integers, and use these directly to compute the values of Λ and μ quickly. However, noting that $\Lambda(n)=0$ whenever two distinct primes p and q divide n, while $\mu(n)=0$ if p^2 divides n for some prime p, it becomes clear that such an approach is needlessly inefficient. Therefore we want to adapt the prime factorization sieve to be more suited for the functions Λ and μ. We will again focus on segmented implementations. It should be mentioned that to reduce the space complexity even further, we could consider approaches as in [Hel20], as briefly mentioned in Section 2.3.1.

2.4.1 Sieving values of Λ

As before, we consider segments $I_k = (k\sqrt{N}, (k+1)\sqrt{N}] \cap [2, N]$ for $k = 0, 1, \ldots, \lfloor\sqrt{N}\rfloor$. We first apply the sieve of Eratosthenes to determine the primes up to $\sqrt{N}$. Let us now fix a segment I. We start with an empty table consisting of entries for each of the integers in I. Because of the properties of Λ, it suffices to find the prime powers in I. For each prime $p \leq \sqrt{N}$, we mark the multiples of p by updating their value in the table to be 0. Afterwards, we check for powers of p in I, and mark them by replacing their value in the table by $\log p$. Finally, after having considered all primes $p \leq \sqrt{N}$, the remaining integers are primes. Therefore, to account for the primes p in $I \neq I_0$, we update their respective value in the table to be $\log p$. At the end, we obtain a table of values of $\Lambda(n)$ for $n \in I$, as desired.

As we take similar steps as in the (segmented) sieve of Eratosthenes, their time complexities are similar. In fact, the only difference between the two algorithms is that prime powers are marked one additional time. As this can only lead to an increment of the constant factor of the time complexity of the algorithm, we note that the algorithm runs in $O(N \log\log N)$ time.

To discuss the space complexity, we have to note that, from a computational perspective, the values of Λ are stored as integers using a placeholder for logarithms. For example, $\Lambda(9)$ is stored as an object `Log(3)`, which can later be evaluated to an approximation of $\log 3$. As a result, storing values of $\Lambda(n)$ for n inside some interval $I = (N_1, N_2]$ takes space proportional to the length of I, as

$$\sum_{p^k \in I} \log p = \sum_{p^k \leq N_2} \log p - \sum_{p^k \leq N_1} \log p = \psi(N_2) - \psi(N_1) \sim N_2 - N_1.$$

As a result, the space complexity of the algorithm is $O(\sqrt{N})$.

2.4.2 Sieving values of μ

We again consider segments $I_k = (k\sqrt{N}, (k+1)\sqrt{N}] \cap [2, N]$ for $k = 0, 1, \ldots, \lfloor\sqrt{N}\rfloor$ and apply the sieve of Eratosthenes to determine the primes up to $\sqrt{N}$. Fixing a segment I, we start with a table consisting of ones for each of the integers in I. We will also keep track of a table used to mark. As μ is multiplicative and satisfies $\mu(p) = -1$ and $\mu(p^j) = 0$ for $j > 1$, it suffices to flip the sign of values corresponding to multiples of p while values corresponding to multiples of p^2 should become 0. After we have done this procedure for primes $p \leq \sqrt{N}$, it remains to take care of the integers with a prime factor $p > \sqrt{N}$. For this, we use an analogous method as used for the prime factorization sieve: for each integer n in the table, we store an additional integer $P_n = \prod_{p|n, p \leq \sqrt{N}} p$. These products can be computed during the procedure without an increase in the overall time complexity. Note that if n is squarefree satisfying $P_n \neq n$, then n has (exactly) one prime factor $p > \sqrt{N}$. To take account for this, we simply flip the sign of their corresponding entries in the table for μ.

By a similar analysis as done for the prime factorization sieve, we can observe that the time complexity of this algorithm is $O(N \log\log N)$. To compute the space complexity, we have to analyze the memory usage in a bit more detail. It turns out that on average we also need $\log N$ bits for each of the products P_n, since the space complexity to store the products P_n is of order

$$\sum_{n \in I} \log P_n = \sum_{n \in I} \sum_{\substack{p|n \\ p \leq \sqrt{N}}} \log p \ll \sqrt{N} \sum_{p \leq \sqrt{N}} \frac{\log p}{p} \ll \sqrt{N} \log N,$$

where in the last step we used that $\sum_{p \leq K} \log(p)/p \sim \log K$, which is known as Mertens' first theorem [Mer74; Vil05]. As a result, the space complexity to store the products P_n, and therefore to compute the table of values of μ is $O(\sqrt{N} \log N)$.

Chapter 3

Vaughan's Identity

The goal of this thesis is to provide different approaches to computing $\psi(N)$. Vaughan's identity forms the basis of these approaches, so we introduce the identity in this chapter. To formulate and prove Vaughan's identity we first establish some standard definitions and results from analytic number theory.

We recall from the introductory chapter that a number theoretic function is a complex-valued function from the natural number. Apart from the examples given in the introduction, simple but important examples of number theoretic functions are $\mathbf{1}, \delta : \mathbb{N} \to \mathbb{C}$, where for any $n \in \mathbb{N}$,

$$\mathbf{1}(n) = 1 \qquad \text{and} \qquad \delta(n) = \begin{cases} 1, & \text{if } n = 1 \\ 0, & \text{otherwise.} \end{cases}$$

Definition 3.1. *Given two number theoretic functions f and g, we define their Dirichlet convolution to be $f * g : \mathbb{N} \to \mathbb{C}$, where for any $n \in \mathbb{N}$,*

$$(f * g)(n) = \sum_{d_1 d_2 = n} f(d_1) g(d_2).$$

We remark that δ acts as an identity element for Dirichlet convolutions: given any number theoretic function f, we observe that

$$(f * \delta)(n) = \sum_{d_1 d_2 = n} f(d_1)\delta(d_2) = f(n).$$

In fact, the set of number theoretic functions f satisfying $f(1) \neq 0$ form a (commutative) group with operator $*$ and identity element δ. While this result is not particularly important to prove Vaughan's identity, it gives rise to the following useful examples.

Example 3.2. *The Möbius function μ satisfies the identity*

$$\sum_{d|n} \mu(d) = \begin{cases} 1, & \textit{if } n = 1 \\ 0, & \textit{otherwise.} \end{cases}$$

That is, we have an identity between number theoretic functions $\mathbf{1} * \mu = \delta$.

Example 3.3. *Given an integer $n \geq 1$, write $n = p_1^{k_1} \cdots p_t^{k_t}$ using unique prime factorization. We observe that by definition of Λ,*

$$\sum_{d|n} \Lambda(d) = \sum_{j=1}^{t} \sum_{k=1}^{k_t} \log p_j = \sum_{j=1}^{t} \log p_j^{k_j} = \log \prod_{j=1}^{t} p_j^{k_j} = \log n.$$

As a result, we note that $\mathbf{1} * \Lambda = \log$, or equivalently, using Example 3.2, $\Lambda = \mu * \log$.

The above example already gives a useful relation between Λ, μ and $\log$. However, to do computations using these relations, we would like to restrict the number of divisor pairs (d_1, d_2) with $d_1 d_2 = n$ to be considered. Vaughan's identity restricts these pairs by bounding divisors. For simplicity, given a number theoretic function f and a fixed integer M, we define the number theoretic functions $f_{\leq M}$, $f_{>M}$ such that for any $n \in \mathbb{N}$,

$$f_{\leq M}(n) = \begin{cases} f(n), & \text{if } n \leq M \\ 0, & \text{otherwise,} \end{cases} \qquad \text{and} \qquad f_{>M}(n) = \begin{cases} f(n), & \text{if } n > M \\ 0, & \text{otherwise.} \end{cases}$$

We can now state Vaughan's identity as follows.

Lemma 3.4 (Cf. [Vau80]). *Fix integers $D, M \geq 1$. Then for any integer $n > M$, we have that*

$$\Lambda(n) = (\mu_{\leq D} * \log)(n) - (\mu_{\leq D} * \Lambda_{\leq M} * \mathbf{1})(n) + (\mu_{>D} * \Lambda_{>M} * \mathbf{1})(n).$$

That is, for $n > M$,

$$\Lambda(n) = \sum_{\substack{dk=n \\ d \leq D}} \mu(d) \log(k) - \sum_{\substack{dmk=n \\ d \leq D, m \leq M}} \mu(d)\Lambda(m) + \sum_{\substack{dmk=n \\ d>D, m>M}} \mu(d)\Lambda(m).$$

This identity can be proven using a relation between number theoretic functions and their respective *Dirichlet series*.

Definition 3.5. *Let f be a number theoretic function. We define the formal Dirichlet series of f as*

$$D(f, s) = \sum_{n=1}^{\infty} \frac{f(n)}{n^s}.$$

Below, we consider formal Dirichlet series on subsets of $\mathbb{C}$ where it converges absolutely. We remark that any Dirichlet series has a unique *abscissa of absolute convergence* $\sigma \in \mathbb{R} \cup \{\pm\infty\}$ such that for $s \in \mathbb{C}$, $D(f, s)$ converges absolutely when $\mathfrak{Re}(s) > \sigma$, while $D(f, s)$ either converges conditionally or diverges when $\mathfrak{Re}(s) < \sigma$. For simplicity, we will write

$$H_\sigma = \{s \in \mathbb{C} : \mathfrak{Re}(s) > \sigma\}.$$

A number theoretic function is completely characterized by its Dirichlet series: if there exists a non-empty open subset $U \subset \mathbb{C}$ such that for two number theoretic functions f and g we have $D(f, s) = D(g, s)$ for all $s \in U$, then $f = g$. Therefore, any identity between Dirichlet series can be translated into an identity between their respective underlying number theoretic functions. We will use this to prove Vaughan's identity.

We will now discuss the foreshadowed relation between Dirichlet convolutions and Dirichlet series.

Lemma 3.6. *Let f and g be number theoretic functions and suppose that $D(f, \cdot)$ and $D(g, \cdot)$ have abscissa of absolute convergence at least σ. Then $D(f * g, \cdot)$ converges absolutely on H_σ and for all $s \in H_\sigma$, we have*

$$D(f * g, s) = D(f, s)D(g, s).$$

Proof. Let $s \in H_\sigma$. By absolute convergence of $D(f, s)$ and $D(g, s)$, we may rearrange terms to obtain

$$\begin{aligned} D(f, s)D(g, s) &= \left(\sum_{n=1}^{\infty} f(n)n^{-s}\right)\left(\sum_{m=1}^{\infty} g(m)m^{-s}\right) = \sum_{n=1}^{\infty}\sum_{m=1}^{\infty} f(n)g(m)(nm)^{-s} \\ &= \sum_{k=1}^{\infty}\left(\sum_{nm=k} f(n)g(m)\right)k^{-s}. \end{aligned}$$

Remarking that $(f*g)(k) = \sum_{nm=k} f(n)g(m)$, we observe that the identity $D(f*g,s) = D(f,s)D(g,s)$ holds. It remains for us to prove that the series converges absolutely. To this end, remark that

$$\begin{aligned}\sum_{k=1}^{\infty} \left| k^{-s} \sum_{nm=k} f(n)g(m) \right| &= \sum_{k=1}^{\infty} k^{-\mathfrak{Re}(s)} \left| \sum_{nm=k} f(n)g(m) \right| \\ &\leq \sum_{k=1}^{\infty} k^{-\mathfrak{Re}(s)} \sum_{nm=k} |f(n)| \cdot |g(m)|.\end{aligned}$$

By rearranging terms as before, we obtain that

$$\sum_{k=1}^{\infty} k^{-\mathfrak{Re}(s)} \sum_{nm=k} |f(n)| \cdot |g(m)| = \left(\sum_{n=1}^{\infty} |f(n)| n^{-\mathfrak{Re}(s)} \right) \left(\sum_{m=1}^{\infty} |g(m)| m^{-\mathfrak{Re}(s)} \right) < \infty,$$

by absolute convergence of $D(f,s)$ and $D(g,s)$. Hence $D(f*g,\cdot)$ converges absolutely on H_σ. ■

We finally prove Vaughan's identity.

Proof of Lemma 3.4. We recall that given integers $D, M \geq 1$, we need to show that for any integer $n > M$,

$$\Lambda(n) = (\mu_{\leq D} * \log)(n) - (\mu_{\leq D} * \Lambda_{\leq M} * \mathbf{1})(n) + (\mu_{>D} * \Lambda_{>M} * \mathbf{1})(n).$$

We have argued that it suffices to prove the identity for their corresponding Dirichlet series. To this end, let $s \in \mathbb{C}$ with $\mathfrak{Re}(s) > 1$, so that all considered Dirichlet series converge absolutely. We first note that

$$D(\mu_{>D}, s) = D(\mu, s) - D(\mu_{\leq D}, s) \qquad \text{and} \qquad D(\Lambda_{>M}, s) = D(\Lambda, s) - D(\Lambda_{\leq M}, s).$$

Hence, using Lemma 3.6, we can write

$$D(\mu_{>D} * \Lambda_{>M} * \mathbf{1}, s) = D(\Lambda_{>M}, s)\big(D(\mu, s) - D(\mu_{\leq D}, s)\big)D(\mathbf{1}, s).$$

Since $\mu * \mathbf{1} = \delta$, we have $D(\mu, s)D(\mathbf{1}, s) = D(\delta, s) = 1$, again by Lemma 3.6. As a result,

$$D(\mu_{>D} * \Lambda_{>M} * \mathbf{1}, s) = D(\Lambda_{>M}, s)\big(1 - D(\mu_{\leq D}, s)D(\mathbf{1}, s)\big).$$

By expanding the above product and by using that $D(\Lambda_{>M}, s) = D(\Lambda, s) - D(\Lambda_{\leq M}, s)$, we obtain that

$$D(\mu_{>D} * \Lambda_{>M} * \mathbf{1}, s) = D(\Lambda_{>M}, s) - D(\Lambda, s)D(\mu_{\leq D}, s)D(\mathbf{1}, s) + D(\Lambda_{\leq M}, s)D(\mu_{\leq D}, s)D(\mathbf{1}, s).$$

Rearranging terms, while noting that $D(\Lambda, s)D(\mathbf{1}, s) = D(\log, s)$ using Lemma 3.6, we find that

$$D(\Lambda_{>M}, s) = D(\mu_{\leq D}, s)D(\log, s) - D(\Lambda_{\leq M}, s)D(\mu_{\leq D}, s)D(\mathbf{1}, s) + D(\mu_{>D} * \Lambda_{>M} * \mathbf{1}, s).$$

By applying Lemma 3.6 once again and by using the unique correspondence between Dirichlet series and their underlying number theoretic functions, we have proven the desired identity. ■

It's worth noting that there exist various formulations of Vaughan's identity, which often can be found to be equivalent using techniques such as Möbius inversion, as mentioned in [HT23]. In fact, a generalization of Vaughan's identity can be found in [Hea82, pp. 1366–1369] and [IK04], which is useful in different settings.

We conclude this chapter with the following corollary of Vaughan's identity, which will provide the basis for later chapters.

Corollary 3.7. *Fix an integer $N \geq 1$ and write $M = \lfloor \sqrt{N} \rfloor$. Then for any integer $M < n \leq N$, we have that*

$$\Lambda(n) = \sum_{\substack{dk=n \\ d \leq M}} \mu(d) \log(k) - \sum_{\substack{dmk=n \\ d,m \leq M}} \mu(d)\Lambda(m).$$

In particular,

$$\psi(N) = \psi(M) + \sum_{n \leq N} \sum_{\substack{dk=n \\ d \leq M}} \mu(d) \log(k) - \sum_{n \leq N} \sum_{\substack{dmk=n \\ d,m \leq M}} \mu(d)\Lambda(m).$$

Proof. Apply Vaughan's identity with $D = M = \lfloor \sqrt{N} \rfloor$. Then we obtain that for $M < n \leq N$,

$$\Lambda(n) = \sum_{\substack{dk=n \\ d \leq M}} \mu(d) \log(k) - \sum_{\substack{dmk=n \\ d,m \leq M}} \mu(d)\Lambda(m) + \sum_{\substack{dmk=n \\ d,m > M}} \mu(d)\Lambda(m).$$

We are able to omit the sum

$$\sum_{\substack{dmk=n \\ d,m > M}} \mu(d)\Lambda(m)$$

as it is empty for any $n \leq N < (M+1)^2$. By adding the terms for $n = M+1, \dots, N$, we obtain that

$$\psi(N) = \psi(M) + \sum_{M < n \leq N} \sum_{\substack{dk=n \\ d \leq M}} \mu(d) \log(k) - \sum_{M < n \leq N} \sum_{\substack{dmk=n \\ d,m \leq M}} \mu(d)\Lambda(m). \tag{3.1}$$

Finally, note that for $n \leq M$,

$$\sum_{\substack{dk=n \\ d \leq M}} \mu(d) \log(k) = \sum_{dk=n} \mu(d) \log(k) = (\mu * \log)(n) = \Lambda(n),$$

using that $\mu * \log = \Lambda$ as we discussed in Example 3.3. Similarly, for $n \leq M$ we have that

$$\sum_{\substack{dmk=n \\ d,m \leq M}} \mu(d)\Lambda(m) = \sum_{dmk=n} \mu(d)\Lambda(m) = (\mu * \Lambda * \mathbf{1})(n) = \Lambda(n),$$

since $\mu * \mathbf{1} = \delta$. As a result, the difference of the two above double sums cancel each other out for $n \leq M$. We conclude that (3.1) can be rewritten as

$$\psi(N) = \psi(M) + \sum_{n \leq N} \sum_{\substack{dk=n \\ d \leq M}} \mu(d) \log(k) - \sum_{n \leq N} \sum_{\substack{dmk=n \\ d,m \leq M}} \mu(d)\Lambda(m). \qquad \blacksquare$$

Chapter 4

An Elementary Approach

In this chapter, we will discuss an elementary approach to calculate $\psi(N) = \sum_{n \le N} \Lambda(n)$. The method we apply is based on [HT23]; we first apply an identity to split the desired sum into smaller sums. Then we use some techniques to make computations feasible. This generic approach is also used for similar elementary algorithms discussed in [LMO85] and [DR96b], for example.

The identity we are going to use is Vaughan's identity, which has been discussed in the previous chapter. In particular, we will use Corollary 3.7. We recall that Corollary 3.7 states that given N, writing $M = \lfloor \sqrt{N} \rfloor$, we have that

$$\psi(N) = \psi(M) + \sum_{\substack{dk \le N \\ d \le M}} \mu(d) \log(k) - \sum_{\substack{dmk \le N \\ d,m \le M}} \mu(d)\Lambda(m). \tag{4.1}$$

The term $\psi(M)$ can be computed naively using a computational sieve for values of Λ, as discussed in Chapter 2. However, by observing that

$$\psi(M) = \sum_{m \le M} \Lambda(m) = \sum_{p \le M} \lfloor \log_p(M) \rfloor \log(p),$$

we can save on the number of operations needed as we now only need to consider the primes up to M instead of all prime powers up to M.

We will now quickly glance over the computation of the first double sum appearing in Equation 4.1. Note that

$$\sum_{\substack{dk \le N \\ d \le M}} \mu(d) \log(k) = \sum_{d \le M} \mu(d) \sum_{k \le N/d} \log(k).$$

To store the sums $\sum_{k \le N/d} \log(k) = \log\left(\lfloor N/d \rfloor !\right)$ exactly, we essentially need to store exact representations of $n!$ for certain integers $n \ge M$. This would take too much time and space. As a result, we will store these sums by considering approximations. Given these approximations, we can compute the above sum by storing a table of the values of $\mu(d)$ for $d \le M$, using the corresponding computational sieve discussed in Chapter 2.

As a result, to compute the first double sum in Equation 4.1, we need to find approximations of $\log(n!)$ for large integer values for n. It makes sense to consider the expression $\log \Gamma(x)$ for $x > 0$ instead, as Γ is analytic on the half-plane $\{s \in \mathbb{C} : \mathfrak{Re}(s) > 0\}$, while $\Gamma(n) = (n-1)!$ for any integer $n \ge 1$. The following lemma allows us to find arbitrarily close approximations of $\log \Gamma(x)$ relatively quickly.

Lemma 4.1 (Cf. [Tem96]). *For any integer $k \geq 0$, let us denote $B_k \in \mathbb{Q}$ for the k-th Bernoulli number, uniquely characterized by the property that for any $z \in \mathbb{C}$ with $0 < |z| < 2\pi$,*

$$\frac{z}{e^z - 1} = \sum_{k=0}^{\infty} \frac{B_k}{k!} z^k.$$

Then for any real number $x > 0$ and for any integer $m \geq 1$, we have that

$$\log \Gamma(x) = \left(x - \frac{1}{2}\right) \log x - x + \log \sqrt{2\pi} + \sum_{n=1}^{m-1} \frac{B_{2n}}{2n(2n-1)} x^{1-2n} + R_m(x),$$

where

$$|R_m(x)| \leq \frac{|B_{2m}|}{2m(2m-1)} x^{1-2m}.$$

We observe that the larger x is chosen, the faster the convergence is of the series in the above lemma. To give a sense for how quickly this series converges, note that for a practical situation in which $N > 10^{10}$, it suffices to consider $m = 3$ to get approximations of $\log(n!)$ for integers $n \geq M > 10^5$ with accuracy $7.0 \cdot 10^{-29}$ at worst. As a result, the practical limitation of these computations rather depends on the precision bound guaranteed by compilers. In Chapter 6 we elaborate on this limitation in more detail.

We will finally focus on the second double sum appearing on the right-hand side of Equation 4.1. Following [HT23], we consider a parameter $M_0 \leq M$ and we observe that

$$\sum_{\substack{mdk \leq N \\ m,d \leq M}} \Lambda(m)\mu(d) = \sum_{\substack{mdk \leq N \\ M_0 < \max\{m,d\} \leq M}} \Lambda(m)\mu(d) + \sum_{\substack{mdk \leq N \\ m,d \leq M_0}} \Lambda(m)\mu(d). \tag{4.2}$$

The analysis of the two double sums on the right-hand side of Equation 4.2 will take considerably longer. We will therefore discuss these in two separate sections.

4.1 Dependent variable case

We start by considering the double sum in Equation 4.2 in the case where m and d cannot be taken to be completely independent. That is, we construct an algorithm to compute the double sum

$$\sum_{\substack{mdk \leq N \\ M_0 < \max\{m,d\} \leq M}} \Lambda(m)\mu(d). \tag{4.3}$$

We consider the cases $m \leq d$ and $d \leq m$ separately. As we essentially consider the case $d = m$ twice, we have that

$$\sum_{\substack{mdk \leq N \\ M_0 < \max\{m,d\} \leq M}} \Lambda(m)\mu(d) = \sum_{\substack{mdk \leq N \\ M_0 < d \leq M \\ m \leq d}} \Lambda(m)\mu(d) + \sum_{\substack{mdk \leq N \\ M_0 < m \leq M \\ d \leq m}} \Lambda(m)\mu(d) - \sum_{M_0 < m \leq M} \Lambda(m)\mu(m) \left\lfloor \frac{N}{m^2} \right\rfloor. \tag{4.4}$$

Observing that $\mu(m) = 0$ whenever m is not squarefree, while $\Lambda(m) = 0$ whenever m is not a prime power, it follows that $\mu(m)\Lambda(m) = 0$ when m is composite. Thus, the latter sum on the right-hand side of Equation 4.4 simplifies to

$$-\sum_{M_0 < m \leq M} \Lambda(m)\mu(m) \left\lfloor \frac{N}{m^2} \right\rfloor = -\sum_{M_0 < p \leq M} \Lambda(p)\mu(p) \left\lfloor \frac{N}{p^2} \right\rfloor = \sum_{M_0 < p \leq M} \log(p) \left\lfloor \frac{N}{p^2} \right\rfloor.$$

Note that we can compute approximations of the above sum using a segmented computational sieve for finding primes, as discussed in Chapter 2. Hence we only have to focus on the two double sums on the right-hand side of Equation 4.4. In order to reduce the space needed to discuss these cases, we will consider the general expression

$$\sum_{\substack{abk\leq N\\ M_0<a\leq M\\ b\leq a}} f(a)g(b),$$

where f and g are arithmetic functions. Note that this allows us to retrieve our original sums by considering $(f,g)\in\{(\Lambda,\mu),(\mu,\Lambda)\}$. By writing $D_g(n,a)=\sum_{b|n,b\leq a} g(b)$, we can rewrite the summation as

$$\sum_{\substack{abk\leq N\\ M_0<a\leq M\\ b\leq a}} f(a)g(b) = \sum_{M_0<a\leq M} f(a)\sum_{n\leq N/a}\sum_{\substack{b|n\\ b\leq a}} g(b) = \sum_{M_0<a\leq M} f(a)\sum_{n\leq N/a} D_g(n,a). \tag{4.5}$$

Suppose that we can find an upper bound $C(n)$ for the expected number of operations needed to compute $D_g(k,a)$ for some $k=1,\dots,n$ and some fixed value of a. Then we can recursively compute and keep track of the sums $S_g(n,a)=\sum_{k\leq n} D_g(k,a)$ for $n\in[N/M,N/M_0]$ in $O\big((N/M_0)C(N/M_0)\big)$ time and negligible space. Hence, given a segmented table of values $f(a)$ for the integers $a\in[M_0,M]$, we are able to compute the double sum in Equation 4.5 in $O\big((N/M_0)C(N/M_0)\big)$ time. Note that for $f\in\{\mu,\Lambda\}$, we can compute tables of values of f using (segmented) computational sieves discussed in Chapter 2. We discuss the corresponding time and space complexity in Section 4.1.1.

Following our previous argument, it remains for us to find an upper bound only dependent on n in which we are able to compute $D_g(k,a)$ on average over $k=1,\dots,n$ given some fixed integer a, where $g\in\{\mu,\Lambda\}$. As the values of $\mu(n)$ and $\Lambda(n)$ depend on the decomposition of n into prime factors, we want to compute and store the prime factorizations of integers $n\leq N/M_0$. Note that this again can be done using (segmented) computational sieves discussed in Chapter 2. We elaborate on the corresponding time and space complexity in Section 4.1.1.

In [HT23], an algorithm is presented to compute $D_\mu(n,a)$ for any $n\in[K,2K]$ and fixed a in $O(\log\log K)$ operations on average, given the prime factorization of n. In particular, it is possible to compute $D_\mu(n,a)$ for $n\in[1,\dots,N/M_0]$ and fixed a in $O(\log\log N/M_0)$ operations on average. In order to not repeat their work on this matter, we will instead focus on the simpler case for $D_\Lambda(n,a)$. Suppose the prime factorization of n is given by $p_1^{e_1}\cdots p_t^{e_t}$. Then we have

$$D_\Lambda(n,a)=\sum_{\substack{d|n\\ d\leq a}}\Lambda(d)=\sum_{\substack{1\leq j\leq t\\ 1\leq k\leq e_j\\ p_j^k\leq a}}\Lambda(p_j^k)=\sum_{\substack{1\leq j\leq t\\ 1\leq k\leq e_j\\ p_j^k\leq a}}\log p_j=\sum_{j=1}^{t}\min\left\{e_j,\left\lfloor\frac{\log a}{\log p_j}\right\rfloor\right\}\log p_j.$$

As a result, the computation of $D_\Lambda(n,a)$ takes time proportional to the number of distinct prime divisors of n, also denoted $\omega(n)$. To analyze the amount of operations needed, we note that since $\omega(n)=\sum_{p|n}1$, we have

$$\sum_{n\leq K}\omega(n)=\sum_{n\leq K}\sum_{p|n}1=\sum_{p\leq K}\sum_{\substack{n\leq K\\ p|n}}1=\sum_{p\leq K}\left\lfloor\frac{K}{p}\right\rfloor=\sum_{p\leq K}\left(\frac{K}{p}+O(1)\right)\sim K\log\log K.$$

Here, we repeat that from Chapter 2 that the last step follows from Mertens' second theorem [Mer74; Vil05], stating that $\sum_{p\leq K}p^{-1}\sim\log\log K$. We conclude that the average number of distinct prime divisors of $n\leq K$ is $O(\log\log K)$. Thus, for $n=1,\dots,N/M_0$ we can compute $D_\Lambda(n,a)$ in $O(\log\log(N/M_0))$ time on average, given the prime factorization of n.

In conclusion, we compute the double sum

$$\sum_{\substack{mdk\le N\\ M_0<\max\{m,d\}\le M}} \Lambda(m)\mu(d)$$

via a case distinction according to Equation 4.4. The latter sum on the right-hand side of Equation 4.4 is computed naively. The remaining two double sums have been rewritten so that they can essentially be computed via calculation of a couple of simpler summations while storing partial results.

4.1.1 Analysis of the time and space complexity

We have referred a couple of times to Chapter 2 for the computation of tables of values for Λ and μ, as well as a list of prime factorizations for integers in some interval. Here we did not mention details regarding the time and space complexity. This was done deliberately as we consider two variants of segmented sieves in Chapter 2: the traditional approach and an adaptation by Helfgott discussed in [Hel20]. We will briefly analyze the time and space complexity of the computation of the dependent variable sum for both approaches.

In case of the traditional segmented sieve, we can store tables of $\Lambda(n)$ and $\mu(n)$ for $n \in [M_0, M]$ via segments of length roughly $\sqrt{M}$ resulting in a time complexity of $O(M\log\log M)$ and space complexity of $O(\sqrt{M})$. To store the prime factorizations of integers $n \in [1, N/M_0]$, we can proceed by using segments of length roughly $\sqrt{N/M_0}$ resulting in a time complexity of $O\big((N/M_0)\log\log N\big)$ and space complexity of $O\big(\sqrt{N/M_0}\log N\big)$. The time complexity for the remaining tasks described above is given by $O\big((N/M_0)\log\log N\big)$, with negligible space consumption. We conclude that the desired sum in (4.3) can be computed in $O\big((N/M_0)\log\log N\big)$ time and $O\big(\sqrt{N/M_0}\log N\big)$ space, since $N/M_0 \ge M$.

If we were to use the adaptation by Helfgott, then we can store the tables of $\Lambda(n)$ and $\mu(n)$ for $n \in [M_0, M]$ using segments of length in the order of $\sqrt[3]{M}(\log N)^{2/3}$, resulting in a time and space complexity of $O(M\log M)$ and $O\big(\sqrt[3]{M}(\log N)^{5/3}\big)$ respectively, assuming we first compute the prime factorizations of integers in a segment. Similarly, to find the prime factorizations of integers in $[1, N/M_0]$, Helfgott's segmented sieve considers segments of length in the order of $\sqrt[3]{N/M_0}(\log N)^{2/3}$. This results in a time and space complexity of $O\big((N/M_0)\log N\big)$ and $O\big(\sqrt[3]{N/M_0}(\log N)^{5/3}\big)$ respectively.

4.2 Independent variable case

In this section we compute the double sum

$$\sum_{\substack{mdk\le N\\ m,d\le M_0}} \Lambda(m)\mu(d) = \sum_{m,d\le M_0} \Lambda(m)\mu(d)\left\lfloor \frac{N}{md}\right\rfloor. \tag{4.6}$$

The approach we consider is identical to the approach taken in [HT23], so we mainly discuss the general steps.

The main problem with calculating the above double sum is the fact the expression inside the floor function is dependent on both m and d. Suppose for an instance that the expression could be written as a sum $F(m) + G(d)$. Then we could proceed using a separation of variables:

$$\sum_{m,d\le M_0} \Lambda(m)\mu(d)\left(F(m)+G(d)\right) = \sum_{m\le M_0}\Lambda(m)F(m)\sum_{d\le M_0}\mu(d) + \sum_{m\le M_0}\Lambda(m)\sum_{d\le M_0}\mu(d)G(d). \tag{4.7}$$

Provided that values of F and G can be computed in constant time, this makes evaluation of the desired sum possible in $O(M_0 \log\log M_0)$ operations and $O(\sqrt{M_0}\log M_0)$ space by using traditional segmented sieves discussed in Chapter 2 to obtain tables of Λ and μ.

With the above observation in mind, it makes sense to consider the local linear approximation of the function $f(x,y) = N/(xy)$ at a point $(m_0, d_0) \in \mathbb{N}^2$, which is given by

$$\frac{N}{xy} \approx \frac{N}{m_0 d_0} + c_x(x - m_0) + c_y(y - d_0), \tag{4.8}$$

where

$$c_x = \frac{\partial}{\partial x}\left(\frac{N}{xy}\right)\bigg|_{(x,y)=(m_0,d_0)} = -\frac{N}{m_0^2 d_0} \qquad \text{and} \qquad c_y = \frac{\partial}{\partial y}\left(\frac{N}{xy}\right)\bigg|_{(x,y)=(m_0,d_0)} = -\frac{N}{m_0 d_0^2}.$$

Note that the individual terms in the local linear approximation are not dependent on both x and y. This motivates the analysis of the difference between, for instance,

$$S = \sum_{m,d\le M_0} \Lambda(m)\mu(d)\left\lfloor \frac{N}{md}\right\rfloor \tag{4.9}$$

and

$$S_1 = \sum_{m,d\le M_0} \Lambda(m)\mu(d)\left(\left\lfloor \frac{N}{m_0 d_0} + c_x(m - m_0)\right\rfloor + \lfloor c_y(d - d_0)\rfloor\right), \tag{4.10}$$

as S_1 can be computed using separation of variables as illustrated in Equation 4.7. This illustrates the basic idea used in [HT23] to compute the desired appearing in (4.6). It turns out that it is desirable to compute the difference $S - S_1$ by considering an intermediate step. More specifically, in [HT23], a method is described which we will use to compute $S - S_1$ via computation of the differences $S - S_0$ and $S_0 - S_1$, where

$$S_0 = \sum_{m,d\le M_0} \Lambda(m)\mu(d)\left\lfloor \frac{N}{m_0 d_0} + c_x(m - m_0) + c_y(d - d_0)\right\rfloor. \tag{4.11}$$

Similar as in [HT23], from now on, we consider the quantities

$$L(m,d) = \left\lfloor \frac{N}{md}\right\rfloor, \qquad L_0(m,d) = \left\lfloor \frac{N}{m_0 d_0} + c_x(m - m_0) + c_y(d - d_0)\right\rfloor$$

and

$$L_1(m,d) = \left\lfloor \frac{N}{m_0 d_0} + c_x(m - m_0)\right\rfloor + \lfloor c_y(d - d_0)\rfloor,$$

so that $S = \sum_{m,d\le M_0} \Lambda(m)\mu(d)L(m,d)$, while $S_j = \sum_{m,d\le M_0} \Lambda(m)\mu(d)L_j(m,d)$ for $j \in \{0,1\}$.

To analyze the differences $S - S_0$ and $S_0 - S_1$, we need to look at the differences $L(m,d) - L_0(m,d)$ and $L_0(m,d) - L_1(m,d)$. As a result, we need to have control over the error term in Equation 4.8, for which it makes sense to consider the above local linear approximations of $N/(xy)$ on small rectangular regions. We will mimic the setup used in [HT23], which can be summarized as follows. We first partition the region $[1, M_0]^2$ into dyadic rectangles of the form $R(A,B) = [A, 2A) \times [B, 2B)$. A fixed rectangle $R = R(A,B)$ is then subdivided into rectangles $I_x \times I_y$ of fixed width $2a$ and height $2b$ with a specified center (m_0, d_0), so that $I_x = [m_0 - a, m_0 + a) \cap [1, M_0]$, and $I_y = [d_0 - b, d_0 + b) \cap [1, M_0]$. The values of a and b are determined for each region $R = R(A,B)$ and are used to bound the error term in Equation 4.8. We discuss values for these parameters in Section 4.2.1.

Below we fix a rectangular region $I_x \times I_y$ using the notation as above. In [HT23], formulas are established for the differences $L(m,d) - L_0(m,d)$ and $L_0(m,d) - L_1(m,d)$ for $(m,d) \in I_x \times I_y$. Before we state these results, we need to establish some additional notation used in [HT23]. First of all, the coefficient c_y appearing in Equation 4.8 is approximated by a reduced fraction a_0/q where $q \leq Q := 2b$, with the property that the difference

$$\delta := c_y - \frac{a_0}{q}$$

satisfies $|\delta| \leq 1/(qQ)$. We note that this bound on δ can be accomplished by Dirichlet's approximation theorem. Additionally, for fixed $m \in I_x$, we write $r_0 \in \{0, 1, \ldots, q\}$ such that

$$\beta := \left\{ \frac{N}{m_0 d_0} + c_x(m - m_0) \right\} - \frac{r_0}{q}$$

is minimized. If multiple choices of r_0 exists, the greater one is chosen to ensure that $\beta \in [-\frac{1}{2q}, \frac{1}{2q})$. We discuss details regarding the implementation of obtaining a_0 and q in Chapter 6.

Using the above notation, we are able to state the results from [HT23] on the differences between $L(m,d)$, $L_0(m,d)$ and $L_1(m,d)$. We first consider the general case.

Lemma 4.2 (Cf. [HT23]). *Suppose that $(m,d) \in I_x \times I_y$. If $a_0(d-d_0) + r_0 \not\equiv 0, -1 \pmod{q}$, then we have that $L(m,d) = L_0(m,d)$. Moreover, if $q \nmid a_0(d-d_0) + r_0$, then*

$$L_0(m,d) - L_1(m,d) = \begin{cases} 1 & \textit{if } \overline{a_0(d-d_0)} > q - r_0, \\ 1 & \textit{if } q \mid (d-d_0) \textit{ and } \delta(d-d_0) < 0, \\ 0 & \textit{otherwise,} \end{cases}$$

where $\overline{k}$ represents the unique integer in $\{0, 1, \ldots, q-1\}$ congruent to k modulo q.

The result for the remaining case for the difference $L_0(m,d) - L_1(m,d)$ can be stated as follows.

Lemma 4.3 (Cf. [HT23]). *Suppose that $(m,d) \in I_x \times I_y$ while $a_0(d-d_0) + r_0 \equiv 0 \pmod{q}$. Then, if $r_0 \not\equiv 0 \pmod{q}$,*

$$L_0(m,d) - L_1(m,d) = \begin{cases} 1 & \textit{if } \beta + \delta(d-d_0) \geq 0, \\ 0 & \textit{otherwise.} \end{cases}$$

If instead $r_0 \equiv 0 \pmod{q}$, then

$$L_0(m,d) - L_1(m,d) = \begin{cases} 1 & \textit{if } \beta < 0 \textit{ and } \delta(d-d_0) < 0, \\ 1 & \textit{if } \beta\delta(d-d_0) < 0 \textit{ and } \beta + \delta(d-d_0) \geq 0, \\ 0 & \textit{otherwise.} \end{cases} \tag{4.12}$$

Let us now fix an integer $m \in I_x$. To discuss the remaining cases for the difference $L(m,d) - L_0(m,d)$, we define integer intervals $I = I(m)$ and $J_r = J_r(m)$ for $r = 0, 1$ so that

$$I = \{d \in \mathbb{Z} : \beta + \delta(d-d_0) < 0\} \qquad \text{and} \qquad J_r = \{d \in \mathbb{Z} : \gamma_2 d^2 + \gamma_{1,r} d + \gamma_0 < 0\},$$

where $\gamma_0 = qN$, $\gamma_2 = -a_0 m$ and

$$\gamma_{1,r} = m \left(a_0 d_0 - (r_0 + r) - q \left\lfloor \frac{N}{m_0 d_0} + c_x(m - m_0) \right\rfloor \right).$$

As noted in [HT23], the coefficient γ_2 is non-negative. As a result, we note that J_r is indeed an interval for $r = 0, 1$.

Below we consider the difference $L(m,d) - L_0(m,d)$ when $a_0(d-d_0) + r_0 \equiv 0, -1 \pmod{q}$. Note that the cases $q = 1$ and $q > 1$ are considered separately to take into account that for $q = 1$ the residue classes 0 and -1 modulo q overlap.

Lemma 4.4 (Cf. [HT23]). *Suppose that $(m,d) \in I_x \times I_y$. If $q > 1$ and $a_0(d-d_0)+r_0 \equiv 0 \pmod q$, then*
$$L(m,d) - L_0(m,d) = \begin{cases} 1 & \text{if } d \in I - (I \cap J_0), \\ 0 & \text{otherwise.} \end{cases}$$

Lemma 4.5 (Cf. [HT23]). *Suppose that $(m,d) \in I_x \times I_y$. If $q > 1$ and $a_0(d-d_0)+r_0 \equiv -1 \pmod q$, then*
$$L(m,d) - L_0(m,d) = \begin{cases} 1 & \text{if } d \notin J_1, \\ 0 & \text{otherwise.} \end{cases}$$

Lemma 4.6 (Cf. [HT23]). *Suppose $(m,d) \in I_x \times I_y$. If $q = 1$, then*
$$L(m,d) - L_0(m,d) = \begin{cases} 1 & \text{if } d \notin (I \cap J_0) \sqcup ((\mathbb{N}\backslash I) \cap J_1), \\ 0 & \text{otherwise.} \end{cases}$$

We will not discuss the proofs of Lemmas 4.2–4.6 in detail, as this has already been done in [HT23]. However, we do want to remark that the proofs are very elementary in essence.

As we stated earlier, I, J_0 and J_1 are integer intervals. This makes them easy to work with from a computational perspective, as we can store them in constant time using only the two endpoints of the interval. It should be noted that the complement of I, i.e., $\mathbb{N}\backslash I$, is also an integer interval. Moreover, the intersection of two intervals is again an interval.

As mentioned in [HT23], the roots of a quadratic equation can be computed in constant time, using the quadratic formula. As a result, computation of the intervals J_0 and J_1 does not require more than constant time as well. Similarly, the intervals I and $\mathbb{N}\backslash I$ can be computed in constant time, as well as any intersection of two intervals. As a result, for fixed $m \in I_x$, as claimed in [HT23], it is possible to compute the sums
$$S_0' = \sum_{d \in I_y} \mu(d)\big(L(m,d) - L_0(m,d)\big) \qquad \text{and} \qquad S_1' = \sum_{d \in I_y} \mu(d)\big(L_0(m,d) - L_1(m,d)\big)$$
in constant time, assuming we have stored tables of values
$$\rho_{r,d'} = \sum_{\substack{d \in I_y,\, d \le d' \\ a_0(d-d_0) \equiv r \pmod q}} \mu(d) \qquad \text{and} \qquad \sigma_r = \sum_{\substack{d \in I_y \\ \overline{a_0(d-d_0)} > q-r}} \mu(d),$$
for $r \in \{0,1,\ldots,q-1\}$ and $d' \in [d_0-b, d_0+b)$, as well as the sum
$$\tau = \sum_{\substack{d \in I_y \\ \delta(d-d_0) < 0 \\ q \mid (d-d_0)}} \mu(d).$$

We recall that we use $\overline{n}$ to denote the integer representative of n modulo q in $[0,q)$.

To illustrate how the sums S_1' and S_2' can be computed in constant time by storing the above objects, we consider the computation of the contribution of Lemma 4.5. Assume that we have determined that $J_1 = [d_1, d_2]$. Then the contribution corresponding with Lemma 4.5 is given by
$$\sum_{\substack{d \in I_y \\ a_0(d-d_0) \equiv -r_0-1 \pmod q}} \mu(d) - \sum_{\substack{d \in I_y,\, d_1 \le d \le d_2 \\ a_0(d-d_0) \equiv -r_0-1 \pmod q}} \mu(d).$$

If we were to write $\rho'_d = \rho_{\overline{-r_0-1},\min\{d,d_0+b-1\}}$ for $d \geq d_0 - b$, while $\rho'_d = 0$ for $d < d_0 - b$, the above expression simplifies to

$$\rho'_{d_0+b-1} - \big(\rho'_{d_2} - \rho'_{d_1-1}\big),$$

which can be computed in constant time, given ρ. We note that the table ρ suffices for the computation of the contributions corresponding with Lemmas 4.3–4.6, while the table σ and the value τ are used to compute the contribution of Lemma 4.2.

It remains for us to elaborate on the computation and storage of the objects $\rho_{r,d'}$, σ and τ. We note that the tables ρ and σ, as well as the value τ, can be computed in $O(b)$ operations, assuming we have access to a table of the values of μ. To see why, we note that ρ and τ essentially consider partial sums of μ, which can be computed recursively. Then σ can be computed as a partial sum of values of $\rho^*_r = \rho_{r,d_0+b-1}$:

$$\sigma_{r+1} = \sum_{\substack{d \in I_y \\ \overline{a_0(d-d_0)} \geq q-r}} \mu(d) = \sigma_r + \rho^*_{q-r}.$$

To elaborate on the space complexity of storing the above objects, we note that ρ actually stores redundant information as $\rho_{r,d'} = \rho_{r,d''}$, where $d'' \leq d'$ is the largest integer congruent to r modulo q, as mentioned in [HT23]. This essentially means that we can reduce the size of ρ by a factor of q, leading to ρ only storing $O(b)$ entries. As these entries are of size $O(b)$, this means that ρ can be stored using $O(b \log b)$ space. The objects σ and τ also require $O(b \log b)$ space, meaning that the overall memory usage is $O(b \log b)$.

In conclusion, the computation of

$$S = \sum_{m,d \leq M_0} \Lambda(m)\mu(d) \left\lfloor \frac{N}{md} \right\rfloor = \sum_{m,d \leq M_0} \Lambda(m)\mu(d)L(m,d)$$

is done by considering local linear approximations of $N/(xy)$ centered at small rectangles. Given a specific small rectangle $I_x \times I_y$, we compute

$$S' = \sum_{(m,d) \in I_x \times I_y} \Lambda(m)\mu(d)L(m,d)$$

by relating it to

$$S'_j = \sum_{(m,d) \in I_x \times I_y} \Lambda(m)\mu(d)L_j(m,d),$$

for $j = 0, 1$. As S'_1 allows separation of variables, it is possible to compute S'_1 relatively efficiently, in $O(a+b)$ operations, assuming we have access to tables of Λ and μ. Additionally, by fixing $m \in I_x$, we elaborated on the method discussed in [HT23] allowing computation of

$$\sum_{d \in I_y} \mu(d)\big(L(m,d) - L_0(m,d)\big) \qquad \text{and} \qquad \sum_{d \in I_y} \mu(d)\big(L_0(m,d) - L_1(m,d)\big)$$

in constant time. This was done by computing and storing certain tables using $O(b)$ operations and $O(b \log b)$ space, excluding the computation and storage of the necessary table of values of μ. As these tables can be used for all values of $m \in I_x$, it is possible to compute the differences $S' - S'_0$ and $S'_0 - S'_1$ in $O(a+b)$ operations. As a result, we have shown that S' can be computed in $O(a+b)$ operations and $O(b \log b)$ memory, excluding the time and space necessary to compute and store the values of μ.

4.2.1 Choices for parameters a and b and analysis of the time complexity

There are a couple of things left for us to discuss. First of all, we should obtain values of a and b. Recall that a and b were defined such that $I_x = [m_0 - a, m_0 + a)$ and $I_y = [d_0 - b, d_0 + b)$, and that they should be chosen such that the absolute error in Equation 4.8 is at most $1/(2b)$. Moreover, as it stands, we have only given an upper bound on the number of operations needed to compute

$$\sum_{(m,d)\in I_x\times I_y} \Lambda(m)\mu(d)\left\lfloor \frac{N}{md}\right\rfloor .$$

In this section, we will establish an upper bound for the number of operations needed to compute the complete sum related to the independent variable case, i.e.,

$$\sum_{m,d\le M_0} \Lambda(m)\mu(d)\left\lfloor \frac{N}{md}\right\rfloor .$$

In [HT23], a method is described on how to choose values of a and b given the constraint on the error term. In short, for any region $R(A,B) = [A, 2A)\times[B, 2B)$, it suffices to take

$$a = \left\lfloor A\cdot\sqrt[3]{\frac{A}{6N}}\right\rfloor \qquad \text{and} \qquad b = \left\lfloor B\cdot\sqrt[3]{\frac{A}{6N}}\right\rfloor . \tag{4.13}$$

The choices for a and b become problematic in case that $a = 0$ or $b = 0$, as the corresponding rectangles $I_x \times I_y \subset R(A,B)$ would be empty. We explain how to solve this issue in Section 4.2.2.

From now on, we assume that $a, b \ge 1$. We compute the time complexity of the computation over a dyadic rectangle $R(A,B)$. We recall that, provided we have computed and stored values of Λ and μ, the computation over a fixed rectangle $I_x \times I_y \subset R(A,B)$ can be done in $O(a+b)$ additional operations and $O(b\log b)$ additional memory. To store values of Λ and μ efficiently, we process the rectangle $R(A,B)$ in batches of width and length of order $\sqrt{M_0}$. This way, we can use traditional segmented sieves discussed in Chapter 2 to compute the values of Λ and μ corresponding to the rectangle $R(A,B)$ in $O\big((A+B)\log\log N\big)$ operations and $O(\sqrt{M_0})$ space. For each individual batch, being a subset of some rectangle $I_x \times I_y$, we can then compute its contribution to

$$\sum_{(m,d)\in I_x\times I_y} \Lambda(m)\mu(d)\left\lfloor \frac{N}{md}\right\rfloor , \tag{4.14}$$

with a negligible time increment. As a result, the computation of (4.14) over all rectangles $I_x \times I_y$ in a dyadic region $R(A,B)$ is possible in

$$O\left((A+B)\log\log N + \frac{AB}{ab}\cdot(a+b)\right) = O\left(\sqrt[3]{\frac{N}{A}}\cdot(A+B)\right) = O\Big(N^{1/3}\big(A^{2/3} + BA^{-1/3}\big)\Big) \tag{4.15}$$

operations and

$$O\big(\sqrt{M_0} + b\log b\big) = O\left(\sqrt{M_0} + B\sqrt[3]{\frac{A}{N}}\log N\right) = O\left(\sqrt{M_0} + \sqrt[3]{\frac{M_0^4}{N}}\log N\right) \tag{4.16}$$

memory, since $A, B \le M_0$.

Note that we have simplified the space complexity in (4.16) so that it is independent of A, B. Therefore, the estimate of the space complexity suffices for all rectangles $R(A, B) \subset [1, M_0]^2$. In other words, the space complexity for the computation of the sum

$$\sum_{d,m\leq M_0} \Lambda(m)\mu(d) \left\lfloor \frac{N}{md} \right\rfloor$$

is $O(\max\{\sqrt{M_0}, \sqrt[3]{M_0^4/N}\log N\})$. To compute its corresponding time complexity, we note that it remains to sum (4.15) over A, B of the form 2^i and 2^j with $i, j = 0, 1, \ldots, \lfloor \log_2(M_0) \rfloor$ to cover the region $[1, M_0]^2$ completely. That is, the time complexity is majorized by

$$N^{1/3} \sum_{i,j\leq \log_2 M_0} \left(2^{2i/3} + 2^{j-i/3}\right)$$

operations. Using identities for geometric series, we note that the first part of the sum simplifies to

$$N^{1/3} \sum_{i,j\leq \log_2 M_0} 2^{2i/3} \ll N^{1/3} \log M_0 \sum_{i\leq \log_2 M_0} 2^{2i/3} \ll N^{1/3} M_0^{2/3} \log M_0.$$

Likewise, the second part of the sum simplifies to

$$N^{1/3} \sum_{i\leq \log_2 M_0} 2^{-i/3} \sum_{j\leq \log_2 M_0} 2^j \ll N^{1/3} M_0^{2/3}.$$

We conclude that we can compute the sum corresponding to the independent variable sum in

$$O(N^{1/3} M_0^{2/3} \log M_0) = O(N^{1/3} M_0^{2/3} \log N) \tag{4.17}$$

operations and

$$O\left(\sqrt{M_0} + \sqrt[3]{\frac{M_0^4}{N}} \log N\right) \tag{4.18}$$

memory.

4.2.2 Alternative approach in case that a or b is small

As we mentioned in the previous section, the analysis above does not account for the case where $a = 0$ or $b = 0$. In this case, we can instead compute the sum naively. In fact, as in [HT23], we will consider naive computation in a more general setting. More specifically, as in [HT23], we will consider naive computation in case that $\min\{a, b\} < C$, for some constant $C \geq 1$ which can be chosen freely.

The corresponding rectangles being relatively small seems to suggest that naive computation should not have an impact on the time complexity. However, without being careful, this is actually false. To illustrate this, we note that as it stands, we want to evaluate the sum

$$\sum_{(m,d)\in I_x\times I_y} \Lambda(m)\mu(d) \left\lfloor \frac{N}{md} \right\rfloor$$

naively if either $a < C$ or $b < C$, where $|I_x| = 2a$ and $|I_y| = 2b$. Noting that all rectangles to be considered in the dyadic region $R(A, B) \supset I_x \times I_y$ are of equal length and width, this directly implies that we want to compute the sum

$$\sum_{(m,d)\in R(A,B)} \Lambda(m)\mu(d) \left\lfloor \frac{N}{md} \right\rfloor \tag{4.19}$$

naively if either $A \leq (6C^3N)^{1/4}$ or $A^{1/3}B \leq (6C^3N)^{1/3}$. Here, we transformed the conditions on a and b into conditions on A and B by using (4.13). Note that the naive computation of (4.19) takes $O(AB)$ operations, representing the size of the rectangle $R(A,B) = [A, 2A) \times [B, 2B)$.

Focusing on the case $A \leq (6C^3N)^{1/4} \ll N^{1/4}$, the time complexity becomes of order at least

$$\left(\sum_{i \leq \log_2 \sqrt[4]{N}} 2^i \right) \left(\sum_{j \leq \log_2 M_0} 2^j \right) \gg N^{1/4} M_0, \tag{4.20}$$

where we identified A and B with 2^i and 2^j respectively, sharing similarities with the end of Section 4.2.1. It is not immediately clear that this becomes problematic. So, to put it into perspective, we note that the final time complexity of the described algorithm is approximately $O(N^{3/5})$, while (4.20) would result in a time complexity of order at least $N^{13/20}$.

The previous suggests that our current restrictions for the consideration of naive computation are too lenient. As a result, we should consider a more restrictive setting. In [HT23], it was a tangible decision to consider the restriction $B \leq A$. The reason for this decision can be explained by noting that they originally consider the sum

$$\sum_{m,d \leq M_0} \mu(m)\mu(d) \left\lfloor \frac{N}{md} \right\rfloor,$$

which is symmetric in both variables. As the computations over rectangles $[A, 2A) \times [B, 2B)$ and $[B, 2B) \times [A, 2A)$ yield the same value, the restriction $B \leq A$ is useful to essentially halve the number of operations needed. A less evident reason to consider this restriction is to reduce the time complexity corresponding to the naive computation for these small rectangles: the time complexity corresponding to the case $B \leq A \leq (6C^3N)^{1/4}$ is majorized by

$$\sum_{0 \leq j \leq i \leq \log_2 \sqrt[4]{N}} 2^{i+j} \leq \sum_{i,j \leq \log_2 \sqrt[4]{N}} 2^{i+j} \ll N^{1/2}.$$

Additionally, the time complexity corresponding to the case $A^{1/3}B \leq (6C^3N)^{1/3} \ll N^{1/3}$ is of order

$$\sum_{\substack{i,j \leq \log_2 M_0 \\ i/3 + j \leq \log_2 \sqrt[3]{N}}} 2^{i+j} \ll \sum_{i \leq \log_2 M_0} \left(2^i \sum_{j \leq -i/3 + \log_2 \sqrt[3]{N}} 2^j \right) \ll N^{1/3} \cdot \sum_{i \leq \log_2 M_0} 2^{2i/3} \ll N^{1/3} M_0^{2/3}.$$

We note that the above estimates of the time complexity can be determined using identities for geometric series and are negligible in comparison with (4.15).

The drawback of using the restriction $B \leq A$ is obvious; they cannot be considered for any rectangles $R(A,B)$ where $B > A$. As these rectangles do need to be considered, at a first glance, this may seem to be introducing a new problem. However, we want to note that the functions Λ and μ appearing in (4.6), (4.7), (4.9), (4.10) and (4.11) can essentially be replaced by any two functions f and g, without needing to change anything essential in the described method to compute their differences. The only thing to keep track of is the fact that we would then need to consider tables of values of f instead of Λ, or tables of values of g instead of μ. In particular, we can swap the roles of Λ and μ to obtain the desired symmetry to consider dyadic rectangles $R(A,B)$ where $B > A$.

To conclude this section, we note that for small rectangles it is possible to consider naive computation without increasing the overall time complexity by restricting the computation to dyadic regions $R(A,B)$ for which $B \leq A$. To compute the contributions on dyadic regions $R(A,B)$ for which $B > A$, we can swap the roles of Λ and μ throughout Section 4.2 without other issues arising.

4.3 Final estimates of the time and space complexity

We have described an algorithm to calculate $\psi(N)$ by first rewriting it using Vaughan's identity. At the end of the introductory discussion of the sum, we determined that the sum could be computed using a case distinction for large indices and small indices based on a parameter $M_0 \leq M = \lfloor \sqrt{N} \rfloor$.

The case where one of the indices is large is handled by writing the sum towards divisors sums to be computed recursively. We showed that it requires either $O\big((N/M_0)\log\log N\big)$ operations and $O(\sqrt{N/M_0}\log N)$ memory if we consider traditional segmented sieves, or $O\big((N/M_0)\log N\big)$ operations and $O\big(\sqrt[3]{N/M_0}(\log N)^{5/3}\big)$ memory when considering Helfgott's adaptation discussed in [Hel20].

On the other hand, the case where both indices are small requires $O\big(N^{1/3}M_0^{2/3}\log N\big)$ operations and $O\big(\max\{\sqrt[3]{M_0^4/N}\log N, \sqrt{M_0}\}\big)$ memory by subdividing the calculation over small rectangles. This allows us to consider local linear approximations to reduce double sums to a couple of related simple sums. We took care of small rectangles using a naive approach. We showed that it is possible to implement naive computation without increasing the order of the time complexity.

To obtain the optimal time bound using this approach, we consider both variants of usage of segmented computational sieves separately. In the case of usage of traditional segmented sieves, we should take M_0 to be of order $N^{2/5}(\log N)^{-3/5}(\log\log N)^{3/5}$ which would result in a time complexity of $O\big(N^{3/5}(\log N)^{3/5}(\log\log N)^{2/5}\big)$ and a space complexity of $O\big(N^{3/10}(\log N)^{13/10}(\log\log N)^{-3/10}\big)$. If we were to consider Helfgott's adaptation instead, then we optimal time bound by taking M_0 to be of order $N^{2/5}$, with corresponding time and space complexity given by $O(N^{3/5}\log N)$ and $O\big(N^{1/5}(\log N)^{5/3}\big)$ respectively. We finally remark that the usage of the traditional segmented sieve throughout Section 4.2 does not impact the overall space complexity.

Chapter 5

An Approach using Fourier Theory

We will now consider an approach which uses ideas from Fourier theory. The actual usage of Fourier theory will not be discussed in this chapter, and can instead be found in Appendix A.

As before, let $N \geq 1$ be an integer and write $M = \lfloor \sqrt{N} \rfloor$. Applying Vaughan's identity, we obtain that

$$\psi(N) = \psi(M) + \sum_{\substack{dk \leq N \\ d \leq M}} \mu(d) \log(k) - \sum_{\substack{mdk \leq N \\ m,d \leq M}} \Lambda(m)\mu(d). \tag{5.1}$$

As explained in Chapter 4, we can approximate both $\psi(M)$ and the first double sum in (5.1) relatively easily. We will therefore shift focus to the second double sum.

We recall from Chapter 3 that given an arithmetic function f, we define its restriction to integers less than M as $f_{\leq M}$. That is,

$$f_{\leq M}(n) = \begin{cases} f(n), & \text{if } n \leq M, \\ 0, & \text{otherwise.} \end{cases}$$

In order to compute the latter double sum, we now follow an approach similar to that considered in [HKM23] for which we first rewrite it as

$$\sum_{n \leq N} (\Lambda_{\leq M} * \mu_{\leq M} * \mathbf{1})(n) \tag{5.2}$$

We consider a parameter $\Delta = \Delta_N > 0$ and define the segmentation array of a function (with respect to Δ). For reasons which only become apparent later, we will assume that $\Delta \to 0$ as $N \to \infty$, while $\Delta \gg N^{-\varepsilon}$ for some $\varepsilon \in (0,1)$. When optimizing Δ, we will observe that taking Δ to be of order $1/\sqrt{N}$ minimizes the time complexity, thereby satisfying the above constraints.

Definition 5.1 (Segmentation array, cf. [HKM23])**.** *Given a number theoretic function $f : \mathbb{N} \to \mathbb{C}$, we define its segmentation array $\overline{f} : \{0, 1, \ldots, \overline{N}\} \to \mathbb{C}$, with $\overline{N} = \lfloor \Delta^{-1} \log_2 N \rfloor$, via*

$$\overline{f}[k] = \sum_{2^{k\Delta} \leq n < 2^{(k+1)\Delta}} f(n), \qquad k = 0, 1, \ldots, \overline{N}.$$

We also define the segmentation index $\overline{k}(n) = \lfloor \Delta^{-1} \log_2 n \rfloor$ so that for $k = 0, 1, \ldots, \overline{N}$,

$$\overline{f}[k] = \sum_{n \,:\, \overline{k}(n) = k} f(n).$$

The segmentation index $\overline{k}(n)$ has the following property if n is a product of integers, which turns out to be useful later on.

Lemma 5.2 (Cf. [HKM23]). *Fix a positive integer $t \geq 2$. For any choice of positive integers $n_1, \dots, n_t$, we have the following inequalities:*

$$\overline{k}\Big(\prod_{i=1}^{t} n_i\Big) - t + 1 \leq \sum_{i=1}^{t} \overline{k}(n_i) \leq \overline{k}\Big(\prod_{i=1}^{t} n_i\Big).$$

Proof. The proof follows easily by induction after proving the base case ($t = 2$), stating that

$$\overline{k}(n_1 n_2) - 1 \leq \overline{k}(n_1) + \overline{k}(n_2) \leq \overline{k}(n_1 n_2),$$

or in other words,

$$\left\lfloor \frac{\log_2(n_1 n_2)}{\Delta} \right\rfloor - 1 \leq \left\lfloor \frac{\log_2(n_1)}{\Delta} \right\rfloor + \left\lfloor \frac{\log_2(n_2)}{\Delta} \right\rfloor \leq \left\lfloor \frac{\log_2(n_1 n_2)}{\Delta} \right\rfloor.$$

Recall that for any two real numbers $x, y > 0$, $\lfloor x + y \rfloor - 1 \leq \lfloor x \rfloor + \lfloor y \rfloor \leq \lfloor x + y \rfloor$, which shows the above holds using logarithm properties. ■

Given two arithmetic functions f and g, we now wish to define a convolution of $\overline{f}$ and $\overline{g}$ and relate it to the Dirichlet convolution $f * g$.

Definition 5.3 (Discrete convolution). *Given two arrays $\overline{f}$ and $\overline{g}$ of the same size $\overline{N} + 1$, we define their discrete convolution via*

$$(\overline{f} \star \overline{g})[k] = \sum_{k_1 + k_2 = k} \overline{f}[k_1]\overline{g}[k_2],$$

for $k = 0, 1, 2, \dots, \overline{N}$.

The reason for introducing discrete convolutions while (5.2) considers Dirichlet convolutions can be explained by the fact that discrete convolutions can be transformed into products relatively easily, while Dirichlet convolutions cannot. More specifically, using the *discrete Fourier transformation*, say $\mathcal{F}$, the convolution $\overline{f} \star \overline{g}$ can be related to $\mathcal{F}^{-1}(\mathcal{F}(\overline{f}) \cdot \mathcal{F}(\overline{g}))$, where multiplication is taken entry-wise. This relation is discussed in more detail in Appendix A. In particular, we show that this allows us to compute all entries of $\overline{f} \star \overline{g}$ in $O(\overline{N} \log \overline{N})$ operations and $O(\overline{N})$ memory using a so-called *Fast Fourier Transform*. For comparison, the naive approach would take $O(\overline{N}^2)$ operations and $O(\overline{N})$ memory.

We should mention that our method differs slightly from the method described in [HKM23]. Instead of discrete Fourier transformations, [HKM23] considers *number theoretic transformations*. In our current setting, we cannot use the same kind of transformation, as number theoretic transformations operate on integer-valued functions, which would not work for Λ. We discuss more details regarding the differences in Section 6.4.2. There we also discuss a potential method to apply number theoretic transformations by reconsidering our setting.

To relate the Dirichlet convolution as in the summand of (5.2) to the corresponding discrete convolution $\overline{\Lambda_{\leq M}} \star \overline{\mu_{\leq M}} \star \overline{\mathbf{1}}$, we will rewrite

$$\begin{aligned}(\overline{\Lambda_{\leq M}} \star \overline{\mu_{\leq M}} \star \overline{\mathbf{1}})[k] &= \sum_{k_1 + k_2 + k_3 = k} \overline{\Lambda_{\leq M}}[k_1] \star \overline{\mu_{\leq M}}[k_2] \star \overline{\mathbf{1}}[k_3] \\ &= \sum_{\substack{d_1, d_2, d_3: \\ \overline{k}(d_1) + \overline{k}(d_2) + \overline{k}(d_3) = k}} \Lambda_{\leq M}(d_1)\mu_{\leq M}(d_2)\mathbf{1}(d_3),\end{aligned}$$

where in the second step, we used the second characterization of the segmentation array as in Definition 5.1. This implies that

$$\sum_{k=0}^{\overline{k}(N)} (\overline{\Lambda_{\leq M}} \star \overline{\mu_{\leq M}} \star \overline{\mathbf{1}})[k] = \sum_{\substack{d_1, d_2, d_3: \\ \overline{k}(d_1)+\overline{k}(d_2)+\overline{k}(d_3) \leq \overline{k}(N)}} \Lambda_{\leq M}(d_1)\mu_{\leq M}(d_2)\mathbf{1}(d_3).$$

The idea now is to distinguish the cases $d_1d_2d_3 \leq N$ and $d_1d_2d_3 > N$. Namely, suppose that $d_1d_2d_3 \leq N$. Then, by Lemma 5.2, the condition $\overline{k}(d_1) + \overline{k}(d_2) + \overline{k}(d_3) \leq \overline{k}(N)$ is satisfied, so that

$$\begin{aligned} \sum_{\substack{d_1d_2d_3 \leq N \\ \overline{k}(d_1)+\overline{k}(d_2)+\overline{k}(d_3) \leq \overline{k}(N)}} \Lambda_{\leq M}(d_1)\mu_{\leq M}(d_2)\mathbf{1}(d_3) &= \sum_{d_1d_2d_3 \leq N} \Lambda_{\leq M}(d_1)\mu_{\leq M}(d_2)\mathbf{1}(d_3) \\ &= \sum_{n \leq N} (\Lambda_{\leq M} * \mu_{\leq M} * \mathbf{1})(n). \end{aligned}$$

As a result, analogous to [HKM23], we have proven the following lemma.

Lemma 5.4 (Cf. [HKM23]). *For any integer $N \geq 1$, we have that*

$$\sum_{k \leq \overline{k}(N)} (\overline{\Lambda_{\leq M}} \star \overline{\mu_{\leq M}} \star \overline{\mathbf{1}})[k] - \sum_{n \leq N} (\Lambda_{\leq M} * \mu_{\leq M} * \mathbf{1})(n) = \sum_{\substack{d_1d_2d_3 > N \\ \overline{k}(d_1)+\overline{k}(d_2)+\overline{k}(d_3) \leq \overline{k}(N)}} \Lambda_{\leq M}(d_1)\mu_{\leq M}(d_2)\mathbf{1}(d_3). \tag{5.3}$$

As we mentioned above, in Appendix A we show that the first sum appearing in Equation 5.3 can be calculated in $O(\overline{N} \log \overline{N})$ operations and $O(\overline{N})$ space, where $\overline{N} = \overline{k}(N) \ll \Delta^{-1} \log N$, given the entries of $\overline{\Lambda_{\leq M}}$, $\overline{\mu_{\leq M}}$ and $\overline{\mathbf{1}}$. That is, given the required segmentation arrays, we can compute the first sum appearing in Equation 5.3 in $O\big(\Delta^{-1}(\log N)^2\big)$ operations and $O(\Delta^{-1} \log N)$ memory.

We note that we can compute the arrays $\overline{\Lambda_{\leq M}}$ and $\overline{\mu_{\leq M}}$ in $O(M \log \log M + \Delta^{-1} \log N)$ operations and $O(\sqrt{M} + \Delta^{-1} \log N)$ memory by first computing tables of the values of $\Lambda(n)$ and $\mu(n)$ for $n \leq M$ using computational sieves discussed in Chapter 2. Moreover, the individual entries of the segmentation array $\overline{\mathbf{1}}$ can be computed in constant time, resulting in an additional time and space complexity of $O(\Delta^{-1} \log N)$.

To compute the desired sum

$$\sum_{n \leq N} (\Lambda_{\leq M} * \mu_{\leq M} * \mathbf{1})(n),$$

it remains for us to compute the right-hand side of Equation 5.3, which we will refer to as the error term. As in [HKM23], we compute the error term using a naive approach. More specifically, we first compute a bound on the integers $n > N$ which can be written as a product of three terms, say $n = d_1d_2d_3$, such that $\overline{k}(d_1) + \overline{k}(d_2) + \overline{k}(d_3) \leq \overline{k}(N)$. We then compute the prime factorizations of these integers to find all of their decompositions into three factors. Finally, we determine the decompositions satisfying the constraint on the segmentation indices, and for those satisfying the constraint we evaluate their contribution to the error term.

We first establish a bound on the integers $n > N$ satisfying $n = d_1d_2d_3$ with $\overline{k}(d_1) + \overline{k}(d_2) + \overline{k}(d_3) \leq \overline{k}(N)$ for some triplet (d_1, d_2, d_3) of positive integers.

Lemma 5.5 (Cf. [HKM23]). *Suppose that a tuple (d_1, d_2, d_3) of positive integers with $n = d_1d_2d_3 > N$ satisfies $\overline{k}(d_1) + \overline{k}(d_2) + \overline{k}(d_3) \leq \overline{k}(N)$. Then n lies in the interval $(N, N + S]$ for some S of order ΔN.*

Proof. By Lemma 5.2, we have that $\overline{k}(n) - 2 \leq \overline{k}(d_1) + \overline{k}(d_2) + \overline{k}(d_3) \leq \overline{k}(N)$. By definition of $\overline{k}$, this implies that $n \leq 2^{3\Delta}N$. Since $\Delta \to 0$ as $N \to \infty$, Taylor's Theorem implies that $2^{3\Delta} - 1$ is of order Δ. Letting $S = (2^{3\Delta} - 1)N$, we conclude that $n \leq N + S$, where S is of order ΔN. ■

To determine the decompositions of $n \in (N, N+S]$, we need to find the complete prime factorizations of these integers. Using the prime factorization sieve discussed in Chapter 2, this can be done in $O(\max\{\sqrt{N}, \Delta N\} \log\log N)$ operations and $O(\sqrt{N} \log N)$ memory. To elaborate on this in a bit more detail, we note that in order to compute the prime factorizations of integers in $(N, N + S]$, we first need to compute the primes up to $\sqrt{N+S} \ll \sqrt{N}$, which takes $O(\sqrt{N} \log\log N)$ operations and $O(\sqrt{N} \log N)$ memory. We can then process the interval $(N, N + S]$ in batches of length roughly $\sqrt{N}$, and proceed by following the method described in Chapter 2.

Given $n \in (N, N + S]$, it remains for us to compute the sum

$$\sum_{\substack{d_1d_2d_3=n \\ \overline{k}(d_1)+\overline{k}(d_2)+\overline{k}(d_3)\leq\overline{k}(N)}} \Lambda_{\leq M}(d_1)\mu_{\leq M}(d_2)\mathbf{1}(d_3),$$

given the prime factorization of n, say $n = p_1^{e_1} \cdots p_t^{e_t}$. As stated previously, this is done using the naive method involving calculating the individual decompositions into three factors of integers n in $(N, N + S]$. To obtain all decompositions of n into three factors, we essentially have to iterate through the prime divisors of n: for $j = 1, \ldots, t$ we need to subdivide factors of $p_j^{e_j}$ over the three parts of the decomposition.

In Section 6.2.2, we mention that the number of operations needed to compute the above decompositions is majorized by their count. Additionally, it should be noted that since we essentially store the prime factorizations of the decompositions, it is possible to compute $\Lambda(d_1)\mu(d_2)\mathbf{1}(d_3)$ in negligible time. In particular, there is no need to store tables of values of Λ and μ. We conclude that, given the factorization of n, the time complexity to compute the error term is bounded by the total number of decompositions into three factors of n.

In order to bound the number of decompositions, we use the following lemma.

Lemma 5.6 (Cf. [Shi80; NT98]). *Let f be a non-negative multiplicative arithmetic function and assume that there exist constants C, D such that $f(p^l) \leq C \cdot l^D$ for any prime p and any integer $l \geq 1$. Then, given $N^\alpha \leq S \leq N$ for some $0 < \alpha < 1$, we have that*

$$\sum_{N<n\leq N+S} f(n) \ll \frac{S}{\log N} \cdot \exp\left(\sum_{p\leq N} \frac{f(p)}{p}\right),$$

where the implied constant depends only on C, D and α.

In short, the lemma gives an upper bound on the average value of $f(n)$ when n ranges over a short interval $(N, N+S]$, where the bound can be computed using only the values of f at primes $p \leq N$. This allows us to quickly derive an upper bound on the number of decompositions of integers in $(N, N+S]$ into a fixed number of factors.

Corollary 5.7 (Cf. [HKM23]). *The number of decompositions of integers in* $(N, N+S]$ *into* k *factors is* $O(\Delta N \log^{k-1} N)$.

Proof. We apply Lemma 5.6 with the function $f = \tau_k$, where $\tau_k(n)$ counts the number of decompositions of n into k factors. It is not hard to see that τ_k satisfies the conditions of the lemma: τ_k is non-negative and multiplicative and with counting arguments it can be observed that

$$\tau_k(p^l) = \binom{l+k-1}{k-1} \ll l^{k-1}.$$

Moreover, since $\Delta \gg N^{-\varepsilon}$ for some $\varepsilon \in (0,1)$, it follows that there exists $\alpha \in (0,1)$ such that for sufficiently large N, $N^\alpha \leq S \leq N$, given that S is of order ΔN. We conclude that the conditions in order to apply Lemma 5.6 are satisfied and hence

$$\sum_{n=N+1}^{N+S} \tau_k(n) \ll \frac{S}{\log N} \cdot \exp\left(\sum_{p \leq N} \frac{k}{p}\right).$$

By Mertens' second theorem [Mer74; Vil05], $\exp(\sum_{p\leq N} p^{-1}) \sim \exp(\log\log N) = \log N$, so that by Lemma 5.5

$$\sum_{n=N+1}^{N+S} \tau_k(n) \ll S \log^{k-1} N \ll \Delta N \log^{k-1} N. \qquad \blacksquare$$

Using the above corollary, we obtain a time complexity of $O(\Delta N \log^2 N)$ to compute the error term.

5.1 Final estimates of the time and space complexity

To summarize the Fourier theoretic approach, we note that we used Vaughan's identity, discussed in Chapter 3, to find an expression for $\psi(N)$. Using this identity, it remained for us to consider the sum

$$\sum_{\substack{mdk \leq N \\ m,d \leq M}} \Lambda(m)\mu(d) = \sum_{n \leq N} (\Lambda_{\leq M} * \mu_{\leq M} * \mathbf{1})(n),$$

as we explained in Chapter 4. We then used a similar approach as in [HKM23], where it was observed that Dirichlet convolutions were similar enough to discrete convolutions so that the difference

$$\sum_{k \leq \bar{k}(N)} (\overline{\Lambda_{\leq M}} \star \overline{\mu_{\leq M}} \star \bar{\mathbf{1}})[k] - \sum_{n \leq N} (\Lambda_{\leq M} * \mu_{\leq M} * \mathbf{1})(n)$$

is given explicitly by

$$\sum_{\substack{d_1 d_2 d_3 > N \\ \bar{k}(d_1) + \bar{k}(d_2) + \bar{k}(d_3) \leq \bar{k}(N)}} \Lambda_{\leq M}(d_1)\mu_{\leq M}(d_2)\mathbf{1}(d_3). \tag{5.4}$$

We argued that it was possible to compute the necessary segmentation arrays in $O(M \log\log M + \Delta^{-1} \log N)$ operations and $O(\sqrt{M} + \Delta^{-1} \log N)$ memory. Then, by the use of methods discussed in Appendix A, we noted that it is possible to compute

$$\sum_{k \leq \bar{k}(N)} (\overline{\Lambda_{\leq M}} \star \overline{\mu_{\leq M}} \star \bar{\mathbf{1}})[k]$$

in $O(\Delta^{-1} \log^2 N)$ operations and $O(\Delta^{-1} \log N)$ memory.

Afterwards, it remained for us to compute the error term as given in (5.4), which was done using a naive approach. We argued that if a tuple (d_1, d_2, d_3) contributes to the error term, then the product $n = d_1 d_2 d_3$ lies in $(N, N+S]$, where $S = (2^{3\Delta} - 1) \cdot N \ll \Delta N$. By computing the prime factorizations of integers $n \in (N, N+S]$, we could compute all decompositions of integers in $(N, N+S]$ into three factors. For each explicit decomposition, we can then check the condition on the segmentation indices directly and find its corresponding value in negligible time, assuming we keep track of the prime factorizations of the individual factors in the decomposition. We showed that the error term can be computed in $O(\sqrt{N} \log\log N + \Delta N \log^2 N)$ operations and $O(\sqrt{N} \log N)$ memory.

As a result, we obtain a total time complexity of $O(\sqrt{N} \log\log N + \max\{\Delta^{-1}, \Delta N\} \log^2 N)$. To optimize the time bound, we take Δ to be of order $1/\sqrt{N}$, resulting in a time complexity of $O(\sqrt{N} \log^2 N)$ and a space complexity of $O(\sqrt{N} \log N)$.

Finally, we want to mention that it is possible to decrease the space complexity at the cost of a larger time complexity, which is also discussed in [HKM23]. For this, it is necessary to increase the order of Δ. However, in this case we also have to reconsider the method we used to factorize the integers in $(N, N+S]$, since its corresponding space complexity is also given by $O(\sqrt{N} \log N)$. Using Helfgott's adaptation of the factorization sieve discussed in [Hel20], which we also briefly mentioned in Section 2.3.1, we can adapt the choice of Δ so that for any fixed $0 < \varepsilon < \frac{1}{6}$, we can compute $\psi(N)$ using the Fourier-theoretic approach with time complexity $O(N^{1/2+\varepsilon})$ and space complexity $O(N^{1/2-\varepsilon})$, where the implicit constants may depend on ε.

Chapter 6

Implementation in C++

In this chapter we discuss details regarding the implementation of the methods described in previous chapters. We have implemented an algorithm in C++. The main reason for this choice is that it has been recognized to have better performance than other well-known programming languages, such as C# or Python, while still being relatively easy to use. C++ also seems to be the standard when implementing scientific computational algorithms. Our implementation can be found on GitHub.

Below we discuss the use of external libraries and we highlight certain parts of the implementation which have only been mentioned briefly in the previous chapters. We will also analyze the results we have obtained, focusing on bottlenecks in the algorithms for both of the described methods. Finally, we discuss several aspects which could be considered in the future.

6.1 External libraries

As we briefly mentioned in Chapter 4, some computations necessary to evaluate $\psi(N)$ need to be approximated in order to keep a reasonable time and space complexity. The standard C++ library does allow to store approximations using floating-point number types such as `double` and `long double`. However, on 64-bit processors, the precision of such approximations typically extends to roughly 15 digits. This does not suffice for our purposes. For example, suppose that we wish to compute $\psi(10^{10})$ up to 8 digits after the decimal. Since $\psi(N) \sim N$, this would require at least 18 digits of precision. As a result, using the standard data types provided in C++, it is not possible to get the desired precision. We also need to point out that the accuracy of floating-point numbers may decrease after performing many computations with them, which makes it desirable to work with 20 digits of precision at a minimum in this fictive example.

As a result, we require custom data types which allow for more precision. We decided to use Boost for this purpose. Boost is a collection of external C++ libraries covering different useful extensions of the standard C++ library. Boost provides a *multiprecision* library, which allows us to store approximations of real numbers with at least 100 digits of precision, which is more than enough for our purposes. We also use this library for its implementation of 128 bit integers to ensure certain computations with integers do not overflow.

Coincidentally, Boost also provides implementations of special math-related functions. One of these functions happens to be $x \mapsto \log \Gamma(x)$. We discussed in Chapter 4 that approximations of values of this functions are useful for the computation of $\psi(N)$ using both the elementary and the Fourier-theoretic approach. It is worth noting these implementations work in combination with the Multiprecision library we mentioned above.

Boost also provides methods to implement parallelization of the algorithm. Although we have not used this in our current implementation, this might be worth looking at in the future, which is why we discuss this option in more detail in Section 6.4.

6.2 Highlighted parts of the algorithm

6.2.1 Diophantine approximation

In Chapter 4 we described a method to compute $\psi(N)$ using elementary methods which was based on [HT23]. When we elaborated on the method, we briefly mentioned that we need to approximate the differential coefficient c_y by some fraction a_0/q where q was bounded by some constant, while a_0 is relatively prime with q. In this section we want to elaborate on the algorithm used in [HT23] used to compute these values.

To be a bit more precise with how accurate the fraction a_0/q should represent c_y, we note that in the corresponding chapter, q should be taken less than or equal to Q, where Q is some predefined constant, and the difference should satisfy $|c_y - a_0/q| \leq 1/(qQ)$. In Chapter 4, we mentioned that such an approximation is possible by Dirichlet's Approximation Theorem.

To find such an approximation, it helps to look at the *continued fraction representation* of c_y. For simplicity, we define a continued fraction to be an expression of the form

$$[a_0; a_1, a_2, \ldots, a_n] := a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots + \cfrac{1}{a_n}}}},$$

where $n \geq 0$ and $a_0, a_1 \ldots, a_n$ are rational numbers, with $a_1, \ldots, a_n \neq 0$. In fact, often it is desired to consider $a_0 \in \mathbb{Z}$ and $a_1, \ldots, a_n \in \mathbb{N}$, which ensures uniqueness if $a_n \neq 1$. We remark that this only allows us to define continued fractions of rational numbers. Since c_y is rational, this does not pose any problems.

The representation of c_y as a continued fraction gives rise to the so-called convergents: by writing $c_y = [a_0; a_1, \ldots, a_n]$, for $k = 0, 1, \ldots, n$ we define the k-th convergent of c_y as $[a_0; a_1, a_2, \ldots, a_k]$. It is known that the convergents of a rational number α essentially represent the best approximations of α as fractions with bounded denominator: if p/q represents a convergent of α, then there is no rational p'/q' with $q' \leq q$ such that $|\alpha - p'/q'| < |\alpha - p/q|$. This suggests that in order to obtain the desired approximation of c_y, we should iteratively compute the convergents of c_y and find their corresponding representation as a reduced fraction.

To obtain the continued fraction of $\alpha_0 = c_y$, we proceed as follows. We first set $a_0 = \lfloor \alpha_0 \rfloor$. If $\alpha_0 \in \mathbb{Z}$, then we are done. Otherwise, we define $\alpha_1 = 1/(\alpha_0 - a_0) > 1$, and note that

$$\alpha = a_0 + \frac{1}{\alpha_1}.$$

We can now set $a_1 = \lfloor \alpha_1 \rfloor$, and we can proceed as before by defining $\alpha_2 = 1/(\alpha_1 - a_1) > 1$ if $\alpha_1 \notin \mathbb{Z}$ and set $a_2 = \lfloor \alpha_2 \rfloor$, and so on. Since $\alpha_0 = c_y$ is rational, there will be a point at which $\alpha_n \in \mathbb{Z}$, in which case the continued fraction of α is given by $[a_0; a_1, a_2, \ldots, a_n]$.

It remains for us to convert convergents to fractions, for which we use the following lemma.

Lemma 6.1. *Let $a_0 \in \mathbb{Q}$ and let $a_1, a_2, \ldots, a_n \in \mathbb{Q}_{\neq 0}$. Write $p_{-2} = 0, p_{-1} = 1$, $q_{-2} = 1$ and $q_{-1} = 0$. Then for $0 \leq k \leq n$,*

$$[a_0; a_1, a_2, \ldots, a_k] = \frac{p_k}{q_k}, \tag{6.1}$$

where we define $p_k = a_k \cdot p_{k-1} + p_{k-2}$ and $q_k = a_k \cdot q_{k-1} + q_{k-2}$ recursively.

Proof. We proceed by induction on k. We observe that

$$\frac{p_0}{q_0} = \frac{a_0 p_{-1} + p_{-2}}{a_0 q_{-1} + q_{-2}} = a_0,$$

which indeed corresponds with the first convergent of $[a_0; a_1, a_2, \ldots, a_n]$. To handle the general case, we note that $[a_0; a_1, a_2, \ldots, a_{k-1}, a_k, a_{k+1}] = [a_0; a_1, a_2, \ldots, a_{k-1}, (a_k a_{k+1} + 1)/a_{k+1}]$. As a result, if we are to assume that the induction hypothesis holds for some $k \leq n-1$, we can substitute a_k for $(a_k a_{k+1} + 1)/a_{k+1}$ to obtain that

$$\begin{aligned}
[a_0; a_1, a_2, \ldots, a_{k-1}, a_k, a_{k+1}] &= \frac{p_{k-1} \cdot (a_k a_{k+1} + 1)/a_{k+1} + p_{k-2}}{q_{k-1} \cdot (a_k a_{k+1} + 1)/a_{k+1} + q_{k-2}} \\
&= \frac{(a_k a_{k+1} + 1) \cdot p_{k-1} + a_{k+1} \cdot p_{k-2}}{(a_k a_{k+1} + 1) \cdot q_{k-1} + a_{k+1} \cdot q_{k-2}} \\
&= \frac{a_{k+1}(a_k p_{k-1} + p_{k-2}) + p_{k-1}}{a_{k+1}(a_k q_{k-1} + q_{k-2}) + q_{k-1}} = \frac{a_{k+1} p_k + p_{k-1}}{a_{k+1} q_k + q_{k-1}} = \frac{p_{k+1}}{q_{k+1}},
\end{aligned}$$

by applying the definitions of p_k, q_k, p_{k+1} and q_{k+1}. We have thus shown that the induction step holds, which concludes the proof. ■

In conclusion, we obtain the following algorithm.

Algorithm 1 Rational Approximation Algorithm (Cf. [HT23])

```
Input: $\alpha \in \mathbb{R}$, $Q \in \mathbb{Z}_{\geq 1}$.
Output: $(p, q) \in \mathbb{Z}^2$ such that $|\alpha - p/q| \leq (qQ)^{-1}$ with $\gcd(p, q) = 1$.
  function ApprByFrac($\alpha$, $Q$)
      $p \leftarrow [0, 1]$
      $q \leftarrow [1, 0]$
      while $q_1 \leq Q$ do
          $a \leftarrow \lfloor \alpha \rfloor$
          $p \leftarrow [p_1, a \cdot p_1 + p_0]$
          $q \leftarrow [q_1, a \cdot q_1 + q_0]$
          if $\alpha = a$ and $q_1 \leq Q$ then
              return $(p_1, q_1)$
          else if $\alpha = a$ then
              return $(p_0, q_0)$
          end if
          $\alpha \leftarrow 1/(\alpha - a)$
      end while
      return $(p_0, q_0)$
  end function
```

6.2.2 Computation of the segmentation error

In Chapter 5 we split the computation of $\psi(N)$ up into two parts. One part consists of the computation of some kind of convolution using the Fast Fourier Transform, as explained in Appendix A. For the other part we considered naively computing the segmentation error, i.e.,

$$\sum_{\substack{N<d_1d_2d_3\leq N+S \\ \bar{k}(d_1)+\bar{k}(d_2)+\bar{k}(d_3)\leq \bar{k}(N)}} \Lambda_{\leq M}(d_1)\mu_{\leq M}(d_2)\mathbf{1}(d_3).$$

In this subsection, we go over some of the details of the implementation of this part.

To briefly summarize what we meant by naive computation of the segmentation error; we first find the prime factorization for each the integers in $(N, N+S]$. Using the prime factorization of a fixed integer $n \in (N, N+S]$, we can obtain all decompositions of n into three factors, corresponding to the divisors d_1, d_2 and d_3 appearing in the sum above. For each decomposition (d_1, d_2, d_3) of n, we finally check if the condition regarding the segmentation indices is satisfied. If so, we add the corresponding term to the error.

The prime factorizations of integers $n \in (N, N+S]$ can be computed using one of the computational sieves discussed in Chapter 2. From now on, fix an integer $n \in (N, N+S]$ and suppose that n has prime factorization $p_1^{e_1} \cdots p_t^{e_t}$. Then to obtain the decompositions of n into three factors, we can make use of the following observation: a decomposition (d_1, d_2, d_3) of n, where d_i has prime factorization $\prod_{i=1}^{t} p_i^{e_{ij}}$ for $j = 1, 2, 3$, is uniquely characterized by the property that for each $i \in \{1, \ldots, t\}$, we have that $e_{i1} + e_{i2} + e_{i3} = e_i$. The subdivisions of the exponents e_i into three parts can be computed recursively using the following algorithm by considering $k = 3$.

Algorithm 2 Subdivisions of exponent e into k parts (non-negative integers).

```
Input: $e, k \in \mathbb{Z}_{\geq 1}$.
Output: The list of all $f_i = \{e_{i1}, \ldots, e_{ik}\}$ such that $\sum_{j=1}^{k} e_{ij} = e$ with $e_{ij} \geq 0$ for $j = 1, \ldots, k$.
  function SubdivideExponents($e$, $k$)
      $E \leftarrow \emptyset$
      if $k = 1$ then
          $E \leftarrow \{\{e\}\}$
      else
          for $a = 0, 1, \ldots, e$ do        ▷ The last entry gets value $a$, subdivide $e - a$ into $k - 1$ parts.
              $F \leftarrow$ SubdivideExponents($e - a, k - 1$)
              for $f \in F$ do
                  $E \leftarrow E \cup \{f \cup \{a\}\}$
              end for
          end for
      end if
      return $E$
  end function
```

We note that the number of operations in the above computation is majorized by the number of elements in the outputted list.

As the Möbius function is multiplicative and satisfies $\mu(p^2) = 0$ for any prime p, we can save time and space by only considering subdivisions of exponents whose corresponding part is either 0 or 1. While this idea has been incorporated in the actual implementation, we have neglected this component in the above algorithm as it is a relatively minor addition.

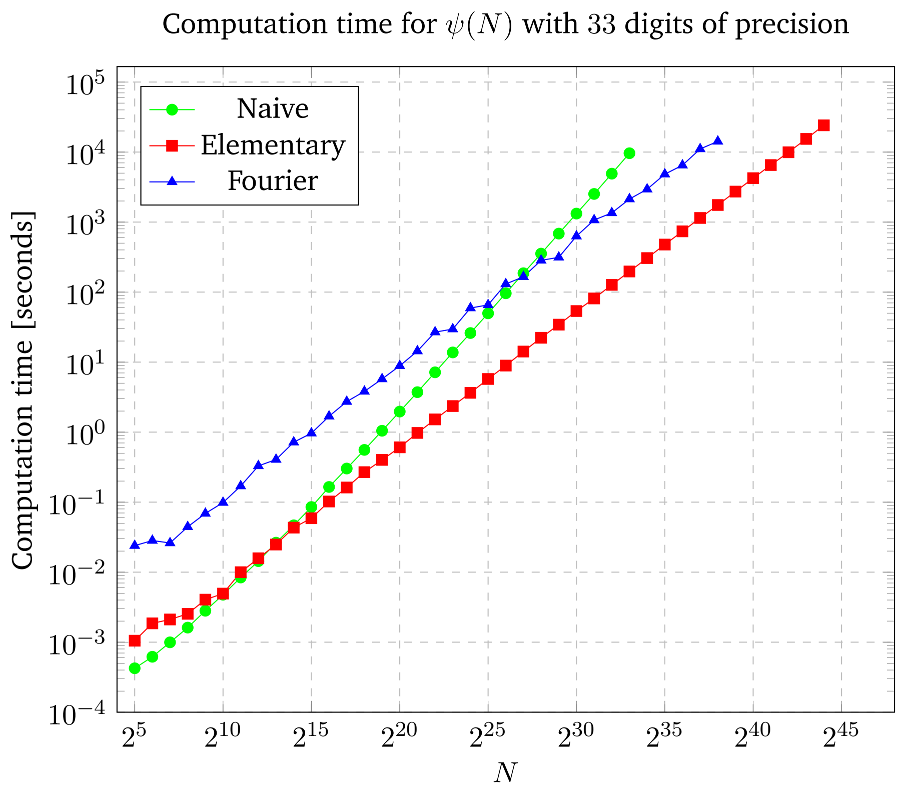
The remaining part of the algorithm is to merge the subdivisions of the exponents corresponding to different prime factors to obtain the desired decompositions (d_1, d_2, d_3). Then, for a fixed decomposition, we need to check if it satisfies the constraint on the segmentation indices and if it does, add its corresponding contribution to the error term. While the process is relatively involved, it does not add much to talk about it in more detail here.

6.3 Discussion of the results

In Appendix B, we have organized our results in tables. Below we analyze these results in more detail, mainly focusing on the computation time.

In the following graph, we compare the computation times of the elementary and the Fourier-theoretic method. We chose to include a naive implementation for reference. We recall from Chapter 4 that the naive way to compute $\psi(N)$ was to determine the primes up to N and use these to evaluate $\psi(N)$ directly using the identity

$$\psi(N) = \sum_{p \leq N} \lfloor \log_p(N) \rfloor \log(p).$$



The above graph suggests that the elementary method outperforms the Fourier-theoretic method. The basic explanation for this phenomenon is that the implicit constant appearing in the algorithm for the Fourier-theoretic method is large in comparison with the factor corresponding to the elementary method. This becomes apparent when closely looking at the graph, where we notice that even for small values of N, the Fourier-theoretic method performs poorly in contrast with both the elementary and the naive approach.

The significantly larger implicit constant for the Fourier-theoretic method can partially be explained by noting that our implementation of the FFT algorithm is very basic. We already illustrated this in Appendix A when we explicitly mentioned that it only operates on arrays of lengths which are powers of 2. In the worst case, this leads to double the time complexity. We observe that this also explains the bumps we see in the corresponding plot: some values of N simply result in advantageous sizes of the corresponding arrays to be convolved. Another contributing factor for the bad performance by the Fourier-theoretic method is the fact that we need to operate with approximations during a significant proportion of the computation. In contrast, the elementary method (and the naive method, for that matter) enables us to defer the use of approximations until the last stage where we compute its contribution to the sum.

However, as should be expected, the Fourier-theoretic method will eventually outperform the elementary method. By trying the fit the crude estimates of the time complexities corresponding to both methods, the turning point seems to be around $N = 2^{85}$. We do not expect this turning point to be possible to achieve in the near future. However, if it is somehow possible to increase the performance of the Fourier-theoretic method, we might be able to decrease the turning point so that it can actually be achieved. In Section 6.4 we discuss some potential improvements we could consider in the future.

6.4 Future considerations

6.4.1 Parallelization

It has become common for computers to have multiple processing units. This allows for parallel execution of code, which in turn results in an increase in performance. The method to implement parallel execution is often referred to as *parallelization*. It essentially reduces the time needed to complete a task simply by using more processing power and memory.

It is possible to implement parallelization in C++ using Boost's Thread library. For our purposes, *threads* can be thought of objects managed by the different processing units. For the elementary method, we could consider parallelization of the computation of the sums corresponding to the dependent and independent variable cases. For both cases, we could consider the use of so-called *thread pools*. In essence, a thread pool is a collection of threads following instructions from a central component. The central component keeps track of a list of moderate tasks which it hands out to the threads. If a thread completes a task, it reports the results back to the central component, and then potentially gets assigned a new task. This process continues until there are no tasks left.

To describe how thread pools can be used to compute the sum corresponding to the dependent variable case, we note that the most time is spent on the computation of the sums of divisor sums $\sum_{n\leq N/a} D_g(n,a)$ for $a = M_0+1,\ldots,M$, as present in (4.5). As a result, it might be useful to consider tasks such as the computation of

$$\sum_{N/(a+1)<n\leq N/a} D_g(n,a),$$

for $a = M_0+1,\ldots,M$.

Similarly, for the independent variable case, the computation of

$$\sum_{m,d \leq M_0} \Lambda(m)\mu(d) \left\lfloor \frac{N}{md} \right\rfloor$$

is split over dyadic regions $R(A,B)$, which are further subdivided into fixed rectangles $I_x \times I_y$. As a result, given a region $R(A,B)$, it is natural to consider tasks which compute the contribution corresponding of an individual rectangle $I_x \times I_y \subset R(A,B)$. However, for smaller regions $R(A,B)$, it may be better to consider tasks computing the contribution over the whole region $R(A,B)$ directly. This is explained by noting that the introduction of many trivial tasks may lead to threads needing to wait significant amounts of time before getting assigned tasks and/or report their contributions to the central component, which could nullify the purpose of threads.

For the Fourier-theoretic approach, most time is spent on the computation of discrete Fourier transformations. As a result, to parallelize this approach, we should focus on parallelization of the FFT algorithm. Note that the FFT algorithm was essentially based on repeatedly splitting the computation into two parts. As a result, we could consider running multiple of these smaller parts in parallel. For example, in a scenario where we have 4 processing units and we are considering an array of length 2^n with $n \geq 2$, we could run the FFT algorithm in parallel on inputs of size 2^{n-2}. In a more general setting, we should be able to apply similar ideas.

6.4.2 Number theoretic transformations

In Chapter 5, we described a method to compute $\psi(N)$ partially via the evaluation of the convolution $\overline{\Lambda_{\leq M}} \star \overline{\mu_{\leq M}} \star \overline{\mathbf{1}}$. We have also related convolutions with discrete Fourier transformations (DFTs) in Appendix A. However, we could also take a different approach. Instead of relating convolutions with DFTs, under some circumstances it is also possible to relate convolutions to *number theoretic transformations*, abbreviated to NTTs. In fact, the following idea is also used in [HKM23], which is the main paper we used to develop the method described in Chapter 5.

Below, $N \geq 1$ is a fixed integer assumed to be a power of 2. We also let p be a (large) prime number such that N divides $p-1$, so that there exists a primitive N-th root ζ_N in $(\mathbb{Z}/p\mathbb{Z})^\times$.

Definition 6.2. *Consider a function $f : \{0,1,\ldots,N-1\} \to \mathbb{Z}/p\mathbb{Z}$. We define the NTT of f as the function $\mathcal{N}f : \{0,1,\ldots,N-1\} \to \mathbb{Z}/p\mathbb{Z}$ via*

$$\mathcal{N}f(n) = \sum_{k=0}^{N-1} f(k) \cdot \zeta_N^{kn}.$$

If we compare the definition of NTTs with the definitions of DFTs (present in Appendix A, we see that they have a lot in common. In fact, the lemmas provided for DFTs in Appendix A can also be shown to hold for NTTs, with identical proofs. In particular, it can be shown that if we are given two functions $f, g : \{0,1,\ldots,N-1\} \to \mathbb{Z}/p\mathbb{Z}$, then for any $n \in \{0,1,\ldots,N-1\}$, $\mathcal{N}(f \star g)(n) = (\mathcal{N}f)(n) \cdot (\mathcal{N}g)(n)$, where equality should be taken modulo p. This in turn can be used to consider convolutions of functions $f, g : \{0,1,\ldots,N-1\} \to \mathbb{Z}$, by considering their images modulo a sufficiently large prime p of the form $kN+1$ for some $k \in \mathbb{N}$.

As explained above, NTTs only make sense for functions whose image lies in $\mathbb{Z}$. Hence, it does not make sense to consider the NTT of $\Lambda_{\leq M}$, for example. This is why we originally decided to take a different approach as discussed in [HKM23]. However, the advantage of NTTs over DFTs is that they work on integers instead of floating-point numbers. This makes partial results stay exact, leaving less room for error. Moreover, integers require significantly less memory to store. Hence, it may be useful to look into this further.

The main idea on how to develop an approach involving NTTs is the following. As the entries of the segmentation arrays $\overline{\mu_{\leq M}}$ and $\overline{\mathbf{1}}$ are integers, it is possible to consider using an algorithm computing the convolution $\overline{\mu_{\leq M}} \star \overline{\mathbf{1}}$ using NTTs. Now, noting that we want to compute

$$\sum_{k \leq \overline{k}(N)} (\overline{\Lambda_{\leq M}} \star \overline{\mu_{\leq M}} \star \overline{\mathbf{1}})[k] = \sum_{k_1 + k_2 \leq \overline{k}(N)} \overline{\Lambda_{\leq M}}[k_1](\overline{\mu_{\leq M}} \star \overline{\mathbf{1}})[k_2],$$

we could proceed by rearranging terms:

$$\sum_{k_1 + k_2 \leq \overline{k}(N)} \overline{\Lambda_{\leq M}}[k_1](\overline{\mu_{\leq M}} \star \overline{\mathbf{1}})[k_2] = \sum_{k \leq \overline{k}(N)} (\overline{\mu_{\leq M}} \star \overline{\mathbf{1}})[k] \sum_{m \leq \overline{k}(N) - k} \overline{\Lambda_{\leq M}}[m]. \tag{6.2}$$

By keeping track of the partial sum

$$\sum_{m \leq K} \overline{\Lambda_{\leq M}}[m] = \sum_{m < 2^{(K+1)\Delta}} \Lambda_{\leq M}(m),$$

for $K = 0, 1, \dots, \overline{k}(N)$, we can compute the sum in (6.2) by iterating over k in descending order. As a result, we observe that it is possible to use NTTs without changing the order of the time and space complexity corresponding to the Fourier-theoretic approach. In fact, we expect this method to perform better by a constant factor, as it does not rely as much on computations with floating-point numbers requiring many digits of precision.

6.4.3 External library for computing discrete Fourier transformations

Because of the inefficiency of the FFT algorithm we provide, we have already considered the use of an external library specialized in the computation of discrete Fourier transformations. However, popular choices for performance such as FFTW, KFR or oneMKL only work with standard data types such as `floats` and `doubles`. As we already discussed in Section 6.1, without changing the method used to evaluate $\psi(N)$ in some capacity, we cannot achieve the desired level of precision using these data types.

That being said, in Section 6.4.2 we introduced an alternative approach that involves computing convolutions on integer-valued arrays. While this approach is used to motivate the use of a different kind of transformation, this approach also prompts us to reconsider the use of one of the external libraries mentioned above. To elaborate on why, we note that can store entries of integer valued arrays using `doubles` or even `floats`, assuming that these integers are not excessively large. Since the convolution of two integer-valued arrays is again integer-valued, we can round the entries of the array after performing convolutions. However, we note that for large arrays, it is possible for values to be rounded incorrectly. This is something to keep in mind when implementing this approach, as the corresponding errors they introduce will likely be significant.

Bibliography

[Apo13] T. M. Apostol. *Introduction to Analytic Number Theory*. 1st ed. Undergraduate Texts in Mathematics. Springer New York, NY, 2013. ISBN: 978-1-4757-5579-4. DOI: 10.1007/978-1-4757-5579-4.

[BJ99] P. Butzer and F. Jongmans. "P. L. Chebyshev (1821–1894): A Guide to his Life and Work". In: *Journal of Approximation Theory* 96.1 (1999), pp. 111–138. ISSN: 0021-9045. DOI: https://doi.org/10.1006/jath.1998.3289. URL: https://www.sciencedirect.com/science/article/pii/S0021904598932890.

[Che52a] P. L. Chebyshev. "Mémoire sur les nombres premiers". In: *Journal de Mathématiques Pures et Appliquées* 17.1 (1852), pp. 366–390.

[Che52b] P. L. Chebyshev. "Sur la fonction qui détermine la totalité des nombres premiers inférieurs à une limite donnée". In: *Journal de Mathématiques Pures et Appliquées* 17.1 (1852), pp. 341–365.

[Cor+09] T. H. Cormen et al. *Introduction to Algorithms*. 3rd ed. The MIT Press, 2009. ISBN: 978-0-262-03384-8.

[DR96a] M. Deléglise and J. Rivat. "Computing $\pi(x)$: The Meissel, Lehmer, Lagarias, Miller, Odlyzko Method". In: *Mathematics of Computation* 65.213 (1996). URL: https://www.jstor.org/stable/2153843.

[DR96b] M. Deléglise and J. Rivat. "Computing the summation of the Möbius function." In: *Experimental Mathematics* 5.4 (1996), pp. 291–295. DOI: 10.1080/10586458.1996.10504594.

[DR98] M. Deléglise and J. Rivat. "Computing $\psi(x)$". In: *Mathematics of Computation* 67.224 (1998), pp. 1691–1696. ISSN: 0025-5718.

[Hea82] D. R. Heath-Brown. "Prime Numbers in Short Intervals and a Generalized Vaughan Identity". In: *Canadian Journal of Mathematics* 34.6 (1982), pp. 1365–1377. DOI: 10.4153/CJM-1982-095-9.

[Hel20] H. A. Helfgott. "An improved sieve of Eratosthenes". In: *Mathematics of Computation* 89 (2020), pp. 333–350. DOI: 10.1090/mcom/3438.

[HKM23] D. Hirsch, I. Kessler, and U. Mendlovic. *Computing $\pi(N)$: An elementary approach in $\tilde{O}(\sqrt{N})$ time*. 2023. arXiv: 2212.09857 [math.NT].

[HT23] H. A. Helfgott and L. Thompson. "Summing $\mu(n)$: a faster elementary algorithm". In: *Research in Number Theory* 9.6 (2023). DOI: 10.1007/s40993-022-00408-8.

[IK04] H. Iwaniec and E. Kowalski. *Analytic Number Theory*. Providence, RI: American Mathematical Society, 2004. ISBN: 978-0-821-83633-0.

[KR06] T. Kotnik and H. J. J. te Riele. "The Mertens Conjecture Revisited". In: *Algorithmic Number Theory*. Ed. by F. Hess, S. Pauli, and M. Pohst. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 156–167. ISBN: 978-3-540-36076-6.

[Leg08] A. M. Legendre. *Essai sur la théorie des nombres*. Seconde édition. Accessible via https://www.google.com/books/edition/Essai_sur_la_th%C3%A9orie_des_nombres/5TEVAAAAQAAJ. Paris, 1808, pp. 412–423.

[Leh59] D. H. Lehmer. "On the exact number of primes less than a given limit". In: *Illinois Journal of Mathematics* 3.3 (1959), pp. 381–388. DOI: 10.1215/ijm/1255455259.

[LMO85] J. C. Lagarias, V. Miller, and A. M. Odlyzko. "Computing $\pi(x)$: The Meissel-Lehmer Method". In: *Mathematics of Computation* 44.170 (1985), pp. 537–560. DOI: 10.2307/2007973.

[LO87] J. C. Lagarias and A. M. Odlyzko. "Computing $\pi(x)$: An analytic method". In: *Journal of Algorithms* 8.2 (1987), pp. 173–191. ISSN: 0196-6774. DOI: https://doi.org/10.1016/0196-6774(87)90037-X. URL: https://www.sciencedirect.com/science/article/pii/019667748790037X.

[Mei70] E. D. F. Meissel. "Über die Bestimmung der Primzahlenmenge innerhalb gegebener Grenzen". In: *Mathematische Annalen* 2 (1870), pp. 636–642.

[Mei71] E. D. F. Meissel. "Berechnung der Menge von Primzahlen, welche innerhalb der ersten hundert Millionen natürlicher Zahlen vorkommen". In: *Mathematische Annalen* 3 (1871), pp. 523–525.

[Mei83] E. D. F. Meissel. "Über Primzahlenmengen". In: *Mathematische Annalen* 21 (1883), p. 304.

[Mei85] E. D. F. Meissel. "Berechnung der Menge von Primzahlen, welche innerhalb der ersten Milliarde natürlicher Zahlen vorkommen". In: *Mathematische Annalen* 25 (1885), pp. 289–292.

[Mer74] F. Mertens. "Ein Beitrag zur analytischen Zahlentheorie." In: *Journal für die reine und angewandte Mathematik* 78 (1874), pp. 46–62. DOI: 10.1515/crll.1874.78.46.

[NT98] M. Nair and G. Tenenbaum. "Short sums of certain arithmetic functions". In: *Acta Mathematica* 180 (1998), pp. 119–144. DOI: 10.1007/BF02392880.

[OR85] A. M. Odlyzko and H. J. J. te Riele. "Disproof of the Mertens conjecture." In: *Journal für die reine und angewandte Mathematik* 357 (1985), pp. 138–160. DOI: 10.1515/crll.1985.357.138.

[Pla13] D. J. Platt. *Computing $\pi(x)$ Analytically*. 2013. arXiv: 1203.5712 [math.NT].

[Shi80] P. Shiu. "A Brun-Titschmarsh theorem for multiplicative functions." In: *Journal für die reine und angewandte Mathematik* 313 (1980), pp. 161–170. DOI: 10.1515/crll.1980.313.161.

[Tem96] N. M. Temme. "The Gamma Function". In: *Special Functions: An Introduction to the Classical Functions of Mathematical Physics*. John Wiley & Sons, Ltd, 1996. Chap. 3, pp. 41–77. ISBN: 978-1-118-03257-2. DOI: 10.1002/9781118032572.ch3.

[Vau80] R. C. Vaughan. "An elementary method in prime number theory". In: *Acta Arithmetica* 37.1 (1980), pp. 111–115. DOI: 10.4064/aa-37-1-111-115.

[Vil05] M. B. Villarino. *Mertens' Proof of Mertens' Theorem*. 2005. arXiv: math/0504289 [math.HO].

Appendix A

Convolutions and the Fast Fourier Transform

In this appendix, we relate the computation of convolutions to the computation of discrete Fourier transformations and their inverses. We also describe a relatively basic algorithm to compute these transformations, known as the radix-2 Fast Fourier Transform (FFT). Finally, we will show that the corresponding time and space complexities are given by $O(N \log N)$ and $O(N)$ respectively.

A.1 Relating convolutions to discrete Fourier transformations

The definition of a convolution has been given in Chapter 3, but in order to make this chapter self-contained, we repeat the definition.

Definition A.1. *Given functions $f, g : \{0, 1, \ldots, N-1\} \to \mathbb{C}$, we define the convolution of f and g as the function $f \star g : \{0, 1, \ldots, N-1\} \to \mathbb{C}$ such that for $n \in \{0, 1, \ldots, N-1\}$,*

$$(f \star g)(n) = \sum_{n_1+n_2=n} f(n_1)g(n_2).$$

As suggested in the introduction of this appendix, we want to relate convolutions with discrete Fourier transformations. To elaborate on the relation, we give a definition of a discrete Fourier transformations. Below we let $\zeta_N = e^{2\pi i/N}$, so that ζ_N is a primitive N-th root of unity in $\mathbb{C}$.

Definition A.2. *Given a function $f : \{0, 1, \ldots, N-1\} \to \mathbb{C}$, we define its discrete Fourier transformation $\mathcal{F}f : \{0, 1, \ldots, N-1\} \to \mathbb{C}$ where for any $n \in \{0, 1, \ldots, N-1\}$,*

$$(\mathcal{F}f)(n) = \sum_{k=0}^{N-1} f(k)\zeta_N^{kn}.$$

Similarly, we define the inverse discrete Fourier transformations $\mathcal{F}^{-1}f : \{0, 1, \ldots, N-1\} \to \mathbb{C}$, where for any $n \in \{0, 1, \ldots, N-1\}$,

$$(\mathcal{F}^{-1}f)(n) = \frac{1}{N} \cdot \sum_{k=0}^{N-1} f(k)\zeta_N^{-kn}.$$

Lemma A.3. *For any function $f : \{0, 1, \ldots, N-1\} \to \mathbb{C}$ and for any $n \in \{0, 1, \ldots, N-1\}$, we have that*

$$\big(\mathcal{F}^{-1}(\mathcal{F}f)\big)(n) = f(n).$$

Proof. By definition, we have that

$$\big(\mathcal{F}^{-1}(\mathcal{F}f)\big)(n) = \frac{1}{N} \cdot \sum_{k=0}^{N-1} (\mathcal{F}f)(k)\zeta_N^{-kn} = \frac{1}{N} \cdot \sum_{k=0}^{N-1}\sum_{m=0}^{N-1} f(m)\zeta_N^{k(m-n)} = \frac{1}{N} \cdot \sum_{m=0}^{N-1} f(m) \left(\sum_{k=0}^{N-1} \zeta_N^{k(m-n)}\right).$$

As ζ_N is a primitive N-th root of unity, for $m \in \{0, 1, \ldots, N-1\}$ we have that $\sum_{k=0}^{N-1} \zeta_N^{k(m-n)} = N \cdot \delta_{mn}$, where δ_{mn} is equal to 1 if $m = n$ and 0 otherwise. As a result,

$$\big(\mathcal{F}^{-1}(\mathcal{F}f)\big)(n) = \sum_{m=0}^{N-1} f(m)\delta_{mn} = f(n). \qquad \blacksquare$$

Lemma A.4. *Consider functions $f, g : \{0, 1, \ldots, N-1\} \to \mathbb{C}$ and let $n \in \{0, 1, \ldots, N-1\}$. Then*

$$(\mathcal{F}f)(n) \cdot (\mathcal{F}g)(n) = \sum_{a=0}^{N-1} \left(\sum_{\substack{n_1, n_2 \in \{0, \ldots, N-1\} \\ n_1 + n_2 \equiv a \bmod N}} f(n_1)g(n_2) \right) \zeta_N^{an}.$$

Proof. We first note that

$$(\mathcal{F}f)(n) \cdot (\mathcal{F}g)(n) = \left(\sum_{k=0}^{N-1} f(k)\zeta_N^{kn}\right) \cdot \left(\sum_{l=0}^{N-1} g(l)\zeta_N^{ln}\right).$$

Writing $\bar{l} = l \bmod N$, the function $\hat{g} : \mathbb{Z} \to \mathbb{C}, l \to g(\bar{l})\zeta_N^{ln}$ is N-periodic, it follows that for fixed $k \in \mathbb{Z}$,

$$\sum_{l=0}^{N-1} g(l)\zeta_N^{ln} = \sum_{l=0}^{N-1} g(\overline{l-k})\zeta_N^{(l-k)n}.$$

In particular,

$$\left(\sum_{k=0}^{N-1} f(k)\zeta_N^{kn}\right) \cdot \left(\sum_{l=0}^{N-1} g(l)\zeta_N^{ln}\right) = \sum_{k=0}^{N-1}\sum_{l=0}^{N-1} f(k)g(\overline{l-k})\zeta_N^{ln} = \sum_{a=0}^{N-1} \left(\sum_{k=0}^{N-1} f(k)g(\overline{a-k})\right) \zeta_N^{an}.$$

By rewriting the inner summation, we derive the desired identity. $\blacksquare$

Note that the previous lemma almost relates the product of two Fourier transformations of two functions to the Fourier transformation of their convolution. Using a process called *padding zeros*, this relation can be made explicit. We state the result in the following corollary.

Corollary A.5. *Given functions $f, g : \{0, 1, \ldots, N-1\} \to \mathbb{C}$, define functions $\tilde{f}, \tilde{g} : \{0, 1, \ldots, 2N-2\} \to \mathbb{C}$ such that for $h \in \{f, g\}$ and for $n \in \{0, 1, \ldots, 2N-2\}$,*

$$\tilde{h}(n) = \begin{cases} h(n) & \text{if } n < N, \\ 0 & \text{otherwise.} \end{cases}$$

Then $\mathcal{F}(f \star g) = (\mathcal{F}\tilde{f})(\mathcal{F}\tilde{g})$. That is, for any $n \in \{0, 1, \ldots, 2N-2\}$, $\big(\mathcal{F}(f \star g)\big)(n) = (\mathcal{F}\tilde{f})(n) \cdot (\mathcal{F}\tilde{g})(n)$.

This relation can be extended to chains of convolutions. We state the general result in another corollary.

Corollary A.6. *Given functions* $f_1, \dots, f_k : \{0, 1, \dots, N-1\} \to \mathbb{C}$*, for* $i = 1, \dots, k$*, we define functions* $\widetilde{f_i} : \{0, 1, \dots, k(N-1)\} \to \mathbb{C}$ *similar as in Corollary A.5. Then*

$$\mathcal{F}(f_1 \star \dots \star f_k) = \prod_{i=1}^{k} \mathcal{F}\widetilde{f_i}.$$

In short, the strategy to compute values of a chain of convolutions $f_1 \star \dots \star f_k$, we first compute the discrete Fourier transformations of the functions $\widetilde{f_i}$ for $i = 1, \dots, k$. We then compute the values of the functions $g : \{0, 1, \dots, k(N-1)\} \to \mathbb{C}$, where for $n = 0, 1, \dots, k(N-1)$,

$$g(n) = \prod_{i=1}^{k} (\mathcal{F}\widetilde{f_i})(n).$$

Finally, by performing the inverse transformation, using Lemma A.3 we obtain that

$$(f_1 \star \dots \star f_k)(n) = (\mathcal{F}^{-1} g)(n).$$

A.2 Computation of discrete Fourier transformations

In the previous section we showed that convolutions are related to discrete Fourier transformations (DFTs). In this section, we elaborate on a relatively basic method used to compute DFTs on arrays of size N in $O(N \log N)$ time and $O(N)$ space. Recall that such an algorithm is called a Fast Fourier Transform (FFT).

On a surface level, the idea is to compute the DFT of size N via a reduction to two DFTs of size $N/2$, in a way such that we can easily deduce the original DFT by combining the results of the two smaller DFTs. The two smaller DFTs are then to be computed using recursion. For the recursion to work, we will assume that N is a power of 2. The idea is made more explicit in the following diagram.

$$\begin{tikzcd}
 & \boxed{[a_0, a_1, a_2, \dots, a_{N-1}]} \arrow[dl] \arrow[dr] & \\
\boxed{[a_0, a_2, a_4, \dots, a_{N-2}]} \arrow[d, "\text{FFT}"'] & & \boxed{[a_1, a_3, a_5, \dots, a_{N-1}]} \arrow[d, "\text{FFT}"'] \\
\boxed{[b_0, b_2, b_4, \dots, b_{N-2}]} \arrow[dr] & & \boxed{[b_1, b_3, b_5, \dots, b_{N-1}]} \arrow[dl] \\
 & \boxed{[c_0, c_1, c_2, \dots, c_{N-1}]} &
\end{tikzcd}$$

Figure A.1: Diagram of the radix-2 FFT algorithm.

In the diagram above, we use the correspondence between a function $f : \{0, 1, \ldots, M\} \to \mathbb{C}$ and an array $[a_0, \ldots, a_M]$ with $a_n = f(n)$ for any $n \in \{0, 1, \ldots, M\}$. If we continue using the notation as in the diagram, it suffices to find formulas for the coefficients $c_0, \ldots, c_{N-1}$ in terms of $b_0, \ldots, b_{N-1}$. Hence, let us write out the coefficients $b_0, \ldots, b_{N-1}$ and $c_0, \ldots, c_{N-1}$. By definition, for any $n \in \{0, 1, \ldots, N-1\}$, we should have that

$$c_n = \sum_{k=0}^{N-1} a_k \cdot \zeta_N^{kn},$$

while for $n = 0, 1, \ldots, N/2 - 1$, it is given that

$$b_{2n} = \sum_{k=0}^{N/2-1} a_{2k} \cdot \zeta_{N/2}^{kn} \quad \text{and} \quad b_{2n+1} = \sum_{k=0}^{N/2-1} a_{2k+1} \cdot \zeta_{N/2}^{kn}.$$

Noting that $\zeta_{N/2} = \zeta_N^2$, we make the observation that $c_n = b_{2n} + \zeta_N^n \cdot b_{2n+1}$ for $n \in \{0, 1, \ldots, N/2 - 1\}$. To obtain the coefficients c_n for $n \in \{N/2, N/2 + 1, \ldots, N - 1\}$, we observe that $\zeta_N^{N/2} = -1$, so that by writing $n = N/2 + m$ with $m \in \{0, 1, \ldots, N/2 - 1\}$, it follows that

$$c_n = \sum_{k=0}^{N-1} a_k \cdot \zeta_N^{k(N/2+m)} = \sum_{k=0}^{N/2-1} a_{2k} \cdot \zeta_N^{2km} - \sum_{k=0}^{N/2-1} a_{2k+1} \cdot \zeta_N^{(2k+1)m} = b_{2m} - \zeta_N^m \cdot b_{2m+1}.$$

Having established identities for the coefficients $c_0, \ldots, c_{N-1}$ in terms of $b_0, \ldots, b_{N-1}$, we obtain the following algorithm.

Algorithm 3 Radix-2 FFT

```
Require: $[a_0, a_1, \ldots, a_{N-1}]$ is an array of complex numbers, where $N$ is a power of 2.
Output: $[c_0, c_1, \ldots, c_{N-1}] = \mathcal{F}([a_0, a_1, \ldots, a_{N-1}])$
  function FFT($[a_0, a_1, \ldots, a_{N-1}]$)
      if $N = 1$ then
          $c_0 = a_0$
      else
          $[b_0, b_2, \ldots, b_{N-2}] \leftarrow$ FFT($[a_0, a_2, \ldots, a_{N-2}]$)
          $[b_1, b_3, \ldots, b_{N-1}] \leftarrow$ FFT($[a_1, a_3, \ldots, a_{N-1}]$)
          for $n = 0, 1, \ldots, N/2 - 1$ do
              $c_n \leftarrow b_{2n} + \zeta_N^n \cdot b_{2n+1}$
              $c_{n+N/2} \leftarrow b_{2n} - \zeta_N^n \cdot b_{2n+1}$
          end for
      end if
      return $[c_0, c_1, \ldots, c_{N-1}]$
  end function
```

We can use the same approach to compute inverse discrete Fourier transformations (IDFTs). The main difference is that we should now replace ζ_N with ζ_N^{-1}. Additionally, the final output should be scaled by a factor of $1/N$. Remark that this indeed returns the desired inverse transformation, by using Definition A.2 and Lemma A.3.

We now briefly analyze the time and space complexity of the algorithm. It should be clear that the space complexity of the algorithm, when applied to arrays of size N, is $O(N)$; when implemented correctly, no more than $2N$ entries need to be stored at any time. The analysis for the time complexity is a bit more involved. If we are to write $F(N)$ for the time complexity of the algorithm applied to arrays of size N, then we have the following recurrence relation for $F(N)$: $F(1) \ll 1$ and given that N is a proper power of 2, then

$$F(N) = 2 \cdot F(N/2) + f(N),$$

where $f(N) \ll N$. As a result, using induction, we have that for any $k \geq 1$,

$$F(2^k) = 2^k \cdot F(1) + \sum_{l=0}^{k} f(2^l) \cdot 2^{k-l} \ll 2^k + \sum_{l=0}^{k} 2^k \ll k \cdot 2^k,$$

implying the bound $F(N) = O(N \log N)$, as N was assumed to be a power of 2. We note that for the analysis of the time complexity, we have essentially considered a special case of [Cor+09, Thm 4.1].

Finally, we remark that we can extend the above algorithm to any positive integer N, although the algorithm becomes less efficient in the general case. The reason for this is that given the size of the array is not a power of 2, we can extend its size to be a power of 2 by padding zeros at the end. It should be mentioned that this does not affect the DFT. At worst, this results in double the time complexity and space complexity, so the order of the complexities stay the same.

Appendix B

Results

To analyze the computation time of the elementary and the Fourier-theoretic method, we have computed values of ψ at powers of 2 and 10. These tables may also serve as a verification of the implementations, as the given approximations can be compared with known values, such as those present in [DR98]. However, we note that in [DR98], approximations are done using 33 digits of precision, of which they discard the last 12 digits to ensure correctness of the results.

To verify relatively small results up to 100 digits, we note that for any integer $N \geq 1$,

$$\psi(N) = \log \operatorname{lcm}(1, 2, \dots, N).$$

Hence, using a table of values for $\operatorname{lcm}(1, 2, \dots, N)$, it is possible to find better approximations of $\psi(N)$ for relatively small values of N. We note that a table of values of $\operatorname{lcm}(1, 2, \dots, N)$ for $N = 1, \dots, 2308$ can be found at OEIS.

Below we provide computations with both 33 digits of precision, as well as with 100 digits. As we present our results with the number of digits that the specified precision would allow, it is possible that the final digits of results may slightly misrepresent the digits that should appear in the values of ψ. This is illustrated, for example, when comparing the results for 33 digits and 100 digits. We note that, generally, it seems that at worst the two last digits of an approximation are incorrect.

We note that the Fourier-theoretic method yields incorrect results for $N = 10^{10}$. The precise cause of this issue is currently unknown, but it is possible that it is caused due to limitations of floating-point numbers. More specifically, the Fourier-theoretic approach occasionally works with the quantity $2^{k\Delta}$ for $k = 0, 1, \dots, \lfloor \Delta^{-1} \log_2 N \rfloor$, where we recall that Δ is of order $1/\sqrt{N}$. This is used to determine the integers between $2^{k\Delta}$ and $2^{(k+1)\Delta}$, which are needed to compute segmentation arrays. Due to rounding errors, it is possible that some integers were placed in the incorrect interval, leading to incorrect results.

B.1 Elementary method

Table B.1: Approximations of ψ at powers of 10 with 33 digits of precision.

n	$\psi(10^n)$	Time (in seconds)
2	94.0453112293573922460049312446068	0.002
3	996.680912247175240263021765666415	0.006
4	10013.3966932631147837203245944702	0.032
5	100051.564025657954250923140327612	0.135
6	999586.597495632922033061533011302	0.593
7	9998539.40334597536635289816238934	2.620
8	99998242.7966267823416068404575383	11.697
9	1000001595.99042758043906519942950	50.798
10	10000042119.8334736147597841157928	217.679
11	100000058456.430302189943653434057	931.892
12	1000000040136.76545665644667959163	3989.998
13	10000000171997.1232250780423641037	16774.599

Table B.2: Approximations of ψ at powers of 10 with 100 digits of precision.

n	$\psi(10^n)$	Time (in seconds)
2	94.045311229357392246004931244606927241326097311452⋱ 85310924864280321045214810768851301599372780732813	0.01
3	996.68091224717524026302176566642154166577843690213⋱ 98669138260098914052791521458932321639080369652377	0.04
4	10013.396693263114783720324594470187567072560129718⋱ 13945674321991600537980308750231337336978099304402	0.23
5	100051.56402565795425092314032761162712492914624321⋱ 57729289419878000282687266030514746601286515877966	0.94
6	999586.59749563292203306153301130446359633577788026⋱ 41352302044700225170939094028835979120742512656846	4.34
7	9998539.4033459753663528981623893319752768018964902⋱ 11932942941683953079499316238810603994703425846425	19.31
8	99998242.796626782341606840457535809044522868171269⋱ 60114125131028399761689495560119337885164360379220	85.97
9	1000001595.9904275804390651994295128367679624178605⋱ 68112514510678741800038392277688681944982118559046	381.60
10	10000042119.833473614759784115792952972441071115933⋱ 08179000606626316872889077116588928477095285056352	1621.68
11	100000058456.43030218994365343405605601334959367638⋱ 28079756527152212390844097427799329030625778294828	6988.50
12	1000000040136.7654566564466795916283073918337620519⋱ 32367998351419644089321058042127344614800791898687	29350.63

Table B.3: Approximations of ψ at powers of 2 with 33 digits of precision.

n	$\psi(2^n)$	Time (in seconds)
6	62.337231194609467866435095694222 2	0.002
7	126.93139272208268541104895442978 7	0.002
8	251.49307009837428804173436640792 2	0.003
9	514.79830201695784316277014489216 1	0.004
10	1025.0665608037248682785945422220 9	0.005
11	2040.8127076788827101462589263191 1	0.010
12	4106.6333406235809397129732624231 4	0.016
13	8176.9694252202565289050672910128 6	0.025
14	16408.514900137643900575095227371 8	0.043
15	32736.919484998285805644028484768 0	0.059
16	65466.400464967506771575265219488 9	0.102
17	131090.52025236027320915005828493 3	0.162
18	262034.92911613517461500460243937 2	0.268
19	524474.32575285085668403952735031 7	0.401
20	1048430.4214037888280011160947399 1	0.607
21	2097211.8618086878845755242502579 2	0.974
22	4194440.8768981085818529262831202 1	1.520
23	8388383.5420935264665780342158138 8	2.357
24	16776556.289988168062865962177767 8	3.642
25	33556481.823201979752504266157850 7	5.741
26	67109113.464223632253091727646765 5	8.920
27	134216112.71728191370763236011176 0	14.153
28	268436833.42388429769930952869079 9	22.267
29	536866944.47968824401909839340739 6	34.281
30	1073749873.2323113687037487425586 8	53.622
31	2147479854.2280562249344632989709 0	81.162
32	4294956670.1344873435426145532537 1	126.805
33	8589957985.0530892423883354132392 3	196.982
34	17179891462.629201881592932537203 7	306.093
35	34359687815.675550310821311464645 5	477.336
36	68719431461.327794639067589167331 2	737.598
37	137438955228.48195405364102813809 7	1144.511
38	274877885788.63319049261781765975 0	1752.836
39	549755934915.77529959993283733631 5	2724.370
40	1099511624070.9515243927711747597 1	4236.570
41	2199022895973.2087824946381927586 9	6521.847
42	4398047079564.1338995625975177878 1	9934.160
43	8796092823052.7780699214756443187 6	15404.534
44	17592186538719.432586321142712938 7	24103.794

Table B.4: Approximations of ψ at powers of 2 with 100 digits of precision.

n	$\psi(2^n)$	Time (in seconds)
6	62.337231194609467866435095694222509810030914907948⋱ 45783197599315306203163104509873448717717833667314	0.01
7	126.93139272208268541104895442978764344579478402185⋱ 11694373256382511695212907691969343068819202590657	0.01
8	251.49307009837428804173436640792178837690587789301⋱ 54903407548474616059732128340785671245729830741019	0.02
9	514.79830201695784316277014489216346427304262054221⋱ 70215373261447463887308863481340432222417218560710	0.03
10	1025.0665608037248682785945422220903191180478326344⋱ 50969306340361946417387029786398452646703205106405	0.04
11	2040.8127076788827101462589263191186682722886385178⋱ 75454916359094553273661342013308289222374252413277	0.07
12	4106.6333406235809397129732624231835758364818282635⋱ 58961488299962432361845295691737840488614406586540	0.12
13	8176.9694252202565289050672910128938166000645544393⋱ 15361807820026478436755299616834791703354629748536	0.19
14	16408.514900137643900575095227372036531881429513443⋱ 53329216184568579365323975933738506612497766008774	0.32
15	32736.919484998285805644028484768044045520395152992⋱ 89159411800498893720552022917618836891811631655702	0.45
16	65466.400464967506771575265219489383391088626982692⋱ 62512633711803613235710441159716051155305243864622	0.74
17	131090.52025236027320915005828493383004927207788336⋱ 40502728678234204012088421029784731949596431977109	1.10
18	262034.92911613517461500460243937446034308690979821⋱ 95469049938130545331352243462867816581475512951654	1.86
19	524474.32575285085668403952735032158188670313607556⋱ 51844967210647879050235565584733530800257097171879	3.06
20	1048430.4214037888280011160947399371127545026744801⋱ 88358861632822870519056721891813090658945660840070	4.88
21	2097211.8618086878845755242502579466885641155949621⋱ 47334156524437551415956753754638586383707374125450	7.21
22	4194440.8768981085818529262831202788637710628747992⋱ 52003896897221997964227499964219596062474327035486	11.30
23	8388383.5420935264665780342158140138691759609432451⋱ 60027510961025038919048464340532540683581798656931	17.71

Continued on next page

Table B.4: Approximations of ψ at powers of 2 with 100 digits of precision. (Continued)

n	$\psi(2^n)$	Time (in seconds)
24	16776556.2899881680628659621777681047426869663008700 $\ddots$ 3381834204168595955805031128028186447174639511 4117	27.66
25	33556481.823201979752504266157851202201593279791696 $\ddots$ 62927995885923520851307021189701601760138543947631	44.40
26	67109113.464223632253091727646766536732212763353259 $\ddots$ 34009093994248732737309131842076247356050482288102	68.48
27	134216112.717281913707632360111761844575791079236685 $\ddots$ 42924603864889415340735556760975046681277666435795	106.01
28	268436833.423884297699309528690802662016570401959999 $\ddots$ 77319047281498675409757072368391342618189932105077	162.99
29	536866944.479688244019098393407404532797783482854421 $\ddots$ 94111708913061276153905244648448851466278359678910	252.75
30	1073749873.232311368703748742558695123428711709070 4 $\ddots$ 79909203660673824879988331295130852538505734736802	393.89
31	2147479854.228056224934463298970934144245372099142 8 $\ddots$ 95951956626425468458534612365253065685250811647095	608.96
32	4294956670.134487343542614553253774490326966883936 9 $\ddots$ 06432473650360666084159784066869566243176789648787	945.81
33	8589957985.053089242388335413239352484795031623872 1 $\ddots$ 88674507432179025730616754749279439183165353197190	1464.14
34	17179891462.629201881592932537203944624931525822556 $\ddots$ 68725426998225529914671832325448758604956651156156	2271.12
35	34359687815.675550310821311464646156248089824633846 $\ddots$ 45790361701398764267987507218797979546567949412454	3545.00
36	68719431461.327794639067589167332251215272048248619 $\ddots$ 57415128088309604586449360813899263585924967244511	5471.66

B.2 Fourier-theoretic method

Table B.5: Approximations of ψ at powers of 10 with 33 digits of precision.

n	$\psi(10^n)$	Time (in seconds)
2	94.0453112293573922460049312446068	0.032
3	996.680912247175240263021765666414	0.092
4	10013.3966932631147837203245944702	0.431
5	100051.564025657954250923140327612	1.856
6	999586.597495632922033061533011302	7.620
7	9998539.40334597536635289816238934	32.031
8	99998242.7966267823416068404575383	145.688
9	1000001595.99042758043906519942950	569.871
10	10000042536.4975913483686507492957	2652.443
11	100000058456.430302189943653434056	10207.670
12	1000000040136.76545665644667959114	28825.182

Table B.6: Approximations of ψ at powers of 10 with 100 digits of precision.

n	$\psi(10^n)$	Time (in seconds)
2	94.0453112293573922460049312446069272413260973114 52 $\ddots$ 8531092486428032104521481076885130159937278073281 3	0.11
3	996.68091224717524026302176566664215416657784369021 3 $\ddots$ 9866913826009891405279152145893232163908036965237 7	0.34
4	10013.396693263114783720324594470187567072560129718 $\ddots$ 1394567432199160053798030875023133733697809930440 2	1.82
5	100051.564025657954250923140327611627124929146243 21 $\ddots$ 5772928941987800028268726603051474660128651587796 6	7.47
6	999586.59749563292203306153301130446359633577788026 $\ddots$ 4135230204470022517093909402883597912074251265684 6	29.28
7	9998539.4033459753663528981623893319752768018964902 $\ddots$ 1193294294168395307949931623881060399470342584642 5	121.01
8	99998242.796626782341606840457535809044522868171269 $\ddots$ 6011412513102839976168949556011933788516436037922 0	548.93
9	1000001595.9904275804390651994295128367679624178605 $\ddots$ 6811251451067874180003839227768868194498211855904 6	1982.93
10	10000042536.497591348368650749295950317535021797758 $\ddots$ 4842475114510636819186164922975936214819515525092 4	9516.57
11	100000058456.43030218994365343405605601334959367638 $\ddots$ 2807975652715221239084409742779932903062577829482 8	39336.52

Table B.7: Approximations of ψ at powers of 2 with ~ 33 digits of precision.

n	$\psi(2^n)$	Time (in seconds)
6	62.3372311946094678664350956942222	0.028
7	126.931392722082685411048954429787	0.026
8	251.493070098374288041734366407922	0.044
9	514.798302016957843162770144892161	0.069
10	1025.06656080372486827859454222209	0.099
11	2040.81270767888271014625892631911	0.170
12	4106.63334062358093971297326242314	0.329
13	8176.96942522025652890506729101286	0.406
14	16408.5149001376439005750952273718	0.718
15	32736.9194849982858056440284847680	0.965
16	65466.4004649675067715752652194889	1.684
17	131090.520252360273209150058284933	2.729
18	262034.929116135174615004602439372	3.818
19	524474.325752850856684039527350317	5.739
20	1048430.42140378882800111609473991	8.845
21	2097211.86180868788457552425025792	14.379
22	4194440.87689810858185292628312021	26.820
23	8388383.54209352646657803421581388	29.622
24	16776556.2899881680628659621777678	59.195
25	33556481.8232019797525042661578507	65.640
26	67109113.4642236322530917276467655	130.493
27	134216112.717281913707632360111760	165.444
28	268436833.423884297699309528690799	283.884
29	536866944.479688244019098393407397	313.813
30	1073749873.23231136870374874255868	628.709
31	2147479854.22805622493446329897090	1070.247
32	4294956670.13448734354261455325371	1351.624
33	8589957985.05308924238833541323927	2131.806
34	17179891462.6292018815929325372037	2946.027
35	34359687815.6755503108213114646451	4827.097
36	68719431461.3277946390675891673313	6485.789
37	137438955228.481954053641028138096	11067.220
38	274877885788.633190492617817659798	14210.710

Table B.8: Approximations of ψ at powers of 2 with 100 digits of precision.

n	$\psi(2^n)$	Time (in seconds)
6	62.337231194609467866435095694222509810030914907948⋱ 4578319759931530620316310450987344871771783366 7314	0.08
7	126.931392722082685411048954429787643445794784021 85⋱ 1169437325638251169521290769196934306881920259 0657	0.07
8	251.493070098374288041734366407921788376905877893 01⋱ 5490340754847461605973212834078567124572983074 1019	0.17
9	514.798302016957843162770144892163464273042620542 21⋱ 7021537326144746388730886348134043222241721856 0710	0.30
10	1025.066560803724868278594542222090319118047832634 4⋱ 5096930634036194641738702978639845264670320510 6405	0.41
11	2040.812707678882710146258926319118668272288638517 8⋱ 7545491635909455327366134201330828922237425241 3277	0.77
12	4106.633340623580939712973262423183575836481828263 5⋱ 5896148829996243236184529569173784048861440658 6540	1.37
13	8176.969425220256528905067291012893816600064554439 3⋱ 1536180782002647843675529961683479170335462974 8536	1.78
14	16408.51490013764390057509522737203653188142951344 3⋱ 5332921618456857936532397593373850661249776600 8774	2.91
15	32736.91948499828580564402848476804404552039515299 2⋱ 8915941180049889372055202291761883689181163165 5702	3.74
16	65466.40046496750677157526521948938339108862698269 2⋱ 6251263371180361323571044115971605115530524386 4622	7.30
17	131090.5202523602732091500582849338300492720778833 6⋱ 4050272867823420401208842102978473194959643197 7109	11.32
18	262034.9291161351746150046024393744603430869097982 1⋱ 9546904993813054533135224346286781658147551295 1654	16.93
19	524474.3257528508566840395273503215818867031360755 6⋱ 5184496721064787905023556558473353080025709717 1879	22.62
20	1048430.421403788828001116094739937112754502674480 1⋱ 8835886163282287051905672189181309065894566084 0070	37.79
21	2097211.861808687884575524250257946688564115594962 1⋱ 4733415652443755141595675375463858638370737412 5450	57.54
22	4194440.876898108581852926283120278863771062874799 2⋱ 5200389689722199796422749996421959606247432703 5486	112.84
23	8388383.542093526466578034215814013869175960943245 1⋱ 6002751096102503891904846434053254068358179865 6931	111.99

Continued on next page

Table B.8: Approximations of ψ at powers of 2 with 100 digits of precision. (Continued)

n	$\psi(2^n)$	Time (in seconds)
24	16776556.28998816806286596217776810474268696630087033818342041685959558050311280281864471746395114117	247.00
25	33556481.82320197975250426615785120220159327979169662927995885923520851307021189701601760138543947631	234.55
26	67109113.46422363225309172764676653673221276335325934009093994248732737309131842076247356050482288102	522.87
27	134216112.7172819137076323601117618445757910792368542924603864889415340735556760975046681277666435795	659.25
28	268436833.4238842976993095286908026620165704019599977319047281498675409757072368391342618189932105077	1135.47
29	536866944.4796882440190983934074045327977834828542194111708913061276153905244648448851466278359678910	1070.61
30	1073749873.232311368703748742558695123428711709070479909203660673824879988331295130852538505734736802	2494.60
31	2147479854.228056224934463298970934144245372099142895951956626425468458534612365253065685250811647095	4175.74
32	4294956670.134487343542614553253774490326966883936906432473650360666084159784066869566243176789648787	5256.60