Formulas and Algorithms for the Length of a Farey Sequence

Vladimir Sukhoy[1] **and Alexander Stoytchev**[1,*]

[1]Department of Electrical and Computer Engineering, Iowa State University, Ames IA, 50011, USA
[*]alexs@iastate.edu

ABSTRACT

This paper proves several novel formulas for the length of a Farey sequence of order n. The formulas use different trade-offs between iteration and recurrence and they range from simple to more complex. The paper also describes several iterative algorithms for computing the length of a Farey sequence based on these formulas. The algorithms are presented from the slowest to the fastest in order to explain the improvements in computational techniques from one version to another. The last algorithm in this progression runs in $O(n^{2/3})$ time and uses only $O(\sqrt{n})$ memory, which makes it the most efficient algorithm for computing $|F_n|$ described to date. With this algorithm we were able to compute the length of the Farey sequence of order 10^{18}.

1 Introduction

Farey sequences[1,2] are related to the theory of prime numbers and they show up in many different scientific disciplines. Their fundamental properties, e.g., the mediant property, can be described with basic algebra. At the same time, Farey sequences are linked to unsolved mathematical mysteries, e.g., the Riemann hypothesis[3]. These sequences are also tied to the Stern–Brocot tree, which could be used to find the best rational approximations for irrational numbers[4]. Recently, the elements of a Farey sequence have been linked to the singularities[5] of the Inverse Chirp Z-Transform (ICZT), which is a generalization[6] of the Inverse Fast Fourier Transform (IFFT).

A Farey sequence of order n, which is denoted by F_n, is a sequence formed by all irreducible fractions ${}^{p}\!/_{q}$ between 0 and 1 for which the denominator q is between 1 and n, i.e., $F_n = \left\{{}^{p}\!/_{q} \text{ s.t. } q \in \{1,2,\ldots,n\},\ p \in \{0,1,2,\ldots,q\},\ \text{and } \gcd(p,q)=1\right\}$. By convention, it is assumed that the elements of the sequence F_n are sorted in increasing order. For example, the first five Farey sequences are:

$$F_1 = \left(\frac{0}{1}, \frac{1}{1}\right),$$
$$F_2 = \left(\frac{0}{1}, \frac{1}{2}, \frac{1}{1}\right),$$
$$F_3 = \left(\frac{0}{1}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{1}{1}\right),$$
$$F_4 = \left(\frac{0}{1}, \frac{1}{4}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \frac{1}{1}\right),$$
$$F_5 = \left(\frac{0}{1}, \frac{1}{5}, \frac{1}{4}, \frac{1}{3}, \frac{2}{5}, \frac{1}{2}, \frac{3}{5}, \frac{2}{3}, \frac{3}{4}, \frac{4}{5}, \frac{1}{1}\right).$$

If ${}^{a}\!/_{b}$, ${}^{c}\!/_{d}$, and ${}^{p}\!/_{q}$ are any three adjacent elements of a Farey sequence, then the middle fraction is equal to the *mediant*[2,7] of its neighbors ${}^{a}\!/_{b}$ and ${}^{p}\!/_{q}$, i.e.,

$$\frac{c}{d} = \frac{a+p}{b+q}.$$

This property has been known to mathematicians for centuries[8,9], but it received a name only after Farey stated it formally in a paper[1] that he published in 1816. Previously, Haros[2,9] had used the mediant property in 1802 to generate the tables of irreducible fractions between 0 and 1 for which the denominator was less than 100 (see ref. 2, p. 36). Cauchy published a formal proof of the mediant property[10] for all n in 1816.

As the order n of the Farey sequence F_n increases, the length of F_n grows as a quadratic function of n. More specifically, $|F_n| \sim 3n^2/\pi^2$, where $|F_n|$ denotes the length of F_n [2, p. 268] [7, p. 156] [4, p. 139]. However, no formula for computing the exact value of $|F_n|$ in $O(1)$ time is known.

The length of F_n can be computed by enumerating its elements. Algorithm S1 in Supplementary Section S1 gives the pseudo-code for an algorithm[11] that uses the mediant property to enumerate all elements of a given Farey sequence. The algorithm also counts the elements and returns the sequence length. The computational complexity of this approach is $O(n^2)$, which makes it too slow and impractical for computing the value of $|F_n|$ for large n. The fast algorithms described in this paper do not use enumeration. Their computational complexities are summarized in Supplementary Section S2.

Increasing the value of n only adds new elements to the Farey sequence without removing any of them. That is, F_{n-1} is a subsequence of F_n. To see this, consider the elements of the first five Farey sequences shown below, which are arranged with extra spacing between them. In this view the identical fractions are stacked vertically. The new elements that are added when the order is increased by 1 are highlighted in red. The length of each Farey sequence is shown at the end of each row.

$$
\begin{aligned}
F_1 &= \left(\frac{0}{1}, \qquad\qquad\qquad\qquad \frac{1}{1}\right), & |F_1| &= 2,\\
F_2 &= \left(\frac{0}{1}, \qquad\qquad \frac{1}{2}, \qquad\qquad \frac{1}{1}\right), & |F_2| &= 3,\\
F_3 &= \left(\frac{0}{1}, \qquad \frac{1}{3}, \quad \frac{1}{2}, \quad \frac{2}{3}, \qquad \frac{1}{1}\right), & |F_3| &= 5,\\
F_4 &= \left(\frac{0}{1}, \quad \frac{1}{4}, \frac{1}{3}, \quad \frac{1}{2}, \quad \frac{2}{3}, \frac{3}{4}, \quad \frac{1}{1}\right), & |F_4| &= 7,\\
F_5 &= \left(\frac{0}{1}, \frac{1}{5}, \frac{1}{4}, \frac{1}{3}, \frac{2}{5}, \frac{1}{2}, \frac{3}{5}, \frac{2}{3}, \frac{3}{4}, \frac{4}{5}, \frac{1}{1}\right), & |F_5| &= 11.
\end{aligned}
$$

Each element of the set $F_n \setminus F_{n-1}$ is a fraction ${}^{k}/_{n}$ that is irreducible and lies between 0 and 1. Thus, the difference between $|F_n|$ and $|F_{n-1}|$ is equal to the number of integers between 1 and n that are coprime with n. By definition, this number is equal to the value of Euler's totient function $\varphi(n)$. Therefore, an algorithm[12] for computing the length of F_n can also be used to calculate the sums of the values of Euler's totient function for all integers between 1 and n.

n	$\varphi(n)$	$\lvert F_n\rvert$
1	1	$2 \;= 1+1$
2	1	$3 \;= 1+1+1$
3	2	$5 \;= 1+1+1+2$
4	2	$7 \;= 1+1+1+2+2$
5	4	$11 = 1+1+1+2+2+4$

Figure 1. Visualization of the link between Euler's totient function $\varphi(n)$ and the length of the Farey sequence F_n.

Figure 1 lists the values of Euler's totients $\varphi(1)$, $\varphi(2)$, $\varphi(3)$, $\varphi(4)$, $\varphi(5)$ and expresses the values of $|F_1|$, $|F_2|$, $|F_3|$, $|F_4|$, and $|F_5|$ in terms of these totients. This example also illustrates the following well-known formula for computing $|F_n|$ that is proven in Supplementary Section S3:

$$|F_n| = 1 + \sum_{k=1}^{n} \varphi(k). \tag{1}$$

This equation implies that the value of $\varphi(n)$ can be added to $|F_{n-1}|$ to obtain $|F_n|$, i.e., $|F_n| = |F_{n+1}| + \varphi(n)$.

The value of $|F_n|$ can also be expressed using the following recursive formula[13]:

$$|F_n| = \frac{(n+3)\,n}{2} - \sum_{k=2}^{n} \left|F_{\lfloor n/k \rfloor}\right|. \tag{2}$$

Supplementary Sections S4 and S5 prove this formula using basic algebra, mathematical induction, and a property of Euler's totient function that was proven by Gauss[14]. Algorithm S11 in Supplementary Section S9 implements formula (2) using recursion and optimizes the repeated recursive calls using *memoization*[15]. That algorithm runs in $O(n \log n)$ time, which is still too slow for large values of n.

The algorithms described in this paper build upon these two formulas, extend them, and combine them in different ways. The next section summarizes the key mathematical insights that were used to derive the formulas. The rest of the paper describes the algorithms, proves their properties, and evaluates their run-time performance.

2 Overview and Formulas

This paper describes five different algorithms for computing the length of the Farey sequence F_n, where n is a parameter that specifies the order. The algorithms will be denoted with the letters A, B, C, D, and E. These letters will also be used as suffixes to form the names of the algorithms, e.g., FAREYLENGTHA or FAREYLENGTHB. This section briefly summarizes the formulas for the length of F_n on which the algorithms are based.

Algorithm A is based on the following well-known formula:

$$|F_n| = 1 + \sum_{k=1}^{n} \varphi(k), \tag{3}$$

which was illustrated with an example in the introduction. The algorithm uses a modified linear sieve that returns a list of prime numbers and an array with the smallest prime factor for each integer. From these values, another function computes Euler's totients, which are then summed up to compute the length of F_n. This algorithm runs in $O(n)$ time and uses $O(n)$ memory.

The formula for algorithm B is derived from (3) by splitting the sum into three separate sums:

$$|F_n| = 1 + \sum_{k=1}^{\lfloor\sqrt{n}\rfloor} \varphi(k) + \sum_{k\in S} \varphi(k) + \sum_{k\in \overline{S}} \varphi(k), \tag{4}$$

where S is the set of $\lfloor\sqrt{n}\rfloor$-smooth integers in the interval $\left[\lfloor\sqrt{n}+1\rfloor, n\right]$ and $\overline{S}$ is the set of integers that are not $\lfloor\sqrt{n}\rfloor$-smooth in the same interval. Two helper algorithms are introduced to process the smooth and the non-smooth integers, which are formally defined in one of the next sections. The overall algorithm still runs in $O(n)$ time, but uses only $O(n^{1/2+o(1)})$ memory.

Algorithm C is based on the following novel formula for the length of F_n:

$$|F_n| = \frac{(n+3)n}{2} - \sum_{k=2}^{u(n)} \left|F_{\lfloor n/k\rfloor}\right| - \sum_{k=1}^{\lfloor\sqrt{n}\rfloor} \left(\left\lfloor\frac{n}{k}\right\rfloor - \left\lfloor\frac{n}{k+1}\right\rfloor\right)\cdot |F_k|, \tag{5}$$

where $u(n) = \left\lfloor n/\left(\lfloor\sqrt{n}\rfloor+1\right)\right\rfloor$. The formula works for all $n > 1$. Supplementary Section S6 proves this formula by splitting the sum in formula (2) at $u(n)$ and then expressing the second sum in a different way. This leads to an algorithm that runs in $O(n^{3/4})$ time and uses $O(\sqrt{n})$ memory.

Algorithm D can be derived from formula (5) by splitting the first sum at $v(n) = \left\lfloor n/\left(\lfloor\sqrt[3]{n^2}\rfloor+1\right)\right\rfloor$ as follows:

$$\sum_{k=2}^{u(n)} \left|F_{\lfloor n/k\rfloor}\right| = \sum_{k=2}^{v(n)} \left|F_{\lfloor n/k\rfloor}\right| + \sum_{k=v(n)+1}^{u(n)} \left|F_{\lfloor n/k\rfloor}\right|. \tag{6}$$

Reversing the direction of the last sum and changing the index variable from k to i we get:

$$\sum_{k=2}^{u(n)} \left|F_{\lfloor n/k\rfloor}\right| = \sum_{k=2}^{v(n)} \left|F_{\lfloor n/k\rfloor}\right| + \sum_{i=0}^{w(n)} \left|F_{\left\lfloor\frac{n}{u(n)-i}\right\rfloor}\right|, \tag{7}$$

where $w(n) = u(n) - v(n) - 1$. To continue the derivation, let b_i be the order of the Farey sequence in the last sum in formula (7), i.e., $b_i = \left\lfloor\frac{n}{u(n)-i}\right\rfloor$. Then,

$$\sum_{i=0}^{w(n)} \left|F_{\left\lfloor\frac{n}{u(n)-i}\right\rfloor}\right| = \sum_{i=0}^{w(n)} \left|F_{b_i}\right|. \tag{8}$$

This sum can be expressed using Euler's totients for all integers in the intervals $[a_0, b_0]$, $[a_1, b_1], \dots, [a_{w(n)}, b_{w(n)}]$, where $a_0 = \lfloor\sqrt{n}\rfloor + 1$, $b_0 = \left\lfloor\frac{n}{u(n)}\right\rfloor$, and $a_i = b_{i-1} + 1$ for $i \in \{1, 2, \dots, w(n)\}$. To derive this result, the value of $\left|F_{b_i}\right|$ can be expressed as the value of $\left|F_{b_{i-1}}\right|$ plus the sum of Euler's totients for all integers in the interval $[a_i, b_i]$. That is,

$$\begin{aligned}\left|F_{b_i}\right| &= \left|F_{\lfloor\sqrt{n}\rfloor}\right| + \varphi(\underbrace{\lfloor\sqrt{n}\rfloor+1}_{a_0}) + \varphi(\lfloor\sqrt{n}\rfloor+2) + \cdots + \varphi(b_i - 1) + \varphi(b_i) \\ &= \underbrace{\left|F_{\lfloor\sqrt{n}\rfloor}\right| + \sum_{m=a_0}^{b_0}\varphi(m) + \sum_{m=a_1}^{b_1}\varphi(m) + \cdots + \sum_{m=a_{i-1}}^{b_{i-1}}\varphi(m)}_{\left|F_{b_{i-1}}\right|} + \sum_{m=a_i}^{b_i}\varphi(m) \\ &= \left|F_{b_{i-1}}\right| + \sum_{m=a_i}^{b_i}\varphi(m). \end{aligned} \tag{9}$$

Therefore, formula (8) can be expressed as follows:

$$\sum_{i=0}^{w(n)}\left|F_{\left\lfloor\frac{n}{u(n)-i}\right\rfloor}\right| = \sum_{i=0}^{w(n)}\left(\left|F_{b_{i-1}}\right| + \sum_{m=a_i}^{b_i}\varphi(m)\right). \tag{10}$$

Plugging the right-hand side of the last equation into formula (6) and that result into (5) leads to the following formula:

$$|F_n| = \frac{(n+3)n}{2} - \sum_{k=2}^{v(n)}\left|F_{\lfloor n/k\rfloor}\right| - \sum_{i=0}^{w(n)}\underbrace{\left(\left|F_{b_{i-1}}\right| + \sum_{m=a_i}^{b_i}\varphi(m)\right)}_{|F_{b_i}|} - \sum_{k=1}^{\lfloor\sqrt{n}\rfloor}\left(\left\lfloor\frac{n}{k}\right\rfloor - \left\lfloor\frac{n}{k+1}\right\rfloor\right)\cdot\underbrace{\left(1+\sum_{m=1}^{k}\varphi(m)\right)}_{|F_k|}, \tag{11}$$

where $w(n) = u(n) - v(n) - 1$, $u(n) = \left\lfloor n/\left(\lfloor\sqrt{n}\rfloor+1\right)\right\rfloor$, and $v(n) = \left\lfloor n/\left(\lfloor\sqrt[3]{n^2}\rfloor+1\right)\right\rfloor$. Also, $a_0 = \lfloor\sqrt{n}\rfloor+1$ and $b_0 = \left\lfloor\frac{n}{u(n)}\right\rfloor$. Furthermore, $a_i = b_{i-1}+1$ and $b_i = \left\lfloor\frac{n}{u(n)-i}\right\rfloor$ for $i \in \{1, 2, \ldots, w(n)\}$. The last term in (11) expands the value of $|F_k|$ as a sum of totients, which suggests how it can be computed iteratively. Because the index k in the last sum in (11) goes up to $\lfloor\sqrt{n}\rfloor$, the last value of $|F_k|$ is equal to $|F_{\lfloor\sqrt{n}\rfloor}|$. If the last sum is processed first, then the computed value of $|F_{\lfloor\sqrt{n}\rfloor}|$ can be used to bootstrap the calculation of $|F_{b_i}|$ in the middle sum, e.g., see formula (9). These insights lead to algorithm D, which runs in $O(n^{2/3})$ time and uses $O(n^{2/3})$ memory.

Algorithm E was inspired by a modified version of formula (11) that is shown below:

$$|F_n| = \frac{(n+3)n}{2} - \sum_{k=2}^{v(n)}\left|F_{\lfloor n/k\rfloor}\right| - \sum_{i=0}^{w(n)}\left(\left|F_{b_{i-1}}\right| + \underbrace{\sum_{m\in S_i}\varphi(m) + \sum_{m\in\overline{S_i}}\varphi(m)}_{B[i]}\right) - \sum_{k=1}^{\lfloor\sqrt{n}\rfloor}\left(\left\lfloor\frac{n}{k}\right\rfloor - \left\lfloor\frac{n}{k+1}\right\rfloor\right)\cdot\underbrace{\left(1+\sum_{m=1}^{k}\varphi(m)\right)}_{|F_k|}. \tag{12}$$

In other words, this modification splits one of the sums in (11) as follows:

$$\sum_{m=a_i}^{b_i}\varphi(m) = \underbrace{\sum_{m\in S_i}\varphi(m) + \sum_{m\in\overline{S_i}}\varphi(m)}_{B[i]}, \tag{13}$$

where S_i is the set of $\lfloor\sqrt{n}\rfloor$-smooth integers in the interval $I_i = [a_i, b_i]$ and $\overline{S_i}$ is the set of integers that are not $\lfloor\sqrt{n}\rfloor$-smooth in the same interval. Thus, algorithm E uses a similar approach to summing the totients as in formula (4) that is used by algorithm B. In this case, however, the smooth and the non-smooth numbers are processed separately but the values of their corresponding totients are accumulated in the array B. This change is sufficient to reduce the memory complexity from $O(n^{2/3})$ to $O(\sqrt{n})$. The time complexity remains unchanged, i.e., $O(n^{2/3})$.

3 Related Work

A prime sieve is an efficient algorithm for generating all prime numbers in some interval[16]. The sieve of Eratosthenes is probably the oldest and the most well-known prime sieve algorithm[4,16,17]. It can generate all prime numbers in the interval $[1,n]$ in $O(n \log\log n)$ time. Our algorithms, however, do not use the sieve of Eratosthenes. Instead, they use the linear sieve[18] or the sieve of Atkin[19]. These two sieves do not have the factor $\log\log n$ in their time complexity, i.e., they run in $O(n)$ time.

Algorithms A, B, D, and E use the linear sieve algorithm[18], making it possible to compute Euler's totients for all integers between 1 and n in $O(n)$ time. In addition to the linear sieve, Algorithms B and E also use the sieve of Atkin[19] to reduce the space complexity. Algorithm C does not use a sieve.

The sieve of Atkin generates all prime numbers between 1 and N in $O(N)$ time[19]. It uses $O(N^{1/2+o(1)})$ memory[19]. Algorithm E uses this sieve with $N = \beta \in O(n^{2/3})$ to enumerate numbers that are not $(\alpha-1)$-smooth in the interval $[\alpha,\beta]$. This sub-routine and other optimizations reduce its space complexity down to $O(\sqrt{n})$. This makes it practical to compute $|F_n|$ for very large values of n by using the computer's memory more efficiently. Algorithm E enumerates the smooth numbers in $[\alpha,\beta]$ separately from the non-smooth numbers, which are complementary to them. Smooth numbers[20–22] are often used in computational number theory for primality testing, integer factorization, and computing discrete logarithms.

Computing the length of the Farey sequence is related to several other problems. For example, by subtracting 1 from $|F_n|$ one can obtain[12] the sum of all Euler's totients between 1 and n. Some authors have described approaches for attacking the order statistics problem[23,24] and the rank problem[25] on Farey sequences. Their work is related to methods for summing the values of the Möbius function[26], which is equivalent to computing the Mertens function. The time complexity class of these methods is $O(n^{2/3}(\log\log n)^{1/3})$. More recently, the best time complexity for computing the Mertens function was estimated[27] at $O(n^{2/3+\varepsilon})$, i.e., slightly worse than $O(n^{2/3})$. The time complexity of our most efficient algorithm for computing $|F_n|$ is exactly $O(n^{2/3})$, i.e., without any additional small factors that depend on n. Its space complexity is $O(\sqrt{n})$. All estimates use the same computational model, which assumes that any arithmetic or storage operation on any integer runs in $O(1)$ time and storing any integer requires $O(1)$ memory.

In digital signal processing, the Inverse Chirp Z-Transform (ICZT) is a generalization[6] of the Inverse Fast Fourier Transform (IFFT). This transform is parametrized by the complex numbers A and W. They define a logarithmic spiral contour formed by the sampling points AW^{-k} where $k \in \{0,1,2,\dots,n-1\}$ and n is the size of the transform. For the special case when the magnitudes of both A and W are equal to 1, the contour is restricted to lie on the unit circle in the complex plane. In this case, the ICZT has a singularity[5] if and only if the polar angle of W can be expressed as $2\pi p/q$ where p/q is an element of F_{n-1}. Consequently, the numerical error profile for the ICZT of size n is determined[5] by the elements of F_{n-1}. Therefore, the number of possible values of the parameter W for which the transform is singular is equal to the length of F_{n-1}.

4 Algorithm A

Algorithm 1 shows the pseudo-code for the first algorithm that computes $|F_n|$ without enumerating the sequence elements. On line 2 the algorithm uses a linear sieve to compute a list P of all prime numbers in the interval $[1,n]$. The linear sieve also returns an array L_p of size n such that its k-th element $L_p[k]$ is equal to the smallest prime factor of k. On line 3 the elements of L_p are used to compute an array φ that contains the values of Euler's totient function $\varphi(1), \varphi(2), \dots, \varphi(n)$. The rest of the code (i.e., lines 4–7) uses formula (3) to compute the value of $|F_n|$, i.e., it sums up the n totients and adds 1 to the sum s.

Algorithm 1. Compute the length of the Farey sequence F_n. Runs in $O(n)$ time and uses $O(n)$ memory.

```
1: function FAREYLENGTHA($n$)
2:     $(P, L_p) \leftarrow$ LINEARSIEVE($n$);          // See Algorithm S2
3:     $\varphi \leftarrow$ COMPUTETOTIENTS($n, L_p$);   // See Algorithm S3
4:     $s \leftarrow 1$;
5:     for $k \leftarrow 1$ to $n$ do
6:         $s \leftarrow s + \varphi[k]$;
7:     end for
8:     return $s$;
9: end function
```

Supplementary Section S3 gives the pseudo-code for the linear sieve algorithm and for the function COMPUTETOTIENTS. The appendix also proves a property of Euler's totient function that makes it possible to compute $\varphi(1), \varphi(2), \dots, \varphi(n)$ from the array L_p in $O(n)$ time. Thus, the computational complexity of Algorithm 1 is $O(n)$. It uses $O(n)$ memory.

This algorithm is fairly easy to describe, but it is not very fast. It also uses a lot of memory. It is important to understand how it works, however, as some of the more efficient algorithms use the same approach to solve a part of the problem.

5 Algorithm B

This section describes another algorithm for computing $|F_n|$ that also runs in $O(n)$ time, but uses only slightly more than $O(\sqrt{n})$ memory. The main idea is to split all integers in the interval $[\lfloor\sqrt{n}\rfloor+1, n]$ into two disjoint sets, S and $\overline{S}$, such that their elements and their corresponding Euler's totients can be computed using $O(\sqrt{n})$ memory instead of $O(n)$. The set S is the set of $\lfloor\sqrt{n}\rfloor$-smooth numbers and the set $\overline{S}$ is the set of numbers that are not $\lfloor\sqrt{n}\rfloor$-smooth.

Both smooth and non-smooth numbers are integers. They are defined as follows:

- A positive integer N is B-smooth if its largest prime factor p does not exceed B, i.e., $p \leq B$.
- A positive integer N is not B-smooth if its largest prime factor p is strictly greater than B, i.e., $p > B$.

For example, the number 100 is 5-smooth because $100 = 5^2 \cdot 2^2$. It is also 6-smooth, 7-smooth, 8-smooth, etc. But it is not k-smooth for $k \in \{2, 3, 4\}$. To give another example, the number 84 is 7-smooth because $84 = 7 \cdot 3 \cdot 2^2$. It is also 8-smooth, 9-smooth, etc. But it is not 5-smooth because its largest prime factor is 7, which is greater than 5. In fact, it is not k-smooth for $k \in \{2, 3, 4, 5, 6\}$.

Algorithm 2 starts by computing $r = \lfloor\sqrt{n}\rfloor$ exactly using the method[28] described in Supplementary Section S7. Next, it implements the summation of Euler's totients in a way that avoids allocating an array of length n to store them. In fact, the algorithm uses only slightly more than $O(\sqrt{n})$ memory. This is achieved by running the linear sieve only for $k \in \{1, 2, \ldots, \lfloor\sqrt{n}\rfloor\}$. Starting from $\lfloor\sqrt{n}+1\rfloor$, the sum in formula (3) is split into three separate sums as follows:

$$|F_n| = 1 + \sum_{k=1}^{\lfloor\sqrt{n}\rfloor} \varphi(k) + \sum_{k\in S} \varphi(k) + \sum_{k\in\overline{S}} \varphi(k), \tag{14}$$

where S is the set of $\lfloor\sqrt{n}\rfloor$-smooth numbers in the interval $\left[\lfloor\sqrt{n}+1\rfloor, n\right]$ and $\overline{S}$ is the set of numbers that are not $\lfloor\sqrt{n}\rfloor$-smooth in the same interval. A similar technique is used in Algorithm 8 to reduce its space complexity from $O(n^{2/3})$ to $O(\sqrt{n})$.

Algorithm 2. Compute the length of the Farey sequence F_n. Runs in $O(n)$ time and uses $O(n^{1/2+o(1)})$ memory.

```
1:  function FAREYLENGTHB($n$)
2:      $r \leftarrow$ ISQRT($n$);                                  // compute $\lfloor\sqrt{n}\rfloor$ exactly using Algorithm S4
3:      $(P, L_p) \leftarrow$ LINEARSIEVE($r$);
4:      $\varphi \leftarrow$ COMPUTETOTIENTS($r, L_p$);
5:      $s \leftarrow 1$;
6:      for $k \leftarrow 1$ to $r$ do
7:          $s \leftarrow s + \varphi[k]$;
8:      end for
9:      function VISITOR($m, \phi$)
10:         // The variable $s$ is from the outer scope.
11:         $s \leftarrow s + \phi$;
12:     end function
13:     PROCESSSMOOTHNUMBERS($P, r{+}1, n$, VISITOR);              // $r$-smooth numbers in $[r{+}1, n]$
14:     PROCESSNONSMOOTHNUMBERS($\varphi, r{+}1, n$, VISITOR);     // not $r$-smooth numbers in $[r{+}1, n]$
15:     return $s$;
16: end function
```

The smooth numbers are processed with Algorithm 3. It enumerates the $(\alpha-1)$-smooth numbers in the interval $[\alpha, \beta]$. More specifically, the algorithm implements a depth-first traversal of the search space formed by all integers in this interval that have prime factorizations that include only prime factors from the array P, which contains the prime numbers in the interval $[1, \alpha-1]$. The algorithm calls the visitor function for each $(\alpha-1)$-smooth number m in the interval $[\alpha, \beta]$ and its Euler's totient, which is computed using formula (22) that is described in Supplementary Section S3. Algorithm 3 runs in $O(\beta)$ time and uses $O(\log\beta)$ memory. It traverses no more than 2β values of m and performs $O(1)$ operations for each of them. The space complexity of this algorithm is determined by the maximum depth of the stack, which cannot exceed $\lfloor\log_2\beta\rfloor$ because 2 is the smallest possible prime factor in the list P.

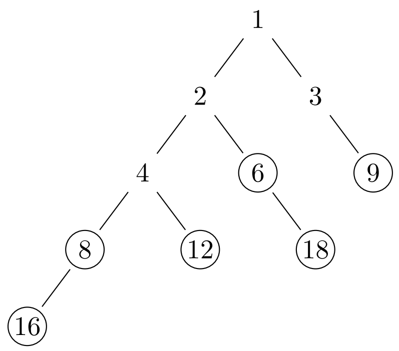
The non-smooth numbers are handled by Algorithm 4. It traverses the numbers that are not $(\alpha-1)$-smooth in the interval $[\alpha, \beta]$ and calls the visitor function for each non-smooth number and for all its integer multiples that fit in the interval. The computational complexity of this algorithm is determined by the sieve of Atkin, which runs in $O(\beta)$ time and uses $O(\beta^{1/2+o(1)})$ memory[19]. In other words, Algorithm 4 performs $O(1)$ operations for each non-smooth number that it visits, which doesn't change its time complexity class. Thus, the algorithm runs in $O(\beta)$ time and uses $O(\beta^{1/2+o(1)})$ memory.

Algorithm 3. Use an iterative depth-first search (DFS) to call a visitor function for each $(\alpha-1)$-smooth number m in the interval $[\alpha, \beta]$ and its Euler's totient ϕ. Runs in $O(\beta)$ time and uses $O(\log \beta)$ memory.

```
function ProcessSmoothNumbers($P, \alpha, \beta, visitor$)
    // The argument $P$ is a list that should consist of prime numbers in the interval $[1, \alpha-1]$.
    $S \leftarrow$ EmptyStack();
    Push($S$, $(1, 1, 0)$);
    while NotEmpty($S$) do
        $(m, \phi, j) \leftarrow$ Pop($S$);
        if $j <$ Length($P$) then
            Push($S$, $(m, \phi, j+1)$);    // advance to the next prime number in $P$
        else
            continue;    // no prime numbers left in $P$
        end if
        $p \leftarrow P[j]$;
        if $m \bmod p = 0$ then
            $\phi \leftarrow \phi \cdot p$;
        else
            $\phi \leftarrow \phi \cdot (p-1)$;
        end if
        $m \leftarrow m \cdot p$;
        if $m > \beta$ then
            Pop($S$);
        end if
        if $\alpha \leq m \leq \beta$ then
            visitor($m, \phi$);
        end if
        if $m < \beta$ then
            Push($S$, $(m, \phi, j)$);
        end if
    end while
end function
```

Algorithm 4. Use the sieve of Atkin to call a visitor function for each number that is not $(\alpha-1)$-smooth in the interval $[\alpha, \beta]$ and its Euler's totient ϕ. Runs in $O(\beta)$ time and uses $O(\beta^{1/2+o(1)})$ memory.

```
function ProcessNonSmoothNumbers($\varphi, \alpha, \beta, visitor$)
    // The argument $\varphi$ is an array that should contain Euler's totients for the integers in the interval $[1, \lfloor \beta/\alpha \rfloor]$.
    for $p$ in SieveOfAtkin($\alpha, \beta$) do
        $m \leftarrow p$;
        $j \leftarrow 1$;
        while $m \leq \beta$ do
            if $j \bmod p = 0$ then
                $\phi \leftarrow p \cdot \varphi[j]$;
            else
                $\phi \leftarrow (p-1) \cdot \varphi[j]$;
            end if
            visitor($m$, $\phi$);
            $j \leftarrow j + 1$;
            $m \leftarrow m + p$;
        end while
    end for
end function
```


(a) 4-smooth numbers.

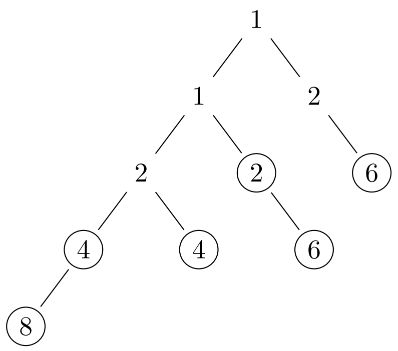
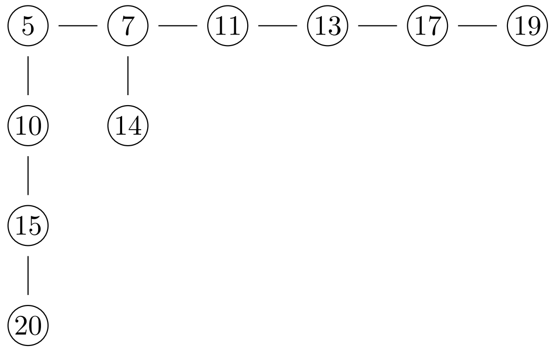

(b) Euler's totients.

Figure 2. Traversal of the 4-smooth numbers in the interval $[5, 20]$ by Algorithm 3 when $n = 20$. The tree of $\lfloor \sqrt{n} \rfloor$-smooth numbers is shown in (a). Their corresponding totients are shown in (b). The numbers for which the visitor function is called are circled. The list of prime numbers in this case is $P = (2, 3)$. Thus, each left branch in (a) multiplies the parent node by 2 and each right branch multiplies it by 3. If the product exceeds 20, then the corresponding branch or leaf is excluded from the tree.


(a) Not 4-smooth numbers.

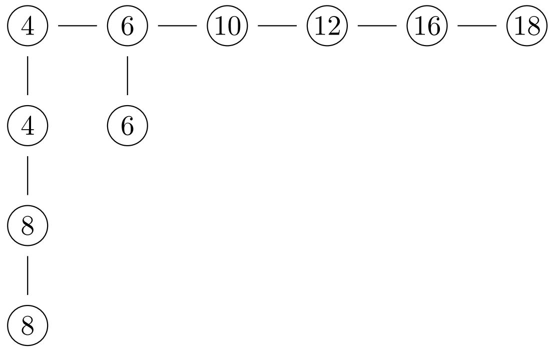

(b) Euler's totients.

Figure 3. Enumeration of not 4-smooth numbers in the interval $[5, 20]$ by Algorithm 4 when $n = 20$. The tree of non-smooth numbers is shown in (a), their corresponding totients are shown in (b). The visitor function is called for all circled numbers.

To give a concrete example, let $n = 20$. Figure 2a shows the search tree for the $\lfloor \sqrt{n} \rfloor$-smooth numbers in the interval $[5, 20]$. Algorithm 3 calls the visitor function for the pairs $(k, \varphi(k))$ where k is 4-smooth (i.e., $\lfloor \sqrt{20} \rfloor = 4$) and $\varphi(k)$ is its totient. The totients are shown in Figure 2b. The algorithm calls the visitor function only for $k \geq \lfloor \sqrt{20} + 1 \rfloor = 5$, i.e., only for the circled integers and totients shown in the figure. The tree in this case is binary because the list $P = (2, 3)$ contains only two prime numbers. Thus, all left branches multiply the parent node by 2 and all right branches multiply it by 3. The order of enumeration is: 1, 2, 4, 8, 16, 12, 6, 18, 3, 9 (i.e., pre-order traversal).

Figure 3 visualizes the enumeration of the numbers that are not $\lfloor \sqrt{n} \rfloor$-smooth for $n = 20$. Algorithm 4 calls the sieve of Atkin to generate the prime numbers in the interval $[5, 20]$, i.e., $5, 7, 11, 13, 17, 19$. This is done one-at-a-time, i.e., without storing the list of primes in memory. The algorithm also calls the visitor function for each pair $(k, \varphi(k))$, where k is each prime number or its integer multiples that fit in the interval. The enumeration order here is: $5, 10, 15, 20, 7, 14, 11, 13, 17$, and 19.

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$\varphi(k)$	1	1	2	2	4	2	6	4	6	4	10	4	12	6	8	8	16	6	18	8

Table 1. Values of Euler's totient function $\varphi(k)$ for all integers k between 1 and 20.

Table 1 shows the values of Euler's totient function $\varphi(k)$ for each k between 1 and 20. The sum of these totients is equal to 128. Therefore, $|F_{20}| = 1 + 128 = 129$. To compute this result, Algorithm 2 first adds $1 + \varphi(1) + \varphi(2) + \varphi(3) + \varphi(4) = 7$. Then, it calls Algorithm 3, which computes the sum of all circled numbers in Figure 2b (i.e., $4 + 8 + 4 + 2 + 6 + 6 = 30$) and adds it to 7. Finally, it calls Algorithm 4, which computes the circled numbers in Figure 3b and adds their sum (i.e., 92) to the previous result. Thus, the return value for $|F_{20}|$ is $7 + 30 + 92 = 129$.

6 Algorithm C

This section describes another algorithm for computing $|F_n|$. Unlike algorithms A and B, it does not sum Euler's totients. Instead, it uses formula (5), which expresses $|F_n|$ in terms of values of $|F_k|$ for $k < n$. The algorithm uses this formula multiple times and stores the computed values for $|F_k|$ in a lookup table, which is described in Supplementary Section S8.

Algorithm 5 gives the pseudo-code for algorithm C. First, the code sets the variables r and u to the upper limits of the two summations in formula (5), which are $\lfloor\sqrt{n}\rfloor$ and $u(n)$, respectively. Next, it creates an empty lookup table and initializes its first entry, which corresponds to $|F_1| = 2$. Then, the two for-loops fill the required entries in the lookup table in an order that makes it possible to eventually compute $|F_n|$. The first for-loop computes $|F_2|, |F_3|, |F_4|, \ldots, |F_{\lfloor\sqrt{n}\rfloor}|$ in that order. The second for-loop computes $|F_{\lfloor n/u(n)\rfloor}|, |F_{\lfloor n/(u(n)-1)\rfloor}|, |F_{\lfloor n/(u(n)-2)\rfloor}|, \ldots, |F_{\lfloor n/2\rfloor}|, |F_{\lfloor n/1\rfloor}|$. The value of $|F_n|$ is computed in the last iteration when the value of the variable j is equal to 1.

Each iteration through line 7 or line 10 corresponds to an instance of formula (5). A helper function implements this computation. Its pseudo-code is listed in Algorithm 6. It has two for-loops that correspond to the two summations in formula (5). Each call to the helper function stores the computed value of $|F_m|$ in the lookup table so that it is available later on.

Supplementary Section S10 proves that the order in which the algorithm fills the lookup table entries is correct. It shows that computing each subsequent entry requires accessing only those lookup table elements that have already been set in previous iterations. The appendix also proves that Algorithm 5 runs in $O(n^{3/4})$ time and uses $O(\sqrt{n})$ memory. Supplementary Section S11 gives a recursive version of this algorithm.

Algorithm 5. Compute the length of the Farey sequence F_n. Runs in $O(n^{3/4})$ time uses $O(\sqrt{n})$ memory.

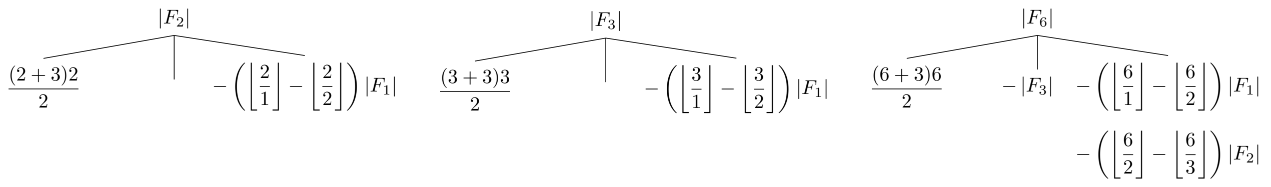
```
1:  function FAREYLENGTHC($n$)
2:      $r \leftarrow$ ISQRT($n$);                    // Compute $\lfloor\sqrt{n}\rfloor$ exactly using Algorithm S4
3:      $u \leftarrow \lfloor \frac{n}{r+1} \rfloor$;
4:      $F \leftarrow$ LOOKUPTABLE($n, r+1$);
5:      $F[1] \leftarrow 2$;
6:      for $m \leftarrow 2$ to $r$ do
7:          UPDATELOOKUPTABLE($F, m$);
8:      end for
9:      for $j \leftarrow u$ down to $1$ do
10:         UPDATELOOKUPTABLE($F, \lfloor n/j \rfloor$);
11:     end for
12:     return $F[n]$;
13: end function
```

Algorithm 6. Helper function used by multiple algorithms. Runs in $O(\sqrt{m})$ time.

```
1:  function UPDATELOOKUPTABLE($F, m$)
2:      ASSERT($m > 1$);
3:      $r \leftarrow$ ISQRT($m$);
4:      $u \leftarrow \lfloor \frac{m}{r+1} \rfloor$;
5:      $s \leftarrow 0$;
6:      for $k \leftarrow 2$ to $u$ do
7:          $q \leftarrow \lfloor m/k \rfloor$;
8:          ASSERT($q$ in $F$);
9:          $s \leftarrow s + F[q]$;
10:     end for
11:     for $k \leftarrow 1$ to $r$ do
12:         ASSERT($k$ in $F$);
13:         $s \leftarrow s + \left(\lfloor \frac{m}{k} \rfloor - \lfloor \frac{m}{k+1} \rfloor\right) \cdot F[k]$;
14:     end for
15:     $F[m] \leftarrow \frac{(m+3)\,m}{2} - s$;
16: end function
```

Figure 4 visualizes the computation performed by algorithm C (i.e., Algorithm 5) for $n = 6$. This process uses formula (5) three times. Each tree shown in this figure corresponds to one call to the helper function. The root node is the term $|F_m|$ that the helper function stores in the lookup table. The three branches show the terms in formula (5) that the function adds to compute the value of $|F_m|$. The first for-loop in algorithm C computes $|F_2|$ and stores its value in the lookup table. Figure 4a visualizes the corresponding instance of formula (5) in which the middle branch is empty. The second for-loop computes the values of $|F_3|$ and $|F_6|$. The corresponding trees are shown in Figure 4b and 4c, respectively. Note that there is no tree for $|F_1|$ because the algorithm stores its value in the lookup table during initialization, before the first for-loop. Also, note that the tree for $|F_6|$ uses the previously computed values for $|F_2|$ and $|F_3|$.

Supplementary Section S10 proves that for each $m > 1$, every term $|F_m|$ that appears in a branch of some tree also appears as a root node for one of the preceding trees. In other words, the order in which the algorithm fills the lookup table is correct.


$|F_2|$
$\frac{(2+3)2}{2}$
$-\left(\left\lfloor\frac{2}{1}\right\rfloor - \left\lfloor\frac{2}{2}\right\rfloor\right)|F_1|$
$|F_3|$
$\frac{(3+3)3}{2}$
$-\left(\left\lfloor\frac{3}{1}\right\rfloor - \left\lfloor\frac{3}{2}\right\rfloor\right)|F_1|$
$|F_6|$
$\frac{(6+3)6}{2}$
$-|F_3|$
$-\left(\left\lfloor\frac{6}{1}\right\rfloor - \left\lfloor\frac{6}{2}\right\rfloor\right)|F_1|$
$-\left(\left\lfloor\frac{6}{2}\right\rfloor - \left\lfloor\frac{6}{3}\right\rfloor\right)|F_2|$

(a) $|F_2| = 3$. **(b)** $|F_3| = 5$. **(c)** $|F_6| = 13$.

Figure 4. Visualizations of the three instances of formula (5) in the execution of algorithm C when it is called with $n = 6$. In this case, the first for-loop of the algorithm invokes the formula to compute $|F_2|$ as shown in (a). The second for-loop computes $|F_3|$ and $|F_6|$. The trees for these two terms are shown in (b) and (c), respectively. The result is $|F_6| = 13$.

7 Algorithm D

This section describes the fourth algorithm for computing $|F_n|$. Algorithm D can be viewed as an optimized version of algorithm C that runs faster, i.e., $O(n^{2/3})$ instead of $O(n^{3/4})$ time. This run-time improvement, however, is achieved at the expense of using more memory than algorithm C, i.e., $O(n^{2/3})$ instead of $O(\sqrt{n})$ memory. The pseudo-code is given in Algorithm 7.

The first for-loop in algorithm C computes the values of $|F_1|, |F_2|, \ldots, |F_{\lfloor\sqrt{n}\rfloor}|$ using the helper function, which implements formula (5). This process requires $O(n^{3/4})$ time. Because the orders of these Farey sequences form a contiguous range of integers, however, we can compute their lengths a bit faster. That is, they can be computed in $O(\sqrt{n})$ time by adding Euler's totients $\varphi(1), \varphi(2), \ldots, \varphi(\lfloor\sqrt{n}\rfloor)$. These totients, in turn, can be generated in $O(\sqrt{n})$ time using Algorithms S2 and S3.

Thus, the computation performed by the first for-loop in algorithm C can be implemented so that it runs in $O(\sqrt{n})$ instead of $O(n^{3/4})$ time. This optimization alone won't affect the overall computational complexity because the second for-loop would still run in $O(n^{3/4})$ time. It cannot be optimized in the same way as the first for-loop because the orders of the Farey sequences $F_{\lfloor n/u(n)\rfloor}, F_{\lfloor n/(u(n)-1)\rfloor}, \ldots, F_{\lfloor n/2\rfloor}, F_n$ do not form a contiguous range of integers.

In addition to the optimization described above, it is also possible to change the split point between the two stages of computation so that the first stage computes more than $\lfloor\sqrt{n}\rfloor$ entries of the lookup table. That is, instead of running the sieve-based approach up to $\lfloor\sqrt{n}\rfloor$, we can let it run for some number of additional iterations beyond that. Correspondingly, the starting value of the loop variable j in the second for-loop can be adjusted to be smaller than $u(n) = \lfloor n/(\lfloor\sqrt{n}\rfloor + 1)\rfloor$. Supplementary Section S12 shows that the optimal running time is achieved when the first stage runs up to $\lfloor n^{2/3}\rfloor$ and when the last for-loop starts from $v(n) = \lfloor n/(\lfloor n^{2/3}\rfloor + 1)\rfloor$ instead of $u(n)$.

Algorithm D is the result of making these changes to algorithm C. The loop on lines 10–14 computes $|F_1|, |F_2|, \ldots, |F_{\lfloor\sqrt{n}\rfloor}|$ by using the linear sieve to generate an array of totients and then summing them similarly to what algorithm A does. The for-loop on lines 15–23 implements formula (10) to compute $\left|F_{\left\lfloor\frac{n}{u(n)}\right\rfloor}\right|, \left|F_{\left\lfloor\frac{n}{u(n)-1}\right\rfloor}\right|, \ldots, \left|F_{\left\lfloor\frac{n}{u(n)-w(n)}\right\rfloor}\right|$ by adding the totients from the array φ. The last value is equivalent to $\left|F_{\left\lfloor\frac{n}{v(n)+1}\right\rfloor}\right|$ because $w(n) = u(n) - v(n) - 1$. The last for-loop (i.e., lines 24–26) computes $\left|F_{\left\lfloor\frac{n}{v(n)}\right\rfloor}\right|, \left|F_{\left\lfloor\frac{n}{v(n)-1}\right\rfloor}\right|, \ldots, \left|F_{\left\lfloor\frac{n}{2}\right\rfloor}\right|, \left|F_n\right|$ using formula (5) that is implemented by the helper function.

Supplementary Appendix S13 gives an alternative, shorter version of this algorithm in which the second for-loop is removed (i.e., lines 15–23). In that version, the first for-loop runs up to $m = c$, instead of $m = r$. This fills more entries in the lookup table than necessary for computing $|F_n|$, but the space complexity remains in $O(n^{2/3})$. The time complexity also remains in $O(n^{2/3})$. Algorithm 7 was selected for presentation in the main paper because it is easier to map it to the formulas and because in this form it is easier to understand the optimization performed by algorithm E, which is discussed next.

Algorithm 7. Compute the length of the Farey sequence F_n. Runs in $O(n^{2/3})$ time and uses $O(n^{2/3})$ memory.

```
function FAREYLENGTHD($n$)
    $r \leftarrow$ ISQRT($n$);                          // Compute $\lfloor\sqrt{n}\rfloor$ exactly using Algorithm S4
    $c \leftarrow$ ICBRT($n^2$);                         // Compute $\lfloor\sqrt[3]{n^2}\rfloor$ exactly using Algorithm S5
    $u \leftarrow \lfloor\frac{n}{r+1}\rfloor$;
    $v \leftarrow \lfloor\frac{n}{c+1}\rfloor$;
    $w \leftarrow u - v - 1$;
    $F \leftarrow$ LOOKUPTABLE($n$, $r+1$);
    $(P, L_p) \leftarrow$ LINEARSIEVE($c$);
    $\varphi \leftarrow$ COMPUTETOTIENTS($c, L_p$);
    $s \leftarrow 1$;
    for $m \leftarrow 1$ to $r$ do
        $s \leftarrow s + \varphi[m]$;
        $F[m] \leftarrow s$;
    end for
    $a \leftarrow r + 1$;
    for $i \leftarrow 0$ to $w$ do
        $b \leftarrow \lfloor\frac{n}{u-i}\rfloor$;
        for $m \leftarrow a$ to $b$ do
            $s \leftarrow s + \varphi[m]$;
        end for
        $F[b] \leftarrow s$;
        $a \leftarrow b + 1$;
    end for
    for $j \leftarrow v$ down to 1 do
        UPDATELOOKUPTABLE($F$, $\lfloor n/j\rfloor$);    // see Algorithm 6
    end for
    return $F[n]$;
end function
```

8 Algorithm E

This section describes algorithm E, which computes the length of a Farey sequence of order n. Its pseudo-code is shown in Algorithm 8. Its time complexity is $O(n^{2/3})$, which is the same as that of algorithm D. However, algorithm E uses only $O(\sqrt{n})$ memory. This improvement makes it the most efficient algorithm for computing $|F_n|$ described in this paper.

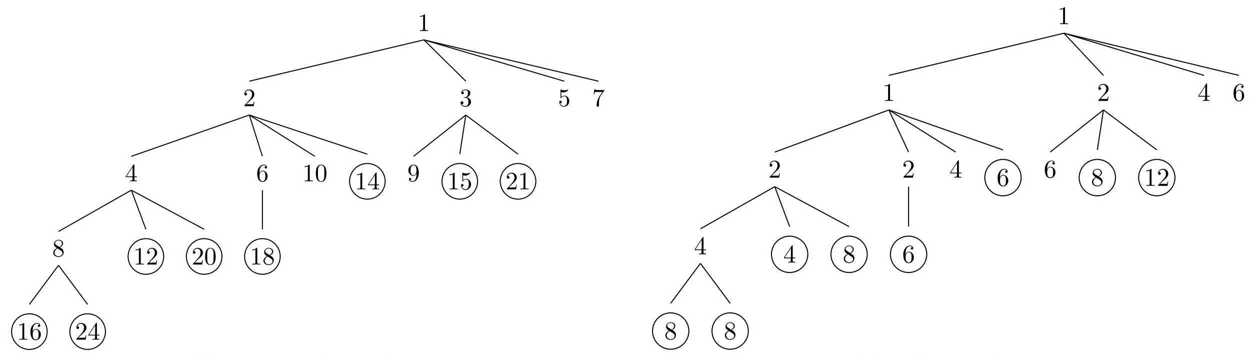
Algorithm E can be viewed as a modified version of algorithm D. It changes how the entries of the lookup table are computed for indices that fall in the interval $[\alpha, \beta]$, where $\alpha = \lfloor\sqrt{n}\rfloor + 1$ and $\beta = \lfloor n/(v(n)+1)\rfloor$. The process of computing all other entries of the lookup table is similar for both algorithms. That is, both algorithms use the linear sieve to compute the totients $\varphi(1), \varphi(2), \ldots, \varphi(\lfloor\sqrt{n}\rfloor)$ and then add them to compute $|F_1|, |F_2|, \ldots, |F_{\lfloor\sqrt{n}\rfloor}|$. Also, both algorithms use the helper function to compute the entries of the lookup table for indices that fall in the interval $[\beta+1, n]$, including n.

For indices in the interval $[\alpha, \beta]$, algorithm D uses the linear sieve. Because the value of β is in $O(n^{2/3})$, its memory complexity is $O(n^{2/3})$. In contrast, algorithm E uses the linear sieve only in the interval $[1, \alpha - 1]$ and $\alpha = \lfloor\sqrt{n}\rfloor$ is in $O(\sqrt{n})$. For each integer m in $[\alpha, \beta]$, algorithm E calls the visitor function with arguments m and $\varphi(m)$. More specifically, Algorithm 3 is used to process all $\lfloor\sqrt{n}\rfloor$-smooth numbers in that interval and Algorithm 4 is called to process all numbers that are not $\lfloor\sqrt{n}\rfloor$-smooth in the same interval. Algorithms 3 and 4 use only $O(\sqrt{n})$ memory because β is in $O(n^{2/3})$.

The visitor function determines the index i of the corresponding interval $I_i = [a_i, b_i]$ in formula (13) where m falls and then adds $\varphi(m)$ to the array element $B[i]$. After processing all integers in $[\alpha, \beta]$, the element $B[i]$ becomes equal to the sum of Euler's totients for all integers in I_i. Then, the algorithm computes the values of $|F_{b_i}|$ by adding the elements of the array B to $|F_{\lfloor\sqrt{n}\rfloor}|$. The length of the array B is equal to $w(n)+1$, which is in $O(\sqrt{n})$. This reduces the amount of memory required to process all integers in the interval $[\alpha, \beta]$. Therefore, the space complexity of algorithm E is $O(\sqrt{n})$. Supplementary Section S14 gives a formal proof for its time and space complexity.

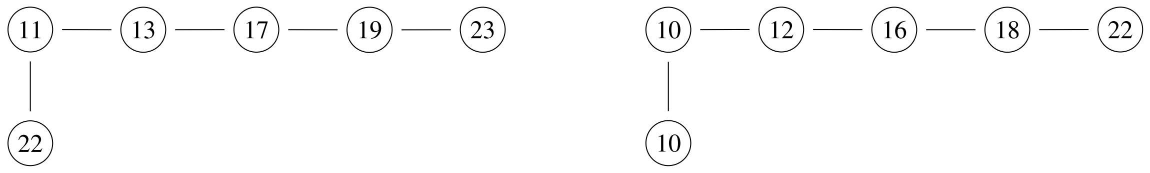
Algorithm 8. Compute the length of the Farey sequence F_n. Runs in $O(n^{2/3})$ time and uses $O(\sqrt{n})$ memory.

```
function FAREYLENGTHE($n$)
    $r \leftarrow$ ISQRT($n$);                                  // Compute $\lfloor \sqrt{n} \rfloor$ using Algorithm S4
    $c \leftarrow$ ICBRT($n^2$);                                // Compute $\lfloor \sqrt[3]{n^2} \rfloor$ using Algorithm S5
    $u \leftarrow \lfloor n/(r+1) \rfloor$;
    $v \leftarrow \lfloor n/(c+1) \rfloor$;
    $w \leftarrow u - v - 1$;
    $F \leftarrow$ LOOKUPTABLE($n$, $r+1$);
    $(P, L_p) \leftarrow$ LINEARSIEVE($r$);
    $\varphi \leftarrow$ COMPUTETOTIENTS($r, L_p$);
    $s \leftarrow 1$;
    for $m \leftarrow 1$ to $r$ do
        $s \leftarrow s + \varphi[m]$;
        $F[m] \leftarrow s$;
    end for
    $\alpha \leftarrow r + 1$;
    $\beta \leftarrow \lfloor n/(v+1) \rfloor$;
    if $\beta \geq \alpha$ then
        $B \leftarrow$ ZEROARRAY($w+1$);
        function VISITOR($m$, $\phi$)
            // The variables $B$, $n$, and $u$ are from the outer scope.
            $i \leftarrow u - \lfloor n/m \rfloor$;
            $B[i] \leftarrow B[i] + \phi$;
        end function
        PROCESSSMOOTHNUMBERS($P$, $\alpha$, $\beta$, VISITOR);        // $r$-smooth numbers in $[\alpha, \beta]$.
        PROCESSNONSMOOTHNUMBERS($\varphi$, $\alpha$, $\beta$, VISITOR);  // not $r$-smooth numbers in $[\alpha, \beta]$
        // Accumulate the computed partial sums into $F$.
        $s \leftarrow F[r]$;
        for $i \leftarrow 0$ to $w$ do
            $s \leftarrow s + B[i]$;
            $b \leftarrow \lfloor n/(u-i) \rfloor$;
            $F[b] \leftarrow s$;
        end for
    end if
    for $j \leftarrow v$ down to $1$ do
        UPDATELOOKUPTABLE($F$, $\lfloor n/j \rfloor$);              // see Algorithm 6
    end for
    return $F[n]$;
end function
```



(a) 10-smooth numbers. **(b)** Euler's totients.

Figure 5. Enumeration of 10-smooth numbers and their Euler's totients in the interval $[11, 24]$.



(a) Not 10-smooth numbers. **(b)** Euler's totients.

Figure 6. Enumeration of numbers that are not 10-smooth and their Euler's totients in the interval $[11, 24]$.

To give a concrete example, let's look at how algorithm E computes $|F_n|$ for $n = 120$. In this case, $\lfloor\sqrt{n}\rfloor = \lfloor\sqrt{120}\rfloor = 10$. First, on lines 8–9 the algorithm uses the linear sieve to compute the values of $\varphi(1), \varphi(2), \ldots, \varphi(\lfloor\sqrt{n}\rfloor)$. Supplementary Section S3 describes this process and gives the pseudo-code for the two algorithms that implement it. Second, the loop on lines 10–14 adds these totients to compute the values of $|F_1|, |F_2|, \ldots, |F_{10}|$. Essentially, this is an implementation of formula (1).

Next, on line 24, the algorithm processes the smooth numbers by invoking Algorithm 3. It calls a visitor function for each 10-smooth number in the interval $[\alpha, \beta]$, which in this case is equal to $[11, 24]$. Figure 5 visualizes this process using two trees. The circled nodes in Figure 5a show the 10-smooth numbers that the algorithm visits; the circled nodes in Figure 5b show their Euler's totients. The nodes that are not circled correspond to the 10-smooth numbers that are less than 11 and for which the visitor function is not called in Algorithm 3. The Euler's totients shown in Figure 5b are computed using formula (22). The algorithm visits the numbers in lexicographic order with respect to their prime factorizations, i.e., $2 \cdot 2 \cdot 2 \cdot 2 = 16$, $2 \cdot 2 \cdot 2 \cdot 3 = 24$, $2 \cdot 2 \cdot 3 = 12$, $2 \cdot 2 \cdot 5 = 20$, $2 \cdot 3 \cdot 3 = 18$, $2 \cdot 7 = 14$, $3 \cdot 5 = 15$, and $3 \cdot 7 = 21$.

On line 25, the algorithm processes the non-smooth numbers by calling Algorithm 4, which visits the numbers that are not 10-smooth in the interval $[11, 24]$ and their Euler's totients. Figure 6 visualizes this process. Algorithm 4 uses the sieve of Atkin to generate the prime numbers in this interval, i.e., 11, 13, 17, and 23. For each of them the algorithm iterates over their integer multiples that fit in the interval. In this example, this results in visiting $11 \cdot 2 = 22$ after 11, but before 13. Figure 6b shows the totients for each of the non-smooth numbers in Figure 6a. Their values are computed using formula (22).

The visitor function adds each circled totient from Figures 5b and 6b to the corresponding element of the array B. After visiting all integers in $[11, 24]$, the elements of B become equal to the sums of Euler's totients in the corresponding sub-interval. Then, the algorithm sums them with $|F_{10}|$ to compute $|F_{12}|$, $|F_{13}|$, $|F_{15}|$, $|F_{17}|$, $|F_{20}|$, and $|F_{24}|$. More formally,

$$\begin{array}{lll}
I_0 = [11, 12] & B[0] = \varphi(11) + \varphi(12) = 14 & |F_{12}| = |F_{10}| + B[0] = 33 + 14 = 47 \\
I_1 = [13, 13] & B[1] = \varphi(13) = 12 & |F_{13}| = |F_{12}| + B[1] = 47 + 12 = 59 \\
I_2 = [14, 15] & B[2] = \varphi(14) + \varphi(15) = 14 & |F_{15}| = |F_{13}| + B[2] = 59 + 14 = 73 \\
I_3 = [16, 17] & B[3] = \varphi(16) + \varphi(17) = 24 & |F_{17}| = |F_{15}| + B[3] = 73 + 24 = 97 \\
I_4 = [18, 20] & B[4] = \varphi(18) + \varphi(19) + \varphi(20) = 32 & |F_{20}| = |F_{17}| + B[4] = 97 + 32 = 129 \\
I_5 = [21, 24] & B[5] = \varphi(21) + \varphi(22) + \varphi(23) + \varphi(24) = 52 & |F_{24}| = |F_{20}| + B[5] = 129 + 52 = 181
\end{array}$$

Finally, on lines 34–36, the algorithm computes $|F_{30}|$, $|F_{40}|$, $|F_{60}|$, and $|F_{120}|$ using the helper function as shown in Figure 7. The colors in this figure indicate the three different methods that the algorithm uses to compute the lengths of Farey sequences and store them in the lookup table. That is, green corresponds to line 13, purple to line 31, and red to line 35.

$$\begin{array}{ccc}
 & |F_{30}| & \\
\frac{(30+3)30}{2} & -|F_{15}| & -\left(\left\lfloor\frac{30}{1}\right\rfloor-\left\lfloor\frac{30}{2}\right\rfloor\right)|F_1| \\
 & -|F_{10}| & -\left(\left\lfloor\frac{30}{2}\right\rfloor-\left\lfloor\frac{30}{3}\right\rfloor\right)|F_2| \\
 & -|F_{7}| & -\left(\left\lfloor\frac{30}{3}\right\rfloor-\left\lfloor\frac{30}{4}\right\rfloor\right)|F_3| \\
 & -|F_{6}| & -\left(\left\lfloor\frac{30}{4}\right\rfloor-\left\lfloor\frac{30}{5}\right\rfloor\right)|F_4| \\
 & & -\left(\left\lfloor\frac{30}{5}\right\rfloor-\left\lfloor\frac{30}{6}\right\rfloor\right)|F_5|
\end{array}$$

(a) $|F_{30}| = 279$.

$$\begin{array}{ccc}
 & |F_{40}| & \\
\frac{(40+3)40}{2} & -|F_{20}| & -\left(\left\lfloor\frac{40}{1}\right\rfloor-\left\lfloor\frac{40}{2}\right\rfloor\right)|F_1| \\
 & -|F_{13}| & -\left(\left\lfloor\frac{40}{2}\right\rfloor-\left\lfloor\frac{40}{3}\right\rfloor\right)|F_2| \\
 & -|F_{10}| & -\left(\left\lfloor\frac{40}{3}\right\rfloor-\left\lfloor\frac{40}{4}\right\rfloor\right)|F_3| \\
 & -|F_{8}| & -\left(\left\lfloor\frac{40}{4}\right\rfloor-\left\lfloor\frac{40}{5}\right\rfloor\right)|F_4| \\
 & & -\left(\left\lfloor\frac{40}{5}\right\rfloor-\left\lfloor\frac{40}{6}\right\rfloor\right)|F_5| \\
 & & -\left(\left\lfloor\frac{40}{6}\right\rfloor-\left\lfloor\frac{40}{7}\right\rfloor\right)|F_6|
\end{array}$$

(b) $|F_{40}| = 491$.

$$\begin{array}{ccc}
 & |F_{60}| & \\
\frac{(60+3)60}{2} & -|F_{30}| & -\left(\left\lfloor\frac{60}{1}\right\rfloor-\left\lfloor\frac{60}{2}\right\rfloor\right)|F_1| \\
 & -|F_{20}| & -\left(\left\lfloor\frac{60}{2}\right\rfloor-\left\lfloor\frac{60}{3}\right\rfloor\right)|F_2| \\
 & -|F_{15}| & -\left(\left\lfloor\frac{60}{3}\right\rfloor-\left\lfloor\frac{60}{4}\right\rfloor\right)|F_3| \\
 & -|F_{12}| & -\left(\left\lfloor\frac{60}{4}\right\rfloor-\left\lfloor\frac{60}{5}\right\rfloor\right)|F_4| \\
 & -|F_{10}| & -\left(\left\lfloor\frac{60}{5}\right\rfloor-\left\lfloor\frac{60}{6}\right\rfloor\right)|F_5| \\
 & -|F_{8}| & -\left(\left\lfloor\frac{60}{6}\right\rfloor-\left\lfloor\frac{60}{7}\right\rfloor\right)|F_6| \\
 & & -\left(\left\lfloor\frac{60}{7}\right\rfloor-\left\lfloor\frac{60}{8}\right\rfloor\right)|F_7|
\end{array}$$

(c) $|F_{60}| = 1103$.

$$\begin{array}{ccc}
 & |F_{120}| & \\
\frac{(120+3)120}{2} & -|F_{60}| & -\left(\left\lfloor\frac{120}{1}\right\rfloor-\left\lfloor\frac{120}{2}\right\rfloor\right)|F_1| \\
 & -|F_{40}| & -\left(\left\lfloor\frac{120}{2}\right\rfloor-\left\lfloor\frac{120}{3}\right\rfloor\right)|F_2| \\
 & -|F_{30}| & -\left(\left\lfloor\frac{120}{3}\right\rfloor-\left\lfloor\frac{120}{4}\right\rfloor\right)|F_3| \\
 & -|F_{24}| & -\left(\left\lfloor\frac{120}{4}\right\rfloor-\left\lfloor\frac{120}{5}\right\rfloor\right)|F_4| \\
 & -|F_{20}| & -\left(\left\lfloor\frac{120}{5}\right\rfloor-\left\lfloor\frac{120}{6}\right\rfloor\right)|F_5| \\
 & -|F_{17}| & -\left(\left\lfloor\frac{120}{6}\right\rfloor-\left\lfloor\frac{120}{7}\right\rfloor\right)|F_6| \\
 & -|F_{15}| & -\left(\left\lfloor\frac{120}{7}\right\rfloor-\left\lfloor\frac{120}{8}\right\rfloor\right)|F_7| \\
 & -|F_{13}| & -\left(\left\lfloor\frac{120}{8}\right\rfloor-\left\lfloor\frac{120}{9}\right\rfloor\right)|F_8| \\
 & -|F_{12}| & -\left(\left\lfloor\frac{120}{9}\right\rfloor-\left\lfloor\frac{120}{10}\right\rfloor\right)|F_9| \\
 & & -\left(\left\lfloor\frac{120}{10}\right\rfloor-\left\lfloor\frac{120}{11}\right\rfloor\right)|F_{10}|
\end{array}$$

(d) $|F_{120}| = 4387$.

Figure 7. Visualization of the last stage of algorithm E in which the helper function is used to compute $|F_{30}|$, $|F_{40}|$, $|F_{60}|$, and $|F_{120}|$ when the algorithm is called with $n = 120$. The three colors correspond to the three different methods for computing intermediate results and storing them in the lookup table. That is, green terms are set on line 13. Purple terms are processed on line 31. Finally, the red terms are computed on line 35, during the call to UPDATELOOKUPTABLE.

9 Results

We evaluated the performance of C and Python implementations of algorithms C, D, and E (i.e., Algorithms 5, 7, and 8). The value of n was varied between 10^0 and 10^{14} in increments of 0.5 on the decimal logarithm scale. For each of the 29 values of n, each of the three algorithms, and each of the two implementations, the evaluation program ran the code 10 times while measuring its run time, memory usage, and the number of CPU instructions (see Methods). We also ran algorithm E for n between 10^{10} and 10^{18} and recorded the corresponding values of $|F_n|$.

Figure 8 visualizes the results of the experiments. Figures 8a and 8b give the run time plots for C and Python, respectively. Figures 8c and 8d show the corresponding memory usage plots. All of them are on a log scale. The vertical coordinate of each point is obtained by averaging the results of the 10 code runs. Figure 8e lists the slopes and intercepts for lines fitted to each of the 12 curves in (a)–(d) using least squares. The fitting process used the region between $n = 10^{10}$ and $n = 10^{14}$. Finally, Figure 8f gives the values of $|F_n|$ for n between 10^{10} and 10^{18} computed using algorithm E.

The experiments confirm the theoretical time complexities of the algorithms. For each of the three algorithms and for each of the two implementations, the slopes are close to the theoretical predictions, i.e., 0.75 for algorithm C and $2/3 \approx 0.66$ for algorithms D and E. The slope for algorithm C agrees with the theory up to the third digit after the decimal point. The slopes for the other two algorithms are close to the theoretical predictions, but slightly higher by about 0.01 to 0.02. In other words, as n increases, these algorithms slow down slightly more than the theory predicts.

Even though this difference is small, we investigated it further by measuring the number of CPU instructions in addition to the run time. As described in Supplementary Section S15, that metric agrees with the theory very well. The slight deviation of the run time from the theory is due to practical aspects of code execution, including cache misses and inaccurate branch predictions. On average, this leads to slower execution of CPU instructions. The number of executed instructions, however, agrees with the theory as confirmed by Figure S4. The reason for this is that for our algorithms the total number of executed instructions is not affected by the need to access the main memory more frequently due to more cache misses. This number is also not affected by the decreased efficiency of the instruction pipeline that results from more frequent branch mispredictions.

For memory usage, the theoretically predicted slope is equal to 0.5 for algorithms C and E and $2/3$ for algorithm D. Figure 8 shows that there is a good match between the empirical results and the theoretical predictions for these algorithms. The slopes deviate from the theoretical predictions by less than 0.01 and there does not appear to be a significant positive or negative bias.

The slight jump in time and memory usage of the C version of algorithm E at $n = 10^3$ is due to the implementation of the sieve of Atkin. For small n it is optimized to return values from a hard-coded list of small prime numbers. That is, in this case there is no sieving. In our experiments, more computationally intensive sieving starts only when n reaches 10^3. For the Python version, the jump in run time occurs earlier, i.e., between 10^1 and 10^2. The reason for this is that our Python code runs the sieve of Atkin in a separate process. This process is started only when $n \geq 32 \approx 10^{1.5}$, which leads to a run time spike at that point. For smaller n, algorithm E does not call the sieve of Atkin and the Python code does not start the process.

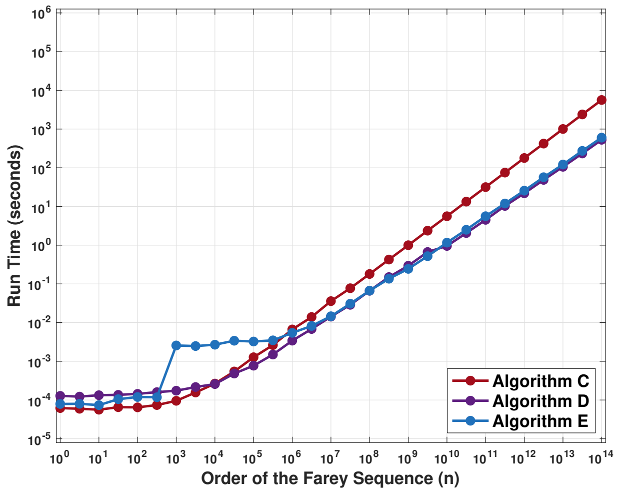
10 Conclusion

This paper introduced several novel formulas for the length $|F_n|$ of a Farey sequence of order n. They extend two classic results and combine them in different ways to achieve various trade-offs between iteration and recurrence. The paper also studied the problem of how to efficiently compute $|F_n|$. It described several algorithms that implement the formulas in ways that reduce both the computational time and the memory usage requirements. Our most efficient algorithm runs in $O(n^{2/3})$ time and uses only $O(\sqrt{n})$ memory. These properties make it the most efficient algorithm for computing $|F_n|$ that has been described so far.

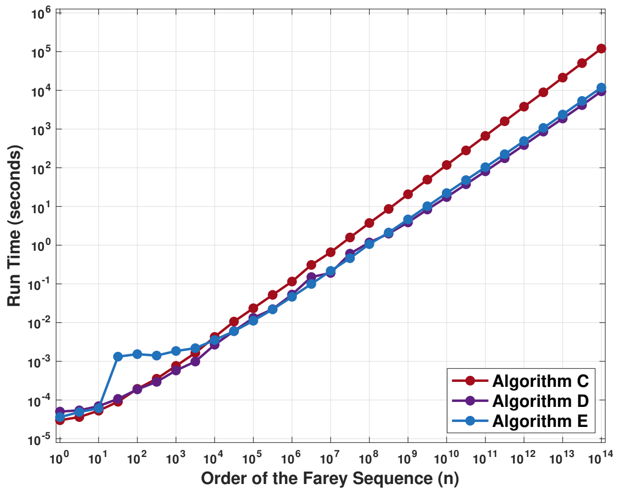
Algorithm E is based on formula (12). Even though this formula is long, it leads to the fastest algorithm. It combines the computational optimizations and approaches used by the other algorithms described in the paper. More specifically, it uses the linear sieve to help compute the lengths of Farey sequences of orders up to $\lfloor\sqrt{n}\rfloor$. Next, it enumerates smooth and non-smooth numbers in the interval $[\alpha, \beta]$, where $\alpha = \lfloor\sqrt{n}\rfloor + 1$, $\beta = \lfloor n/(v(n) + 1)\rfloor$, and $v(n) = \lfloor n/(\lfloor\sqrt[3]{n^2}\rfloor + 1)\rfloor$. The sieve of Atkin is used to enumerate the non-smooth numbers separately from the smooth numbers in order to improve the run time and memory usage while computing the values of $|F_m|$ in that interval. In its final stage, algorithm E uses formula (5) several times to compute $|F_n|$ using previously computed values of $|F_m|$ for $m < n$.

This paper also showed that the empirical time and memory usage of the algorithms agree with the corresponding theoretical time and space computational complexities. The experiments also showed that with algorithm E it is possible to compute the length of the Farey sequence of order 10^{18}. In other words, this paper makes it possible to explore the properties of $|F_n|$ for larger n than was previously possible, given the same amount of computational resources.

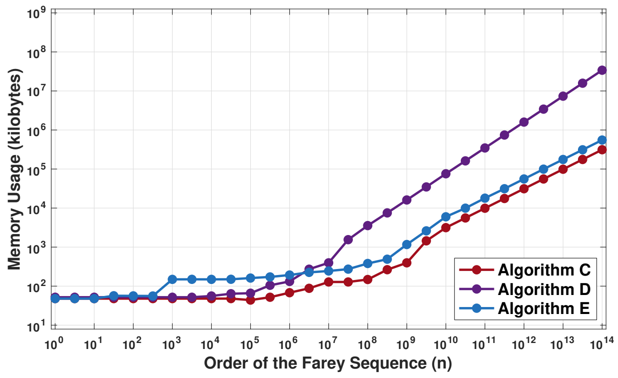
Future work could explore the applicability of other prime sieves[29,30], some of which may be faster and more compact than the sieves used in our algorithms. However, the time and space complexities of our most efficient algorithm are not tied directly to the prime sieves. They result from a combination of theoretical insights and computational techniques. In other words, merely switching to a more efficient prime sieve may not result in better time or space complexity without other changes.



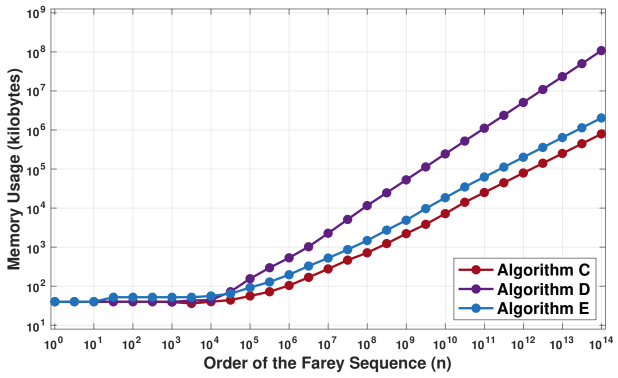
(a) Run time results for the C code.



(b) Run time results for the Python code.



(c) Memory usage results for the C code.



(d) Memory usage results for the Python code.

	Alg.	C Code		Python Code	
		Slope	Intercept	Slope	Intercept
Time	C	0.7507	-6.7584	0.7519	-5.4473
	D	0.6851	-6.8754	0.6824	-5.5939
	E	0.6771	-6.7085	0.6811	-5.4749
Memory	C	0.4992	-1.4947	0.5056	-1.1730
	D	0.6631	-1.7541	0.6614	-1.2299
	E	0.4944	-1.1835	0.5081	-0.7992

(e) Lines fitted to the curves in (a), (b), (c), and (d) for $n \geq 10^{10}$.

n	$\|F_n\|$
10^{10}	30396355092886216367
10^{11}	3039635509283386211141
10^{12}	303963550927059804025911
10^{13}	30396355092702898919527445
10^{14}	3039635509270144893910357855
10^{15}	303963550927013509478708835153
10^{16}	30396355092701332166351822199505
10^{17}	3039635509270133156701800820366347
10^{18}	303963550927013314319686824781290349

(f) Length of F_n for some large values of n.

Figure 8. Evaluation results for C and Python implementations of algorithms C, D, and E. The run time curves plotted in (a) and the memory usage curves plotted in (c) are for the C code. The run time and memory usage plots for the Python code are shown in (b) and (d). The plots use the log scale for both axes and each point represents the average of 10 runs. The table in (e) gives the slopes and intercepts for lines that were fitted to the twelve curves in (a)–(d) in the region between $n = 10^{10}$ and $n = 10^{14}$. The table in (f) shows the lengths of F_n computed with algorithm E for several large values of n.

Methods

The experiments evaluated the run time and memory usage of algorithms C, D, and E (i.e., Algorithms 5, 7, and 8). This was done for two different implementations of the algorithms and their dependencies: one written in C and another in Python. We used version 4.4.7 of GCC, the default C compiler on the experimental platform, to build the C code. It generated native binaries using the 64-bit version of the x86 instruction set. The performance of our Python code was measured with version 3.9.0 of the CPython interpreter[31], which serves as the reference implementation of the Python language.

Run time measurements. We used the same high-level Python script to run the evaluation and collect the time and memory usage measurements in all experiments. Each instance of each algorithm ran in a separate process that the evaluation script spawned before running the algorithm. This process exited after completing the run and transmitting the time and memory measurements to the evaluation script. For the Python implementation of the algorithms, this process called the corresponding Python function directly. For the C implementation, we compiled the C code into a shared library from which the evaluation process invoked the corresponding function using Python's 'ctypes' module.

In all cases, the run time of an algorithm was measured using the function 'time.process_time' in Python 3, i.e., by subtracting the return value of this function recorded right before launching the algorithm from its value returned right after the algorithm's completion. We disabled Python's garbage collector by calling 'gc.disable()' before starting an evaluation run. In other words, the run time measurements don't include the time that Python would normally spend on garbage collection, because the garbage collector was disabled.

Memory usage measurements. The memory usage of an algorithm was measured by subtracting the peak amount of physical memory used by the spawned process (after initialization but before launching the algorithm) from the peak amount of physical memory recorded after its completion. The memory usage plots report the value of this difference in kilobytes. The script obtained these values from the 'VmHWM' record in the special file '/proc/[pid]/status' provided[32] by our GNU/Linux system, where '[pid]' is the process identifier.

The evaluation script also monitored the number of virtual memory pages transferred from the physical RAM to the designated swap storage on the computer. We used the 'pswpout' record in the file '/proc/vmstat' made available by the OS. During the experiments, its value remained the same before and after running each instance of each algorithm, which implies that all virtual memory pages that our program used during these time intervals remained resident in RAM.

Counting CPU instructions. We used the perf-stat utility[33] provided by the OS to count the number of central processing unit (CPU) instructions that the code executed. Each process spawned by the evaluation script launched the command 'perf stat -e instructions -p [pid]', where '[pid]' was its process identifier. The standard output of perf-stat was redirected to a temporary file. The monitoring process started after initialization but before running the designated algorithm. After the algorithm finished, the spawned process sent the SIGINT signal[34] to the perf-stat process, which wrote the measurements to its standard output before exiting. The printed text included the number of instructions executed by the spawned process while perf-stat was monitoring. This information, together with the run time and memory usage statistics, was stored for subsequent analysis.

Averaging results from multiple runs. The logarithms of the run time, the memory usage, and the number of instructions were averaged over 10 independent runs. This was done for each of the two implementations (i.e., C and Python) of the three algorithms (i.e., C, D, and E). The order of the 29 values of n between 10^0 and 10^{14} used for the plots was randomized independently in each run. Multiple instances of the algorithms ran in parallel on our server to the extent that they could fit in the available memory without swapping. The number of simultaneously running algorithm instances never exceeded 15. The number of cores on the machine was 32. In other words, there were at least 2 cores available for each algorithm at run time.

Experimental platform. All results were computed on a 32-core Dell PowerEdge R720 server with 315 gigabytes of RAM. The processor on this machine was 2.20 GHz Intel Xeon E5-2660. The operating system was Red Hat Enterprise Linux (RHEL) version 6.10.

Native and long integers in Python. Python uses a unified implementation of integer arithmetic that automatically switches from native to long integers when necessary to avoid overflow[35]. On the experimental platform, the size of a native integer was equal to 64 bits. In other words, the Python interpreter automatically switched to long integers whenever it encountered an integer less than -2^{63} or greater than $2^{63}-1$. This switch occurs for $n > 10^{9.5}$.

128-bit integers in C. Our C code used 128-bit integers for all integer values that would not fit in 64 bits for large n. More specifically, we built the C code using GCC 4.4.7 and used the '__uint128_t' unsigned 128-bit integer type provided by this compiler.

Computational model. The theoretical model used for estimating the computational complexity of the algorithms assumes that adding, subtracting, multiplying, dividing, and storing any integer requires $O(1)$ time. Similarly, the model assumes that the size of an integer is also in $O(1)$. The 128-bit integers used in the C code were sufficiently large to avoid overflow in all experiments. The Python interpreter automatically switched to larger integers when necessary[35].

The sieve of Atkin. The experiments used a C implementation of the sieve of Atkin from the 'primegen' package[36], version 0.97, which supports generating prime numbers in any interval $[\alpha, \beta]$ where $\beta \leq 10^{15}$. This bound was sufficiently large for each of our experiments. This implementation uses a relatively small fixed-size static memory buffer instead of dynamic memory allocation. That is, even though in theory the sieve of Atkin requires $O(N^{1/2+o(1)})$ memory, in practice the memory usage of this particular implementation was constant. For $n > 10^{10}$ the memory usage for computing $|F_n|$ was dominated by the arrays used by our algorithms, i.e., the memory used by the sieve was only a tiny fraction of all memory used by the code.

Data availability

All data and procedures are described in the main paper or in the supplementary information.

References

1. Farey, J. On a curious property of vulgar fractions. *The Philos. Mag.* **47**, 385–386 (1816).

2. Hardy, G. & Wright, E. *An Introduction to the Theory of Numbers* (Oxford University Press, London, 1975), 4 edn.

3. Borwein, P., Choi, S., Rooney, B. & Weirathmueller, A. *The Riemann Hypothesis: A Resource for the Afficionado and Virtuoso Alike* (Springer, New York, 2007). pp. 48–49.

4. Graham, R., Knuth, D. & Patashnik, O. *Concrete Mathematics* (Addison-Wesley, Reading, MA, 1994), 2 edn.

5. Sukhoy, V. & Stoytchev, A. Numerical error analysis of the ICZT algorithm for chirp contours on the unit circle. *Sci. Reports* **10**, 4852 (2020).

6. Sukhoy, V. & Stoytchev, A. Generalizing the inverse FFT off the unit circle. *Sci. Reports* **9**, 14443 (2019).

7. Conway, J. & Guy, R. *The Book of Numbers* (Copernicus, New York, 1995), corrected edn. pp. 152–156.

8. Flegg, G., Hay, C. & Moss, B. *Nicolas Chuquet, Renaissance Mathematician: A Study with Extensive Translation of Chuquet's Mathematical Manuscript Completed in 1484* (Springer, New York, 1984), 1985 edn.

9. Guthery, S. *A Motif of Mathematics* (Docent Press, Boston, MA, 2011).

10. Cauchy, A.-L. Démonstation d'un theórème curieux sur les nombres (in French). *Bull. des Sci. par la Socièté Philomatique de Paris* **3**, 133–135 (1816).

11. Routledge, N. Computing Farey series. *The Mathematical Gazette* **92**, 55–62 (2008).

12. Routledge, N. Summing Euler's φ-function. *The Mathematical Gazette* **92**, 242–251 (2008).

13. OEIS Foundation Inc. The On-Line Encyclopedia of Integer Sequences. Sequence A005728 (2019). https://oeis.org/A005728.

14. Gauss, C. F. & translated by Clarke, A. *Disquisitiones Arithmeticae* (Yale University Press, New Haven, CT, 1965). Article 39.

15. Michie, D. "Memo" functions and machine learning. *Nature* **218**, 19–22 (1968).

16. Crandall, R. & Pomerance, C. *Prime Numbers: A Computational Perspective* (Springer, 2005), 2 edn. Chapter 3.

17. Bach, E. & Shallit, J. Algorithmic Number Theory, Volume 1: Efficient Algorithms (1996).

18. Gries, D. & Misra, J. A linear sieve algorithm for finding prime numbers. *Commun. ACM* **21**, 999–1003 (1978).

19. Atkin, A. O. L. & Bernstein, D. J. Prime sieves using binary quadratic forms. *Math. Comput.* **73**, 1023–1030 (2003).

20. Galbraith, S. *Mathematics of Public Key Cryptography* (Cambridge University Press, Cambridge, United Kingdom, 2012). Chapter 15.

21. Granville, A. Smooth numbers: computational number theory and beyond. In Buhler, J. & Stevenhagen, P. (eds.) *Algorithmic Number Theory: Lattices, Number Fields, Curves, and Cryptography*, vol. 44, 267–323 (MSRI Publications, Cambridge University Press, New York, 2008).

22. Pomerance, C. The role of smooth numbers in number theoretic algorithms. In *Proceedings of the International Congress of Mathematicians*, 411–422 (Birkhäuser, Basel, Switzerland, 1995).

23. Pătraşcu, C. & Pătraşcu, M. Computing order statistics in the Farey sequence. In Buell, D. (ed.) *Proceedings of the 6th International Symposium on Algorithmic Number Theory*, 358–366 (Burlington, VT, 2004).

24. Pawlewicz, J. Order statistics in the Farey sequences in sublinear time. In *Proceedings of the European Symposium on Algorithms*, 218–229 (Eilat, Israel, 2007).

25. Pawlewicz, J. & Pătraşcu, M. Order statistics in the Farey sequences in sublinear time and counting primitive lattice points in polygons. *Algorithmica* **55**, 271–282 (2009).

26. Deléglise & Rivat, J. Computing the summation of the Möbius function. *Exp. Math.* **5**, 291–295 (1996).

27. Hurst, G. Computations of the Mertens function and improved bounds on the Mertens conjecture. *Math. Comput.* **87**, 1013–1028 (2018).

28. Ye, Y. Combining binary search and Newton's method to compute real roots for a class of real functions. *J. Complex.* **10**, 271–280 (1994).

29. Pritchard, P. A sublinear additive sieve for finding prime numbers. *Commun. ACM* **24**, 18–23 (1981).

30. Sorenson, J. Two compact incremental prime sieves. *LMS J. Comput. Math.* **18**, 675–683 (2015).

31. The Python Software Foundation. Python Release 3.9.0 (2020). https://www.python.org/downloads/release/python-390/.

32. The Linux man-pages project. proc – process information pseudo-filesystem (2020). https://man7.org/linux/man-pages/man5/proc.5.html.

33. The Linux man-pages project. perf-stat – Run a command and gather performance counter statistics (2020). https://man7.org/linux/man-pages/man1/perf-stat.1.html.

34. The Linux man-pages project. signal – overview of signals (2020). https://man7.org/linux/man-pages/man7/signal.7.html.

35. Zadka, M. & van Rossum, G. PEP 237 – Unifying Long Integers and Integers (2001). https://www.python.org/dev/peps/pep-0237/.

36. Bernstein, D. J. Primegen: a small, fast library for generating prime numbers in order. (1999). Version 0.97. https://cr.yp.to/primegen.html.

Author contributions

V.S. developed the algorithms, wrote the evaluation code, and generated the tables and the figures. A.S. designed the scope of the study and the structure of the paper. A.S. advised on all experiments and supervised the work. Both authors wrote the paper.

Additional Information

Supplementary information was submitted together with the paper. Supplementary source code for all algorithms was also submitted with the paper.

Competing interests: The authors declare no competing interests.

Submitted: April 23, 2021; Revised: September 17, 2021.

Supplementary Information for "Formulas and Algorithms for the Length of a Farey Sequence"

Vladimir Sukhoy[1] & Alexander Stoytchev[1,*]

[1] Department of Electrical and Computer Engineering, Iowa State University, Ames, IA 50011, USA. Correspondence and requests for materials should be addressed to A.S. (email: alexs@iastate.edu).

S1. Algorithm for Enumerating and Counting the Elements of a Farey Sequence

For the Farey sequence F_n, the mediant property implies that the numerator p and the denominator q of the *third* fraction in a triple of adjacent fractions $({}^a/_b, {}^c/_d, {}^p/_q)$ can be expressed in terms of the values of a, b, c, and d as follows:

$$p = \left\lfloor \frac{n+b}{d} \right\rfloor c - a, \quad \text{and} \quad q = \left\lfloor \frac{n+b}{d} \right\rfloor d - b. \tag{15}$$

This pair of formulas enables an elegant algorithm[11] for enumerating the elements of F_n. The main idea is to use a window of adjacent fractions $({}^a/_b, {}^c/_d)$ that 'slides' over the elements of F_n until it reaches ${}^1/_1$. This window is initialized with the pair of fractions $({}^0/_1, {}^1/_n)$ that are the first two elements of the sequence F_n.

Algorithm S1 uses this approach to compute the length of the Farey sequence of order n by explicitly enumerating its elements and counting them. Because the length of F_n grows quadratically with n, this algorithm runs in $O(n^2)$ time, which makes it too slow for large values of n. Nevertheless, for small n, it can be used to cross-check the results of the faster algorithms described in the main paper. This algorithm uses a tiny amount of memory because it only prints the sequence elements and does not need to store them in memory. In other words, its space complexity is $O(1)$.

Algorithm S1. Enumerate all elements of the Farey sequence F_n and return $|F_n|$. Runs in $O(n^2)$ time and uses $O(1)$ memory.

```
function EnumerateFareyFractions($n$)
    $s \leftarrow 0$;
    $(a, b, c, d) \leftarrow (0, 1, 1, n)$;
    while true do
        $s \leftarrow s + 1$;
        Print$(a, \text{" / "}, b)$;
        if $a = b$ then
            break;
        end if
        $k \leftarrow \left\lfloor \frac{n+b}{d} \right\rfloor$;
        $(a, b, c, d) \leftarrow (c,\ d,\ kc - a,\ kd - b)$;
    end while
    return $s$;
end function
```

S2. Summary of the Time and Memory Complexities of the Algorithms

The main paper describes five algorithms for computing the length of a Farey sequence. To distinguish between them, the algorithms are denoted with the first five letters of the alphabet. These letters are also used as suffixes in the function name for each algorithm, e.g., FareyLengthA or FareyLengthE.

The algorithms are presented from the slowest to the fastest. This is done in order to explain the computational techniques and optimizations that were needed to derive the most efficient algorithm, i.e., algorithm E, which runs in $O(n^{2/3})$ time and uses $O(\sqrt{n})$ memory. Figure S1 provides a visual overview of the time and memory complexities of all five algorithms.

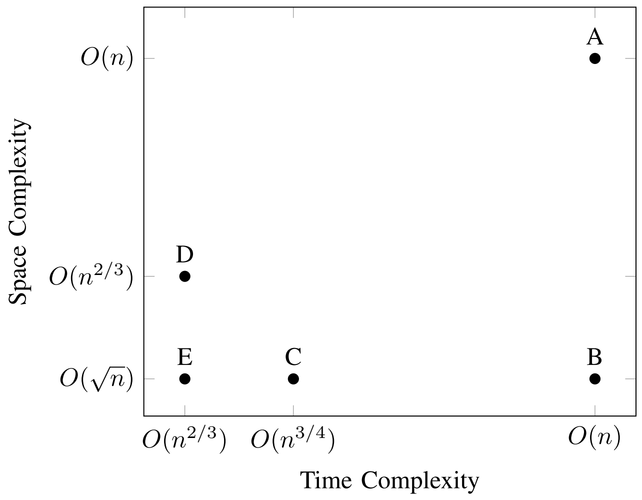


Figure S1: Visual summary of the time and space complexities of the five algorithms for computing the length of the Farey sequence of order n that are described in the main paper.

S3. Computing Euler's Totient Function

This section defines Euler's totient function $\varphi(n)$ and proves some of its properties. It also gives the pseudo-code for two algorithms that can be used to compute it. An example that illustrates these algorithms is also provided.

Definition 1. *Euler's totient function $\varphi(n)$ maps each positive integer n to the number of integers in the range $[1, n]$ that are coprime with n. More formally,*

$$\varphi(n) = \left|\left\{k \in \{1, 2, \ldots, n\} \text{ s.t. } \gcd(n, k) = 1\right\}\right|. \tag{16}$$

Theorem 1. *For each $n > 1$, the length of the Farey sequence of order n can be expressed as the sum of $\varphi(n)$ and the length of the Farey sequence of order $n - 1$. That is,*

$$|F_n| = |F_{n-1}| + \varphi(n), \qquad \textit{for each } n > 1. \tag{17}$$

Proof. Let D_n be a set that contains the elements of the sequence F_n that are not members of the sequence F_{n-1}, i.e.,

$$D_n = \left\{{}^{p}/_{q} \in F_n \text{ s.t. } {}^{p}/_{q} \notin F_{n-1}\right\}. \tag{18}$$

Then,

$$|F_n| = |F_{n-1}| + |D_n|. \tag{19}$$

It remains to show that $|D_n| = \varphi(n)$. For each irreducible fraction ${}^{p}/_{q} \in D_n$ the denominator q must be equal to n, otherwise ${}^{p}/_{q}$ would be in F_{n-1}. Thus, the cardinality of D_n is equal to the number of positive integers p between 1 and n for which the fraction ${}^{p}/_{n}$ is irreducible. This fraction is irreducible if and only if p is coprime with n. Therefore, $|D_n| = \varphi(n)$, which completes the proof. □

Theorem 2. *For each positive integer $n \geq 1$ the length of the Farey sequence of order n can be computed by adding 1 to the sum of the values of Euler's totient function from 1 to n, i.e.,*

$$|F_n| = 1 + \sum_{k=1}^{n} \varphi(k). \tag{20}$$

Proof. For $n = 1$ the formula reduces to $|F_1| = 1 + \varphi(1) = 1 + 1 = 2$, which is correct. For $n > 1$ the proof follows from Theorem 1, i.e.,

$$\begin{aligned} |F_n| &= |F_{n-1}| + \varphi(n) \\ &= |F_{n-2}| + \varphi(n-1) + \varphi(n) \\ &= |F_{n-3}| + \varphi(n-2) + \varphi(n-1) + \varphi(n) \\ &\ldots \\ &= |F_1| + \varphi(2) + \varphi(3) + \cdots + \varphi(n) \\ &= 1 + \varphi(1) + \varphi(2) + \varphi(3) + \cdots + \varphi(n) \\ &= 1 + \sum_{k=1}^{n} \varphi(k). \end{aligned} \tag{21}$$

□

Table S1 shows the values of Euler's totient function $\varphi(k)$ for all integer arguments k from 1 and 20. Table S2 shows the length of the Farey sequence F_n for values of n from 1 to 20. Using these numbers we can verify that formula (20) holds for some small values of n. For example, $|F_5| = 1 + \varphi(1) + \varphi(2) + \varphi(3) + \varphi(4) + \varphi(5) = 11$.

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$\varphi(k)$	1	1	2	2	4	2	6	4	6	4	10	4	12	6	8	8	16	6	18	8

Table S1: Values of Euler's totient function $\varphi(k)$ for k between 1 and 20.

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$\lvert F_n\rvert$	2	3	5	7	11	13	19	23	29	33	43	47	59	65	73	81	97	103	121	129

Table S2: The length of the Farey sequence $|F_n|$ for n between 1 and 20.

The following theorem gives a formula for computing the value of $\varphi(p \cdot k)$, where p is a prime number and k is any positive integer. This formula is used as an update rule by several algorithms described in this manuscript.

Theorem 3. *Let p be a prime number and let k be a positive integer. Then, the value of Euler's totient $\varphi(p \cdot k)$ can be expressed as follows:*

$$\varphi(p \cdot k) = \begin{cases} p \cdot \varphi(k), & \textit{if } p \textit{ is a factor of } k, \\ (p-1) \cdot \varphi(k), & \textit{if } p \textit{ is not a factor of } k. \end{cases} \tag{22}$$

Proof. Euler's product formula allows us to express the values of $\varphi(k)$ and $\varphi(p \cdot k)$ as shown below:

$$\varphi(k) = k \prod_{q|k} \left(1 - \frac{1}{q}\right), \tag{23}$$

$$\varphi(p \cdot k) = p \cdot k \prod_{q|k \cdot p} \left(1 - \frac{1}{q}\right), \tag{24}$$

where the products run over the distinct prime numbers q that divide k and $k \cdot p$, respectively.

If p is a factor of k, then $\varphi(p \cdot k)$ can be expressed as follows:

$$\varphi(p \cdot k) = p \cdot k \prod_{q|k \cdot p} \left(1 - \frac{1}{q}\right) = p \cdot k \prod_{q|k} \left(1 - \frac{1}{q}\right) = p \cdot \varphi(k), \tag{25}$$

which proves the first branch in formula (22).

If p is not a factor of k, then formulas (23) and (24) imply that $\varphi(p \cdot k)$ has the following value:

$$\varphi(p \cdot k) = p \cdot k \prod_{q|k \cdot p} \left(1 - \frac{1}{q}\right) = p \cdot k \prod_{q|k} \left(1 - \frac{1}{q}\right) \left(1 - \frac{1}{p}\right) = p \left(1 - \frac{1}{p}\right) k \prod_{q|k} \left(1 - \frac{1}{q}\right) = (p-1) \cdot \varphi(k), \tag{26}$$

which proves the second branch in formula (22). □

One way to compute Euler's totients for all integers between 1 and N is to use formula (22) with the array L_p that the linear sieve algorithm[18] generates. Algorithm S2 gives the pseudo-code for the linear sieve. Its main goal is to compute a list P that consists of all prime numbers in the interval $[1, N]$. The algorithm solves this problem in $O(N)$ time and uses $O(N)$ memory. In addition to P, the linear sieve algorithm also fills an array L_p that contains the smallest prime factor for each integer in the interval $[1, N]$ (the algorithm sets the value of $L_p[1]$ to 1 as a special case). The array L_p is sufficient for computing the Euler's totients $\varphi(1), \varphi(2), \ldots, \varphi(N)$ using a special case of formula (22) as described below.

Algorithm S3 fills the array φ such that its elements $\varphi[1], \varphi[2], \ldots, \varphi[N]$ are equal to Euler's totients $\varphi(1), \varphi(2), \ldots, \varphi(N)$. The algorithm starts by setting $\varphi[1]$ to $\varphi(1)$, which is equal to 1. Then, the algorithm iterates over m from 2 to N and sets $\varphi[m]$ to Euler's totient $\varphi(m)$. Each iteration of this for-loop uses the following version of formula (22):

$$\varphi(m) = \begin{cases} p \cdot \varphi(k), & \text{if } L_p[k] = p, \\ (p-1) \cdot \varphi(k), & \text{if } L_p[k] \neq p, \end{cases} \tag{27}$$

where $p = L_p[m]$ is the smallest prime number that divides m and $k = \lfloor m/p \rfloor = m/p$ (we use the floor function to indicate that integer division should be used here). In other words, $m = k \cdot p$. Because p is the smallest prime number that divides m, the prime factorization of m can be derived from the prime factorization of k by adding one more factor p to it. This also implies that the smallest prime factor of k must be greater than or equal to p, i.e., $L_p[k] \geq L_p[k \cdot p] = p$. Moreover, p is a factor of k if and only if $L_p[k] = p$. Conversely, if $L_p[k] \neq p$, then the prime number p is not a factor of k. This logic leads to formula (27). Algorithm S3 runs in $O(N)$ time because it performs $O(1)$ operations for each integer between 1 and N. The space complexity of this algorithm is also in $O(N)$ because it creates an array φ that consists of N integers.

Algorithm S2. Find all prime numbers in the interval $[2, N]$ using a linear sieve. This algorithm also returns an array L_p that contains the smallest prime factor for each integer in $[2, N]$. Runs in $O(N)$ time and uses $O(N)$ memory.

```
function LinearSieve($N$)
    $P \leftarrow$ EmptyList( );             // list of all prime numbers in the interval $[2, N]$
    $L_p \leftarrow$ NullArray($N$+1);
    $L_p[1] \leftarrow 1$;                   // array of smallest prime factors
    for $k \leftarrow 2$ to $N$ do
        if $L_p[k]$ is null then
            // $k$ is a prime number
            $L_p[k] \leftarrow k$;
            Append($P$, $k$);
        end if
        for $p$ in $P$ do
            $q \leftarrow p \cdot k$;
            if $p > L_p[k]$ or $q > N$ then
                break;
            end if
            $L_p[q] \leftarrow p$;
        end for
    end for
    return ($P$, $L_p$);
end function
```

Algorithm S3. Compute Euler's totient function $\varphi(m)$ for each $m \in \{1, 2, \dots, N\}$ by traversing the array of smallest prime factors L_p generated by the linear sieve algorithm. This algorithm runs in $O(N)$ time and uses $O(N)$ memory.

```
function ComputeTotients($N$, $L_p$)
    $\varphi \leftarrow$ EmptyArray($N$+1);          // values of Euler's totient
    $\varphi[1] \leftarrow 1$;
    for $m \leftarrow 2$ to $N$ do
        $p \leftarrow L_p[m]$;
        $k \leftarrow \lfloor m/p \rfloor$;
        if $L_p[k] = p$ then
            $\varphi[m] \leftarrow p \cdot \varphi[k]$;
        else
            $\varphi[m] \leftarrow (p - 1) \cdot \varphi[k]$;
        end if
    end for
    return $\varphi$;
end function
```

Figure S2 visualizes a sample run of the linear sieve (i.e., Algorithm S2) with $N = 5$. It shows the state of the array of smallest prime factors L_p and the list of prime numbers P for $k \in \{1, 2, 3, 4, 5\}$. The elements of L_p and P that change after each iteration are colored in red. All elements of L_p are initially set to null, which is indicated with $\emptyset$ in the figure. The list P starts empty.

k	L_p	P
1	[1, ∅, ∅, ∅, ∅]	()
2	[1, 2, ∅, 2, ∅]	(2)
3	[1, 2, 3, 2, ∅]	(2, 3)
4	[1, 2, 3, 2, ∅]	(2, 3)
5	[1, 2, 3, 2, 5]	(2, 3, 5)

Figure S2: Illustration of the linear sieve algorithm for $N = 5$.

The algorithm starts by setting $L_p[1]$ to 1. Then, it proceeds to the for-loop that starts at $k = 2$. Because $L_p[k]$ is null at the beginning of this iteration, the algorithm marks $k = 2$ as a prime number by setting $L_p[2]$ to 2 and appending 2 to the list P. Subsequently, the algorithm iterates over each p in $P = (2)$, sets the variable q to the product $p \cdot k$, and then sets $L_p[q]$ to p until q exceeds N. In our example, this leads to setting $L_p[4]$ to 2. Next, the algorithm proceeds to $k = 3$. Because $L_p[k]$ is null here, this value of k is also a prime number, which leads to setting $L_p[3]$ to 3 and appending 3 to P. No other elements of L_p change because $2 \cdot 3 > N = 5$, which leads to breaking out of the inner for-loop on its first iteration. Subsequently, the algorithm continues to $k = 4$. Because $L_p[k]$ is not null, this value of k is not a prime number and the list P remains unchanged here. The inner for-loop also terminates early and does not modify any elements of L_p. Finally, the algorithm advances to $k = 5$. The value of $L_p[k]$ is null here, which leads to setting $L_p[5]$ to 5 and appending 5 to the list P (i.e., 5 is a prime number). The inner for loop, again, terminates early and the function returns the final state of P and L_p.

Figure S3 shows an example for Algorithm S3 with $N = 5$. This algorithm computes Euler's totients $\varphi(1)$, $\varphi(2)$, $\varphi(3)$, $\varphi(4)$, and $\varphi(5)$ using the array L_p computed by the linear sieve algorithm in the previous example, i.e., $L_p = [1, 2, 3, 2, 5]$. Algorithm S3 stores the computed totients in the array φ, which starts uninitialized. The uninitialized elements of φ are denoted with the symbol $\emptyset$ in the figure.

m	L_p	$p = L_p[m]$	$k = \lfloor m/p \rfloor$	$L_p[k]$	φ	Computing $\varphi[m]$
1	[1, 2, 3, 2, 5]	1	1	1	[1, ∅, ∅, ∅, ∅]	$\varphi[1] = 1$
2	[1, 2, 3, 2, 5]	2	1	1	[1, 1, ∅, ∅, ∅]	$\varphi[2] = (p-1) \cdot \varphi[k] = (2-1) \cdot 1 = 1$
3	[1, 2, 3, 2, 5]	3	1	1	[1, 1, 2, ∅, ∅]	$\varphi[3] = (p-1) \cdot \varphi[k] = (3-1) \cdot 1 = 2$
4	[1, 2, 3, 2, 5]	2	2	2	[1, 1, 2, 2, ∅]	$\varphi[4] = p \cdot \varphi[k] = 2 \cdot 1 = 2$
5	[1, 2, 3, 2, 5]	5	1	1	[1, 1, 2, 2, 4]	$\varphi[5] = (p-1) \cdot \varphi[k] = (5-1) \cdot 1 = 4$

Figure S3: Illustration of the algorithm for computing Euler's totients with $N = 5$.

The algorithm starts by setting $\varphi[1]$ to 1. Then, it continues to the for-loop, which starts at $m = 2$. It sets the value of the variable p to $L_p[m]$, which in this case is equal to 2. The variable k is set to $\lfloor m/p \rfloor = \lfloor 2/2 \rfloor = 1$. In this iteration, $L_p[k] \neq p$ and therefore the value of $\varphi[m]$ is set to $(p - 1) \cdot \varphi[k] = (2 - 1) \cdot \varphi[1] = 1$. Next, the algorithm proceeds to $m = 3$. It sets the variable p to $L_p[m]$, which is equal to 3 and sets the variable k to $\lfloor m/p \rfloor$, which is equal to 1. The if-statement, again, sets the value of $\varphi[m]$ to $(p - 1) \cdot \varphi[k] = (3 - 1) \cdot 1 = 2$. Subsequently, the algorithm proceeds to $m = 4$. In this iteration, p becomes 2 because $L_p[4] = 2$. The variable k is also set to 2, because $\lfloor m/p \rfloor = \lfloor 4/2 \rfloor = 2$. In this case, $L_p[k] = p$ and therefore the first branch of the if-statement is executed, which sets $\varphi[m]$ to $p \cdot \varphi[k] = 2 \cdot \varphi[2] = 2$. Finally, the algorithm continues to $m = 5$. Here, p becomes equal to $5 = L_p[m]$ and k becomes equal to $1 = \lfloor 5/5 \rfloor$. Because $L_p[k] \neq p$, the value of $\varphi[m]$ is set to $(p - 1) \cdot \varphi[k] = (5 - 1) \cdot 1 = 4$. After the last iteration, the function returns the final state of the array φ.

S4. Lemmas Used in the Proofs

The following lemma is used in the proof of Theorem 13.

Lemma 4. *Let k and n be two positive integers and let $S(k,n)$ be a set of all integers m such that $\lfloor n/m \rfloor = k$, i.e.,*

$$S(k,n) = \big\{ m \in \mathbb{N} \ \text{ s.t. } \ \lfloor n/m \rfloor = k \big\}. \tag{28}$$

Then, the set $S(k,n)$ consists of all integers in the interval $\big(\lfloor \frac{n}{k+1} \rfloor, \lfloor \frac{n}{k} \rfloor\big]$, i.e.,

$$S(k,n) = \left\{ \left\lfloor \frac{n}{k+1} \right\rfloor + 1,\ \left\lfloor \frac{n}{k+1} \right\rfloor + 2, \ldots,\ \left\lfloor \frac{n}{k} \right\rfloor \right\}. \tag{29}$$

Moreover, if $k_1 \neq k_2$, then the sets $S(k_1,n)$ and $S(k_2,n)$ are disjoint, i.e.,

$$S(k_1,n) \cap S(k_2,n) = \emptyset, \quad \text{if } \ k_1 \neq k_2. \tag{30}$$

Finally, the union of the sets $S(k,n)$ over all $k \in \{1,2,\ldots,\lfloor \sqrt{n} \rfloor\}$ is equal to the set of integers that fall in the interval $[u(n){+}1, n]$, where $u(n) = \left\lfloor \frac{n}{\lfloor \sqrt{n} \rfloor + 1} \right\rfloor$. That is,

$$\bigcup_{k=1}^{\lfloor \sqrt{n} \rfloor} S(k,n) = \{u(n)+1,\ u(n)+2, \ldots,\ n\}. \tag{31}$$

Proof. First, we will prove formula (29). Let $m \in S(k,n)$. Then, $\lfloor n/m \rfloor = k$, i.e.,

$$k \le \frac{n}{m} < k+1. \tag{32}$$

Reciprocating the three terms leads to:

$$\frac{1}{k+1} < \frac{m}{n} \le \frac{1}{k}. \tag{33}$$

Multiplying this inequality by n leads to the following interval for the value of m:

$$\frac{n}{k+1} < m \le \frac{n}{k}. \tag{34}$$

Because m is an integer, the floor function can be applied to the lower and upper bounds for the value of m in the previous inequality, i.e.,

$$\left\lfloor \frac{n}{k+1} \right\rfloor < m \le \left\lfloor \frac{n}{k} \right\rfloor. \tag{35}$$

Therefore, $m \in \big(\lfloor \frac{n}{k+1} \rfloor, \lfloor \frac{n}{k} \rfloor\big]$, which implies that $S(k,n) \subseteq \big(\lfloor \frac{n}{k+1} \rfloor, \lfloor \frac{n}{k} \rfloor\big]$.

Conversely, let m be an integer that lies in the interval $\big(\lfloor \frac{n}{k+1} \rfloor, \lfloor \frac{n}{k} \rfloor\big]$. That is,

$$\left\lfloor \frac{n}{k+1} \right\rfloor + 1 \le m \le \left\lfloor \frac{n}{k} \right\rfloor. \tag{36}$$

Dividing by n and reciprocating leads to:

$$\frac{n}{\lfloor \frac{n}{k} \rfloor} \le \frac{n}{m} \le \frac{n}{\lfloor \frac{n}{k+1} \rfloor + 1}. \tag{37}$$

Because $\lfloor n/k \rfloor \le n/k$, the lower bound can be stated as:

$$k = \frac{n}{\frac{n}{k}} \le \frac{n}{\lfloor \frac{n}{k} \rfloor} \le \frac{n}{m}. \tag{38}$$

On the other hand, $\lfloor \frac{n}{k+1} \rfloor + 1 > \frac{n}{k+1}$. Therefore, the upper bound in (37) can be expressed as:

$$\frac{n}{m} \le \frac{n}{\lfloor \frac{n}{k+1} \rfloor + 1} < \frac{n}{\frac{n}{k+1}} = k+1. \tag{39}$$

Inequalities (38) and (39) imply that $k \le n/m < k+1$ and, thus, $\lfloor n/m \rfloor = k$. Therefore, $\mathbb{N} \cap \big(\lfloor \frac{n}{k+1} \rfloor, \lfloor \frac{n}{k} \rfloor\big] \subseteq S(k,n)$, which proves formula (29).

Formula (30) is proven by contradiction. Suppose that there is a pair of integers k_1 and k_2 such that $k_1 \neq k_2$ and that the intersection of $S(k_1,n)$ and $S(k_2,n)$ is not empty. Then, there is an integer m that is an element of both sets, which implies that $\lfloor n/m \rfloor = k_1$ and $\lfloor n/m \rfloor = k_2$. Thus, $k_1 = k_2$, which is a contradiction that proves formula (30).

Formula (31) is proven by showing that the union of the sets in the left-hand side forms a contiguous range of integers that matches the set in the right-hand side. In this case, larger values of k correspond to smaller elements of $S(k,n)$. From formula (29) we can derive that the maximum element in the set $S(k+1,n)$ can be obtained by decrementing the minimum element in the set $S(k,n)$. More formally,

$$\max S(k+1,n) = \left\lfloor \frac{n}{k+1} \right\rfloor = \left\lfloor \frac{n}{k+1} \right\rfloor + 1 - 1 = \min S(k,n) - 1. \tag{40}$$

The minimum element in the set $S(\lfloor\sqrt{n}\rfloor, n)$ is equal to $\left\lfloor \frac{n}{\lfloor\sqrt{n}\rfloor+1} \right\rfloor + 1 = u(n) + 1$. The maximum element in the set $S(1,n)$ is equal to n. Thus, all integers between $u(n)+1$ and n are elements of the union. □

Many formulas in this manuscript use the function $u(n) = \lfloor n/(\lfloor\sqrt{n}\rfloor + 1)\rfloor$ to simplify the notation. Depending on the value of the integer n, this function evaluates to either $\lfloor\sqrt{n}\rfloor - 1$ or $\lfloor\sqrt{n}\rfloor$. For example, if $n = 4$, then it evaluates to $1 = \lfloor\sqrt{4}\rfloor - 1$. For $n = 6$, however, it evaluates to $2 = \lfloor\sqrt{6}\rfloor$. The next lemma formally proves this property.

Lemma 5. *For any positive integer n, the value of the function $u(n) = \lfloor {}^{n}/(\lfloor\sqrt{n}\rfloor+1) \rfloor$ is equal to either $\lfloor\sqrt{n}\rfloor - 1$ or $\lfloor\sqrt{n}\rfloor$.*

Proof. For the fraction in the definition of $u(n)$, the value of the denominator lies between $\sqrt{n}$ and $\sqrt{n}+1$, i.e.,

$$\sqrt{n} < \lfloor\sqrt{n}\rfloor + 1 \leq \sqrt{n} + 1. \tag{41}$$

Reciprocating all three values changes the direction of the inequality and leads to:

$$\frac{1}{\sqrt{n}+1} \leq \frac{1}{\lfloor\sqrt{n}\rfloor + 1} < \frac{1}{\sqrt{n}}. \tag{42}$$

Multiplying by n leads to the following bounds for the fraction in $u(n)$ that is used as the argument of the floor function:

$$\frac{n}{\sqrt{n}+1} \leq \frac{n}{\lfloor\sqrt{n}\rfloor + 1} < \frac{n}{\sqrt{n}} = \sqrt{n}. \tag{43}$$

The value of the term $n/(\sqrt{n}+1)$ in the left-hand side can be expressed as follows:

$$\frac{n}{\sqrt{n}+1} = \frac{n-1+1}{\sqrt{n}+1} = \frac{(\sqrt{n}-1)(\sqrt{n}+1)+1}{\sqrt{n}+1} = \sqrt{n} - 1 + \frac{1}{\sqrt{n}+1}. \tag{44}$$

Therefore,

$$\sqrt{n} - 1 < \frac{n}{\sqrt{n}+1}. \tag{45}$$

Combining inequality (43) with inequality (45) leads to the following formula:

$$\sqrt{n} - 1 < \frac{n}{\lfloor\sqrt{n}\rfloor + 1} < \sqrt{n}. \tag{46}$$

Applying the floor function to the previous inequality makes the middle term equal to $u(n)$. It also changes the inequality such that it is no longer strict when $\sqrt{n}$ is an integer. This completes the proof, i.e.,

$$\lfloor\sqrt{n}\rfloor - 1 = \lfloor\sqrt{n} - 1\rfloor \leq u(n) \leq \lfloor\sqrt{n}\rfloor, \tag{47}$$

where $\lfloor\sqrt{n}\rfloor - 1 = \lfloor\sqrt{n} - 1\rfloor$ follows from the definition of the floor function. □

The following two lemmas are used in the proof of Theorem 12. They express the value of $\lfloor\frac{n+1}{k}\rfloor$ in two ways depending on whether k divides $n+1$ or not.

Lemma 6. *Let n be a positive integer and let k be an integer between 1 and n. If k does not divide $n+1$, then the value of $\left\lfloor\frac{n+1}{k}\right\rfloor$ is equal to $\left\lfloor\frac{n}{k}\right\rfloor$. That is,*

$$\left\lfloor \frac{n+1}{k} \right\rfloor = \left\lfloor \frac{n}{k} \right\rfloor, \quad \textit{if } (n{+}1) \bmod k \neq 0. \tag{48}$$

Proof. The properties of integer division imply that there is exactly one pair of integers x and y such that the value of $n+1$ can be expressed in the following form:

$$n + 1 = xk + y, \tag{49}$$

where $0 \leq y < k$. In this case, $x = \left\lfloor\frac{n+1}{k}\right\rfloor$ is the quotient and $y = (n{+}1) \bmod k$ is the remainder. The lemma states that k does not divide $n+1$, which implies that y cannot be zero, i.e., $1 \leq y < k$.

Subtracting 1 from both sides of equation (49) leads to the following expression for the value of n:

$$n = xk + y - 1. \tag{50}$$

The value of the term $y - 1$ is an integer that lies in the interval $[0, k - 1)$. Once again, the uniqueness of integer division implies that $y - 1 = n \bmod k$ and that $x = \left\lfloor \frac{n}{k} \right\rfloor$, which completes the proof. □

Lemma 7. *Let n be a positive integer. Also, let k be an integer between 1 and n. If k divides $n+1$, then the value of $\left\lfloor \frac{n+1}{k} \right\rfloor$ is a whole number equal to $\left\lfloor \frac{n}{k} \right\rfloor + 1$, i.e.,*

$$\left\lfloor \frac{n+1}{k} \right\rfloor = \frac{n+1}{k} = \left\lfloor \frac{n}{k} \right\rfloor + 1, \quad \textit{if } (n+1) \bmod k = 0. \tag{51}$$

Proof. Similarly to the previous proof, the properties of integer division imply that:

$$n + 1 = xk + y = xk, \tag{52}$$

where $x = \left\lfloor \frac{n+1}{k} \right\rfloor$ and $y = (n+1) \bmod k = 0$. In this case, $y = 0$ because $n + 1$ is evenly divisible by k.

Once again, we can subtract 1 from both sides of (52), which leads to the following formula for the value of n:

$$n = xk - 1. \tag{53}$$

Furthermore, we can express xk as $(x - 1)k + k$, which leads to

$$n = (x - 1)k + k - 1. \tag{54}$$

Here the uniqueness of integer division implies that $\left\lfloor \frac{n}{k} \right\rfloor = x - 1$. Recalling that $x = \left\lfloor \frac{n+1}{k} \right\rfloor$ completes the proof. □

The next lemma proves an interesting property of the floor function, which is used in the correctness proof for Algorithm C.

Lemma 8. *Let k, j, and n be three positive integers. Then,*

$$\left\lfloor \frac{\lfloor n/k \rfloor}{j} \right\rfloor = \left\lfloor \frac{n/k}{j} \right\rfloor. \tag{55}$$

Proof. Similarly to the previous two lemmas, we start by using integer division to express the value of $\lfloor n/k \rfloor$ as follows:

$$\lfloor n/k \rfloor = xj + y, \tag{56}$$

where $x = \left\lfloor \frac{\lfloor n/k \rfloor}{j} \right\rfloor$ and $y = \lfloor n/k \rfloor \bmod j$. The integer y lies in the closed interval $[0, j - 1]$.

The value of n/k is equal to the sum of its integer and fractional parts, i.e.,

$$n/k = \lfloor n/k \rfloor + f, \tag{57}$$

where $f = n/k - \lfloor n/k \rfloor$. Because n and k are integers, the value of f is an element of the closed interval $\left[0, \frac{k-1}{k}\right]$, i.e., it never reaches 1.

Combining (56) and (57) allows us to derive the following formula for the value of $\frac{n/k}{j}$:

$$\frac{n/k}{j} = \frac{\lfloor n/k \rfloor}{j} + \frac{f}{j} = \frac{xj + y}{j} + \frac{f}{j} = x + \frac{y}{j} + \frac{f}{j}. \tag{58}$$

This formula, in turn, implies that the following upper and lower bounds hold for $\frac{n/k}{j}$:

$$x = \left\lfloor \frac{\lfloor n/k \rfloor}{j} \right\rfloor \leq \frac{\lfloor n/k \rfloor}{j} \leq \frac{n/k}{j} < x + \frac{j-1}{j} + \frac{1}{j} = x + 1. \tag{59}$$

In other words, $x \leq \frac{n/k}{j} < x + 1$. Applying the floor function to all three terms leads to $x \leq \left\lfloor \frac{n/k}{j} \right\rfloor < x + 1$, since x is an integer. Because the term in the middle must be an integer and because the second inequality remains strict, it follows that $\left\lfloor \frac{n/k}{j} \right\rfloor$ must be equal to $x = \left\lfloor \frac{\lfloor n/k \rfloor}{j} \right\rfloor$. This completes the proof. □

Lemma 9. *Let k, j, and n be three positive integers such that $n \geq kj$. Then, the lengths of the following two Farey sequences are equal:*

$$\left| F_{\lfloor \lfloor n/k \rfloor / j \rfloor} \right| = \left| F_{\lfloor n/(kj) \rfloor} \right|. \tag{60}$$

Proof. The proof follows from Lemma 8, which proved that the values of the two subindices in (60) are equal. Also, when $n \geq kj$, the value of $\lfloor n/(kj) \rfloor$ is an integer that is greater than or equal to 1, i.e., it identifies an order of a Farey sequence. □

The next two lemmas are used to prove the computational complexities of FAREYLENGTHL and FAREYLENGTHC (i.e., Algorithm S10 and Algorithm 5), respectively.

Lemma 10. *Let N be a positive integer. Then,*

$$\sum_{k=1}^{N} O\left(\frac{1}{k}\right) = O(\log N). \tag{61}$$

Proof. Let $f(x) = \frac{1}{x}$. This function is monotonically decreasing in the interval $(0, \infty)$. Therefore,

$$\int_{k}^{k+1} \frac{1}{x}\, \mathrm{d}x < \frac{1}{k} < \int_{k-1}^{k} \frac{1}{x}\, \mathrm{d}x, \tag{62}$$

for each $k \in \mathbb{N} = \{1, 2, \dots\}$. This implies that we can find lower and upper bounds for the sum in formula (61) as follows:

$$\int_{1}^{N+1} \frac{1}{x}\, \mathrm{d}x = \sum_{k=1}^{N} \int_{k}^{k+1} \frac{1}{x}\, \mathrm{d}x < \sum_{k=1}^{N} \frac{1}{k} = 1 + \sum_{k=2}^{N} \frac{1}{k} \leq 1 + \sum_{k=2}^{N} \int_{k-1}^{k} \frac{1}{x}\, \mathrm{d}x = 1 + \int_{1}^{N} \frac{1}{x}\, \mathrm{d}x. \tag{63}$$

The left-most integral is equal to:

$$\int_{1}^{N+1} \frac{1}{x}\, \mathrm{d}x = \ln x \Big|_{1}^{N+1} = \ln(N+1). \tag{64}$$

The right-most term in this inequality evaluates to:

$$1 + \int_{1}^{N} \frac{1}{x}\, \mathrm{d}x = 1 + \ln x \Big|_{1}^{N} = 1 + \ln N. \tag{65}$$

Therefore, the bounds for the sum are:

$$\ln(N+1) < \sum_{k=1}^{N} \frac{1}{k} \leq 1 + \ln N, \tag{66}$$

which completes the proof. □

Lemma 11. *Let N be a positive integer. Then,*

$$\sum_{k=1}^{N} O\left(\frac{1}{\sqrt{k}}\right) = O\left(\sqrt{N}\right). \tag{67}$$

Proof. Let $f(x) = \frac{1}{\sqrt{x}}$. This function is monotonically decreasing in the interval $(0, \infty)$. Thus, for each $k \in \mathbb{N} = \{1, 2, \dots\}$ the following inequality holds:

$$\int_{k}^{k+1} \frac{1}{\sqrt{x}}\, \mathrm{d}x < \frac{1}{\sqrt{k}} < \int_{k-1}^{k} \frac{1}{\sqrt{x}}\, \mathrm{d}x. \tag{68}$$

From this inequality we can derive lower and upper bounds for the value of the sum $1/\sqrt{1} + 1/\sqrt{2} + \cdots + 1/\sqrt{N}$, i.e.,

$$\int_{1}^{N+1} \frac{1}{\sqrt{x}}\, \mathrm{d}x = \sum_{k=1}^{N} \int_{k}^{k+1} \frac{1}{\sqrt{x}}\, \mathrm{d}x < \sum_{k=1}^{N} \frac{1}{\sqrt{k}} < \sum_{k=1}^{N} \int_{k-1}^{k} \frac{1}{\sqrt{x}}\, \mathrm{d}x = \int_{0}^{N} \frac{1}{\sqrt{x}}\, \mathrm{d}x. \tag{69}$$

The left-most integral evaluates to:

$$\int_{1}^{N+1} \frac{1}{\sqrt{x}}\, \mathrm{d}x = 2\sqrt{x} \Big|_{1}^{N+1} = 2\sqrt{N+1} - 2. \tag{70}$$

Similarly, the right-most integral has the following value:

$$\int_{0}^{N} \frac{1}{\sqrt{x}}\, \mathrm{d}x = 2\sqrt{x} \Big|_{0}^{N} = 2\sqrt{N}. \tag{71}$$

Combining (69) with (70) and (71) leads to:

$$2\sqrt{N+1} - 2 < \sum_{k=1}^{N} \frac{1}{\sqrt{k}} < 2\sqrt{N}, \tag{72}$$

from which (67) follows. □

S5. Proof of Formula (2)

This section proves formula (2), which expresses the length of the Farey sequence F_n recursively in terms of the lengths of the Farey sequences of lower orders. This formula is well-known, but its proof is hard to find. Our proof uses only basic algebra and mathematical induction. The proof also uses Euler's totient function $\varphi(n)$, which is defined in Section S3.

Theorem 12. *The length of the Farey sequence of order n can be computed recursively as follows:*

$$|F_n| = \frac{(n+3)n}{2} - \sum_{k=2}^{n} \left|F_{\lfloor n/k \rfloor}\right| , \tag{73}$$

where $\lfloor x \rfloor$ denotes the largest integer that does not exceed x.

Proof. The proof is by mathematical induction. The base case of the induction is formed by the length of F_n when $n = 1$, i.e.,

$$|F_1| = \frac{(1+3)\cdot 1}{2} = 2. \tag{74}$$

To prove the inductive step, we will introduce the term T_n that is defined as follows:

$$T_n = |F_n| + \sum_{k=2}^{n} \left|F_{\lfloor n/k \rfloor}\right| . \tag{75}$$

Our goal is to prove that

$$T_n = \frac{(n+3)n}{2}. \tag{76}$$

We have already established that this is true for $n = 1$. Assuming that this formula holds for some $n \geq 1$, our next goal is to prove that it also holds for T_{n+1}. In other words, we would like to show that the value of T_{n+1} can be expressed as follows:

$$\begin{aligned} T_{n+1} &= |F_{n+1}| + \sum_{k=2}^{n+1} \left|F_{\lfloor (n+1)/k \rfloor}\right| \\ &= \frac{((n+1)+3)(n+1)}{2} \\ &= \frac{(n+3)n}{2} + \frac{2n+4}{2} \\ &= T_n + n + 2. \end{aligned} \tag{77}$$

Using Theorem 1 and Definition 1 (see Section S3), we can express the term T_{n+1} as shown below:

$$\begin{aligned} T_{n+1} &= |F_{n+1}| \ + \sum_{k=2}^{n+1} \left|F_{\lfloor \frac{n+1}{k} \rfloor}\right| \\ &= |F_n| + \varphi(n+1) \ + \sum_{k=2}^{n+1} \left|F_{\lfloor \frac{n+1}{k} \rfloor}\right| \\ &= |F_n| + \varphi(n+1) + \sum_{k=2}^{n} \left|F_{\lfloor \frac{n+1}{k} \rfloor}\right| + \underbrace{\sum_{k=n+1}^{n+1} \left|F_{\lfloor \frac{n+1}{k} \rfloor}\right|}_{|F_1|} \\ &= |F_n| + \varphi(n+1) + \sum_{k=2}^{n} \left|F_{\lfloor \frac{n+1}{k} \rfloor}\right| + 2. \end{aligned} \tag{78}$$

Lemmas 6 and 7 (see Section S4) imply that in (78) the value of each term in the sum from 2 to n can be expressed as follows:

$$\left|F_{\lfloor \frac{n+1}{k} \rfloor}\right| = \begin{cases} \left|F_{\lfloor \frac{n}{k} \rfloor}\right|, & \text{if } (n{+}1) \bmod k \neq 0, \\ \left|F_{\lfloor \frac{n}{k} \rfloor + 1}\right|, & \text{if } (n{+}1) \bmod k = 0. \end{cases} \tag{79}$$

Moreover, for the second case in (79), Theorem 1 and Lemma 7 imply that:

$$\left|F_{\lfloor \frac{n+1}{k} \rfloor}\right| = \left|F_{\lfloor \frac{n}{k} \rfloor + 1}\right| = \left|F_{\lfloor \frac{n}{k} \rfloor}\right| + \varphi\left(\frac{n+1}{k}\right), \qquad \text{if } (n+1) \bmod k = 0. \tag{80}$$

Let D be the set of all positive integer divisors of $n{+}1$ that lie between 2 and n. That is,

$$D = \big\{k \in \{2,3,\ldots,n\} \text{ s.t. } (n+1) \bmod k = 0\big\}. \tag{81}$$

Then, the sum in equation (78) can be expressed as follows:

$$\sum_{k=2}^{n} \left| F_{\left\lfloor \frac{n+1}{k} \right\rfloor} \right| = \sum_{k=2}^{n} \left| F_{\left\lfloor \frac{n}{k} \right\rfloor} \right| + \sum_{k \in D} \varphi\left(\frac{n{+}1}{k}\right). \tag{82}$$

If an integer k is an element of the set D, then the value of ${}^{(n+1)}\!/_{k}$ is also an element of D and vice versa. Therefore,

$$\sum_{k \in D} \varphi\left(\frac{n{+}1}{k}\right) = \sum_{k \in D} \varphi(k). \tag{83}$$

Thus, equation (82) can be restated as:

$$\sum_{k=2}^{n} \left| F_{\left\lfloor \frac{n+1}{k} \right\rfloor} \right| = \sum_{k=2}^{n} \left| F_{\left\lfloor \frac{n}{k} \right\rfloor} \right| + \sum_{k \in D} \varphi(k). \tag{84}$$

Plugging this result into (78) leads to the following formula for the value of T_{n+1}:

$$T_{n+1} = |F_n| + \varphi(n{+}1) + \sum_{k=2}^{n} \left| F_{\left\lfloor \frac{n}{k} \right\rfloor} \right| + \sum_{k \in D} \varphi(k) + 2. \tag{85}$$

Because $\varphi(1) = 1$, it follows that $\varphi(1) + 1 = 2$. This allows us to rearrange the terms in the last equation as follows:

$$\begin{aligned} T_{n+1} &= \underbrace{|F_n| + \sum_{k=2}^{n} \left| F_{\left\lfloor \frac{n}{k} \right\rfloor} \right|}_{T_n} + \underbrace{\varphi(n{+}1) + \sum_{k \in D} \varphi(k) + \varphi(1)}_{\sum\limits_{k|n+1} \varphi(k)} + 1 \\ &= T_n + \sum_{k|n+1} \varphi(k) \; + \; 1. \end{aligned} \tag{86}$$

In this formula, $\sum\limits_{k|n+1} \varphi(k)$ denotes the sum of the values of Euler's totient function $\varphi(k)$ for all k that are positive integer divisors of $n+1$. In 1798, Gauss proved[14] that this sum is equal to $n{+}1$. Thus,

$$T_{n+1} = T_n + (n+1) + 1 = T_n + n + 2, \tag{87}$$

which proves equation (77) as required. □

S6. Proof of Formula (5)

This section proves another recursive formula for the value of $|F_n|$. This formula is derived from formula (2) by splitting the sum into two segments and grouping the repeated terms in the second segment. The resulting formula is longer, but it needs to add fewer terms. It leads to Algorithm 5 (i.e., FAREYLENGTHC), which is described in the main paper.

Theorem 13. *For each $n > 1$, the length of the Farey sequence F_n can be expressed using the following recursive formula:*

$$|F_n| = \frac{(n+3)n}{2} - \sum_{k=2}^{u(n)} \left|F_{\lfloor n/k \rfloor}\right| - \sum_{k=1}^{\lfloor \sqrt{n} \rfloor} \left(\left\lfloor \frac{n}{k} \right\rfloor - \left\lfloor \frac{n}{k+1} \right\rfloor \right) \cdot |F_k|, \tag{88}$$

where $u(n) = \left\lfloor \frac{n}{\lfloor \sqrt{n} \rfloor + 1} \right\rfloor$.

Proof. This formula is derived from formula (2), which is replicated below:

$$|F_n| = \frac{(n+3)n}{2} - \sum_{k=2}^{n} \left|F_{\lfloor n/k \rfloor}\right|. \tag{89}$$

The first step is to split the summation over k at $u(n)$ into two separate sums:

$$\sum_{k=2}^{n} \left|F_{\lfloor n/k \rfloor}\right| = \sum_{k=2}^{u(n)} \left|F_{\lfloor n/k \rfloor}\right| + \sum_{k=u(n)+1}^{n} \left|F_{\lfloor n/k \rfloor}\right|. \tag{90}$$

And then change the index variable from k to m in the second sum:

$$\sum_{k=2}^{n} \left|F_{\lfloor n/k \rfloor}\right| = \sum_{k=2}^{u(n)} \left|F_{\lfloor n/k \rfloor}\right| + \sum_{m=u(n)+1}^{n} \left|F_{\lfloor n/m \rfloor}\right|. \tag{91}$$

The next step is to express the last sum in formula (91) so that it has only $\lfloor \sqrt{n} \rfloor$ terms. Lemma 4 proves that the set of the indices m used in that sum is equal to the following union:

$$\bigcup_{k=1}^{\lfloor \sqrt{n} \rfloor} S(k,n) = \{u(n)+1,\, u(n)+2, \ldots,\, n\}, \tag{92}$$

where each set $S(k,n)$ consists of all positive integers m for which $\lfloor n/m \rfloor = k$, i.e.,

$$S(k,n) = \left\{ \left\lfloor \frac{n}{k+1} \right\rfloor + 1, \left\lfloor \frac{n}{k+1} \right\rfloor + 2, \ldots, \left\lfloor \frac{n}{k} \right\rfloor \right\}. \tag{93}$$

Lemma 4 also proves that all sets $S(k,n)$ in this union are disjoint. Each of these sets has $\left\lfloor \frac{n}{k} \right\rfloor - \left\lfloor \frac{n}{k+1} \right\rfloor$ elements. Therefore, for each $n > 1$ the last sum in (91) can be expressed as follows:

$$\sum_{m=u(n)+1}^{n} \left|F_{\lfloor n/m \rfloor}\right| = \sum_{k=1}^{\lfloor \sqrt{n} \rfloor} \sum_{m \in S(k,n)} |F_k| = \sum_{k=1}^{\lfloor \sqrt{n} \rfloor} \left(\left\lfloor \frac{n}{k} \right\rfloor - \left\lfloor \frac{n}{k+1} \right\rfloor \right) |F_k|. \tag{94}$$

Substituting this expression into (91) and that result into (89) proves the theorem. □

Formula (88) cannot express F_1 because the last sum leads to an infinite recursion when $n = 1$. Also, in this special case the sum in formula (89) is degenerate. In contrast to the cases when $n \geq 2$, the sum cannot be split in a way that leads to formula (90) when $n = 1$. Therefore, any algorithm that implements formula (88) should handle $n = 1$ as a special case. For example, F_1 can be hard-coded to be equal to 2.

S7. Computing $\lfloor\sqrt{n}\rfloor$ and $\lfloor\sqrt[3]{n^2}\rfloor$ exactly

Algorithms S4 and S5 give the pseudo-code for two helper functions that are used by several of the algorithms. The first function computes $\lfloor\sqrt{n}\rfloor$ exactly using Newton's method and integer arithmetic.[28] The second function computes the value of $\lfloor\sqrt[3]{n}\rfloor$ exactly using a similar approach. All algorithms, however, call ICBRT with an argument that is equal to n^2 instead of n. Thus, it is used to compute $\lfloor\sqrt[3]{n^2}\rfloor$. Both algorithms run in $O(\log\log n)$ time and use $O(1)$ memory.

Algorithm S4. Compute $\lfloor\sqrt{n}\rfloor$ exactly using integer arithmetic. Runs in $O(\log\log n)$ time and uses $O(1)$ memory.

```
1: function ISQRT($n$)
2:     x ← $n$;
3:     y ← $\lceil n/2 \rceil$;
4:     while y < x do
5:         x ← y;
6:         y ← $\left\lfloor \frac{x+\lfloor n/x \rfloor}{2} \right\rfloor$;
7:     end while
8:     return x;
9: end function
```

Algorithm S5. Compute $\lfloor\sqrt[3]{n}\rfloor$ exactly using integer arithmetic. Runs in $O(\log\log n)$ time and uses $O(1)$ memory.

```
1: function ICBRT($n$)
2:     x ← $n$;
3:     y ← $\lceil n/3 \rceil$;
4:     while y < x do
5:         x ← y;
6:         y ← $\left\lfloor \frac{2x+\lfloor n/x^2 \rfloor}{3} \right\rfloor$;
7:     end while
8:     return x;
9: end function
```

S8. Lookup Table Design

Some of the algorithms described in this paper use a lookup table called F to store the values of $|F_m|$ for different m. The computational complexity estimates assume that each entry in this lookup table can be accessed in $O(1)$ time. They also assume that storing a lookup table of size N requires $O(N)$ memory.

This appendix describes an implementation for a lookup table that satisfies these time and space complexity constraints. The implementation is specific to our algorithms, i.e., it works for the special case when all keys (or indices) that are used are in the set $\{1, 2, \ldots, d\} \cup \{\lfloor \frac{n}{\lfloor n/d \rfloor} \rfloor, \ldots, \lfloor \frac{n}{2} \rfloor, \lfloor \frac{n}{1} \rfloor\}$. Alternatively, it is possible to use a generic hash table as a lookup table (e.g., the hash table implemented in the standard dictionary class in the Python language). In that case the computational complexity of setting or getting entries is in $O(1)$ on average. Even though this is a minor technical point, this appendix shows that the desired performance can be achieved in all cases, i.e., not just on average.

Algorithm S6 gives the pseudo-code for initializing the lookup table. The function has two parameters: n and d. The value of n determines the size of the problem, i.e., the order of the Farey sequence F_n for which we want to compute the length. The parameter d controls the split between the two sets of keys that the table supports. The table is organized as two arrays. The first array, X, stores the entries for the keys in the set $\{1, 2, \ldots, d\}$. The second array, Y, stores the entries for the keys in the set $\{\lfloor \frac{n}{\lfloor n/d \rfloor} \rfloor, \ldots, \lfloor \frac{n}{2} \rfloor, \lfloor \frac{n}{1} \rfloor\}$. In most cases, $d = \lfloor\sqrt{n}\rfloor + 1$, except for Algorithm S13, where it is set to $\lfloor\sqrt[3]{n^2}\rfloor + 1$.

Algorithm S7 gives the pseudo-code for getting the value of an element from the lookup table, given a key k. This function is called when the square bracket notation is used with F in our algorithms, i.e., $F[k]$ translates to GetItem(F, k);

Algorithm S8 gives the pseudo-code for setting an element of the lookup table. Once again, it is assumed that an assignment $F[k] \leftarrow x$ in our algorithms translates to a call to SetItem(F, k, x).

Finally, Algorithm S9 gives the pseudo-code for a function that checks if the value for the key k is set in the lookup table F. This check is used in some of our algorithms for diagnostic purposes. That is, it is assumed that a check k **in** F translates to a function call Contains(F, k).

Algorithm S6. Initialize a lookup table. Uses $O(d + n/d)$ memory.

```
function LookupTable($n, d$)
    $F \leftarrow$ NewObject();
    $F.n \leftarrow n$;
    $F.X \leftarrow$ NullArray($d$);
    $F.Y \leftarrow$ NullArray( $\lfloor n/d \rfloor + 1$);
    return $F$;
end function
```

Algorithm S7. Get the value for a key from the lookup table. Runs in $O(1)$ time.

```
function GetItem($F, k$)
    if $k <$ Length($F.X$) then
        return $F.X[k]$;
    else
        return $F.Y[\,\lfloor F.n/k \rfloor\,]$;
    end if
end function
```

Algorithm S8. Set the value for a key in the lookup table. Runs in $O(1)$ time.

```
function SetItem($F, k, x$)
    if $k <$ Length($F.X$) then
        $F.X[k] \leftarrow x$;
    else
        $F.Y[\,\lfloor F.n/k \rfloor\,] \leftarrow x$;
    end if
end function
```

Algorithm S9. Check if the value for a key is set in the lookup table. Runs in $O(1)$ time.

```
function Contains($F, k$)
    if $k <$ Length($F.X$) then
        return ($F.X[k]$ is not null);
    else
        $m \leftarrow \lfloor F.n/k \rfloor$;
        if $k \neq \lfloor F.n/m \rfloor$ then
            return False;
        end if
        return ($F.Y[m]$ is not null);
    end if
end function
```

S9. Two algorithms that run in $O(n \log n)$ time and use $O(\sqrt{n})$ memory

This section describes two $O(n \log n)$ algorithms for computing the length of the Farey sequence of order n. The algorithms are based on a combination of formulas (2) and (5) from the main paper, which are reproduced below:

$$|F_n| = \frac{(n+3)\,n}{2} - \sum_{k=2}^{n} \left|F_{\lfloor n/k \rfloor}\right|, \tag{95}$$

$$|F_n| = \frac{(n+3)\,n}{2} - \sum_{k=2}^{u(n)} \left|F_{\lfloor n/k \rfloor}\right| - \sum_{k=1}^{\lfloor \sqrt{n} \rfloor} \left(\left\lfloor \frac{n}{k} \right\rfloor - \left\lfloor \frac{n}{k+1} \right\rfloor \right) \cdot |F_k|. \tag{96}$$

Algorithm S10 gives the pseudo-code for the first algorithm, which is given a suffix letter L. The helper function in this case implements formula (95). The main algorithm, however, structures the calls to this function in the same way as Algorithm 5, which is based on formula (96). Theorem 14 (see below) proves that the computational complexity of this algorithm is in $O(n \log n)$. The space complexity is in $O(\sqrt{n})$. This is easy to show because FareyLengthL is structured similarly to FareyLengthC, except for using a different helper function (see also the proofs in Section S10).

Algorithm S10. Compute the length of the Farey sequence F_n. Runs in $O(n \log n)$ time and uses $O(\sqrt{n})$ memory.

```
function FareyLengthL($n$)
    $r \leftarrow$ ISQRT($n$);
    $u \leftarrow \lfloor \frac{n}{r+1} \rfloor$;
    $F \leftarrow$ LookupTable($n$, $r+1$);
    $F[1] \leftarrow 2$;
    for $m \leftarrow 2$ to $r$ do
        UpdateLookupTableL($F, m$);
    end for
    for $j \leftarrow u$ down to $1$ do
        UpdateLookupTableL($F, \lfloor n/j \rfloor$);
    end for
    return $F[n]$;
end function

function UpdateLookupTableL($F, m$)        // helper function
    $s \leftarrow 0$;
    for $k \leftarrow 2$ to $m$ do
        $q \leftarrow \lfloor m/k \rfloor$;
        Assert($q$ in $F$);
        $s \leftarrow s + F[q]$;
    end for
    $F[m] \leftarrow \frac{(m+3)\,m}{2} - s$;
end function
```

Algorithm S11 is a recursive version of Algorithm S10. This algorithm also runs in $O(n \log n)$ time and uses $O(\sqrt{n})$ memory. This can be established by unpacking the recursion, which leads to the iterative version presented above. This is the shortest algorithm described in this paper. Unfortunately, it is also one of the slowest. For small values of n, however, it could be very useful. For example, it can be used to verify the output of the faster algorithms.

Algorithm S11. Recursive algorithm for computing the length of F_n. Runs in $O(n \log n)$ time and uses $O(\sqrt{n})$ memory.

```
function FAREYLENGTHLR($n$, $F$ = null)
    if $F$ is null then
        $F \leftarrow$ LOOKUPTABLE($n$, ISQRT($n$) + 1);        // initialize the lookup table $F$ for memoization
    end if
    if $n$ in $F$ then
        return $F[n]$;
    end if
    $s \leftarrow 0$;
    for $k \leftarrow 2$ to $n$ do
        $s \leftarrow s +$ FAREYLENGTHLR($\lfloor n/k \rfloor, F$);
    end for
    $F[n] \leftarrow \frac{(n+3)\,n}{2} - s$;        // update the lookup table
    return $F[n]$;
end function
```

Theorem 14. *Algorithm S10 (i.e.,* FAREYLENGTHL*) runs in* $O(n \log n)$ *time.*

Proof. One call to the helper function UPDATELOOKUPTABLEL computes exactly one new entry in the lookup table F. The main algorithm calls this function in the correct order such that the elements of F that the helper function uses are already computed by the time it is called. Thus, the computational complexity of the helper function is in $O(m)$, because it has only one loop that runs for $m - 1$ iterations in order to compute $F[m]$.

The main algorithm calls the helper function from within two for-loops, but they are not nested. Therefore, the computational complexity of the algorithm can be expressed as follows:

$$\sum_{m=2}^{\lfloor\sqrt{n}\rfloor} O(m) + \sum_{j=1}^{u(n)} O(\lfloor n/j \rfloor) = O\left(\sum_{m=2}^{\lfloor\sqrt{n}\rfloor} m\right) + O\left(\sum_{j=1}^{u(n)} \lfloor n/j \rfloor\right). \tag{97}$$

The first sum in the right-hand side of this formula is in $O(n)$. That is,

$$\sum_{m=2}^{\lfloor\sqrt{n}\rfloor} m = \frac{(\lfloor\sqrt{n}\rfloor - 1)(\lfloor\sqrt{n}\rfloor + 2)}{2} = \frac{n}{2} + \frac{\lfloor\sqrt{n}\rfloor}{2} - 1 = O(n). \tag{98}$$

The second sum in the right-hand side of (97) is in $O(n \log n)$. This can be shown as follows:

$$\sum_{j=1}^{u(n)} \lfloor n/j \rfloor \leq \sum_{j=1}^{\lfloor\sqrt{n}\rfloor} \lfloor n/j \rfloor \leq \sum_{j=1}^{\lfloor\sqrt{n}\rfloor} \frac{n}{j} = n\left(\sum_{j=1}^{\lfloor\sqrt{n}\rfloor} \frac{1}{j}\right). \tag{99}$$

In this formula, the left-most inequality follows from Lemma 5, which implies that $u(n) \leq \lfloor\sqrt{n}\rfloor$. The derivation can be continued using Lemma 10, which implies that:

$$\sum_{j=1}^{\lfloor\sqrt{n}\rfloor} \frac{1}{j} = O\big(\log\lfloor\sqrt{n}\rfloor\big) = O(\log n). \tag{100}$$

Therefore,

$$\sum_{j=1}^{u(n)} \lfloor n/j \rfloor = O(n \log n). \tag{101}$$

Combining formulas (98) and (101) with formula (97) proves that Algorithm S10 runs in $O(n \log n)$ time. □

S10. Computational Complexity Analysis of FareyLengthC

This section analyzes the computational complexity of FareyLengthC (i.e., Algorithm 5) and proves that it runs in $O(n^{3/4})$ time and uses $O(\sqrt{n})$ memory. It also proves that the elements of the lookup table are computed in such a way that they are always available when the helper function needs them.

The following theorem gives the time complexity of the helper function UpdateLookupTable (i.e., Algorithm 6).

Theorem 15. *Algorithm 6 runs in $O(\sqrt{m})$ time.*

Proof. This algorithm implements a helper function that uses formula (5) to compute the value of F_m. That is,

$$|F_m| = \frac{(m+3)m}{2} - \sum_{k=2}^{u(m)} \left|F_{\lfloor m/k \rfloor}\right| - \sum_{k=1}^{\lfloor\sqrt{m}\rfloor} \left(\left\lfloor \frac{m}{k} \right\rfloor - \left\lfloor \frac{m}{k+1} \right\rfloor \right) \cdot |F_k|. \tag{102}$$

This function has two for-loops, but they are not nested. These loops implement the first and the second summation in the formula, respectively. One call to this function computes exactly one new entry in the lookup table F. The main algorithm (i.e., Algorithm 5) ensures that all calls to this function are performed in the correct order such that the values from F that it uses are already set because they were computed earlier (see the proofs on the next page).

The computational complexity of each iteration in either of the two loops is in $O(1)$. Thus, the computational complexity of the helper function can be determined as follows:

$$O\left(\sum_{k=2}^{u(m)} 1 + \sum_{k=1}^{\lfloor\sqrt{m}\rfloor} 1 \right) = O\big(u(m) - 1 + \lfloor\sqrt{m}\rfloor \big) = O\big(\sqrt{m}\big) . \tag{103}$$

This result follows from Lemma 5, which proves that $u(m) \leq \lfloor\sqrt{m}\rfloor$. This, in turn, implies that $u(m) - 1 + \lfloor\sqrt{m}\rfloor < 2\lfloor\sqrt{m}\rfloor$. Therefore, this function runs in $O(\sqrt{m})$ time, where m is its argument, i.e., not the n for which the main algorithm is called. □

The next theorem establishes the time and space complexity of FareyLengthC (i.e., Algorithm 5).

Theorem 16. *Algorithm 5 runs in $O(n^{3/4})$ time and uses $O(\sqrt{n})$ memory.*

Proof. The algorithm calls the helper function from inside two for-loops, which are not nested. The first loop iterates over m from 2 to $\lfloor\sqrt{n}\rfloor$. The second loop iterates over j from $u(n)$ down to 1. Thus, the overall computational complexity can be expressed as follows:

$$\sum_{m=2}^{\lfloor\sqrt{n}\rfloor} O\big(\sqrt{m}\big) + \sum_{j=1}^{u(n)} O\left(\sqrt{\left\lfloor \frac{n}{j} \right\rfloor} \right) = O\left(\sum_{m=2}^{\lfloor\sqrt{n}\rfloor} \sqrt{m} \right) + O\left(\sum_{j=1}^{u(n)} \sqrt{\left\lfloor \frac{n}{j} \right\rfloor} \right) . \tag{104}$$

The first sum in the right-hand side of (104) can be bounded from above as follows:

$$\sum_{m=2}^{\lfloor\sqrt{n}\rfloor} \sqrt{m} \leq \sum_{m=2}^{\lfloor\sqrt{n}\rfloor} \sqrt{\lfloor\sqrt{n}\rfloor} \leq \sum_{m=2}^{\lfloor\sqrt{n}\rfloor} \sqrt{\sqrt{n}} < \lfloor\sqrt{n}\rfloor \sqrt{\sqrt{n}} \leq \sqrt{n}\sqrt{\sqrt{n}} = n^{1/2}\, n^{1/4} = n^{3/4}. \tag{105}$$

This result implies that

$$O\left(\sum_{m=2}^{\lfloor\sqrt{n}\rfloor} \sqrt{m} \right) = O(n^{3/4}). \tag{106}$$

The second sum in the right-hand side of (104) can also be bounded from above:

$$\sum_{j=1}^{u(n)} \sqrt{\left\lfloor \frac{n}{j} \right\rfloor} \leq \sum_{j=1}^{\lfloor\sqrt{n}\rfloor} \sqrt{\left\lfloor \frac{n}{j} \right\rfloor} \leq \sum_{j=1}^{\lfloor\sqrt{n}\rfloor} \sqrt{\frac{n}{j}} = \sqrt{n} \sum_{j=1}^{\lfloor\sqrt{n}\rfloor} \frac{1}{\sqrt{j}} = \sqrt{n}\, O\big(\sqrt{\lfloor\sqrt{n}\rfloor}\big) = O(n^{3/4}), \tag{107}$$

which follows from Lemmas 5 and 11.

To summarize, we showed that both terms in the right-hand side of (104) are in $O(n^{3/4})$. Therefore, the computational complexity of FareyLengthC is also in $O(n^{3/4})$. The algorithm fills $\lfloor\sqrt{n}\rfloor$ entries of the lookup table F in the first for-loop and $u(n)$ entries in the second for-loop. Because $u(n) \leq \lfloor\sqrt{n}\rfloor$, the space complexity of the algorithm is in $O(\sqrt{n})$. □

Algorithm 5 uses the helper function to compute the value of $|F_m|$ for each integer m in two lists. The first list has $\lfloor\sqrt{n}\rfloor - 1$ elements and consists of the numbers $2, 3, \ldots, \lfloor\sqrt{n}\rfloor$. The second list has $u(n)$ elements and consists of the numbers $\left\lfloor\frac{n}{u(n)}\right\rfloor$, $\left\lfloor\frac{n}{u(n)-1}\right\rfloor$, ..., $\left\lfloor\frac{n}{1}\right\rfloor$. The algorithm processes the first list sequentially starting from 2. After that, it processes the second list sequentially starting from $\lfloor n/u(n)\rfloor$. In total, the algorithm processes $\lfloor\sqrt{n}\rfloor - 1 + u(n) \leq 2\lfloor\sqrt{n}\rfloor$ values of m.

Let $Q(m)$ be the set of keys in the lookup table F that are used by the first for-loop of the helper function (i.e., Algorithm 6). For a given value of m this set is equal to:

$$Q(m) = \left\{\left\lfloor\frac{m}{2}\right\rfloor, \left\lfloor\frac{m}{3}\right\rfloor, \ldots, \left\lfloor\frac{m}{u(m)}\right\rfloor\right\}. \tag{108}$$

Let $K(m)$ be the set of keys in the lookup table F that are used by the second for-loop of the helper function. That is,

$$K(m) = \{1, 2, \ldots, \lfloor\sqrt{m}\rfloor\}. \tag{109}$$

The following theorem helps us to prove that all entries in the lookup table F that the helper function uses to compute the value of $F[m]$ are already set when the function is called in the first for-loop of Algorithm 5.

Theorem 17. *Let m be an integer that is greater than 1. Then, both $Q(m)$ and $K(m)$ are subsets of the set of all integers in the interval $[1, m-1]$. More formally,*

$$Q(m) \subseteq \{1, 2, \ldots, m-1\}, \tag{110}$$

$$K(m) \subseteq \{1, 2, \ldots, m-1\}. \tag{111}$$

Proof. The elements of the set $Q(m)$ fall between 1 and $\lfloor m/2\rfloor$, which does not exceed $m-1$. The reason for this is that $u(m) \geq 2$ when $m \geq 2$, i.e., the maximum element of $Q(m)$ is $\lfloor m/2\rfloor$. The minimum element is equal to $\lfloor m/u(m)\rfloor$, which is a positive integer. Thus, $Q(m) \subseteq \{1, 2, \ldots, m-1\}$.

The elements of the set $K(m)$ fall between 1 and $\lfloor\sqrt{m}\rfloor$. Also, the value of $\lfloor\sqrt{m}\rfloor$ does not exceed $m-1$ for $m \geq 2$. Therefore, $K(m) \subseteq \{1, 2, \ldots, m-1\}$, as required. □

Corollary 18. *All entries in the lookup table F that are required for computing $F[m]$ during each iteration of the first for-loop in* FAREYLENGTHC *have already been computed.*

Proof. This follows from Theorem 17. The algorithm sets $F[1]$ to 2 and then loops over the values of m in the list $(2, 3, \ldots, \lfloor\sqrt{n}\rfloor)$ in increasing order. Thus, when the algorithm reaches the iteration that computes $F[m]$, the lookup table already contains the entries for the keys in the set $\{1, 2, \ldots, m-1\}$. Therefore, Theorem 17 implies that all prerequisite elements for computing $F[m]$ are already computed when the helper function is called to calculate it. □

Let $Y(j)$ be the set of keys in the lookup table F that are already computed by Algorithm 5 before it calls the helper function in the second for-loop with the parameter m equal to $\lfloor n/j\rfloor$. More formally,

$$Y(j) = \{1, 2, \ldots, \lfloor\sqrt{n}\rfloor\} \cup \left\{\left\lfloor\frac{n}{u(n)}\right\rfloor, \left\lfloor\frac{n}{u(n)-1}\right\rfloor, \ldots, \left\lfloor\frac{n}{j+1}\right\rfloor\right\}. \tag{112}$$

Then, the following theorem helps us to prove that the second for-loop in Algorithm 5 also uses only those entries of F that have already been set.

Theorem 19. *Let n be a positive integer and let j be an integer between 1 and $u(n)$. Then,*

$$K(\lfloor n/j\rfloor) \subseteq Y(j), \tag{113}$$

$$Q(\lfloor n/j\rfloor) \subseteq Y(j). \tag{114}$$

Proof. Clearly, formula (113) holds because

$$K(\lfloor n/j\rfloor) = \left\{1, 2, \ldots, \left\lfloor\sqrt{n/j}\right\rfloor\right\} \subseteq \{1, 2, \ldots, \lfloor\sqrt{n}\rfloor\} \subseteq Y(j). \tag{115}$$

To prove formula (114), suppose that $k \in Q(\lfloor n/j\rfloor)$. Then, there is an integer q such that $q \in \{2, 3, \ldots, u(\lfloor n/j\rfloor)\}$ and $k = \left\lfloor\frac{\lfloor n/j\rfloor}{q}\right\rfloor$. Lemma 8 implies that $k = \left\lfloor\frac{n}{jq}\right\rfloor$. If $k \leq \lfloor\sqrt{n}\rfloor$, then $k \in Y(j)$ because, by definition, $Y(j)$ includes all integers between 1 and $\lfloor\sqrt{n}\rfloor$. If $k > \lfloor\sqrt{n}\rfloor$, then $k \geq \lfloor\sqrt{n}\rfloor + 1$, which implies that $\frac{n}{jq} \geq \lfloor\sqrt{n}\rfloor + 1$. Therefore, $jq \leq \frac{n}{\lfloor\sqrt{n}\rfloor+1}$. Because the value of the product jq is an integer, $jq \leq \left\lfloor\frac{n}{\lfloor\sqrt{n}\rfloor+1}\right\rfloor = u(n)$. Also, from $q \geq 2$ it follows that $jq > j$. Therefore, $jq \in \{j+1, j+2, \ldots, u(n)\}$, and, thus,

$$\left\lfloor\frac{n}{jq}\right\rfloor \in \left\{\left\lfloor\frac{n}{u(n)}\right\rfloor, \left\lfloor\frac{n}{u(n)-1}\right\rfloor, \ldots, \left\lfloor\frac{n}{j+1}\right\rfloor\right\}. \tag{116}$$

This implies that $k \in Y(j)$, which shows that $Q(\lfloor n/j\rfloor) \subseteq Y(j)$ and completes the proof of the theorem. □

Corollary 20. *For each iteration of the second for-loop in Algorithm 5, all entries in the lookup table F that are necessary for computing the value of $\left|F_{\lfloor n/j \rfloor}\right|$ are already available when the helper function is called to compute it.*

Proof. This corollary follows from Theorem 19. After the end of the first for-loop of Algorithm 5 the values of $|F_m|$ for $m \in \{1, 2, \ldots, \lfloor\sqrt{n}\rfloor\}$ are already computed. The second for-loop iterates over values of j from the list $(u(n), u(n)-1, \ldots, 1)$ in decreasing order starting from $u(n)$. Thus, the entries for keys from the set $Y(j)$ are already set when the control flow enters the helper function with $m = \lfloor n/j \rfloor$. The function uses keys from the sets $Q(\lfloor n/j \rfloor)$ and $K(\lfloor n/j \rfloor)$ to compute $\left|F_{\lfloor n/j \rfloor}\right|$. Because both $Q(\lfloor n/j \rfloor)$ and $K(\lfloor n/j \rfloor)$ are subsets of $Y(j)$, all required entries are available and the helper function can compute $\left|F_{\lfloor n/j \rfloor}\right|$ successfully. □

S11. Recursive version of FareyLengthC

Algorithm S12 gives the pseudo-code for a recursive version of Algorithm 5 (i.e., FareyLengthC). This version is a direct implementation of formula (5), which is replicated below:

$$|F_n| = \frac{(n+3)\,n}{2} - \sum_{k=2}^{u(n)} \left|F_{\lfloor n/k \rfloor}\right| - \sum_{k=1}^{\lfloor\sqrt{n}\rfloor} \left(\left\lfloor \frac{n}{k} \right\rfloor - \left\lfloor \frac{n}{k+1} \right\rfloor \right) \cdot |F_k|. \tag{117}$$

In this case, however, the required values of $|F_k|$ and $|F_{\lfloor n/k \rfloor}|$ are computed with recursive calls. These intermediate values are stored in the lookup table F. Similarly to the iterative version, this algorithm also runs in $O(n^{3/4})$ time and uses $O(\sqrt{n})$ memory. This can be shown by unpacking the recursive calls, which leads to the iterative version described in the main text.

Algorithm S12. Recursive algorithm for the length of the Farey sequence F_n. Runs in $O(n^{3/4})$ time and uses $O(\sqrt{n})$ memory.

```
function FareyLengthCR($n$, $F$ = null)
    $r \leftarrow$ ISQRT($n$);                    // call Algorithm S4 to compute $\lfloor\sqrt{n}\rfloor$ exactly
    $u \leftarrow \lfloor \frac{n}{r+1} \rfloor$;
    if $F$ is null then
        $F \leftarrow$ LookupTable($n$, $r+1$);   // initialize the lookup table $F$ for memoization
        $F[1] \leftarrow 2$;                      // set $F[1]$ to $|F_1| = 2$
    end if
    if $n$ in $F$ then
        return $F[n]$;
    end if
    $s \leftarrow 0$;
    for $k \leftarrow 2$ to $u$ do
        $s \leftarrow s +$ FareyLengthCR($\lfloor \frac{n}{k} \rfloor$, $F$);
    end for
    for $k \leftarrow 1$ to $r$ do
        $s \leftarrow s + (\lfloor \frac{n}{k} \rfloor - \lfloor \frac{n}{k+1} \rfloor) \cdot$ FareyLengthCR($k$, $F$);
    end for
    $F[n] \leftarrow \frac{(n+3)\,n}{2} - s$;     // update the lookup table
    return $F[n]$;
end function
```

S12. Computational Complexity Analysis of FareyLengthD

FareyLengthD can be viewed as a mixture between FareyLengthA and FareyLengthC. It picks an optimal point to divide the computation between these two algorithms as described below.

In order to compute $|F_n|$, FareyLengthC (i.e., Algorithm 5) computes $|F_m|$ for values of m that can be split into two sets M_1 and M_2 that correspond to the two for-loops in the algorithm. That is, the first for-loop computes the length of F_m for $m \in M_1$ and the second for-loop computes the length of F_m for $m \in M_2$. The set M_1 includes all integers between 1 and $\lfloor \sqrt{n} \rfloor$, i.e.,

$$M_1 = \big\{1, 2, \ldots, \lfloor \sqrt{n} \rfloor\big\}. \tag{118}$$

The set M_2 is defined by the following formula:

$$M_2 = \big\{\lfloor \tfrac{n}{u(n)} \rfloor, \lfloor \tfrac{n}{u(n)-1} \rfloor, \ldots, \lfloor \tfrac{n}{2} \rfloor, \lfloor \tfrac{n}{1} \rfloor\big\}. \tag{119}$$

Because $u(n) = \lfloor \frac{n}{\lfloor \sqrt{n} \rfloor + 1} \rfloor$, it follows that $\lfloor \frac{n}{u(n)} \rfloor \geq \lfloor \sqrt{n} \rfloor + 1$. Thus, each element of M_2 falls between $\lfloor \sqrt{n} \rfloor + 1$ and n. This set is sparse because it has $u(n)$ elements, but the total number of integers between $\lfloor \sqrt{n} \rfloor + 1$ and n is equal to $n - \lfloor \sqrt{n} \rfloor$. In other words, $u(n) \leq \lfloor \sqrt{n} \rfloor$ (see Lemma 5), but $n - \lfloor \sqrt{n} \rfloor > \lfloor \sqrt{n} \rfloor$ for each $n > 2$.

The first for-loop in FareyLengthC computes the values of $|F_m|$ for $m \in M_1$ in $O(n^{3/4})$ time. However, this is not the fastest way of performing this computation because FareyLengthA (i.e., Algorithm 1) solves a similar problem in linear time with respect to the number of computed values of $|F_m|$. That is, instead of $O(n^{3/4})$ time it is possible to perform this computation in $O(\sqrt{n})$ time.

Simply speeding up the first for-loop in Algorithm 5 won't affect its overall time complexity class because the second for-loop still runs in $O(n^{3/4})$ time. The second for-loop uses the recursive formula for computing the values of $|F_m|$ for each m in the set M_2. Because M_2 is sparse, switching to a sieve-based algorithm instead of the recursive formula won't speed up the computation of the required values of $|F_m|$ for $m > \lfloor \sqrt{n} \rfloor$.

Nevertheless, it is possible to change the split between the sets M_1 and M_2 so that the sieve-based approach processes more values of m, including some values that are greater than $\lfloor \sqrt{n} \rfloor$. Because that approach computes the values of $|F_m|$ for each integer m in the designated interval, it would still compute $|F_m|$ for each $m \in M_1$. In other words, the modified algorithm can run faster despite computing more values of $|F_m|$ than required for computing $|F_n|$ using the recursive formula.

Let $M_1^*(x)$ be a function that maps a real number x to the following set:

$$M_1^*(x) = \{1, 2, \ldots, \lfloor x \rfloor\}. \tag{120}$$

Also, let $\hat{u}(x)$ be the following function:

$$\hat{u}(x) = \left\lfloor \frac{n}{\lfloor x \rfloor + 1} \right\rfloor. \tag{121}$$

Furthermore, let $M_2^*(x)$ be another function that maps a real number to a set, which is defined as follows:

$$M_2^*(x) = \left\{ \left\lfloor \frac{n}{\hat{u}(x)} \right\rfloor, \left\lfloor \frac{n}{\hat{u}(x) - 1} \right\rfloor, \ldots, \left\lfloor \frac{n}{1} \right\rfloor \right\}. \tag{122}$$

Then, $M_1 = M_1^*(\sqrt{n})$ and $M_2 = M_2^*(\sqrt{n})$.

Using FareyLengthA we can compute $|F_m|$ for each $m \in M_1^*(x)$ in $O(x)$ time. Using FareyLengthC, we can compute $|F_m|$ for each $m \in M_2^*(x)$ in $O(g(x))$ time, where $g(x)$ is the following function:

$$g(x) = \sum_{m \in M_2^*(x)} \lfloor \sqrt{m} \rfloor = \sum_{j=1}^{\hat{u}(x)} \lfloor \sqrt{n/j} \rfloor. \tag{123}$$

The value of $g(x)$ can be bounded from above as follows:

$$g(x) = \sum_{j=1}^{\hat{u}(x)} \left\lfloor \sqrt{\frac{n}{j}} \right\rfloor \leq \sum_{j=1}^{\lfloor \frac{n}{x} \rfloor} \left\lfloor \sqrt{\frac{n}{j}} \right\rfloor \leq \sum_{j=1}^{\lfloor \frac{n}{x} \rfloor} \sqrt{\frac{n}{j}} = \sqrt{n} \sum_{j=1}^{\lfloor \frac{n}{x} \rfloor} \sqrt{\frac{1}{j}} = \sqrt{n} \sum_{j=1}^{\lfloor \frac{n}{x} \rfloor} \frac{1}{\sqrt{j}}. \tag{124}$$

Lemma 11 implies that the sum in the right-hand side of this inequality is in $O(\sqrt{\lfloor n/x \rfloor}\,)$. In other words,

$$g(x) \leq \sqrt{n} \sum_{j=1}^{\lfloor \frac{n}{x} \rfloor} \frac{1}{\sqrt{j}} = \sqrt{n}\, O\big(\sqrt{\lfloor n/x \rfloor}\,\big) = \sqrt{n}\, O\big(\sqrt{n/x}\,\big) = O\big(n/\sqrt{x}\,\big). \tag{125}$$

The time complexity of computing $|F_m|$ for each $m \in M_1^*(x) \cup M_2^*(x)$ by an algorithm that uses the sieving approach to process the set $M_1^*(x)$ and the recursive approach to process $M_2^*(x)$ is in $O(f(x))$, where $f(x)$ is the following function:

$$f(x) = x + \frac{n}{\sqrt{x}} \;=\; x + n\,x^{-\frac{1}{2}}. \tag{126}$$

What is the optimal value of x for which $f(x)$ is minimized? Differentiating $f(x)$ leads to the following formula for its first derivative:

$$f'(x) = 1 - \frac{1}{2}n\,x^{-\frac{3}{2}}. \tag{127}$$

The equation $f'(x) = 0$ has only one root at $x = ({}^{n}\!/\!{}_{2})^{2/3}$. Multiplying this solution by a constant factor that is independent of n does not affect the time complexity class of the overall computation because any constant factor is subsumed by the big-O notation. For simplicity, the optimal split x^* can be set to $\lfloor n^{2/3} \rfloor$.

FAREYLENGTHD is the result of the modifications to FAREYLENGTHC described above. It replaces the first for-loop with FAREYLENGTHA in order to process the set $M_1^*(\lfloor n^{2/3} \rfloor)$. It also modifies the second for-loop to start from $v = \left\lfloor {}^{n}\!/\!\left(\lfloor \sqrt[3]{n^2} \rfloor + 1\right) \right\rfloor$ instead of $u = \left\lfloor {}^{n}\!/\!\left(\lfloor \sqrt{n} \rfloor + 1\right) \right\rfloor$ in order to compute $|F_m|$ for each m in the set $M_2^*(\lfloor n^{2/3} \rfloor)$. The resulting algorithm runs in $O(n^{2/3})$ time because each of these two sub-parts requires $O(n^{2/3})$ time. The space complexity of this algorithm is determined by the sieve-based approach that processes the set $M_1^*(\lfloor n^{2/3} \rfloor)$. Because this step requires holding arrays of length $\lfloor n^{2/3} \rfloor$ in memory, FAREYLENGTHD uses $O(n^{2/3})$ memory.

S13. Alternative Version of FAREYLENGTHD

Algorithm S13 gives the pseudo-code for an alternative version of Algorithm 7 (i.e., FAREYLENGTHD). This version has the same run-time complexity and the same memory complexity as the one described in the main paper. It is shorter than the other algorithm, but updates more entries of the lookup table than necessary to compute $|F_n|$. It also obscures the link to Algorithm 8 (i.e., FAREYLENGTHE), which improves the space complexity. Additional details are provided in the main paper.

Algorithm S13. Alternative version of Algorithm 7. Runs in $O(n^{2/3})$ time and uses $O(n^{2/3})$ memory.

```
1:  function FAREYLENGTHD2($n$)
2:      $c \leftarrow$ ICBRT($n^2$);
3:      $v \leftarrow \lfloor \frac{n}{c+1} \rfloor$;
4:      $F \leftarrow$ LOOKUPTABLE($n$, $c+1$);
5:      $(P, L_p) \leftarrow$ LINEARSIEVE($c$);
6:      $\varphi \leftarrow$ COMPUTETOTIENTS($c, L_p$);
7:      $s \leftarrow 1$;
8:      for $m \leftarrow 1$ to $c$ do
9:          $s \leftarrow s + \varphi[m]$;
10:         $F[m] \leftarrow s$;
11:     end for
12:     for $j \leftarrow v$ down to $1$ do
13:         UPDATELOOKUPTABLE($F$, $\lfloor n/j \rfloor$);
14:     end for
15:     return $F[n]$;
16: end function
```

S14. Computational Complexity Analysis of FareyLengthE

The following theorem proves the time and space complexity of FareyLengthE (i.e., Algorithm 8).

Theorem 21. *Algorithm 8 runs in $O(n^{2/3})$ and uses $O(\sqrt{n})$ memory.*

Proof. The time complexity of Algorithm 8 is in the same complexity class as Algorithm 7, i.e., $O(n^{2/3})$. More specifically, FareyLengthE starts by processing the integers in the interval $[1, \lfloor\sqrt{n}\rfloor]$ using the linear sieve and Algorithm S3. Next, the algorithm enumerates all $\lfloor\sqrt{n}\rfloor$-smooth numbers in the interval $[\lfloor\sqrt{n}\rfloor+1, \lfloor n^{2/3}\rfloor]$ using Algorithm 3 and adds their totients to the corresponding elements of the array B. This step requires $O(n^{2/3})$ time and uses $O(\sqrt{n})$ memory. Next, the algorithm enumerates all numbers that are not $\lfloor\sqrt{n}\rfloor$-smooth in the interval $[\lfloor\sqrt{n}\rfloor+1, \lfloor n^{2/3}\rfloor]$ using Algorithm 4 and also adds their totients to the corresponding elements of B. This step requires $O(n^{2/3})$ time and $O(\sqrt{n})$ memory because it uses the sieve of Atkin[19] to find the prime numbers between $\lfloor\sqrt{n}\rfloor+1$ and $\lfloor n^{2/3}\rfloor$. Finally, the algorithm computes the value of $|F_m|$ for each m in the set $\left\{\left\lfloor\frac{n}{v(n)}\right\rfloor, \left\lfloor\frac{n}{v(n)-1}\right\rfloor, \dots, \left\lfloor\frac{n}{1}\right\rfloor\right\}$ using the helper function. Similarly to FareyLengthD, this step requires $O(n^{2/3})$ time and $O(\sqrt{n})$ memory. Therefore, FareyLengthE runs in $O(n^{2/3})$ time and its space complexity is $O(\sqrt{n})$. □

The following three theorems prove that the computation in Algorithm 8 is correct. They show that the set of integers in the interval $[\alpha, \beta]$ that the algorithm enumerates is the correct set, where $\alpha = \lfloor\sqrt{n}\rfloor + 1$ and $\beta = \lfloor n/(v(n)+1)\rfloor$. The theorems also show that the formula for mapping k to the index i of the array element $B[i]$ is also correct for each integer k in $[\alpha, \beta]$.

The first theorem shows that the set $S(u(n), n)$, which consists of all integers j such that $\left\lfloor\frac{n}{j}\right\rfloor = u(n)$, includes the value of $\lfloor\sqrt{n}\rfloor+1$. This implies that $\alpha = \lfloor\sqrt{n}\rfloor+1$ is a suitable split point for switching between the linear sieve approach and enumeration of smooth and non-smooth integers. For this value of α the algorithm computes the value of $B[0]$ correctly because the split point is an element of $S(u(n), n)$ so that all integers in $S(u(n), n)$ are accounted for either by the computation of $|F_1|, |F_2|, \dots, |F_{\lfloor\sqrt{n}\rfloor}|$ using the linear sieving approach or by enumerating smooth and non-smooth numbers.

Theorem 22. *Let $S(k, n)$ be a set of all positive integers m such that the value of $\left\lfloor\frac{n}{m}\right\rfloor$ is equal to k. That is,*

$$S(k,n) = \left\{m \in \mathbb{N} \text{ s.t. } \left\lfloor\tfrac{n}{m}\right\rfloor = k\right\}. \tag{128}$$

Then, $\lfloor\sqrt{n}\rfloor + 1 \in S(u(n), n)$. Moreover, the set $S(u(n), n)$ is a contiguous range of integers such that

$$S(u(n),n) = \left\{\left\lfloor\tfrac{n}{u(n)+1}\right\rfloor+1, \left\lfloor\tfrac{n}{u(n)+1}\right\rfloor+2, \dots, \left\lfloor\tfrac{n}{u(n)}\right\rfloor\right\}, \tag{129}$$

$$\min S(u(n),n) = \left\lfloor\tfrac{n}{u(n)+1}\right\rfloor+1 \le \lfloor\sqrt{n}\rfloor+1 \le \max S(u(n),n) = \left\lfloor\tfrac{n}{u(n)}\right\rfloor. \tag{130}$$

Proof. Formula (129) follows from Lemma 4 after plugging $u(n)$ as the value of k in formula (29). Formula (130) follows from (129) and the definitions for the set $S(u(n), n)$ and the function $u(n)$. That is, $\lfloor\sqrt{n}+1\rfloor \in S(u(n), n)$ because $S(u(n), n)$ is a contiguous range of integers. Thus, $\lfloor\sqrt{n}+1\rfloor$ lies between its minimum and maximum elements. □

The second theorem shows that the endpoint $\beta = \left\lfloor\frac{n}{v(n)+1}\right\rfloor$ that Algorithm 8 uses as the upper limit for enumerating smooth and non-smooth numbers is the correct endpoint. That is, it shows that the enumeration process covers all elements of the sets $S(u(n)-1, n), S(u(n)-2, n), \dots, S(v(n)+1, n)$, which is required for computing the elements of the array B correctly.

Theorem 23. *The union of the sets $S(u(n)-1, n), S(u(n)-2, n), \dots, S(v(n)+1, n)$ is equal to the contiguous range of integers between $\left\lfloor\frac{n}{u(n)}\right\rfloor+1$ and $\left\lfloor\frac{n}{v(n)+1}\right\rfloor$, where $v(n) = \left\lfloor {}^{n}\!/\!\left(\lfloor\sqrt[3]{n^2}\rfloor+1\right)\right\rfloor$. More formally,*

$$\bigcup_{k=v(n)+1}^{u(n)-1} S(k,n) = \left\{\left\lfloor\tfrac{n}{u(n)}\right\rfloor+1, \left\lfloor\tfrac{n}{u(n)}\right\rfloor+2, \dots, \left\lfloor\tfrac{n}{v(n)+1}\right\rfloor\right\}. \tag{131}$$

Proof. This theorem follows from equation (29) in the statement of Lemma 4, which implies that the set $S(u(n)-1, n)$ is a contiguous range of integers that starts from $\left\lfloor\frac{n}{u(n)}\right\rfloor+1$, the set $S(u(n)-2, n)$ is a contiguous range of integers that follows $S(u(n)-1, n)$ without any gaps in-between, and so forth until $S(v(n)+1, n)$. The lemma also shows that the maximum element of $S(v(n)+1, n)$ is equal to $\left\lfloor\frac{n}{v(n)+1}\right\rfloor$. □

The last theorem proves that the formula $i = u(n) - \left\lfloor\frac{n}{m}\right\rfloor$ that Algorithm 8 uses to compute the value of the index for the array B in the visitor function correctly identifies the zero-based indices of these elements. This theorem implies that $i = 0$ corresponds to $S(u(n), n)$, $i = 1$ corresponds to $S(u(n)-1, n)$, etc., until $i = w(n)$, which maps to $S(v(n)+1, n)$, where $w(n) = u(n) - v(n) - 1$.

Theorem 24. *Let m and n be two positive integers and let m be an element of the set $S(k, n)$. Then, $k = u(n) - i$, where $i = u(n) - \left\lfloor\frac{n}{m}\right\rfloor$.*

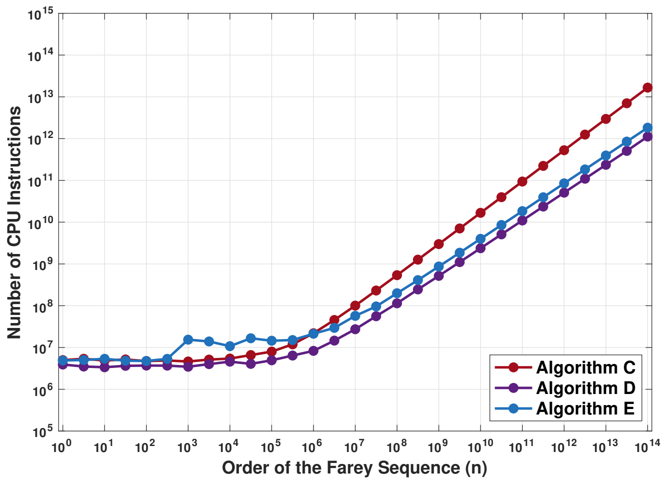
Proof. The proof follows from the definition of the set $S(k, n)$ in Lemma 4. That is, $m \in S(k, n)$ implies that $\left\lfloor\frac{n}{m}\right\rfloor = k$. Therefore, $i = u(n) - \left\lfloor\frac{n}{m}\right\rfloor = u(n) - k$. Thus, $k = u(n) - i$. □

S15. Results for the Number of CPU Instructions

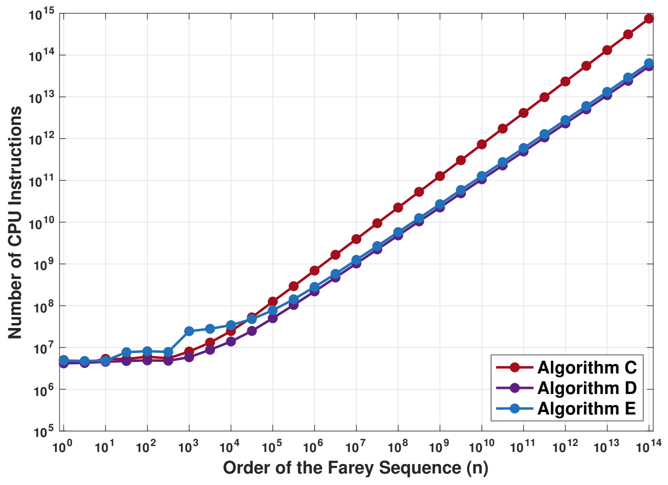
In addition to the run time and memory usage results described in the main paper, we also measured the number of CPU instructions that the code executed (see Methods). Figure S4a visualizes these additional results for the C implementations of algorithms C, D, and E. Figure S4b shows the results obtained with the Python versions of these three algorithms. In both plots the value of n was varied from 10^0 to 10^{14} in increments of 0.5 on the decimal logarithm scale. Figure S4c lists the slopes and intercepts for six lines that were fitted to the six curves in (a) and (b) using least squares. The fitting procedure used the region between $n = 10^{10}$ and $n = 10^{14}$.

These results show that the number of CPU instructions agrees with the theoretical time complexities. Each slope is within 0.01 of the corresponding power of n in the time complexity class, i.e., $3/4$ for algorithm C and $2/3$ for algorithms D and E. More specifically, the slopes for the two programming language implementations of algorithm C are equal to 0.7494 and 0.7520, respectively. For algorithm D they are equal to 0.6672 and 0.6754. For algorithm E they are equal to 0.6655 and 0.6744.

This precise agreement with the theory suggests that the run time results reported in Figure 8 may be slightly higher than the theoretical predictions due to the practical aspects of running the code on modern processors. One of them could be increasingly inaccurate branch prediction as n increases. Another could be a greater number of cache misses for larger n. Because the time spent waiting for the cache to be updated does not affect the CPU instruction counter for the current process, this metric agrees with the theory very well.



(a) Number of instructions executed by the C code.



(b) Number of instructions executed by the Python code.

Alg.	C Code		Python Code	
	Slope	Intercept	Slope	Intercept
C	0.7494	2.7289	0.7520	4.3427
D	0.6672	2.7017	0.6754	4.2639
E	0.6655	2.9445	0.6744	4.3561

(c) Lines fitted to the curves in (a) and (b) for $n \geq 10^{10}$.

Figure S4: Number of CPU instructions executed by algorithms C, D, and E. The plots in (a) show the results for the C implementations of the algorithms. The results for the Python versions are shown in (b). The table in (c) lists the slopes and intercepts for lines fitted to the six curves in (a) and (b) in the region between $n = 10^{10}$ and $n = 10^{14}$.