Order Statistics in the Farey Sequences in Sublinear Time and Counting Primitive Lattice Points in Polygons*

Jakub Pawlewicz
pan@mimuw.edu.pl
Institute of Informatics
Warsaw University

Mihai Pătrașcu
mip@mit.edu
MIT

August 3, 2008

Abstract

We present the first sublinear-time algorithms for computing order statistics in the Farey sequence and for the related problem of ranking. Our algorithms achieve a running times of nearly $O(n^{2/3})$, which is a significant improvement over the previous algorithms taking time $O(n)$.

We also initiate the study of a more general problem: counting primitive lattice points inside planar shapes. For rational polygons containing the origin, we obtain a running time proportional to $D^{6/7}$, where D is the diameter of the polygon.

1 Introduction

1.1 Algorithms for the Farey Sequence

The Farey sequence of order n (denoted $\mathcal{F}_n$) is the increasing sequence of all irreducible fractions from the interval $[0,1]$ with denominators less than or equal to n. In other words, $\mathcal{F}_n$ denotes the sorted set $\{\frac{a}{b} \mid 0 \leq a \leq b \leq n\}$. Farey sequences have numerous interesting properties, and are well known in number theory and combinatorics. We refer the reader to [7] for a comprehensive list of such properties and applications.

Several algorithms for generating the entire sequence $\mathcal{F}_n$ in $O(n^2)$ time are known, e.g. [7, Problem 4-61]. This running time is optimal, as the sequence $\mathcal{F}_n$ has $\frac{3}{\pi^2} \cdot n^2 + O(n \log n)$ terms. In this paper, we study two algorithmic problems which give us "local" access to the sequence, even when we cannot afford to generate it entirely:

- the *order statistic problem* is to find the fraction at a given index in the sequence. Formally, for $k \leq |\mathcal{F}_n|$, let $\textsc{Stat}(k, n)$ be the k-th value in $\mathcal{F}_n$ (in sorted order).
- the *rank problem* is to find the rank of a given fraction in the sequence. Formally, given a number $\alpha \in [0,1]$, let $\textsc{Rank}(\alpha, n) = \big|\mathcal{F}_n \cap [0, \alpha]\big|$.

These problems have been studied previously in [20], where the following results are obtained:

- the $\textsc{Stat}$ problem can be solved with $O(\log n)$ calls to the $\textsc{Rank}$ problem, plus additional $O(n)$ time.
- a $\textsc{Rank}$ problem can be solved in $O(n)$ time.

*This work represents a merging of [19] and [21], with additional extensions.

- a number n can be *factored* by $O(\log n)$ calls to STAT *or* RANK. Under the typical assumption that factoring is hard, we conclude that it is not possible to find algorithms for these problems running in time $(\log n)^{O(1)}$, i.e. polynomial time.

The authors of [20] remark that their solution to the rank problem could run in time $n^{5/6+o(1)}$ if it was possible to compute the sum $\sum_{i=a}^{b} \lfloor x \cdot i \rfloor$, for rational x, in time $n^{o(1)}$. This sum is related to counting lattice points in right triangles, and in fact, a simple algorithm for that task running in logarithmic time has been discovered [24]. For completeness, we present another logarithmic solution in Appendix A, which is also useful in our improved results.

Unfortunately, this $n^{5/6+o(1)}$ solution is complicated. It involves summation of the Möbius function, and subexponential integer factorization. As the main result of this paper, Section 2 presents a simple and faster algorithm for the rank problem with time complexity $O(n^{2/3} \log^{1/3} n)$. Our result assumes the RAM model of computation, where usual arithmetic operations can be performed on cells of $O(\log n)$ bits in constant time.

We also show that the rank problem is reducible to the problem of evaluating the Mertens function (the summatory Möbius function) at all values $\lfloor \frac{n}{d} \rfloor$, for integer $d \geq 1$. Using this technique, we obtain a slight improvement of our algorithm to $O\big(n^{2/3}(\log \log n)^{1/3}\big)$.

Order Statistics. Using the reduction in [20], a sublinear-time algorithm for RANK does not imply a sublinear algorithm for STAT, due to the additive $O(n)$ term. However, a better reduction is already known, using a technique called *rational search.* The rational search problem is to find an unknown fraction $\frac{a}{b}$, with $a \leq b \leq n$, by using only comparison queries. In our case, a comparison query is implemented by the RANK problem: if the rank is greater than k, the guessed fraction is larger than the target fraction.

Integer search is of course solved by binary search with $\lceil \log_2 n \rceil$ queries. In the rational case, Papadimitriou [18], Reiss [22], Kwek and Mehlhorn [11], and Forišek [6] give similar solutions, which all use $O(\log n)$ queries.

Combining with our running time for RANK, this gives an $O\big(n^{2/3} \log n (\log \log n)^{1/3}\big)$ algorithm for the order statistic problem. It should be noted that RANK can be easily solved with $O(\log n)$ calls to STAT, by a simple binary search. Thus, the complexity of RANK and STAT are equal up to a logarithmic factor.

Overall, our results put local access to the Farey sequence on a similar footing to other summatory problems in algorithmic number theory. In problems such as computing the number of primes less than n [1], or computing the Mertens function [5], algorithms with some sublinear running time have been developed, while running times of $n^{o(1)}$ are generally considered unattainable.

1.2 Counting Primitive Lattice Points

In this paper, a lattice point refers to a point in $(x, y) \in \mathbb{Z}^2$. A *primitive* lattice point is a lattice point (x, y) satisfying $\gcd(x, y) = 1$. These are sometimes called "points visible from the origin", since a point (x, y) is visible by a straight line only if the slope of the line, $\frac{x}{y}$, is irreducible.

One can observe that answering $\text{RANK}(\alpha, n)$ is equivalent to counting primitive lattice points in the right triangle defined by the origin, $(n, 0)$, and $(n, \alpha n)$. See Figure 1. This follows because a Farey fraction $\frac{y}{x} \in \mathcal{F}_n$ with $\frac{y}{x} \leq \alpha$ is equivalent to a primitive point (x, y) with $0 \leq x \leq n$ and $0 \leq y \leq \alpha x$, i.e. inside the region defined by the triangle.

Counting primitive lattice points in planar shapes is a relatively new topic in mathematics, but one that is gathering significant momentum [12, 14, 8, 15, 9, 13, 16, 10, 26, 4, 23, 25, 17]. In the mathematical sense, "counting" refers to estimating the number of primitive points with a small error, as the size of the shape goes to infinity. Exact counting is hopeless by an elementary formula.

In this paper, we initiate the study of the *algorithmic* problem of counting (exactly) the number of primitive lattice points inside a given shape. More precisely, we study this problem for polygons containing the origin. The condition that the shape should contain the origin also appears in the mathematical works referenced above, and is natural given that we are counting points visible from the origin.

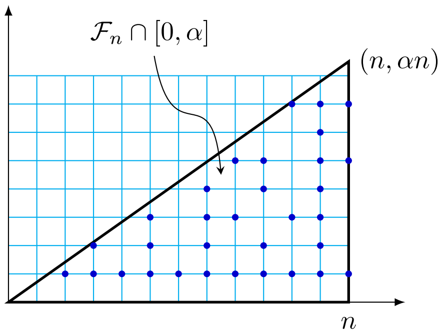

Figure 1: The RANK problem is equivalent to counting primitive lattice points inside a right triangle.

Theorem 1. *Let P be a polygon containing the origin, defined by k vertices at b-bit rational coordinates. If D is the diameter of the polygon, one can count the number of primitive lattice points inside P in time $D^{6/7} \cdot k \cdot b^{O(1)}$.*

This theorem is shown in Section 3.

A very pertinent question is how efficient this running time actually is. Remember that [20] shows that the Farey rank problem can be used to factor integers. Since counting primitive lattice points is a generalization, we conclude that a polynomial-time algorithm, i.e. $\mathrm{poly}(k \cdot b)$, is likely impossible. Thus, the algorithm needs to depend on some parameter describing the polygon, which can be exponential in b. One such parameter is the diameter D. Clearly, however, this is not the only choice, and it is conceivable that other measures lead to better results. For right triangles such as those in the Farey rank problem, the diameter is n, yet we know a better algorithm with time essentially $n^{2/3}$.

A trivial alternative to Theorem 1 is the algorithm which iterates over all lattice points inside the polygon, and runs Euclid's algorithm on each point. It is possible [2, 3, 24] to list all lattice points with a polynomial $\mathrm{poly}(k \cdot b)$ cost per point. If the polygon has integral coordinates, Pick's formula shows that the number of lattice points inside is asymptotically equal to the area. Thus, the exhaustive algorithm has complexity proportional to the area, times polynomial factors.

Unfortunately, the area and the diameter are not related in the worst-case (e.g., for very skinny shapes). However, in the more "typical" case when the polygon is fat, the area is $A = \Theta(D^2)$. Thus our running time of $D^{6/7}$ can be rewritten as $A^{3/7}$, which gives a significant saving over the exhaustive algorithm. It is an interesting open problem to construct an algorithm which beats exhaustive search for *any* polygon.

2 An Algorithm for the Rank Problem

Let S_n be the quantity we are searching for, i.e. the number of irreducible fractions $\frac{a}{b} \le x$ with $b \le n$. Grouping reducible fractions by gcd, we obtain the following recursive formula, which will be the starting point of our algorithm:

$$\begin{aligned} S_n(x) &= \left|\left\{\frac{a}{b} \mid b \le n \ \wedge\ \frac{a}{b} \le x \ \wedge\ \gcd(a,b) = 1\right\}\right| \\ &= \left|\left\{\frac{a}{b} \mid b \le n \ \wedge\ \frac{a}{b} \le x\right\}\right| - \sum_{d \ge 2}\left|\left\{\frac{a}{b} \mid b \le n \ \wedge\ \frac{a}{b} \le x \ \wedge\ \gcd(a,b) = d\right\}\right| \\ &= \sum_{b=1}^{n} \lfloor bx \rfloor - \sum_{d \ge 2} S_{\lfloor \frac{n}{d} \rfloor}(x). \end{aligned}$$

Let us explain the steps of the reasoning. The number of irreducible fractions can be computed as the total number of fractions minus the number of fractions with gcd of numerator and denominator equal to some

$d \geq 2$. The total number of fractions with denominator b less than or equal to x is $\lfloor bx \rfloor$. Finally, every fraction $\frac{a}{b}$ with $\gcd(a,b) = d$ can be reduced to $\frac{a'}{b'}$, with $a' = \frac{a}{d}, b' = \frac{b}{d}$. Such a fraction is counted in $S_{\lfloor \frac{n}{d} \rfloor}(x)$ because it is irreducible, $\frac{a'}{b'} = \frac{a}{b} \leq x$, and $b' = \frac{b}{d} \leq \frac{n}{d}$.

We thus obtain the following recursive formula:

$$S_n = \sum_{b=1}^{n} \lfloor bx \rfloor - \sum_{d \geq 2} S_{\lfloor \frac{n}{d} \rfloor}. \tag{1}$$

Observe that as we unravel the recursion, a quantity S_i can only appear if $i = \lfloor \frac{n}{d} \rfloor$, for some positive integer d. This property follows from the equality:

$$\left\lfloor \frac{\lfloor \frac{n}{d_1} \rfloor}{d_2} \right\rfloor = \left\lfloor \frac{n}{d_1 d_2} \right\rfloor.$$

Thus, our task is to compute all S_i values, where i is from set $I = \left\{ \left\lfloor \frac{n}{d} \right\rfloor \mid d \geq 1 \right\}$. Though the set I iterates over $d = 1$ to $d = n$, it is not hard to see that it contains only $O(\sqrt{n})$ distinct values. For $d \leq \sqrt{n}$ all expressions $S_{\lfloor \frac{n}{d} \rfloor}(x)$ are unique. If $d > \sqrt{n}$, then $\frac{n}{d} < \sqrt{n}$, and there are at most $\sqrt{n}$ distinct values.

Our task will be to compute these $O(\sqrt{n})$ different quantities, implementing recursion (1) by dynamic programming. The "constant" $\sum_{b=1}^{i} \lfloor bx \rfloor$ in the recursion for S_i can be calculated in $O(\text{polylog}(n))$, as shown in Appendix A. All these computations can be done in the beginning, in time $O(\sqrt{n} \cdot \text{polylog}\, n)$. It remains to see how efficiently the dynamic program can be evaluated.

An $O(n^{3/4})$ algorithm. As explained above, the sum $\sum_{d \geq 2} S_{\lfloor \frac{i}{d} \rfloor}$ only contains $O(\sqrt{i})$ distinct summands, since values start repeating for $d > \sqrt{i}$. For each symbol S_j occurring in the sum we can find its multiplicity in constant time, by finding the interval of d values for which $\lfloor \frac{n}{d} \rfloor = j$. Therefore, if we have already computed S_j for $j < i$, the time complexity of calculating S_i is $O(\sqrt{i})$.

Surprisingly, this algorithm works in time $O(n^{3/4})$. We prove this in two parts. First, let us determine the time for calculating S_i for all $i \leq \lfloor \sqrt{n} \rfloor$:

$$\sum_{i=1}^{\sqrt{n}} O(\sqrt{i}) \leq \sum_{i=1}^{\sqrt{n}} O\left(\sqrt{\sqrt{n}}\right) = O\left(\sqrt{n} \cdot n^{1/4}\right) = O(n^{3/4}).$$

Now consider $i \in \{ \lfloor \frac{n}{d} \rfloor \mid 1 \leq d \leq \sqrt{n} \}$. The time needed to calculating all these S_i's is:

$$\sum_{d=1}^{\sqrt{n}} O\left(\sqrt{\left\lfloor \frac{n}{d} \right\rfloor}\right) = O\left(\sqrt{n} \sum_{d=1}^{\sqrt{n}} \frac{1}{\sqrt{d}}\right) = O\left(\sqrt{n} \cdot \sqrt{\sqrt{n}}\right) = O(n^{3/4}).$$

Here we have used the asymptotic equality $\sum_{1 \leq i \leq x} \frac{1}{\sqrt{i}} = O(\sqrt{x})$.

An $O(n^{2/3} \log^{1/3} n)$ algorithm. The key to this improvement is to show that $S_1, \ldots, S_k$ can be computed in time $O(k \log k)$, for any k. Then, the remaining terms can be computed by the old algorithm in time:

$$\sum_{d=1}^{n/k} O\left(\sqrt{\left\lfloor \frac{n}{d} \right\rfloor}\right) = O\left(\sqrt{n} \sum_{d=1}^{n/k} \frac{1}{\sqrt{d}}\right) = O\left(\sqrt{n} \cdot \sqrt{\frac{n}{k}}\right) = O\left(\frac{n}{\sqrt{k}}\right).$$

The total running time is then $O(k \log k + \frac{n}{\sqrt{k}})$. We optimize k by setting $k\sqrt{k} \cdot \log k = n$, leading to $k = (n / \log n)^{2/3}$. Thus, the running time will be $O(n^{2/3} \log^{1/3} n)$.

To compute $S_1, \ldots, S_k$ efficiently, we make the observation that the composition (with respect to recursive terms) of S_i and S_{i-1} are not too different. Specifically, $\sum_{d\geq 2} S_{\lfloor i/d \rfloor}$ differs from $\sum_{d\geq 2} S_{\lfloor (i-1)/d \rfloor}$ only for the values of d that i is a multiple of.

To maintain understanding of divisibility by all d's, and compute $S_i(x)$ values in order, we use an algorithm similar in flavor to Eratosthenes sieve. We first create an array $D[1 \ldots k]$, where $D[i]$ holds a list of all divisors of i. To create the array, simple consider all d, and add d to $D[md]$, for all m. This takes time $\sum_{d\geq 1} \frac{k}{d} = O(k \log k)$.

Now, iterate i from 1 to k, maintaining $\sum_{d\geq 2} S_{\lfloor i/d \rfloor}$ at all times. The sum is updated by considering all $d \in D[i]$, subtracting $S_{\lfloor (i-1)/d \rfloor}$ and adding $S_{\lfloor i/d \rfloor}$. The complexity is linear in the size of $D[1 \ldots k]$, and is thus $O(k \log k)$.

An application. Our algorithm works whenever we have a recursive formula of the following form:

$$\sum_{d\geq 1} f\left(\left\lfloor \frac{n}{d} \right\rfloor\right) = g(n). \tag{2}$$

If $g(\cdot)$ is computable in "reasonable" time, our algorithm is able to compute some $f(n)$ in time proportional to $n^{2/3}$.

As an example, we can compute the Mertens function $M(n)$, which is the summation of the Möbius function, $M(n) = \sum_{k=1}^{n} \mu(k)$. It is known that $M(n)$ satisfies $\sum_{d\geq 1} M(\lfloor \frac{n}{d} \rfloor) = 1$, i.e. we have $g(n) = 1$. Thus, we immediately obtain an algorithm for computing $M(n)$ in time $O(n^{2/3} \log^{1/3} n)$.

Using a more careful analysis of the Mertens function, Deléglise and Rivat [5] show that the sequence $M(1), \ldots, M(k)$ can be computed even faster, in time $O(k \log\log k)$. This leads to overall algorithm running in time $O\big(n^{2/3} (\log\log n)^{1/3}\big)$. This is the same running time as that of [5], but their algorithm is based on more complicated combinatorial identities.

An $O\big(n^{2/3}(\log\log n)^{1/3}\big)$ algorithm. In fact, the connection to the Mertens function feeds back to our problem, generating a slightly better solution. Applying Möbius inversion to (2), we have:

$$f(n) = \sum_{d\geq 1} \mu(d) g\left(\left\lfloor \frac{n}{d} \right\rfloor\right). \tag{3}$$

Here we have to compute $g(i)$, when i is in the set $I = \left\{ \lfloor \frac{n}{d} \rfloor \mid d \geq 1 \right\}$. As we have seen, this set has size $O(\sqrt{n})$ and many terms $g(i)$ are repeating on the right hand side of (3). Let us rewrite (3) to take advantage of this fact:

$$\begin{aligned} f(n) &= \sum_{d\geq 1} \mu(d) g\left(\left\lfloor \frac{n}{d} \right\rfloor\right) = \sum_{i\in I} \sum_{\lfloor \frac{n}{d} \rfloor = i} \mu(d) g(i) = \sum_{i \in I} g(i) \sum_{i \leq \frac{n}{d} < i+1} \mu(d) \\ &= \sum_{i\in I} g(i) \sum_{\frac{n}{i+1} < d \leq \frac{n}{i}} \mu(d) = \sum_{i\in I} g(i) \left(M\left(\left\lfloor \frac{n}{i} \right\rfloor\right) - M\left(\left\lfloor \frac{n}{i+1} \right\rfloor\right)\right). \end{aligned}$$

Observe that our $O\big(n^{2/3}(\log\log n)^{1/3}\big)$ algorithm for computing $M(n)$ also computes all $M(\lfloor \frac{n}{d} \rfloor)$ for all $d \geq 1$. We have to compute $g(i)$ only $O(\sqrt{n})$ times. Therefore, an efficient algorithm for computing $g(i)$ will make this component a second-order term, and imply an $O\big(n^{2/3}(\log\log n)^{1/3}\big)$-time algorithm for $f(n)$.

For Rank we know how to compute $g(i)$ (counting lattice points in right triangles) in time $O(\log i)$, which is more than sufficient for using in the above scheme, and imply the desired running time. Moreover, if a faster algorithm is discovered for computing the sequence $M(\lfloor \frac{n}{d} \rfloor)$, for all $d \geq 1$, then we immediately get a faster algorithm for the rank problem. Note, however, that this approach cannot obtain a running time below $O(\sqrt{n})$.

3 An Algorithm for Counting Primitive Points

Let P be a polygon, defined by rational points $(x_1, y_1), \ldots, (x_k, y_k)$. Then, let $P_{/d}$ be the polygon defined by $(\frac{x_1}{d}, \frac{y_1}{d}), \ldots, (\frac{x_k}{d}, \frac{y_k}{d})$. Define $A(P)$ to be the number of lattice points inside polygon P, and $S(P)$ the number of primitive lattice points inside P.

We first observe the following recursive formula:

$$S(P) = A(P) - \sum_{d \geq 2} S(P_{/d}). \tag{4}$$

Indeed, every point in $A(P)$, but not in $S(P)$ is a nonprimitive lattice point (x, y). If $\gcd(x, y) = d > 1$, then $(\frac{x}{d}, \frac{y}{d})$ is a primitive lattice point. Furthermore, such a point is inside $P_{/d}$, so we can remove all points with a greater common divisor equal to d by subtracting $P_{/d}$.

We can bound the recursion depth in (4), by appealing to the diameter D. We note that the diameter of $P_{/(D+1)}$ is less than 1, so it does not contain any lattice point outside the origin. Thus, $S(P_{/d}) = 0$ for all $D < d < \infty$, and $S(P_{/\infty}) = 1$ (the origin). Then, it suffices to consider only $P, P_{/2}, \ldots, P_{/D}$ in the algorithm.

To compute the "constants" $A(P_{/i})$ in the recursion, one needs to compute the number of lattice points inside polygons with rational coordinates. This is a well-studied problem, and [2, 3, 24] given polynomial-time algorithms. Observe that for $i \leq D \leq 2^b$, coordinates of $P_{/i}$ have at most $2b$ bits of precision. Thus, computing $A(P_{/i})$ takes time $O(k \cdot \text{poly}(b))$; the dependence on k is linear by triangulating the polygon.

Note that formula (4) is very similar to the recursive formula for the rank problem (1). Indeed, (1) is simply a transcription of (4), where the polygons are right triangles as in Figure 1.

It is thus tempting to conjecture that we can use the same dynamic program to evaluate (4). Unfortunately, this is not the case, for a somewhat subtle reason. Due to the geometry of the rank problem, $S_{n/d}$ was the same as $S_{\lfloor n/d \rfloor}$. By taking floors, we concluded that among $S_{\lfloor n/d \rfloor}$'s with $d \in \{\sqrt{n}, \ldots, n\}$, there are only $\sqrt{n}$ distinct quantities. Unfortunately, in general we cannot round the vertices of $P_{/d}$ to lattice points, and thus we cannot conclude that for $d \geq \sqrt{D}$ there are only $\sqrt{D}$ distinct cases.

A more careful analysis. Since we are not interested in factors of $k \cdot b^{O(1)}$ in the running time, let us define $O^*(f) = f \cdot k \cdot b^{O(1)}$. To use ideas similar to the previous dynamic program, we begin by obtaining a weaker bound for computing the small terms of the recurrence:

Lemma 2. *We can (implicitly) compute $S(P_{/\tau}), S(P_{/(\tau+1)}), \ldots, S(P_{/D})$ in time $O^*\big(\frac{D^2}{\tau^2}\big)$.*

Proof. Note that $P_{/D} \subseteq P_{/(D-1)} \subseteq \cdots \subseteq P_{/\tau}$. Also, the diameter of $P_{/\tau}$ is D/τ, implying that $P_{/\tau}$, and any smaller polygon only contain $O\big((\frac{D}{\tau})^2\big)$ lattice points. Since $S(P_{/i})$ can only grow between 1 and $O\big((\frac{D}{\tau})^2\big)$ when i goes from D to τ, there are only so many distinct values that can appear.

To actually compute these values, we perform an exhaustive enumeration of all lattice points in $P_{/\tau}$. Points which are not primitive are discarded. For every primitive point (x, y), we compute a value $\varphi(x, y)$ which is the minimum i such that $P_{/i}$ does not contain it. This is done by a binary search for i. Every comparison is a point-in-polygon test, which takes $O(k)$ time.

We now sort the $\phi(x, y)$ values from largest to smallest. The sorted list gives us an *implicit* representation for $S(P_{/i})$ for every $i \geq \tau$. Indeed, it suffices to binary search for the first occurrence of i; the elements to the left correspond to primitive points which are inside $P_{/i}$. □

Let us now consider the problem of computing a term using our recursion:

$$S(P_{/i}) = A(P_{/i}) - \sum_{d \geq 2} S(P_{/id}).$$

We assume terms $S(P_{/j})$ for $j \geq i$ have already been computed; in particular, for $j \geq \tau$ we have the implicit representation from Lemma 2.

In principle, the expression for $S(P_{/i})$ has $\lfloor \frac{D}{i} \rfloor$ terms. However, as in Lemma 2, we can observe that for fixed δ, there are only $O\left(\frac{D^2}{\delta^2}\right)$ distinct terms among all $S(P_{/id})$'s with $id \geq \delta$. These terms can actually be summed up in $O^*\left(\frac{D^2}{\delta^2}\right)$ time. Indeed, we begin with $d_0 = \lceil \frac{\delta}{i} \rceil$, and binary search for the minimum d_1 such that $S(P_{/id_1}) < S(P_{/id_0})$. We add to the sum $S(P_{/id_0}) \cdot (d_1 - d_0)$, then binary search for the minimum d_2 leading to a different value, and so on.

After dealing like this with all terms $d \geq \frac{\delta}{i}$, we can simply add the remaining terms. The running time for computing $S(P_{/i})$ is therefore $O^*\left(\frac{D^2}{\delta^2} + \frac{\delta}{i}\right)$. We can optimize by setting $\delta = D^{2/3} i^{1/3}$, yielding a running time of $O^*\left(\left(\frac{D}{i}\right)^{2/3}\right)$.

We now want to optimize the parameter τ in Lemma 2. A choice of τ implies that we spend $O^*\left(\frac{D^2}{\tau^2}\right)$ time by Lemma 2, and then run the dynamic program for terms $S(P_{/i})$ with $i < \tau$. This second part will take time:

$$\sum_{i=1}^{\tau} O^*\left(\frac{D}{i}\right)^{2/3} = O^*(D^{2/3}\tau^{1/3}).$$

Then, we can optimize by setting $\frac{D^2}{\tau^2} = D^{2/3}\tau^{1/3}$, i.e. $\tau = D^{4/7}$. The final running time is therefore $O^*(D^{6/7})$.

References

[1] Eric Bach and Jeffrey Shallit. *Algorithmic Number Theory, Volume I: Efficient Algorithms*. MIT Press, 1996.

[2] Alexander I. Barvinok. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *Mathematics of Operations Research*, 19(4):769–779, 1994.

[3] M. Beck and S. Robins. Explicit and efficient formulas for the lattice point count in rational polygons using Dedekind–Rademacher sums. *Discrete and Computational Geometry*, 27(4):443–459, 2002.

[4] Florin P. Boca, Cristian Cobeli, and Alexandru Zaharescu. Distribution of lattice points visible from the origin. *Communications in Mathematical Physics*, 213(2):433–470, 2000.

[5] Marc Deléglise and Joël Rivat. Computing the summation of the Möbius function. *Experimental Mathematics*, 5(4):291–295, 1996.

[6] Michal Forišek. Approximating rational numbers by fractions. In *Proc. 4th International Conference on Fun with Algorithms*, pages 156–165, 2007.

[7] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, 2nd edition, 1994.

[8] Doug Hensley. The number of lattice points within a contour and visible from the origin. *Pacific Journal of Mathematics*, 166(2):295–304, 1994.

[9] Martin N. Huxley and Werner Georg Nowak. Primitive lattice points in convex planar domains. *Acta Arithmetica*, 76(3):271–283, 1996.

[10] Ekkehard Krätzel and Werner Georg Nowak. Primitive lattice points in a thin strip along the boundary of a convex planar domain. *Acta Arithmetica*, 99:331–341, 2001.

[11] Stephen Kwek and Kurt Mehlhorn. Optimal search for rationals. *Information Processing Letters*, 86(1):23–26, 2003.

[12] B. Z. Moroz. On the number of primitive lattice points in plane domains. *Journal Monatshefte für Mathematik*, 99(1):37–42, 1985.

[13] Wolfgang Müller. Lattice points in convex planar domains: Power moments with an application to primitive lattice points. In *Proc. Conference on Analytic and Elementary Number Theory, European Congress on Mathematics*, pages 189–199, 1996.

[14] Werner Georg Nowak. Primitive lattice points in rational ellipses and related arithmetic functions. *Journal Monatshefte für Mathematik*, 106(1):57–63, 1988.

[15] Werner Georg Nowak. Sums and differences of two relative prime cubes II. In *Proc. Czech and Slovake Number Theory Conference*, 1995.

[16] Werner Georg Nowak. Primitive lattice points in starlike planar sets. *Pacific Journal of Mathematics*, 179(1):163–178, 1997.

[17] Werner Georg Nowak. Primitive lattice points inside an ellipse. *Czechoslovak Mathematical Journal*, 55(2):519–530, 2005.

[18] Christos Papadimitriou. Efficient search for rationals. *Information Processing Letters*, 8(1):1–4, 1979.

[19] Jakub Pawlewicz. Order statistics in the Farey sequences in sublinear time. In *Proceedings of 15th European Symposium on Algorithms (ESA 2007)*, volume 4698 of *LNCS*, pages 218–229. Springer, 2007.

[20] Corina E. Pătraşcu and Mihai Pătraşcu. Computing order statistics in the Farey sequence. In *Algorithmic Number Theory*, volume 3076 of *LNCS*, pages 358–366. Springer, 2004.

[21] Mihai Pătraşcu. Farey statistics in time $O(n^{2/3})$ and counting primitive lattice points in polygons. *ArXiv e-prints 0708.0080*, 2007.

[22] Steven P. Reiss. Efficient search for rationals. *Information Processing Letters*, 8(2):89–90, 1979.

[23] Jie Wu. On the primitive circle problem. *Journal Monatshefte für Mathematik*, 135(1):69–81, 2002.

[24] Hiroki Yanagisawa. A simple algorithm for lattice point counting in rational polygons. Research report, IBM Research, Tokyo Research Laboratory, 2005.

[25] Wenguang Zhai. On primitive lattice points in planar domains. *Acta Arithmetica*, 109(1):1–26, 2003.

[26] Wenguang Zhai and Xiaodong Cao. On the number of coprime integer pairs within a circle. *Acta Arithmetica*, 90(1):1–16, 1999.

A Counting Lattice Points in a Class of Triangles

In this section we present a simple algorithm for computing the sum $\sum_{i=1}^{n} \lfloor \frac{a}{b} i \rfloor$, where $\frac{a}{b}$ is a non-negative irreducible fraction. The value of the sum is the number of lattice points in a triangle bounded by the x-axis and lines $x = n$ and $y = \frac{a}{b}x$, excluding lattice points on the x-axis. Refer to Figure 2.

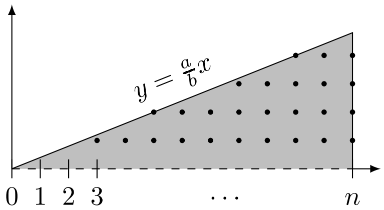

Figure 2: The sum $\sum_{i=1}^{n} \lfloor \frac{a}{b} i \rfloor$ counts lattice points in a right triangle.

We remark that [2] and [3] have previously given polynomial (in the bit precision) algorithms for counting lattice points in any polygon with rational coordinates. However, the algorithms in these papers do not render themselves to easy implementation. A much simpler algorithm for counting lattice points in a rational right triangle was presented in [24]. Here, we give a similar algorithm, which is a simplification of [24] for our special case of a rational right triangles.

Let us denote

$$T(n, a, b) = \sum_{i=1}^{n} \left\lfloor \frac{a}{b} i \right\rfloor.$$

We develop recursive formulas for $T(n, a, b)$, analyzing several possible cases.

The case $n \geq b$. If n is divisible by b, we can derive a closed form for the answer. Let $n = qb$, and refer to Figure 3. Observe that the lower right triangle is identical to the upper left triangle, so it contains the same

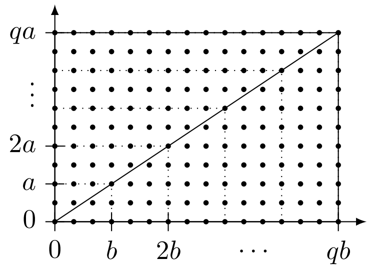

Figure 3: The case when n is divisible by b.

number of lattice points. Summing up both triangles we get the rectangle with diagonal counted twice. The number of lattice points in the rectangle is $(qb + 1)(qa + 1)$, and the number of points on the diagonal is equal to $q + 1$. The sum of both values divided by two gives the number of lattice points in a triangle. Now, we subtract the number of lattice points on the x-axis, obtaining:

$$T(qb, a, b) = \frac{(qa + 1)(qb + 1) + q + 1}{2} - (qb + 1) = \frac{q(qab - b + a + 1)}{2}. \tag{5}$$

More generally, suppose $n \geq b$ and let $n = qb + r$, where $q \geq 1$ and $0 \leq r < b$. The sum can be split into three parts:

$$\begin{aligned} \sum_{i=1}^{qb+r} \left\lfloor \frac{a}{b} i \right\rfloor &= \sum_{i=1}^{qb} \left\lfloor \frac{a}{b} i \right\rfloor + \sum_{i=qb+1}^{qb+r} \left\lfloor \frac{a}{b} i \right\rfloor = \sum_{i=1}^{qb} \left\lfloor \frac{a}{b} i \right\rfloor + \sum_{i=1}^{r} \left\lfloor \frac{a}{b} (qb + i) \right\rfloor \\ &= \sum_{i=1}^{qb} \left\lfloor \frac{a}{b} i \right\rfloor + r \cdot aq + \sum_{i=1}^{r} \left\lfloor \frac{a}{b} i \right\rfloor. \end{aligned}$$

See Figure 4 for intuition. As a result we obtain the equation:

$$T(qb + r, a, b) = T(qb, a, b) + rqa + T(r, a, b). \tag{6}$$

Combining the above formula with (5), we can reduce n to $n \bmod b$ in a single step. Therefore, in the remainder we assume that $n < b$. Observe that a consequence of this is that there is no integral point on the line $y = \frac{a}{b}x$ for $x = 1, 2, \ldots, n$.

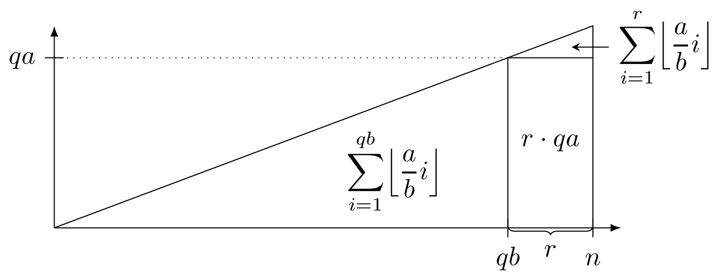

Figure 4: Case $n \geq b$.

The case $a \geq b$. If $a = qb + r$ for some $q \geq 1$ and $0 \leq r < b$, we can rewrite:

$$\sum_{i=1}^{n}\left\lfloor\frac{a}{b}i\right\rfloor = \sum_{i=1}^{n}\left\lfloor\frac{qb+r}{b}i\right\rfloor = \sum_{i=1}^{n} qi + \sum_{i=1}^{n}\left\lfloor\frac{r}{b}i\right\rfloor = q\frac{n(n+1)}{2} + \sum_{i=1}^{n}\left\lfloor\frac{r}{b}i\right\rfloor.$$

Thus, in this case we have the formula:

$$T(n, qb + r, b) = \frac{n(n+1)}{2}q + T(n, r, b). \tag{7}$$

Inverting $\frac{a}{b}$. We now use the graphical representation of Figure 5 to correlate sums $\sum\lfloor\frac{a}{b}i\rfloor$ and $\sum\lfloor\frac{b}{a}i\rfloor$. In Figure 5, the area labeled S_1 represents the sum $\sum_{i=1}^{n}\lfloor\frac{a}{b}i\rfloor$. The x and y coordinates of lattice points in this

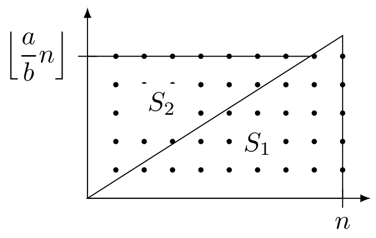

Figure 5: Graphical representation of sums $\sum\lfloor\frac{a}{b}i\rfloor$ and $\sum\lfloor\frac{b}{a}i\rfloor$.

area do not exceed n, and $\lfloor\frac{a}{b}n\rfloor$ respectively. Consider the rectangular set $R = \{1, \ldots, n\} \times \{1, \ldots, \lfloor\frac{a}{b}n\rfloor\}$, containing $n\lfloor\frac{a}{b}n\rfloor$ lattice points. Let S_2 be complement of S_1 in R. We assumed that $n < b$ and there is no element in R lying on the line $y = \frac{a}{b}x$. Therefore, for given $j = 1, \ldots, \lfloor\frac{a}{b}n\rfloor$, the number of lattice points from S_2 with y coordinate equal to j will be $\lfloor\frac{b}{a}j\rfloor$. Hence, the size of S_2 is $\sum_{j=1}^{\lfloor\frac{a}{b}n\rfloor}\lfloor\frac{b}{a}j\rfloor$. Since $|S_1| + |S_2| = |R|$, we have

$$\sum_{i=1}^{n}\left\lfloor\frac{a}{b}i\right\rfloor + \sum_{j=1}^{\lfloor\frac{a}{b}n\rfloor}\left\lfloor\frac{b}{a}j\right\rfloor = n\left\lfloor\frac{a}{b}n\right\rfloor.$$

Thus, the last recursive formula is:

$$T(n, a, b) = n\left\lfloor\frac{a}{b}n\right\rfloor - T\left(\left\lfloor\frac{a}{b}n\right\rfloor, b, a\right). \tag{8}$$

It allows to swap a with b in $T(\cdot, a, b)$. It can be used to make $a \geq b$. Notice that after swapping a with b, our assumption that $n < b$ holds, since if $n < b$, then $\frac{a}{b}n < a$ and $\lfloor\frac{a}{b}n\rfloor < a$.

The final algorithm. We combine the recursive formulas for $T(n, a, b)$ to design the final algorithm. The procedure is similar to the Euclidean algorithm. First, if $n \geq b$, we reduce n using (6) making $n < b$, then repeat the following steps until either n or a reach zero. If $a < b$, use (8) to exchange a with b. Next, use (7) to reduce a to $a \bmod b$. The total number of steps in this procedure is $O(\log \max\{n, a, b\})$, as it is in the Euclidean algorithm.