Polynomial time algorithms in algebraic number theory

Written by
D.M.H. van Gent

Based on lectures of
H.W. Lenstra

Mathematisch Instituut
Universiteit Leiden
Netherlands
20th January, 2020

1 Introduction

[TODO]

1.1 Algorithms

To be able to talk about polynomial-time algorithms, we should first define what an algorithm is. We equip the natural numbers $\mathbb{N} = \mathbb{Z}_{\geq 0}$ with a length function $l : \mathbb{N} \to \mathbb{N}$ that sends n to the number of digits of n in base 2, with $l(0) = 1$.

Definition 1.1. A *problem* is a function $f : I \to \mathbb{N}$ for some set of inputs $I \subseteq \mathbb{N}$ and we call f a *decision problem* if $f(I) \subseteq \{0, 1\}$. An *algorithm* for a problem $f : I \to \mathbb{N}$ is a 'method' to compute $f(x)$ for all $x \in I$. This algorithm is said to run in *polynomial time* if there exist $c_0, c_1, c_1 \in \mathbb{R}_{>0}$ such that for all $x \in I$ the time required to compute $f(x)$, called the *run-time*, is at most $(c_0 + c_1 l(x))^{c_2}$. We say a problem f is *computable* if there exists an algorithm for f.

This definition is rather empty: we have not specified what a 'method' is, nor have we explained how to measure run-time. We will briefly treat this more formally. The reader for which the above definition is sufficient can freely skip the following paragraph. The main conclusion is that we will not heavily rely on the formal definition of run-time in these notes.

In these notes, the word algorithm will be synonymous with the word *Turing machine*. A Turing machine is a model of computation described by Alan Turing in 1936 that defines an abstract machine which we these days think of as a computer. The main differences between a Turing machine and a modern day computer is that the memory of a Turing machine is a tape as opposed to random-access memory, and that a Turing machine has infinite memory. The run-time of a Turing machine is then measured as the number of elementary tape operations: reading a symbol on the tape, writing a symbol on the tape and moving the tape one symbol forward or backward. It is then immediately clear that it is expensive for a Turing machine to move the tape around much to look up data, as opposed to the random-access memory model where the cost of a memory lookup is constant, regardless of where the data is stored in memory. This also poses a problem for our formal treatment of run-time, as it may depend on our model of computation. However, both models of computation are able to emulate eachother in such a way that it preserves the property of computability in polynomial time, even though the constants c_0, c_1 and c_2 as in Definition 1.1 may increase drastically. We use this as an excuse to be informal in these notes about determining the run-time of an algorithm.

1.2 Basic computations

In these notes we build up our algorithms from basic building blocks. First and foremost, we remark that the basic operations in $\mathbb{Z}$ and $\mathbb{Q}$ are fast. Addition, subtraction, multiplication and division (with remainder in the case of $\mathbb{Z}$) can be done in polynomial time, as well as checking the sign of a number and whether numbers are equal. We assume here that we represent our rational numbers by a pairs of integers, a numerator and a denominator. We may even assume the numerator and denominator are coprime: Given $a, b \in \mathbb{Z}$ we can compute their greatest common divisor $\gcd(a, b)$ and solve the Bézout equation $ax + by = \gcd(a, b)$ for some $x, y \in \mathbb{Z}$ using the (extended) Euclidean algorithm in polynomial time. Applying these techniques in bulk we can also do addition, subtraction and multiplication of integer and rational matrices in polynomial time. Least trivially of our building blocks, using the theory of lattices we can compute bases for the kernel and the image of an integer matrix in polynomial time.

1.3 Number theory

Definition 1.2. A *number field* is a field K containing the field of rational numbers $\mathbb{Q}$ such that the dimension of K over $\mathbb{Q}$ as vector space is finite. A *number ring* is a subring of a number field.

For a number ring R, write $R_{\mathfrak{p}}$ for the localization at the prime ideal $\mathfrak{p}$ of R.

Definition 1.3. Let R be a number ring and let $K = R_{(0)}$ be its field of fractions. A *fractional ideal* of R is a non-zero R-submodule of K. A fractional ideal of R is *integral* if it is contained in R. A fractional ideal of R is *principal* if it is of the form xR for some $x \in K$. A fractional ideal I of R is *invertible* if there exists some fractional ideal J of R such that $I \cdot J$ is principal. For fractional ideals I and J of R write $I : J = \{x \in K \mid xJ \subseteq I\}$.

Definition 1.4. An *order* is a commutating ring whose additive group is isomorphic to $\mathbb{Z}^n$ for some n. Unless otherwise stated our orders are *reduced*, meaning that $x^k = 0$ for some x in our order and $k \in \mathbb{Z}$ implies $x = 0$.

That an order is reduced implies that it is contained in some finite product of number fields. In our algorithms we encode an order by first specifying its rank n, and then writing down its $n \times n$ multiplication table for the standard basis vectors. By distributivity this completely and unique defines a multiplication on the order. We do the same for number fields, as their additive group is isomorphic to $\mathbb{Q}^n$ for some n.

1.4 Exercises

Exercise 1.1. Let K be a number field.
a. Show that there are precisely $\#\mathbb{N}$ number fields up to isomorphism.
b. Show that there are precisely $\#\mathbb{N}$ orders and $\#\mathbb{R}$ subrings in K with field of fractions K.
c. Argue why this justifies restricting our algorithms to only take orders and number fields as input as opposed to number rings.

2 Coprime base factorization

In this section we treat the following problem, which will be the motivation for the *coprime base algorithm*.

Theorem 2.1. *There is a polynomial time algorithm that on input $t \in \mathbb{N}$, $q_1, \dots, q_t \in \mathbb{Q}^*$ and $n_1, \dots, n_t \in \mathbb{Z}$ decides whether*

$$\prod_{i=1}^{t} q_i^{n_i} = 1. \tag{2.1}$$

It is clear we can determine whether such a product has the correct sign: Simply take the sum of all n_i for which $q_i < 0$ and check whether the result is even. It is then sufficient to prove the following theorem instead.

Theorem 2.2. *There is a polynomial time algorithm that on input $t \in \mathbb{N}$, $a_1, \dots, a_t, b_1, \dots, b_t \in \mathbb{Z}_{>0}$ and $n_1, \dots, n_t, m_1, \dots, m_t \in \mathbb{Z}$ decides whether*

$$\prod_{i=1}^{t} a_i^{n_i} = \prod_{i=1}^{t} b_i^{m_i}. \tag{2.2}$$

In this form, the problem looks deceptively easy. Consider for example the most straightforward method to decide (2.2).

Method 2.3. Compute $\prod_{i=1}^{t} a_i^{n_i}$ and $\prod_{i=1}^{t} b_i^{m_i}$ explicitly and compare the results.

This method is certainly correct in that it is able to decide (2.2). However, it fails to run in polynomial time even when $t = 1$. For $n \in \mathbb{Z}_{>0}$ we have that $l(2^n) = n + 1 \approx 2^{l(n)}$. Hence the length of 2^n is not bounded by any polynomial in $l(n)$. We wouldn't even have enough time to write down the number regardless of our proficiency in multiplication because the number is too long.

Another method uses the Fundamental Theorem of Arithmetic, also known as unique prime factorization in $\mathbb{Z}$.

Method 2.4. Factor $a_1, \dots, a_t, b_1, \dots, b_t$ into primes and for each prime that occurs compute the number of times it occurs in the products $\prod_{i=1}^{t} a_i^{n_i}$ and $\prod_{i=1}^{t} b_i^{m_i}$ and compare the results.

It is true that once we have factored all integers into primes only a polynomial number of steps remains. If we write x_{ip} for the exponent of the prime p in a_i, then we may compute $\sum_{i=1}^{t} n_i x_{ip}$, the exponent of p in $\prod_{i=1}^{t} a_i^{n_i}$, in polynomial time. Moreover, the number of prime factors of $n \in \mathbb{Z}_{>0}$ is at most $l(n)$, so the number of primes occuring is at most $\sum_{i=0}^{t}(l(a_i) + l(b_i))$, which is less than the length of the combined input. The problem lies in the fact that we have not specified how to factor integers into primes. As of January 2020, nobody has been able to show that we can factor integers in polynomial time. Until this great open problem is solved, Method 2.4 is out the window.

An interesting observation is that the main obstruction in Method 2.3 lies in the exponents being large, while for Method 2.4 the obstruction is in the bases. Our proof for Theorem 2.2 will be to slightly tweak Method 2.4. Namely, observe that we do not need to factor into prime elements but that it sufficies to factor into pairwise coprime elements. The following lemma follows readily from unique prime factorization.

Lemma 2.5 (Unique coprime factorization)**.** *Let $s \in \mathbb{N}$ and let $c_1, \dots, c_s \in \mathbb{Z}_{>0}$ be pairwise coprime. If for $n_1, \dots, n_s, m_1, \dots, m_s \in \mathbb{Z}_{\geq 0}$ we have*

$$\prod_{i=1}^{s} c_i^{n_i} = \prod_{i=1}^{s} c_i^{m_i}, \tag{2.3}$$

then $n_i = m_i$ for all i. □

We now propose the following algorithm for deciding (2.2).

Method 2.6. Factor $a_1, \ldots, a_t, b_1, \ldots, b_t$ into pairwise coprime $c_1, \ldots, c_s \in \mathbb{Z}_{>0}$. For each c_i compute the number of times it occurs in $\prod_{i=1}^{t} a_i^{n_i}$ and $\prod_{i=1}^{t} b_i^{m_i}$ and compare the results.

Now to prove Theorem 2.2 and in turn Theorem 2.1 it suffices to prove the following.

Theorem 2.7 (Coprime base factorization)**.** *There is a polynomial time algorithm that on input $t \in \mathbb{N}$ and $a_1, \ldots, a_t \in \mathbb{Z}_{>0}$ computes $s \in \mathbb{N}$, $c_1, \ldots, c_s \in \mathbb{Z}_{>0}$ and $(n_{ij}) \in \mathbb{Z}_{\geq 0}^{t \times s}$ such that $c_1, \ldots, c_s$ are pairwise coprime and $a_i = \prod_{j=1}^{s} c_j^{n_{ij}}$ for all i.*

We state the algorithm first and prove the theorem later.

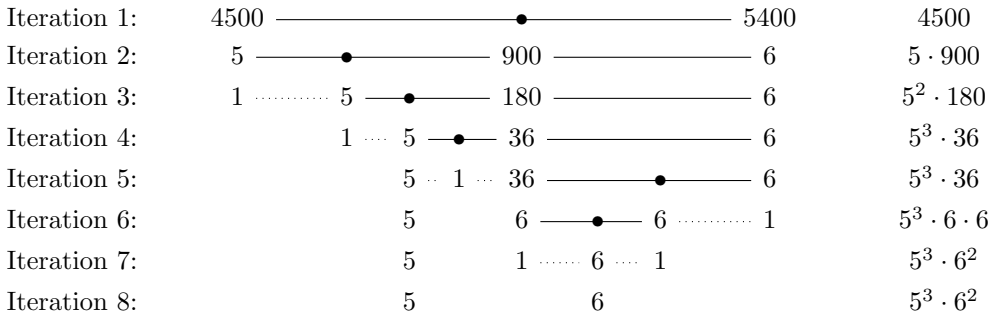
Method 2.8. Construct a complete simple graph G and label the vertices with $a_1, \ldots, a_s$. We call it a labeling because the map sending a vertex to its label need not be injective. While there are edges in G, repeat the following 5 steps:

1. Choose an edge $\{U, V\}$ of G and let u and v be the labels of U respectively V.
2. Compute $w = \gcd(u, v)$ using the Euclidean algorithm.
3. Add a vertex W labeled w to G and connect it to U, V and those vertices which are neighbours of both U and V.
4. Update the labels of U and V to u/w and v/w respectively.
5. For each $S \in \{U, V, W\}$, if the label of S is 1, then delete S and its incident edges from G.

Now $V = \{c_1, \ldots, c_s\}$ consists of the required pairwise coprime elements. The remaining output can now be computed in polynomial time.

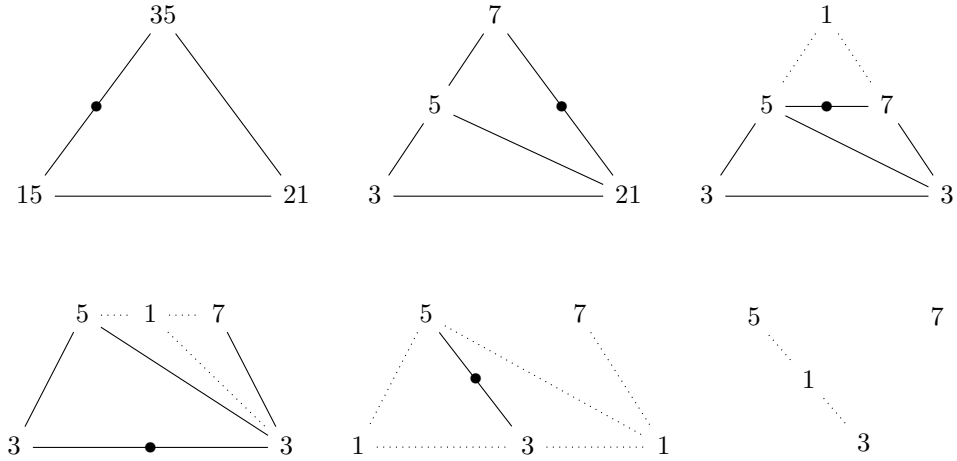
In the graph we construct and update the edges represent the pairs of numbers of which we do not yet know whether they are coprime, while a missing edge denotes that we know the pair to be coprime.

Example 2.9. We apply Method 2.8 to $(a_1, a_2) = (4500, 5400)$. Since there are only two vertices our graphs will fit on a single line. We denote the edge we choose in each iteration with a bullet and edges we have to erase are dotted. On the right we show how to keep track of the factorization of 4500 with minimal bookkeeping by writing it as a product of vertices in the graph.



We obtain $(c_1, c_2) = (5, 6)$ and $4500 = 5^3 \cdot 6^2$. By trial division we obtain $5400 = 5^2 \cdot 6^3$.

Example 2.10. We apply Method 2.8 to $(a_1, a_2, a_3) = (15, 21, 35)$.



The resulting coprime base is $(c_1, c_2, c_3) = (3, 5, 7)$. In the fifth graph something interesting happens. Vertex 7 suddenly becomes disconnected from the graph because we know it is coprime to one of the 3's.

Proof of Theorem 2.7. We claim Method 2.8 is correct and runs in polynomial time. One can show inductively that throughout the algorithm two vertices in the graph G are coprime when there is no edge between them. When the algorithm terminates because there are no edges in the graph, we may conclude that $c_1, \ldots, c_s$ are coprime. Additionally, one shows inductively that the numbers $a_1, \ldots, a_t$ can be written as some product of the vertices of G. Hence $c_1, \ldots, c_s$ forms a coprime base for $a_1, \ldots, a_t$, so Method 2.8 is correct. It remains to show that it is fast.

Write P_n for the product of all vertices in the graph at step n. Note that $P_0 = a_1 \cdots a_t$ and that $P_{n+1} \mid P_n$ for all $n \in \mathbb{N}$. Since P_0 has at most $B = l(P_0)$ prime factors counting multiplicities, there are at most B steps n for which $P_{n+1} < P_n$ and at most B vertices in G. The steps for which $P_n = P_{n+1}$ are those where the edge we chose is between coprime integers, meaning no vertices or edges are added to the graph and one edge is deleted. As the number of edges is at most B^2, then so is the number of consecutive steps for which $P_n = P_{n+1}$. Hence the total number of steps is at most B^3, which is polynomial in the length of the input. Lastly, note that each step takes only polynomial time because the values of the vertices are bounded from above by B and the Euclidean algorithm runs in polynomial time. Hence Method 2.8 runs in polynomial time. □

The speed of this algorithm heavily depends on how fast the product of all vertices decreases. In each step we want to choose our edge $\{u, v\}$ such that $\gcd(u, v) \gg 1$. A heuristic for this would be to choose edges between large numbers.

2.1 Coprime base factorization in number fields

We would like to generalize Theorem 2.1 to arbitrary number fields, by which we mean that there is an additional input K, a number field, and that we take $q_1, \ldots, q_t \in K^*$. When we try to do this, we run into some classic problems in (computational) number theory.

Theorem 2.2, to which we reduce, is a statement about integers. Hence we replace $\mathbb{Z}$ by an order R in K. But for our reduction to Theorem 2.2 to generalize we need to be able to compute R^*, which is considered difficult. Even if we disregard run-time issues, we need a Lemma 2.5 for orders. However, generally R will not be a UFD like $\mathbb{Z}$, making Theorem 2.7 hard to generalize. The 'correct' way to generalize the theory is to translate it into a theorem about ideals, since the maximal order $\mathcal{O}_K$ of K has unique ideal factorization. However, computing $\mathcal{O}_K$ is also difficult. We do have the following.

Lemma 2.11 (Unique coprime factorization for ideals)**.** *Let $s \in \mathbb{N}$, R an order and let $\mathfrak{c}_1, \ldots, \mathfrak{c}_s \subseteq R$ be pairwise coprime non-trivial invertible integral ideals. If for $n_1, \ldots, n_s, m_1, \ldots, m_s \in \mathbb{Z}_{\geq 0}$ we have*

$$\prod_{i=1}^{s} \mathfrak{c}_i^{n_i} = \prod_{i=1}^{s} \mathfrak{c}_i^{m_i}, \tag{2.4}$$

then $n_i = m_i$ for all i.

Proof. Since the ideals are invertible we may divide out $\mathfrak{c}_i^{\min\{n_i, m_i\}}$ and thus assume without loss of generality that $n_i = 0$ or $m_i = 0$ for all i. But then the product on the left hand side of (2.4) and the product on the right hand side of (2.4) are coprime, so the products equal R. By the Chinese remainder theorem for ideals we get $1 = R/(\prod_{i=1}^{s} \mathfrak{c}_i^{n_i}) = \prod_{i=1}^{s} (R/\mathfrak{c}_i^{n_i})$, so $\mathfrak{c}_i^{n_i} = R$ for all i. If $n_i > 0$ we have $\mathfrak{c}_i \supseteq \mathfrak{c}_i^{n_i} = R$, so $\mathfrak{c}_i = R$, a contradiction. Thus $n_i = m_i = 0$ for all i. □

2.2 Exercises

Exercise 2.1. Suppose we can compute a basis of the image and kernel of a morphism $\mathbb{Z}^n \to \mathbb{Z}^m$ in polynomial time. Show that for an order R and fractional ideals $\mathfrak{a}$ and $\mathfrak{b}$ of R we may compute $\mathfrak{a} + \mathfrak{b}$, $\mathfrak{a} \cdot \mathfrak{b}$ and $\mathfrak{a} : \mathfrak{b}$ in polynomial time.