1 The LLL Algorithm

Recall the definition of an LLL-reduced lattice basis.

Definition 1.1. A lattice basis $\mathbf{B}$ is *LLL-reduced* if the following two conditions are met:

1. For every $i < j$, we have $|\mu_{i,j}| \leq \frac{1}{2}$. (Such a basis is said to be "sized reduced.")
2. For every $1 \leq i < n$, we have $\frac{3}{4}\|\widetilde{\mathbf{b}}_i\|^2 \leq \|\mu_{i,i+1}\widetilde{\mathbf{b}}_i + \widetilde{\mathbf{b}}_{i+1}\|^2$. (This is the "Lovász condition.")

The LLL algorithm works as follows: given an *integral* input basis $\mathbf{B} \in \mathbb{Z}^{n\times n}$ (the integrality condition is without loss of generality), do the following:

1. Compute $\widetilde{\mathbf{B}}$, the Gram-Schmidt orthogonalized vectors of $\mathbf{B}$.
2. Let $\mathbf{B} \leftarrow \mathsf{SizeReduce}(\mathbf{B})$.

   (This algorithm, defined below, ensures that the basis is size reduced, and does not change $\mathcal{L}(\mathbf{B})$ or $\widetilde{\mathbf{B}}$.)
3. If there exists $1 \leq i < n$ for which the Lovász condition is violated, i.e., $\frac{3}{4}\|\widetilde{\mathbf{b}}_i\|^2 > \|\mu_{i,i+1}\widetilde{\mathbf{b}}_i + \widetilde{\mathbf{b}}_{i+1}\|^2$, then swap $\mathbf{b}_i$ and $\mathbf{b}_{i+1}$ and go back to Step 1. Otherwise, output $\mathbf{B}$.

The idea behind the $\mathsf{SizeReduce}(\mathbf{B})$ subroutine is, in the Gram-Schmidt decomposition $\mathbf{B} = \widetilde{\mathbf{B}} \cdot \mathbf{U}$, to shift the entries in the upper triangle of $\mathbf{U}$ by integers (via unimodular transformations), so that they lie in $[-\frac{1}{2}, \frac{1}{2})$. Because changing an entry of $\mathbf{U}$ may affect the ones above it (but not below it) in the same column, we must make the changes upward in each column. Formally, the algorithm works as follows:

- For each $j = 2, \ldots, n$ (in any order) and $i = j - 1$ *down to* 1, let $\mathbf{b}_j \leftarrow \mathbf{b}_j - \lfloor \mu_{i,j} \rceil \cdot \mathbf{b}_i$, where $\mu_{i,j} = \langle \mathbf{b}_j, \widetilde{\mathbf{b}}_i \rangle / \langle \widetilde{\mathbf{b}}_i, \widetilde{\mathbf{b}}_i \rangle$ is the (i,j)th entry of the upper-unitriangular matrix in the Gram-Schmidt decomposition of the *current* basis $\mathbf{B}$. (Note that previous iterations can change this matrix.)

  In matrix form, in the (i,j)th iteration we are letting $\mathbf{B} \leftarrow \mathbf{B} \cdot \mathbf{W}$, where $\mathbf{W}$ is the upper unitriangular matrix with just one potentially nonzero off-diagonal entry $-\lfloor \mu_{i,j} \rceil$, at position (i,j).

We make a few important observations about the SizeReduce algorithm. First, it clearly runs in time polynomial in the bit length of the input basis $\mathbf{B}$. Second, even though $\mathbf{B}$ may change, the Gram-Schmidt vectors $\widetilde{\mathbf{B}}$ are preserved throughout, because the only changes to $\mathbf{B}$ are via multiplication by upper-unitriangular matrices, i.e., if $\mathbf{B} = \widetilde{\mathbf{B}} \cdot \mathbf{U}$ is the Gram-Schmidt decomposition prior to some iteration, then $\mathbf{B} = \widetilde{\mathbf{B}} \cdot (\mathbf{UW})$ is the decomposition afterward, since $\mathbf{UW}$ is upper unitriangular. Finally, the (i,j)th iteration ensures that the value $\langle \mathbf{b}_j, \widetilde{\mathbf{b}}_i \rangle / \langle \widetilde{\mathbf{b}}_i, \widetilde{\mathbf{b}}_i \rangle \in [-\frac{1}{2}, \frac{1}{2})$ (by definition of $\mu_{i,j}$), and following the iteration, that value never changes, because $\mathbf{b}_k$ is orthogonal to $\widetilde{\mathbf{b}}_i$ for all $k < i$. (This is why it important that we loop from $i = j - 1$ *down to* 1; $\mathbf{b}_k$ may not be orthogonal to $\widetilde{\mathbf{b}}_i$ for $k > i$.) Putting these observation together, we have the following lemma on the correctness of SizeReduce.

Lemma 1.2. *Given an integral basis $\mathbf{B} \in \mathbb{Z}^{n\times n}$ with Gram-Schmidt decomposition $\mathbf{B} = \widetilde{\mathbf{B}} \cdot \mathbf{U}$, the* SizeReduce *algorithm outputs a basis $\mathbf{B}'$ of $\mathcal{L} = \mathcal{L}(\mathbf{B})$ having Gram-Schmidt decomposition $\mathbf{B}' = \widetilde{\mathbf{B}} \cdot \mathbf{U}'$, where every entry $u'_{i,j}$ for $i < j$ is in $[-\frac{1}{2}, \frac{1}{2})$.*

We now state the main theorem about the LLL algorithm.

Theorem 1.3. *Given an integral basis $\mathbf{B} \in \mathbb{Z}^{n\times n}$, the LLL algorithm outputs an LLL-reduced basis of $\mathcal{L} = \mathcal{L}(\mathbf{B})$ in time $\mathrm{poly}(n, |\mathbf{B}|)$, where $|\mathbf{B}|$ denotes the bit length of the input basis.*

The remainder of this section is dedicated to an (almost complete) proof of this theorem. First, it is clear that the LLL algorithm, if it ever terminates, is correct: all the operations on the input basis preserve the lattice it generates, and the algorithms terminates only when the basis is LLL-reduced.

We next prove that the number of iterations is $O(N)$ for some $N = \mathrm{poly}(n, |\mathbf{B}|)$. This uses a clever "potential argument," which assigns a value to all the intermediate bases produced by the algorithm. We show three facts: that the potential starts out no larger than 2^N, that it never drops below 1, and that each iteration of the algorithm decreases the potential by a factor of at least $\sqrt{4/3} > 1$. This implies that the number of iterations is at most $\log_{\sqrt{4/3}} 2^N = O(N)$.

The potential function is defined as follows: for a basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$, let $\mathcal{L}_i = \mathcal{L}(\mathbf{b}_1, \ldots, \mathbf{b}_i)$ for each $1 \le i \le n$. The potential is the product of these lattices' determinants:

$$\Phi(\mathbf{B}) := \prod_{i=1}^{n} \det(\mathcal{L}_i) = \prod_{i=1}^{n} \left(\|\widetilde{\mathbf{b}}_1\| \cdots \|\widetilde{\mathbf{b}}_i\| \right) = \prod_{i=1}^{n} \|\widetilde{\mathbf{b}}_i\|^{n-i+1}.$$

Claim 1.4. *The potential of the initial input basis $\mathbf{B}$ is at most 2^N where $N = \mathrm{poly}(n, |\mathbf{B}|)$, and every intermediate basis the algorithm produces has potential at least 1.*

Proof. The potential of the original basis $\mathbf{B}$ is clearly bounded by $\prod_{i=1}^{n} \|\mathbf{b}_i\|^n \le \max_i \|\mathbf{b}_i\|^{n^2} = 2^{\mathrm{poly}(n, |\mathbf{B}|)}$. Every intermediate basis is integral and has positive integer determinant, hence so do the lattices $\mathcal{L}_i$ associated with that basis. Therefore, the potential of that basis is at least 1. □

We next analyze how the potential changes when we perform a swap in Step 3.

Claim 1.5. *Suppose $\mathbf{b}_i$ and $\mathbf{b}_{i+1}$ are swapped in Step 3, and let the resulting basis be denoted $\mathbf{B}'$. Then $\widetilde{\mathbf{b}}'_j = \widetilde{\mathbf{b}}_j$ for all $j \notin \{i, i+1\}$, and $\widetilde{\mathbf{b}}'_i = \mu_{i,i+1}\widetilde{\mathbf{b}}_i + \widetilde{\mathbf{b}}_{i+1}$.*

Proof. For $j < i$, the vector $\widetilde{\mathbf{b}}'_j$ is unaffected by the swap, because by definition it is the component of $\mathbf{b}'_j = \mathbf{b}_j$ orthogonal to $\mathrm{span}(\mathbf{b}'_1, \ldots, \mathbf{b}'_{j-1}) = \mathrm{span}(\mathbf{b}_1, \ldots, \mathbf{b}_{j-1})$. Similarly, for $j > i+1$, the vector $\widetilde{\mathbf{b}}'_j$ is the component of $\mathbf{b}'_j = \mathbf{b}_j$ orthogonal to $\mathrm{span}(\mathbf{b}'_1, \ldots, \mathbf{b}'_{j-1}) = \mathrm{span}(\mathbf{b}_1, \ldots, \mathbf{b}_{j-1})$, where the equality holds because both $\mathbf{b}_i$ and $\mathbf{b}_{i+1}$ are in the span. Finally, $\widetilde{\mathbf{b}}'_i$ is the component of $\mathbf{b}'_i = \mathbf{b}_{i+1}$ orthogonal to $\mathrm{span}(\mathbf{b}'_1, \ldots, \mathbf{b}'_{i-1}) = \mathrm{span}(\mathbf{b}_1, \ldots, \mathbf{b}_{i-1})$, which is $\mu_{i,i+1}\widetilde{\mathbf{b}}_i + \widetilde{\mathbf{b}}_{i+1}$ by construction. □

Lemma 1.6. *Suppose $\mathbf{b}_i$ and $\mathbf{b}_{i+1}$ are swapped in Step 3, and let the resulting basis be denoted $\mathbf{B}'$. Then $\Phi(\mathbf{B}')/\Phi(\mathbf{B}) < \sqrt{3/4}$.*

Proof. Let $\mathcal{L}_i = \mathcal{L}(\mathbf{b}_1, \ldots, \mathbf{b}_{i-1}, \mathbf{b}_i)$ and $\mathcal{L}'_i = \mathcal{L}(\mathbf{b}_1, \ldots, \mathbf{b}_{i-1}, \mathbf{b}_{i+1})$. By Claim 1.5, we have

$$\frac{\Phi(\mathbf{B}')}{\Phi(\mathbf{B})} = \frac{\det(\mathcal{L}'_i)}{\det(\mathcal{L}_i)} = \frac{\|\widetilde{\mathbf{b}}_1\| \cdots \|\widetilde{\mathbf{b}}_{i-1}\| \|\mu_{i,i+1}\widetilde{\mathbf{b}}_i + \widetilde{\mathbf{b}}_{i+1}\|}{\|\widetilde{\mathbf{b}}_1\| \cdots \|\widetilde{\mathbf{b}}_{i-1}\| \|\widetilde{\mathbf{b}}_i\|} = \frac{\|\mu_{i,i+1}\widetilde{\mathbf{b}}_i + \widetilde{\mathbf{b}}_{i+1}\|}{\|\widetilde{\mathbf{b}}_i\|} < \sqrt{3/4},$$

where the last inequality follows from the Lovász condition. □

This completes the proof that the number of iterations is $O(N) = \mathrm{poly}(n, |\mathbf{B}|)$. Moreover, each iteration of the algorithm is polynomial time in the bit length of the current basis. However, this does *not* necessarily guarantee that the LLL algorithm is polynomial time overall, since the bit length of the intermediate bases could *increase* with each iteration. (For example, if the bit length doubled in each iteration, then by the end

the bit length would be exponential in n.) To it suffices to show that the sizes of all intermediate bases are polynomial in the size of the original basis. This turns out to be the case, due to the size-reduction step. The proof of this fact is somewhat grungy and uninteresting, though, so we won't cover it.

We conclude with some final remarks about the LLL algorithm. The factor $3/4$ in the Lovász condition is just for convenience of analysis. We can use any constant between $1/4$ and 1, which yields a tradeoff between the final approximation factor and the number of iterations, but these will still remain exponential (in n) and polynomial, respectively. By choosing the factor very close to 1, we can obtain an approximation factor of $(2/\sqrt{3})^n$ in polynomial time, but we cannot do any better using LLL. We can get slightly better approximation factors of $2^{O(n(\log\log n)^2)/(\log n)}$ (still in polynomial time) using Schnorr's generalization of LLL, where the analogue of the Lovász condition deals with blocks of $k \geq 2$ consecutive vectors.

2 Coppersmith's Method

One nice application of LLL is a technique of Coppersmith that finds all *small* roots of a polynomial modulo a given number N (even when the factorization of N is unknown). This technique has been a very powerful tool in cryptanalysis, as we will see next time.

Theorem 2.1. *There is an efficient algorithm that, given any monic, degree-d integer polynomial $f(x) \in \mathbb{Z}[x]$ and an integer N, outputs all integers x_0 such that $|x_0| \leq B = N^{1/d}$ and $f(x_0) = 0 \bmod N$.*

We make a few important remarks about the various components of this theorem:

1. When N is prime, i.e., $\mathbb{Z}_N$ is a finite field, there are efficient algorithms that output *all* roots of a given degree-d polynomial $f(x)$ modulo N, of which there are at most d. Similarly, there are efficient algorithm that factor polynomials over the rationals (or integers). Therefore, the fact that the theorem handles a composite modulus N is a distinguishing feature.
2. For composite N, the number of roots of $f(x)$ modulo N can be nearly exponential in the bit length of N, even for quadratic $f(x)$. For example, if N is the product of k distinct primes, then any square modulo N has exactly 2^k distinct square roots. (This follows from the Chinese Remainder Theorem, since there are two square roots modulo each prime divisor of N.) Since k can be as large as $\approx \log N/\log\log N$, the number of roots can be nearly exponential in $\log N$. Therefore, in general no efficient algorithm can output *all* roots of $f(x)$ modulo N; the restriction to *small* roots in the theorem statement circumvents this problem.[1]
3. The size restriction appears necessary for another reason: knowing two square roots $r_1 \neq \pm r_2$ of a square modulo a composite N reveals a nontrivial factor of N, as $\gcd(r_1 - r_2, N)$. So even if the number of roots is small, finding them all is still at least as hard as factoring. However, it is easy to show that a square cannot have more than one "small" square root, of magnitude at most $N^{1/2}$. Therefore, the theorem does appear to yield an efficient factoring algorithm.[2]

To highlight the heart of the method, in the remainder of the section we prove the theorem for a weaker bound of $B \approx N^{2/(d(d+1))}$. (We prove the bound $B \approx N^{1/d}$ next time.) The strategy is to find another nonzero polynomial $h(x) = \sum h_i x^i \in \mathbb{Z}[x]$ such that:

[1] Indeed, the theorem implies that the number of small roots is always polynomially bounded. Surprisingly, this fact did not appear to be known before Coppersmith's result!

[2] However, it can be used to factor when some partial information about a factor is known.

1. every root of $f(x)$ modulo N is also a root of $h(x)$, and
2. the polynomial $h(Bx)$ is "short," i.e., $|h_i B^i| < N/(\deg(h)+1)$ for all i.

For any such $h(x)$, and for any x_0 such that $|x_0| \leq B$, we have $|h_i x_0^i| \leq |h_i B^i| < N/(\deg(h)+1)$, which implies that $|h(x_0)| < N$. Hence, for every *small* root x_0 (such that $|x_0| \leq B$) of $f(x)$ modulo N, we have that $h(x_0) = 0$ *over the integers* (not modulo anything). To find the small roots of $f(x)$ modulo N, we can therefore factor $h(x)$ over the integers, and test whether each of its (small) roots is a root of $f(x)$ modulo N.

We now give an efficient algorithm to find such an $h(x)$. The basic idea is that adding integer multiples of the polynomials $g_i(x) = Nx^i \in \mathbb{Z}[x]$ to $f(x)$ certainly preserves the roots of f modulo N. So we construct a lattice whose basis corresponds to the coefficient vectors of the polynomials $g_i(Bx)$ and $f(Bx)$, find a short nonzero vector in this lattice, and interpret it as the polynomial $h(Bx)$. The lattice basis is

$$\mathbf{B} = \begin{pmatrix} N & & & & a_0 \\ & BN & & & a_1 B \\ & & B^2 N & & a_2 B^2 \\ & & & & \\ & & & B^{d-1}N & a_{d-1}B^{d-1} \\ & & & & B^d \end{pmatrix}.$$

Note that the lattice dimension is $d+1$, and that $\det(\mathbf{B}) = B^{d(d+1)/2} \cdot N^d$. By running the LLL algorithm on this basis, we obtain a $2^{d/2}$-approximation $\mathbf{v}$ to a shortest vector in $\mathcal{L}(\mathbf{B})$. By Minkowski's bound,

$$\|\mathbf{v}\| \leq 2^{d/2}\sqrt{d+1} \cdot B^{d/2} \cdot N^{d/(d+1)} = c_d \cdot B^{d/2} \cdot N^{1-1/(d+1)},$$

where $c_d = 2^{d/2}\sqrt{d+1}$ depends only on the degree d.

Define $h(Bx)$ to be the polynomial whose coefficients are given by $\mathbf{v}$, i.e., $h(x) = v_0 + (v_1/B)x + \cdots + (v_d/B^d)x^d$. Notice that $h(x) \in \mathbb{Z}[X]$, because B^i divides v_i for each i by construction of the lattice basis, and that every root of $f(x)$ modulo N is also a root of $h(x)$ by construction. Finally, we see that

$$|h_i B^i| = |v_i| \leq \|v\| < \frac{N}{d+1},$$

if we take $B < N^{2/d(d+1)}/c'_d$ where $c'_d = (c_d(d+1))^{2/d} = O(1)$ is bounded by a small constant. This concludes the proof.