Lecture 3
LLL, Coppersmith

Instructor: Chris Peikert
Scribe: Yan Wang

1 The LLL Algorithm

Recall the definition of an LLL-reduced lattice basis, as defined by Lenstra, Lenstra, and Lovász [LLL82], and how it approximates a short nonzero lattice vector. (In all that follows, recall that $\mathbf{B} = \widetilde{\mathbf{B}} \cdot \mathbf{U}$ is the Gram–Schmidt decomposition of a lattice basis $\mathbf{B}$, where the columns of $\widetilde{\mathbf{B}}$ are orthogonal and $\mathbf{U} \in \mathbb{R}^{n \times n}$ is upper unitriangular, with entries $\mu_{i,j}$.)

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

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

Lemma 1.2. *In an LLL-reduced lattice basis $\mathbf{B} \in \mathbb{Z}^{n \times n}$, we have that $\|\widetilde{\mathbf{b}}_{i+1}\|^2 \geq \frac{1}{2}\|\widetilde{\mathbf{b}}_i\|^2$ for all $i < n$, and hence $\|\mathbf{b}_1\| \leq 2^{(n-1)/2} \min_i \|\widetilde{\mathbf{b}}_i\| \leq 2^{(n-1)/2} \cdot \lambda_1(\mathcal{L}(\mathbf{B}))$.*

Algorithm 1 defines the LLL algorithm, which transforms any lattice basis into an LLL-reduced one of the same lattice. The algorithm simply alternates between two steps: *size-reducing* the current basis (i.e., making Item 1 hold), and checking whether the Lovász condition (Item 2) holds; if not, it simply swaps a pair of offending basis vectors. From this description, we can immediately see that the output is indeed LLL-reduced, so the algorithm is correct—if it ever terminates. The key challenge is in showing that it does indeed terminate, and in time polynomial in the input size.

Algorithm 1 The LLL algorithm.

```
1: function LLL($\mathbf{B} \in \mathbb{Z}^{n \times n}$)        ▷ Outputs an LLL-reduced basis of the lattice $\mathcal{L}(\mathbf{B})$
2:     let $\mathbf{B} \leftarrow$ SizeReduce($\mathbf{B}$)
3:     if there exists an $i < n$ for which the Lovász condition is violated then
4:         swap $\mathbf{b}_i$ and $\mathbf{b}_{i+1}$ and go to Line 2
5:     return $\mathbf{B}$
6: function SizeReduce($\mathbf{B} \in \mathbb{Z}^{n \times n}$)        ▷ Outputs a size-reduced basis of $\mathcal{L}(\mathbf{B})$, preserving $\widetilde{\mathbf{B}}$
7:     compute (or update) the Gram–Schmidt orthogonalization $\widetilde{\mathbf{B}}$ of $\mathbf{B}$
8:     for $j = 2, \ldots, n$ (in any order) do
9:         for $i = j - 1$ down to 1 do
10:            let $\mathbf{b}_j \leftarrow \mathbf{b}_j - \lfloor u_{i,j} \rceil \cdot \mathbf{b}_i$, where $u_{i,j} = \langle \mathbf{b}_j, \widetilde{\mathbf{b}}_i \rangle / \langle \widetilde{\mathbf{b}}_i, \widetilde{\mathbf{b}}_i \rangle$
```

The SizeReduce($\mathbf{B}$) subroutine deserves some elaboration. Its purpose 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, so that they lie in $[-\frac{1}{2}, \frac{1}{2})$. Because we must preserve the lattice, this is done by subtracting appropriate integer multiples of the basis vectors $\mathbf{b}_i$ (*not* the Gram–Schmidt vectors $\widetilde{\mathbf{b}}_i$!) from the $\mathbf{b}_j$, for $i < j$. In matrix form, the (i, j)th iteration lets $\mathbf{B} \leftarrow \mathbf{B} \cdot \mathbf{W} = \widetilde{\mathbf{B}} \cdot (\mathbf{U}\mathbf{W})$, where $\mathbf{W}$ is the upper unitriangular matrix with just one possibly nonzero off-diagonal entry $-\lfloor u_{i,j} \rceil$, at position (i, j). This implicitly updates the (upper unitriangular) matrix $\mathbf{U} \leftarrow \mathbf{U}\mathbf{W} \in \mathbb{R}^{n \times n}$. This matrix is identical to $\mathbf{U}$, except possibly at position (i, j) *and the entries above it, but not below it* in the same column. This is why we need to make the changes going *upward* in each column (see Line 9).

Lemma 1.3. *Given an integral lattice basis $\mathbf{B} \in \mathbb{Z}^{n \times n}$ with Gram–Schmidt orthogonalization $\widetilde{\mathbf{B}}$, 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 $u'_{i,j} \in [-\frac{1}{2}, \frac{1}{2})$ for all $i < j$.*

Proof. First, even though SizeReduce may change $\mathbf{B}$, 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. Second, the (i,j)th iteration ensures that the value $\langle \mathbf{b}_j, \widetilde{\mathbf{b}}_i \rangle / \langle \widetilde{\mathbf{b}}_i, \widetilde{\mathbf{b}}_i \rangle \in [-\frac{1}{2}, \frac{1}{2})$, and that value never changes in later iterations, because for all $h < i$, the basis vector $\mathbf{b}_h$ (a multiple of which may later be subtracted from $\mathbf{b}_j$) is orthogonal to $\widetilde{\mathbf{b}}_i$. $\square$

We now state the main theorem about the LLL algorithm.

Theorem 1.4. *Given an integral lattice 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 $\operatorname{poly}(n, |\mathbf{B}|)$, 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 algorithm can only output an LLL-reduced basis.

We next prove that the number of iterations is $O(N)$ for some $N = \operatorname{poly}(n, |\mathbf{B}|)$. This uses a "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 $\sqrt{4/3} > 1$. All 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 $i \leq n$. The potential is the product of the determinants of these lattices:

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

Claim 1.5. *The potential of the input basis $\mathbf{B}$ is at most 2^N, and every intermediate basis the algorithm produces has potential at least 1.*

Proof. Because $\|\widetilde{\mathbf{b}}_i\| \leq \|\mathbf{b}_i\|$ and $\|\mathbf{b}_i\| \geq 1$ for all i, the potential of the input basis $\mathbf{B}$ is bounded by

$$\Phi(\mathbf{B}) \leq \prod_{i=1}^{n} \|\mathbf{b}_i\|^{n-i+1} \leq \prod_{i=1}^{n} \|\mathbf{b}_i\|^{n} \leq \max_i \|\mathbf{b}_i\|^{n^2} = 2^{\operatorname{poly}(n, |\mathbf{B}|)}.$$

Every intermediate basis is integral, hence the lattices $\mathcal{L}_i$ associated with that basis have determinant at least 1.[1] Therefore, the potential of that basis is at least 1. $\square$

We next analyze how the potential changes when we perform a swap in Line 4.

Claim 1.6. *Suppose that $\mathbf{b}_i$ and $\mathbf{b}_{i+1}$ are swapped in Line 4, 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 $\widetilde{\mathbf{b}}'_i = \mu_{i,i+1} \widetilde{\mathbf{b}}_i + \widetilde{\mathbf{b}}_{i+1}$.*

Note that this claim lets us optimize Line 7: following a swap, we need not orthogonalize $\mathbf{B}'$ from scratch, but only need to update two of the vectors from $\widetilde{\mathbf{B}}$.

[1] This requires some care to verify, because the lattices $\mathcal{L}_i$ are *not full rank* for $i < n$. Letting $\mathbf{B}_i = (\mathbf{b}_1, \ldots, \mathbf{b}_i) \in \mathbb{Z}^{n \times i}$ be a basis for $\mathcal{L}_i$, we have $\det(\mathcal{L}_i) = \sqrt{\det(\mathbf{B}_i^t \mathbf{B}_i)} \geq 1$, because the Gram matrix $\mathbf{B}_i^t \mathbf{B}_i \in \mathbb{Z}^{i \times i}$ is integral and nonsingular.

Proof. For all $j < i$, we have $\widetilde{\mathbf{b}}'_j = \widetilde{\mathbf{b}}_j$, because it is the component of $\mathbf{b}'_j = \mathbf{b}_j$ orthogonal to $\operatorname{span}(\mathbf{b}'_1, \ldots, \mathbf{b}'_{j-1}) = \operatorname{span}(\mathbf{b}_1, \ldots, \mathbf{b}_{j-1})$. Similarly, for any $j > i+1$, we have $\widetilde{\mathbf{b}}'_j = \widetilde{\mathbf{b}}_j$, because it is the component of $\mathbf{b}'_j = \mathbf{b}_j$ orthogonal to $\operatorname{span}(\mathbf{b}'_1, \ldots, \mathbf{b}'_{j-1}) = \operatorname{span}(\mathbf{b}_1, \ldots, \mathbf{b}_{j-1})$, where here the equality holds because both $\mathbf{b}_i$ and $\mathbf{b}_{i+1}$ are in the lists of arguments (in reversed order). Finally, $\widetilde{\mathbf{b}}'_i$ is the component of $\mathbf{b}'_i = \mathbf{b}_{i+1}$ orthogonal to $\operatorname{span}(\mathbf{b}'_1, \ldots, \mathbf{b}'_{i-1}) = \operatorname{span}(\mathbf{b}_1, \ldots, \mathbf{b}_{i-1})$, which is $\mu_{i,i+1}\widetilde{\mathbf{b}}_i + \widetilde{\mathbf{b}}_{i+1}$ by construction. □

Lemma 1.7. *Suppose that $\mathbf{b}_i$ and $\mathbf{b}_{i+1}$ are swapped in Line 4, 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 1.6,

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

where the last inequality follows from the fact that the swap occurs because the Lovász condition (Item 2) is *not* satisfied. □

This completes the proof that the number of iterations is $O(N) = \operatorname{poly}(n, |\mathbf{B}|)$. Moreover, we can see by inspection that 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 possibly *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 complete the full proof that LLL is polynomial time, it suffices to show that the sizes of *all* intermediate bases are some fixed polynomial in the size of the original basis. This turns out to be the case, specifically due to the size-reduction step (which we have not used up to this point!). The proof of this fact is somewhat grungy, 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 and polynomial (in n), respectively. By choosing a value 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 [Sch87] 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 [Cop96b] 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 a polynomial-time 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.[2]
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 all of them 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 not appear to yield an efficient factoring algorithm. (However, it can be used to factor when some partial information about a factor is known, as shown in the related work [Cop96a].)

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

1. 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 just 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 $g_i(x) = N x^i \in \mathbb{Z}[x]$ 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

$$\mathbf{B} = \begin{pmatrix} N & & & & f_0 \\ & BN & & & f_1 B \\ & & B^2 N & & f_2 B^2 \\ & & & \ddots & \vdots \\ & & & B^{d-1} N & f_{d-1} B^{d-1} \\ & & & & B^d \end{pmatrix}.$$

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

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 nonzero lattice vector $\mathbf{v} \in \mathcal{L}(\mathbf{B})$ for which

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

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 $(2B)^{d/2} < N^{1/(d+1)}/(d+1)$, which is equivalent to $B < N^{2/(d(d+1))}/(2(d+1)^{2/d})$; note that the denominator is upper bounded by a constant. This concludes the proof.

References

[Cop96a] D. Coppersmith. Finding a small root of a bivariate integer equation; factoring with high bits known. In *EUROCRYPT*, pages 178–189. 1996. Page 4.

[Cop96b] D. Coppersmith. Finding a small root of a univariate modular equation. In *EUROCRYPT*, pages 155–165. 1996. Page 3.

[LLL82] A. K. Lenstra, H. W. Lenstra, Jr., and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534, December 1982. Page 1.

[Sch87] C.-P. Schnorr. A hierarchy of polynomial time lattice basis reduction algorithms. *Theor. Comput. Sci.*, 53:201–224, 1987. Page 3.