The complexity of factoring univariate polynomials over the rationals

Mark van Hoeij
Florida State University

ISSAC'2013

June 26, 2013

Papers

- [Zassenhaus 1969]. Usually fast, but can be exp-time.
- [LLL 1982]. Lattice reduction (LLL algorithm).
- [LLL 1982]. First poly-time factoring algorithm.
- [Schönhage 1984] improved complexity to $\tilde{\mathcal{O}}(N^4(N+h)^2)$
- [vH 2002]. New algorithm, outperforms prior algorithms, but no complexity bound.
- [Belabas 2004] Gave the best-tuned version of [vH 2002].
- [Belabas, vH, Klüners, Steel 2004]. Poly-time bound for a slow version of [vH 2002], bad bound for a practical version.
- [vH and Novocin, 2007, 2008, 2010]. Asymptotically sharp bound $\mathcal{O}(r^3)$ for # LLL-swaps in a fastest version.
- [Hart, vH, Novocin], ISSAC'2011, implementation.

Progress in factoring, brief history

Factoring in practice:

year	performance
< 1969	really slow
1969	fast for most inputs
2002	fast for all inputs
2011	added early termination

Factoring in theory:

year	complexity
< 1982	exp-time
1982	poly-time
1984	$\tilde{\mathcal{O}}(N^4(N+h)^2)$
2011	$\tilde{\mathcal{O}}(r^6) + \mathrm{Pol}_{\mathrm{deg}<6}(N, h)$

Histories have little overlap!

Comparing factoring algorithms

Suppose $f \in \mathbb{Z}[x]$ has degree N and the largest coefficient has h digits. Suppose f is square-free mod p, and f factors as $f \equiv f_1 \cdots f_r \bmod p$. The algorithms from the previous slide do:

Step 1: Hensel lift so that $f \equiv f_1 \cdots f_r \bmod p^a$ for some a

- [Zassenhaus]: $\log(p^a) = \mathcal{O}(N + h)$.
- [LLL, Schönhage, BHKS]: $\log(p^a) = \mathcal{O}(N(N + h))$.
- [HHN, ISSAC'2011]: p^a is initially less than in [Zassenhaus], but might grow to:
  - $\log(p^a) = \tilde{\mathcal{O}}(N + h)$ (conjectured linear upper bound)
  - $\log(p^a) = \mathcal{O}(N(N + h))$ (proved quadratic upper bound)

Comparing factoring algorithms, continued

Step 2: (**combinatorial problem**): [Zassenhaus] checks all subsets of $\{f_1, \ldots, f_r\}$ with $d = 1, 2, \ldots, \lfloor r/2 \rfloor$ elements, to see if the product gives a "true" factor (i.e. a factor of f in $\mathbb{Q}[x]$).
If f is irreducible, then it checks 2^{r-1} cases.

Step 2: [LLL, Schönhage] **bypass the combinatorial problem** and compute

$$L := \{(a_0, \ldots, a_{N-1}) \in \mathbb{Z}^N, \ \sum a_i x^i \equiv 0 \mod (f_1, p^a)\}.$$

LLL reduce, take the first vector $(a_0, \ldots, a_{N-1})$, and compute $\gcd(f, \sum a_i x^i)$. This is a non-trivial factor iff f is reducible.

Step 2: [vH 2002], solves the **combinatorial problem** by constructing a lattice, for which LLL reduction produces those $v = (v_1, \ldots, v_r) \in \{0, 1\}^r$ for which $\prod f_i^{v_i}$ is a "true" factor.

Comparing factoring algorithms, an example

Suppose $f \in \mathbb{Z}[x]$ has degree $N = 1000$ and the largest coefficient has $h = 1000$ digits. Suppose f factors as $f \equiv f_1 \cdots f_r \bmod p$. Suppose $r = 50$. The algorithms do:

Step 1: [Zassenhaus] Hensel lifts to p^a having $\approx 10^3$ digits, while [LLL, Schönhage] lift to p^a having $\approx 10^6$ digits.

Step 2: [Zassenhaus] might be fast, but might also be slow: If f has a true factor consisting of a small subset of $\{f_1, \ldots, f_r\}$, then [Zassenhaus] quickly finds it. But if f is irreducible, then it will check 2^{r-1} cases.

Comparison of the algorithms, example, continued

Step 2: [LLL, Schönhage], will take a very long time because L has dimension 1000 and million-digit entries. This explains why these poly-time algorithms were not used in practice.

Step 2: [vH 2002], L has dimension $50 + \epsilon$ and small entries. After *one or more* LLL calls, the combinatorial problem is solved.

Stating it this way suggests that

- [vH 2002] is much faster than [LLL, Schönhage] (indeed, that is what all experiments show),
- and hence, [vH 2002] should be poly-time as well.....

However, it took a long time to prove that
("one or more" = how many?)
(actually, that's not the right question)

Introduction to lattices

Let $b_1, \ldots, b_r \in \mathbb{R}^n$ be linearly independent over $\mathbb{R}$.
Consider the following $\mathbb{Z}$-module $\subset \mathbb{R}^n$

$$L := \mathbb{Z}b_1 + \cdots + \mathbb{Z}b_r.$$

Such L is called a *lattice* with basis $b_1, \ldots, b_r$.

Lattice reduction (LLL): Given a "bad" basis of L, compute a "good" basis of L.

What does this mean? Attempt #1: $b_1, \ldots, b_r$ is a "bad basis" when L has another basis consisting of much shorter vectors.

However: To understand lattice reduction, it does not help to focus on lengths of vectors. What matters are: *Gram-Schmidt lengths*.

Gram-Schmidt

$L = \mathbb{Z}b_1 + \cdots + \mathbb{Z}b_r$

Given $b_1, \ldots, b_r$, the Gram-Schmidt process produces vectors $b_1^*, \ldots, b_r^*$ in $\mathbb{R}^n$ (not in L!) with:

$$b_i^* := b_i \quad \text{reduced mod} \quad \mathbb{R}b_1 + \cdots + \mathbb{R}b_{i-1}$$

i.e.

$$b_1^*, \ldots, b_r^* \text{ are orthogonal}$$

and

$$b_1^* = b_1$$

and

$$b_i^* \equiv b_i \quad \text{mod prior vectors.}$$

Gram-Schmidt, continued

$b_1, \ldots, b_r$: A basis (as $\mathbb{Z}$-module) of L.

$b_1^*, \ldots, b_r^*$: Gram-Schmidt vectors (not a basis of L).

$b_i^* \equiv b_i$ mod prior vectors

$||b_1^*||, \ldots, ||b_r^*||$ are the *Gram-Schmidt lengths* and
$||b_1||, \ldots, ||b_r||$ are the *actual lengths* of $b_1, \ldots, b_r$.

G.S. lengths are far more informative than actual lengths, e.g.

$$\min\{||v||, \quad v \in L, v \neq 0\} \geqslant \min\{||b_i^*||, \quad i = 1 \ldots r\}.$$

G.S. lengths tell us immediately if a basis is bad
(actual lengths do not).

Good/bad basis of L

We say that $b_1, \dots b_r$ is a *bad basis* if $||b_i^*|| \ll ||b_j^*||$ for some $i > j$.

Bad basis = later vector(s) have much smaller G.S. length than earlier vector(s).

If $b_1, \dots, b_r$ is bad in the G.S. sense, then it is also bad in terms of actual lengths. We will ignore actual lengths because:

- The actual lengths provides no obvious strategy for finding a better basis, making LLL a mysterious black box.
- In contrast, in terms of G.S. lengths the strategy is clear:

  (a) Increase $||b_i^*||$ for large i, and
  (b) Decrease $||b_i^*||$ for small i.

Tasks (a) and (b) are equivalent because $\det(L) = \prod_{i=1}^{r} ||b_i^*||$ stays the same.

Quantifying good/bad basis

The goal of lattice reduction is to:
(a) Increase $||b_i^*||$ for large i, and
(b) Decrease $||b_i^*||$ for small i.

Phrased this way, there is a an obvious way to measure progress:

$$P := \sum_{i=1}^{r} i \cdot \log_2(||b_i^*||)$$

Tasks (a),(b), improving a basis, can be reformulated as:

- Moving G.S.-length forward, in other words:
- Increasing P.

Operations on a basis of $L = \mathbb{Z}b_1 + \cdots + \mathbb{Z}b_r$

Notation: Let $\mu_{ij} = (b_i \cdot b_j^*)/(b_j^* \cdot b_j^*)$ so that

$$b_i = b_i^* + \sum_{j<i} \mu_{ij}\, b_j^* \qquad (\text{recall} : b_i \equiv b_i^* \text{ mod prior vectors})$$

LLL performs two types of operations on a basis of L:

(I) Subtract an integer multiple of b_j from b_i (for some $j < i$).

(II) Swap two adjacent vectors b_{i-1}, b_i.

Deciding which operations to take is based solely on:

- The G.S. lengths $||b_i^*|| \in \mathbb{R}$.
- The $\mu_{ij} \in \mathbb{R}$ that relate G.S. to actual vectors.

These numbers are typically computed to some error tolerance ϵ.

Operations on a basis of $L = \mathbb{Z}b_1 + \cdots + \mathbb{Z}b_r$, continued

Operation (I): Subtract $k \cdot b_j$ from b_i ($j < i$ and $k \in \mathbb{Z}$).

1. No effect on: $b_1^*, \ldots, b_r^*$
2. Changes μ_{ij} by k (also changes $\mu_{i,j'}$ for $j' < j$).
3. After repeated use: $|\mu_{ij}| \leqslant 0.5 + \epsilon$ for all $j < i$.

Operation (II): Swap b_{i-1}, b_i, but only when (Lovász condition)

$$p_i := \log_2 ||\text{new } b_i^*|| - \log_2 ||\text{old } b_i^*|| \geqslant 0.1$$

1. $b_1^*, \ldots, b_{i-2}^*$ and $b_{i+1}^*, \ldots, b_r^*$ stay the same.
2. $\log_2(||b_{i-1}^*||)$ decreases and $\log_2(||b_i^*||)$ increases by p_i
3. **Progress counter** P increases by $p_i \geqslant 0.1$.

Lattice reduction, the LLL algorithm:

Input: a basis $b_1, \ldots, b_r$ of a lattice L

Output: a good basis $b_1, \ldots, b_r$

Step 1. Apply operation (I) until all $|\mu_{ij}| \leqslant 0.5 + \epsilon$.
Step 2. If $\exists_i \; p_i \geqslant 0.1$ then swap b_{i-1}, b_i and return to Step 1.
Otherwise the algorithm ends.

Step 1 has no effect on G.S.-lengths and P. It improves the μ_{ij} and p_i's. A swap increases progress counter

$$P = \sum i \cdot \log_2(||b_i^*||)$$

by $p_i \geqslant 0.1$, so

$$\begin{aligned} \#\text{calls to Step 1} &= 1 + \#\text{swaps} \\ &\leqslant 1 + 10 \cdot (P_{\text{output}} - P_{\text{input}}). \end{aligned}$$

Lattice reduction, properties of the output:

LLL stops when every $p_i < 0.1$. A short computation, using $|\mu_{i,i-1}| \leqslant 0.5 + \epsilon$, shows that

$$||b_{i-1}^*|| \leqslant 1.28 \cdot ||b_i^*||$$

for all i. So later G.S.-lengths are not much smaller than earlier ones; the output is a *good basis*.

Denote $l_i := \log_2 ||b_i^*||$. A swap $b_{i-1} \leftrightarrow b_i$ is only made if it decreases l_{i-1} and increases l_i by at least 0.1.

$$l_{i-1}^{\text{old}} > l_{i-1}^{\text{new}} \geqslant l_i^{\text{old}}$$

$$l_{i-1}^{\text{old}} \geqslant l_i^{\text{new}} > l_i^{\text{old}}$$

The new l_{i-1}, l_i are between the old ones.

Properties of the LLL output in our application:

$l_i := \log_2(||b_i^*||)$. Our algorithm calls LLL with two types of inputs.

Type I. $l_1 = 3r$ and $0 \leqslant l_i \leqslant r$ for $i > 1$.
Type II. $0 \leqslant l_i \leqslant 2r$ for all i.

New l_i's are between the old l_i's, so the output for Type I resp. II has $0 \leqslant l_i \leqslant \{3r$ resp. $2r\}$ for all i.

Moreover, an l_i can only increase if a larger l_{i-1} decreases by the same amount. This implies that for an input of Type I, there can be at most one i at any time with $l_i > 2r$.

Whenever the last vector has G.S.-length $> \sqrt{r+1}$, we remove it. So if $b_1, \ldots, b_s$ are the remaining vectors, then

$$||b_i^*|| \leqslant (1.28)^{s-i} \cdot \sqrt{r+1} \leqslant 2^r.$$

Using LLL to solve (or partially solve!) a problem

LLL solves many problems. Suppose a vector v encodes the solution of a problem, and we construct $b_1, \ldots, b_r$ with

$$v \in \mathbb{Z}b_1 + \cdots + \mathbb{Z}b_r$$

Solving a problem with a single call to LLL: If every vector outside of $\mathbb{Z}v$ is much longer than v, then the first vector in the LLL output is $\pm v$. The original LLL paper factors $f \in \mathbb{Z}[x]$ by constructing the coefficient vector v of a factor in this way.

Partial reduction in the combinatorial problem: If $||b_r^*|| > ||v||$ then

$$v \in \mathbb{Z}b_1 + \cdots + \mathbb{Z}b_{r-1}.$$

The initial basis is usually bad, i.e. $||b_r^*||$ is small: We need LLL to make $||b_r^*|| >$ an upper bound for $||v||$.

Applications of LLL, partial progress

Suppose v is a solution of a combinatorial problem, and $\tilde{v} = (v, *, \dots, *)$, and

$$\tilde{v} \in \mathbb{Z}b_1 + \cdots + \mathbb{Z}b_r.$$

LLL-with-removals: If LLL raises $||b_r^*||$ above a bound for $||\tilde{v}||$, then we can throw away b_r and reduce the combinatorial problem:

$$\tilde{v} \in \mathbb{Z}b_1 + \cdots + \mathbb{Z}b_{r-1}.$$

Progress towards finding $\tilde{v}$ (and hence v) is measured in terms of

- the number of vectors removed so far, and
- $P := \sum i \cdot \log_2(||b_i^*||)$, which increases when a swap moves G.S. length forward (bringing us closer to dropping another vector).

A combinatorial problem; a knapsack-type example

Example: Find every subset of $\{D_1, \ldots, D_r\}$ whose sum has length $\leqslant B := 10^5$. We search for $(v_1, \ldots, v_r) \in \{0,1\}^r$ with

$$||\sum_{i=1}^{r} v_i D_i|| \leqslant 10^5 \qquad (1)$$

$$\begin{aligned}
D_1 &= (-3688921210179725062, -2273798960376704320) \\
D_2 &= (-8233711656052404457, \ \ 4387151750437596892) \\
D_3 &= (-6364897933038701741, \ \ 4649403233638190799) \\
D_4 &= (\ \ 8078326574087734047, -8222488196628042845) \\
D_5 &= (\ \ 5967023339103355205, \ \ 4383442706458045299) \\
D_6 &= (-9489167261573791775, -2346235634434299474)
\end{aligned}$$

The last digits are completely irrelevant for problem (1). We can throw them away (divide by B and then round).

A combinatorial problem, continued

$$
\begin{aligned}
D'_1 &= (-368892121017972\cancel{50620}, -227379896037670\cancel{43201}) \\
D'_2 &= (-823371165605240\cancel{44572}, \quad 438715175043759\cancel{68929}) \\
D'_3 &= (-636489793303870\cancel{17417}, \quad 464940323363819\cancel{07992}) \\
D'_4 &= (\quad 807832657408773\cancel{40475}, -822248819662804\cancel{28459}) \\
D'_5 &= (\quad 596702333910335\cancel{52058}, \quad 438344270645804\cancel{52994}) \\
D'_6 &= (-948916726157379\cancel{17758}, -234623563443429\cancel{94743})
\end{aligned}
$$

We can throw the last 5 digits away, or equivalently, divide by $B = 10^5$ and round. The condition

$$||\sum v_i D_i|| \leqslant B$$

implies

$$||\sum v_i D'_i|| \leqslant r.$$

A combinatorial problem, continued

We search for $(v_1, \ldots, v_r) \in \{0, 1\}^r$ for which $\sum v_i D_i$ is short.

$D_1 = (-368892121017972506200, -227379896037670432010)$

We divide by $B = 10^5$ and round. Next, we turn D_1 into:

$\tilde{D}_1 = (1, 0, 0, 0, 0, 0, -368892121017973, -227379896037670)$

The first r entries are called *combinatorial entries*, those are used to recover $v_1, \ldots, v_r$.

The last two entries are called the *data entries*.

A combinatorial problem, continued

$$\begin{aligned}
\tilde{D}_1 &= (1,0,0,0,0,0,-368892121017972,-227379896037670)\\
\tilde{D}_2 &= (0,1,0,0,0,0,-823371165605240,\ \ 438715175043759)\\
\tilde{D}_3 &= (0,0,1,0,0,0,-636489793303870,\ \ 464940323363819)\\
\tilde{D}_4 &= (0,0,0,1,0,0,\ \ 807832657408773,-822248819662804)\\
\tilde{D}_5 &= (0,0,0,0,1,0,\ \ 596702333910335,\ \ 438344270645804)\\
\tilde{D}_6 &= (0,0,0,0,0,1,-948916726157379,-234623563443429)
\end{aligned}$$

We can solve

$$||\sum_{i=1}^{r} v_i D_i|| \leqslant B, \quad v_i \in \{0,1\}$$

by computing all vectors $\tilde{v}$ in

$$L := \mathbb{Z}\tilde{D}_1 + \cdots + \mathbb{Z}\tilde{D}_r$$

of length $\leqslant \sqrt{r \cdot 1^2 + 2 \cdot r^2}$ and then looking at the first r entries.

A combinatorial problem, solving with LLL

Let $b_1, \ldots, b_r$ be an LLL reduced basis of $L := \mathbb{Z}\tilde{D}_1 + \cdots + \mathbb{Z}\tilde{D}_r$.

As long as the last vector has G.S.-length $> \sqrt{r + 2r^2}$, we can throw it away. Say $b_1, \ldots, b_s$ are the remaining vectors. (If we did not LLL reduce, then the last vector would have G.S.-length ≈ 1 even though its actual length is large).

Now any $\tilde{v} \in L$ of length $\leqslant \sqrt{r + 2r^2}$ will be in $\mathbb{Z}b_1 + \cdots + \mathbb{Z}b_s$. So the combinatorial problem has been reduced from dimension r to dimension s.

In our example, s is now 0, so there is no non-zero solution. The same could have been done with less CPU time.

A combinatorial problem, scaling down more

We searched for $(v_1, \ldots, v_r) \in \{0,1\}^r$ for which $|| \sum v_i D_i || \leqslant B$. We divided every D_i by B and then rounded.

But we could have divided by $S \cdot B$ where S is an additional scaling factor. The reason for doing so is because each vector had 30 data-digits, but we're only looking for 6-bit vectors in $\{0,1\}^r$. So we probably did not need 30 data-digits. Lets take $S = 10^{10}$. Dividing by $S \cdot B$ instead of B produces

$$\tilde{D}_1 = (1,0,0,0,0,0,-36889,-22738) \quad (5+5 \text{ data-digits})$$
$$\tilde{D}_2 = (0,1,0,0,0,0,-82337,\ \ 43872)$$
$$\ldots$$

Applying LLL-with-removals to this lattice suffices to prove that there is no non-zero solution.

A combinatorial problem, scaling down and back up

We scaled down by an additional factor S. What happens if we scaled down too much? Then the lattice does not contain enough data to solve the combinatorial problem. However, *any removals we might have made are still correct!* (if the scaled down vector has G.S.-length $>$ bound, then so does the original one).

So if we scale down "too much", then LLL may not solve the problem, but it *can still make partial progress, at a low cost.* Scaling down "too much" is a good idea!

In the example, scale down a factor $S = 10^{10}$, apply LLL-with-removals, and check if the problem is solved. If not, partially scale the data entries back up (reduce S to say 10^5), and apply LLL-with-removals again. Repeat until either $S = 1$ or the problem is solved.

Solving a combinatorial problem, gradual feeding

Suppose that

- the amount of data available is large, while
- the $(v_1, \ldots, v_r)$ we want are small.

With N available data-entries, we can start by using just 1 data-entry,

$$D_1' = (-368892121017972, \cancel{-227379896037670})$$

scale down, and apply LLL-with-removals. As long as the problem is not solved, partially scale a data-entry back up, or insert another data-entry.

The advantage is that we never insert large vectors into LLL, leading to faster LLL reductions, and, that we may solve the problem with a small subset of the data.

Solving a combinatorial problem, gradual feeding

To get a complexity bound we need to

- Make sure that a non-trivial amount of new data is inserted in each step.

  For example, if you scale a data entry back up, scale it up enough so that $\log(\det(L))$ increases at least $\mathcal{O}(r)$.
- Quantify progress so a bound can be derived.

This strategy was originally analyzed for factoring, but turned out to be useful for other LLL applications as well.

Why do we need gradual feeding for factoring?

Combinatorial problem: For which $v = (v_1, \ldots, v_r) \in \{0,1\}^r$ will $\prod f_i^{v_i}$ produce a factor of f in $\mathbb{Q}[x]$.

The paper [BHKS] gives gives *sufficient* data (CLD's, more details later) that can be appended to the vectors v.

Problem 1: The amount of data in [BHKS] is very large. A small subset should suffice.
Problem 2: But we do not know a priori which subset.

Strategy: Gradually add data, selected in such a way that a bounded progress counter is guaranteed to increase.

A practical advantage, early termination

Strategy: Gradually add data, selected in such a way that a bounded progress counter is guaranteed to increase.

This strategy was designed to prove a sharp complexity $\mathcal{O}(r^3)$ for the total number of LLL swaps. But it is useful in practice as well!

Early Termination Strategy: Suppose $\mathrm{MaxCoeff}(f) \approx 10^{1000}$.
Lifting to say $p^a \approx 10^{80}$ often (depends on the Newton polygon) suffices to solve the combinatorial problem.
If $f = g_1 g_2$, each with MaxCoeff $> p^a$ then we'll need to lift more.
But if $\mathrm{MaxCoeff}(g_1)$ is small then we've lifted enough ($g_2 := f/g_1$).

Problem: What about the LLL work done when $p^a \approx 10^{80}$ did not solve the combinatorial problem?
Answer: Our progress counter shows that no LLL work is wasted.

The data entries from [BHKS]

$f \equiv f_1 \cdots f_r \bmod p^a, \qquad N = \mathrm{degree}(f).$

The vector $e_1 = (1, 0, \ldots, 0) \in \{0, 1\}^r$ represents f_1. The paper [BHKS] appends $N - 1$ data entries to e_1:

$$\tilde{e}_1 \;=\; (1, 0, \ldots, 0, \frac{\mathrm{CLD}_0(f_1)}{B_0}, \ldots, \frac{\mathrm{CLD}_{N-2}(f_1)}{B_{N-2}}) \;\in \mathbb{Z}^r \times \mathbb{Q}^{N-1}$$

where

$$\mathrm{CLD}_i(f_1) = \mathrm{Coefficient}_{x^i}\left(f \cdot \frac{f_1'}{f_1}\right)$$

and B_i is an upper bound for $\sqrt{N} \cdot \mathrm{CLD}_i(g)$ for any $g \in \mathbb{Z}[x]$ dividing f. (see [ISSAC'2011] for computing B_i)

Similarly, it computes $\tilde{e}_1, \ldots, \tilde{e}_r \in \mathbb{Z}^r \times \mathbb{Q}^{N-1}$, one vector $\tilde{e}_j$ for each p-adic factor f_j of f.

The data entries from [BHKS], continued

Let L be the $\mathbb{Z}$-span of $\tilde{e}_1, \ldots, \tilde{e}_r \in \mathbb{Z}^r \times \mathbb{Q}^{N-1}$ and the following vectors:

$$(0, \ldots, 0, 0, \ldots, \frac{p^a}{B_i}, \ldots, 0) \in \mathbb{Z}^r \times \mathbb{Q}^{N-1} \quad (i = 0 \ldots N-2)$$

(these additional vectors are needed since $\mathrm{CLD}_i(f_j)$ is only computed mod p^a).

If $v \in \{0,1\}^r$ is a solution to the combinatorial problem then the corresponding $\tilde{v} \in \mathbb{Z}^r \times \mathbb{Q}^{N-1}$ has length $\leqslant \sqrt{r+1}$.

Thus, we can apply LLL-with-removals. The last vector is removed whenever its G.S. length is $> \sqrt{r+1} \; + \; \epsilon$.

For efficiency, we round the data entries in $\frac{1}{B_i}\mathbb{Z}$ to say $\frac{1}{2^r}\mathbb{Z}$.

Gradually feeding the CLD data

Problem 1: [BHKS] gives many data-entries
Problem 2: and they contain large numbers.

Gradual feeding: Insert only 1 data entry at a time, say CLD_i, and process that CLD_i gradually, inserting only $\mathcal{O}(r)$ bits at a time before calling LLL-with-removals again.

This strategy ensures that LLL will never encounter vectors of (G.S.) length $> 2^{3r}$. However, there could be dozens of LLL calls to process just one CLD_i. And we do not know in advance how many CLD_i's are needed to solve the combinatorial problem.

Gradual feeding, continued

Initially, $b_1, \ldots, b_r$ is the standard basis of $\mathbb{Z}^r$. Clearly, any solution to the combinatorial problem is then in $L := \mathbb{Z}b_1 + \cdots + \mathbb{Z}b_r$. We want to decrease $\dim(L)$, but initially, we have to increase it a small amount.

Suppose we have already added d data entries (using $\mathrm{CLD}_{N-2}, \mathrm{CLD}_{N-3}, \ldots$ or $\mathrm{CLD}_0, \mathrm{CLD}_1, \ldots$), and we have also added/removed some vectors, and now have

$$L = \mathbb{Z}b_1 + \cdots + \mathbb{Z}b_s \subset \mathbb{Z}^r \times (\frac{1}{2^r}\mathbb{Z})^d$$

To prove the main complexity result, we design the algorithm in such a way that $s \leqslant \frac{5}{4}r + 1$ at all times.

Suppose we now decide to insert data from say CLD_2.

Gradual feeding, inserting CLD_2

$L = \mathbb{Z}b_1 + \cdots + \mathbb{Z}b_s$ where b_1 looks like $b_1 = (a_1, \ldots, a_r, *, \ldots, *)$.
Let

$$b_0 := (0, \ldots, 0, \frac{p^a}{S \cdot B_2}) \quad \text{rounded to } \frac{1}{2^r}\mathbb{Z}$$

where scaling factor S makes $||b_0|| \approx 2^{3r}$.
Compute (for $i = 1, \ldots, r$)

$$R_i := \frac{\mathrm{CLD}_2(f_i)}{S \cdot B_2} \quad \text{rounded to } \frac{1}{2^r}\mathbb{Z}$$

and

$$b_1^{\text{new}} := (a_1, \ldots, a_r, *, \ldots, *, \sum a_i R_i) \ \in \ \mathbb{Z}^r \times (\frac{1}{2^r}\mathbb{Z})^{d+1}.$$

New basis: $b_0, b_1^{\text{new}} \ldots, b_s^{\text{new}}$

(**Or:** $b_1^{\text{new}} \ldots, b_s^{\text{new}}, \cancel{b_0}$ if max length $\approx 2^{2r}$).

Gradual feeding, gradually scaling back up

Due to scaling, every b_i now has G.S. length $\leqslant 2^{3r}$ and, due to rounding, every entry has bitsize bounded by $\mathcal{O}(r)$.

After LLL-with-removals, the last vector will have G.S. length $\leqslant \sqrt{r+1}+\epsilon$ and from this one can show that every vector will have G.S. length $\leqslant 2^r$.

The recently added entry was scaled down by a (potentially large) factor S. Now reduce S (scaling back up) so that
(a) the largest length becomes $\approx 2^{2r}$, or (b) S becomes 1.

Apply again LLL-with-removals (then the largest G.S. length is again $\leqslant 2^r$). Then scale back up again. Repeat until:

- The combinatorial problem is solved, or
- S becomes 1 (then move on to the next CLD_i).

Gradual feeding, practical observations

We ensure that LLL never encounters vectors of length $> 2^{\mathcal{O}(r)}$ by inserting only $\mathcal{O}(r)$ new bits of data at a time. That results in excellent practical performance.

1. We insert little data at a time, so there could be many LLL calls (say n_{calls}).
2. Even if n_{calls} is large, the majority of the CPU time could be concentrated in just 2 or 3 calls! (example in [Belabas 2004]).

   So if B_L is the bound for 1 LLL call, then $n_{\text{calls}} \cdot B_L$ will be a bad bound, it could be almost n_{calls} times higher than observed behavior. A good bound must have:

Key property: The bound for all LLL calls combined should have the same $\mathcal{O}(\ldots)$ as the bound for a single call! (= a great hint!)

Combinatorial problem; properties of $b_1, \ldots, b_s$

$v \in \{0,1\}^r$ is a *good vector* if $\prod f_i^{v_i}$ gives a factor of f in $\mathbb{Q}[x]$.

At any stage we have $b_1, \ldots, b_s \in \mathbb{Z}^r \times (\frac{1}{2^r}\mathbb{Z})^d$ with:

- For every *good vector* v, there exists $\tilde{v} \in \mathbb{Z}b_1 + \cdots + \mathbb{Z}b_s$ whose $\mathbb{Z}^r$-component is v, and length is $\leqslant \sqrt{r+1}$.
- $s \leqslant \frac{5}{4}r + 1$
- At most one i has $2^{2r} < ||b_i^*|| \leqslant 2^{3r}$ (**the large vector**).
- $1 \leqslant ||b_i^*|| \leqslant 2^{2r}$ for all other i
- Actual lengths are bounded by $2^{\mathcal{O}(r)}$ as well.
- d is bounded by $\mathcal{O}(r^2)$ (Note: if we store the Gram-matrix then we need not store the $(\frac{1}{2^r}\mathbb{Z})^d$-components of $b_1, \ldots, b_s$).

One stage in the Combinatorial problem

Each stage in solving the combinatorial problem consists of

1. Adding CLD-data, either
   - Scale up a data-entry, or
   - Add a data-entry (increases d by 1), or
   - Add a data-entry and a new vector $(0, \ldots, 0, p^a/(S \cdot B_i))$ of length 2^{3r} (increases s and d by 1).
2. LLL-reduce $b_1, \ldots, b_s$
3. While $||b_s^*|| > \sqrt{r+1} + \epsilon$ decrease s by 1.
4. Test if solved (if so, return the factorization $f = g_1 \cdots g_s$)

vH, Novocin:

- **Using Progress Counter**: The total number of LLL swaps in all stages combined is $\mathcal{O}(r^3)$ (= bound for one LLL call).
- **Using Active Determinant**: #stages $\leqslant \mathcal{O}(r^2)$.
- **LLL Cost**: $\mathcal{O}(r^3) \cdot \tilde{\mathcal{O}}(r^3) + \mathcal{O}(r^2) \cdot \tilde{\mathcal{O}}(r^4) = \tilde{\mathcal{O}}(r^6)$.

Progress counter, overview

$b_1, \ldots, b_s$ = current vectors.

$$P := \sum_{i=1}^{s} (2r + (i-1)\frac{4}{5}) \cdot \log_2(||b_i^*||) + (r-s) \cdot 3r \cdot 2r.$$

The last term counts the progress that occurs when s decreases (when vectors are removed).

The $(i-1)\frac{4}{5} \cdot \log_2(||b_i^*||)$ counts progress that occurs when an LLL-swap moves G.S.-length forward (bringing us closer to removing more vectors).

The $2r \cdot \log_2(||b_i^*||)$ is new here, so we can prove #stages $\leqslant \mathcal{O}(r^2)$ without having to introduce the "Active Determinant".

Progress counter over time

$$P := \sum_{i=1}^{s} (2r + (i-1)\frac{4}{5}) \cdot \log_2(||b_i^*||) + (r - s) \cdot 3r \cdot 2r.$$

The properties of $b_1, \ldots, b_s$ imply that P can not be larger than $\mathcal{O}(r^3)$. The initial value is 0.

With some simple tests, we can avoid adding data with little impact on the G.S.-lengths of $b_1, \ldots, b_s$. This way, every stage increases P by at least $\mathcal{O}(r)$.

Inserting a vector decreases $r - s$ by 1, but the inserted vector has G.S.-length $=$ actual length $= 2^{3r}$. So P does not decrease.

Every LLL swap increases P by at least $\frac{4}{5} \cdot 0.1$.

If a vector is removed, then $r - s$ increases by 1, and since $2r + (i-1)\frac{4}{5} \leqslant 3r$, vector-removal does not decrease P *except* if the removed vector has G.S.-length $> 2^{2r}$. This case is analyzed separately.

The only time that P can decrease

$P_0 := P$. Now insert $b_0 := (0, \ldots, 0, p^a/(S \cdot B_i) \text{ rounded})$ of size 2^{3r} and insert the CLD_i-data into $b_1, \ldots, b_s$.

$P_1 := P$. Now call LLL, say output is $b_1, \ldots, b_{s+1}$.

$P_2 := P$. If $||b_{s+1}^*|| > \sqrt{r+1} + \epsilon$ then remove b_{s+1}.

$P_3 := P$. Now P_3 could be $< P_2$ but only if $||b_{s+1}^*||$ was $> 2^{2r}$. We can still show

$$P_3 - P_0 > \text{const} \cdot (r \ + \ \#\text{swaps})$$

if at least one of $b_1, \ldots, b_s$ received a data-entry $\geqslant 2^{2r}$.

Recipe: If the minimal amount of scaling, $S = 1$, produces no data-entry $\geqslant 2^{2r}$ then our vectors already had small CLD_i. Then move on to the next CLD (increase p^a if no suitable CLD remains).
Termination: [BHKS] proved a quadratic bound for $\log(p^a)$. Observations indicate it should be linear.

Complexity

$f \in \mathbb{Z}[x]$, degree N, largest coefficient has h digits, and f has r factors mod p.

Even if we can not prove a linear bound for $\log(p^a)$, we still get an improved complexity:

$$\tilde{\mathcal{O}}(r^6) + \mathrm{Pol}_{\mathrm{deg}<6}(N, h)$$

[Schönhage] also had degree 6, but our degree-6 term depends solely on r (which is almost always $\ll N, h$).

The costs of Hensel lifting and preparing the LLL input (computing $\mathrm{CLD}_i(f_j)$'s, scaling, rounding, taking linear combinations) have degree < 6 so theorists may consider them unimportant. However:

Difficult Open Problem: Hensel lifting dominates the CPU time for most inputs, so proving a linear bound for $\log(p^a)$ is important for accurately describing the behavior of the algorithm.