1 Roots

Given an odd positive integer N we want to test if N is a perfect power. If so, we'd like to write $N = n^k$ for k maximal.

1. design and implement an algorithm to compute the k-th root of N, k odd. Compare the runtime(s) of

   - using reals (bisection, Newton, ??)
   - p-adically, p odd
   - 2-adically

   Compare the costs (complexity) of *finding* the root (assuming it exists) to the cost of *verifying* the root.

2. the same for $k = 2$, in particular, the 2-adic version. You can get some inspiration from `Oscar/examples/PerfectPower.jl`.

3. Lenstra and Bernstein suggest the following algorithm:

   - for all prime powers p^l up to $\log_2(N)$
   - use the function above to compute the p^l-root by iterated p-th roots (or garbage if the root does not exist)
   - collect all those roots in some array A, append A by N
   - compute a coprime basis C for A
   - the gcd of the valuations of N at the elements of C should be the exponent.

The coprime basis can be computed in linear time (Berstein, see source for refrences) In julia/ Oscar: `coprime_base` will work, in Magma: `CoprimeBase`

2 Multiplicative Dependencies

See `Hecke/examples/MultDep.jl` and `Hecke/examples/pAdicConj.jl` for hints In julia/ Oscar you can use `C = Hecke.qAdicConj(k, p)` to intialize the complection(s) above all primes above p. `conjugates(a, C)` will then get the vector of q-adics. Note, this can only handle "easy" primes - but its optimised for speed. The generic case is currently (again) under development.

In Magma, at the user-level, you probably have to create the order, split the prime and then use `Completion` to get the completion at each prime. In comparison to above, `Completion` can handle the generic situation. At least internally, a more optimised, but restricted function is used.

Use this and the logarithms to find multiplicative dependencies between units. To get units: in julia/ Oscar: define a field then compute the maximal order follwed by the unit group (using e.g. `maximal_order, unit_group`), in Magma, you'll need `MaximalOrder, UnitGroup`.

In either case, you get an abstract unit group back (as some form of abelian group). As the group is mostly ininite, you cannot just ask for random elements. In julia/ Oscar s.th. like `sum(rand(-100:100)*g for g = gens(U))` should work, in Magma `&+ [Random([-100..100])*g : g in Generators(U)]`. To get a unit then use the second return value: the map.

For larger exponents, you'll need to use `unit_group_fac_elem` in julia. Now you should *not* ask for the conjugates, but directly for the logarithms `conjugates_log`.

3 Linear Algebra

1. Given $A \in \mathrm{Gl}(n, \mathbb{Z})$ find the inverse by p-adic lifting. (This is not the fastest method, but fun to try)
2. Given an integral square matrix A, find the (pseudo) inverse using p-adic lifting and rational reconstruction. `rational_reconstruction, RationalReconstruction`
3. Given integral matrices A and b, solve $Ax = b$, maybe start with the case of unique solutions.
4. Lets do determinants! Let A be integral and square
   - for a random b, solve $Ax = b$. Write $x = s/d$ for an integral vector s and $d > 0$ s.th. the content of s and d are coprime (write in lowest terms).
   - Then d is the index of the module generated by A inside the module generated by A and b.
   - d divides the determinant, but in general, d is smaller
   - what is the relationship between A, d and $\det A$?
   - to continue: 2-options: compute $\det A$ modulo p for some additional p and then $\det A$ using CRT or
   - interpret $\det A$ as the index of the module generated by A in $\mathbb{Z}^n$, use the vector s above to get a basis for $\langle A, b\rangle$ and iterate.

   Generically for random matrices, the 1st d above is the determinant - possibly up to a small factor. For bad matrices $d = \sqrt[n]{\det A}$.
5. Now repeat, over a number field
6. Now repeat, for polynomial matrices

4 Factoring

1. Use `lll` to implement the rational reconstruction: given $M > 0$ and $r \bmod M$ write $r = a/b \bmod M$, ie. $br = a \bmod M$ and a, b small.

2. Given some ideal A in some (maximal) order and r in the same order, try to find a small representative.

3. use `factor_mod_pk` to implement your own version of Zassenhaus factoring or the van Hoeij one. Compare or try the power sums as well as the logarithmic derivatives

4. Fix $K = \mathbb{Q}[t]/f$ for f monic, integral and irreducible. Let p be a prime s.th. f is square-free modulo p and that f has a root $a \in \mathbb{Z}$ so $f(a) = 0 \bmod p$

   - $P = \langle p, \alpha - a\rangle$ is a prime ideal of degree 1 over p
   - $H = (h_{i,j})$ s.th. $h_{1,1} = p$, $h_{i,i} = 1$, $h_{i,1} = a^i \bmod p$ and 0 otherwise is a HNF-basis matrix for P.
   - Now, if $b \in \mathbb{Z}$ s.th $f(b) = 0 \bmod p^k$ is a lift of a, then this construction will give the HNF basis of P^k "for free". Compare this to actually computing `basis_matrix(P^k)`.
   - generalise this for primes of degree > 1
   - Let P and Q be prime ideals / powers of prime ideals of degree 1, use CRT to write down the HNF basis for PQ.

5. Compare the different factoring algorithms by runtime. Zassenhaus and van Hoeij are in `Hecke/src/NumField/NfAbs/PolyFact.jl`, Trager's method is in `Hecke/src/NumField/NfAbs/Elem.jl`. Careful with the overhead due to compiling.

6. In order to multiply polynomials/ power series over $\mathbb{Q}_q$, Kronnecker-segmentation can be used. The idea is to map a polynomial over $\mathbb{Q}_q$ to a single polynomial over $\mathbb{Z}$ such that the product over $\mathbb{Q}_q$ can be obtained from the product over $\mathbb{Z}$. This allows to leverage the FFT-techniques easily. Try this. A slightly more complicated version for polynomials over power series over q-adics is in `Hecke/src/Misc/Series.jl` in `mymul_ks`. In Magma this would work too, but needs to be in the kernel. The Magma-language is not fast enough.

5 Galois Theory

In julia/ Oscar this is implemented in `Oscar/experimental/GaloisGrp/GaloisGrp.jl`, in Magma in `package/Ring/Galois` I think. But `methods(galois_group)` or `GaloisGroup:Maximal;` should be of help.

1. Let k be any number field (of Galois group size < 20, say.) Use `galois_group, fixed_field, (maximal_)subgroups`, or `GaloisGroup, GaloisSubfield, Subgroups` to find algebraic descriptions of subfields of the splitting field

2. (Magma only) find some solvable polynomial of degree > 4 and use `SolveByRadical` to see why we can, but should not

3. Let C be the Galois-Ctx of either Oscar or Magma. Then it is possible to obtain the roots of the underlying polynomial to any desired precision. Use this to write a simple procedure to map a (potential) block system to a subfield.

4. given block systems and their subfields, find block systems for the compositum and the intersection

5. Use multivariate polynomials (or sl-polys) and the Galois-infrastructure to work in the splitting field of the underlying polynomial: each $f \in \mathbb{Z}[x]$ defines, via evaluation at the roots, an element in the splitting field. You'll need

   - `Oscar.GaloisGrp.upper_bound`
   - `Oscar.GaloisGrp.isinteger`

   `fixed_field` and `GaloisSubfield` are implemented in this fashion. The hard part is to find suitable polynomials to evaluate. In `Oscar/experimental/GaloisGrp/Qt.jl` is also a block-system-to-subfield implementation for function fields using this technique.