

# Computer Algebra

Wieb Bosma

*Radboud Universiteit Nijmegen*

6 February 2007

**PART I**  
INTRODUCTION

## What is Computer Algebra?

- No generally accepted definition
  - Algorithms for algebraic objects
  - Exact vs Approximative
  - Symbolic vs Numerical Computing
- This course:
  - Algorithms central
  - Practical usage in mind: complexity!
- In summary:
  - *what* can be computed with modern computer algebra systems, and
  - *how* is it done?

## Computational domains

Rough outline of the scope:

algorithms to compute with combinatorial objects (like graphs), and in those groups, rings, fields, and their associated modules, algebras, etc. for which the objects can be represented and tested for equality on a computer, and for which the operations can be performed effectively.

$\mathbb{Z}, \mathbb{Q}, \mathbb{Q}(\alpha), \mathbb{Q}_p$

$\mathbb{Z}/n\mathbb{Z}, \mathbb{F}_p, \mathbb{F}_q$

$R[x], R[x]/(f), R((x)),$

$R[x_1, x_2, \dots, x_n], R[x_1, x_2, \dots, x_n]/I,$

$\text{Hom}(V, W), R[[x]], R((x))$

$\text{Sym}(n), K_n$

## Representation of Objects

Objects stored as a finite number of *bits*. The *size* of an object is the number of bits. Objects may have several distinct *representations*, between which we may have to do *conversions*. But within a fixed representation, objects may have more than one representation: a *normal form* is desirable.

For example:

**integers** in  $g$ -adic representation, or fully factored in primes

**polynomials** dense (coefficient vectors) or sparse (coefficient, exponent pairs)

**permutations** cycles, image lists, products of transpositions

## Computational tasks

- *Perform the arithmetic operations in the computational domains*
  - Addition, multiplication, inversion, powering, composition, actions
- *Normal form computation, conversion between representations*
  - Basis representation, factorization
- *Membership and equality testing*
  - Conversion, comparison
- *Structural computation, mappings*
  - Generators and relations

Many of the most important tasks can be interpreted as conversion between representations!

## Computational models

Tasks are executed by way of *algorithms* on *multitape Turing machine* operating on strings of bits. *Computational complexity* is measured in number of *bit operations*.

Sometimes we express operations in a higher level *algebraic model* of computation, where steps are *elementary algebraic operations*.

**Example** Multiplication of  $f, g \in R[x]$ , where  $\deg f = m$  and  $\deg g = n$  can be done with  $(m+1)(n+1)$  multiplications and  $m \cdot n$  additions in  $R$ .

*Note that the complexity depends on the representation!*

## Asymptotics

Complexity functions: partial  $f : \mathbb{R} \rightarrow \mathbb{R} \cup \infty$  that is defined and non-negative for all integers  $n \geq N$ .

$f = \mathcal{O}(g)$ :

$$\exists C > 0, N \in \mathbb{N} : \forall x > N : f(x) \leq C \cdot g(x)$$

$f = \Omega(g)$ :

$$\exists C > 0, N \in \mathbb{N} : \forall x > N : f(x) \geq C \cdot g(x)$$

$f = \Theta(g)$ :

$$f = \mathcal{O}(g) \quad \text{and} \quad f = \Omega(g)$$

$f = o(g)$ :

$$f(n)/g(n) \rightarrow 0 \quad \text{when} \quad n \rightarrow \infty$$

## Complexity classes

P

the class of algorithms with deterministic polynomial time complexity: the complexity is  $\mathcal{O}(x^d)$  for some  $d \in \mathbb{N}$

NP

the class of algorithms with non-deterministic polynomial time complexity: the complexity of *verifying* the correctness of a solution (provided by some oracle, say) is  $\mathcal{O}(x^d)$  for some  $d \in \mathbb{N}$ ; *finding* the correct solution may not be possible in polynomial time

Sometimes trade-offs between time and space complexity

*The true picture of easy versus hard problems may be much more complicated!*

## Some general techniques

- *probabilistic* rather than deterministic methods (gives to *expected* running times)  
Ex: Pollard  $\rho$  algorithm (below)
- iterative and *recursive* methods: divide and conquer  
Ex: Karatsuba algorithm (exercise)
- *homomorphism* methods: mapping to easier structure combined with bounds  
Ex: modular methods (polynomial factorization)
- rewriting  
Ex: Gröbner basis algorithm

## Elementary Algorithms

- integer addition and subtraction in  $\mathcal{O}(\log n)$
- integer multiplication and division in  $\mathcal{O}((\log n)^2)$
- exponentiation (powering) by repeated squaring and multiplication
- polynomial evaluation: Horner's method

## First Example: Pollard- $\rho$

Pollard's  $\rho$  method for integer factorization is based on the 'birthday paradox':

*taking a random sample of size  $O(\sqrt{n})$  of a set of cardinality  $n$  is expected to give a collision*

choosing a random  $f : \mathbb{Z}/n\mathbb{Z} \rightarrow \mathbb{Z}/n\mathbb{Z}$  and  $x_1$ , then  $x_1, x_2 = f(x_1), x_3 = f(f(x_1)) = f(x_2), \dots$  mod  $p$  will behave randomly in  $\mathbb{Z}/p\mathbb{Z}$ , for any prime divisor  $p$  of  $n$ . Hence a collision  $x_i \equiv x_j \pmod{p}$  is expected after  $O(\sqrt{p})$  steps! Since  $x_i \equiv x_j \pmod{N}$  is unlikely (especially if  $p \ll n$ ), we detect the unknown  $p$  by computing  $\gcd(x_i - x_j, n)$ .

$f(x) = x^2 + 1 \pmod{n}$ , with  $x_1 = 2$  is standard

## An optimisation

By the pigeonhole principle the sequence mod  $p$  will become periodic after say  $s + t$  steps, so that  $x_{s+1} \equiv x_{s+t+1} \pmod{p}$ , and the ' $\rho$ ' has a 'tail' of length  $s$  and a 'cycle' of length  $t$ .

Instead of comparing any two entries  $x_i, x_j$ , the same result can be achieved more efficiently by noting that for some  $m$  one gets  $x_{2m} \equiv x_m \pmod{p}$ , the least such  $m$  being the smallest multiple of  $t$  exceeding  $s$ .

Pollard's  $\rho$  method runs heuristically in expected time essentially  $\sqrt{p}$  to find the prime factor  $p$

## Second Example: baby-step giant step

The *Discrete Logarithm Problem* asks, for given finite abelian group  $G$ , and elements  $g, h \in G$  to decide whether  $g = h^m$  for some  $m \in \mathbb{Z}_{\geq 0}$ , and if so, to find  $m = \log_h g$ .

It will be assumed that group operations in  $G$  can be performed efficiently. Also, un-equality testing is necessary; unique representation of group elements and efficient equality testing preferable.

*Note that the representation of group elements is important for the discrete logarithm problem: in the additive group  $\mathbb{Z}/n\mathbb{Z}$  the problem is trivial, and any finite cyclic group is isomorphic to some  $\mathbb{Z}/n\mathbb{Z}$ .*

Determining the order of a (sub)group is a closely related problem, as we will see.

The discrete logarithm  $\log_h g$  is only determined modulo the order  $n = n_H = \#H$  of  $H$ .

Important special case:  $H = G$ , so  $G$  is cyclic and  $h$  is a generator; the decision problem is trivial.

The trivial discrete logarithm algorithm proceeds by computing  $1 = h^0, h = h^1, h^2, \dots$  until either  $h^m = g$  and  $m = \log_h g$  or  $h^k = 1$  for some  $k \geq 1$ , in which case  $g \notin H$ . In any case the algorithm takes  $O(n_H)$  operations in  $G$ . Note that in the second case the order  $n_H = \#H$  has been determined (if equality testing easy).

*The following baby-step giant-step algorithm finds discrete logarithms in  $O(\sqrt{n_H} \log n_H)$  multiplications and comparisons in  $G$ ; we require storage for  $O(\sqrt{n_H})$  elements.*

Let  $B$  be an upper bound on  $\log_h g$ ; take  $B = n_H$  if it is known, and otherwise one tries  $B = 2^1, 2^2, 2^3, \dots$  in succession.

Put  $b = \lceil \sqrt{B} \rceil$ . If  $g \in H$  then  $\log_h g < b^2$ , and so there exist  $0 \leq i, j < b$  such that  $g = h^{ib+j}$ , that is,  $\log_h g = ib + j$ . Compute a sorted (or hashed) lookup table of  $h^j$  for  $j = 0, 1, \dots, b-1$  which takes  $O(b \log b)$  group operations. Next compute  $g \cdot h^{-ib}$  for  $i = 0, 1, \dots, b-1$  until a match in the lookup table is found, so  $g \cdot h^{-ib} = h^j$ , and therefore  $g = h^{ib+j}$ .

If  $g \notin H$  no match will be found.

The order of  $H$  can be found by taking  $g = 1$  (and excluding  $i = 0 = j$ ).

## **An Overview of Algorithms to Follow**

- The Fast Fourier Transform  
and consequences for fast multiplication
- The Euclidean Algorithm  
in many incarnations with many applications
- The Gaussian Elimination Algorithm  
for solving systems, finding matrix inverses and determinants
- The Lenstra-Lenstra-Lovász algorithm  
with surprising applications for short vectors
- Hensel and Newton iteration  
and other methods for root isolation, separation etc.
- Gröbner Basis Algorithm  
again with various applications
- (towards) the Risch Algorithm