Elementary complexity theory

7 septembre 2011

1 Languages, automata

A standard reference is the book by John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman [6]. A survey by J.-E. Pin is available on the web [12].

Let Σ be an *alphabet* i.e. a finite set of symbols. The set of words on Σ is denoted Σ^*. A *language* is a subset of Σ^*.

For a language $L \subset \Sigma^*$ the *decision problem* is as follows. On input a word $w \in \Sigma^*$ we want to decide if w belongs to L or not. Examples : even integers, prime integers, etc.

For a function $f : \Sigma^* \to \Sigma^*$, the *functional problem* is as follows. On input a word $w \in \Sigma^*$ we want to evaluate $f(w)$. Examples : multiplying, factoring, etc.

There are three *regular operations* on languages : the union $L_1 \cup L_2$, the concatenation $L_1 L_2$, and the transitive closure L^* or Kleene star.

We denote ϵ the empty word, $\emptyset$ the empty language, $\{\epsilon\}$ the language consisting of the single empty word, and $\{a\}$ the language consisting of the single word a with length 1. We call $\{a\}$, $\{\epsilon\}$, and $\emptyset$ the elementary languages.

Applying iteratively the regular operations to these elementary languages we obtain *regular languages.* A regular language is described by a *regular expression.*

Question 1 *Give a regular expression for the language in* $\{0,1\}^*$ *consisting of all words ending with* 00.

Same question for the language consisting of all words containing three consecutive 1.

Same question for the language consisting of all words whose seventh letter is a 0.

A (finite deterministic) *automaton* is a 5-uple $M = (Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is an alphabet, $\delta : Q \times \Sigma \to Q$ is the *transition function*, q_0 the *initial state* and F the set of *final states.* We extend δ to a function $\hat{\delta} : Q \times \Sigma^* \to Q$. The *accepted* language is $L = \hat{\delta}^{-1}(F)$.

Question 2 *Find an automaton that accepts words in* $\{0,1\}^*$ *with even length.*

Find an automaton that accepts words in $\{0,1\}^*$ *terminating with a* 1.

Find an automaton that accepts words in $\{0,1\}^*$ *with even length, terminating with a* 1.

Question 3 *Find an automaton that accepts words in* $\{0,1\}^*$ *with as many zeros as ones, or prove that such an automaton does not exist.*

The languages accepted by finite deterministic automata are exactly the regular languages.

Question 4 *Is the intersection of two regular languages a regular language ?*

Non-deterministic automata are a generalization. There are several possible transitions at every step. A word is accepted if there is a sequence of possible transitions that leads to a final state. Formally the transition function is now $\delta : Q \times \Sigma \to 2^Q$. We define $\hat{\delta} : Q \times \Sigma^* \to 2^Q$ by $\hat{\delta}(q, \epsilon) = \{q\}$ and

$$\hat{\delta}(q, wa) = \cup_{r \in \hat{\delta}(q,w)} \delta(r, a).$$

The accepted language is the inverse image by $\hat{\delta}$ of the set of all subsets of Q containing a final state.

Question 5 *Prove that non-deterministic automata accept the same languages as deterministic automata.*

Non-deterministic automata with ϵ-transitions are a further generalization of non-deterministic automata. This time, there exist transitions for free : one can follow these ϵ-arrows without spending a letter. Formally the transition function is now $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$. If $R \subset Q$ is a set of states, we define the ϵ-closure of R to be the set of states that can be reached from R using ϵ-transitions only. This closure is denoted $\epsilon{-}CLOSURE(R)$. We define $\hat{\delta} : Q \times \Sigma^* \to 2^Q$ by $\hat{\delta}(q, \epsilon) = \epsilon{-}CLOSURE(\{q\})$ and

$$\hat{\delta}(q, wa) = \epsilon{-}CLOSURE\left(\cup_{r \in \hat{\delta}(q,w)} \delta(r, a)\right).$$

The accepted language is the inverse image by $\hat{\delta}$ of the set of all subsets of Q containing a final state

Question 6 *Prove that non-deterministic automata with ϵ-transitions accept the same languages as deterministic automata.*

Question 7 *Prove that any regular language is accepted by an automaton.*

Prove that if L is a regular language, there exists an integer n such that for every word w in L with length $|w| \geq n$ one can write $w = abc$ with

1. $|ab| \leq n$,
2. $|b| \geq 1$,
3. ab^kc *belongs to* L *for every integer* $k \geq 0$.

Les $L \subset \{0, 1\}^*$ be the set of words having has many zeros as ones. Is L a regular language ?

Question 8 *Give an automaton with ϵ-transitions that accepts* $10 + (O + 11)O^*1$.

Same question for $01((10)^* + 111)^* + 11$.

Question 9 *Give a deterministic automaton that accepts* $(0 + 1)^*110(0 + 1)^*$.

2 Complexity

We refer the reader to Papadimitriou's book [11] for a complete treatment of these matters. A first natural approach is to count *elementary operations*. The complexity of addition of two integers given in decimal representation is linear in the *size of the input*. The complexity of the naive multiplication algorithm for multiplication of two integers given in decimal representation is quadratic in the *size of the input*. Same remarks hold true for polynomials.

Question 10 *Give a rough estimate for the complexity of Euclidean division of polynomials (resp. integers).*

The *memory space* is the number of bits used by the algorithm and the *running time* is the number of elementary logic or arithmetic bit operations done by the processor. In this model, memory access is assumed to require constant time. The *time* (resp. *space*) *complexity* of a given algorithm is a function from $\mathbb{N}$ to $\mathbb{N}$ that to any instance x associates an upper bound $f(|x|)$ for the running time (resp. used memory space) of the algorithm. This upper bound should only depend on the size $|x|$ of x.

The complexity is therefore not well defined as a function unless one restricts to a given simple family of natural functions, in which case one can define the complexity as the smaller function in the family bounding the running time (resp. memory space).

2.1 Sorting

Let $L = (L_1, L_2, ..., L_n)$ be a list (vector) of integers. One looks for an ordered list $M = (M_1, M_1, ..., M_n)$ i.e. $M_1 \leq M_2 \leq ... \leq M_n$ such that $\{L_1, ..., L_n\} = \{M_1, ..., M_n\}$.

The principle of *bubble sort* is to scan the list from left to right and permute any too unordered succesive elements. One loops until the list is sorted (the last scan results in no permutation). One can easily prove that this algorithm finishes. We measure its complexity as the maximum number of memory accesses or comparisons that may be necessary to sort any list with n elements.

Question 11 *Show that this number is bounded by a constant times n^2.*

Thus if the complexity is to be chosen among all functions $n \mapsto An^\alpha$ for A and α any two real numbers, one may choose $\alpha = 2$. We do not ask about A. We say that bubble sort has quadratic time complexity.

The intrinsic complexity of a problem may be defined to be the complexity of the asymptotically fastest algorithm that solves this problem. This means that one restricts to a family of complexity functions that is well adapted to the problem.

Another way of sorting consists in selecting the largest element by successive pairwise eliminations like in a tournament. One forms pairs of elements and select the larger one in each pair. The winners of this first round then compete again by pairs and so on until on reaches the top of the pyramide. This process only finds out the largest element. To deduce a sorting algorithm one imagines a binary tree-like hierarchic organization. The top (root) of the tree corresponds to the chairman's position. The chairman has too deputy chairmen and each deputy chairman has two subordinates and so on. At the beginning of the algorithm all positions are assumed to be vacant excepted the lowest ones (the leaves of the tree) that are filled with the elements to be sorted. In every pair of sister leaves one chooses the larger value and on promotes it to the second hierarchic level. The entries in this second level are then compared by pairs and the best is promoted again to the third level and so on. However one is very careful when promoting to fill backward recursively all vacant positions (except possibly the lowest ones) with the better among the two subordinates of the last promoted. This selection process is continued until a chairman's position has been filled (with the largest value in the list). One then assumes that this chairman retires and one fills the chairman's position with the better among the two deputy chairmen. One then fills the vacant deputy chairman's position with the better among the two subordinates of the former deputy chairman. And so on. One then assumes that every successive chairman retires and iteratively promote again and again until all the values in the initial list have been removed after reaching (in decreasing order) the root of the tree.

Question 12 *Show that this algorithm sorts a list of size n with less than $An \log n$ elementary operations for a given real number A that we do not try to make explicit. In view of that, what can be said on the complexity of sorting ?*

It may be difficult to find out the exact intrinsic complexity of a problem. Any algorithm provides an upper bound. Lower bounds often come from naive considerations from information theory and they are very rough.

Concerning the sorting problem, inefficient algorithms like bubble sort are said to be quadratic while efficient ones like selection sort are said to be quasi-linear. Although this difference of efficiency may be spectacular for large values of n (think about the Chinese social system) both algorithms are *polynomial time* in the size n of the problem.

2.2 Graphs

We consider the problem of finding a path in a graph. We are given a graph by its finite set of vertices V and $E \subset V \times V$ the set of edges. We are given two vertices A and B and we look for a path from A to B (if there is some). Such a path should be output as a list of vertices $v_0 = A$, v_1, ..., $v_I = B$ such that for any i, there is an edge between v_i et v_{i+1}. A silly algorithm would enumerate all pathes starting from A and would have exponential complexity in the number $n = |V|$ of vertices in the graph.

Question 13 *What is the size of the input in that case ? Give a polynomial time algorithm that finds a path (if there is some) between two vertices in a graph. Compute the smallest real number α such that the number of necessary elementary operations is $O(n^\alpha)$ where $n = |V|$.*

2.3 Groups

One is sometimes surprised by the complexity of some very simple and natural problems.

For example, it is a natural question, given a group G and elements σ_0, σ_1, ..., σ_k in it, to ask if σ_0 is in the subgroup generated by σ_1, ..., σ_n. This is called the membership problem.

If G is the group of permutations on n letters then this problem is solved in polynomial time in n using interesting general techniques invented by nineteenth century topologists (see [8, chapitre 2] and [3, chapitre 8]).

If G is the linear group $GL_2(\mathbb{Z})$ of integer two by two invertible matrices then the membership problem is undecidable i.e. there is no algorithm for it. This is also the case in general when G is a group given by generators and relations (see [13] for a survey).

Coming back to the case where G is finite we may ask for its order or more generally the order of the subgoup generated by a given set of elements.

This problem is at least as difficult as the membership problem. Indeed, if one knows how to compute the order of a subgroup, one can compute the order of $< \sigma_1, \sigma_2, ..., \sigma_n >$ and the order of $< \sigma_0, \sigma_1, ..., \sigma_n >$. These two numbers are equal if and only if $\sigma_0 \in < \sigma_1, \sigma_2, ..., \sigma_n >$.

This way of solving the membership problem using an algorithm for computing the cardinality of a subgroup is called a *reduction*. The existence of a reduction of a problem to another one defines an order on problems (here one says that the size problem for a finite group is as difficult as the membership one).

One has to make clear which kind of reduction is allowed. This may depend on the context. For example, in order to solve an instance of problem $\mathcal{A}$, does one need to be given the solution to one or many instances of problem $\mathcal{B}$? And assuming this information to be given for free, how much computation is needed to deduce the solution to problem $\mathcal{A}$?

In the example we just gave, one uses two instances of the size problem to solve one instance of the membership problem. And the translation is done in linear time in the size of the entries.

Question 14 *Let G be the subgroup of S_{11} generated by $s_1 = (1,10)(2,8)(3,11)(5,7)$ and $s_2 = (1,4,7,6)(2,11,10,9)$.*

Compute the orbit of 1 under the action of G. For every element x in this orbit, give an element g in G such that $g.1 = x$.

Give a generating system for the stabilizer G_1 of 1 in G.

Let $G_{1,2} = G_1 \cap G_2$ be the stabilizer of 1 and 2. Define similarly

$$G_1 \supset G_{1,2} \supset G_{1,2,3} \supset \dots G_{1,2,3,\dots,11} = \{\mathrm{Id}\}.$$

A strong generating set for G is a subset $S \subset G$ that contains generators for all these subgroups.

Let

$$\begin{aligned}
s_3 &= (2,3)(4,5)(6,11)(8,9),\\
s_4 &= (3,5,7,9)(4,8,11,6),\\
s_5 &= (4,6)(5,11)(7,10)(8,9),\\
s_6 &= (4,10,6,7)(5,9,11,8),\\
s_7 &= (4,11,6,5)(7,8,10,9).
\end{aligned}$$

The set $S = \{s_1, \dots, s_7\}$ is a strong generating set (admitted).

Extract a generating set for G_1, $G_{1,2}$, ...

Give coset representatives for the left cosets of G_1 in G. For the left cosets of $G_{1,2}$ in G_1. And so on.

What is the cardinality of G ?

Given a permutation s in $\mathcal{S}_{11}$, how would you check efficiently whether it belongs to G ?

Prove that any permutation group of degree d has a strong generating set consisting of at most $d(d+1)/2$ permutations. Such a small strong generating set is provided by the Schreier-Sims algorithm. See the notes on [7] Alexander Hulpke's home web page.

3 Turing machines

Turing machines are a theoretical model for computers. They are finite automata (they have finitely many inner states), but they can write or read on an infinite tape with a tape head. A Turing machine can be defined by a transition table. For a given inner state and current character read by the head, the transition table provides the next inner state, which character to write on the tape in place of the current one, and how the head should move on the tape (one step left, one step right, or no move at all).

More formally, a Turing machine is a 4-uple $M = (K, \Sigma, \delta, s)$ where K is the finite set of states, $s \in K$ is the initial state, Σ is a finite set of symbols (the alphabet). One assumes that Σ contains two special symbols $\sqcup$ and $\triangleright$, the spacing and the starting symbol. And the transition function is

$$\delta : K \times \Sigma \to (K \cup \{h, "yes", "no"\}) \times \Sigma \times \{\leftarrow, \rightarrow, -\}.$$

The state h is the end state, $"yes"$ is the accepting state, and $"no"$ is the rejecting state. The moving instructions are $\leftarrow$ to move one step on the left, $\rightarrow$ to move one step on the right, and $-$ not to move.

The transition function δ associates to the current state $q \in K$ and to the read symbol $\sigma \in \Sigma$, the triple (p, ρ, D) where p is the next state, ρ is the symbol to be written in place of σ, and D indicates how the head should move.

The head initially stays at the beginning of the tape, and the symbol $\triangleright$ is written there. The input of the algorithme is written just after the symbol $\triangleright$. The transition function must satisfy the following condition : if the symbol $\triangleright$ is read, moving to the left is prohibited. And the symbol $\triangleright$ cannot be erased. This ensures that the head never leaves the tape on the left.

This model is called *one tape Turing machine*. One may also consider Turing machines with several tapes, and one head on each tape.

Question 15 *How fast can a one tape Turing machine copy the input on its right ?*
How fast can a two tape Turing machine copy the input on its right ?
What about sorting ? How fast can a multitape Turing machine merge two sorted lists ?

More on Turing machines can be found in Chapter 2 of [11].

Remind a *decision problem* is a question that must be answered by yes or no : for example, deciding whether an integer is prime. The answer to a *functional problem* is a more general function of the question. For example, factoring an integer is a functional problem. If we want to solve a functional problem with a Turing machine, we write the *input* on the tape, we run the Turing machine, and we wait until it stops. We then read the *output* on the tape. If the machine always stops and returns the correct answer, we say that it solves the problem in question. The time *complexity* is the number of steps before the Turing machine has solved a given problem. Such a Turing machine is said to be *deterministic* because its behavior only depends on the input. The *size* of the input is the number of bits required to encode it. This is the space used on the tape to write this input. For example, the size of an integer is the number of bits in its binary expansion. A problem is said to be *deterministic polynomial time* if there exists a deterministic Turing machine that solves it in time polynomial in the size of the input. The *class* of all functional problems that can be solved in deterministic polynomial time is denoted **FP** or **FPTIME**. The class of deterministic polynomial time decision problems is denoted **P** or **PTIME**.

Question 16 *Is every language $L \subset \{0, 1\}^*$ accepted by a Turing machine ?*

We saw that there exist various models for complexity theory. All the reasonable models lead to equivalent definitions of the polynomial complexity classes. An algorithm can be given a sequence of elementary operations and instructions. Any algorithm can be turned into a Turing machine, but this is fastidious and rather useless since the conceptual description of the algorithm suffices to evaluate the complexity.

For example, we saw that if we want to multiply two positive integers N_1 and N_2 given by their decimal representations, then the size of the input (N_1, N_2) is the number of digits in N_1 and N_2, and this is

$$\lceil \log_{10}(N_1+1)\rceil + \lceil \log_{10}(N_2+1)\rceil.$$

The number of elementary operations required by the elementary school algorithm for multiplication is $\Theta \times \lceil \log_{10}(N_1+1)\rceil \times \lceil \log_{10}(N_2+1)\rceil$. There exists a multitape Turing machine that computes the sum of two integers in that amount of time.

Question 17 *Recall the extended Euclidean algorithm. Give en example. Prove that computes coefficients in Bézout's identity in deterministic polynomial time.*

So addition, subtraction, multiplication, and inversion in the ring $\mathbb{Z}/N\mathbb{Z}$ can be performed in time polynomial in $\log N$. The class $a \bmod N$ in $\mathbb{Z}/N\mathbb{Z}$ is represented by its smallest non-negative element. We denote it $a\%N$. This is the remainder in the Euclidean division of a by N.

4 Exponentiation

A very important problem is exponentiation : given $a \bmod N$ with a in $[0, N[$ and an integer $e \geq 1$, compute $a^e \bmod N$.

Computing a^e then reducing modulo N is not a good idea because a^e might be very large. Another option would be to set $a_1 = a$ and compute $a_k = (a_{k-1} \times a)\,\%N$ for $2 \leq k \leq e$. This requires $e-1$ multiplications and $e-1$ Euclidean divisions. And we never deal with integers bigger than N^2. The complexity of this method is thus $\Theta \times e \times (\log N)^2$ using elementary school algorithms. It is well known, however, that we can do much better. We write the expansion of e in base 2,

$$e = \sum_{0\leq k\leq K} \epsilon_k 2^k,$$

and we set $b_0 = a$ and $b_k = b_{k-1}^2\%N$ for $1 \leq k \leq K$. We then notice that

$$a^e \equiv \prod_{0\leq k\leq K} b_k^{\epsilon_k} \bmod N.$$

So we can compute $(a^e)\%N$ at the expense of $\Theta \times \log e$ multiplications and Euclidean divisions between integers $\leq N^2$. The total number of elementary operations is thus $\Theta \times \log e \times (\log N)^2$ with this method. So exponentiation in $\mathbb{Z}/N\mathbb{Z}$ lies in **FPTIME**. This is an elementary but decisive result in algorithmic number theory. The algorithm above is called *fast exponentiation* and it makes sense in any group. We shall use it many times and in many different contexts. Fast exponentiation admits many variants and improvements [5]. Its first known occurrence dates back to Piṅgala's Chandah-sûtra (before -200). See [4, I,13].

A first interesting consequence is that for p an odd prime and a an integer such that $1 \leq a \leq p-1$, we can compute the Legendre symbol

$$\left(\frac{a}{p}\right) \equiv a^{\frac{p-1}{2}} \bmod p$$

at the expense of $\Theta \times (\log p)^3$ elementary operations. So testing quadratic residues is achieved in polynomial deterministic time.

5 Probabilistic classes

Assume now that we are interested in the following problem :

Given an odd prime integer p, find an integer a such that $1 \leq a \leq p-1$ *and a is not a square modulo p.* $(\star)$

This looks like a very easy problem because half of the nonzero residues modulo p are not squares. So we may just pick a random integer a between 1 and $p-1$ and compute the Legendre symbol $\left(\frac{a}{p}\right) = a^{\frac{p-1}{2}}$. If the symbol is -1 we output a. Otherwise we output FAIL. The probability of success is $1/2$ and failing is not a big problem because we can rerun the algorithm : we just pick another random integer a.

This is a typical example of a *randomized Las Vegas* algorithm. The behavior of the algorithm depends on the input, of course, but also on the result of some random choices. One has to flip coins. A nice model for such an algorithm would be a Turing machine that receives besides the input, a long enough (say infinite) one-dimensional array $\mathcal{R}$ consisting of 0s and 1s. Whenever the machine needs to flip a coin, it looks at the next entry in the array $\mathcal{R}$. So the Turing machine does not need to flip coins : we provide enough random data at the beginning. We assume that the running time of the algorithm is bounded from above in terms of the size of the input only (this upper bound should not depend on the random data $\mathcal{R}$). For *each* input, we ask that the probability (on $\mathcal{R}$) that the Turing machine returns the correct answer is $\geq 1/2$. The random data $\mathcal{R}$ takes values in $\{0,1\}^{\mathbb{N}}$. The measure on this latter set is the limit of the uniform measures on $\{0,1\}^k$ when k tends to infinity. If the Turing machine fails to return the correct answer, it should return FAIL instead.

We have just proved that finding a nonquadratic residue modulo p can be done in Las Vegas probabilistic polynomial time. At present (September 2011) there is no known algorithm that can be proved to solve this problem in deterministic polynomial time. The class of Las Vegas probabilistic polynomial time decision problems is denoted **ZPP**.

Question 18 *How can we pick a random integer in* $[1, p-1]$ *with uniform probability, using a coin ?*

Let E be the set of pairs (x, y) *such that* $x, y \in \mathbb{Z}/p\mathbb{Z}$ *and* $y^2 = x^3 + 1$. *How can we pick a random element in E with uniform probability, using a coin ?*

We now consider another slightly more difficult problem :

Given an odd prime integer p, find a generating set $(g_i)_{1\leq i\leq I}$ *for the cyclic group* $(\mathbb{Z}/p\mathbb{Z})^*$. $(\star\star)$

We have a simple probabilistic algorithm for this problem. We compute an integer I such that

$$\log_2(3\log_2(p-1)) \leq I \leq \log_2(3\log_2(p-1)) + 2$$

and we pick I random integers $(a_i)_{1\leq i\leq I}$ in the interval $[1, p-1]$. The a_i are uniformly distributed and pairwise independent. We set $g_i = a_i \bmod p$, and we show that the $(g_i)_{1\leq i\leq I}$ generate the group $(\mathbb{Z}/p\mathbb{Z})^*$ with probability $\geq 2/3$. Indeed, if they don't, they must all lie in a maximal subgroup of $(\mathbb{Z}/p\mathbb{Z})^*$. The maximal subgroups of $(\mathbb{Z}/p\mathbb{Z})^*$ correspond to prime divisors of $p-1$. Let q be such a prime divisor. The probability that the $(g_i)_{1\leq i\leq I}$ all lie in the subgroup of index q is bounded from above by

$$\frac{1}{q^I} \leq \frac{1}{2^I},$$

so the probability that the $(g_i)_{1\leq i\leq I}$ don't generate $(\mathbb{Z}/p\mathbb{Z})^*$ is bounded from above by 2^{-I} times the number of prime divisors of $q-1$. Since the latter is $\leq \log_2(p-1)$, the probability of failure is

$$\leq \frac{\log_2(p-1)}{2^I},$$

and this $\leq 1/3$ by definition of I.

Note that here we have a new kind of probabilistic algorithm : the answer is correct with probability $\geq 2/3$ but when the algorithm fails, it may return a false answer. Such an algorithm (a Turing machine) is called *Monte Carlo* probabilistic. This is weaker than a Las Vegas algorithm. We just proved that problem ($\star\star$) can be solved in Monte Carlo probabilistic polynomial time. We don't know of any Las Vegas probabilistic polynomial time algorithm for this problem. The class of Monte Carlo probabilistic polynomial time decision problems is denoted **BPP**.

In general, a Monte Carlo algorithm can be turned into a Las Vegas one provided the answer can be checked efficiently, because we then can force the algorithm to admit that it has failed. Note also that if we set

$$I = f + \lceil \log_2(\log_2(p-1)) \rceil,$$

where f is a positive integer ; then the probability of failure in the algorithm above is bounded from above by 2^{-f}. So we can make this probability arbitrarily small at almost no cost.

Références

[1] Gilbert Baumslag. *Topics in combinatorial group theory.* Lectures in Mathematics ETH Zürich. Birkhäuser Verlag, Basel, 1993.

[2] Michael Ben-Or. Probabilistic algorithms in finite fields. In *FOCS*, pages 394–398, 1981.

[3] A. M. Cohen, H. Cuypers, and H. Sterk, editors. *Some Tapas of Computer Algebra*, number 4 in Algorithms and Computation in Mathematics. Springer-Verlag, 1998.

[4] B. Datta and A.N. Singh. *History of Hindu Mathematics.* Motilal Banarsi Das, Lahore, 1935.

[5] Daniel M. Gordon. A survey of fast exponentiation methods. *J. Algorithms*, 27(1) :129–146, 1998. `math.ucalgary.ca/~kwooding/papers/hardware/gordon98survey.pdf`.

[6] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation - international edition (2. ed).* Addison-Wesley, 2003.

[7] Alexander Hulpke. Notes on computational group theory. `http://www.math.colostate.edu/~hulpke/CGT/cgtnotes.pdf`.

[8] D. L. Johnson. *Presentations of groups*, volume 15 of *London Mathematical Society Student Texts.* Cambridge University Press, Cambridge, second edition, 1997.

[9] A. K. Lenstra, H. W. Lenstra, Jr., and L. Lovász. Factoring polynomials with rational coefficients. *Math. Ann.*, 261(4) :515–534, 1982. `http://www.cs.elte.hu/~lovasz/scans/lll.pdf`.

[10] Andrew M. Odlyzko. Discrete logarithms in finite fields and their cryptographic significance. In *EUROCRYPT*, pages 224–314, 1984. `http://www.dtc.umn.edu/~odlyzko/doc/arch/discrete.logs.pdf`.

[11] Christos H. Papadimitriou. *Computational complexity.* Academic Internet Publ., 2007.

[12] Jean-Éric Pin. *Automates finis.* `liafa.jussieu.fr/~jep/PDF/Automates.pdf`.

[13] Carl Pomerance. A tale of two sieves. *Notices Amer. Math. Soc.*, 43(12) :1473–1485, 1996. `http://www.crypto-world.com/FactorPapers.html`.

[14] René Schoof. Counting points on elliptic curves over finite fields. *J. Théor. Nombres Bordeaux*, 7(1) :219–254, 1995. `http://www.mat.uniroma2.it/~schoof/papers.html`.

[15] René Schoof. Four primality testing algorithms. In *Algorithmic number theory : lattices, number fields, curves and cryptography*, volume 44 of *Math. Sci. Res. Inst. Publ.*, pages 101–126. Cambridge Univ. Press, Cambridge, 2008. `http://www.mat.uniroma2.it/~schoof/papers.html`.