3. Computational Complexity.

(A) Introduction.

As we will see, most cryptographic systems derive their supposed security from the presumed inability of any adversary to crack certain (number theoretic) problems. In recent years effort has been put in to formalize the notions of security, and their relation to the theory of computational complexity. Most results in this area are of the type: there exists a provably secure cryptosystem (of some kind) if we can prove that a certain number theoretic problem is hard to solve. Unfortunately, no such problem has ever been shown to be 'hard' to solve, in a sense described below, but progress has been made in the area of classification problems of like difficulty – hierarchies of complexity classes.

Complexity theory may be seen as an attempt to formalize the concept of 'intractable computational problems'. One is naturally led to fundamental problems about the nature of algorithms, machine models and information theory. In these notes we will only pick a few topics that are relevant and more or less mathematical in nature.

The usual dichotomy between 'easy' and 'hard' problems is that between problems allowing a solution in 'polynomial time' and those that do not allow such a solution (taking 'exponential time', which in this context means anything worse than polynomial). Unfortunately, there are not many problems for which it has been *proven* that no polynomial time solution exists; the few for which this has been achieved fall into two categories. The first comprises the problems that are undecidable – that is, those that *no* algorithm will be able to solve (examples are Hilbert's tenth problem, deciding the solvability of polynomial equations in integers, and the problem of deciding whether or not a finitely presented group is trivial) – and the second consists of problems that cannot even be solved with a 'non-deterministic' (see below) computer in polynomial time. Most interesting problems are solvable non-deterministically in polynomial time, but for those no one has been able to show that they cannot possibly be solved in polynomial time.

It should be remarked that even with the right definition of 'polynomial time algorithms' it is not always the case that *in practice* polynomial time algorithms are useful and exponential time algorithms are useless.

We will use the following standard notation: a function f in the real or integer variable x satisfies $f = O(g)$ for a positive function g if there exists an integer C such that $|f(x)| \leq Cg(x)$ for all values in the domain of f, and f satisfies (the stronger) $f = o(g)$ if $f(x)/g(x) \to 0$ for $x \to \infty$. Thus, for example, any polynomial function is $O(x^n)$ for some suitable $n \geq 0$, a constant function is $O(1)$, etc.

A *problem*, generally, consist of a question involving several *parameters* (free variables), and can be specified by describing all parameters as well as the the requirements for a *solution*. An *instance* of a problem is obtained by specifying values for each of the parameters. In the same general way an *algorithm* is a step-by-step procedure for solving a problem. Here 'solving a problem' means that a solution will be found for every instance of the problem.

The most important problem for our purposes is the following.

Problem: Integer Factorization.
Instance: *Positive integer n.*
Question: *Find an integer d such that $1 < d < n$ and $d \mid n$.*

Many cryptographic systems will be rendered 'unsafe' if it could be shown that factoring integers is easy, that is, has a 'polynomial time solution'. The first obvious remark is that the time a solution takes will be dependent on the the input, say the integer n. Thus, *trial division*, the method of factoring n by searching for divisors among $2, 3, \ldots, \lfloor\sqrt{n}\rfloor$, will never take more than $\sqrt{n}$ steps, which is sub-polynomial in n. However, it is not n but the *size* of n which is of importance; usually we take the number of (binary) digits of n as a measure of its size, that is, roughly, $\log(n)$. Suddenly trial division has become an 'exponential' algorithm in the size of the input.

The second important remark is that not all 'steps ' in an algorithm are equally expensive. To underline that we mention the following – at first sight surprising – fact.†

(3.1) Theorem. *There exists an algorithm to find a factor of n in $O(\log n)$ steps.*

Thus in only linear 'time' in the size of the input we can find a factor of a number, so it seems – the problem lies in the definition of 'steps'. In (3.1) it is meant to be *arithmetic steps*, that is, one of the four principal integer operations of addition, subtraction, multiplication, and division (with remainder). The algorithm indicated by (3.1), however, operates on integers potentially as large as $n!$, consisting of order n^2 digits. A fair measure for the complexity of an algorithm is not the number of arithmetic operations, but the number of *bit operations* required. Addition and subtraction require $O(\log n)$, multiplication and division $O((\log n)^2)$ bit operations (or even fewer with *fast multiplication techniques*). With these notations of size and primitive operations both trial division and the algorithm of (3.1) will be exponential time.

(B) Turing Machines.

The standard model for a computer is the Turing machine. A *Turing machine* consists of a finite state control, an (infinite) input tape comprising cells, and a tape head reading and writing a cell at a time. A move is determined by the symbol scanned and the state of the control head, and consists of changing the control and overwriting the current cell, followed by a left or right move. Formally, a program for a Turing machine specifies $(Q, \Sigma, \Gamma, \delta, q_0)$:

Q the finite set of states, including halting states q_h;

Γ the finite set of tape symbols including the blank,

Σ the set of input symbols (not containing the blank) $\Sigma \subset \Gamma$,

δ the (partial) next move function: $\delta\colon (Q \setminus \{q_h\}) \times \Gamma \to Q \times \Gamma \times \{l, r\}$,

q_0 the start state $q_0 \in Q$,

† A. Shamir, *Factoring numbers in $O(\log n)$ arithmetic steps*, Information Processing Letters **8** (1979), 28–31.

A computation starts with the control in state q_0, scanning the first cell, which contains the first of a finite sequence of input symbols x over Σ (called a *word*, and denoted $x \in \Sigma^*$), and proceeds as prescribed by the transition function δ until a halting state is reached. The set Σ is the *alphabet*, and a collection of words in the alphabet (that is, a subset of Σ^*) forms *language*.
There are three essentially different ways in which this can now be used:

(i) for the computation of *partial functions*: the output consists of the (interpretation of) the contents of the finitely many tape cells that contain symbols once the computation has halted, that is, the output is a string $y \in \Sigma^*$. This defines a partial function since the machine is not required to halt on every possible input;

(ii) for the *recognition* of languages: the machine is required to reach a halting state on every possible input, and there will be two halting states, the *accepting halting state* q_{yes} and the *rejecting halting state* q_{no};

(iii) for *accepting* languages: not every computation need to reach a halting state, and the language that is accepted consists of the input words on which the machine reaches a halting state.

We will be mainly interested in the second and third application. *Recursive* languages are those recognized by a computation of the form (ii) above; the language (over Σ) recognized by Turing machine (program) M is

$$L_M = \{x \in \Sigma^* : M \text{ halts on input } x \text{ in state } q_{\text{yes}}\}.$$

Recursively enumerable languages are those accepted by a computation of the form (iii). Since it is easy to change a program in such a way that instead of reaching the state q_{no} it will continue forever, it is clear that every recursive language is recursively enumerable, but, as we shall see below, the converse is not true.

There exists a close relationship between recognizing languages and solving decision problems. A *decision problem* is a problem allowing only yes/no answers; the set $Y_\Pi \subset I_\Pi$ is the subset of the set of all instances of problem Π that consists of all yes-instances.

To obtain an *algorithm* for solving decision problems as above, we need an *encoding function* $e : I \to \Sigma^*$, describing each instance of our problem by an appropriate string of symbols over the alphabet Σ. A Turing machine program M then *solves* problem Π if M halts for all input strings over the alphabet, and $L_M = \{e(i) \in \Sigma^* : i \in Y_\Pi\}$.

We will usually ignore the dependence on the encoding function, and assume that a 'reasonable' encoding exists. The alphabet Σ will usually consist of the binary digits $0, 1$. By way of example we present a solution to the the following problem.

Problem: Divisibility by Four.
Instance: *Positive integer n.*
Question: *Is $n = 4m$ for some integer m?*

A Turing machine program for solving this problem is constructed as follows: $Q = \{q_0, q_1, q_2, q_3, q_{\text{yes}}, q_{\text{no}}\}$, where q_{yes} is the accepting state ('yes, divisible by four') and q_{no} is the rejecting state ('no, not divisible by four'); $\Sigma = \{0, 1\}$, $\Gamma = \{0, 1, b\}$ and δ is specified

as in the table below. Moreover, the encoding function maps any integer into its binary expansion (most significant digit first). This Turing machine recognizes

$$\{x \in \{0,1\}^* \ : \ \text{the two rightmost symbols of } x \text{ are } 0\}.$$

δ	0	1	b
q_0	$(q_0, 0, R)$	$(q_0, 1, R)$	(q_1, b, L)
q_1	(q_2, b, L)	(q_{no}, b, L)	(q_{no}, b, L)
q_2	(q_{yes}, b, L)	(q_{no}, b, L)	(q_{no}, b, L)

Note that we have not been very precise in distinguishing between a Turing machine and a program – in fact the two are more or less the same since a Turing machine may be thought of a a fixed piece of 'hardware' together with a program.

Turing observed that the programs are finite, and can be enumerated, that is, we can encode them and they can serve as input for other Turing machine (programs). In fact, taking encodings for pairs of programs and their input as input for a new Turing machine, we arrive at a Turing machine that simulates all others: a *universal Turing machine* that recognizes pairs of programs and their recognizable inputs.

(3.2) Theorem. *There exists an enumeration of Turing machines $M_1, M_2, \ldots$, and there exists a universal Turing machine M_u with the property:*

$$\forall x \forall M_i \quad (x, M_i) \in L_{M_u} \iff x \in L_{m_i}.$$

We are now in a position to see that not every recursively enumerable language is recursive. The halting problem provides a diagonalization argument.

Problem: Halting Problem.
Instance: *A Turing machine program M, with input x.*
Question: *Does the computation of M on x complete in finitely many steps?*

(3.3) Theorem. *The halting problem is undecidable, that is, there exists no Turing machines that recognizes the pairs M, x that lead to a halting computation.*

Sketch of Proof. Let M_D be a Turing machine that decides the halting the problem, and define M_0 to be the Turing machine accepting the integer i if and only if M_i does not accept it, using M_D to decide this. Then M_0 must different from all others in Turing's list, a contradiction.

The Turing machine we have described so far is *deterministic*: the next move is completely determined by the initial state and the transformation function δ. Strictly speaking, a program for a deterministic Turing machine corresponds to an *algorithm* only if it halts on every possible input string (but we will not always strictly adhere to this).

A *non-deterministic* Turing machine has an extra feature, namely a 'guessing module', with a write-only head, that is solely used in a first stage that precedes the second stage of the computation, which takes place exactly as described above for the deterministic machine. In the guessing stage the guessing module writes an arbitrary string from Γ^* to the left of the beginning of the input string for the second stage. This second, deterministic, 'checking' stage progresses as before, and usually involves scanning the 'guess' during the computation. An *accepting* computation is one that halts, in some halting state q_{halt}, the others are *rejecting*. An input string x is accepted if and only if it is accepted for at least one of the (infinitely many) possible guesses, and the language recognized by the non-deterministic machine is that of all inputs that are accepted.

The intuitive idea behind non-deterministic Turing machines is that they provide a *solution-verifier* rather than a *problem-solver*: given a guessed solution to an instance of the problem, the second stage can deterministically verify that it is a correct solution to the problem.

A non-deterministic 'algorithm' *solves* a decision problem Π if for every yes-instance $y_\Pi \in Y_\Pi \subset I_\Pi$ there exists at least one guessed solution leading to an accepting (halting) computation while for every no-instance $n_\Pi \notin Y_\Pi$ no guess will lead to an accepting computation.

There is one other type of Turing machine that we need to consider, namely a *random (or probabilistic) Turing machine*: this is basically a non-deterministic Turing machine where instead of a guessing module a new operation is built in that can be employed at each step, namely that of a *coin toss*. The result of the computation will in general depend on the tosses made. A language L will be accepted by a random Turing machine if for every word x in L the probability (taken over all possible outcomes over the coin tosses) that x is accepted exceeds a fixed number, say $2/3$. In the next chapter we will come across several important random algorithms; the idea behind probabilistic algorithms is that it is often necessary to find auxiliary elements (integers) satisfying certain properties that are known to hold for 'many' but for which it is difficult to give a deterministic constructive method.

Turing machines can be generalized and restricted in various ways (more tapes, one-sided tape, more read-write heads, restricted jumps etc.) without affecting the class of partial functions that can be computed, or the class of languages that can be recognized or accepted. It is now generally accepted that the intuitive notion of 'effectively computable' is equivalent to 'computable by a Turing machine'.

(C) Complexity Classes.

The *time* $t_M(x)$ taken by a deterministic Turing machine program M on input x is the number of steps taken until a halting state is reached. The *time complexity function* for an always halting Turing machine program M then is the function $T_M : \mathbf{Z}_{\geq 1} \to \mathbf{Z}_{\geq 1}$ given by

$$T_M(n) = \max\{t_M(x) : x \in \Sigma^* \text{ such that } |x| = n\}$$

(where use the notation $|x|$ for the *length* of a string).

The *time* $t'_M(x)$ taken by a non-deterministic Turing machine program M to accept $x \in L_M$ is the minimum number of steps (during guess and check stages) leading to an accepting computation for x. The *time complexity function* for M is defined by

$$T'_M(n) = \max(\{1\} \cup \{t'_M(x) : x \in L_M \text{ such that } |x| = n\}).$$

In particular, rejecting (non-halting!) computations do not affect $T'_M(n)$.

A *polynomial time* deterministic Turing program is one for which there exists a polynomial $P(n)$ such that $T_M(n) = O(P(n))$. The *class* P of languages recognized in (deterministic) polynomial time consists of the languages L such that there exists a polynomial time deterministic Turing program M for which $L = L_M$ (i.e., recognizing L). A decision problem Π will belong to the *complexity class* P if and only if there exists a polynomial time deterministic Turing machine solving Π (under some 'reasonable' encoding scheme).

A *polynomial time* non-deterministic Turing program is one for which $T'_M(n) = O(P(n))$ for some polynomial P. The *class* NP of languages recognized in non-deterministic polynomial time consists of the languages L such that there exists a polynomial time non-deterministic Turing program for which $L = L_M$. A decision problem Π will belong to the *complexity class* NP if and only if there exists a polynomial time non-deterministic Turing machine solving Π. Note that a polynomial bound on a problem in NP imposes a polynomial bound on the size of the solution (since it is to be written out in the guessing stage).

Very briefly, the complexity class BPP of *random (probabilistic) polynomial time* problems consists of those decision problems for which there exists a probabilistic algorithm that solves the problem and for which the running time is bounded by a polynomial in the length of the input. Sometimes one also considers *expected random polynomial time* algorithms, where the expected running time (averaged over all possible coin tosses) is bound by a polynomial function. A problem in BPP will definitely be in BPP.

(3.4) Theorem. *Let Π be a decision problem. Then:*

(i) $\Pi \in \mathrm{P} \Rightarrow \Pi \in \mathrm{NP}$.

(ii) $\Pi \in \mathrm{NP} \Rightarrow T_\Pi(n) = O(2^{P}(n))$ *for some polynomial P.*

Proof. (i) Use the polynomial time algorithm for Π as a checking stage and ignore the guessing stage.

(ii) Let Q be a polynomial bound on T_Π. Then at most $|\Gamma|^{Q(n)}$ guesses need to be considered, and each takes $O(Q(n))$ to verify.

Note one important difference between P and NP: the lack of symmetry between the 'yes' and 'no' cases for NP. If we let co-P be the class of problems whose complement is in P, we immediately find co-P=P, because a deterministic algorithm must halt on every input and the complementary problem is obtained by simply interchanging the output states. It is not clear at all that co-NP=NP.

The most important open question in complexity theory is whether or not P = NP. There are lots of problems that are known to be in NP for which no polynomial time solution is known (see the next section). The most important technique for classifying

problems uses polynomial transformations. A *polynomial transformation* from a language L_1 over Σ_1 to a language L_2 over Σ_2 is a function $f\colon \Sigma_1^* \to \Sigma_2^*$ satisfying

(i) f can be computed by a polynomial time deterministic Turing machine;

(ii) for all $x \in \Sigma_1^*$: $x \in L_1 \iff f(x) \in L_2$.

If there exists a polynomial transformation from L_1 to L_2 we say that L_1 transforms to L_2. It is easy to see (by composing two polynomial time Turing machines) that if L_1 transforms to L_2 and L_2 is in P, then L_1 is in P as well.

It is also not hard to prove (by composing polynomial transformations) *transitivity* of polynomial transformations: if L_1 transforms to L_2 and L_2 transforms to L_3, then there is a polynomial transformation from L_1 to L_3.

For decision problems we define that Π_1 transforms to Π_2 if there exists a polynomial time algorithm providing a function from the instances of Π_1 to those of Π_2 such that the 'yes'-instances correspond under f. A problem Π_1 that transforms to a problem Π_2 in P must be in P as wells; but if Π_1 is 'intractable' then so must be Π_2: Π_2 is at least as hard as Π_1.

Two languages or decision problems are polynomially equivalent if each transforms to the other. The complexity class P is the equivalence class of easiest decision problems, and the class of NP-complete decision problems is the equivalence class of the hardest problems in NP: a (language or) decision problem is called NP-*complete* if it is in NP and every other (language or) decision problem can be transformed to it. If any NP-complete problem allows a polynomial time solution, then every NP problem is in P. Thus, if P $\neq$ NP then NP $-$ P contains the NP-complete problems. The class of NP-*hard* problems contains the problems that have the property that any NP problem can be transformed to it (but which are not necessarily in NP). These cannot be solved in polynomial time, unless P=NP.

The existence of NP-complete problems was proven by Cook's Theorem which states that the *satisfiability problem* is NP-complete.

Problem: Satisfiability
Instance: *Finite set* $U = \{u_1, u_2, \ldots, u_k\}$ *of variables, and a collection* $\mathcal{C}$ *of clauses over* U*, where a clause* $C \in \mathcal{C}$ *is a set of the form* $\{v_1, \ldots, v_m, \bar{w_1}, \ldots \bar{w_n}\}$ *with* $v_i, w_j \in U$.
Question: *Is there a truth assignment* $t : U \to \{\text{true}, \text{false}\}$ *such that* $\mathcal{C}$ *is satisfied, that is, such that for each* $C = \{v_1, \ldots, v_m, \bar{w_1}, \ldots \bar{w_n}\} \in \mathcal{C}$ *there exists at least some* $1 \leq i \leq m$ *for which* $t(v_i) = \text{true}$ *or at least some* $1 \leq j \leq n$ *such that* $t(w_j) = \text{false}$*?*

Once members of the class NP are known, another problem Π may be shown to belong to the class NP-complete by proving that it belongs to NP and that a known member transforms to Π. A long list of NP-complete problems is now known, some of them are listed in the next section.

NP-hard problems are, in the sense of polynomial transformations, at least as hard as NP-complete problems. They often arise in problems that are more general than decision problems, namely in search problems. A *search problem* has the property that for every instance there exists a (possibly empty) set of solutions of that instance, and a solution to the problem requires for every input instance either the a no-state is reached (when there exist no solution to that instance) or a solution for that instance is returned. In corresponding decision problem only the yes-state needs to be arrived at in the second

case. A more general transformation notion (polynomial Turing reduction) is required to deal with such search problems. In practical situations one is often confronted with search problems in the guise of *optimization problems*, in which the requirement for a solution for an instance is that it minimizes a certain cost function, or maximizes a certain profit function. If the cost function is easy to evaluate, a decision problem can easily be associated with the optimization problem that is not harder to solve: the decision problem 'does there exist a solution for which the cost does not exceed B' can be solved if we can solve the optimization problem 'find a solution with minimal cost', when we are permitted to add the calculation of the cost of that solution.

As for the relations between the complexity classes we mention the following. It is not known whether NP $\neq$ co$-$NP; in fact

$$\text{NP} \neq \text{co}-\text{NP} \Rightarrow P \neq \text{NP}.$$

On the other hand, the existence of any NP-complete problem for which the complement is also in NP would imply that NP = co$-$NP. As a consequence any problem Π with both Π and its complement Π^c in NP (so $\Pi \in \text{NP} \cap \text{co}-\text{NP}$) cannot be NP-complete, unless NP=co-NP. If NP and co-NP are different, then so are P and NP, and in that case it is known that there must be intermediate problems between P and NP. Possible candidates would be problems for which both Π and Π^c are in NP: if NP $\neq$ co$-$NP such problems must either be in P or of intermediate difficulty.

Optimization problems are often NP-hard. In particular, it can be shown that if the associated decision problem is NP-complete then the optimization problem will be hard. Also, the complements of NP-complete and NP-hard problems will be NP-hard.

(D) Examples.

One of the most studied combinatorial problems (with practical applications) is the traveling salesman problem. The decision version reads as follows.

Problem: Traveling Salesman
Instance: *Finite set* $C = \{c_1, c_2, \ldots, c_m\}$ *of cities, a distance* $d(c_i, c_j) \in \mathbf{Z}_{\geq 1}$ *for* $c_i, c_j \in C$ *and a bound* $B \in \mathbf{Z}_{\geq 1}$.
Question: *Is there a tour of all cities having length at most* B*, that is, a permutation* $c_{i_1}, \ldots, c_{i_m}$ *such that*

$$d(c_{i_m}, c_{i_1}) + \sum_{j=1}^{m} d(c_{i_j}, c_{i_{j+1}}) \leq B?$$

Traveling Salesman is known to be NP-complete (see also below). The optimization problem can be formulated as follows, and is NP-hard. Interestingly enough it can be shown that in this case the optimization problem is not any harder than the decision problem, in the sense that the optimization problem is Turing reducible to the decision problem.

Problem: Traveling Salesman Optimization
Instance: *Finite set $C = \{c_1, c_2, \ldots, c_m\}$ of cities, a distance $d(c_i, c_j) \in \mathbf{Z}_{\geq 1}$ for $c_i, c_j \in C$ and a bound $B \in \mathbf{Z}_{\geq 1}$.*
Question: *Find a tour of all cities having minimal length.*

Another collection of combinatorial problems are of the knapsack family. These are of interest for cryptographic applications too. The following general form is known to be NP-complete.

Problem: Knapsack.
Instance: *Finite set A, size $s(a) \in \mathbf{Z}_{\geq 1}$ and value $v(a) \in \mathbf{Z}_{\geq 1}$ for each $a \in A$, a size restraint $B \in \mathbf{Z}_{\geq 1}$ and a value goal $G \in \mathbf{Z}_{\geq 1}$.*
Question: *Does there exist a subset $A' \subset A$ such that $\sum_{x \in A'} s(x) \leq B$ and $\sum_{x \in A'} v(x) \geq G$?*

This problem remains NP-complete if size and value are identical, in which case the problem is usually formulated as follows.

Problem: Subset Sum.
Instance: *Finite set A, size $s(a) \in \mathbf{Z}_{\geq 0}$ for each $a \in A$, and a positive integer B.*
Question: *Does there exist a subset $A' \subset A$ such that $\sum_{x \in A'} s(x) = B$?*

A closely related NP-complete problem asks for the partition of a set in parts of equal size.

Problem: Partition.
Instance: *Finite set A, size $s(a) \in \mathbf{Z}_{\geq 0}$ for each $a \in A$.*
Question: *Does there exist a subset $A' \subset A$ such that $\sum_{x \in A'} s(x) = \sum_{x \in A \setminus A'} s(x)$?*

Hamiltonian circuit.
Instance: *A graph G consisting of vertices V and edges E.*
Question: *Does G contain a Hamiltonian circuit, that is, does there exist a sequence $s = [v_1, v_2, \ldots, v_k]$ such that s contains all vertices V exactly once, and such that (v_i, v_{i+1}) is an edge in E for $i = 1, 2, \ldots, k-1$ as well as (v_k, v_1).*

Hamiltonian Circuit is known to be NP-complete. By way of easy example we prove that the Hamiltonian Circuit can be used to prove Traveling Salesman NP-complete, by transforming the former to the latter.

First one has to see that Traveling Salesman is in NP: that is not hard, since all that verifying a proposed solution entails is checking that the tour visits all cities and that the sum of the distances remains below the given bound.

A transformation f from Hamiltonian Circuit can be constructed as follows. Let $G = (V, E)$ be an instance of Hamiltonian Circuit, with $\#V = m$, say. The corresponding instance of Traveling Salesman will consist of a set of cities $C = \{c_1, c_2, \ldots, c_m\}$ and a distance d defined by

$$d(c_i, c_j) = \begin{cases} 1 & \text{if } (v_i, v_j) \in E, \\ 2 & \text{otherwise.} \end{cases}$$

The bound on the length of the tour is defined to be $B = m$.

This transformation is easily computed polynomially, as only at most m^2 pairs of vertices need be looked up in the list of edges (which may for example be arranged as an incidence matrix).

Suppose that G has a Hamiltonian circuit, say $[v_{i_1}, \ldots, v_{i_m}]$; then the tour $[c_{i_1}, \ldots, c_{i_m}]$ has length m, so solves the Traveling Salesman instance. Conversely, any tour $[c_{i_1}, \ldots, c_{i_m}]$ in $f(G)$ of length at most m must have length exactly m and consist of intercity distances all equal to 1, that is, the corresponding sequence $[v_{i_1}, \ldots, v_{i_m}]$ forms a Hamiltonian circuit in G.

Anti-Pellian Equation.
Instance: *An integer d.*
Question: *Does there exist a solution $x, y \in \mathbf{Z}$ for $x^2 - dy^2 = -1$?*

This problem is interesting in several respects. It is known to be in NP, but in general writing down integers x, y solving the equation does *not* provide a solution that can be verified in polynomial time. For example, the smallest solution for $d_k = 5^{2k+1}$ is given by the integers x, y such that $x + y\sqrt{5} = (2 + \sqrt{5})^{5^k}$, and hence the solution x, y cannot be written down in polynomial time. However, it turns out that there exists a criterion that is equivalent to the solvability of $x^2 - dy^2 = -1$, and a non-deterministic polynomial algorithm that will answer 'yes' or 'no' according to whether there exists a solution or not. Note that both 'yes' and 'no' can be verified in polynomial time, and therefore the criterion that is equivalent to the Anti-Pellian Equation problem is one of the few problem known to be in NP $\cap$ co$-$NP.

Exercises.

1. *Let $n = 2^m + 1$ with $m \geq 2$.*

(i) *Show that*

$$n \quad \text{is prime} \qquad \Longleftrightarrow \qquad 3^{\frac{n-1}{2}} \equiv -1 \bmod n.$$

(ii) *Show that the problem of deciding whether or not n (of the above form) is prime is in* P.

(iii) *Prove: n prime $\Rightarrow$ m even.*

(iv) *Write $m = 2^k r$, with r odd. Find a non-trivial factorization of n if $r > 1$.*

(v) *Give an alternative encoding for the problem of deciding whether or not n of the given form is prime that makes the test in part* (i) *exponential instead of polynomial.*

(vi) *Find a close analogue to the test in* (i) *that works for numbers of the form $N = h \cdot 2^m + 1$, with h odd and $m \geq 2$. Formulate it, and show that for prime N the test runs in expected random polynomial time.*

2. *Let it be given that the Partition problem is* NP*-complete.*

(i) Prove that the Knapsack is NP*-complete (by showing that Partition can be obtained from it as a special case).*

(ii) *Prove (by a polynomial transformation) that Subset Sum is* NP*-complete.*

4. Four Number Theoretic Problems.

In this Chapter we will briefly discuss four problems in number theory that have cryptographic significance, together with their complexity status.

Some quotes to indicate how the perception of the status of some of these problems has changed, partly due to cryptography:

> *The problem of distinguishing prime numbers from composite numbers and of resolving the latter into their prime factors is known to be one of the most important and useful in arithmetic. It has engaged the industry and wisdom of ancient and modern geometers to such an extent that it would be superfluous to discuss the problem at length. [...] The dignity of science seems to demand that every aid to the solution of such an elegant and celebrated problem be zealously cultivated.* (C. F. Gauß, Disquisitiones Arithmeticae, 1801).

> *[the problem] is almost devoid of application. [...] I shall be surprised if anyone regularly factors numbers of size* 10^{80} *without special form during the present century.* (R. K. Guy, 'How to factor a number', 1975).

> *Until recently the factorization of large integers was not considered a decent subject for mathematicians, but the advent of the RSA-cryptosystem has increased its interest.* (M. Voorhoeve, 'Factorization algorithms of exponential order', 1980).

(A) Primality Proving.

The first observation is that primes can be recognized in non-deterministic polynomial time.

Problem: Prime Numbers.
Instance: *Positive integer* N.
Question: *Do there not exist* $m, n \in \mathbf{Z}_1$ *such that* $N = mn$*?*

(4.1) Theorem. *'Prime Numbers' is in* NP.

Proof. Let N be a positive integer, and suppose that

$$(*) \qquad N - 1 = 2^{\alpha} p_1^{\alpha_1} \cdots p_k^{\alpha_k}.$$

To prove that N is prime, it suffices to write down an integer $a < N$, and integers $p_0 = 2$, $p_1, \ldots, p_k$, to verify that $(*)$ holds and that the order of a modulo N is $N - 1$ by showing

$$a^{(n-1)/2} \equiv -1 \bmod N$$

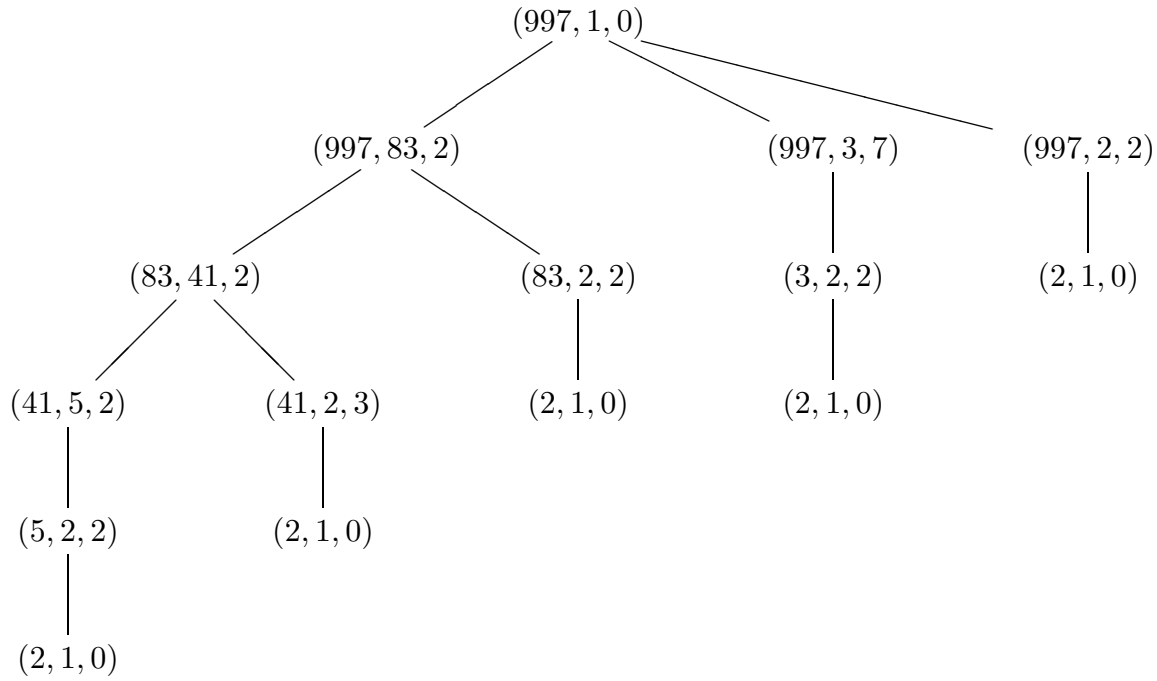
and

$$a^{(n-1)/p_i} \not\equiv 1 \bmod N \qquad \text{for } i = 1, \ldots, k.$$

Proving recursively that each of the p_i is prime, leads to a polynomial bound on the primality proof of N (a bound $O((\log n)^4)$ can be achieved).

(4.2) Example The above result, due to Pratt is often stated in the form 'short certificates for primes exist'. Here is such a certificate for $N = 997$.

In this tree, each node consists of a triple (n, p, a), one for each of the prime divisors p of $n - 1$, where a is as in the proof above.



The complement of the problem Prime Numbers is called Composite Numbers.

Problem: Composite Numbers.
Instance: *Positive integer N.*
Question: *Are there $m, n \in \mathbf{Z}_1$ such that $N = mn$?*

This problem is obviously in NP, since verification of a proof for compositeness may consist of a single multiplication of two non-trivial divisors and comparison with N. Hence 'Prime Numbers' is one of the few known problems in $\mathrm{NP} \cap co - \mathrm{NP}$.

For the next results it is convenient to introduce the notion of witness. For an odd integer $N > 2$, an integer a is *witness to the compositeness of N* if, with $N - 1 = 2^s d$ (and d odd):

(i) N does not divide a; and

(ii) $a^d \not\equiv 1 \bmod N$; and

(iii) $a^{d2^r} \not\equiv -1 \bmod N$, for $r = 0, 1, \ldots, s - 1$.

(4.3) Lemma. *If a is a witness for N then N is composite.*

Let a be a witness for N and suppose that N is prime, then $a^{N-1} \equiv 1 \bmod N$, and in the sequence

$$a^d, a^{d\cdot 2}, a^{d\cdot 2^2}, \ldots, a^{d\cdot 2^{s-1}}, a^{N-1}$$

the last term is $1 \bmod N$, but the first term is not, by (i). There exists some k with $0 \leq k < s$ such that $b = a^{d\cdot 2^k} \not\equiv 1 \bmod N$ but $b^2 \equiv a^{d\cdot 2^{k+1}} \equiv 1 \bmod N$. By (iii) $b \not\equiv -1 \bmod N$ so we have found a square root of 1 in the field $\mathbf{Z}/N\mathbf{Z}$ that is not ± 1: a contradiction.

(4.4) Theorem. *'Composite Numbers' is in* BPP. *That is, there is an algorithm that recognizes composite numbers in random polynomial time.*

Proof. It can be shown that for every odd composite N at least $3 \cdot (N-1)/4$ integers a with $1 \leq a < N$ are witness to the compositeness for N.

(4.5) Theorem. *Assuming the generalized Riemann hypothesis, 'Prime Numbers' is in* P. *That is, under the assumption of GRH, primality can be decided in deterministic polynomial time.*

Proof. Under GRH a witness less than $c \cdot (\log N)^2$ exists for every positive odd composite number N. This is a consequence of the fact that all non-witnesses are contained in the subgroup

$$\{a \in (\mathbf{Z}/N\mathbf{Z})^* : a^m = \pm 1\},$$

for some m (in fact $m = d \cdot 2^j$ with $j = \max\{i : b^{2^i} \equiv -1 \bmod N \text{ for some } b \in \mathbf{Z}/N\mathbf{Z}\}$), and the result that should have been quoted in Chapter 2:

Theorem. *Let m be a positive integer. Assuming the generalized Riemann hypothesis, the smallest positive integer x outside a given subgroup G of $(\mathbf{Z}/m\mathbf{Z})^*$ satisfies $x \leq 2(\log m)^2$.*

(4.6) Theorem. *'Prime Numbers' is in* BPP. *That is, there is an algorithm that recognizes prime numbers in random polynomial time.*

(4.7) Theorem. *There exists an algorithm that, for some constant $c \in \mathbf{Z}_{\geq 0}$, recognizes prime numbers in $(\log N)^{c \log\log\log N}$*

(4.8) Remarks. Theorem (4.6) is based on Adleman-Huang's generalization to 'abelian varieties' of Goldwasser-Kilian's 'random elliptic curve test'. Neither of these is practical, and the Theorem is only of theoretical value (it seems that not even the degree of the polynomial has been estimated).

Theorem (4.7) on the other hand is based on the highly practical 'Jacobi sum test' (albeit that the practical version is probabilistic). It can routinely prove primes of hundreds of decimal digits prime.

There is one other test based on elliptic curves that has similar performance. Although a rigorous complexity analysis has never been completed, there are heuristics indicating that it may run in expected polynomial time.

In any case, for practical purposes the primality problem is often regarded as 'easy'.

Some other miscellaneous results. First an analogue to the result mentioned in (3.1): if n is prime, this can be proved in $O(1)$ arithmetic operations. This is due to the fact that, as a byproduct of research into Hilbert's tenth problem it was shown that there exists a polynomial $f \in \mathbf{Z}[a, b, c, \ldots, x, y, z]$ of degree 25 with the property that the set of prime numbers coincides with the positive values taken on by the polynomial when evaluated in non-negative integers. Moreover, the result is constructive in the sense that non-negative integers $A, \ldots, Z$ can be constructed such that $n = f(A, \ldots, Z)$, and, in fact, it then takes at most 87 multiplications and additions to verify this. However, it can also be shown that the largest of $A, \ldots, Z$ exceeds

$$n^{n^{n^{n}}}.$$

Using elliptic curves (again) it is also possible to give prime certificates, the verification of which can even shown to be done in $O((\log n)^3)$ whereas the bound proven for verification of the above certificates is $O((\log n)^4)$.

(B) Factorization.

The situation regarding the factorization problem is much less clear.

Problem: Integer Factorization.
Instance: *Positive integer N.*
Question: *Find $m, n \in \mathbf{Z}_1$ such that $N = mn$*

Here are some proven results. Whenever in the exponent an ϵ appears, this indicates that there are logarithmic factors involved as well, but for any positive ϵ these are in $O(N^{\epsilon})$. The deterministic 'trial division algorithm' for example can find the smallest prime factor of N in $O(N^{\frac{1}{2}}(\log n)^2)$, which is abbreviated to $O(N^{\frac{1}{2}+\epsilon})$.

(4.9) Theorem. *There exists a deterministic algorithm that factors N completely in $O(N^{\frac{1}{4}+\epsilon})$.*

(4.10) Theorem. *There is a deterministic algorithm that, assuming the generalized Riemann hypothesis, factors N completely in $O(N^{\frac{1}{5}+\epsilon})$.*

Almost all non-exponential results, either heuristic or proven, involve the function

$$L[u, v](n) = \exp^{(v+o(1))(\log n)^u (\log\log n)^{(1-u)}}.$$

Note that $L[0, v](n) = (\log n)^v$ and $L[1, v](n) = n^v$ (up to the $o(1)$).

(4.11) Theorem. *There exists a probabilistic algorithm that completely factors N in expected time $L[\frac{1}{2}, 1](N)$.*

(4.12) Remarks. The algorithm that (4.9) refers to is 'Pollard-Strassen', and uses in fact time $(\log n)^{O(1)}\sqrt{p}$ to split n where p is the least prime factor of n. (It uses Fourier transforms to evaluate a polynomial through many points, and is only of theoretical interest).

The algorithm in (4.10) is based on class groups, and the bound was proven by Rene Schoof; it is not faster if small primes occur.

The theorem in (4.11) is the best that has been proven, and refers to a combined class group / elliptic curve method, or Seyssen's class group algorithm. Not faster if small primes are present.

Neither of these are practical.

The rest of this section will be devoted to factorization methods that are practical, and of interest to us for one reason or other.

We start with Pollard's ρ method, which of interest because a variant will appear in the next section.

Pollard's ρ method is based on the fact that a random sequence of elements from a set of cardinality t is expected to have a collision (contain two identical elements) among $O(\sqrt{t})$ entries. If one chooses a random function $f : \mathbf{Z}/N\mathbf{Z} \to \mathbf{Z}/N\mathbf{Z}$, and an initial value x_1, the sequence $x_1, x_2 = f(x_1), x_3 = f(f(x_1)) = f(x_2), \ldots$ will behave randomly when taken modulo a (prime) divisor p of N, that is, on $\mathbf{Z}/p\mathbf{Z}$. A collision consisting of indices $j > i \geq 1$ for which $x_i \equiv x_j \bmod p$ is expected after $O(\sqrt{p})$ steps. Since it will be unlikely (in particular if $p \ll N$) that $x_i \equiv x_j \bmod N$, such a collision will enable us to detect the factor p by computing $\gcd(x_i - x_j, N)$.

An easy to compute, and apparently sufficiently random function is obtained by taking $f(x) = x^2 + c \bmod N$, where $c = 1$ and $x_1 = 2$ are the standard choices. Instead of comparing any two entries x_i, x_j, the same result can be achieved more efficiently by noting that the sequence will become periodic after say $s + t$ steps (so that $x_{s+1} \equiv x_{s+t+1} \bmod p$, and the '$\rho$' has a 'tail' of length s and a 'cycle' of length t); then for some m one gets $x_{2m} \equiv x_m \bmod p$, the least such m being the smallest multiple of t exceeding s.

Pollard's ρ method runs heuristically in expected time essentially $\sqrt{p}$ to find the prime factor p – that is $O(n^{\frac{1}{4}})$.

The second practical method, which is also fast, is the so-called 'elliptic curve method' of H. W. Lenstra. We will deal with it later, but it is very similar to Pollard's $p-1$ method, which we will now describe. Pollard's $p-1$ method works well if N contains a prime factor p for which $p-1$ is 'smooth', that is, built up entirely from powers of small primes. More precisely, a number x is y-smooth if all its prime divisors are less than or equal to y.

Here the L function comes in again: a number $x \leq n^\alpha$ is $L[1/2, \beta](n)$-smooth with probability $L[1/2, -\alpha/(2\beta)](n)$.

If N does contain a y-smooth factor we may detect it (if not all factors are y-smooth) by choosing a random $a \in \mathbf{Z}/N\mathbf{Z}$ and raising it in the power $k(y)$ modulo N, for $k(y)$ consisting of powers of the primes up to y; then $a^{k(y)} \equiv 1 \bmod p$ and so $1 < \gcd(a^{k(y)} - 1, N)$. It remains to choose k as a function of y; sometimes $k = \mathrm{lcm}_{p^k \leq y}(p^k)$ is taken, and y is determined empirically.

Lenstra's elliptic curve method uses groups of points on elliptic curves that need to be smooth, and runs in $O((\log n)^2 L[\sqrt{2}, 1](p)) = O(L[1/2, 1](n)$ heuristically, where p is the smallest prime factor of n.

(4.13) Example. As a quick example we show how quickly $N = 9495468075263$ can be factored with straightforward implementations of Pollard's ρ and $p-1$ methods in the Magma language. The ρ method finds the factor 823 after 31 iterations in a fraction of a second (because it is so small), and the $p-1$ method finds 209497 in about the same time because $p-1$ it is so smooth: $209497 - 1 = 2^3 \cdot 3 \cdot 7 \cdot 29 \cdot 43$. In fact $N = 823 \cdot 209497 \cdot 55073$, while $823 - 1 = 2 \cdot 3 \cdot 137$, and $55073 - 1 = 2^5 \cdot 1721$.

The final two (related) methods concern the best-known practical methods to find factors of large numbers without small divisors (say products of primes with more than 40 decimal digits).

Both the *quadratic sieve* method and the *number field sieve* are members of a family of 'relation method' algorithms, that have applications in other areas (including discrete logarithms). In its most general form, an algorithm in this family can be described in three staps in the following general way.

(i) Generate a large number of random 'relations' of some desired form over a certain 'factor base';

(ii) Use linear algebra to reduce the large system of relations to a few;

(iii) Try to derive the desired conclusion.

In the case of factorization, one chooses a bound b, attempts to find many relations of the form

$$m^2 \equiv r_m = \prod_{p_i \leq b} p_i^{k_{m,i}} \bmod N$$

and combines these to obtain a few relations of the form

$$x^2 = \prod_{m \in M} m^2 \equiv \prod_{m \in M} \left(\prod p_i^{k_{m,i}} \right)^2 = y^2 \bmod N,$$

in the hope that $x^2 - y^2 = (x+y)(x-y) \equiv 0 \bmod N$ will give rise to a non-trivial factorization of N.

More precisely, one generates $L[1/2, b]$ integers m for which the least positive residue r_m of m^2 modulo N is $L[1/2, b]$ smooth. (The factor base consists of the primes up to b.) One expects that there will be a linear dependency among the vectors of zeroes and ones of $k_{m,i} \bmod 2$ (telling whether p_i occurs an even or odd number of times in r_m), and such a dependency will lead to a product $\prod_{m \in M} r_m = y^2$ that is an even (including 0) power of all the primes less than b, so the corresponding product x^2 of the m^2 gives a relation of the form $x^2 \equiv y^2 \bmod N$. It can in fact be proven that for composite N that is not a prime power and that is free of factors less than $L[1/2, b]$ with probability at least $1/2$ the $\gcd(x+y, N)$ will yield a factor of N.

There are several ways to generate the necessary relations: by choosing m at random (not very practical, but one can prove something about it), by using the numerator n_i and

denominator d_i of the convergents of the continued fraction expansion of $\sqrt{N}$ (which have the advantage that $r_i = |n_i^2 - nd_i^2| < 2\sqrt{N}$), or by using certain quadratic polynomials. In the latter case one uses polynomials like

$$P(x) = (\lfloor \sqrt{N} \rfloor + X)^2 - N,$$

which, evaluated for any integer m, yields a square modulo N, whose residue can be used as r_m above (when it is smooth). Now $|r_m|$ is $O(L[1/2, \alpha](N)\sqrt{N})$ if $m \leq L[1/2, \alpha](N)$, so if random it should be $L[1/2, b]$ smooth with probability $L[1/2, -1/(4b)]$. so we should take $\alpha \geq b + 1/(4b)$.

One can use the elliptic curve method as smoothness tester! But for quadratic sieve also 'sieving': let p be a prime in the factor base, then, provided $\left(\frac{N}{p}\right) = 1$, the equation $R(x) \equiv 0 \bmod p$ has 2 solutions modulo p (that can be found – see last section of this chapter!), say m_1 and m_2, but then $R(m_i + kp) \equiv 0 \bmod p$ for any integer k.

Pomerance's multiple polynomial variation of the quadratic sieve, runs heuristically in $O(L[1/2, 1](n)$.

The number field sieve, which uses two factor bases (one as above, and one in a number field) runs heuristically in $O(L[1/3, \sqrt[3]{\frac{64}{9}}](n))$.

With these methods it is still a major effort to factor arbitrary integers with more than 100 decimal digits.

(C) Discrete Logarithms.

The third important problem we consider is that of finding discrete logarithms. Let h be an element of a finite abelian group G, and let $H =< h >$ be the subgroup generated by h. For an element $g \in G$ the *discrete logarithm problem with respect to* h is to decide whether or not $g \in H$, and if $g \in H$ to compute the discrete logarithm $\log_h g$, which is an integer $\log_h g = m \in \mathbf{Z}_{\geq 0}$ such that $g = h^m$.

Problem: Discrete Logarithm.
Instance: *Finite abelian group G, elements $g, h \in G$.*
Question: *Decide whether $g = h^m$ for some $m \in \mathbf{Z}_{\geq 0}$, and if so, find $m = \log_h g$.*

A few remarks are in order. In the first place, it will be assumed that the group operations in G can be performed efficiently; it will, however, not always be the case that group elements have a unique representation, in which case equality testing and membership testing may not be possible. When this occurs we will see that un-equality testing will be easy and, in fact, will be sufficient. Examples of groups in which we can efficiently compute are: groups $(\mathbf{Z}/n\mathbf{Z})^*$, multiplicative groups $\mathbf{F}_q^*$ of finite fields, groups of points on elliptic curves over finite fields. Note that the representation of group elements is important for the discrete logarithm problem: in the additive group $\mathbf{Z}/n\mathbf{Z}$ the problem is trivial, and any finite cyclic group is isomorphic to some $\mathbf{Z}/n\mathbf{Z}$. In other words, an alternative formulation of the discrete logarithm problem is to find the image of g under the isomorphism between H and $\mathbf{Z}/n\mathbf{Z}$, if $n = n_H = \#H$.

Secondly, it may not be the case that the order of the group G or of the subgroup H is known; sometimes we will have to assume that a small multiple of the order is known though. In fact 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 of H.

Thirdly, an important special case arises when $H = G$, that is, when G is cyclic and h is a generator; the decision question will be trivial then.

If $\#H$ is not too big and many logarithms will have to be computed, it may be worthwhile to compute a complete table of logarithms. In general this will of course not be feasible.

The trivial discrete logarithm algorithm proceeds by computing $1 = h^0, h = h^1, h^2, \ldots$ 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. We do need unique representation of group elements. The following improves significantly on this.

(4.14) Theorem. *There exists a deterministic algorithm to find discrete logarithms that takes $O(\sqrt{n_H}\log n_H)$ multiplications and comparisons in G.*

Again, we need to be able to test for equality; also, we require considerable storage, namely for $O(\sqrt{n_H})$ elements.

The method, known as *Shanks's baby-step giant-step* algorithm, is quite practical, and proceeds as follows.

First one needs an upper bound B on the logarithm $\log_h g$; if n_H is known then $B = n_H$, and if no upper bound is known one may use this method first to determine n_H. 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, $\log_h g = ib + j$. So, one next computes a sorted (or hashed) lookup table of h^j for $j = 0, 1, \ldots, b-1$ which takes $O(b \log b)$ group operations. Next compute $g \cdot h^{-ib}$ for $i = 0, 1, \ldots, b-1$ 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 $i = 0 = j$); if no upper bound on n_H is known, one just tries $B = 2^1, 2^2, 2^3, \ldots$ in succession.

The following variant of *Pollard's ρ method* runs heuristically in expected time $O(\sqrt{n_H})$. One partitions G in three (random) subsets G_1, G_2, G_3 of roughly equal size; we need n_G (or a small multiple) and membership testing in G_i, $i = 1, 2, 3$ for this. Define the recurrent sequence $g_0, g_1, \ldots$ by $g_0 = g$ and

$$g_{i+1} = \begin{cases} h \cdot g_i & \text{if } g_i \in G_1; \\ g_i^2 & \text{if } g_i \in G_2; \\ g \cdot g_i & \text{if } g_i \in G_3; \end{cases}$$

for $i \geq 0$. As in the original ρ algorithm, we expect a collision after $O(\sqrt{n_G})$ steps if this sequence is random in G. Again, we also expect $g_{2r} = g_r$ for r that is $O(\sqrt{n_G})$. Keeping track of the integers a_i and b_i, defined modulo n_G, such that $g_i = g^{a_i}h^{b_i}$ (so $a_0 = 1$, $b_0 = 0$), we then find

$$g_{2r} = g^{a_{2r}}h^{b_{2r}} = g^{a_r}h^{b_r} = g_r,$$

so $g^e = h^f$, with $e \equiv a_{2r} - a_r \bmod n_G$ and $f \equiv b_r - b_{2r} \bmod n_G$. Then $e \log_h g \equiv f \bmod n_G$. If $d = \gcd(e, n_G) = 1$ this determines $\log_h g$ modulo n_G; if not, there are d possibilities left, and it may require it little bit extra work to determine $\log_h g$.

Note that the recurrent sequence remains entirely in H if $g \in H$, and if the sequence behaves randomly in H we expect to find a collision in $O(\sqrt{n_H})$ steps.

The next algorithm (often attributed to Pohlig-Hellman and to Silver) works if the order of H is y-smooth, for some small $y \in \mathbf{Z}_{\geq 1}$. If a bound B on the order is also known, smoothness can easily be tested, for example (much like in the $p-1$ method) by raising h in the power $k = p_1^{k_1} \cdots p_t^{k_t}$ which is the product of all prime powers $p^k < B$ with $p < y$.

Suppose that $n_H = \prod p_1^{\alpha_1} \cdots p_m^{\alpha_m}$; then $\log_h g$ is determined by $\log_h g \bmod p_i^{\alpha_i}$, $i = 1, \ldots, m$, so if $g \in H$ we need to determine $e = \log_h g \bmod p^\alpha$ only, for various prime powers, and use the Chinese remainder Theorem. Let $e = e_0 + e_1 p + \cdots + e_{\alpha-1} p^{\alpha-1}$ with $0 \leq e_i < p$. We determine e_0 by using

$$g^{n_H/p} = (h^e)^{n_H/p} = (h^{n_H/p})^e = (h^{n_H/p})^{e_0} = k^{e_0},$$

that is, by determining $\log_k(g^{n_H/p})$ in the subgroup $K =< k >$ of H of order p. Then

$$(g \cdot h^{-e_0})^{n_H/p^2} = k^{e_1}$$

so we find e_1 again by a discrete logarithm computation in K (using for example the ρ-method above). If $g \in H$ then the discrete logarithm can be found in $O(\sum \alpha_i (\log n_H + \sqrt{p_i} \log p_i))$ group operations.

Finally we come to the subexponential methods, which are of the 'relation type' outlined above, and usually called *index calculus methods* in this case. These methods work in groups for which a smoothness concept makes sense, but they are usually confined in practice to finite fields. In that case one assumes that a primitive element h is given, and the order $\#H = \#G$ is known.

(4.15) Theorem. *Let $\mathbf{F}_q$ be a finite field, then there exist probabilistic algorithms to find discrete logarithms in $\mathbf{F}_q$ that have expected running time*

$$T = \begin{cases} L[1/2, \sqrt{2}](q) & \text{if } q = 2^m; \\ L[1/2, \sqrt{2}](q) & \text{if } q = p; \\ L[1/2, c(m)](q) & \text{if } q = p^m, \text{ for } m \text{ fixed;} \end{cases}$$

(4.16) Remarks. Heuristically, algorithms running in time $L[1/3, c](q)$ exist in all three cases – for $q = 2^m$ a result due to Coppersmith, for $q = p$ and $q = p^m$ with m fixed due to use of the number field sieve.

An open problem is still to find an algorithm that runs (even heuristically) in subexponential time if *both* p and m tend to infinity.

In the first stage of the index calculus method we find discrete logarithms of all elements $s \in S \subset G$, in the factor base S consisting of y-smooth 'prime' elements p_i of G. This is done by generating y-smooth g^e at random: $g^e = \prod p_i^{\alpha_i}$; each e gives rise to a linear relation $e = \sum_{i=1}^{k} \alpha_i \log_h p_i \bmod n_H$. After generating enough relations. one expects a solution for the unknowns $\log_h p_i$, one finds that by linear algebra.

To find a specific discrete logarithm, one picks random integers f until gh^f is y-smooth, then

$$\log_h g = \sum(\beta_i \log_h p_i) - f \bmod n_H.$$

We end this section with some relations between the factorization problem and the discrete logarithm problem in $(\mathbf{Z}/N\mathbf{Z})^*$.

(4.17) Theorem. *If 'Discrete Logarithm' for $(\mathbf{Z}/N\mathbf{Z})^*$ is in* P *then 'Integer Factorization' is in* BPP*; that is, a deterministic polynomial time algorithm for finding discrete logarithms in $(\mathbf{Z}/N\mathbf{Z})^*$ provides a probabilistic polynomial time method for factoring N.*

Proof. We sketch a proof.

First we use the relation between discrete logarithms and orders of elements as follows: suppose $b \in (\mathbf{Z}/N\mathbf{Z})^*$ is given, then we find a small multiple of the order of $b \in (\mathbf{Z}/N\mathbf{Z})^*$ using the discrete log algorithm in the following manner. Choose a small prime p and attempt to compute $\log_h(b^p)$ and $\log_h b$. If p is coprime to $\phi(N)$ this will produce y such that $b^{py} \equiv b \bmod N$, if $\gcd(p, \phi(N)) \neq 1$ we proceed to the next prime (a suitable prime less than $\log N$ must exist), until y is obtained. Then $x = py - 1 = 2^s t$ (with t odd) is a multiple of the order of b.

Secondly, let $N = p_1^{\alpha_1} \cdots p_k^{\alpha_k}$ be odd and let $\lambda = 2^\sigma \mu$ be the exponent $\gcd_{p_i|N}(\phi(p_i^{\alpha_i}))$ of $(\mathbf{Z}/N\mathbf{Z})^*$. Suppose that b is not in the subgroup

$$A = \{x \in (\mathbf{Z}/N\mathbf{Z})^* : x^{\lambda/2} \equiv \pm 1 \bmod N\}$$

of $(\mathbf{Z}/N\mathbf{Z})^*$. Then the order of $b \in (\mathbf{Z}/N\mathbf{Z})^*$ must be of the form $2^\sigma \nu$, for a divisor ν of μ; also $s \geq \sigma$. In the sequence

$$b^t, b^{t \cdot 2}, b^{t \cdot 2^2}, \ldots, b^{t \cdot 2^\sigma}$$

the last term is $\equiv 1 \bmod N$ but (by choice of b) the next to last term is not $\equiv \pm 1 \bmod N$. Therefore $\gcd(N, b^{t \cdot 2^{\sigma-1}} - 1)$ is non-trivial.

Finally, if N is not a prime power, the subgroup A of $(\mathbf{Z}/N\mathbf{Z})^*$ is proper, as we can choose

$$a \equiv \begin{cases} g_i \bmod p_i^{\alpha_i} & \text{for } i = 1; \\ g_i^2 \bmod p_i^{\alpha_i} & \text{for } i > 1 \end{cases}.$$

Prime powers can be recognized in random polynomial time (see Exercise 4).

(4.18) Theorem. *If the integer factorization problem and the problem of finding discrete logarithms in $(\mathbf{Z}/p\mathbf{Z})^*$ are both in* P*, then so is the problem of finding discrete logarithms in $(\mathbf{Z}/p\mathbf{Z})^*$.*

For a sketch of the proof see Exercise 5.

(D) Square Roots.

To conclude this chapter we consider the problem of finding square roots in $(\mathbf{Z}/m\mathbf{Z})^*$.

Problem: Modular Square Roots.
Instance: *Positive integer m, integer a.*
Question: *Find all square roots of a modulo m.*

First consider the case where the modulus $m = p$ is prime. The Legendre symbol provides an efficient means for finding the number of square roots of a modulo p. If $\left(\frac{a}{p}\right) = 0$ then 0 is the only square root, and if $\left(\frac{a}{p}\right) = -1$ then square roots do not exist modulo p. If a is a quadratic residue modulo p, the problem is to find one square root x such that $x^2 \equiv a \bmod p$ (the other square root is then simply $-x$).

Obviously, a search will succeed in time $O(p)$. But we can do much better.

(4.19) Theorem. *There exists a deterministic algorithm to find square roots modulo p that runs in $O((\log p)^6)$.*

The algorithm to which this alludes is Schoof's algorithm, and uses elliptic curves. We will consider it in a later Chapter. Its practicality is not entirely clear.

We next consider probabilistic methods. If $p \equiv 3 \bmod 4$ one can in fact write down the solution explicitly, namely, let

$$x = a^{(p+1)/4} \bmod p,$$

then, by Euler's Criterion (2.13)(i)

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

In half the remaining cases (namely if $p \equiv 5 \bmod 8$, see Exercise 6) one can still write down an explicit solution, but to push this further becomes cumbersome. However, there exists a probabilistic solution in general.

(4.20) Theorem. *There exists a probabilistic algorithm to find square roots modulo p that runs in expected polynomial time.*

In fact, as we will see, the only non-deterministic part will consist of finding a quadratic non-residue modulo p; thus, as an immediate result we find a polynomial time algorithm under the assumption of GRH (but (4.19) is unconditional).

The algorithm underlying (4.17) is called Tonelli-Shanks, and works as follows. First write $p - 1 = 2^s \cdot d$, with d odd. Next find a quadratic non-residue c modulo p (in expected

polynomial time, the rest of the algorithm will be deterministic). Determine $z \equiv c^d \cdot p$. Since c was a quadratic non-residue, this z will generate the (2-Sylow) subgroup H of order 2^s of $(\mathbf{Z}/p\mathbf{Z})^*$. The element a^d will be a square in H, so $a^d = z^{2m}$ for some $m \leq 2^{s-1}$, and then $a^{(d+1)/2}z^{-m}$ is the desired root.

We now determine a strictly decreasing sequence of integers $r_0 = s > r_1 > \cdots > r_k = 0$ and the corresponding chain of subgroups $H_0 = H \supset H_1 \supset \cdots \supset H_k = 1$ of order 2^{r_i}, together with generators h_i and auxiliary elements x_i, $b_i \in H_i$, with b_i a square in H_i, in such a way that $ab_i = x_i^2$. This is done by taking initially $r_0 = s$, $H_0 = H$, $h_0 = z^{-1}$, $x_0 = a^{(d+1)/2}$ and $b_0 = a^d$, and by using the following rules to go from i to $i+1$: let r_{i+1} be the least positive integer such that $b_i^{2^r} \equiv 1 \bmod p$, let $h_{i+1} = h_i^{2^{r_i - r_{i+1}}}$, let $x_{i+1} = x_i \cdot h_i^{2^{r_i - r_{i+1} - 1}} = x_i\sqrt{h_{i+1}}$ and $b_{i+1} = b_i \cdot h_{i+1}$. This will terminate with $b_k = 1$ and then $a = x^2$.

That settles the problem for prime modulus in an entirely satisfactory way. For composite moduli we have the following.

(4.21) Theorem. *'Modular Square Roots' is in* P *if and only if 'Integer Factorization' is in* P*.*

Proof. Let N be composite, and let $a = d^2$. If x is a square root of a such that $x \not\equiv \pm d \bmod N$, then $N | x^2 - d^2$ but $N \not| x - d$ and $N \not| x + d$ so $1 < \gcd(x - d, N) < N$.

The converse is an immediate consequence of (4.19) and the fact that a square root modulo p can easily be lifted to a square root modulo p^k, for any $k \geq 1$.

To conclude this Chapter we mention some peculiarities that are sometimes used in cryptography.

Let $N = p \cdot q$, with $p, q \equiv 3 \bmod 4$; such N are sometimes called *Blum numbers*. An integer a coprime to N will be a square modulo N if and only if it is a square modulo both p and q, that is, if and only if $\left(\frac{a}{p}\right) = \left(\frac{a}{q}\right) = 1$. Let x_0 be a square root of a modulo p; note that the other square root is $-x_0$ and that $\left(\frac{-x_0}{p}\right) = \left(\frac{-1}{p}\right)\left(\frac{x_0}{p}\right) = -\left(\frac{x_0}{p}\right)$. Therefore, among the four different square roots of a modulo N there is exactly one (say x) for which $\left(\frac{x}{p}\right) = \left(\frac{x}{q}\right) = \left(\frac{x}{N}\right) = 1$, that is, which itself is a square modulo N. And, if y is a square root of a such that $y \not\equiv \pm x \bmod N$ then $\left(\frac{y}{N}\right) = -\left(\frac{x}{N}\right)$.

Exercises.

1. *Give a prime certificate for 1031.*
2. *Let N be an odd number, and let p be prime.*

(i) *Prove that $S = \{a \in (\mathbf{Z}/N\mathbf{Z})^* : a^{(N-1)/2} \equiv \left(\frac{a}{N}\right)\}$ forms a subgroup of $(\mathbf{Z}/N\mathbf{Z})^*$.*

(ii) *Prove: if $p^k | N$ and $a^{(N-1)/2} \equiv \left(\frac{a}{N}\right)$ for every $a \in (\mathbf{Z}/N\mathbf{Z})^*$ then $k = 1$.*

(iii) *Let $N = p \cdot r$, with $\gcd(r, p) = 1$ and $r \in \mathbf{Z}_{\geq 3}$. Let $\left(\frac{b}{p}\right) = -1$ and let $c \equiv b \bmod p$ and $c \equiv 1 \bmod r$. Prove that $c^{(N-1)/2} \not\equiv \left(\frac{c}{N}\right)$.*

(iv) *Prove $\#S < \#(\mathbf{Z}/N\mathbf{Z})^*$ for the subgroup S in* (i).

(v) *Prove that if $x \notin S$ then x is a witness for the compositeness for N. Conclude that at least $\phi(N)/2$ elements of $(\mathbf{Z}/N\mathbf{Z})^*$ are witness to the compositeness of N.*

3. *In the 'Fermat factorization' algorithm one attempts to find a factor of N by initially taking $x = \lceil \sqrt{N} \rceil$, and (if $N \neq x^2$) then repeating the two steps of incrementing x by 1 and checking whether $z = x^2 - N$ is the square of an integer, until that condition holds.*

(i) *Show how to recover a factor of N when $z = y^2$.*

(ii) *Show that if N is composite, this method will find a non-trivial factor of N.*

(iii) *How many times does one have to repeat the two steps above?*

(iv) *Using this method to factor N as in Example (4.13) is thousands times slower than using Pollard's methods. However, Fermat's method finds a factor of $217 \cdot N$ faster than Pollard's methods. Explain this, and find an even better 'multiplier' k.*

4. *Let N be a composite number, and let $a \in \mathbf{Z}/N\mathbf{Z}$ be a witness for its compositeness. Let $c = a^{N-1}$.*

(i) *Show that if $c \equiv 1 \bmod N$ then there exists some $k \geq 0$ such that $\gcd(a^{d \cdot 2^{k+1}} - 1, N)$ yields a non-trivial factor of N (where d is the odd part of $N - 1$).*

(ii) *Show that if $c \equiv 0 \bmod N$ then $\gcd(a, N)$ yields a non-trivial factor of N.*

(iii) *Show that if $c \not\equiv 0, 1 \bmod N$ then either $\gcd(c - 1, N)$ is a non-trivial factor of N or $\gcd(c - 1, N) = 1$ and N is not a prime power.*

(iv) *Show that prime powers can be recognized in random polynomial time.*

5. *Let p be prime and $p \equiv 5 \bmod 8$, and let a be a quadratic residue modulo p. Prove:*

(i) $a^{(p-1)/4} \equiv \pm 1 \bmod p$.

(ii) *If $a^{(p-1)/4} \equiv 1 \bmod p$ then $x^2 \equiv a \bmod p$ where $x \equiv a^{(p+3)/8} \bmod p$.*

(iii) *If $a^{(p-1)/4} \equiv -1 \bmod p$ then $x^2 \equiv a \bmod p$ where $x \equiv 2 \cdot a \cdot (4 \cdot a)^{(p-5)/8} \bmod p$.*