Informationssicherheit und Kryptographie

WS 2002/03

Folienkopien zu Kapitel 2 (Teil 2)

Definition: A **pseudo-random bit generator** with key space (or seed space) $\mathcal{S}$ is an efficiently computable mapping from $\mathcal{S}$ to the set of semi-infinite binary strings, satisfying or assumed to satisfy certain properties, e.g.

- large period
- all 2^k k-tuples occur approximately equally often (for certain k).
- unpredictability (cryptographic).

Applications:

- whenever a true random generator is needed but not available (but the seed must in many applications be generated at random).
- reproducible statistical simulations or experiments
- cryptographic key generation and key expansion
- additive stream ciphers

Complexity theory

Goal of complexity theory: Determine exactly, or at least by lower and upper bounds, the **computational resources (time, space, randomness)** needed to solve relevant computational tasks/problems.

Model of computation: One needs a well-defined model of computation (e.g. a universal Turing machine) in order to define computational complexity.

Note: To prove an upper bound it suffices to give an algorithm together with an analysis of its complexity. Proving a lower bound is much more involved: One must prove the **non-existence of an algorithm**.

No significant complexity lower bound has been proved for any practically relevant model of computation and a cryptographically relevant problem.

Computational problems and algorithms

Problem instances:

- Is 63483303847523012744638405834573484018353762722292763454601 prime? The answer is a binary value (yes/no). Such problems are called **decision problems**.
- Factoring the number 7463280903378416522836449811907328953849379.

It does not seem to make sense to define the difficulty or complexity of a particular problem instance. There exists always a trivial algorithm for solving it, namely an algorithm that simply outputs the solution.

Problem: A problem is a well-defined and compactly described large class of instances, characterized by the **input** and the corresponding **output**.

Examples: Deciding primality, factoring integers, graph isomorphism

Problem class: Class of problems with common property (e.g. **P**)

Algorithm: well-defined computational procedure for a well-defined computational model for solving a well-defined problem.

Computational models

- **Boolean circuit.** Circuit of binary-input gates. Complexity measure: number of gates. Every circuit is specific for inputs of a certain size.
- **Turing machine.** Finite state machine with a read-write tape (memory) of unlimited length. Read/write head can move only one position per time unit. Complexity measures: time, space = number of memory cells used.
- **Random access machine.** Finite state machine with memory of unlimited size. Arbitrary (random) access to memory cells. Corresponds to modern computers. Complexity measures: time, space = # of memory cells used.

These models are **equivalent** with respect to which problems are computable in principle. They are also equivalent with respect to computational complexity, up to polynomial transformations. For instance, simulating a random access machine by a Turing machine generally squares the number of step.

- **Quantum computers.** Consistent with the laws of quantum physics. Potentially much more powerful than classical computers. Factoring and computing discrete logarithms can be done in polynomial time on a quantum computer. Today it is unclear whether such devices can ever be built.

Input size and running time

Input size: The length of the binary string needed to represent the input, for a well-defined description method.

Running time: The running time of an algorithm is the number of primitive operations or steps executed.

- **Worst-case** running time: Maximum of all running times for all inputs of a particular size, expressed as a function of the input size.
- **Average-case** running time: Average of all running times for all inputs of a particular size, expressed as a function of the input size.

An algorithm is **uniform** if it works for all sizes of inputs. It is **non-uniform** if it can be different for every input size such that there is no compact description of all these algorithm. **We consider uniform algorithms and complexity classes.**

Asymptotic analysis and notation

Asymptotic analysis: Often one is interested in the behavior of the complexity of a problem as a function of the input size n (in bits) or of some other parameter (e.g. a security parameter).

Example: For a constant $c > 1$, $1 < \ln n < n^c < n^{\ln n} < e^{\sqrt{n \ln n}} < c^n < n^n < c^{c^n}$

Order notation:

- $f(n) = O(g(n))$ if there exists a positive constant c and a positive integer n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$.
- $f(n) = o(g(n))$ if for every positive constant c there exists n_0 such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0$.

Definition: A function $f(n)$ is **polynomial** if $f(n) = O(n^c)$ for some $c > 0$, otherwise it is **super-polynomial**.

A function $f(n)$ is **negligible** if $f(n) = o(n^{-c})$ for all $c > 0$.

A function $f(n)$ is **non-negligible** if for some $c > 0$ and some n_0, $f(n) > n^{-c}$ for all $n > n_0$.

Polynomial equivalence: motivation

For a fixed input size, the complexity of a problem depends on the particular representation and on the computational model.

Any reasonable representation of an input can be transformed efficiently into any other reasonable representation of the input, with a length expansion of at most a polynomial factor.

Any reasonable classical (i.e. not quantum) model of computation can be simulated on any other reasonable model with an overhead factor that is polynomial (e.g. linear of quadratic) in the input size.

One would like to analyse the complexity independently of the particular input representation and the particular model of computation. This can be achieved by **neglecting polynomial factors** for the purpose of a coarse analysis.

Definition: An algorithm runs in **polynomial time** if the running time is upper bounded by a polynomial in the size of the input.

In complexity theory, one often considers **polynomial time** as being **efficient** and super-polynomial time as inefficient. In practice, a super-polynomial algorithm can be quite practical for the input size of interest; conversely, a polynomial-time algorithm can be completely impractical.

Example: $n^{1000} \gg n^{\ln \ln \ln n}$ for all n that can ever be of relevance.

But: In practice, for all problems that can be solved in polynomial time, the polynomial has **small degree** (e.g. ≤ 6).

Definition: A **formal language** L is a subset of the set $\{0,1\}^*$ of all binary strings of finite length.

Example: The set of bitstring that represent prime numbers.

Definition: Consider a computing device (finite state machine, Turing machine, algorithm) taking as input binary strings, with a special accept state. The language **accepted** by the device is the set of input strings for which the accept state is reached. For the other inputs it may run forever.

A **decision problem** (e.g. primality testing) can be interpreted as a language membership problem when the problem instances are encoded as binary strings using an arbitrary but fixed encoding.

Definition: The **complexity class P** is the set of languages L for which there exists a Turing machine accepting L and such that each accepting computation takes at most **polynomial time**.

Equivalently, the **complexity class P** is the set of **decision problems** that can be solved in **deterministic worst-case polynomial time**.

The **complexity class NP** is the set of languages L for which there exists a **non-deterministic** (i.e. arbitrary parallel) Turing machine accepting L in poly-time.

Equivalently, **NP** is the set of decision problems for which every YES answer can be **verified in polynomial time** when given an extra piece of information, called a **certificate**, which may depend on the given input.

The **complexity class co-NP**: same as **NP**, but replace YES by NO.

Examples:

- Deciding whether a system of linear equations has a solution is in **P**.
- Deciding whether a system of non-linear equations has a solution is trivially in **NP** but is not known to be in **P**.
- Deciding whether a number is prime has recently been shown to be in **P**. Even showing that it is in **NP**is not trivial.
- Deciding whether two graphs are **not** isomorphic is not known to be in **NP**.

Fact: **P** $\subseteq$ (**NP** $\cap$ **co-NP**).

Big open questions: **P=NP**? **NP=co-NP**? **P**= (**NP**$\cap$ **co-NP**)?

Randomization often helps dramatically to improve the running time (e.g. primality testing).

The **complexity class BPP** is the set of decision problems for which there is an **expected polynomial-time** randomized algorithm such that the probability of outputting YES for the two cases where the answer should be YES and NO, differs by at least a constant (arbitrarily small).

Polynomial-time reductions and NP-completeness

Definition: Let L_1 and L_2 be two languages (decision problems). L_1 can be **polynomially reduced** to L_2, written $L_1 \leq_P L_2$, if there is a polynomial-time algorithm that accepts L_1, given free access to an oracle for deciding membership in L_2. If both $L_1 \leq_P L_2$ and $L_2 \leq_P L_1$, then L_1 and L_2 are **polynomially equivalent**.

Note: Informally, if $L_1 \leq_P L_2$, then L_2 is at least as difficult as L_1.

Fact: If $L_1 \leq_P L_2$ and $L_2 \in$ **P**, then $L_1 \in$ **P**.

Definition: A language (decision problem) L is said to be **NP-complete** if

(1) $L \in$ **NP** and

(2) $L' \leq_P L$ for every $L' \in$ **NP**.

The class of **NP**-complete problems is denoted **NPC**.

A problem (not necessarily a decision problem) L is called **NP-hard** if (2) but not necessarily (1) is satisfied.

Fact: If any **NP**-hard (or **NP**-complete) problem is in **P**, then **P=NP**.

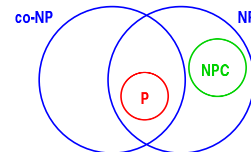
The **NP**-complete problems are hence the most difficult problems in **NP**.

Examples: The following are **NP**-complete problems

- Deciding whether a Boolean formula is satisfiable.
- Deciding whether a directed graph has a Hamiltonian cycle.

Finding the satisfying assignment (or the cycle) are **NP**-hard problems.

Relations between complexity classes:



Cryptographic significance of NP-completeness

- **NP**-completeness is a **worst-case** complexity measure. (However, Ajtai showed that a certain problem on lattices is hard on the average, if it is hard at all.)
- In cryptography, almost all instances (not just the worst case) of a cryptanalyst's problem should be infeasibly difficult.
- A cryptographic system can hence be completely **insecure even if breaking it is NP-hard**.
- If **P=NP**, then in a complexity-theoretic sense there are no secure cryptographic schemes.
- However, even if **P=NP**, there can still exist secure cryptographic systems, provided the complexity ratio between using the system and breaking it is a polynomial of sufficiently high degree.
- A system may be practical even if there exists no polynomial algorithm for using it. Every system has concrete parameters.

Example: **Complexity-theoretic definition of a pseudo-random bit generator (PRBG):**

A PRBG is understood to be a **class of generators** with a **security parameter** k (that can take on any integer value). k could for instance be the size of the seed.

The PRBG algorithm runs in polynomial time (in k) when generating any polynomial number of output bits.

Definition: A PRBG is **cryptographically secure** if for all polynomial time (in k) distinguisher algorithms D, the probability of distinguishing the PRBG (with randomly selected seed) from a true random bit generator is **negligible** (i.e. vanishes faster than 1 over any polynomial in k).