FoC: The Computational Model

Oded Goldreich
Department of Computer Science
Weizmann Institute of Science
Rehovot, ISRAEL.
oded.goldreich@weizmann.ac.il

November 19, 2008

Preface. Our approach to cryptography is heavily based on computational complexity. Thus, some background on computational complexity is required for our discussion of cryptography. Below, we briefly recall the definitions of the complexity classes $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{BPP}$, "non-uniform $\mathcal{P}$" (i.e., $\mathcal{P}/\mathrm{poly}$), and the concept of oracle machines. In addition, we discuss the type of intractability assumptions used throughout the course.

1 P, NP, and NP-completeness

A conservative approach to computing devices associates efficient computations with the complexity class $\mathcal{P}$. Jumping ahead, we note that the approach taken in this course is a more liberal one in that it allows the computing devices to be randomized.

Definition 1.1 (The complexity class $\mathcal{P}$): *A language L is* recognizable in (deterministic) polynomial-time *if there exists a deterministic Turing machine M and a polynomial $p(\cdot)$ such that*

- *On input a string x, machine M halts after at most $p(|x|)$ steps.*
- *$M(x) = 1$ if and only if $x \in L$.*

$\mathcal{P}$ is the class of languages that can be recognized in (deterministic) *polynomial-time.*

Likewise, the complexity class $\mathcal{NP}$ is associated with computational problems having solutions that, once given, can be efficiently tested for validity. It is customary to define $\mathcal{NP}$ as the class of languages that can be recognized by a non-deterministic polynomial-time Turing machine. A more fundamental formulation of $\mathcal{NP}$ is given by the following equivalent definition.

Definition 1.2 (The complexity class $\mathcal{NP}$): *A language L is in $\mathcal{NP}$, if there exists a Boolean relation $R_L \subseteq \{0,1\}^* \times \{0,1\}^*$ and a polynomial $p(\cdot)$ such that R_L can be recognized in (deterministic) polynomial-time and $x \in L$ if and only if there exists a y such that $|y| \leq p(|x|)$ and $(x,y) \in R_L$. Such a y is called a* witness for membership of *$x \in L$.*

Thus, $\mathcal{NP}$ consists of the set of languages for which there exist short proofs of membership that can be efficiently verified. It is widely believed that $\mathcal{P} \neq \mathcal{NP}$, and settling this conjecture is certainly the most intriguing open problem in Computer Science. If indeed $\mathcal{P} \neq \mathcal{NP}$ then there exists a language $L \in \mathcal{NP}$ so that every algorithm recognizing L has super-polynomial running-time *in the worst-case*. Certainly, all NP-complete languages (see definition below) will have super-polynomial time complexity *in the worst-case*.

Definition 1.3 (NP-completeness): *A language is* NP-complete *if it is in $\mathcal{NP}$ and every language in $\mathcal{NP}$ is polynomially-reducible to it. A language L is* polynomially-reducible *to a language L' if there exist a polynomial-time computable function f so that $x \in L$ if and only if $f(x) \in L'$.*

Among the languages known to be NP-complete are *Satisfiability* (of propositional formulae), *Graph Colorability*, and *Graph Hamiltonicity*.

2 Probabilistic Polynomial-Time

Randomized algorithms play a central role in cryptography. They are needed in order (to allow the legitimate parties) to generate secrets, and are (therefore) allowed also to the adversaries. The reader is assumed to be familiar and comfortable with such algorithms.

2.1 Randomized algorithms – an example

To demonstrate the notion of a randomized algorithm, we present a simple randomized algorithm for deciding whether a given (undirected) graph is connected (i.e., there is a path between each pair of vertices in the graph). We comment that the following algorithm is interesting because it uses significantly less space than the standard (BFS or DFS-based) deterministic algorithms.

Testing whether a graph is connected is easily reduced to testing connectivity between any given pair of vertices.[1] Thus, we focus on the task of determining whether two given vertices are connected in a given graph.

Algorithm: On input a graph $G = (V, E)$ and two vertices, s and t, we take a *random walk* of length $O(|V| \cdot |E|)$, starting at vertex s, and test at each step whether vertex t is encountered. If vertex t is ever encountered then the algorithm accepts, otherwise it rejects. By a random walk we mean that, at each step, we uniformly select one of the edges incident at the current vertex, and traverse this edge to the other endpoint.

Analysis: Clearly, if s is not connected to t in the graph G then the probability that the above algorithm accepts is zero. The harder part of the analysis is proving that if s is connected to t in the graph G then the algorithm accepts with probability at least $2/3$. Thus, either way, the algorithm errs with probability at most $1/3$. The error probability may be further reduced by invoking the algorithm several times (using fresh random choices in each try).

2.2 Randomized algorithms – two points of view

Randomized algorithms (machines) can be viewed in two equivalent ways. One way of viewing randomized algorithms is to allow the algorithm to make random moves (i.e., "toss coins"). Formally this can be modeled by a Turing machine in which the transition function maps pairs of the form $(\langle\text{state}\rangle, \langle\text{symbol}\rangle)$ to two possible triples of the form $(\langle\text{state}\rangle, \langle\text{symbol}\rangle, \langle\text{direction}\rangle)$. The next step of such a machine is determined by a random choice of one of these triples. Namely, to make a step, the machine chooses at random (with probability one half for each possibility) either the first triple or the second one, and then acts accordingly. These random choices are called the *internal coin*

[1] The space complexity of such a reduction is low; we merely need to store the names of two vertices (currently being tested). Alas, the time complexity is indeed relatively high; we need to invoke the two-vertex tester $\binom{n}{2}$ times, where n is the number of vertices in the graph.

tosses of the machine. The output of a probabilistic machine, M, on input x is not a string but rather a random variable assuming strings as possible values. This random variable, denoted $M(x)$, is induced by the internal coin tosses of M. By $\Pr[M(x)=y]$ we mean the probability that machine M on input x outputs y. The probability space is that of all possible outcomes for the internal coin tosses taken with uniform probability distribution.[2] Since we only consider polynomial-time machines, we may assume without loss of generality, that the number of coin tosses made by M on input x is independent of their outcome, and is denoted by $t_M(x)$. We denote by $M_r(x)$ the output of M on input x when r is the outcome of its internal coin tosses. Then, $\Pr[M(x)=y]$ is merely the fraction of $r \in \{0,1\}^{t_M(x)}$ for which $M_r(x) = y$. Namely,

$$\Pr[M(x)=y] = \frac{|\{r \in \{0,1\}^{t_M(x)} : M_r(x)=y\}|}{2^{t_M(x)}}$$

The second way of looking at randomized algorithms is to view the outcome of the internal coin tosses of the machine as an auxiliary input. Namely, we consider deterministic machines with two inputs. The first input plays the role of the "real input" (i.e., x) of the first approach, while the second input plays the role of a possible outcome for a sequence of internal coin tosses. Thus, the notation $M(x,r)$ corresponds to the notation $M_r(x)$ used above. In the second approach one considers the probability distribution of $M(x,r)$, for any *fixed* x and a uniformly chosen $r \in \{0,1\}^{t_M(x)}$. Pictorially, here the coin tosses are not "internal" but rather supplied to the machine by an "external" coin tossing device.

Before continuing, let us remark that one should not confuse the fictitious model of "non-deterministic" machines with the model of probabilistic machines. The first is an unrealistic model which is useful for talking about search problems the solutions to which can be efficiently verified (e.g., the definition of $\mathcal{NP}$), while the second is a realistic model of computation.

Throughout the entire course , unless otherwise stated, a *probabilistic polynomial-time Turing machine* means a probabilistic machine that always (i.e., independently of the outcome of its internal coin tosses) halts after a polynomial (in the length of the input) number of steps. It follows that the number of coin tosses of a probabilistic polynomial-time machine M is bounded by a polynomial, denoted T_M, in its input length. Finally, without loss of generality, we assume that on input x the machine always makes $T_M(|x|)$ coin tosses.

2.3 Associating "efficient" computations with BPP

The basic thesis underlying our discussion is the association of "efficient" computations with probabilistic polynomial-time computations. Namely, we will consider as efficient only randomized algorithms (i.e., probabilistic Turing Turing machines) whose running time is bounded by a polynomial in the length of the input.

Thesis: *Efficient computations correspond to computations that can be carried out by probabilistic polynomial-time Turing machines.*

A complexity class capturing these computations is the class, denoted $\mathcal{BPP}$, of languages recognizable (with high probability) by probabilistic polynomial-time Turing machines. The probability refers to the event *the machine makes correct verdict on string* x.

[2] The last sentence is slightly more problematic than it seems. The simple case is when, on input x, machine M always makes the same number of internal coin tosses (independent of their outcome). In general, the number of coin tosses may depend on the outcome of prior coin tosses. Still, for every r, the probability that the outcome of the sequence of internal coin tosses is r equals $2^{-|r|}$ if the machine does not terminate when the sequence of outcomes is a strict prefix of r and equals zero otherwise. Fortunately, since we consider polynomial-time machines, we can modify all machines so that they satisfy the structure of the simple case (and so avoid the above complication).

Definition 2.1 (Bounded-Probability Polynomial-time — $\mathcal{BPP}$): *We say that L is* recognized by the probabilistic polynomial-time Turing machine *M if*

- *For every $x \in L$ it holds that* $\Pr[M(x)=1] \geq \frac{2}{3}$.
- *For every $x \notin L$ it holds that* $\Pr[M(x)=0] \geq \frac{2}{3}$.

$\mathcal{BPP}$ is the class of languages that can be recognized by a probabilistic polynomial-time Turing machine (i.e., randomized algorithm).

The phrase "bounded-probability" indicates that the success probability is bounded away from $\frac{1}{2}$. In fact, substituting in Definition 2.1 the constant $\frac{2}{3}$ by any other constant greater than $\frac{1}{2}$ does not change the class defined. Likewise, the constant $\frac{2}{3}$ can be replaced by $1 - 2^{-|x|}$, while the class remains invariant. We conclude that languages in $\mathcal{BPP}$ can be recognized by probabilistic polynomial-time algorithms with a negligible error probability. By *negligible* we call any function that decreases faster than one over any polynomial. Namely,

Definition 2.2 (negligible): *We call a function $\mu : \mathsf{N} \rightarrow \mathsf{R}$* negligible *if for every positive polynomial $p(\cdot)$ there exists an N such that for all $n > N$*

$$\mu(n) < \frac{1}{p(n)}$$

For example, the functions $2^{-\sqrt{n}}$ and $n^{-\log_2 n}$, are negligible (as functions in n). Negligible function stay this way when multiplied by any fixed polynomial. Namely, for every negligible function μ and any polynomial p, the function $\mu'(n) \stackrel{\text{def}}{=} p(n) \cdot \mu(n)$ is negligible. It follows that an event which occurs with negligible probability is highly unlikely to occur even if we repeat the experiment polynomially many times.

Convention: In Definition 2.2 we used the phrase "there exists an N such that for all $n > N$". In the future we will use the shorter and less tedious phrase "for all sufficiently large n". This makes one quantifier (i.e., the $\exists N$) implicit, and is particularly beneficial in statements that contain several (more essential) quantifiers.

3 Non-Uniform Polynomial-Time

A stronger (and actually unrealistic) model of efficient computation is that of non-uniform polynomial-time. This model will be used only in the negative way; namely, for saying that even such machines cannot do something (specifically, even if the adversary employs such a machine it cannot cause harm).

A *non-uniform polynomial-time "machine"* is a pair $(M, \overline{a})$, where M is a two-input polynomial-time Turing machine and $\overline{a} = a_1, a_2, ...$ is an infinite sequence of strings such that $|a_n| = \text{poly}(n)$.[3] For every x, we consider the computation of machine M on the input pair $(x, a_{|x|})$. Intuitively, a_n may be thought as an extra "advice" supplied from the "outside" (together with the input $x \in \{0,1\}^n$). We stress that machine M gets the same advice (i.e., a_n) on all inputs of the same

[3] Recall that poly() stands for some (unspecified) fixed polynomial; that is, we say that there exists some polynomial p so that $|a_n| = p(n)$, for all $n \in \mathsf{N}$.

length (i.e., n). Intuitively, the advice a_n may be useful in some cases (i.e., for some computations on inputs of length n), but it is unlikely to encode enough information to be useful for all 2^n possible inputs.

Another way of looking at non-uniform polynomial-time "machines" is to consider an infinite sequence of Turing machines, $M_1, M_2, ...$ so that both the length of the description of M_n and its running time on inputs of length n are bounded by polynomial in n (fixed for the entire sequence). Machine M_n is used only on inputs of length n. Note the correspondence between the two ways of looking at non-uniform polynomial-time. The pair $(M, (a_1, a_2, ...))$ (of the first definition) gives rise to an infinite sequence of machines $M_{a_1}, M_{a_2}, ...$, where $M_{a_{|x|}}(x) \stackrel{\text{def}}{=} M(x, a_{|x|})$. On the other hand, a sequence $M_1, M_2, ...$ (as in the second definition) gives rise to a pair $(U, (\langle M_1\rangle, \langle M_2\rangle, ...))$, where U is the universal Turing machine and $\langle M_n\rangle$ is the description of machine M_n (i.e., $U(x, \langle M_{|x|}\rangle) = M_{|x|}(x)$).

In the first sentence of the current section, non-uniform polynomial-time has been referred to as a stronger model than probabilistic polynomial-time. This statement is valid in many contexts (e.g., language recognition as in Theorem 3.2 below). In particular it will be valid in all contexts we discuss in this course . So we have the following informal "meta-theorem"

Meta-Theorem: *Whatever can be achieved by probabilistic polynomial-time machines can be achieved by non-uniform polynomial-time "machines".*

The meta-theorem is clearly wrong if one thinks of the task of tossing coins... So the meta-theorem should not be understood literally. It is merely an indication of real theorems that can be proven in reasonable cases. Let us consider, for example, the context of language recognition.

Definition 3.1 *The complexity class* non-uniform polynomial-time (*denoted* $\mathcal{P}/\text{poly}$) *is the class of languages L that can be recognized by a non-uniform (sequence) of polynomial-time "machines". Namely, $L \in \mathcal{P}/\text{poly}$ if there exists an infinite sequence of machines $M_1, M_2, ...$ satisfying*

1. *There exists a polynomial $p(\cdot)$ such that, for every n, the description of machine M_n has length bounded above by $p(n)$.*
2. *There exists a polynomial $q(\cdot)$ such that, for every n, the running time of machine M_n on each input of length n is bounded above by $q(n)$.*
3. *For every n and every $x \in \{0,1\}^n$, machine M_n accepts x if and only if $x \in L$.*

Note that the non-uniformity is implicit in the lack of a requirement concerning the construction of the machines in the sequence. It is only required that these machines exist. In contrast, if one augments Definition 3.1 by requiring the existence of a polynomial-time algorithm that on input 1^n (n presented in unary) outputs the description of M_n then one gets a cumbersome way of defining $\mathcal{P}$. On the other hand, it is obvious that $\mathcal{P} \subseteq \mathcal{P}/\text{poly}$ (in fact strict containment can be proven by considering non-recursive unary languages). Furthermore,

Theorem 3.2 $\mathcal{BPP} \subseteq \mathcal{P}/\text{poly}$.

Proof: Let M be a probabilistic polynomial-time Turing machine recognizing $L \in \mathcal{BPP}$. Let $\chi_L(x) \stackrel{\text{def}}{=} 1$ if $x \in L$ and $\chi_L(x) \stackrel{\text{def}}{=} 0$ otherwise. Then, for every $x \in \{0,1\}^*$,

$$\Pr[M(x) = \chi_L(x)] \geq \frac{2}{3}$$

Assume, without loss of generality, that on each input of length n, machine M uses the same number, denoted $m = \text{poly}(n)$, of coin tosses. Let $x \in \{0,1\}^n$. Clearly, we can find for each $x \in \{0,1\}^n$ a sequence of coin tosses $r \in \{0,1\}^m$ such that $M_r(x) = \chi_L(x)$ (in fact most sequences r have this property). But can one sequence $r \in \{0,1\}^m$ fit all $x \in \{0,1\}^n$? Probably not (provide an example!). Nevertheless, we can find a sequence $r \in \{0,1\}^n$ that fits $\frac{2}{3}$ of all the x's of length n. This is done by a averaging argument (which asserts that if $\frac{2}{3}$ of the r's are good for each x then there is an r that is good for at least $\frac{2}{3}$ of the x's). However, this does not give us an r that is good for all $x \in \{0,1\}^n$. To get such an r we have to apply the above argument on a machine M' with exponentially vanishing error probability. Such a machine is guaranteed by the foregoing discussion (regarding error reduction). Namely, for every $x \in \{0,1\}^*$,

$$\Pr[M'(x) = \chi_L(x)] > 1 - 2^{-|x|}$$

Applying the averaging argument now we conclude that there exists an $r \in \{0,1\}^m$, denoted r_n, that is good for *more than* a $1 - 2^{-n}$ fraction of the x's in $\{0,1\}^n$. It follows that r_n is good for all the 2^n inputs of length n. Machine M' (viewed as a deterministic two-input machine) together with the infinite sequence $r_1, r_2, ...$ constructed as above, demonstrates that L is in $\mathcal{P}/\text{poly}$. ■

Non-uniform circuits families. A more convenient way of viewing non-uniform polynomial-time, which is actually the way used in this course, is via (non-uniform) families of polynomial-size Boolean circuits. A *Boolean circuit* is a directed acyclic graph with internal nodes marked by elements of $\{\wedge, \vee, \neg\}$. Nodes with no in-going edges are called *input nodes*, and nodes with no outgoing edges are called *output nodes*. A node marked $\neg$ may have only one child. Computation in the circuit begins with placing input bits on the input nodes (one bit per node) and proceeds as follows. If the children of a node (of in-degree d) marked $\wedge$ have values $v_1, v_2, ..., v_d$ then the node gets the value $\wedge_{i=1}^{d} v_i$. Similarly for nodes marked $\vee$ and $\neg$. The output of the circuit is read from its output nodes. The *size* of a circuit is the number of its edges. A *polynomial-size circuit family* is an infinite sequence of Boolean circuits, $C_1, C_2, ...$ such that, for every n, the circuit C_n has n input nodes and size $p(n)$, where $p(\cdot)$ is a polynomial (fixed for the entire family).

The computation of a Turing machine M on inputs of length n can be simulated by a single circuit (with n input nodes) having size $O((|\langle M \rangle| + n + t(n))^2)$, where $t(n)$ is a bound on the running time of M on inputs of length n. Thus, a non-uniform sequence of polynomial-time machines can be simulated by a non-uniform family of polynomial-size circuits. The converse is also true since machines with polynomial description length can incorporate polynomial-size circuits and simulate their computations in polynomial-time. The thing which is nice about the circuit formulation is that there is no need to repeat the polynomiality requirement twice (once for size and once for time) as in the first formulation.

Convention: For sake of simplicity, we often take the liberty of considering circuit families $\{C_n\}_{n \in \mathbb{N}}$, where each C_n has $\text{poly}(n)$ input bits rather than n.

4 Intractability Assumptions

We will consider as *intractable* those tasks that cannot be performed by probabilistic polynomial-time machines. However, the adversarial tasks in which we will be interested (e.g., "breaking an encryption scheme", "forging signatures", etc.) can be performed by non-deterministic polynomial-time machines (since the solutions, once found, can be easily tested for validity). Thus, the computational approach to cryptography (and in particular most of the material in this course) is

interesting only if $\mathcal{NP}$ is not contained in $\mathcal{BPP}$ (which certainly implies $\mathcal{P} \neq \mathcal{NP}$).[4] We use the phrase "not interesting" (rather than "not valid") since all our statements will be of the form "if ⟨`intractability assumption`⟩ then ⟨`useful consequence`⟩". Such a statement remains valid even if $\mathcal{P} = \mathcal{NP}$ (or just ⟨`intractability assumption`⟩, which is never weaker than $\mathcal{P} \neq \mathcal{NP}$, is wrong); but in such a case the implication is of little interest (since everything is implied by a fallacy).

In most places where we state that "if ⟨`intractability assumption`⟩ then ⟨`useful consequence`⟩" it will be the case that ⟨`useful consequence`⟩ either implies ⟨`intractability assumption`⟩ or some weaker form of it, which in turn implies $\mathcal{NP} \setminus \mathcal{BPP} \neq \emptyset$. Thus, in light of the current state of knowledge in complexity theory, one cannot hope for asserting ⟨`useful consequence`⟩ without any intractability assumption.

In few cases an assumption concerning the limitations of probabilistic polynomial-time machines will not suffice, and we will use instead an assumption concerning the limitations of non-uniform polynomial-time machines. Such an assumption is of course stronger. But also the consequences in such a case will be stronger, since they will also be phrased in terms of non-uniform complexity. However, since all our proofs are obtained by reductions, an implication stated in terms of probabilistic polynomial-time is stronger (than one stated in terms of non-uniform polynomial-time), and will be preferred unless it is either not known or too complicated. This is the case since a probabilistic polynomial-time reduction (proving implication in its probabilistic formalization) always implies a non-uniform polynomial-time reduction (proving the statement in its non-uniform formalization), but the converse is not always true.[5]

Finally, we mention that intractability assumptions concerning worst-case complexity (e.g., $\mathcal{P} \neq \mathcal{NP}$) will not suffice, because we will *not be satisfied* with their corresponding consequences. Cryptographic schemes that are only guaranteed to be *hard to break in the* WORST-CASE are useless. A cryptographic scheme must be unbreakable on "most cases" (i.e., "typical case") which implies that it is *hard to break on the average*. It follows that, since we are not able to prove that "worst-case intractability" implies analogous "intractability for average case" (such a result would be considered a breakthrough in complexity theory), our intractability assumption must concern average-case complexity.

5 Oracle Machines

The original utility of oracle machines in complexity theory is to capture notions of reducibility. In the context of Cryptography, we use oracle machines mainly for a different purpose altogether: to model an adversary that may use a cryptosystem in course of its attempt to break it.

Loosely speaking, an oracle machine is a machine that is augmented so that it may ask questions to the outside. We consider the case in which these questions (called queries) are answered consistently by some function $f : \{0,1\}^* \to \{0,1\}^*$, called the oracle. That is, if the machine makes a query q then the answer it obtains is $f(q)$. In such a case, we say that the oracle machine is given access to the oracle f.

Definition 5.1 (oracle machines): *A* (deterministic / probabilistic) oracle machine *is a* (deterministic / probabilistic) *Turing machine with an additional tape, called the* oracle tape*, and two*

[4] We remark that $\mathcal{NP}$ is not known to contain $\mathcal{BPP}$. This is the reason we state the above conjecture as $\mathcal{NP}$ *is not contained in* $\mathcal{BPP}$, rather than $\mathcal{BPP} \neq \mathcal{NP}$. Likewise, although "sufficiently strong" one-way functions imply $\mathcal{BPP} = \mathcal{P}$, this equality is not known to hold unconditionally.

[5] The current paragraph may be better understood in the future after seeing some concrete examples.

special states, called oracle invocation *and* oracle appeared*. The* computation of the deterministic oracle machine M on input x and access to the oracle $f : \{0,1\}^* \to \{0,1\}^*$ *is defined by the* successive configuration relation*. For configurations with state different from* oracle invocation *the next configuration is defined as usual. Let γ be a configuration in which the state is* oracle invocation *and the contents of the oracle tape is q. Then the configuration following γ is identical to γ, except that the state is* oracle appeared*, and the contents of the oracle tape is $f(q)$. The string q is called M's* query *and $f(q)$ is called the* oracle reply*. The computation of a probabilistic oracle machine is defined analogously. The output distribution of the oracle machine M, on input x and access to the oracle f, is denoted $M^f(x)$.*

We stress that the running time of an oracle machine is the number of steps made during its computation, and that the oracle's reply on each query is obtained in a single step.