CS 127/CSCI E-127: Introduction to Cryptography

Prof. Salil Vadhan Fall 2013

Lecture Notes 4:

Review of Algorithms & Complexity

Reading.

- Katz–Lindell §A.2, B.0, B.1 (1st edition).
- Cormen, Leiserson, Rivest, Stein. *Introduction to Algorithms* (2nd ed), Ch. 1–3, 5, 31, 34.
- Sipser. *Introduction to the Theory of Computation.* (various parts)

1 Algorithms

In this course, we will not use a specific model of computation when talking about algorithms. An algorithm can be a program for a Turing machine, RAM model, or your favorite model of computation, as they are roughly equivalent in terms of efficiency.

The inputs of an algorithm are strings over an alphabet Σ, Σ being often $\{0,1\}$. Hence an algorithm A is a function from Σ^* to $\Sigma^\star$.

- *A computes* a function $f : \{0,1\}^* \to \{0,1\}^*$: on input x in $\{0,1\}^*$, A outputs $f(x) \in \{0,1\}^*$
- *A decides* a language $L \subset \{0,1\}^*$: on input x, A outputs 1 if $x \in L$ and 0 if $x \notin L$. (equivalent to computing a boolean function $f : \{0,1\}^* \to \{0,1\}$).
- *A runs in time $T : \mathbb{N} \to \mathbb{N}$*: for all x, $A(x)$ halts after at most $T(\|x\|)$ time units, where $\|x\|$ is the *length* of the string x.
- *A runs in polynomial time* if it runs in time $O(n^c)$ for some constant c.

  Efficient (or *feasible*) algorithms are ones that run in polynomial time. (Although the constant c is important for practical efficiency, the "polynomial time" efficiency criterion has the advantage that it does not depend on the encoding of the data or the model of computation (given reasonable choices for these).) An algorithm that does not run in polynomial time is viewed as *infeasible*.

  Note that numbers are encoded using their *binary representation* so if the input is an integer N, an algorithm that runs in polynomial time should run in time polynomial in $\|N\| \approx \log N$.

2 Polynomial time

The complexity class **P** consists of all problems solvable in polynomial time. (Technically, it is a class of languages)

- problems seen in CS124: sorting, shortest path, maximum flow, linear programming, ...
- many arithmetic functions which are important in cryptography.

Examples:

- DIV$(x, y) = (q, r)$ such that $x = y \cdot q + r$ where $r < y$. The remainder r is also denoted $x \bmod y$.
- GCD(x, y) is defined as the largest z such that $z|x$ and $z|y$. It can be computed in polynomial time using Euclid's algorithm.
- exponentiation: EXP$(x, y) = x^y$
- modular exponentiation: MODEXP$(x, y, z) = x^y \mod z$

2.1 Problems believed to not be in P

Computational problems which are not solvable in poynomial time are said to be *intractable*. Here are some examples.

- A problem *provably* not in **P**?
- The complexity class **NP** is the class of decision problems (yes/no answer) or languages that have "short proofs" of membership.
  More formally: $L \in \mathbf{NP}$ if there exists a polynomial time verifier V and a polynomial q such that
  $$x \in L \iff \exists w : \ |w| \leq q(\|x\|) \ , \ V(x, w) = 1.$$
  The string w is the *witness* for x.
  Example: SAT.

- **NP**-complete problems are the "hardest problems in **NP**". Formally, problem Π is **NP**-*complete* if it is in **NP** and every problem in **NP** reduces to Π.
  - Satisfiability, Travelling Salesman Problem, and Graph 3-Coloring.
  - If an **NP**-complete problem were solvable in polynomial time, then every **NP** problem would be solvable in polynomial time.
  - $\mathbf{P} \neq \mathbf{NP}$ widely believed (but still unproven).
- Integer Factorization: given a composite number N, find a non-trivial (=other than 1 and N) factor of N.
  - Neither known to be in **P** nor **NP**-complete.

  Some algorithms:
  - exhaustive search / trial division
  - quadratic sieve: this is the best provable algorithm, it runs in time roughly $2^{\sqrt{\|N\|}}$
  - number field sieve: the unproven time bound for this algorithm is roughly $2^{\sqrt[3]{\|N\|}}$

  So far, there is no known polynomial-time algorithm.

2.2 Asymptotic vs. concrete complexity

For theoretical study, it is convenient to focus on the asymptotic complexity of an algorithm. However, when evaluating the security or efficiency of a specific implementation, we are also interested in complexity for fixed input length, e.g. "how many cycles on an Intel Core i7 does it take to factor a 1000 digit composite number?" This notion is less robust because it depends on the choice of model of computation and the encoding of the data. We must also measure the program size in addition to the running time — why?

3 Randomized Algorithms

3.1 Definition

A fundamental assumption in cryptography is that randomization is possible, i.e. we can have algorithms that "toss coins". We assume there is a black box in the computer that gives us the results of fair coin tosses. We will not consider where this randomness comes from: we just assume that every party/algorithm has its own physical source of randomness.

Let A be a randomized algorithm. We write $A(x;r)$ for the output of A on input x and coin tosses $r \in \{0,1\}^*$, and write $A(x)$ for the random variable giving A's output when the coin tosses are chosen uniformly at random. The probability space is the set of all r's, where every sequence of coin tosses is equally likely.

What does it mean for a randomized algorithm to solve a problem?

A computes $f : \{0,1\}^* \to \{0,1\}^*$ if $\forall x \Pr_r[A(x;r) = f(x)] \geq 2/3$.

The algorithm works on every input correctly with high probability. The bound $2/3$ can be replaced by any constant $c > .5$, as we can reduce the error probability to 2^{-k} by doing $O(k)$ independent repetitions and taking the majority vote. It is important to note that the probability is only over the coin tosses and *not the input*.

A randomized algorithm which can give the wrong answer with small probability is referred to as being a "Monte Carlo" algorithm (as opposed to "Las Vegas" where the algorithm always outputs the correct answer or `fail`).

BPP is the class of problems that can be solved by polynomial-time randomized algorithms. These are what we will consider to be "easy problems". We will use *PPT* as an abbreviation for *probabilistic polynomial-time*. It is widely believed **NP** $\not\subset$ **BPP** (in fact, there is evidence that **P** = **BPP** — take CS225!)

3.2 Primality testing

- The Primality Testing problem: given an integer N, is it prime?
- In **BPP**: Solovay–Strassen, Miller–Rabin (1970's).
- In **P**: Agarwal–Kayal–Saxena (2002).
- Example of a **BPP** algorithm for primality testing, on input an odd integer N:
  - Repeat several times (i being the index for the loop)
    - Choose a random number y_i between 1 and N : $y_i \xleftarrow{R} \{1, \cdots, N\}$

    * If $gcd(y_i, N) = 1$ then compute $z_i = y_i^{\frac{N-1}{2}} \mod N$
  - If all the z_i's belong to $\{1, N-1\}$ and both values 1 and $N-1$ occurr, output "prime". Otherwise, output "composite".

This is a polynomial-time algorithm because the gcd is computed with Euclid's algorithm in polynomial time and the modular exponentiation is computed with the repeated squares and multiply algorithm in polynomial time. For this randomized algorithm, the probability is taken over the random choices of y_i's. The Miller–Rabin primality test is analyzed in Katz–Lindell. One disadvantage of this algorithm over other primality testing algorithms is that there is a two-sided error. (For the other algorithms, an error occurs only when a composite number is said to be prime)