16/10/2023. Lecture 6.

1 Fast exponentiation

In the class we discussed one standard algebraic algorithm — the algorithm of fast exponentiation. This algorithm takes as the input a triple of integer numbers, (a, k, p), and returns the value $a^k \mod p$. The problem of exponentiation may look trivial: we can take the number a, multiply it by itself k times,

$$\underbrace{a \times a \times \ldots \times a}_{k}$$

and the reduce the obtained result modulo p. However, this naive scheme is too expensive. Indeed, this procedure requires k operations of multiplication. If the binary expansion of k consists of s binary digits, then the suggested procedure runs in time exponential in s (i.e., exponential in the length of the input). Fortunately, there exists a much more efficient algorithm. We will explain it in to different way.

The first explanation (adapted for the human perception). We begin with a representation of the number k by it binary expansion, $(k)_2 = \overline{k_s k_{s-1} \ldots k_1 k_0}$, which means that

$$k = k_0 + 2k_1 + 4k_2 + 8k_3 + \ldots + 2^s k_s$$

(each k_i is a binary digit, i.e., either 0 or 1). Then a^k can be represented as follows:

$$a^k = a^{k_0} \cdot a^{2k_1} \cdot a^{4k_2} \cdot a^{8k_3} \cdot \ldots \cdot a^{2^s k_s} = \prod_{j\,:\,k_j=1} a^{2^j}.$$

Now it is clear that we can compute $a^k \mod p$ in two stages:

(i) Compute sequentially the values $a^{2^j} \mod p$ for $j = 1, 2, \ldots, s$. Each next value can be computed as $a^{2^{j+1}} = \left(a^{2^j}\right)^2 \mod p$.

(ii) Compute the product $\prod_{j\,:\,k_j=1} a^{2^j} \mod p$, combining the values a^{2^j} such that $k_j = 1$.

The first stage consists of exactly s multiplications, the second stage consists of at most s multiplications (where s is the number of binary digits in k, i.e., $s = \lceil \log_2 k \rceil$). Each operation of multiplication modulo p requires $\mathrm{poly}(\log p)$ elementary operation (on each stage we multiply two numbers with at most $\lceil \log_2 p \rceil$ binary digits and then divide the product by p with a reminder). Thus, we have $O(\log k)$ stages, and each one can be done in time $\mathrm{poly}(\log p)$.

If the number a is much larger than k and p, then the very first stage can be more expensive: we need to reduce a modulo p, which requires $\mathrm{poly}(\log a, \log p)$ operations ($\lceil \log_2 a \rceil$ is the number of digits in the standard binary expansion of a).

The second explanation (adapted for the computer programming). Substantially the same algorithm of exponentiation can be reformulated as follows:

```
inputs: a, k, p;
z:= 1;
t:= k;
y:= a;
while t>0 {
  if ( t is odd )  {
    z := z * y mod p;
    t:= t-1;
  } else {
    y:= y * y mod p;
    t:= t/2;
  }
}
return z.
```

It is easy to see that this algorithm maintains the invariant

$$z \times y^t = a^k \mod p$$

Thus, when the value of t achieves 0, the variable z contains the value $a^k \mod n$.

In this algorithm, the operations

```
y:= y * y mod p;
t:= t/2;
```

are executed $\lceil \log_2 k \rceil$ times. The operations

```
z := z * y mod p;
t:= t-1;
```

are executed as many times as there are 1's in the binary representation of k, i.e., at most $\lceil \log_2 k \rceil$ times. It follows that the algorithm runs in time that polynomially depends on the size of the input (on the number of digits in the numbers a, k, p).

2 Pseudo-random generators and computationally secure schemes

In this section we show that a computationally secure encryption scheme can be constructed with help of pseudo-random generator. We begin with the definition of a pseudo-random generator.

Definition 1. We say that a function

$$G \,:\, \{0,1\}^{\ell(n)} \to \{0,1\}^n$$

is a *pseudo-random generator* if

- $\ell(n) < n$

- $G(x)$ is computed (by a deterministic algorithm) in polynomial time
- for every poly-time algorithms D (deterministic or randomised) the difference

$$\left|\mathrm{Prob}_{x\in_R\{0,1\}^{\ell(n)}}[D(G(x))=1]-\mathrm{Prob}_{y\in_R\{0,1\}^{n}}[D(y)=1]\right|$$

  is negligibly small.

This definition can be interpreted as follows. A pseudo-random generator is a function G that transforms a *seed* x of lenght $\ell(n)$ in a longer output $y = G(x)$ of length n. If we choose a seed x at random (with a uniform distribution on the set of all strings $\{0,1\}^{\ell(n)}$), then the generator induces some probability distribution on the set of values $G(x)$ on the set of strings of length $\{0,1\}^n$. Of course, this distribution is *not* a uniform distribution on $\{0,1\}^n$. However, for an observer with a polynomial computational power this output looks "very similar" to a uniform distribution. This means that if a test/discriminator D tries to distinguish between "good" and "bad" outcomes, than the fractions of "good" and "bad" strings among truly random ones (i.e., $\mathrm{Prob}_{y\in_R\{0,1\}^n}[D(y)=1]$) and pseudo-random ones (i.e., $\mathrm{Prob}_{x\in_R\{0,1\}^{\ell}}[D(G(x))=1]$) are "almost the same". The word "almost" means that the difference between these probabilities is negligibly small. This condition means that for practical reasons we can use pseudo-random strings instead of truly random ones, and all realisable tests would not see the difference.

Remark 1. The very fact that *pseudo-random generators exist* is highly non-trivial. It is conjectured that they do exist, but this hypothesis remains unproven. This hypothesis is stronger than the famous unproven conjecture P $\neq$ NP.

Proposition 1. *If $G : \{0,1\}^{\ell(n)} \to \{0,1\}^n$ is a pseudo-random generator, then $\ell(n) > \log n$ for almost all n.*

(We proved this proposition in the class.)

Theorem 1. *If $G : \{0,1\}^{\ell(n)} \to \{0,1\}^n$ is a pseudo-random generator, then a version of Vernam's encryption scheme $\Pi = \langle Gen, Enc, Dec\rangle$, where*

- *the algorithm $Gen(\underbrace{11\ldots1}_{n})$ takes a random $k \in \{0,1\}^{\ell(n)}$ and returns $k' = G(k)$*
- *the algorithm $Enc(m, k')$ computes a bitwise XOR of the open message m and the key k'*
- *the algorithm $Dec(e, k')$ computes a bitwise XOR of the encrypted message e and the key k'*

is a computationally secure scheme.

Sketch of the proof. In the class we proved this theorem using a proof by contradiction: is the scheme does not satisfy the definition of a computationally secure scheme, then (by the definition of computationally security) there is an opponent that can distinguish with non-negligible probability encodings of two messages m_a and m_b; we use this algorithm to construct a discriminator D that can distinguish between truly random strings of bits and pseudo-random strings of bits produced by G, which contradicts the definition of a pseudo-random generator. $\square$

Observe that this scheme allows to reduce the size of the secret key from n to a strictly smaller $\ell(n)$ (which is impossible for perfectly secure schemes).