PRIMALITY TEST, FACTORIZATION AND DISCRETE LOGARITHM

DIRK KUSSIN

Essentially the same material can be found also in Chapter 6 and 7 of the textbook [2] and – much more detailed – in the "Handbook" [1] (Ch. 3 and 4).

1. Generating large prime numbers

Definition 1.1. A positive integer $n \geq 2$ is called *prime*, if from $n = ab$, where a and b are positive integers, it follows, that either $a = 1$ oder $b = 1$. Otherwise it is called *composite*.

A composite n always has a non-trivial factorization $n = ab$ with $1 < a, b < n$.

For the RSA algorithm large prime numbers (about 150 digits) have to be generated. We consider the following approach

1. Generate as *candidate* a random odd number n of appropriate size.
2. Test n for primality.
3. If n is composite, return to the first step.

(See section 5 for a more detailed discussion.)

The outcome of the primality test in step 2 might be either a so-called *provable prime* or a so-called *probable prime*. In the first case the test proves that our candidate is a prime, in the second case the test is weaker in the sense that we cannot conclude with absolute certainty from the result that our candidate is prime, and the declaration "prime" may be incorrect. Nevertheless, in practice such a probability test will be used since it is usually running much faster and the confidence that the input was indeed prime can be increased to whatever level desired. (In these notes we will mainly discuss the probability tests only.)

A naive approach for testing whether an integer n, say of 200 digits, is prime or not is to divide it by all positive integers $a \geq 2$, or by all primes, which are smaller than $\sqrt{n}$. But there are around 4×10^{97} primes less than 10^{100}. This is far more than the number of atoms in the universe. If the computer can handle 10^9 primes per second the caluclation would take around 10^{81} years. Therefore, better methods are needed.

Date: November 14, 2007.

2. Probabilistic primality tests

A probabilistic primality test has the following general framework: For each odd positive n a set $W(n) \subseteq \mathbb{Z}_n = \{0, 1, \ldots, n-1\}$ is defined having the following properties:

1. given $a \in \mathbb{Z}_n$, it can be checked in deterministic polynomial time whether $a \in W(n)$;
2. if n is prime then $W(n) = \emptyset$;
3. if n is composite then $\#W(n) \geq n/2$.

Definition 2.1. If n is composite the elements of $W(n)$ are called *witnesses* (to the compositeness of n), and the elements of the complementary set $L(n) = \mathbb{Z}_n - W(n)$ are called *liars*.

Suppose that n is an (odd) integer whose primality is to be checked. An integer $a \in \mathbb{Z}_n$ is chosen at random, and it is checked whether $a \in W(n)$. If $a \in W(n)$, then the output of the test is "composite"; in this case n is said to *fail the primality test for the base* a, and n is *certainly* composite. If $a \notin W(n)$ then the output of the test is "prime", and n is said to *pass the primality test for base* a; however, in this case it is not certain that n is prime, and the declaration "prime" may be incorrect. (Therefore such a test should be more properly called a *compositeness test.*)

A single "witness" to the compositeness is enough to conclude with certainty that n is composite. On the other hand, successive independet runs of the test all of which return the answer "prime" allow the confidence that the input is indeed prime to be increased to whatover level is desired. If the test is run t times independently on the composite number n the probability that n is declared "prime" all t times is at most $(1/2)^t$, by condition 3.

Definition 2.2. An integer n which is believed to be prime on the basis of a probabilistic primality test is called a *probable prime*.

3. Fermat's test

For historical reason we discuss Fermat's test though it is not truly a probabilistic primality test since it usually fails for a special class of composites, the Carmichael numbers.

Recall Fermat's little theorem:

Theorem 3.1. *Let n be a prime number, and let a be an integer with $1 \leq a \leq n-1$. Then $a^{n-1} \equiv 1 \bmod n$.*

Therefore, finding a with $1 \leq a \leq n-1$ with $a^{n-1} \not\equiv 1 \bmod n$, then n cannot be prime, that is, a would be a witness to compositeness.

Definition 3.2. Let n be an odd composite integer. An integer a with $1 \leq a \leq n-1$ such that $a^{n-1} \not\equiv 1 \bmod n$ is called a *Fermat witness* (to compositeness) for n.

Conversely, an a between 1 and $n-1$ such that $a^{n-1} \equiv 1 \bmod n$ makes n appear to be prime, in the sense that it satisfies Fermat's little theorem for the base a.

Definition 3.3. Let n be an odd composite integer and a an integer with $1 \leq a \leq n-1$. Then n is said to be *pseudoprime to the base* a if $a^{n-1} \equiv 1 \bmod n$, and the integer a is called a *Fermat liar* (to primality) for n.

Example 3.4. The composite $n = 341 = 11 \cdot 31$ is a pseudoprime to the base 2, since $2^{340} \equiv 1 \bmod 341$.

Algorithm 3.5 (Fermat primality test)**.**
INPUT: an odd integer $n \geq 3$ and a security parameter $t \geq 1$
OUTPUT: an answer "prime" or "composite" to the question: "Is n prime?"

```
fermat := proc(n,t)
1. For i from 1 to t do
     1.1 Choose a random integer a between 2 and n−2
     1.2 Compute r := a&^n−1 mod n
     1.3 If r <> 1 then return(``composite'')
2. return(``prime'')
```

Remark 3.6. Of course, there are versions of this (and of the following) implementation. E.g. the security parameter can be chosen as $t = 1$, and one can perform the algorithm on a given base a.

If this algorithm declares "composite" then n is certainly composite. On the other hand, if it declare "prime" then no proof is provided that n is indeed prime. Nonetheless, pseudoprimes for a given base a are known to be rare, therefore Fermat's test provides a correct answer on most inputs. But this is distinct from providing a correct answer in most of the times (e.g. if run with different bases) on *every* input. In fact there are (even rarer) composite numbers which a pseudoprime to *every* base a for which $\gcd(a, n) = 1$:

Definition 3.7. A *Carmichael number* n is a composite integer such that $a^{n-1} \equiv 1 \bmod n$ for all integers a such that $\gcd(a, n) = 1$.

If n is a Carmichael number then the only Fermat witnesses for n are those integers a between 1 and $n-1$, for which $\gcd(a, n) > 1$. If, for example all prime factors of n are large, then with high probability the Fermat test declares n to be "prime", even if the number of iterations t is large. E.g., if $n = p \cdot q$ (p, q primes) the only witnesses for n are p and q.

It was only proved in 1994 by Alford, W. R.; Granville, A.; and Pomerance, C. that there are infinitely many Carmichael numbers.

4. The Miller-Rabin test

The Miller-Rabin primality test is based on the following refinement of Fermat's little theorem:

Theorem 4.1. *Let n be an odd prime. Write $n-1$ as $n-1=2^s r$ where r is odd. Let a be an integer such that $\gcd(a,n)=1$. Then either*

$$a^r \equiv 1 \bmod n$$

or

$$a^{2^j r} \equiv -1 \bmod n \quad \text{for some} \quad j,\ 0 \le j \le s-1.$$

Proof. If $a^r \equiv 1 \bmod n$ we are fine. Thus assume $a^r \not\equiv 1 \bmod n$. In particular $a^{2^0 r} \not\equiv 1 \bmod n$. By Fermat's little theorem we know $a^{2^s r} \equiv 1 \bmod n$. Thus, there must be some j, $1 \le j \le s$, such that $a^{2^j r} \equiv 1 \bmod n$ (e.g. $j=s$), but $a^{2^{j-1} r} \not\equiv 1 \bmod n$ (e.g. $j=1$). Let $b = a^{2^{j-1} r} \bmod n$. Then $b^2 \equiv a^{2^j r} \equiv 1 \bmod n$. We can reformulate this by saying that $n \mid b^2-1 = (b-1)(b+1)$. Since n is prime (!) it divides $b-1$ or $b+1$. Since $b \not\equiv 1 \bmod n$, it divides $b+1$, and hence $b \equiv -1 \bmod n$. (Remark: Here primality of n is needed, since in general (for n composite) 1 has more square roots modulo n then ± 1, e.g. $5^2 \equiv 1 \bmod 12$.) □

Definition 4.2. Let n be an odd composite integer. Write $n-1$ as $n-1=2^s r$ where r is odd. Let a be an integer with $1 \le a \le n-1$.

(1) If $a^r \not\equiv 1 \bmod n$ and if $a^{2^j r} \not\equiv -1 \bmod n$ for all j with $0 \le j \le s-1$, then a is called a *strong witness* (to compositeness) for n.
(2) Otherwise (i.e. if either $a^r \equiv 1 \bmod n$ or $a^{2^j r} \equiv -1 \bmod n$ for some j with $0 \le j \le s-1$), then a is said to be a strong pseudoprime to the base a. The integer a is called a *strong liar* (to primality) for n.

Algorithm 4.3 (Miller-Rabin probabilistic primality test)**.**
INPUT: an odd integer $n \ge 3$ and a security parameter $t \ge 1$
OUTPUT: an answer "prime" or "composite" to the question: "Is n prime?"

```
miller_rabin := proc(n,t)
1. Write n−1 =2^s r such that r is odd.
2. For i from 1 to t do
     2.1 Choose a random integer a between 2 and n−2
     2.2 Compute y := a&^r mod n
     2.3 If y <> 1 and y <> n−1 then do
           j := 1
           While j <= s−1 and y <> n−1 do
             Compute y := y&^2 mod n
             If y=1 then return(''composite'')
```

```
            j := j+1
        If y <> n−1 then return(''composite'')
3. Return(''prime'')
```

Fact 4.4. If n is an odd composite number then at most $1/4$ of all the numbers a with $1 \leq a \leq n-1$ are strong liars for n. It follows that if we run the Rabin-Miller test t times independently on the odd composite n (that is, with input (n,t)) then the probability that the algorithm outputs "prime" (each time) is at most $(1/4)^t$.

Remark 4.5. Let n be a composite integer. Each strong liar for n is also a Fermat liar. In fact, write $n-1 = 2^s r$ (with r odd). If $a^r \equiv 1 \bmod n$ or $a^{2^j r} \equiv -1 \bmod n$ for some j, $0 \leq j \leq s-1$, then also $a^{n-1} \equiv 1 \bmod n$, since a^{n-1} is obtained by a^r and $a^{2^j r}$ be squaring several times. Usually there are much more Fermat liars for n then strong liars.

Consider for example the composite $n = 65 = 5 \cdot 13$. Its Fermat liars are 1, 8, 12, 18, 21, 27, 31, 34, 44, 47, 51, 53, 57, 64, whereas the strong liars are only 1, 8, 18, 47, 57, 64.

5. Prime number generation

By the prime number theorem, the proportion of odd positive integers $\leq x$ that are prime is approximately $2/\ln x$. For example, the proportion of all odd integers $\leq 2^{512}$ that are prime is $\approx 1/177$. This suggests the reasonable strategy for selecting a random k-bit odd integer and check with the Miller-Rabin test whether it is (probable) prime. If not, select a different random odd k-bit number, and so on. Since often composite integers have small prime divisors it is more efficient to perform trial divisions before starting the Miller-Rabin test.

Algorithm 5.1 (Random search for a prime using the Miller-Rabin test)**.**

INPUT: An integer k and a security parameter t.
OUTPUT: a random k-bit probable prime.

```
random_search := proc(k,t)
1. Generate an odd k−bit integer n at random.
2. Use trial division checking whether n is divisbible by any
   odd prime <= B. If it is then goto step 1.
3. If miller_rabin(n,t) outputs ''prime'' then return(n).
   Otherwise goto step 1.
```

The explicit size of the parameter B depends on various parameters like the available computing power and so on.

For the RSA algorithm, apply `random_search(k,t)` independently twice. The prime numbers p and q, as part of the modulus $n = pq$

should be of the same size but not too close together:

$$\varepsilon_1 < |\log_2(p) - \log_2(q)| < \varepsilon_2$$

where something like $\varepsilon_1 \approx 0.1$ and $\varepsilon_2 \approx 30$ is proposed. Besides that the two numbers should be generated randomly and independently. The requirement that p and q should be so-called *strong primes* seems to be not longer justified and is therefore considered to be dispensable. (The recommended bit length's of the primes were already discussed before.) As upper bound for the risk (probability) that the probable prime numbers are actually composite the number 2^{-80} is propososed (until the end of 2009; later 2^{-100}).

6. Factorization

There are two related problems:

FACTORING. Integer factorization problem: Given a positive integer n, find its prime factorization.

SPLITTING. Given a positive integer n, find a non-trivial factorization $n = ab$ with $1 < a, b < n$.

Having an efficient algorithm which finds a nontrivial factor a of n then recursive application of this algorithm to a will compute the prime factorization.

The problem of deciding whether an integer is composite or prime seems to be in general much easier than the factoring problem. Hence before attempting to factor an integer the integer should be tested to make sure that it is indeed composite.

Like for primality tests there is *trial division*, which, in the worst case that n is a product of two primes of the same size, takes roughly $\sqrt{n}$ divisions. We will not further discuss this method.

7. Fermat factorization method

We need the following fact from basic numbertheory:

Lemma 7.1. *Let a, b and c integers. Assume $a \mid bc$. If $\gcd(a, b) = 1$ then $a \mid c$.*

Proof. There is an integer k such that $ka = bc$. Since $\gcd(a, b) = 1$, there are integers x and y such that $1 = ax + by$. Multiplication with c gives

$$c = axc + bcy = axc + kay = a(cx + ay),$$

hence $a \mid c$. □

We deduce the following basic principle:

Lemma 7.2. *Let n be an integer, and suppose there exist integers x and y with $x^2 \equiv y^2 \bmod n$, but $x \not\equiv \pm y \bmod n$. Then n is composite and $\gcd(x - y, n)$ is a nontrivial factor of n.*

Proof. A similar, slightly weaker statement was already shown in the proof of Fermat's primality test. Let $d = \gcd(x - y, n)$. $d = n$ is not possible (since $x \not\equiv y \bmod n$. If $d = 1$, then since n divides the product $(x - y)(x + y)$ by the preceding lemma it divides one of these factors, but this is impossible since $x \not\equiv \pm y \bmod n$. It follows that $d \neq 1, n$. Hence it is nontrivial factor of n. □

Example 7.3. Since $12^2 \equiv 2^2 \bmod 35$, but $12 \not\equiv \pm 2 \bmod 35$ we conclude that 35 is composite and $\gcd(12 - 2, 35) = 5$ are nontrivial factor of n.

This suggests the following, so-called *Fermat factorization method*, which is useful when n is a product of primes that are very close together: Compute $n + 1^2$, $n + 2^2$, $n + 3^2$, ... until we find a square. For example, if $n = 295927$, then $295927 + 3^2 = 295936 = 544^2$, therefore $295927 = (544 + 3)(544 - 3) = 547 \cdot 541$.

If $n = pq$, then it takes $|p-q|/2$ steps to find the factorization. But if p and q are two randomly chosen prime numbers with, say, 100 digits, then $|p - q|$ will be very large, probably also around 100 digits.

8. Pollard's $p - 1$ factorizing algorithm

Pollard's $p - 1$ factorizing algorithm a special-purpose factoring algorithm that is useful when applied to n which have a prime factor p of n such that $p - 1$ has "small" prime factors in a certain sense.

Definition 8.1. Let B a positive integer. A (positive) integer n is said to be B-*smooth*, if all its prime factors are $\leq B$.

For every real number x denote by $\lfloor x \rfloor$ (say "floor" of x) the largest integer which is smaller than or equal to x.

Let B be a smoothness bound. Let Q be a the least common multiple[1] of all powers of primes $\leq B$ that are $\leq n$. If $q^l \leq n$, then $l \ln q \leq \ln n$, and so $l \leq \lfloor \frac{\ln n}{\ln q} \rfloor$. It follows, that

$$Q = \prod_{q \leq B \text{ prime}} q^{\lfloor \ln n / \ln q \rfloor}.$$

If p is a prime factor of n such that $p-1$ is B-smooth, then $p-1 \mid Q$, and Fermat's theorem says that $a^Q \equiv 1 \bmod p$ for any a with $\gcd(a, p) = 1$. For $d = \gcd(a^Q - 1, n)$ we then get $p \mid d$. It may happen that $d = n$, in which case the algoritm fails. But this is very unlikely if n is a product of two large primes.

Algorithm 8.2 (Pollard's $p - 1$ algorithm for factorizing integers)**.**
INPUT: a composite integer n that is not a prime power.
OUTPUT: a non-trivial factor d of n.

[1]The least common multiple $v = \operatorname{lcm}(a, b)$ of two integers a and b is defined by the following properties: $v \geq 0$; $a \mid v$ and $b \mid v$; if v' is an integer such that $a \mid v'$ and $b \mid v'$, then $v \mid v'$. For example, $v \mid ab$.

```
pollard_p−1 := proc(n)
  1. Select a smoothness bound B.
  2. Select a random integer a between 2 and n−1
       and compute d := gcd(a,n).
       If d >= 2 then return(d).
  3. For each prime q <= B do
       3.1 Compute l := floor( ln(n)/ln(q) )
       3.2 Compute a := a^(q^l) mod n
  4. Compute d := gcd(a−1,n).
  5. If d=1 or d=n then terminate the algorithm with failure
       else return(d)
```

Example 8.3. We want to find a non-trivial factor of $n = 19048567$ using Pollard's $p-1$ algorithm.

1. Select the smoothness bound $B = 19$.
2. Select the integer $a = 3$ and compute $\gcd(3, n) = 1$.
3. The following table lists the intermediate values of the variables q, l and a after each iteration of step 3 in the algorithm:

q	l	a
2	24	2293244
3	15	13555889
5	10	16937223
7	8	15214586
11	6	9685355
13	6	13271154
17	5	11406961
19	5	554506

4. Compute $d = \gcd(554506 - 1, n) = 5281$.
5. Two non-trivial factors of n are $p = 5281$ and $q = n/p = 3607$ (these factors are actually prime).

Note that $p - 1 = 5280 = 2^5 \cdot 3 \cdot 5 \cdot 11$ and $q - 1 = 3606 = 2 \cdot 3 \cdot 601$, that is, $p - 1$ is 19-smooth but $q - 1$ is not.

Fact 8.4. Let n be an integer having a prime factor p such that $p - 1$ is B-smooth. The running time of Pollard's $p-1$ algorithm for finding the factor p is $O(B \ln n / \ln B)$ modular multiplications[2].

[2]Here we use the following notation: If f, $g : \mathbb{N} \to \mathbb{R}_+$ are functions, then we write

$$f(n) = O(g(n))$$

(for $n \to \infty$) if there is a constant $C > 0$ and an $n_0 \in \mathbb{N}$ such that $f(n) \leq C \cdot g(n)$ for all $n \geq n_0$.

9. Pollard's rho factoring algorithm

Let S be any finite set, for example, $S = \{0, 1, \ldots, n-1\}$. Let $f : S \to S$ be a function, called random function. Let x_0 be a random element in S. Consider the sequence $x_0, x_1, x_2, \ldots$ defined by $x_{i+1} = f(x_i)$ for $i \geq 0$. Since S is finite, there must by some m and some $c > 0$ such that $x_{m+c} = x_m$ (where we assume that m and c are minimal with this property), and then $x_{i+c} = x_i$ for all $i \geq m$, that is, the sequence becomes a cycle of length c after m steps. (One can imagine the picture with the shape of the greek letter ρ.) Then $\lambda = m$ is called the length of tail and $\mu = c$ the length of cycle. The expected value of $\lambda + \mu$ is $\sqrt{\pi n/2}$.

An obvious method for finding such a *collision* where $x_{m+c} = x_m$ is to compute and store the x_i for $i = 0, 1, 2, \ldots$ and look for duplicates. Therefore this method requires $O(\sqrt{n})$ memory and $O(\sqrt{n})$ time, if the x_i are stored in a hash table so that new entries can be added in constant time.

Remark 9.1 (Floyd's cycle-finding algorithm)**.** The large *storage* requirements in the above method for finding a collision can be eliminated: one starts with the pair (x_1, x_2), and iteratively computes (x_i, x_{2i}) (only) from the previous pair $(x_{i-1}, x_{2(i-1)})$, until $x_m = x_{2m}$. (Note that $x_i = f(x_{i-1})$ and $x_{2i} = f(f(x_{2(i-1)}))$.)

If the tail length is λ and the cycle length μ then $x_m = x_{2m}$ happens the first time when $m = \mu(1 + \lfloor \lambda/\mu \rfloor)$: In fact, doubling the value $\mu(1 + \lfloor \lambda/\mu \rfloor)$ is the same as adding to $\mu(1 + \lfloor \lambda/\mu \rfloor)$ to cycle length μ $k = (1 + \lfloor \lambda/\mu \rfloor)$-times. EXAMPLE. Since $\lambda < m < \lambda + \mu$ the expected running time of this algorithm is $O(\sqrt{n})$.

Let p be a (unknown) prime factor of a composite integer n. Pollard's rho algorithm for factoring n attempts to find duplicates in the sequence $x_0, x_1, x_2, \ldots$ defined by

$$x_0 = 2, \; x_{i+1} = f(x_i) = x_i^2 + 1 \bmod p \;\text{ for }\; i \geq 0.$$

Floyd's cycle-finding algorithm finds m such that $x_m \equiv x_{2m} \bmod p$. But p is unknown. Therefore we compute the x_i modulo n and tests if $d = \gcd(x_i - x_{2i}, n) > 1$. This happens at least for $i = m$ (since then p is a common divisor). If also $d < n$ then a non-trivial factor d is found. ($d = n$ happens very rarely.)

Algorithm 9.2 (Pollard's rho algorithm for factorizing integers)**.**
INPUT: a composite integer n that is not a prime power.
OUTPUT: a non-trivial factor d of n.

```
pollard_rho := proc(n)
  1. Set a := 2 and b := 2.
  2. For i = 1,2,3,... do
       2.1 Compute a := a^2 + 1 mod n,
```

b := b^2 + 1 mod n,
b := b^2 + 1 mod n.
2.2 Compute d = gcd(a−b,n)
2.3 If 1 < d < n then **return**(d) and terminate with success.
2.4 If d=n then terminate with failure.

Example 9.3. Let $n = 455459$. Step 2 of Pollard's rho algorithm computes the entries in the following table.

a	b	d
5	26	1
26	2871	1
677	179685	1
2871	155260	1
44380	416250	1
179685	43670	1
121634	164403	1
155260	247944	1
44567	68343	743

We conclude that two non-trivial factors of 455459 are 743 and $455459/743 = 613$.

Fact 9.4. Assuming that the function $f(x) = x^2 + 1 \bmod p$ behaves like a random function the expected time for Pollard's rho algorithm to find a factor p of n is $O(\sqrt{p})$ modular multiplications. This implies that the expected time to find a non-trivial factor of $n = pq$ ($p \neq q$ of same size) is $O(n^{1/4})$.

10. Quadratic sieve factoring

Recall the basic principle 7.2:

Suppose there exist integers x and y with $x^2 \equiv y^2 \bmod n$, but $x \not\equiv \pm y \bmod n$. Then $\gcd(x - y, n)$ is a non-trivial factor of n.

The Fermat factoring method is one simple example of the *random square methods*. These try to find integers x and y at random such that $x^2 \equiv y^2 \bmod n$. If $n = pq$ is a product of different primes, then the congruence $x^2 \equiv a^2 \bmod n$ has precisely 4 solutions modulo n, two of which are $x = a$ and $x = -a$.

Example 10.1. Let $n = 35$. The equation $x^2 \equiv 4 \bmod 35$ has the solutions $x = 2$, 12, 23 and 33.

Thus, if $n = pq$, and if x and y with $x^2 \equiv y^2$ are randomly chosen, then $x \not\equiv \pm y \bmod n$ with probability $1/2$. (If n is not a prime power, then this probability is $\geq 1/2$.)

A common random square strategy for finding x and y at random with $x^2 \equiv y^2 \bmod n$ is the following: Start with a set $S = \{p_1, p_2, \ldots, p_t\}$ of the first t primes. S is called *factor base*. For every $i = 1, \ldots, t$ find pairs (a_i, b_i) satisfying

(i) $a_i^2 \equiv b_i \bmod n$;

(ii) $b_i = \prod_{j=1}^{t} p_j^{e_{ij}}$, with $e_{ij} \geq 0$. (That is, b_i is p_t-smooth.)

Next find a subset of the b_i's whose product is a perfect square, that is, find $T \subseteq \{1, \ldots, t\}$ such that $\prod_{i \in T} b_i$ is a perfect square. Thus, letting y to be the integer square root of $\prod_{i \in T} b_i$ and $x \stackrel{def}{=} \prod_{i \in T} a_i$ then obviously $x^2 \equiv y^2 \bmod n$ holds. If also $x \not\equiv \pm y \bmod n$, then we get a non-trivial factor of n. If not, we replace some of the pairs (a_i, b_i) by new pairs. With high probabilty this will work.

In order to get such a subset T, we have to form a product of the b_i's such that the power of each prime p_j appearing in their product is even. Thus, only the parity of the exponents e_{ij} needs to be considered. For each i we associate with the vector of exponents $(e_{i1}, e_{i2}, \ldots, e_{it})$ the binary vector $(v_{i1}, v_{i2}, \ldots, v_{it})$, where $v_{ij} = e_{ij} \bmod 2$. If we have produced $t+1$ pairs, then the corresponding $t+1$ binary vectors must be linearly dependend as elements in the t-dimensional vector space over $\mathbb{Z}_2$. That is, there must be some subset $T \subseteq \{1, \ldots, t+1\}$ such that $\sum_{i \in T} v_i = 0$ (over $\mathbb{Z}_2$). Then we can procede as describe just before.

We have to explain how to produce pairs (a_i, b_i) fulfilling the above properties. Let $m = \lfloor \sqrt{n} \rfloor$. Consider the polynomial

$$q(x) = (x + m)^2 - n.$$

We have

$$q(x) = x^2 + 2mx + m^2 - n \approx x^2 + 2mx,$$

which is small relative to n if x is small in absolute value. The quadratic sieve algorithm selects $a_i = x + m$ and tests whether $b_i = q(x)$ is p_t-smooth. Note that $a_i^2 \equiv b_i \bmod n$.

If p is prime and dividing b_i then $(x+m)^2 \equiv n \bmod p$. That is, n is a square modulo[3] the prime p. Thus the factor base needs only contain such primes. Since b_i may be negative, -1 is included in the factor base.

Algorithm 10.2 (Quadratic sieve algorithm for factoring integers)**.**

INPUT: a composite integer n that is not a prime power.
OUTPUT: a non-trivial factor d of n.

quadratic_sieve := proc(n)

1. Select the factor base $S = \{p_1, ..., p_t\}$ where $p_1 = 1$ and p_j is

[3]This can be easily checked: Let p be an odd prime. An integer n such that $p \nmid n$ is a square modulo p if and only if $n^{(p-1)/2} \equiv 1 \bmod p$.

the $(j-1)-$th prime p for which n is a square modulo p.

2. Compute $m := \lfloor \sqrt{n} \rfloor$.
3. Set $i := 1$. **While** $i \leq t+1$ **do**
   3.1 Compute $b := (x+m)^2 - n$ and test using trial division by elements in S whether b is p_j-smooth. If not, pick up a new x and repeat step 3.1. (x values are chosen in the order $0, \pm 1, \pm 2, \ldots$)
   3.2 If b is p_t-smooth and
   $b = p_1^e(i,1) \cdot \ldots \cdot p_t^e(i,t)$ then
   set $a_i := x+m$, $b_i := b$ and
   for j from 1 to t **do** $v(i,j) := e(i,j) \bmod 2$.
   Set $v_i := (v(i,1), \ldots v(i,t))$.
   3.3 Set $i := i+1$.
4. Find a non$-$empty subset T of $\{1, \ldots, t+1\}$ such that the sum of all v_i (where $i \in T$) is $0 \mod 2$ (using linear algebra)
5. Set $x := 1$. For all i in T **do** $x := x \cdot a_i \bmod n$.
6. For j from 1 to t **do**
   $l_j := 0$.
   for i in T **do**
   compute $l_j := l_j + e(i,j)$.
   $l_j := l_j/2$.
7. Set $y := 1$.
   For j from 1 to t **do**
   $y := y \cdot (p_j)^{l_j} \bmod n$.
8. If $x = y \bmod n$ or $x = -y \bmod n$ then find another subset T as above and **goto** step 5.
   (In the unlikely case such a subset T does not exist, replace a few of the pairs (a_i, b_i) with new pairs (in step 3) and go to step 4 again.)
9. Compute $d := \gcd(x-y, n)$ and **return**(d).

Example 10.3. $n = 24961$.

1. Select the factor base $S = \{-1, 2, 3, 5, 13, 23\}$ of size $t = 6$. (The primes 7, 11, 17 and 19 are omitted since n is not a square modulo them.)
2. Compute $m = \lfloor \sqrt{24961} \rfloor = 157$.
3. For the first $t+1$ values of x for which $q(x)$ is 23-smooth we get the data listed in the following table:
4. one "sees" $v_1 + v_2 + v_5 = 0$. That is, $T = \{1, 2, 5\}$.
5. Compute $x = a_1 a_2 a_5 \bmod n = 936$.
6. Compute $l_1 = 1$, $l_2 = 3$, $l_3 = 2$, $l_4 = 0$, $l_5 = 1$, $l_6 = 0$.
7. Compute $y = -2^3 \cdot 3^2 \cdot 13 \bmod n = 24025$.
8. Since $936 \equiv -24025 \bmod n$ another linearly dependency must be found.
9. We see $v_3 + v_6 + v_7 = 0$, thus $T = \{3, 6, 7\}$.

i	x	$q(x)$	factorization of $q(x)$	a_i	v_i
1	0	-312	$-2^3 \cdot 3 \cdot 13$	157	$(1,1,1,0,1,0)$
2	1	3	3	158	$(0,0,1,0,0,0)$
3	-1	-625	-5^4	156	$(1,0,0,0,0,0)$
4	2	320	$2^6 \cdot 5$	159	$(0,0,0,1,0,0)$
5	-2	-936	$-2^3 \cdot 3^2 \cdot 13$	155	$(1,1,0,0,1,0)$
6	4	960	$2^6 \cdot 3 \cdot 5$	161	$(0,0,1,1,0,0)$
7	-6	-2160	$-2^4 \cdot 3^3 \cdot 5$	151	$(1,0,1,1,0,0)$

10. Compute $x = a_3 a_6 a_7 \bmod n = 23405$.
11. Compute $l_1 = 1$, $l_2 = 5$, $l_3 = 2$, $l_4 = 3$, $l_5 = 0$, $l_6 = 0$.
12. Compute $y = -2^5 \cdot 3^2 \cdot 5^3 \bmod n = 13922$.
13. Now, $23405 \not\equiv \pm 13922 \bmod n$, so we compute $\gcd(x - y, n) = \gcd(9483, 24961) = 109$. Hence two non-trivial factors are of 24961 are 109 and $24961/109 = 229$.

Remark 10.4. For information about the running time of the quadratic sieve method and how to choose t we refer to [1, 3.23, 3.24]. There is also explained the actual *sieving process* which played no role in our short presentation. More advanced techniques like *elliptic curve factoring* and the *number field sieve factoring* are beyond the scope of this lecture.

11. Finding primitive roots

Let p be a prime number. Let $n = p - 1$. There are $\varphi(n)$ primitive roots mod p. If we pick randomly an integer a with $1 \leq a \leq n$, then the probability that it is a primitive root is $\varphi(n)/n$. Whether a given a is primitive or not can be checked by testing whether n is the order of $a \pmod p$, that is, whether n is the *smallest* positive integer such that $a^n \equiv 1 \bmod p$. This can be done by factoring $n = \prod_{i=1}^{t} p_i^{e_i}$ into prime factors, and checking for each $i = 1, \ldots, t$ whether $a^{n/p_i} \equiv 1 \bmod p$ or not. If yes, then a is not primitive. If not (for all i), then a is primitive.

Since by some known lower bound for the Euler function we have $\varphi(n)/n \geq 1/(6 \log\log n)$ there is an efficient randomized algorithm (if the prime factorization of $n = p - 1$ is known) to choose a randomly until the test that n is the smallest positive integer with $a^n \equiv 1 \bmod p$ is positive. (The reader may implement this simple algorithm in a pseudo-programming language as an exercise.)

In cryptographic applications for which a primitive root mod p is required one usually has the flexebility of selecting the prime p. The above method is relatively inefficient since one has to compute the prime factorization of $n = p - 1$. One can avoid this in the following way: One first choses a large prime q and then selects a relative small integer R at random until $p = 2Rq + 1$ is prime. Then $p - 1 = 2Rq$,

and the factorization of $p-1$ can be obtained by factoring R. In the particular nice case when $R=1$, the prime p is called a *safe prime*.

12. A birthday attack on discrete logarithms

Suppose p is a prime and we want to solve $\alpha^x \equiv \beta \bmod p$. Of course, one could try any number $x = 1, 2, \ldots, p-1$ and check whether $\alpha^x \equiv \beta$. But if p is large this is not useful.

With high probability the problem can be solved by a birthday attack, if the size of $\sqrt{p}$ is in a range where computations are still managable.

Make two lists, both of length around $\sqrt{p}$:

1. The first list contains numbers $\alpha^k \bmod p$ for about $\sqrt{p}$ randomly chosen values of k.
2. The second list contains numbers $\beta\alpha^{-\ell} \bmod p$ for about $\sqrt{p}$ randomly chosen values of ℓ.

There is a good chance that there is a match between some element of the first list and some element of the second list. In that case we have

$$\alpha^k \equiv \beta\alpha^{-\ell}, \text{ hence } \alpha^{k+\ell} \equiv \beta \bmod p.$$

Thus, $x \equiv k+\ell \bmod p-1$ is the desired discrete logarithm.

13. The baby-step giant-step algorithm

A similar but determinstic way is the so-called baby-step giant-step algorithm for computing discrete algorithms. It is also called Shanks' algorithm.

Let p be a prime and α be a primitive root mod p. Let $n = p-1$ and $m = \lceil\sqrt{n}\rceil$. If $\beta \equiv \alpha^x$ we can write $x = im + j$ where $0 \le i,\, j < m$. Hence $\beta(\alpha^{-m})^i = \alpha^j$. This suggest the following algorithm:

Algorithm 13.1 (Baby-step giant-step algorithm for computing discrete logarithms)**.**
INPUT: a primitive root α mod p and an integer β, $0 \le \beta \le n = p-1$.
OUTPUT: $x = L_\alpha(\beta)$.

baby_giant_step := proc(p,α,β)
1. Set $m := \lceil\sqrt{p-1}\rceil$.
2. Construct a table with entries $(j, \alpha^j \bmod p)$ for $0 \le j < m$.
   Sort this table by the second component.
3. Compute α^{-m} and set $\gamma := \beta$.
4. For i from 0 to $m-1$ **do**
   4.1 Check if γ is the second component of some entry in the table.
   4.2 If $\gamma = \alpha^j \bmod p$ then **return**$(x = im + j)$.
   4.3 Set $\gamma := \gamma\alpha^{-m} \bmod p$.

Fact 13.2. The running time of this algorithm is $O(\sqrt{n})$ modular multiplications.

Example 13.3. Let $p = 113$. Then $\alpha = 3$ is a primitive root mod p. Consider $\beta = 57$. Then $L_3(57)$ is calculated as follows.

1. Set $m = \lceil \sqrt{112} \rceil = 11$.
2. Construct a table with entries $(j, \alpha^j \bmod p)$ for $0 \le j < 11$

j	0	1	2	3	4	5	6	7	8	9	10
$3^j \bmod 113$	1	3	9	27	81	17	51	40	7	21	63

and sort by the second component:

j	0	1	8	2	5	9	3	7	6	10	4
$3^j \bmod 113$	1	3	7	9	17	21	27	40	51	63	81

3. Compute $\alpha^{-1} = 3^{-1} \bmod 113 = 38$, using the extended Euclidean algorithm. Then compute $\alpha^{-m} = 38^{11} \bmod 113 = 58$.
4. Next, $\gamma = \beta\alpha^{-mi} \bmod 113$ is computed for $i = 0, 1, 2, \ldots$ until a value in the second row of the table above is obtained. This yields

j	0	1	2	3	4	5	6	7	8	9
$\gamma = 57 \cdot 58^i \bmod 113$	57	29	100	37	112	55	26	39	2	3

Since $\beta\alpha^{-9m} \equiv 3 \equiv \alpha^1$, $\beta = \alpha^{100}$ we get $L_3(57) = 100$.

14. The Pohlig-Hellman algorithm

Let p be a prime and α be a primitive root mod p. Let $n = p - 1$ and $n = \prod_{i=1}^{t} p_i^{e_i}$ the prime factorization of n. If $x = L_\alpha(\beta)$ then the approach is to determine $x_i \equiv x \bmod p_i^{e_i}$ for each i, $1 \le i \le t$, and then use the chinese remainder theorem to recover $x \bmod n$.

Each integer x_i can be written in its p_i-ary representation:

$$x_i = l_0 + l_1 p_i + \cdots + l_{e_i - 1} p_i^{e_i - 1} \quad \text{with } 0 \le l_j \le p_i - 1.$$

Algorithm 14.1 (The Pohlig-Hellman algorithm for computing discrete logarithms)**.**
INPUT: a primitive root α mod p and an integer β, $0 \le \beta \le n = p-1$.
OUTPUT: $x = L_\alpha(\beta)$.

pohlig_hellman := proc(p,α,β)

1. Find the prime factorization of n: $n = p_1^{e_1} p_2^{e_2} \ldots p_t^{e_t}$
2. For i from 1 to t **do**
   - 2.1 Set $q := p_i$ and $e := e_i$.
   - 2.2 Set $\gamma := 1$ and $l_{-1} := 0$.
   - 2.3 Compute $\overline{\alpha} := \alpha^{n/q}$.
   - 2.4 For j from 0 to $e - 1$ **do**
     - Compute $\gamma := \gamma \alpha^{l_{j-1} q^{j-1}}$ and $\overline{\beta} := (\beta\gamma^{-1})^{n/q^{j+1}}$.
     - Compute $l_j := L_{\overline{\alpha}}(\overline{\beta})$ (e.g. using Algorithm 13.1).
   - 2.5 Set $x_i := l_0 + l_1 q + \cdots + l_{e-1} q^{e-1}$.
3. Use the Chinese remainder theorem to compute integer x with $0 \le x \le n - 1$ such that $x \equiv x_i \bmod p_i^{e_i}$ for $1 \le i \le t$.

4. **Return**(x).

Note that the l_j are computed in the right way. After iteration step j in 2., we have to show that $l_j = L_{\overline{\alpha}}(\overline{\beta})$. In fact, in step 2.4 we have $\gamma = \alpha^{l_0+l_1q+\cdots l_{j-1}q^{j-1}}$, and

$$\begin{aligned}\overline{\beta} &\equiv (\beta/\gamma)^{n/q^{j+1}} \equiv (\alpha^{x-l_0-l_1q-\cdots-l_{j-1}q^{j-1}})^{n/q^{j+1}} \\ &\overset{(*)}{\equiv} (\alpha^{n/q^{j+1}})^{x_i-l_0-l_1q-\cdots-l_{j-1}q^{j-1}} \\ &\equiv (\alpha^{n/q^{j+1}})^{l_jq^j+\cdots+l_{e-1}q^{e-1}} \\ &\equiv (\alpha^{n/q})^{l_j+\cdots+l_{e-1}q^{e-1-j}} = \overline{\alpha}^{l_j}.\end{aligned}$$

Note that in $(*)$ we could replace x by x_i since $x \equiv x_i \bmod q^e$, and then $(\alpha^{n/q^{j+1}})^x \equiv (\alpha^{n/q^{j+1}})^{x_i}$ since $j+1 \le e$.

Example 14.2. Let $p = 251$. The element $\alpha = 71$ is a primitive root mod p. Consider $\beta = 210$. Then $x = L_{71}(210)$ is computed as follows:

1. The prime factorization of n is $250 = 2 \cdot 5^3$.
2. (a) (Compute $x_1 = x \bmod 2$.)
   Compute $\overline{\alpha} = \alpha^{n/2} \bmod p = 250$ and $\overline{\beta} = \beta^{n/2} \bmod p = 250$. Then $x_1 = L_{250}(250) = 1$.
   (b) (Compute $x_2 = x \bmod 5^3 = l_0 + l_1 5 + l_2 5^2$.)
   (i) Compute $\overline{\alpha} = \alpha^{n/5} \bmod p = 20$.
   (ii) Compute $\gamma = 1$ and $\overline{\beta} = (\beta\gamma^{-1})^{n/5} \bmod p = 149$. Compute $l_0 = L_{20}(149) = 2$.
   (iii) Compute $\gamma = \gamma\alpha^2 \bmod p = 21$ and $\overline{\beta} = (\beta\gamma^{-1})^{n/25} \bmod p = 113$. Compute $l_1 = L_{20}(113) = 4$.
   (iv) Compute $\gamma = \gamma\alpha^{4\cdot 5} \bmod p = 115$ and $\overline{\beta} = (\beta\gamma^{-1})^{n/125} \bmod p = 149$. Compute $l_2 = L_{20}(149) = 2$.
   Hence $x_2 = 2 + 4 \cdot 5 + 2 \cdot 5^2 = 72$.
3. Solve the pair of congruences

$$\begin{aligned}x &\equiv 1 \mod 2 \\ x &\equiv 72 \mod 125\end{aligned}$$

to get $x = L_{71}(210) = 197$.

Fact 14.3. Given the factorization of n, the running time of the algorithm is $O\big(\sum_{i=1}^r e_i(\log n + \sqrt{p_i})\big)$ modular multiplications. So the Pohlig-Hellman alogorithm is efficient only if each prime divisor p_i of $n = p - 1$ is relatively small.

15. The index-calculus algorithm

The index-calculus algorithm is the most powerful method known for computing discrete logarithm. It is similar to the quadratic sieve methods for factorization.

Algorithm 15.1 (The index-calculus algorithm for computing discrete logarithms)**.**
INPUT: a primitive root α mod p and an integer β, $0 \leq \beta \leq n = p-1$.
OUTPUT: $x = L_\alpha(\beta)$.

index_calculus := proc(p,α,β)

1. (Select a factor base S) Choose a subset $S = \{p_1, p_2, \ldots, p_t\}$ of integers p_i with $1 \leq p_i \leq n = p-1$ such that a "significant proportion" of all integers a with $1 \leq a \leq n$ can be efficiently expressed as a product of elements from S.
2. (Collect linear relations involving logarithms of elements in S)
   2.1 Select a random integer k with $0 \leq n \leq n-1$ and compute α^k.
   2.2 Try to write α^k as a product of elements in S:
   $$\alpha^k \equiv \prod_{i=1}^{t} p_i^{c_i} \mod p, \quad c_i \geq 0.$$
   If successful, take logarithms of both sides to obtain a linear relation
   $$k = \sum_{i=1}^{k} c_i L_\alpha(p_i) \mod n.$$
   2.3 Repeat steps 2.1 and 2.2 until $t+c$ such relations are obtained (c is a small positive integer, e.g. $c = 10$, such that the system of equations given by the $t+c$ relations has a unique solution with high probability).
3. (Find the logarithm of elements in S) Working modulo n, solve the linear system of $t+c$ equations in t unknowns collected in step 2 to obtain the values of $L_\alpha(p_i)$.
4. (Compute x)
   4.1 Select a random integer k with $0 \leq k \leq n-1$ and compute $\beta\alpha^k$.
   4.2 Try to write $\beta\alpha^k$ as a product of elements in S:
   $$\beta\alpha^k \equiv \prod_{i=1}^{t} p_i^{d_i} \mod p, \quad d_i \geq 0$$
   If unsuccessful then repeat step 4.1. Otherwise taking logarithms of both sides yields $L_\alpha(\beta) = \sum_{i=1}^{t} d_i L_\alpha(p_i) - k \bmod n$; thus compute $x = \sum_{i=1}^{t} d_i L_\alpha(p_i) - k \bmod n$ and **return**(x).

Example 15.2. Let $p = 229$. The element $\alpha = 6$ is a primitive root mod p. Consider $\beta = 13$. Then $L_6(13)$ is computed as follows:

1. The factor base is chosen to be the first 5 primes: $S = \{2, 3, 5, 7, 11\}$.
2. The following six relations involving elements of the factor base are obtained (unsuccessful attempts are not shown):

$$\begin{aligned}
6^{100} &\mod 229 = 180 &&= 2^2 \cdot 3^2 \cdot 5 \\
6^{18} &\mod 229 = 176 &&= 2^4 \cdot 11 \\
6^{12} &\mod 229 = 165 &&= 3 \cdot 5 \cdot 11 \\
6^{62} &\mod 229 = 154 &&= 2 \cdot 7 \cdot 11 \\
6^{143} &\mod 229 = 198 &&= 2 \cdot 3^2 \cdot 11 \\
6^{206} &\mod 229 = 210 &&= 2 \cdot 3 \cdot 5 \cdot 7.
\end{aligned}$$

We thus get the following six relations:

$$\begin{aligned} 100 &\equiv 2L_6(2) + 2L_6(3) + L_6(5) \mod 228 \\ 18 &\equiv 4L_6(2) + L_6(11) \mod 228 \\ 12 &\equiv L_6(3) + L_6(5) + L_6(11) \mod 228 \\ 62 &\equiv L_6(2) + L_6(7) + L_6(11) \mod 228 \\ 143 &\equiv L_6(2) + 2L_6(3) + L_6(11) \mod 228 \\ 206 &\equiv L_6(2) + L_6(3) + L_6(5) + L_6(7) \mod 228. \end{aligned}$$

3. Solving the linear system of equations yields the solutions $L_6(2) = 21$, $L_6(3) = 208$, $L_6(5) = 98$, $L_6(7) = 107$ and $L_6(11) = 162$.
4. Suppose the integer $k = 77$ is selected. Since $\beta\alpha^k = 13 \cdot 6^{77} \bmod 229 = 147 = 3 \cdot 7^2$ we get

$$L_6(13) = L_6(3) + 2L_6(7) - 77 \mod 228 = 117.$$

Remark 15.3. For the running time and further more details compare [1, 3.71].

References

[1] A. J. Menezes, P. C. van Oorschot, S. A. Vanstone: *Handbook of Applied Cryptography*. CRC Press, 1997. Available in the internet `http://www.cacr.math.uwaterloo.ca/hac/`

[2] W. Trappe and L. Washington: *Introduction to Cryptography with Coding Theory*, (2nd edition). Pearson Prentice Hall, Upper Saddle River, 2006.