THE MATHEMATICS OF PUBLIC KEY CRYPTOGRAPHY.

IAN KIMING

1. Introduction.

The title is clearly far too ambitious: Modern Public Key cryptography is a large area with many ramifications of both theoretical and technical nature.

I will limit myself to a discussion of a basic example of a Public Key cryptosystem as well as 1 of the mathematical issues that arises in connection with a security analysis of the system. This issue is the question about possible effective methods of computing the prime factorization of integers, — a topic of independent mathematical interest.

Let us agree that a cryptosystem consists of an injective map f of $\mathbb{Z}/\mathbb{Z}n$ — the set (ring) of residues modulo n — into itself (here, n is a 'large' natural number): The elements of $\mathbb{Z}/\mathbb{Z}n$ represent possible messages that the system will transmit in encrypted form (to get from 'real-world messages', consisting for instance of long strings written in some alphabet, to the set $\mathbb{Z}/\mathbb{Z}n$ one can use so-called 'hash functions', but this is another story that we will not go into); now, if B(ob) wants to send a message $a \in \mathbb{Z}/\mathbb{Z}n$ to A(lice) then he computes $f(a)$ and transmits this to A; using a *certain piece of knowledge* — that depends on the specific system f — A is able to compute $a = f^{-1}(f(a))$ from $f(a)$.

The security of the system now obviously depends on the impossibility, or at least impossibility from a practical point of view, of computing a from $f(a)$ *without* A's 'certain piece of knowledge': For otherwise an eavesdropper E(ve) could listen to the transmission, discover $f(a)$, and thus do what A can do. (We are here discussing classical electromagnetic transmission and not more recent transmission systems that rely on quantum mechanical principles and where it is not possible to discover $f(a)$ without destroying the message).

Take a very simple example (that, however, has played a large role historically): The map f could be a simple cyclic permutation $f(x) := x + b$ for some fixed $b \in \mathbb{Z}/\mathbb{Z}n$. The 'certain piece of knowledge' that A has is here simply the knowledge of f: For if f is known, that is, if b is known, we can immediately determine $f^{-1}(x) = x - b$. So in this case, security certainly depends on f being kept secret.

This causes a major inconvenience: For B clearly has to know f, and so does A (at least A must know f^{-1} but that is essentially the same as knowing f in this case). So if A has chosen the system f then how does she communicate this choice to B? A little reflection reveals that there really is no alternative to informing B about the choice of f via some kind of physical exchange (A and B have to meet physically, or use a courier). This is not acceptable in an era of fast, large-scale electronic communication.

Public Key cryptosystems are systems where f, the system itself, can safely be publicly known and yet it is still (believed to be) impossible from a practical point of view to determine a from $f(a)$. If such systems can be easily constructed in large numbers then every user A can have her own system f_A that is publicly known. The system f_A is essentially A's 'public key'. Anybody who wants to send A an encrypted message simply looks up f_A in some central database connected to the internet and the uses f_A to send an encrypted message $f_A(a)$ to A.

Let us now look at one of the 2 most basic examples of such 'Public Key cryptosystems', namely the RSA system.

2. The RSA system.

The RSA system is named after its inventors: Rivest, Shamir, Adleman who introduced the system in 1978, cf. [7]. The system and its use is very easy to describe:

User A wants to construct her own system. She chooses 2 (large) distinct prime numbers p and q, and computes:

$$n_A := pq \ .$$

By the Chinese Remainder Theorem we then have:

$$(*) \qquad \mathbb{Z}/\mathbb{Z}n_A \cong \mathbb{Z}/\mathbb{Z}p \times \mathbb{Z}/\mathbb{Z}q \ ;$$

this is an isomorphism of rings so one sees that the number of multiplicatively invertible elements of $\mathbb{Z}/\mathbb{Z}n_A$ is:

$$(**) \qquad \phi(n_A) = (p-1)\cdot(q-1) \ .$$

Here, ϕ is Euler's ϕ-function, — $\phi(m)$ denotes the number of multiplicatively invertible elements of the ring $\mathbb{Z}/\mathbb{Z}m$; this is precisely the number of integers c in the interval $[1, m[$ that are relatively prime to m, i.e., for which $\gcd(c, m)$ — the greatest common divisor with m — is 1. For n_A as above this means exactly that the 2 'components' of c under the Chinese remainder isomorphism are both non-zero.

Having computed $\phi(n_A)$ user A now additionally chooses a natural number e_A ($< n_A$) relatively prime to $\phi(n_A)$. Thus, e_A is multiplicatively invertible modulo $\phi(n_A)$; that is, there exists an integer d_A such that:

$$(***) \qquad e_A d_A \equiv 1 \quad (\phi(n_A)) \ ,$$

in fact, d_A is easily determined by the Euclidean algorithm.

The pair (n_A, e_A) is A's 'Public Key' and is made publicly available. The number d_A is called A's 'Private Key' and is kept secret.

If now B wants to send a message $a \in \mathbb{Z}/\mathbb{Z}n_A$ to A, he computes and sends

$$a^{e_A} \mod n_A \ .$$

Now, since A knows the private key d_A she can compute:

$$(a^{e_A} \mod n_A)^{d_A} = (a^{e_A d_A} \mod n_A) = (a \mod n_A) \ ,$$

where the last equality follows thus: By $(**)$ and $(***)$ we have

$$e_A d_A = 1 + k \cdot (p-1)(q-1)$$

for some $k \in \mathbb{Z}$. Using the ring isomorphism $(*)$ it is thus enough to prove:

$$a^{1+k\cdot(p-1)(q-1)} = a$$

for any a in either $\mathbb{Z}/\mathbb{Z}p$ or $\mathbb{Z}/\mathbb{Z}q$. But this follows immediately from Fermat's little theorem that if ℓ is prime then $a^{\ell-1}$ is 0 or 1 in $\mathbb{Z}/\mathbb{Z}\ell$ according to whether a is 0 or not in $\mathbb{Z}/\mathbb{Z}\ell$.

An inspection of the scheme reveals that security of the system certainly requires A to keep the numbers p and q secret. For once these primes are known anybody can compute $\phi(n_A) = (p-1)(q-1)$, and then — since e_A is publicly known — A's private key d_A.

In other words, the security of the system definitely depends on the computational intractability of factoring a large number $n_A = pq$. This question is the question of independent mathematical interest coming from the RSA system (in fact, historically it was the other way around: RSA was constructed precisely because of the supposed difficulty of factoring large numbers).

Before we get into a sketch of what can be done with the problem of factoring we will need to go briefly into the precise meaning of notions such as 'computationally intractable problem'.

3. Computational complexity.

The theory of computational complexity seeks to estimate the work required to perform one type of computational task or the other. A good measure of this work would be the number of processor cycles required, or, what amounts to essentially the same, the number of really basic operations such as adding 2 bits required for the task.

The 'tasks' that we are talking about could be any problem that is known to be computationally solvable in at least 1 way. We are then asking questions about the most efficient way of solving the problem, in other words, about the least time-consuming algorithm that solves the problem.

Now, this question is not particularly interesting unless we are talking about some kind of a natural, infinite family of problems. For instance, we could be talking about the computational problem of adding 2 natural numbers m and n. Obviously, the time required to compute $m+n$ depends on the size of these numbers. The natural question to ask is about the asymptotic behavior of the minimal time requirement (using any algorithm) in dependence of the size of the input (m, n). Usually, it is too hard a problem to ask for the actual minimal time: It may simply be hopeless to obtain certainty that we know every imaginable algorithm that solves the problem. Hence, the normal situation is that one just has an *asymptotic upper bound* on the time requirement. An upper bound can be found by considering 1 particular algorithm that solves the problem and analyzing that particular algorithm in detail.

We still have to say how we measure the 'size' of an input. A natural measure of the size of a natural number n is the time needed to read it, or, equivalently, the number of digits when the number is written in (say) binary. But that number is $[\log_2 n] + 1$ which is (roughly) proportional to $\log n$. So we simply define $\log n$ as the size of a natural number n (being a mathematician rather than a computer scientist, I tend to like the natural logarithm log better than the logarithm to base

2; I will not even consider the possibility of bringing the logarithm to base 10 into the game ...).

Thus, the asymptotics that we talked about above should take the form of an upper bound of the time required to solve the problem, and expressed as a function of the logarithms of the natural numbers that are taken as input.

For instance, it is easy to see that an addition of numbers m and n can be performed in time

$$O(\log n)$$

if we assume that $n \geq m$ (write the numbers in binary and use the normal school algorithm for addition; remember that we have to count the number of times that we have to add 2 bits).

We used Big-O notation here, and the reason is that we really do not care much about constants in these asymptotic estimates. A much bigger picture is the primary concern:

We say that an algorithm performing some computation on an input $(n_1, \ldots, n_k)$ of natural numbers is a *polynomial time algorithm* if it executes in time bounded by a polynomial in $\log n_1, \ldots, \log n_k$.

A (family of) problem(s) for which we have a polynomial time algorithm is considered computationally 'easy'. On the other hand, if all we have is an algorithm that requires *exponential time*, i.e., time proportional to a fixed power of the largest input number, then the problem is considered computationally intractable.

All the basic arithmetic operations such as addition, multiplication, division with remainder, the Euclidean algorithm, modular arithmetic, can be performed with polynomial time algorithms. Computing something like $(a^m \mod n)$ can also be done in polynomial time: Write m in binary:

$$m = m_0 + m_1 \cdot 2 + \ldots + m_k \cdot 2^k$$

where $m_i \in \{0, 1\}$. One computes $(a^{2^i} \mod n)$, $i = 1, \ldots, k$ by repeated squarings:

$$(a^2 \mod n) = (a \mod n)^2 \ , \ (a^{2^2} \mod n) = (a^2 \mod n)^2 \ , \ \ldots \ ;$$

then $(a^m \mod n)$ is computed as the product of certain of these powers $(a^{2^i} \mod n)$ (those for which $m_i = 1$). Since $k = O(\log m)$ this is all done in polynomial time.

We can now review the RSA scheme with questions about complexity in mind: We see first that every computation that A and B do can actually be done in polynomial time; that's a good thing, — otherwise we would hardly consider the scheme practical ...

But a more serious question arises right at the beginning where A has to 'choose' 2 large prime numbers p and q. How does one do that? Keeping in mind that for practical applications the primes have to be larger than 10^{50} this is not a completely trivial question ...

Clearly we need a *primality test*: An algorithm that takes a natural number n as input and decides whether n is prime or not. We all know a simple algorithm that does the job, namely *trial division*: Perform division with remainder dividing n by $2, 3, 5, \ldots$.... If we find a factor $\leq \sqrt{n}$ then n is not prime, otherwise it is. This works great with small numbers, but for large numbers the obvious problem is that potentially we have to go all the way to $\sqrt{n}$ which is exponential in the size of n.

After the appearance of a brilliant new idea in 2002 by Agrawal, Kayal, Saxena, cf. [1], we now know that there exists a primality test that executes in polynomial time. Before that, one had various *probabilistic polynomial time* primality tests, notably the so-called Miller–Rabin test; I will go briefly into the notion of a probabilistic algorithm in the next section; suffice it here to say that a probabilistic primality test is an algorithm that depends on a certain random input in addition to the number n that we want to test. When the algorithm has executed it will either have found with certainty that n is *not* prime, or else it declares n prime; if the latter happens, the algorithm may however be *in error*; the point is then to repeat the process because one can show — in a sense that can be made precise — that the probability of error decreases exponentially with the number of times the algorithm is run (with different random inputs of course). Thus, a probabilistic primality test gives efficient production of what some people have called 'industrial grade primes', — numbers that are 'almost certainly primes'. Whatever this means, it is the method that is actually used in practice. (By the way, it is interesting to notice that the Miller–Rabin test can actually be turned into a genuine polynomial time primality test, — though only upon assumption of the so-called generalized Riemann hypothesis; but the Riemann hypothesis is 1 of the 7 'Millennium Problems' that each has 1 million dollars allocated to it as prize money for a solution; cf. [6]).

Concerning the security of the RSA system we already remarked that it obviously must be prevented that $n_A = pq$ is factored and thus the primes p and q are revealed. So, how hard is it to factor integers? Does there exist a polynomial time algorithm for factoring?

The truth is that nobody knows the answer to these questions. It seems difficult to imagine a world where factoring numbers is so to speak computationally not a significantly greater problem that multiplying integers. However, if we transcend the above 'classical' understanding of what the word 'algorithm' should stand for, and allow in algorithms such things as making measurements on physical systems, it seems that we do in fact live in such a world:

In 1997, a great surprise resulted when P. Shor showed in [8] that a *quantum computer* can in fact (probabilistically) factor integers in polynomial time; a quantum algorithm is not an algorithm in classical sense but involves an experiment with a certain quantum mechanical system; the notion can be formalized though, so one can introduce natural notions of 'polynomial time quantum algorithms', and so on.

But Shor's result does not have any obvious implications for the question about classical complexity of factoring. Let us turn to this question in the next section.

4. Factoring.

Suppose that we are given an integer $n > 1$ which is known to be (odd and) composite; for example, we may know that n is composite because we have run a fast primality test on n. Our task is now to find a non-trivial factor of n (if one wants the complete prime factorization of n, one can repeat the process on the numbers d and n/d where d is a non-trivial factor of n).

We all know a simple algorithm that will solve this problem: Use trial division with numbers (or just primes) $\leq \sqrt{n}$. Since $n > 1$ is composite this procedure will certainly reveal a non-trivial factor of n. The problem with this simple algorithm is of course that we potentially have to go all the way to $\sim \sqrt{n}$, and $\sqrt{n}$ grows exponentially with the size of n as $n \to \infty$ ...

All modern efficient factoring algorithms are probabilistic: They depend on a certain random input; with this input the algorithm runs for some time and then terminates with either no answer or with a non-trivial factor of n. If we get 'no answer' we repeat the process with a new random input, and continue until we actually get a non-trivial factor of n. We can speak of the *expected time* before the algorithm comes out with an answer, i.e., the expected number of elementary bit operations that a machine will perform before an answer is found. 'Expected' is of course to be taken in the sense of probability theory.

The expected time will depend on 2 things: The 'main loop' of the algorithm that executes once the random input is given will be deterministic and require a certain time to execute; we then have to think about the expected number of iterations (with different random inputs) of this main loop before an answer is found. The expected time of the algorithm will then be this expected number of iterations times the time requirement for the main loop.

Let us now consider the main principle behind most modern factorization algorithms: It is a simple idea due to P. Fermat (1601–1665): We a given an odd, composite number n. Suppose that we can find 2 integers b and c such that:

$$(\sharp) \qquad b^2 \equiv c^2 \pmod{n} \ .$$

The greatest common divisor $d := \gcd(n, b+c)$ of n and $b+c$ is of course a factor of n. We may hope that it is actually a non-trivial factor of n. When is it trivial, i.e., when is it 1 or n? That $d = 1$ means that $b + c$ is relatively prime to n; on the other hand, by ($\sharp$) we have that n divides $b^2 - c^2 = (b + c)(b - c)$; so $d = 1$ implies that n divides $b - c$, i.e., that $b \equiv c \pmod{n}$. Similarly one sees that $d = n$ implies $b \equiv -c \pmod{n}$. In other words, if $b \not\equiv \pm c \pmod{n}$ then d must be a non-trivial factor of n.

The point is now, as can easily be checked via the Chinese Remainder Theorem, that if n is odd and composite then for random pairs (b, c) with ($\sharp$) the probability that we have $b \equiv \pm c \pmod{n}$ is $\leq \frac{1}{2}$. Thus, if we have a number of such pairs at our disposal, and try to find a non-trivial factor of n by computing $\gcd(n, b+c)$, the probability that we are *not* successful decreases exponentially with the number of pairs that we try (it is in order to draw this conclusion that we made the harmless assumption that n be odd).

Most modern factorization algorithms rely on this simple principle and differ only in the methods for producing pairs (b, c) with ($\sharp$). Let me sketch one such algorithm, namely the so-called 'factor base algorithm'. This is one of the (very) few factorization algorithms for which a mathematically rigorous result about the expected execution time can be proved. The first to prove such a result was J. D. Dixon, see [4]; as a result the algorithm is also referred to as Dixon's algorithm.

Let us consider a simple example illustrating the principle of Dixon's algorithm: Suppose that we want to factor the number 4307. Suppose further that a random number generator gives us the numbers 93 and 107. Given these numbers we can then compute:

$$93^2 \equiv 35 = 5 \cdot 7 \pmod{4307} \quad \text{and} \quad 107^2 \equiv 2835 = 3^4 \cdot 5 \cdot 7 \pmod{4307} \ ,$$

so that

$$(93 \cdot 107)^2 \equiv (3^2 \cdot 5 \cdot 7)^2 \pmod{4307} \ .$$

Using the fast Euclidean algorithm we then compute

$$\gcd(4307, 93 \cdot 107 + 3^2 \cdot 5 \cdot 7) = \gcd(4307, 10266) = 59 \ ,$$

and then $4307 = 59 \cdot 73$.

The reader may now think that we were incredibly lucky to get the numbers 93 and 107 from the random number generator. Or, equivalently, wonder about how much time I had to spend in order to construct this simple example. But understanding precisely how much luck we actually need is in fact precisely one of the subtle points in the analysis of Dixon's algorithm: Given a general odd, composite number n how many numbers do we need from the random number generator before a 'lucky' coincidence such as the one above occurs?

Before we go into that, let us give a description of the general algorithm: We are given $n > 1$ which is odd and composite.

We first fix a real number y with $2 \leq y < n$. Later on, y will be chosen in dependence on n. Put:

$$N := \pi(y) \ ,$$

where π is the prime number function, i.e., $N = \pi(y)$ is the number of primes $\leq y$. By the prime number theorem we have

$$\pi(y) \approx \frac{y}{\log y}$$

where the implied error term can be made explicit.

Let the primes $\leq y$, i.e., the first N primes, be $p_1, \ldots, p_N$. The set $\{p_1, \ldots, p_N\}$ will serve as our 'factor basis'.

From now on, when we talk about residue classes mod n we will actually mean the unique representative of the class in the interval $[1, n]$.

Step 1. We choose a number b randomly in the interval $1 < b < n$. We compute $\gcd(b, n)$; if this is > 1 we are quite happy since this number is then a non-trivial factor of n. So we now assume that b is relatively prime to n.

Denote by $Q(b)$ the number

$$Q(b) := b^2 \mod n \ ;$$

thus, by the above convention, $Q(b)$ is the unique representative of $(b^2 \mod n)$ in the interval $[1, n]$.

We determine whether $Q(b)$ has all its prime factors $\leq y$. If not, we choose a new b and test again $Q(b)$.

We continue until we have $N + 1$ numbers b_i, $i = 1, \ldots N + 1$, such that every $Q(b_i)$ has all its prime factors $\leq y$. For each i we compute the factorization:

$$Q(b_i) = \prod_{j=1}^{N} p_j^{\beta_{ij}} \ ,$$

and put:

$$\epsilon_{ij} := (\beta_{ij} \mod 2) \ .$$

Step 2. For each $i = 1, \dots, N+1$ we can consider the vector:

$$\epsilon_i := (\epsilon_{i1}, \dots, \epsilon_{iN})$$

as a vector in the N-dimensional vector space $(\mathbb{Z}/\mathbb{Z}2)^N$ over the finite field with 2 elements $\mathbb{Z}/\mathbb{Z}2$.

Since we have $N+1$ vectors in an N-dimensional vector space, the ϵ_i must be linearly dependent. Since we are working over $\mathbb{Z}/\mathbb{Z}2$ with the only elements 0 and 1, the linear dependence of the ϵ_i means that there is a non-empty subset

$$I \subseteq \{1, \dots, N+1\}$$

such that:

$$\sum_{i \in I} \epsilon_i = 0 \ .$$

Since $\epsilon_{ij} := (\beta_{ij} \mod 2)$ we see that this means that:

$$\sum_{i \in I} \beta_{ij} \equiv 0 \quad (2) \quad \text{for} \quad j = 1, \dots, N \ ,$$

so that we can write:

$$\sum_{i \in I} \beta_{ij} = 2 \cdot \gamma_j \quad \text{for} \quad j = 1, \dots, N \ ,$$

with integers γ_j.

Step 3. Now we put:

$$b := (\prod_{i \in I} b_i \mod n) \quad \text{and} \quad c := (\prod_{j=1}^{N} p_j^{\gamma_j} \mod n) \ ,$$

and obtain:

$$b^2 \equiv \prod_{i \in I} b_i^2 \equiv \prod_{i \in I} Q(b_i) = \prod_{i \in I} \prod_{j=1}^{N} p_j^{\beta_{ij}} = \prod_{j=1}^{N} p_j^{\sum_{i \in I} \beta_{ij}} = \prod_{j=1}^{N} p_j^{2\gamma_j} \equiv c^2 \mod n \ .$$

If then $b \equiv \pm c \ (n)$ we go back to step 1. Otherwise, we compute $\gcd(b+c, n)$ which is then a non-trivial factor of n.

For the analysis of the algorithm, let us first remark that it may seem paradoxical that we are describing a factoring algorithm but then in step 1 talk about computing the factorization of the $Q(b_i)$. However, we are only talking about whether a given $Q(b)$ can be completely factored as product of the primes $p_1, \dots, p_N$, and if so we must actually find the factorization. This can simply be done via trial division with these primes; the point is that if y and hence also $N = \pi(y) \approx \frac{y}{\log y}$ is only moderately large compared to n then the work involved in this is not too large.

But this also brings us to the essential points in the analysis of the algorithm: A detailed analysis will reveal that there are 2 dominating contributions to the expected running time:

First, in step 2 we must find a non-trivial linear relation between the vectors ϵ_i. We do this by viewing the ϵ_i as rows in an $(N+1) \times N$ matrix with entries in $\mathbb{Z}/\mathbb{Z}2$ and then use Gauss elimination on this matrix. Sooner or later during the elimination a row consisting entirely of zeros will appear; if we have kept track of our row operations we will then at that point have found a non-trivial linear relation between the ϵ_i. The time requirement for the Gauss elimination is:

$$O(N^3) = O(\pi(y)^3) \ .$$

Secondly, looking at step 1 it is intuitively clear that question about the optimal choice of y can not be entirely trivial: If we choose y large then N is large and so we have a lot of trial divisions with the primes $p_1, \ldots, p_N$. On the other hand, if we choose y smaller then clearly we reduce the probability that a $Q(b)$ can be completely factored with $p_1, \ldots, p_N$, i.e., with the primes $\leq y$. In other words, if y is small we have to test a lot of random b's to get just one 'good' $Q(b)$.

It is clear that a rigorous analysis thus requires us to quantify the above probability. And this is actually the most complicated point in the analysis.

We are talking about the probability that a random number in $[1, n]$ has all its prime divisors among $p_1, \ldots, p_N$, i.e., among the primes $\leq y$. This probability is:

$$\frac{\psi(n,y)}{n}$$

where ψ is a function well-known from analytic number theory, namely the function $\psi(n, y)$ defined as the number of integers $a \in [1, n]$ that have all prime factors $\leq y$.

The expected number of times we have to test random b's in order to find just 1 'good' $Q(b)$ is then:

$$\left(\frac{\psi(n,y)}{n}\right)^{-1} \ .$$

Since we need $N + 1$ good $Q(b)$'s we expect to be required to test

$$(N+1) \cdot \left(\frac{\psi(n,y)}{n}\right)^{-1}$$

random b's. For each b we have some trial divisions with the $p_1, \ldots, p_N$ to do; an upper bound for the time requirement of that can be found to be:

$$O(N \cdot \log^3 n)$$

so that (an upper bound on) the total, expected time requirement for this part of the algorithm is:

$$O\left(N(N+1) \cdot \log^3 n \cdot \left(\frac{\psi(n,y)}{n}\right)^{-1}\right) = O\left(N^2 \cdot \log^3 n \cdot \left(\frac{\psi(n,y)}{n}\right)^{-1}\right)$$

which turns out to be the second dominating contribution to the expected execution time of the algorithm.

It is now clear that in order to proceed any further, and in particular answer the question about the optimal choice of the parameter y, we need some *lower bound* on the ψ-function. It has been known for many years that we have (in a sense that can be made precise):

$$\psi(n,y) \approx u^{-u} \quad \text{where } u := \frac{\log n}{\log y}$$

but this is far from being enough information for our purposes.

It is possible to use various types of elementary arguments to obtain fairly good lower bounds for $\psi(n, y)$. These 'elementary lower bounds' come remarkably close to giving the right, final answer which however is only obtained by using a tough theorem in analytic number theory, namely a theorem by Canfield, Erdös and

Pomerance from 1983, cf. [3]. We will not quote the theorem but merely state that its use leads to the following conclusion: With the following choice of y:

$$y = e^{\frac{1}{2}\sqrt{\log n \log\log n}}$$

in Dixon's algorithm the expected time requirement to find a non-trivial factor of n is bounded by:

$$O\left(e^{(2+\epsilon)\sqrt{\log n \log\log n}}\right)$$

for any $\epsilon > 0$.

This bound is certainly not a polynomial dependence on n but it is also clearly asymptotically far better than a pure exponential dependence. In mathematical jargon, we speak of a 'subexponential (probabilistic) algorithm'.

The interested reader who wishes to see more of the details in the above complexity analysis can be referred to some lecture notes that I have written, cf. [5].

As we remarked earlier, most modern factorization algorithms rely on Fermat's old idea of using pairs of integers (b, c) with $b^2 \equiv c^2 \pmod{n}$, and differ only in the method used to produce such pairs. Thus, Dixon's algorithm uses a relatively straightforward, 'brute force' method. A much more sophisticated way of producing these pairs relies on algebraic number theory and is used in the so-called *number field sieve*, — also a probabilistic algorithm. With certain heuristic assumptions, i.e., certain unproven but 'reasonable' hypotheses the number field sieve has an expected execution time bounded by:

$$O\left(e^{((\frac{64}{9})^{\frac{1}{3}}+\epsilon)(\log n)^{\frac{1}{3}}(\log\log n)^{\frac{2}{3}}}\right) ,$$

for any $\epsilon > 0$, cf. [2]. This (upper bound on the) expected execution time is — if true — asymptotically the best among all known factorization methods.

The conclusion is that — quite surprisingly — there does in fact exist (classical, probabilistic) factorization algorithms with expected execution times asymptotically much better than a raw exponential dependence on n. But that also, on the other hand, this is still not enough in practice to be a real threat to the RSA cryptosystem. For that, we will probably have to wait for the physical realization of quantum computers!

References

[1] M. Agrawal, N. Kayal, N. Saxena: 'Primes is in P', Preprint 2002.
http://www.cse.iitk.ac.in/news/primality_v3.pdf

[2] J. P. Buhler, H. W. Lenstra, C. Pomerance: 'Factoring integers with the number field sieve', Lecture Notes in Mathematics **1554** (1993), 50–94.

[3] E. R. Canfield, P. Erdös, C. Pomerance: 'On a problem of Oppenheim concerning "Factorisatio Numerorum"', J. Number Theory **17** (1983), 1–28.

[4] J. D. Dixon: 'Asymptotically fast factorization of integers', Math. Comp. **36** (1981), 255–260.

[5] I. Kiming: 'The ψ-function and the complexity of Dixon's factoring algorithm', lecture notes 2004,
http://www.math.ku.dk/~kiming/courses/2004/krypto/psi.pdf

[6] The Millennium Prize Problems from the Clay Mathematics Institute:
http://www.claymath.org/millennium/

[7] R. L. Rivest, A. Shamir, L. Adleman: 'A method for obtaining digital signatures and public-key cryptosystems', Comm. ACM **21** (1978), 120–126.

[8] P. W. Shor: 'Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer', SIAM J. Comput. **26** (1997), 1484–1509.

Department of Mathematics, University of Copenhagen, Universitetsparken 5, DK-2100 Copenhagen Ø, Denmark.

E-mail address: kiming@math.ku.dk