The RSA cryptosystem

Zehra Kolat

MATH 411 Capstone Project 1

Advisor: Doğa Can Sertbaş

Istinye University
Faculty of Engineering and Natural Sciences
Mathematics Department
January 2025

The RSA cryptosystem

Zehra Kolat

01/03/2025

Abstract

A cryptosystem is a set of mathematical and algorithmic methods used to encrypt and securely transmit data. Cryptosystems form the basis of modern digital security infrastructures and are vital for data encryption and transmission. A cryptosystem consists of three basic components: encryption, decryption and key-generation. The aim is to securely transmit information from one point to another, prevent unauthorized people from accessing this information, and ensure that the recipient confirms the accuracy of the information. In this study, one of the asymmetric encryption algorithms, the RSA crypto system developed by Rivest–Shamir–Adleman in 1977, is discussed. Asymmetric cryptography is a system in which two different keys are used: public key (used for encryption and can be shared with everyone) , secret key (used for decryption and only the owner has it). The mathematical basis of RSA is based on the difficulty of factoring large prime numbers, and it is very difficult to factor a number formed by multiplying large prime numbers back into prime factors, which makes RSA secure. In this study, the key generation, encryption and decryption processes of the algorithm are explained step by step and the advantages it provides are evaluated. In addition, the Birthday Paradox, Floyd's trick and Pollard rho, Dixon's random squares are discussed. This study aims to provide a comprehensive resource for understanding the basic principles of RSA.

Acknowledgement

I would like to thank my advisor, Dr. Doğa Can Sertbaş, for the guidance and support he provided throughout this study. His expertise and helpful feedbacks were crucial for my academic progress on this research. I'm deeply grateful for his provided opportunities, guidances and countless helps for me academically and professionally.

Contents

1 Introduction

The RSA is a public key algorithm invented by Rivest, Shamir and Adleman. The key used for encryption is different from (but related to) the key used for decryption. The pair of numbers pk=(N,e) is known as the public key and can be published. The pair of numbers sk=(N,d) is known as the private key and must be kept secret. An algorithm that uses different keys for encryption and decryption is said to be asymmetric. Anybody knowing the public key can use it to create encrypted messages, but only the owner of the secret key can decrypt them.

2 The Cryptosystem RSA

We follow the convention of calling the two players Alice and Bob. Our scenario is that Alice wants to send a message to Bob that he should be able to read, but no one else should be able to read. Suppose that Alice wants to send a (long) string of bits. There is a security parameter n. Alice splits her string into blocks of $n-1$ bits each, and transmits each block separately. So we now explain how to transmit a single block $(x_0, \ldots, x_{n-2})$ of $n-1$ bits in the RSA system. We interpret the block as the binary representation of the natural number

$$x = \sum_{i=0}^{n-2} x_i 2^i$$

This number will be transmitted.

Bob randomly chooses two prime numbers p and q, each of $n/2$ bits, such that their product is $N = p \cdot q$, has n bits. He also chooses some random integer e with $1 \leq e < N$ and $gcd(e, (p-1)(q-1)) = 1$. Bob's public key is $\texttt{pk} = (N, e)$. Alice looks it up and sends the encryption $y = x^e \mod N$ to Bob, that is, the remainder of x^e on division by N. The magic now is that Bob can recover x from Alice's message with the help of his private information derived from (p, q).

2.1 Cryptosystem RSA

Key Generation (`keygen`):
Input: Security parameter n.
Output: Secret key `sk` and public key `pk`.

1. Choose two distinct primes p and q at random with $2^{(n-1)/2} < p, q < 2^{n/2}$.
2. $N \leftarrow p \cdot q$, $L \leftarrow (p-1)(q-1)$. [Here, N is an n-bit number, and $L = \phi(N)$ is Euler's ϕ function.]
3. Choose $e \in \{2, \ldots, L-2\}$ at random, such that $\gcd(e, L) = 1$.
4. Compute d, the modular inverse of e in $\mathbb{Z}_L$.
5. Publish the public key $\texttt{pk} = (N, e)$ and keep $\texttt{sk} = (N, d)$ as the secret key.

Encryption (`enc`):
Input: $x \in \mathbb{Z}_{\mathbb{N}}$, $\texttt{pk} = (N, e)$.
Output: $enc_{pk}(x) \in \mathbb{Z}_{\mathbb{N}}$.

1. Compute $y \leftarrow x^e \mod N$.
2. Return $\texttt{enc}_{\texttt{pk}}(x) = y$.

Decryption (`dec`):
Input: $y \in \mathbb{Z}_N$, $\texttt{sk} = (N, d)$.
Output: $\texttt{dec}_{\texttt{sk}}(y) \in \mathbb{Z}_N$.

1. Compute $z \leftarrow y^d \mod N$.
2. Return $\texttt{dec}_{\texttt{sk}}(y) = z$.

In key-gen, the first step is fast enough. We will explain in the next sections. The second step generates N and finds $L = \phi(N)$. We define the Euler totient function $\varphi(N)$ as the number $\phi(N) = |\mathbb{Z}_N^\times|$ of units in $\mathbb{Z}_N$. In other words, the Euler phi-function is defined to be the number of nonnegative integers b less than N which are prime to N: $\phi(N) = \#\{0 \le b < N | gcd(b, N) = 1\}$

Lemma 1. *Suppose that $N = p_1^{e_1} \cdot p_2^{e_2} \cdots p_r^{e_r}$, with pairwise different primes $p_1, \ldots, p_r$, and all $e_i \geq 1$.Then*

$$\phi(N) = p_1^{e_1-1}(p_1 - 1) \ldots p_r^{e_r-1}(p_r - 1) = N(1 - \frac{1}{p_1}) \ldots (1 - \frac{1}{p_r})$$

Corollary 2. *Let p and q be different primes. Then $\phi(p{\cdot}q) = \phi(p){\cdot}\phi(q) = (p-1)\cdot(q-1)$ Therefore the function ϕ is a multiplicative function. [2]*

Proof. If $p = 1 \longrightarrow \phi(1 \cdot q) = \phi(q)$ and similarly if $q = 1 \longrightarrow \phi(p \cdot 1) = \phi(p)$.Thus, we may assume $p > 1$ and $q > 1$.

Now let's arrange the numbers between 1 and pq so that there are q elements in column p. We will have a matrix like this:

$$\begin{array}{cccccc}
1 & 2 & \cdots & r & \cdots & p \\
p+1 & p+2 & \cdots & p+r & \cdots & 2p \\
2p+1 & 2p+2 & \cdots & 2p+r & \cdots & 3p \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
(q-1)p+1 & (q-1)p+2 & \cdots & (q-1)p+r & \cdots & qp
\end{array}$$

Each element of this matrix is a number related to pq, which is the product of p and q. Our goal is to find the numbers in this matrix that are prime to p and q. An important point here is that $\phi(pq)$ corresponds to counting the numbers that are prime to pq. Similarly, we can say that these numbers must also be prime to p and q. We will now examine each column of the matrix.

Each column is arranged in relation to the divisors of p. For example, when a column is numbered r, the numbers in that column are ordered as follows: Each column is arranged in relation to the divisors of p. For example, when a column is numbered r, the numbers in that column are ordered as follows: $r, p+r, 2p+r, 3p+r, \cdots, (q-1)p+r$.

The thing to note here is that when $gcd(r,p) = 1$, all numbers in that column will be prime to p. Because when $gcd = (np+r, p) = gcd(r,p) = 1$, no number in that column can be divided by p. Therefore, only the column $\phi(p)$ will be filled with numbers that are prime to p.

Now, let's consider whether the numbers in the same column are also prime with respect to q. The numbers in this column are sorted as follows: $r, p+r, 2p+r, 3p+r, \cdots, (q-1)p+r$. We know that all numbers in this column are prime to p, $gcd(p,r) = 1$. Therefore the numbers in this row are unique with respect to mod q since the $gcd(p,q) = 1$. This means that the number in each row is from the set $\{0, 1, 2, \cdots, n-1\}$. In each column, there is a number $\phi(q)$ that is prime to pq. We can now say that in each column there are $\phi(q)$ numbers that are prime with respect to pq and these columns have a total of $\phi(p)$ numbers. Thus there are $\phi(p) \cdot \phi(q)$ numbers which are prime to pq. This completes the proof of the corollary 2.

□

In the third step find e then in the fourth step find the inverse d of e for key-gen. In this step use the extended euclid algorithm to find $gcd(e, \phi(N))$. This algorithm computes not only the gcd but also a representation of it as a linear combination of the inputs and called 'Bezout's identity'.

Extended Euclidean Algorithm:
Input: $a, b \in \mathbb{Z}$, where $a \geq b > 0$.
Output: $\Delta \in \mathbb{N}$, $r_i, s_i, t_i \in \mathbb{Z}$ for $0 \leq i \leq \Delta+1$, and $q_i \in \mathbb{Z}$ for $1 \leq i \leq \Delta$, as computed below.

1. $r_0 \leftarrow a$, $s_0 \leftarrow 1$, $t_0 \leftarrow 0$, $r_1 \leftarrow b$, $s_1 \leftarrow 0$, $t_1 \leftarrow 1$.

```
2. $i \leftarrow 1$.
3. While $r_i \neq 0$ do step 4.
4. $q_i \leftarrow \text{quo}(r_{i-1}, r_i)$, $r_{i+1} \leftarrow r_{i-1} - q_i r_i$, $s_{i+1} \leftarrow s_{i-1} - q_i s_i$, $t_{i+1} \leftarrow t_{i-1} - q_i t_i$, $i \leftarrow i + 1$.
5. $\Delta \leftarrow i - 1$.
6. **If** $r_\Delta$ is a unit ($r_\Delta = \pm 1$), **replace** $(r_\Delta, s_\Delta, t_\Delta)$ by $\left(1, \frac{s_\Delta}{r_\Delta}, \frac{t_\Delta}{r_\Delta}\right)$.
7. **Return** $\Delta, r_i, s_i, t_i$ for $0 \leq i \leq \Delta + 1$, and $q_i$ for $1 \leq i \leq \Delta$.
```

The algorithm terminates because the $d(r_i)$ are strictly decreasing non-negative integers for $1 \leq i \leq \Delta$, where $d(a) = |a|$ for $a \in \mathbb{Z}$. In $\mathbb{Z}$, the quotient and remainder are uniquely defined: $0 \leq r < b$. The elements r_i for $0 \leq i \leq \Delta + 1$ are the remainders and the q_i for $1 \leq i \leq \Delta$ are the quotients in the Euclidean Algorithm.

Modular squaring and multiplication operations are completed in n^2 steps. In the extended euclidean algorithm, our operations are completed in Δ steps. If the size of the number is 2^n, it is completed in at most $n - 2$ steps, that is, Δ is evaluated approximately as n. Thus, the total time complexity is $O(n^3)$ of this algorithm. Thus this algorithm can be run in poly-time.

For an algorithm to be usable, it must run at an acceptable speed, that is, it must be efficient. Time complexity is what shows the efficiency of the algorithm. Time complexity gives us an idea about whether the algorithm is efficient or not.Big O notation is a concept that allows the running times of algorithms to be expressed mathematically, that is, time complexity.

Big-O Notation: Big-O notation is used to describe the upper bound of an algorithm's growth rate as its input size increases. It provides an asymptotic upper bound, showing how a function grows in the worst case.

Mathematically, Big-O is defined as:

$$O(f(n)) = \{g(n) \mid \exists c > 0, n_0 \in \mathbb{N} \text{ such that } 0 \leq g(n) \leq c \cdot f(n), \text{ for all } n \geq n_0\}$$

Here: - $f(n)$ is the growth function of the algorithm. - $g(n)$ is the bounding function. - c and n_0 are constants for the asymptotic upper bound. According to this definition, if we have a polynomial-time algorithm, we can say that this algorithm is efficient.

Big-O notation is commonly used to analyze the efficiency of algorithms [1].

Euler's Theorem: For any $x \in \mathbb{Z}^*_{\mathbb{N}}$, we have $x^{\phi(N)} = 1 \in \mathbb{Z}^*_{\mathbb{N}}$.

Euler's Theorem is a generalization of Fermat's Little Theorem. The fourth step uses Euler's Theorem for find d. We know that the $e \cdot d = 1 \mod \Phi(N)$ by the Euler's theorem.

After choosing e, compute the ciphertext y. But computing y directly requires a lot of bit operation. Therefore we use the repeated squaring algorithm to compute d. Repeated squaring algorithm uses less than $2n$ bits of operations for n bits number.For the compute decrypted message z we also use repeated squaring algorithm.

Repeated Squaring Algorithm:
Input: A group G, a base $x \in G$, and an exponent $e \in \mathbb{Z}$ with $1 \leq e < \#G$.
Output: $x^e \in G$

```
1. Let $\sum_{i=0}^{n} e_i 2^i$be the binary representation of $e$ with $e_0, \ldots, e_{n-1} \in 0, 1$ , $e_{n-1} = 1$, and $n$ its a bit length.
2. $y \leftarrow x$
3. For $i$ from $n-2$ downto 0 do steps $4-5$
4. $y \leftarrow y^2$
5. If $e_i = 1$, then $y \leftarrow y \cdot x$
6. Return $y$.
```

Modular squaring and multiplication operations are completed in n^2 steps, and this is repeated n times.Thus the total time complexity is $O(n^3)$ of repeated squaring algorithm where the bit length of N is n. The correctness of the repeated squaring algorithm is based on realization exponentiation operation using binary representation. The algorithm writes the given exponent e in binary representation and performs squaring and multiplication operations for each bit when necessary, $x^e = x^{e_0 \cdot 2^0 + e_1 \cdot 2^1 + \cdots + e_{n-1} \cdot 2^{n-1}}$. This approach ensures that each digit of the exponent is calculated correctly. The correct multiplication and squaring operations are performed for each bit when calculating e, which ensures that the algorithm produces correct results.

What features should a cryptosystem provide to be useful in daily life? We have to address several questions?
Correctness: Is $z = x$?
Efficiency: How to calculate fast, large primes at random, d from e, powers modulo N? and
Security: In fact, x is uniquely determined! But how long does it take to calculate this value? Is this difficult enough to provide security?

In key generation, since p and q are chosen randomly, a probabilistic algorithm is required. This is the standard type of algorithm in this text.Its expected runtime for a fixed input is obtained by averaging over the algorithm's internal random choices. Thus a bound on the runtime has to hold for all inputs; there is no averaging over the inputs. For RSA encryption and decryption, no random choices are required. This special type of probabilistic algorithms is called deterministic. The deterministic algorithm produces the same outputs with the same inputs every time it runs. That means, the runtime does not involve random choices, which means the runtime is predictable. The probabilistic algorithm is effective if the input length depends on n and the expected runtime is polynomially bounded in n. This is called polynomial time algorithm.

3 The RSA Cryptosystem

We recall the three basic questions posed in Section 2.1: correctness, efficiency, and security of the RSA cryptosystem.

3.1 Analysis of RSA

Theorem 3. *RSA works correctly, that is, for every message x the decrypted encrypted message z equals x.*

Proof. First proof: We know that $y = x^e$ & $z = y^d$ in $\mathbb{Z}_\mathbb{N}$ by the definiton. First, we consider the case where x is a unit modulo N, in other words, that $gcd(x, N) = 1$. Also we have $x^{\phi(N)} = 1$ by the euler's theorem. Hence $ed = 1 + k \cdot \phi(N)$, where $k \in \mathbb{Z}$. Then $z = y^d = x^{ed} = x^{1+k\cdot\phi(N)} = x \cdot x^{\phi(N)^k} = x \cdot 1^k = x\cdot$ in $\mathbb{Z}_\mathbb{N}^*$

Next, we consider the case where $gcd(x, N) \neq 1$. This gcd is then p, q, or N. We first take $gcd(x, N) = p$, in here $\phi(N) = p - 1$ and $ed = 1 +$

for $= k \cdot \phi(N) \in \mathbb{Z}$ and thus $x = u \cdot p$ for $gcd(u, p) = 1$. Then we have $p^{ed} = p^{1+\cdot\phi(N)} = p \cdot p^{\phi(N)} = p \cdot 1 = p$ in $\mathbb{Z}_q$. Since $p^{ed} \equiv 0$ Mod p, we have $p^{ed} \equiv p$ Mod N (by the Chinese Remainder Theorem) and $u^{ed} \equiv u$ Mod N. Altogether, $x^{ed} = (up)^{ed} = u^{ed}p^{ed} = up = x$ in $\mathbb{Z}_{\mathbb{N}}$

The case $gcd(x, N) = q$ is analogous. The remaining case $gcd(x, N) = N$ is trivial, since then $x = 0$.

The cases where $gcd(x, N) \neq 1$ are somewhat unrealistic, since then the secret key can easily be found and henceforth any encrypted message sent can be read. But fortunately, this happens for a random x only with the negligible probability.

$$1 - \frac{\phi(N)}{N} = 1 - \frac{(p-1)(q-1)}{N} = \frac{p+q-1}{N} \approx \frac{2 \cdot 2^{n/2}}{2^n} = 2^{-n/2+1}.$$

□

3.2 Primality Testing

Basically, random numbers are selected and checked if it is prime or not. However, checking every number is inefficient, so some primality tests are applied.

We face the following task from the efficiency question in Section 2.1: given a large B, say $B \geq 2^{1500}$, find a prime p with $B \leq p \leq \sqrt{2}B$. This range provides n bit length of N. As the numbers get larger, the existence of prime numbers become rare and take a long time to find. Prime number density is suitable in the range we define by the prime number theorem.

To find (random) primes in a given range, we randomly pick N integers and test them for primality. By definition, N is prime if and only if it has no divisor between 2 and $N - 1$. If it has any such divisor, then it has one up to $\sqrt{N}$, and it is sufficient to try all prime numbers up to $\sqrt{N}$ as divisors. This leads to about $\sqrt{N}$ operations, which is utterly infeasible in the cryptographic range. A brilliant idea is to replace the definitional property by a property that is true for prime numbers but (hopefully) sufficiently false for composite numbers. This property is Fermat's Little Theorem.

Fermat's Little Theorem:
If N is a prime number, then for any nonzero $x \in \mathbb{Z}_{\mathbb{N}}$ we have $x^{N-1} = 1$ in $\mathbb{Z}^*_{\mathbb{N}}$.

We should abandon deterministic precision and use a probabilistic algorithm instead.

```
Algorithm: Fermat Test
Input: A number $N \in \mathbb{Z}$ with $N \geq 2$
Output: Either "$N$ is composite", or "$N$ is possibly prime".
1. $x \longleftarrow 1, 2, \ldots N-1$
2. $g \leftarrow gcd(x, N)$. If $g \neq 1$, then Return "$N$ is composite".
3. $y \leftarrow x^{N-1}$ in $\mathbb{Z}_{\mathbb{N}}$
4. If $y \neq 1$, then Return "$N$ is composite".
5. Return "$N$ is possibly prime".
```

Fermat Test checks prime number properties using Fermat's Little Theorem. In addition, it keeps processing times short with efficient algorithms such as euclidean algorithm and repeated squring algorithm, allowing fast primality testing for large N's. If we are going to talk about time complexity for this test, we need to add the cost of gcd computation and the cost of exponentiation operations. Modular squaring and multiplication operations are completed in n^2 steps, and this is repeated n times. Thus the total time complexity is $n^3 + n$, so $O(n^3)$ of Fermat test where the bit length of N is n.

If N is prime, the Fermat test will correctly answer that N is possibly prime, by Fermat's Little Theorem. We are looking for prime numbers, so there are no "false negatives." The test erros if N is composite but it returns "possibly prime"; this is a "false positive".

Now assume that N is composite. An element x of $\mathbb{Z}_{\mathbb{N}}$ is called:

- Fermat Witness if $x^{N-1} \neq 1$ in $\mathbb{Z}_{\mathbb{N}}$
- Fermat Liar if $x^{N-1} = 1$ in $\mathbb{Z}_{\mathbb{N}}$

The set L_N of Fermat liars is a subgroup of $\mathbb{Z}_{\mathbb{N}}^*$ and $L_N = \mathbb{Z}_{\mathbb{N}}^*$ (there is no fermat witness) or $|L_N| \leq |\mathbb{Z}_{\mathbb{N}}|/2$ by the Lagrange's Theorem. If there is at least one Fermat witness, then there is more, and at most half of all possible x's are liars. Obviously, $|\mathbb{Z}_{\mathbb{N}}^*| = \phi(N)$.

Composite numbers that do not have any Fermat witness are called Carmichael numbers. The Fermat test fails for them, and gives the wrong answer for all x except the negligibly few for which $g \neq 1$ in step 2. The first Carmichael numbers are $561 = 3 \cdot 11 \cdot 17$, $1105 = 5 \cdot 13 \cdot 17$, and $1729 = 7 \cdot 13 \cdot 19$.

Carmichael numbers are a weakness of Fermat's test. We now have a new test to address this shortcoming. The new test not only distinguishes primes from Carmichael numbers, but also factorizes these seemingly difficult numbers in random polynomial time.

Algorithm: Miller Rabin Test
Input: An odd integer $N \in \mathbb{Z}$ with $N \geq 3$
Output: Either "composite" or "probably prime".

1. Write $N-1 = 2^e m$, where e and m are (uniquely determined) positive integers with m odd
2. $x \longleftarrow \{1, ..., N-1\}$.
3. If $gcd(x, N) \neq 1$then return "composite".
4. $y \leftarrow x^m \in \mathbb{Z}_{\mathbb{N}}$
5. If $y = 1$, then return "probably prime".
6. For i from 0 to $e-1$ do steps 7-8
7. If $y = -1$ then return "probably prime"
8. else $y \leftarrow y^2$ in $\mathbb{Z}_{\mathbb{N}}$.
9. Return "composite".

What is done in the seventh and eighth steps essentially means obtaining x^{N-1}.
$x^{N-1} \neq 1 \pmod N \Rightarrow N$ is composite.
$x^{N-1} \equiv 1 \pmod N \Rightarrow N$ is probably prime and means that:
$x^{N-1} - 1 = 2^e m - 1$ and $N | x^{N-1} - 1 \iff N | (x^m - 1)(x^m + 1)(x^{2m} + 1)(x^{4m} + 1) \ldots (x^{2^e m} + 1) \Rightarrow p | x^{N-1} - 1$ for $N = pq$.

The correctness of this test is based on the following facts.

- if N is prime, then $\mathbb{Z}_{\mathbb{N}}$ is a field
- The only elements x of a field with $x^2 = 1$ are $x \in \pm 1$, that is, $\sqrt{1} = \pm 1$. $S = \sqrt{1} = \{x \in \mathbb{Z}_{\mathbb{N}} : x^2 = 1\}$

Moreover, if we have integers $N \geq 2$ and y with $y^2 = 1$ and $y \backslash \notin \pm 1$ in $\mathbb{Z}_{\mathbb{N}}$, then $gcd(y-1, N)$ is a proper divisor of N. As an example, $67^2 = 1$ and $67 \notin \pm 1$ in $\mathbb{Z}_{561}$, and $gcd(67-1, 561) = 33$ is a nontrivial factor of 561. This will be a recurrent theme in this and later sections: finding a nontrivial square root of 1, that is, an element of $\sqrt{1} \backslash \pm 1$. Any $s, t \in \mathbb{Z}_{\mathbb{N}}^*$ with $s^2 = t^2$ and $s \notin t$ yield $s/t \in \sqrt{1} \backslash \pm 1$.

Lemma 4. *Let N be an odd n-bit integer.*
(i) If $y \in \sqrt{1} \backslash \pm 1 \subseteq \mathbb{Z}_{\mathbb{N}}^*$, then $gcd(y-1, N)$ is a proper factor of N.
(ii) One can test in time polynomial in n whether N is a perfect power ae of an integer a, with $e \geq 2$, and if so, produce such a and e.

Proof. Let $N = q_1 \dots q_r$ be a factorization of N into powers of distinct primes.

(i) since $y^2 = 1$ in $\mathbb{Z}_{\mathbb{N}}$, we also have $y^2 = 1$ in each $\mathbb{Z}_{q_i}$. That is, for each prime factor q_i, the element y must be either 1 or -1 in this prime modular group. But since $y \neq \pm 1$, we will have $y = 1$ in some modular groups and $y = -1$ in others. We distinguish between these two cases and partition y into these prime modular groups: we have $r \geq 2$, some $I \subseteq \{1, \dots, r\}$ with $I \neq \varnothing$, $\{1, \dots, r\}$ and $y = 1$ in $\mathbb{Z}_{q_i}$ for $i \in I$ and $y = -1$ in $\mathbb{Z}_{q_i}$ for $i \in \{1, \dots r\} \backslash I$. Then $gcd(y-1, N) = \prod_{i \in I} q_i$ is proper divisor of N.
(ii) We calculate the real number $r_i = N^{1/i}$ to some, say 5, digits of accuracy after the binary point, for $2 \leq i \leq log_2 N$. For those r_i which are close to an integer, we take the rounded value $s_i = \lceil r_i \rfloor$, test whether $s_i^i = N$, and return the (s_i, i) with the largest value of i that passed the test. The answer is correct, each test can be performed with $O(n^3)$,) operations, and there are $O(n)$ tests. □

If N is power prime, it can be check at the poly-time. Then you can insert it into any primality test.

Lemma 5. *Let N be odd, not prime or a power of a prime, and $x \leftarrow \mathbb{Z}_{\mathbb{N}}^*$. Then with probability at least 1/2, the order k of x is even and $x^{k/2} \notin \pm 1$*

Proof. A proper subgroup $G = \{x \in \mathbb{Z}_{\mathbb{N}}^* | x^c \in \pm 1\}$. G is a proper subgroup means $G \neq \mathbb{Z}_{\mathbb{N}}^* \Rightarrow G < \mathbb{Z}_{\mathbb{N}}^*$. Therefore by the Lagrange's theorem $|G| \leq |\mathbb{Z}_{\mathbb{N}}^*|/2$. If G has at most $|\mathbb{Z}_{\mathbb{N}}^*|/2 = \phi(N)/2$ element, then it has at most $\phi(N)/2$ element with trivial roots. Thus, if the exponent is ± 1 for a randomly chosen x, this x in G. If the most that satisfies this condition is $|\mathbb{Z}_{\mathbb{N}}^*|/2$, the least that does not is $|\mathbb{Z}_{\mathbb{N}}^*|/2$ □

Theorem 6. *The Miller-Rabin test has the following properties.*
(i) If N is prime, the test returns "probably prime".
(ii) If N is composite, the test returns "composite" with probability at least 1/2.
(iii) For an n-bit input N, the test uses $O(n^3)$ bit operations

Proof. When $gcd(x, N) \neq 1$, step 3 returns correctly. For an RSA modulus N, the probability for this to happen is exponentially small (remember the case $gcd(x, N) \neq 1$, $2^{-n/2+1}$). We now show that the error probability is at most 1/2 for the $x \in \mathbb{Z}_{\mathbb{N}}^*$.

(i) When N is prime and $x \in \mathbb{Z}_{\mathbb{N}}^*$, $gcd(x, N) = 1$ equality is achieved. Then $x^{N-1} = 1$ by Fermat's Little Theorem. Either in step 5, since $x^m = 1$, $y = 1$ and the correct answer is returned, or $y^{2^{i+1}} = 1$ for some i with $1 \leq i < e$, but if $y^{2^{i+1}} \neq 1$ is found, the test returns the correct result "probably prime". We pick the smallest such i. Since $\sqrt{1} = \pm 1$, the value of y is $y^{2^i} = -1$ in the ith iteration, and at that stage, the correct answer is returned in step 7.

(ii) Assume N is composite and not carmicheal. Thus there exists at least fermat witness. That is, for some x values of satisfies $x^{N-1} = 1$ in $\mathbb{Z}_{\mathbb{N}}$. The set fermat liars $L_N = \{x \in \mathbb{Z}_{\mathbb{N}}^* : x^{N-1} = 1\}$ is a proper subgroup of $\mathbb{Z}_N^*$, we know $L_N < \mathbb{Z}_{\mathbb{N}}^*$ and $|L_N| \leq |\mathbb{Z}_{\mathbb{N}}^*|/2$ by the Lagrange's Theorem. For random choose $x \in \mathbb{Z}_{\mathbb{N}}^*$. It means that the probability of x being in the L_N is at least 1/2. That is, composite returns with at least 1/2 probability.

Now suppose that N is carmicheal number. Remember, carmicheal numbers doesn't have any fermat witness. Let k be order in $\mathbb{Z}_{\mathbb{N}}^*$ of x from step 2. By Lemma 5, k is even an $x^{k/2} \notin \pm 1$ with probability at least 1/2. Assuming this, we have $y \neq 1$ in step 5, and one of the repeated squares in the loop of step 6 equals 1. The previous value of y is $x^{k/2}$ and in $\sqrt{1} \backslash \pm 1$ (as we described earlier:$\sqrt{1} = \{x \in \mathbb{Z}_{\mathbb{N}} : x^2 = 1\}$), which shows the claimed lower bound on the probability to return correctly from step 9.

(iii) Step 4 can be done with $2log_2 m$ operations in $\mathbb{Z}_{\mathbb{N}}$ using the repeated squaring algorithm, and the loop of step 6 with at most e operations. The total is $2log_2 m + e \leq 2n$.

$2log_2 m + e = log_2 m^2 + log_2 2^e = log_2 m^2.2^e \leq N$

$log_2 m^2.2^e = log_2 N.m \leq log_2 N^2 = 2log_2 N = 2n$ One operation in $\mathbb{Z}_{\mathbb{N}}$ takes $O(n^2)$ bit operations, for a total of $O(n^3)$.

$\square$

For a Carmichael number N, assuming that$x^m \neq 1$in step 4, the last value of y in the loop of step 6 which is not 1 is actually in $\sqrt{1} \backslash \pm 1$, and $gcd(y - 1, N)$ is a nontrivial factor of N. Thus we can even find a factor of N from the intermediate results, with probability at least $1/2$. In fact, Carmichael numbers can be factored completely in random polynomial time.

The probability that the strong pseudoprimality test answers incorrectly “probably prime” for a composite N is at most $1/2$. When we use this test t times independently, the error probability is at most 2^{-t}.

Theorem 7. *The Miller-Rabin test, repeated t times independently, has the following properties on input N.*
(i)If it outputs “composite”, then N is composite.
(ii) If it outputs “probably prime”, then N is prime with probability at least $1 - 2^{-t}$.

Proof. (i) In cases where we check whether a number is prime or not using the properties of primes, if a number does not satisfy the property of primeness and therefore composite is returned, then that number is definitely composite. So clearly, if $gcd(x, N) \neq 1$ or $x^{N-1} \neq 1$, then the number is composite.
(ii) From Theorem 6, we deduce for any composite number N, the probability that a test will return ”probably prime” is at most $1/2$, and for t independent tests, this probability is 2^{-t}. Therefore, for a test returns ”probably prime” as a result of t independent tests, the probability that N is prime is at least $1 - 2^{-t}$. □

3.3 Finding prime numbers

Algorithm: Finding a pseudoprime
Input: An integer n and a confidence parameter t
Output: A pseudoprime number N in the range from $2^{(n-1)/2}$ to $2^{n/2}$.

```
1. $y \leftarrow 2^{(n-1)/2}$
2. Repeat steps 3 and 4 until some $N$ is returned.
3. $N \leftarrow \{\lceil y \rceil \ldots, \lfloor \sqrt{2}y \rfloor\}$
4. Call the Miller-Rabin test with input $N$ for $t$ independently chosen
   $x \leftarrow \{1, \ldots, N-1\}$. Return N if and only if all these tests return
   “probably prime”.
```

Choosing the y value $2^{(n-1)/2}$ ensures that the magnitudes for p and q are correct in the key-gen. Randomness reduces predictability. The probability of a composite number being chosen by mistake after t times of independent testing is 2^{-t}.

To find a prime number, we pick random numbers and test them until we succeed. But if, say, there are no prime numbers in the given range, the algorithm goes into an infinite loop (and therefore would not be an algorithm in the usual sense). Fortunately, this assumption is very wrong: there is not just one prime number in the desired range, there are plenty of them. We can say that there are approximately $\pi(x) \approx \frac{x}{\ln(x)}$ as $x \to \infty$ prime numbers up to some limit x, by the prime number theorem. $\pi(x)$ denote the number of primes less than or equal to x: $\pi(x) = \#\{p \leq x : p \in \mathbb{R}$, p is prime $\}$.

Theorem 8. *On input $n \geq 23$ and t, the output of Algorithm Finding a pseudoprime is prime with probability at least $1 - 2^{-t+1}n$. It uses an expected number of $O(tn^4)$ bit operations.*

Proof. The probability r for a randomly chosen integer in the range to be prime satisfies: $r > 1/n$. At first glance, it might seem that the probability of error for the test is at most 2^{-t}. But for a valid argument, we have to estimate the conditional probability that the test returns a composite number N. We denote the event that N is composite by C and the event that the test accepts N by T. We use conditional probabilities like the probability $prob\{C : T\}$ that C occurs provided that T does. This is defined as $prob\{C \cap T\} \backslash prob\{T\}$, if the denominator does not vanish. By the theorem 8, we know that $prob\{T : C\} \leq 2^{-t}$. But we want an estimate on $prob\{C : T\}$. The two are related as $prob\{T\} \cdot prob\{C : T\} = prob\{C \cap T\} = prob\{C\} \cdot prob\{T : C\}$. Since the test answers correctly for any prime, we know $prob\{Nprime\} \leq prob\{T\}$. Putting all this together, we get

$$\begin{aligned} \frac{1}{2n} prob\{C : T\} &\leq prob\{Nprime\} \cdot prob\{C : T\} \\ &\leq prob\{T\} \cdot prob\{C : T\} \\ = prob\{C\} prob\{T : C\} &\leq prob\{T : C\} \leq 2^{-t} \end{aligned}$$

Hence the probability that our algorithm returns a composite number is at most $n2^{-t+1}$. Since the probability that N is prime is at least 1/n, we expect to make at most n choices. For each choice, the test takes $O(tn^3)$ bit operations. □

For RSA, with $t = \lceil log_2(2n) \rceil + 2$ the algorithm returns a prime with probability at least $1 - 2^{-s}$ and it uses and expected number of $O((s + logn)n^4) bit operations.$ With $s = 20 + log_2 n$ the error proability is at most $0.000001/n$ and we expect $O(n^4 logn)$ bit operations. In practice, $t = 50$ seems sufficient.

To determine the public exponent e, What is the probability that two random integers are relatively prime? More precisely, when x gets large and $c_x = \#\{\ 1 \le a, b \le x : gcd(a, b) = 1\}$, we are interested in the numerical value of c_x/x^2. In fact, a famous theorem of Dirichlet (1849) says that the value is

$$\frac{c_x}{x^2} \in \frac{6}{\pi^2} + O(\frac{logx}{x}) \approx 0.6079271016 + O(\frac{logx}{x})$$

The ratio approaches $\frac{6}{\pi^2}$ as x grows. Fixing one of the arguments, the average of $\frac{\phi(x)}{x}$ also equals $6/\pi^2$ and its value is at least $1/3lnlnx$ for $x \ge 31$.

For RSA, it follows that in order to find a suitable e, we have to test an expected number of at most $O(logn)$ $(O(loglogN) = O(logn))$random values, and usually only one or two. Each test for coprimality with $p-1$ and with $q-1$ takes $O(n^2)$ bit operations.

We can now answer the efficiency questions of Section 2.1, using pseudoprimes instead of primes.

Find $n/2$-bit pseudoprimes at random $O(n^4logn)$,
Find e $O(n^2logn)$,
Calculate N and d $O(n^2)$,
Calculate powers modulo N $O(n^3)$.

Corollary 9. *The key generation in RSA can be done in time $O(n^4logn)$, and the encryption or decryption of one n-bit plaintext block with $O(n^3)$ bit operations.*

3.4 Security of RSA

We consider as an adversary a (probabilistic) polynomial-time computer A. A knows $pk = (N, e)$ and $y = enc_{pk}(x)$. Is it hard enough to calculate x for safety? There are several notions of "breaking RSA". A might be able to compute from its knowledge one of the following data.

B_1: the plaintext x,
B_2: the hidden part d of the secret key $sk = (N, d)$,
B_3: the value$\phi(N)$ of Euler's totient function,
B_4: a factor p (and q) of N.

These four questions are widely assumed to be hard. The RSA assumption is the conjecture that B_1 is hard. To describe the close relationship between these problems concisely, we use the following concept.

Definition 10. Random polynomial time reduction is connecting the solution of one problem to the solution of another problem. (i) If A and B are two computational problems (given by an input/output specification), then a random polynomial-time reduction from A to B is a random polynomial-time algorithm for A which is allowed to make calls to an (unspecified) subroutine for B. The cost of such a call is the combined input plus output length in the call. If such a reduction exists, then A is random polynomial-time reducible to B, and we write

$$A \leq_p B$$

.

If such a reduction exists that does not make use of randomization, then A is polynomial-time reducible to B.

(ii) If also $B \leq_p A$, we call A and B random polynomial-time equivalent and write

$$A \equiv_p B$$

.

The existence of a reduction $A \leq_p B$ has two consequences: if B is easy, then also A is easy, and if A is hard, then also B is hard. We will consider the concept of system breaking as a case of the adversary simply succeeding often enough, rather than the case of the adversary succeeding.

The secret keys in RSA are assumed to be generated uniformly at random. Implementations sometimes use weak random generators to produce these keys. The ensuing danger is forcefully exhibited by the real-world existence of thousands of pairs of RSA keys N_1 and N_2 with $p = gcd(N_1, N_2)$ being one of the prime factors of N_1 and N_2. RSA security is based on the uniquely and random selection of primes.

In Section 2.1, a probabilistic polynomial-time algorithm. This computation shows that

$$B_1 \leq_p B_2 \leq_p B_3 \leq_p B_4$$

If we find the prime factor p and q of N, we can also find the value $\phi(N)$. If we find the value $\phi(N)$, we can also find the hidden part d of the secret key. If we find the value d, we can also find the plaintext x. What about the converse?

Lemma 11. $B_4 \leq_p B_3$.

Proof. We know $N = p \cdot q$, and with one call to a subroutine for B_3 we find $\phi(N) = (p-1)(q-1) = pq - (p+q) + 1$. We substitute $p = N/q$ and multiply up the denominator q to find a quadratic equation in p:

$$q \cdot \phi(N) = Nq - N - q^2 + q \Rightarrow q^2 + q \cdot \phi(N) - N \cdot q - q + N = 0$$

$$\Rightarrow q^2 + q(\phi(N) - N - 1) + N = 0$$

Now we have a quadratic equation, this means we have two roots and to find roots

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

with the equation $ax^2 + bx + c = 0$. Thus, we can solve it, and have solved B_4 □

Example: We take $N = 943$ and suppose we receive $\phi(N) = 880$

$$q^2 + q(\phi(N) - N - 1) + N = q^2 + q(880 - 943 - 1) + 943 = 0$$

$$q^2 + q(-64) + 943 = 0$$

The roots $= \frac{-b\pm\sqrt{b^2-4ac}}{2a} = \frac{-(-64)\pm\sqrt{324}}{2} = \frac{64\pm18}{2} \Rightarrow root_1 = 41$, $root_2 = 23$. Thus $N = 23 \cdot 41$ $p = 23$ and q=41.

If we want to show, $B_3 \leq_p B_2$, we may proceed as follows. We keep N fixed, and call the subroutine for B_2 with several random values $e_1, e_2, \ldots$ for e. Each time we get a value $d_1, d_2, \ldots$ with $e_i d_i \equiv 1 \pmod{\phi(N)}$. In other words, $\phi(N)$ is a divisor of each $e_i d_i - 1$, and hence also of $gcd(e_1 d_1 - 1, e_2 d_2 - 1, \ldots)$.

Now if we could show that our random $e_1, e_2, \ldots$ generate random quotients $(e_1 d_1 - 1)/\phi(N)$, $(e_2 d_2 - 1)/\phi(N)$,..., then we could conclude that a few choices are already likely to give us $\phi(N)$ as gcd. This works quite well in practice.

Example: We take $N = 55$, and choose $e_1 = 7$ and $e_2 = 11$. The subroutine for B_2 returns $d_1 = 23$ and $d_2 = 11$. Then

$$\begin{aligned} g_1 &= e_1 d_1 - 1 = 160, \\ g_2 &= e_2 d_2 - 1 = 120, \\ gcd(g_1, g_2) &= gcd(160, 120) = 40. \end{aligned}$$

In fact, $g_1 = 4 \cdot 40$and $g_2 = 3 \cdot 40$. We now take 40 as a test value for $\phi(N)$, and indeed this is correct. Had we been less lucky, we would have continued with some $e_3, e_4, \ldots$.

In fact, determining $\phi(N)$ is deterministic polynomial time equivalent to computing the secret key d. In other words, $B_1 \leq_p B_2 \equiv_p B_3 \equiv_p B_4$.

Open Question: Does "$B_4 \leq_p B_1$" hold?
This is an unsolved problem and an active area of research. It asks whether efficient deciphering of RSA encryptions implies efficient factorization. This is equivalent to RSA being secure, provided that factoring RSA moduli is hard. If $B_4 \leq_p B_1$, the basic security assumption (factoring difficulty) of the RSA is weakened. And the security of RSA is directly attributed to the difficulty of factoring, if it is easy to find x if it is $B_4 \leq_p B_1$, then the fundamental security of RSA completely collapses.

Conjecture (RSA assumption): There is no probabilistic polynomial-time algorithm A which on input N, e, and y outputs x (problem B_1) with nonnegligible probability.

Nonnegligible probability is the situation where the security level is high and the probability of an algorithm finding the correct solution is almost 0.

The probability space consists of the choices of N, e, y, as prescribed, and the internal randomization of A. For "nonnegligible" and "polynomial time" to make sense, we need a security parameter n to which they refer. For any n, this yields a distribution D_n on n-bit integers, and D is the collection of these. The concept that "factoring integers is hard" can then be quantified as follows.

Conjecture (Distributional factoring assumption): Let$D = (D_n)_{n \in N}$ be a collection of distributions D_n on n-bit integers. For all polynomial-time algorithms A, the probability that A returns a proper factor on input $N \leftarrow D_n$ is negligible.

3.5 Factoring integers

One obvious way of breaking RSA, namely, by factoring N. Integers with n bits can be tested for primality in (probabilistic) polynomial time, with $O^{\sim}(n^3)$ operations. If factoring can be achieved within a similar time bound, then systems like RSA are dead. Before you try to factor an integer, you make sure it is odd, not prime and not a proper power of an integer. You might also remove small factors by taking (repeatedly) the *gcd* with a product of small primes.

3.6 Pollard rho with Floyd's trick

We choose some function $f : \mathbb{Z}_{\mathbb{N}} \to \mathbb{Z}_{\mathbb{N}}$ and a random initial value $x_0 \in \mathbb{Z}_{\mathbb{N}}$, and recursively define $x_1 \in \mathbb{Z}_{\mathbb{N}}$ bu $x_i = f(x_{i-1})$ for all $i > 0$. We hope that $x_0, x_1, x_2, \dots$ behave like a sequence of independent random elements in $\mathbb{Z}_{\mathbb{N}}$. If p is an (unknown) prime divisor of N, then we will have a collision modulo p if there are two integers t and l with $l > 0$ and $x_t = x_{t+l}$ in $\mathbb{Z}_p$. If N is not a prime power and q is a different prime divisor of N, and the x_i's are random residues modulo N, then $x_i \pmod{p}$ are independent random variables. Thus it is very likely that $x_t \neq x_{t+l}$ in $\mathbb{Z}_{\mathbb{N}}$, and then $p = gcd(x_t - x_{t+l,N})$ is a nontrivial factor of N.

We now describe Pollard's p method for factoring N. It generates a sequence $x_0, x_1, \dots \in \mathbb{Z}_{\mathbb{N}}$. We pick x_0 at random and define $x_i \in \mathbb{Z}_{\mathbb{N}}$ by $x_i = f(x_{i-1}) = x_{i-1}^2 + 1$ in $\mathbb{Z}_{\mathbb{N}}$. In fact, it doesn't matter which function you choose, it's enough to have a function that you can distribute uniformly randomly.

Let p be the smallest prime dividing N. Then we have $x_i = x_{i-1}^2 + 1$ in $\mathbb{Z}_p$ for $i \geq 1$. The birthday paradox and heuristic reasoning which says that the x_i's "look" random imply that we can expect a collision in $\mathbb{Z}_p$ after $O(\sqrt{p})$ steps.

The main idea, if $N > gcd(x_t - x_{t+l}, N) > 1$, then there is a nontrivial root here. Choose random numbers in $\pmod{N}$. If $gcd > 1$, this is also a nontrivial root. If I can find a divisor in an almost randomness array, if I know the structure of that sequence, it is easier to calculate gcd in that structure. The fact that the function is $x^2 + 1$ ensures almost randomness. You catch a loop where you can back to the same values with randomness.This idea allows us to try $\sqrt{N}$ instead of trying all the N's one by one. This is what Pollard's-rho does.The next step is generated from the previous step, so if you enter a cycle, you can't get out. This cycle structure allows Floyd to practice. Choose random numbers in $\pmod{N}$. One of them will definitely hit in $\sqrt{p}$ moves. Calculate the gcd with this randomly selected value. This is not Pollard; exactly floyd trick. Because it doesn't keep unnecessary values in memory, and step $\sqrt{p}$ is completed and the memory is reset.

Instead of storing all values for collision detection in Pollard Rho, memory usage is significantly reduced with Floyd's Cycle Detection and provides collision detection by storing a fixed number of values.

`Floyd's cycle detection trick:` Quite generally, suppose we have finite set $\mathbb{Z}$ (here: $\mathbb{Z} = \mathbb{Z}_p$), $x_0 \in \mathbb{Z}$, a function $f : \mathbb{Z} \to \mathbb{Z}$ and define

$x_i = f(x_{i-1})$.

This operation creates an infinite series because the function f always produces values that fall into the same set (for eg.$\mathbb{Z}_p$). However, the set $\mathbb{Z}_p$ is finite. So at some point the values start to repeat.

The idea of Floyd's 1-step/2-step cycle detection method is to use a second sequence $y_0, y_1, \ldots$ that iterates f with double speed, so that $y_i = x_{2i}$ for all i, and to store only the current values of x_i and y_i. The "faster" sequence "overtakes" the slower one for some i, and then we have $x_{2i} = y_i = x_i$.

Lemma 12. *In the above notation, suppose that $x_t = x_{t+l}$. Then $x_i = y_i$ for some $i < t + l$*

Proof. For any $t' \geq t$ and $j \geq 1$, we have $x_{t'} = x_{t'+jl}$. When $t = 0$, we have

$$x_0 = x_{0+l} = x_l = x_{2l} = y_l$$

and i=l sufficient.

Otherwise we set $j = \lceil t/l \rceil$ and $i = jl$. Then $j \geq 1$, $i \geq t$, and $2i = i + jl$. Thus $x_i = x_{2i} = y_i$. Furthermore, $i < (t/l + 1).l = t + l$.

We want a greater value than t so that I can get $x_t = x_{t+l}$.The main thing is whether there is a value of i less than $t + l$ so that $x_t = x_{t+l}$ □

Pollard's rho algorithm

Input: An odd composite integer N.

Output: A proper divisor of N, or "failure".

```
 1. $i \leftarrow 0$.
 2. $x_0 \leftarrow \mathbb{Z}_{\mathbb{N}}$
 3. $y_0 \leftarrow x_0$
 4. Repeat steps 5-8
 5. $i \leftarrow i + 1$
 6. $x_i \leftarrow x_{i-1}^2 + 1$ in $\mathbb{Z}_{\mathbb{N}}$
 7. $y_i \leftarrow (y_{i-1}^2 + 1)^2$ in $\mathbb{Z}_{\mathbb{N}}$
 8. $g \leftarrow gcd(x_i - y_i, N)$
 9. Until $g \neq 1$
10. If $g < N$ then return g else return "failure".
```

Theorem 13. *Let $N \in \mathbb{N}$ be a composite n-bit integer, p its smallest prime factor, and $f(x) = x^2+1$. Under the assumption that the sequence $(f^{(i)}(x_0))_{i\in\mathbb{N}}$ of iterates of x_0 behaves modulo p like a random sequence, the expected number of iterations in Pollard's algorithm for returning a proper factor of N is $O(\sqrt{p}) \subseteq O(N^{1/4})$, using an expected number of $O(N^{1/4}n^2)$ bit operations.*

Proof. Since p is at most $\sqrt{N}$, $p \leq \sqrt{N}$ because p is the smallest prime factor of N, we have $\sqrt{p} \leq N^{1/4}$, , and the Birthday Paradox Theorem implies the first claim. Each step in the algorithm can be performed with $O(n^2)$ bit operations, because It requires x^2+1, $O(n^2)$ operations that you do in each iteration. □

3.7 The birthday paradox:

This section answers the question: how "small" can we expect t and l to be? We certainly have $t + l \leq p$ (if $t + l > p$,it generated at least $p + 1$ value. But at most p values in $\mathbb{Z}_p$.) and the following analysis shows that the expected value is only $O(\sqrt{p})$ for a random sequence $x_0, x_1, \ldots$. This is known as the birthday problem: assuming peoples' birthdays occur randomly, how many people do we need to get together before we have probability at least 1/2 for at least two people to have the same birthday Surprising answer: only 23 are sufficient. In fact, with 23 or more people at a party, the probability of two coinciding birthdays is at least 50.7%.

Theorem 14. *:Birthday paradox*
We consider random choices, with replacement, among m labeled items. The expected number of choices until a collision occurs is $O(\sqrt{m})$.

Before we look the proof, we should know the definition of expectation.

Definition 15. Definition Of Expectation: [3] A random variable that can take on at most a countable number of possible values is said to be discrete. If X is a discrete random variable having a probability mass function $p(x)$, then the expected value of X is defined by

$$\sum_{i=1}^{\infty} Pr(X = x)$$

.

Proof. Let s be the number of choices until a collision occurs, that is, two identical items are chosen. This is a random variable. For $j \geq 2$.
s :The number of selections made until collision.

$prob\{s \geq j\}$: No collision until the jth election. So the collision will be j or later.

$$prob\{s \geq j\} = \prod_{1 \leq i < j} (1 - \frac{i-1}{m})$$

we have used $1 - x \leq e^{-x} \Rightarrow 1 - \frac{i-1}{m} \leq e^{1-i/m}$. The reason why you can write $1 - x \leq e^{-x}$ here can be explained by taylor series. A kind reminder, The Taylor series for the exponential function e^x is written as:$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$, and then Let's write the Taylor series for e^{-x}, replacing x with $-x$: $e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} \ldots$ If we notice, the first two terms of e^{-x} will be $1-x$. That is, $1-x$ is slightly smaller than e^-x.

$$\prod_{1 \leq i < j} (1 - \frac{i-1}{m}) \leq \prod_{1 \leq i < j} e^{-(i-1)/m}$$

$$e^{-1/m(1+2+\ldots j-1)} = e^{-(j-2)(j-1)/2m}$$

$$\Rightarrow prob\{s \geq j\} \leq e^{-(j-2)(j-1)/2m}$$

In here $(j-1)(j-2) > (j-2)(j-2) = (j-2)^2$, so that $-(j-1)(j-2) < -(j-2)^2$

Finally, putting all this together,

$$prob\{s \geq j\} \leq e^{-(j-2)(j-1)/2m} < e^{-(j-2)^2/2m}$$

,

$$E(s) = \sum_{j \geq 1} j \cdot prob\{s = j\}$$

,where s:discreate random variable.

$$prob\{s = j\} = prob\{s \geq j\} - prob\{s \geq j+1\}$$

Thus,

$$E(s) = \sum_{j \geq 1} j \cdot prob\{s \geq j\} - \sum_{j \geq 1} j \cdot prob\{s \geq j+1\}$$

, put $j-1$ instead of j the second summation :

$$\sum_{j\geq 1} j \cdot prob\{s \geq j+1\} = \sum_{j\geq 2}(j-1)\cdot prob\{s \geq j\}$$

$$= \sum_{j\geq 2}(j\cdot prob\{s \geq j\} - prob\{s \geq j\}) = \sum_{j\geq 2} j\cdot prob\{s \geq j\} - \sum_{j\geq 2} prob\{s \geq j\}$$

$$E(s) = \sum_{j\geq 1} j \cdot prob\{s \geq j\} - \sum_{j\geq 2} j \cdot prob\{s \geq j\} + \sum_{j\geq 2} prob\{s \geq j\}$$

Now, type the first term in the first summation:

$$= 1\cdot prob\{s \geq j\} + \sum_{j\geq 2} j\cdot prob\{s \geq j\} - \sum_{j\geq 2} j\cdot prob\{s \geq j\} + \sum_{j\geq 2} prob\{s \geq j\}$$

$$= 1 \cdot prob\{s \geq j\} + \sum_{j=2} prob\{s \geq j\} = \sum_{j\geq 1} prob\{s \geq j\}$$

Therefore,

$$E(s) = \sum_{j\geq 1} prob\{s \geq j\}$$

We know $prob\{s \geq j\} < e^{-(j-2)^2/2m}$

$$E(s) = \sum_{j\geq 1} prob\{s \geq j\} < \sum_{j\geq 1} e^{-(j-2)^2/2m} = \sum_{j>0} e^{-(j-1)^2/2m}$$

$$\sum_{j>0} e^{-(j-1)^2/2m} \leq 1 + \sum_{j>0} e^{-j^2/2m}$$

$$1 + \sum_{j>0} e^{-j^2/2m} \leq 2 + \int_0^\infty e^{-x^2/2m} dx$$

We know that the Gauss Integral says: $\int_{-\infty}^{\infty} e^{-ax^2 dx} = \frac{1}{2}\sqrt{\frac{\pi}{a}}$

$$\leq 2 + \sqrt{2m}\int_0^\infty e^{-x^2} dx = 2 + \sqrt{\frac{m\pi}{2}}$$

$\square$

The theorem says that we expect to find a collision with only about $\sqrt{m\pi/2} \approx 1.25\sqrt{m}$ random choices of x, where m is the number of possible values.

`Example:` For $m = 2^{42}$,the first collision is expected to occur after about $\sqrt{2^{42}\pi/2} = \sqrt{\pi/2}.2^{21} \approx 2621440$ samples. Another example: For $m = 2^{20}$,the first collision is expected to occur after about $\sqrt{2^{20}\pi/2} = \sqrt{\pi/2} \cdot 2^{10} \approx 1283.4$ samples.

3.8 Dixon's random squares

We want to apply Lemma 5 and thus are looking for two values x and y with $x^2 = y^2$ in $\mathbb{Z}_{\mathbb{N}}$, but $x \notin \pm y$. If we find such an x, y, it is possible to factor N. Dixon's random squares use smooth number for solve this problem.

Definition 16. Smooth Number
An integer N is called B-smooth number if the largest of its prime factors $\leq B$. If N is B-smooth number, it does not contain a prime factor greater than B when factored. When $N = p_1^{e_1} \cdot p_2^{e_2} \cdots p_k^{e_k}$ all prime factors provide the condition $p_i \leq B$. For an $B > 0$, the factor base is $\{p_1 \cdots p_h\}$. For example, $N = 30 = 2 \cdot 3 \cdot 5$ and if $B = 5$, then N is the 5-smooth number but N is not 3-smooth number. Its called B-number, if $b^2 \in \mathbb{Z}_{\mathbb{N}}$ is B-smooth number. For example, $N = 1517$, $b = 117$ and $b^2 = 36 = 2^2 \cdot 3^2$ in $\mathbb{Z}_{1517}$, and if $B = 5$, $b = 117$ is 5-number since $b^2 = 36$ is a 5-smooth number.

For factoring N, we write the factor base of B. The basic idea here is to multiply the B-numbers in such a way that the exponent of each prime factor in the factor base turns out to be even. So we get an equation of type $x^2 = y^2$ in $\mathbb{Z}_{\mathbb{N}}$, which allows us to factoring N.

We can relate every B- number to b as, $b^2 = p_1^{e_1} \cdot p_2^{e_2} \cdots p_h^{e_h}$ in $\mathbb{Z}_{\mathbb{N}}$. Then we can write the binary vector $\epsilon = (e_1 \mod 2, e_2 \mod 2 \cdots e_h \mod 2)$ in $\mathbb{Z}_2^h$. For example, the binary vector associated to 117 is $(1, 1, 0)$. Now the idea, it is to consider the set $b_1, b_2, \cdots b_l$ so that $\epsilon_1 + \epsilon_2 \cdots \epsilon_l = 0$ in $\mathbb{Z}_2^h$. The sum of this vectors is zero, which means that the product of the b_i^2's forms a perfectly square number. Writing e_{ij} for the exponent of p_j in b_i^2, $a_j = \sum_{i=1}^{l} e_{ij}$ is the jth component of $\sum_{i=1}^{l} \epsilon_i$ and even. Since the number is a perfect square, we take $\delta_j = a_j/2 \in \mathbb{Z}$, and $x = \prod_{1 \leq i \leq l} b_i$ and $y = \prod_{1 \leq i \leq h} p_j^{\delta_j}$ in $\mathbb{Z}_{\mathbb{N}}$. Then

$$x^2 = \prod_{1 \leq i \leq l} b_i^2 = \prod_{1 \leq i \leq l} \cdot \prod_{1 \leq j \leq h} p_j^{e_{ij}} = \prod_{1 \leq j \leq h} \cdot \prod_{1 \leq i \leq l} p_j^{e_{ij}} = \prod_{1 \leq j \leq h} \dot{\prod}_{1 \leq i \leq l} p_j^{e_{ij}}$$

$$= \prod_{1\le j\le h} p_j^{\sum_{1\le i\le l} e_{ij}} = \prod_{1\le j\le h} p_j^{a_J} = \prod_{1\le j\le h} p_j^{2\delta_J} = \prod_{1\le j\le h} (p_j^{\delta_J})^2 = y^2$$

To factor N from now on, find the $gcd(x-y,N)$ or $gcd(x+y,N)$ with Euclidean algorithm. It's something we need to be careful about, if $x \in \pm y$, doesn't work. In such a case choose another B-number. (A kind reminder: if $x^2 = y^2$ in $\mathbb{Z}_{\mathbb{N}}$, this implies $x^2 - y^2 = 0$ in $\mathbb{Z}_{\mathbb{N}}$, so that $N|x-y$ or $N|x+y$). A subtlety is that $b, b^2, x, andy$ are defined in $\mathbb{Z}_{\mathbb{N}}$, but when we look a gcd, then they are defined in $\mathbb{Z}$.

We expect that $b^2 \in \mathbb{Z}_{\mathbb{N}}$ for $b \longleftarrow \mathbb{Z}_{\mathbb{N}}$ behaves approximately like a uniformly random element of $\mathbb{Z}_{\mathbb{N}}$. Probability that a randomly chosen $b \in \mathbb{Z}_{\mathbb{N}}$ is a B-number is u^{-u}. Where $u = lnN/lnB$.

since probability that a randomly chosen $b \in \mathbb{Z}_{\mathbb{N}}$ is a B-number is u^{-u}, the expected number of trials to find

- one B-number is $1/u^{-u} = u^{-u}$
- $h+1$ B-numbers is $(h+1)u^{-u}$

Dixon's algorithm consists of two stages: 1) Collecting the relations: finding $h+1$ number B and factoring them according to the B factor base(trial division), $h \cdot u^u \cdot B \cdot n^2 \cdot$ 2) linear algebra: Finding linearly dependency using the vector of the found B-number, B^3.

1) $h \cdot u^u$: to find $h+1$ times B-number.
n^2 : Performing a Trial Division on each attempt.
B: making a trial division for each number B.
In this case $h < B$ and $h = \pi(B)$ (approximately B times.
Thus the first stages total time complexity is $O(B^2 u^u n^2)$.
2) Linear algebra is required for a linear dependent vector. Here, a matrix of $(h+1) * h$ size is operated.

$O(B^3 + u^u \cdot B^2 \cdot n^2)$

How to choose B? Choice B is very important as it directly affects time complexity. B^3 grows with B while u and $B^2u^un^2$ decrease with B.

Thus we will choose B such that both terms are approximately equal: $B^3 \approx u^u \cdot B^2 \cdot n^2$.

$$B^3 = e^{3lnB}$$

and $u = lnN/lnB, u^u = \frac{lnN}{lnB}^{\frac{lnN}{lnB}}$ and let $m = lnN$

$$ln(u^u) = \frac{lnN}{lnB} \cdot ln(\frac{lnN}{lnB}) = \frac{lnN}{lnB} \cdot (lnlnN - lnlnB) = \frac{m \cdot (lnm - lnlnB)}{lnB} = ln(u^u)$$

thus,

$$B^2 u^u n^2 = e^{2lnB + m(lnm - lnlnB)/lnB + 2lnn}$$

Ignoring the small summand $2lnn$ and equating exponents approximately yields. And then,

$$3 \ln B = 2 \ln B + m \ln B(\ln m - \ln \ln B),$$

$$\ln^2 B = m(\ln m - \ln \ln B) \approx m \ln m,$$

$$\ln B = \sqrt{m \ln m},$$

$$B = \exp(\sqrt{m \ln m}).$$

The selected value of B determines the cost.

$$O(B^3) = O(exp(3\sqrt{mlnm})$$

References

[1] Wikipedia, *Big O notation.*

[2] David B. Burton, Elementary Number Theory, *Theorem7.2*, seventh Edition.

[3] Sheldon Ross, Introduction To Probability , *Definition Of Expectation*

[4] Neal Kobiltz, Number Theory and Cryptography, second edition.

[5] Joachim von zur Gathen ,CryptoSchool.