

7 Topics in Computational Number Theory

7.1 Some Basic Concepts

Running Time: A fundamental notion that allows one to quantify and compare the efficiency of algorithms. The running time of an algorithm is the number of “basic” operations it takes to complete given an input of “size” n .

Depending on the context, “basic” operation can refer to simple arithmetic operations like addition or multiplication, or single bit operations (AND, OR, etc.), or individual CPU instructions. For the applications we consider here, it does not matter much which interpretation one uses.

The “size” of an input is measured in terms of the number of bits, or digits. Thus, for example, an integer with 1000 digits (decimal or binary) would have size around 1000. In general, if N is a large integer, then its size n is proportional to $\log N$. *It is important to keep in mind that the appropriate yardstick when measuring running times is not the size of the integer itself, but its logarithm.*

Polynomial Time: A key benchmark for running times is “polynomial time”, i.e., a running time that, for an input of size n , is of order of a fixed power of n . In a sense, this is best-possible, as reading in an input of size n in bit-by-bit fashion already requires n bit operations. Finding algorithms that are polynomial time, or proving that no such algorithms exist for a particular problem, is one of the fundamental problems in the theory of algorithms.

Factoring versus Multiplying: By the Fundamental Theorem of Arithmetic, there is a one-to-one correspondence between finite tuples of prime factors (p_1, \dots, p_r) with $p_1 \leq \dots \leq p_r$ and integers $n \geq 2$, given by the bijection $(p_1, \dots, p_r) \leftrightarrow n = p_1 \dots p_r$. However, from a computational point of view the two directions of this bijection are vastly different: Going from left to right simply requires multiplying together the prime factors p_i , a trivial operation that can be carried out in polynomial time. By contrast, the reverse direction requires factoring an integer into its prime factors and is a computationally much harder task. The fact that one direction in this equivalence is easy from a computational point of view, while the other is hard, is the basis of most modern cryptographic schemes.

Primality Testing versus Factoring: Another crucial difference from a computational point of view lies between primality testing algorithms and factoring algorithms. A factoring algorithm takes as input a number n and produces as output its complete prime factorization. By contrast, a primality testing algorithm produces as output PRIME or COMPOSITE (and possibly also INCONCLUSIVE), without exhibiting any factors.

Computationally, primality testing is much easier, and faster, than factoring. There are algorithms known that test primality in polynomial time. By contrast, there are no known polynomial time factoring algorithms, and it is conjectured that none exist.

7.2 Primality Tests

Trial Division: Let $n \geq 2$. For $d = 2, 3, \dots, \lfloor \sqrt{n} \rfloor$, check if $d \mid n$ (e.g., using division with remainder).

- If $d \mid n$ for some $2 \leq d \leq \sqrt{n}$, then n is composite, and d is a proper divisor of n .
- If $d \nmid n$ for all $2 \leq d \leq \sqrt{n}$, then n is prime.

Comments: As a general primality test, Trial Division is not practical since it requires about \sqrt{n} operations to determine the primality of an integer. However, it is useful as an initial test to detect integers that have a small prime factor and eliminate these from further consideration

before applying more sophisticated tests. Also, in contrast to all of the tests below, Trial Division produces explicit factors, and it can be used recursively to completely factor a composite number.

Wilson’s Test: Let $n \geq 2$. Compute $(n - 1)! \bmod n$.

- If $(n - 1)! \not\equiv -1 \pmod n$, then n is composite.
- If $(n - 1)! \equiv -1 \pmod n$, then n is prime.

Comments: Unlike the Fermat Test below, Wilson’s Test is an “if and only if” statement, and thus provides an ironclad guarantee of compositeness or primality. As a practical primality test, however, it is not very useful since there is no efficient (fast) way to compute $(n - 1)!$ modulo n .

Fermat Test: Let $n \geq 2$. Pick an integer a with $(a, n) = 1$, and compute $a^{n-1} \bmod n$.

- If $a^{n-1} \not\equiv 1 \pmod n$, then n is composite.
- If $a^{n-1} \equiv 1 \pmod n$, then n is likely (but not guaranteed) prime.

Comments: The Fermat test is very fast: Using repeated squaring to compute the powers a^{n-1} modulo n , it can be performed in polynomial time.

The test has no “false negatives”: If a number n fails the test, i.e., if $a^{n-1} \not\equiv 1 \pmod n$, then n is *guaranteed* composite.

The test does have “false positives”: There are numbers n that pass the Fermat test, i.e., satisfy $a^{n-1} \equiv 1 \pmod n$, but are composite. Such numbers are called *pseudoprimes* (to base a). Luckily, false positives are extremely rare. For example, out of the first billion integers n that pass the base-2 Fermat test (i.e., satisfy $2^{n-1} \equiv 1 \pmod n$), only about 20,000 are false positives (i.e., pseudoprimes). Thus, in this range, the test is 99.998% accurate. This makes the Fermat test very useful as a test that identifies “probable primes”, i.e., numbers that are, with very high probability, prime, and which can then be subjected to further tests (such as Lucas’ Test below).

Lucas Test: Let $n \geq 2$. Let a be an integer with $(a, n) = 1$, and suppose the following two conditions hold:

- $a^{n-1} \equiv 1 \pmod n$.
- $a^{(n-1)/q} \not\equiv 1 \pmod n$ for every prime divisor $q \mid n - 1$.

Then n is prime.

Comments: This test fixes the “hole” in Fermat’s test: The first condition is simply the Fermat test, while the second, additional, condition guarantees that n is indeed a prime.

The downside of Lucas’ test is that, in order to verify that this additional condition is satisfied, one first needs to find all prime factors of $n - 1$. For general integers n , this is a much harder problem than the problem of testing n for primality. However, the test can be useful for integers for which the prime factorization of $n - 1$ is known in advance, such as Fermat numbers. In the fact, in this special case conditions in Lucas’ test can be further simplified to yield the following test for the primality of Fermat numbers:

Pépin’s Test: Let $F_m = 2^{2^m} + 1$ be the m -th Fermat number. Then F_m is prime if and only if

$$3^{(F_m - 1)/2} \equiv -1 \pmod{F_m}.$$

Comments: This test, and variations of it (for example, with 3 replaced by 5), is useful in testing Fermat numbers for primality. A key feature of this test is the “if and only if” character: If the stated congruence holds, F_m is guaranteed to be prime; if it does not hold, F_m is guaranteed to be composite.

7.3 The RSA Encryption Scheme

Encryption basics: An encryption scheme takes a message M and creates an encrypted version E of this message by applying an appropriate one-to-one function, f , the *encryption function*, to M : $E = f(M)$. The associated decryption scheme works the same way, with the encrypted message E as input, and the inverse function, f^{-1} , as a *decryption function* that recovers the original message: $M = f^{-1}(E)$.

A classic example of an encryption scheme is the “Caesar cipher”, which shifts each letter in the alphabet forward by 3, so that A gets mapped to D, B maps to E, C maps to F, etc.

Private key encryption versus public key encryption: In the example of the Caesar cipher, anyone who knows the encryption function (“shift the letter forward by 3”) is able to deduce the corresponding decryption function (“shift the letter backward by 3”) and thus can decrypt any messages encrypted with the same function. Thus, in order for the encryption to remain secure, *both the encryption function and the decryption function have to be kept secret*.

Encryption schemes with this property are called **private key encryption**, or **symmetrical encryption**, as encryption and decryption play a symmetrical role, and both need to be kept “private” (i.e., secret).

By contrast, **public key encryption**, or **asymmetrical encryption**, is an encryption scheme where the encryption function f is such that knowing f does not give away the decryption function f^{-1} . Hence the encryption function f can safely be made public (in the form of the “public key”), and only the decryption function has to be kept secret (the “private key”).

An encryption function f suitable for a public key encryption must have the following properties: (i) the function f must be easy to compute; (ii) its inverse, f^{-1} , must be hard to compute; (iii) there must exist a “backdoor” approach that allows easy computation of f^{-1} given an appropriate “private key”.

The RSA encryption scheme: This encryption scheme, named after its inventors in the mid 1970s (Rivest, Shamir, and Adleman), has become the gold standard for public key encryption. It works as follows.

- Two primes, p and q , are generated, and *kept secret*. Typically, p and q will have around 100 decimal digits. Computationally, this is relatively easy to do, for example, by randomly testing numbers of the desired size for primality until a prime is found.
- The **encryption modulus**, $m = pq$, is computed, by multiplying the two primes. Computationally, this is a trivial task. The modulus m is *made public*.
- A **public encryption exponent**, e , is chosen, subject to the condition $(e, \varphi(m)) = 1$. A common choice is $2^{16} + 1 = 65,537$, the largest known Fermat prime.
- A corresponding **secret decryption exponent**, d , is computed by solving the congruence $ed \equiv 1 \pmod{\varphi(m)}$. This requires knowing $\varphi(m)$, or equivalently, the (secret) prime factorization of the modulus m . Given this prime factorization, a solution to the above congruence can be found quickly via the Euclidean algorithm.
- **Encryption and decryption:** For simplicity, assume the message M to be encrypted is a positive integer smaller than each of the primes p and q .¹ This assumption ensures that M is in the range $1 \leq M < m$ (so that M is completely determined by its remainder modulo m) and also satisfies $(M, m) = (M, pq) = 1$. The encryption and decryption functions in the RSA scheme are then defined as follows.

¹A general message can be converted to a sequence of such numbers by first encoding the characters as numerical values (e.g., ASCII codes) and then breaking the resulting numerical sequence into blocks of suitable size.

- **Encryption:** To encrypt M , compute $M^e \bmod m$. The result, expressed as the least positive residue modulo m , is the encrypted message E . The pair (e, m) consisting of the (public) encryption exponent and the (public) modulus is the public “encryption key”. Anyone who has this key (i.e., knows the values of e and m) can encrypt a message.
- **Decryption:** To decrypt an encrypted message E , compute $E^d \bmod m$. The result, expressed as the least positive residue modulo m , is the original message M ; see below for a proof. The pair (d, m) consisting of the private (i.e., secret) decryption exponent and the (public) modulus is the private “decryption key”.

Computationally, both encryption and decryption are “modular exponentiations” that can be performed quickly using the repeated squaring trick. For example, with $e = 2^{16} + 1 = 65,537$ as exponent, computing $M^e \bmod m$ requires only 17 operations.²

Why it works:

- **Existence of decryption exponent d .** The decryption exponent d is defined as a solution to the congruence $(*) \quad ed \equiv 1 \pmod{\varphi(m)}$. For the system to work, we need to be guaranteed that such a solution exists. Now, $(*)$ is a linear congruence of the form $ax \equiv 1 \pmod{n}$. By the general theory for such congruences, a solution x exists if and only if $(a, n) = 1$, and in this case there is a unique solution in the range $1 \leq x \leq n$. In the case of $(*)$, the condition for the existence of a solution d becomes $(e, \varphi(m)) = 1$, and since e was chosen to satisfy the latter condition, this shows that $(*)$ has a solution d . Moreover, the general theory guarantees that there is a solution d in the range $1 \leq d \leq \varphi(m)$.
- **Decryption returns the original message.** Here we show that the above decryption algorithm indeed returns the original message, i.e., that $E^d \equiv M \pmod{m}$.

First, note that by the congruence $(*)$, there exists a nonnegative integer k such that $de = 1 + k\varphi(m)$. Then, since $M^{\varphi(m)} \equiv 1 \pmod{m}$ by Euler’s Theorem,

$$E^d = (M^e)^d = M^{ed} = M^{1+k\varphi(m)} = M \cdot (M^{\varphi(m)})^k \equiv M \cdot 1^k = M,$$

What makes RSA (relatively) secure: The above argument shows that the RSA scheme works correctly, in the sense that the decryption function is well-defined (d always exists) and that it returns the original message.

The security of the RSA scheme, and its usefulness in practice, relies on the fact that encrypting a message using the public key (e, m) is computationally easy, while decrypting a message knowing only the public key is computationally hard, to the point of being infeasible if the modulus is in the 200+ digit range. Here is a rundown of the computational complexity of the key tasks involved:

- **Primality testing is easy:** Current algorithms can test integers of thousands of digits in fractions of a second. Thus, finding the large primes needed in the RSA scheme is easy.
- **Factoring is hard (as far as we know):** All known factoring algorithms are only effective up to a hundred digits or so. Factoring a 200+ digit integer composed of two large prime factors, such as those arising in the RSA scheme, cannot be done in a reasonable amount of time.
- **Modular exponentiation is easy:** Given an exponent e and a modulus m , computing $M^e \bmod m$ can be done quickly using the repeated squaring trick.

²Compute successively $M^2, M^{2^2} = (M^2)^2, M^{2^3} = (M^{2^2})^2, \dots, M^{2^{16}}$, and multiply the last result, $M^{2^{16}}$, by M , reducing modulo m at each stage.

- **Computing modular inverses is easy:** Solving an equation $ax \equiv 1 \pmod{n}$ for x can be done quickly by applying the Euclidean algorithm to (a, n) . Hence, determining the decryption exponent d via the congruence (*) is easy, *provided the modulus in (*), $\varphi(m)$ is known.*
- **Computing $\varphi(m)$, given m , is hard if the prime factorization of m is not known.** In fact, in the case when $m = pq$ is a product of two prime factors, computing $\varphi(m)$ is just as hard as factoring m . This makes it near impossible to compute the decryption exponent d via the congruence (*) without knowing the prime factorization of m .
- **Computing $\varphi(m)$ is easy if the prime factorization of m is known.** If the prime factorization of m is known, we can use the explicit formula $\varphi(p_1^{\alpha_1} \dots p_r^{\alpha_r}) = (p_1^{\alpha_1} - p_1^{\alpha_1-1}) \dots (p_r^{\alpha_r} - p_r^{\alpha_r-1})$. In particular, if $m = pq$, then $\varphi(m) = (p-1)(q-1)$.