Chapter 1

Algorithms with Numbers

Is 37 a prime number? Clearly, yes. Is 9679 a prime number? Yes, try all possible divisors smaller than 100. Is 999331 a prime number? Yes, just try all possible divisors smaller than 1000. Is

$$43928057009801352428703213432875417328504713460813\backslash$$
$$27403210384736021873413204861367681213 81$$

(one big number cut in two pieces) a prime number? Yes, it is, but how do we know? Is

$$32487907923450934280658726598724840035927485726300485189594381580 1\backslash$$
$$88210331384762839748213794132404213431143400135204356137463 70$$

a prime number? No, it is not. Its factors are

$$2,\ 3,\ 5,\ 7,\ 11,\ 421,\ 32794163,$$
$$33955479227018536815506600936182726861892858643489623487\backslash$$
$$27575372437232860453520141023585975933734229691820887578 3.$$

How do we know? Well, we can try small divisors up to 40 million. This will give the first seven factors. The remaining number turns out to be a prime number. How do we know that? Is

$$16342141893809875637596193384269185515133878064811498380197349391 2\backslash$$
$$08504985133173452099015606934480971397639419$$

a prime number? No, it is not. I know this because I multiplied two long prime numbers to obtain it. But my favorite computer algebra system will only tell me it

is not prime, but it is not able to find the factors in reasonable time. Is this a little impressive?

The main goal in this chapter is to set up *efficient algorithms* for testing whether a natural number (with several thousand bits, i.e., many hundreds of decimal digits) is a prime number or not, giving a yes/no answer. On the other hand, there is reason to assume that there is no efficient algorithm that will always find the prime factors of an arbitrary given number. In brief:

> **Testing primality** is "easy".
> **Factoring** is presumably "hard" (in many important cases).

Taking these two observation together, we will be able to set up a modern cryptosystem, at least in principle, and show that it is efficient and argue that it is presumably secure. This is the *RSA cryptosystem*. It is an essential tool in today's secure communication in the internet, e.g. for internet banking.

For carrying out this program, we need quite a few intermediate steps, setting up more and more algorithms for numbers.

1.1 Basic arithmetic

We work with different sets of digits.

In everyday life it is the decimal system with digits $\{0, 1, \ldots, 9\}$.

In computers it is the binary system with digits $\{0, 1\}$, or the hexadecimal system with the 16 digits $\{0, 1, \ldots, 9, \mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}\}$. (One digit needs four bits to represent.)

In general, one considers systems to a base b with $b \geq 2$. The digits are $\{0, 1, \ldots, b-1\}$.

Examples for relevant bases are $b = 2, 10, 16, 256 = 2^8, 65536 = 2^{16}, 2^{32}, 2^{64}, \ldots$. In the last four examples we use 8 bits (one byte), 16 bits, 32 bits, or 64 bits to represent one digit.

For representing a natural number $N > 0$ in a system with base b one needs exactly[1]

$$\lceil \log_b(N+1) \rceil \approx \frac{\log N}{\log b}$$

digits from $\{0, 1, \ldots, b-1\}$.

In general one needs $\lceil \log_2 b \rceil$ bits to represent a digit from $\{0, 1, \ldots, b-1\}$ in binary.

[1] Why? If N requires exactly k digits, we know that $b^{k-1} \leq N \leq \sum_{0 \leq i < k} (b-1) b^i = (b-1) \cdot \frac{b^k - 1}{b-1} = b^k - 1$. Hence $k - 1 < \log_b(N+1) \leq k$.

Since $\log_2 b$ is a constant, the representation has $\Theta(\log N)$ digits no matter what b is.

Note: If you are not sure about logarithms, read the box on page 12 in the book. Tiny correction:

4. A binary tree with N leaf nodes has depth at least $\lceil \log N \rceil$; if it is complete, N is a power of 2 and the depth is exactly $\log N$.
   A binary tree with n (inner) nodes has depth $\lfloor \log(n) \rfloor$; if it is complete, $n+1$ is a power of 2 and the depth is $\log(n+1) - 1$.

Representation of k-digit numbers in a computer: Assume a computer word can hold a digit from $\{0, \ldots, b-1\}$. (For this, its length must be at least $\log b$ bits.) Then a number x with representation $(x_{k-1}, \ldots, x_1, x_0)$ to the basis b (leading zeros are allowed) is stored in an array $A[0..k-1]$, as follows: x_i is stored in $A[i]$, for $0 \le i < k$.

1.1.1 Addition

Given a basis b, we need an elementary operation **sum3** for summing three digits, resulting in two digits (carry, sum), i.e., $\mathbf{sum3}(u, v, w) = (c, s)$ with $cb + s = u + v + w$.

Examples: $b = 2$: $\mathbf{sum3}(u, v, w) = (c, s)$ with $s = u \oplus v \oplus w$ and $c = uv \vee vw \vee uw$. (Multiplication of bits is the same as logical AND, and $\oplus$ is XOR: $u \oplus v = u + v - 2uv$. A circuit with the functionality of **sum3** is called a *fulladder*.)
$b = 10$: $\mathbf{sum3}(9, 8, 7) = (2, 4)$ (because $9 + 8 + 7 = 24$).
$b = 16$: $\mathbf{sum3}(\mathrm{A}, \mathrm{B}, \mathrm{E}) = (2, 3)$, since $10 + 11 + 14 = 35 = 2 \cdot 16 + 3$.
$b = 2^{32}$: For adding three 32-bit numbers, we can use the machine instructions. The sum s will have 32 bits, the carry will be 0, 1, or 2.

Given the **sum3** operation, we can formulate addition of two k-digit numbers in pseudocode. It is done in a single pass from the lower-order digits to the higher-order digits, and for each position we carry out one **sum3**-operation. In the book an example with bits is depicted, additions for numbers with larger bases b are done just as in school. The sum will have at most $k + 1$ digits.

Algorithm 1: Addition of two numbers

```
1 function add(A[0..k-1], B[0..k-1]):
2     (c, s) ← sum3(A[0], B[0], 0);
3     C[0] ← s;
4     for i from 1 to k-1 do
5         (c, s) ← sum3(A[i], B[i], c); C[i] ← s;
6     C[k] ← c;
7     return C[0..k].
```

The running time for adding two k-digit numbers is determined by k cycles through the loop, hence it is $O(k)$. Assume the represented numbers have n bits each. If b and the word length w are constant we will have $k = \Theta(n)$, and the running time is $O(n)$.

Can we do better? No, since we must read every one of the $2k = \Theta(n)$ input digits, which takes time $\Omega(k) = \Omega(n)$.

What about *negative integers* and *subtraction*? In one sentence: For binary representation, these operations can be reduced to addition, by an extremely clever trick called "two's complement representation". We'll discuss this issue briefly later in the context of modular arithmetic. For the moment we note that addition and subtraction of arbitrary positive and negative integers with up to n bits take time $O(n)$, "linear time".

1.1.2 Multiplication and division

Multiplication

The "school method" for multiplication of two k-digit numbers, base b (e.g. $b = 10$), runs as follows. One first creates a rectangular scheme by writing the product of $A[0..k-1]$ with $B[j] \cdot b^j$ in row j (multiply $A[0..k-1]$ by $B[j]$ and shift the result j positions to the left; the shift is practically for free). Now we have k numbers of up to $2k-1$ digits in our scheme; these numbers have to be added.

Example: $x \cdot y = 450295 \cdot 39046$

	$\overbrace{\quad\quad\quad\quad\quad\quad}^{x}$							$\overbrace{\quad\quad\quad\quad\quad}^{y=(y_4 \ldots y_0)_{10}}$					
	4	5	0	2	9	5	·	4	9	0	4	6	
				2	7	0	1	7	7	0			$x \cdot y_0$
			1	8	0	1	1	8	0				$x \cdot y_1$
		0	0	0	0	0	0	0					$x \cdot y_2$
	4	0	5	2	6	5	5						$x \cdot y_3$
1	8	0	1	1	8	0							$x \cdot y_4$
2	2	0	8	5	1	6	8	5	7	0			

This addition can be done by different methods. In school (at least in Germany) we do it column by column; the disadvantage is that carries can build up to have several digits. When programming multiplication, it is better to carry out $k-1$ additions of numbers of length up to $2k$. One could add the row $x \cdot y_1$ to the one for $x \cdot y_0$, the row for $x \cdot y_2$ to the result, and so on; in general the row for $x \cdot y_i$ is added to the sum of the rows for $x \cdot y_0, \ldots, x \cdot y_{i-1}$, for $i = 1, \ldots, k-1$. Then we never have big carries, and it is obvious that the total running time is $O(k^2)$, "quadratic time". If the bit length of the input numbers is n, the time for multiplication by the school method is $O(n^2)$.

For $b = 2$ the scheme is pictured in the book, page 13. (Beware: This case is a little simpler than for larger b like our $b = 10$.)

For programming multiplication, we use an elementary operation **maa**(u, v, w) for three digits, which multiplies u and v and adds w. The result can be represented in two digits, as a pair (c, s). (This is because for $u, v, w < b$ we have $uv + w < (b-1)^2 + (b-1) = (b-1)b < b^2$.)
In formulas: $u \cdot v + w = c \cdot b + s$.

Example, for $b = 10$: **maa**$(6, 9, 3) = (5, 7)$ since $6 \cdot 9 + 3 = 57$.

In the pseudocode program (Algorithm 2), the product is built up in an array $C[0..2k-1]$, initially zero. In rounds for $j = 0, \ldots, k-1$ one multiplies $A[0..k-1]$ with $B[j]$, giving $D[0..k]$, and adds this to the array $C[0..2k-1]$ in the right position, namely in $C[j..j+k]$.

Algorithm 2: Multiplication of two k-digit numbers

```
1  function multiply(A[0..k−1], B[0..k−1]) : C[0..2k−1]
2    // Arrays A and B contain two k-digit integers x, y ≥ 0 as input
3    // Output in C[0..2k−1]: the product x · y
4    Initialize C[0..2k−1] with zeros;
5    for j from 0 to k−1 do                  // lines 6–9: D[0..k] ← A[0..k−1] · B[j]
6      (c, s) ← maa(A[0], B[j], 0); D[0] ← s;
7      for i from 1 to k−1 do
8        (c, s) ← maa(A[i], B[j], c); D[i] ← s;
9      D[k] ← c;
10     C[j..j+k] ← add(C[j..j+k], D[0..k]);
         // add D[0..k] to the intermediate result in C[0..j+k], at the right place
11   return C[0..2k−1].
```

Note: (i) The auxiliary array $D[0..k]$ is added into $C[..]$ at the correct position. In this addition there cannot be a carry into position $C[j+k+1]$ since the product of $A[0..k-1]$ and $B[0..j]$ has at most $k+j+1$ digits. (ii) The procedure can be streamlined by combining the loops in lines **7** and **8** and the loop in the **add**-function into one loop: Add s to the correct position in $C[...]$ as it is generated. This even gets rid of the array D.

Time analysis: The main work is in the loops in lines **7** and **8** and in the addition in line **10**. One execution of the body of the j-loop needs time $O(k)$ (i-loop: $O(k)$, addition: $O(k)$). Since there are k such executions, the overall time is $k \cdot O(k) = O(k^2)$. If the factors are n-bit numbers and the base b is constant, we get running time $O(n^2)$ for multiplication.

Can we do better? It is not obvious how one can improve on Algorithm 2. However, in Chapter 2 we will see that the answer is yes. This involves a clever use of recursion and of subtraction, which is a little bit surprising, as we only wish to multiply. The

running time we obtain[2] is $O(n^{1.585})$.

Alternative: **Multiplication à la Français**[3]:

Example: $x \cdot y$ with $x = 13$ and $y = 21$. We write two columns in parallel. In the first one we start with x and double as we go down: $x, 2x, 4x, 8x, \ldots$. In the second one we start with y and keep halving (without remainder) until we hit 1: $y, \lfloor y/2 \rfloor, \lfloor y/4 \rfloor, \lfloor y/8 \rfloor \ldots$. Finally, we strike out rows where in the second column we see an even number, and add the remaining numbers in the first column.

13	21		13
26	10	strike out (10 is even)	–
52	5		52
104	2	strike out (2 is even)	–
208	1		208
sum:			273

What is happening? We note that in the first column the entry in row i is $x \cdot 2^i$, if we number the rows $i = 0, 1, 2, \ldots$. In the second column we have $\lfloor y/2^i \rfloor$ in this row. The procedure is best understood by looking at the binary representation of y. By repeated halving we keep chopping of the last bit, until only the leading bit 1 is left.

$x \cdot 2^i$	$\lfloor y/2^i \rfloor$ in binary		
$13 \cdot 2^0$	10101		$13 \cdot 2^0$
$13 \cdot 2^1$	1010	strike out (1010 ends with 0)	–
$13 \cdot 2^2$	101		$13 \cdot 2^2$
$13 \cdot 2^3$	10	strike out (10 ends with 0)	–
$13 \cdot 2^4$	1		$13 \cdot 2^4$
sum:			$13 \cdot (2^0 + 2^2 + 2^4) = 13 \cdot 21$

We observe that if y has binary representation $b_{k-1} \ldots b_1 b_0$, i.e., $x = \sum_{0 \le i < k} b_i 2^i$ for $b_0, \ldots, b_{k-1} \in \{0, 1\}$, then exactly those rows $x \cdot 2^i$ with $b_i = 1$ are kept for addition. So the result is $\sum_{0 \le i < k} b_i \cdot (x \cdot 2^i) = x \cdot \sum_{0 \le i < k} b_i 2^i = x \cdot y$.

[2]This can be further improved. In 2019 it was even shown that multiplication of two n-bit integers can be done in time $O(n \log n)$. The paper "Integer multiplication in time $O(n \log n)$" by David Harvey and Joris van der Hoeven, to appear in Annals of Mathematics, is only for mathematically trained scholars ...

[3]This is the name used by the authors of our book. Wikipedia says that the more common name for this method is "Russian peasant multiplication" and that the old Egyptians may have done it slightly differently: `https://en.wikipedia.org/wiki/Ancient_Egyptian_multiplication`

Hidden in this is also a simple recursive formula worth noticing:

$$x \cdot y = \begin{cases} 0 & \text{, if } y = 0 \\ x & \text{, if } y = 1 \\ (2 \cdot x) \cdot (y/2) & \text{, if } y \geq 2 \text{ even} \\ (2 \cdot x) \cdot ((y-1)/2) + x & \text{, if } y \geq 2 \text{ odd} \end{cases}$$

With $\lceil \alpha \rceil$ = smallest integer $\geq \alpha$ and $\lfloor \alpha \rfloor$ = largest integer $\leq \alpha$, for arbitrary $\alpha \in \mathbb{R}$, we can write (note there is a little difference to the version in the book, in the positioning of the parentheses):

$$x \cdot y = \begin{cases} 0 & \text{, if } y = 0 \\ x & \text{, if } y = 1 \\ (2 \cdot x) \cdot \lfloor y/2 \rfloor & \text{, if } y \geq 2 \text{ even} \\ (2 \cdot x) \cdot \lfloor y/2 \rfloor + x & \text{, if } y \geq 2 \text{ odd} \end{cases}$$

We get Algorithm 3, which works recursively, but when one looks closely, it just carries out multiplication à la Français.

Note that in the book the parentheses are put at different positions. This also gives a correct algorithm, but it is not really that closely related to multiplication à la Français. In our procedure the connection is clear: When it is called for $x = 13$ and $y = 21$, the recursive calls are for $(13^*, 21)$, $(26, 10)$, $(51^*, 5)$, $(104, 2)$, $(208^*, 1)$, which are just the rows in our two-column scheme from above. The numbers marked with asterisks are added, because they appear at a place where the second component is odd.

Algorithm 3: Multiplication à la Français

```
1 function mult-francais(x, y):
2     // Input: Two nonnegative n-bit integers x and y;
3     // Output: product x · y
4     if y = 0 then return 0;
5     if y = 1 then return x;
6     z ← mult-francais(2 · x, ⌊y/2⌋);
7     if y is odd then z ← z + x;
8     return z
```

Time analysis: The intermediate results are all smaller than $x \cdot y$, and hence have length not more than $2n$ bits. In binary, multiplication by 2 is a shift to the left, division by 2 is a shift to the right: both need only linear time. If the basis is not 2, the multiplication $2 \cdot x$ can be effected by the addition $x + x$, and halving y can be done by iteratively treating the digits of y from the most significant digits to the least significant digits, again in linear time. The addition in line **7** takes time $O(n)$. How many recursive calls do we have ("depth of recursion")? With each recursive call y is halved, meaning that the number of bits of y reduces by 1. So the number of recursive calls is not larger than n. Total time: $O(n^2)$.

Division

Task: Given $x \in \mathbb{Z}$ and $y \in \mathbb{N} \setminus \{0\}$, calculate the *quotient* $q = \lfloor x/y \rfloor$ and the *remainder* $r = x - q \cdot y$.

We can also describe q and r as the unique integers that satisfy $x = qy + r$ with $0 \leq r < q$.

Examples: $x = 13$ divided by $y = 3$ gives quotient $q = 4$ and remainder $r = 1$. $x = -13$ divided by $y = 3$ gives quotient $q = \lfloor (-13)/3 \rfloor = -5$ and remainder $r = 2$.

We could take the "school method" for dividing integers (but note: "school method" may mean different things in different countries!) and show it takes time $O(n^2)$ for n-bit integers.

However, we choose as alternative the following recursive procedure.

Algorithm 4: Division

```
1  function divide(x, y):
2      // Input: Two n-bit integers x, y, with y ≥ 1
3      // Output: (q, r): quotient and remainder when dividing x by y
4      if x = 0 then return (0, 0);
5      if x = −1 then return (−1, y − 1);
6      (q, r) ← divide(⌊x/2⌋, y);    // Recursion! Division by 2 takes time O(n)
7      q ← 2 · q; r ← 2 · r;          // Multiplication by 2 takes time O(n)
8      if x is odd then r ← r + 1;
9      if r ≥ y then (q, r) ← (q + 1, r − y);
                                      // Comparison, subtraction take time O(n)
10     return (q, r).
```

Correctness can "easily" be checked, by induction on recursive calls. The program terminates since in each recursive call $|x|$ strictly decreases. **(Watch out! In the book line 5 is missing, and the program runs into an infinite loop for all inputs (x, y) with negative x, since the case $x = -1$ leads to a recursive call with $x = \lfloor (-1)/2 \rfloor = -1$ again.)** It is essential that in line 8 the number r is smaller than $2y$.

Running time: The depth of the recursion is n, since in each recursive call x is halved. On each level of the recursion we do multiplications and divisions by 2, additions and subtractions, which all take $O(n)$ time. Total time: $O(n^2)$.

1.2 Modular arithmetic

Please read pages 16–17 in the book.

We use the following notation, for integers x, N, where $N \geq 2$:

$$x \bmod N := r, \text{ where } (q, r) \text{ is the result of dividing } x \text{ by } N.$$

(Read: "*x modulo N*".) This means: $0 \leq r < N$, and we can write $x = qN + r$ for some q. Alternatively: $0 \leq r < N$, and $x - r$ is divisible by N.

Examples: $10 \bmod 7 = 3$, $3 \bmod 7 = 3$, $-4 \bmod 7 = 3$.

Closely related, but a little different, is the following helpful notation:

$x \equiv y \pmod N$, if $x \bmod N = y \bmod N$. (Read: "*x is congruent to y modulo N*".)

Examples: $10 \equiv 3$, and $3 \equiv 3$, and $-4 \equiv 3 \pmod 7$.

Claim: $x \equiv y \pmod N$ holds if and only if N is a divisor of $x - y$.

Proof: Assume $x \equiv y \pmod N$. Let $r = x \bmod N = y \bmod N$. Then N divides $x - r$ and $y - r$, so N divides $(x-r)-(y-r) = x-y$. Now assume that N divides $x-y$. Let $r = x \bmod N$ and $r' = y \bmod N$. Then N divides $(x - r) - (y - r') = (x - y) + (r' - r)$. Hence N divides $r - r'$, a number with absolute value smaller than N. This implies $r - r' = 0$. □

From the definition it is immediate that the binary relation $\cdot \equiv \cdot \pmod N$ is an *equivalence relation*[4] on $\mathbb{Z}$. So it splits $\mathbb{Z}$ into *equivalence classes*. Two numbers x and y are in the same equivalence class if $x \equiv y \pmod N$. In our example, one equivalence class is

$$\{\ldots, -11, -4, 3, 10, 17, \ldots\}.$$

We write $[x]$ or $[x]_N$ for the equivalence class that contains x.

We get exactly N equivalence classes, one for each number r with $0 \leq r < N$: $[0], [1], \ldots, [N-1]$.

There are two possibilities for calculating "modulo N": One could write a sequence

$$\text{expr}_1 \bmod N = \text{expr}_2 \bmod N = \cdots = \text{expr}_s \bmod N,$$

by somehow making sure that the remainder does not change from expression to expression. Or one writes

$$\text{expr}_1 \equiv \text{expr}_2 \equiv \cdots \equiv \text{expr}_r \pmod N,$$

again making sure that the remainders always stay the same. The second way is a little more compact and neater, since there are fewer "mod N". So we will prefer it.

We list some rules for transforming expressions so that the remainders do not change. When calculating "modulo N" with additions, subtractions, and multiplications, we can arbitrarily substitute elements of one equivalence class for each other. This helps in keeping arguments small. In formulas:

[4] It is **reflexive** ($x \equiv x$), **symmetric** (if $x \equiv y$ then $y \equiv x$), and **transitive** (if $x \equiv y$ and $y \equiv z$ then $x \equiv z$).

Substitution rules

$$\begin{aligned} \text{If } x \equiv x', y \equiv y' \pmod N \text{ then } x + y &\equiv x' + y' \pmod N, \\ x - y &\equiv x' - y' \pmod N, \\ x \cdot y &\equiv x' \cdot y' \pmod N, \\ x^c &\equiv x'^c \pmod N. \end{aligned}$$

From this it follows that if we have an arbitrary arithmetic expression involving $+, -, \cdot$ and even powers (with equal exponents c), we can substitute congruent numbers or expressions for numbers or subexpressions, without changing the remainder. In particular we can always and everywhere replace a number z by its remainder z mod N (since the two are congruent). This helps a lot in evaluating seemingly complicated expressions, Sometimes it is also advantageous to chose another equivalent number, as in the following example, modulo 33:

$$2^{1001} = 2^{1000} \cdot 2 = (2^5)^{200} \cdot 2 = 32^{200} \cdot 2 \equiv (-1)^{200} \cdot 2 = 1 \cdot 2 = 2 \pmod{33}.$$

We'll see a systematic way of carrying out modular exponentiations with really large exponents very efficiently!

By the way, the three rules marked "Associativity", "Commutativity", "Distributivity" on page 17 of the book are trivial (since, eg., in $xy = yx$ we even have equality, which is stronger than congruence). So you can ignore these rules.

1.2.1 Modular addition and multiplication

Please read page 18 in the book.

Modular addition. The task is the following. Given are $x, y \in \{0, \ldots, N-1\}$, where N has at most n bits. Calculate $(x + y)$ mod N.

At the first glance this takes time $O(n)$ for the addition and $O(n^2)$ for the modulo operation (division), but it is better. We calculate $z = x + y$ (time $O(n)$). We know that $z < 2N$. So there are only two possibilities: Calculate $u = z - N$ and compare u with 0 (this also takes time $O(n)$). If $u < 0$, the result is z; if $u \geq 0$, the result is u.

Modular subtraction $(x-y)$ mod N has the same running time, with a similar method.

Modular multiplication. The task is the following. Given are $x, y \in \{0, \ldots, N-1\}$, where N has at most n bits. Calculate $(x \cdot y)$ mod N.

We need a multiplication, to obtain $z = x \cdot y$, and a division, to calculate the remainder z mod N. Both operations take time $O(n^2)$, so the total time is $O(n^2)$.

There is also an operation called **modular division**. We postpone the discussion of that operation.

1.2.2 Modular exponentiation

Warning: Our method is a little different from that in the book.

Here we consider the following task: Given natural numbers $N \geq 2$, $x < N$, and y, with at most n bits each, calculate $x^y \bmod N$.

Example: Calculate $4321^{7893685021942617015} \bmod 5123$. The result is 840. How does one reach this result? Actually, it can be calculated by hand, with a good measure of patience. (Not an exam problem!)

Modular exponentiation is used in the RSA cryptosystem mentioned before, and there the numbers used, in particular N and y, will have 2000 bits or more, or 600 decimal digits or more. How can a computer evaluate such a formidable expression in fractions of a second? Of course, there is no chance to apply an algorithm that calculates $x^y \bmod N$ by $y - 1 \approx 10^{600}$ multiplications modulo N. Rather, we need a trick. We start with an example.

Example: Calculate $x^y \bmod N$ with $x = 13$, $y = 21$, and $N = 19$. We write two columns. In the first column we start with x and keep squaring, to obtain $x, x^2 \bmod N, x^4 \bmod N, x^8 \bmod N, \ldots$, in the second we start with y and keep halving to obtain $y, \lfloor y/2 \rfloor, \lfloor y/4 \rfloor, \lfloor y/8 \rfloor \ldots$. Finally, we multiply those powers of x from the first column where the second column has an odd entry. (You should be reminded of multiplication à la Français.)

x	13	21		13
x^2	$13^2 \bmod 19 = 17$	10	strike out (10 is even)	–
x^4	$17^2 \bmod 19 = 4$	5		4
x^8	$4^2 \bmod 19 = 16$	2	strike out (2 is even)	–
x^{16}	$16^2 \bmod 19 = 9$	1		9
Product:				$x \cdot x^4 \cdot x^{16} \bmod 19$
				$= 13 \cdot 4 \cdot 9 \bmod 19 = 12.$

General idea: Write $y = (b_{k-1} \ldots b_0)_2 = \sum_{0 \leq i < k} b_i 2^i$ in binary. Calculate powers $x^{2^i} \bmod N$ for $i = 0, 1, \ldots, k-1$ by repeated squaring. Then calculate the product

$$\prod_{i:\, b_i = 1} x^{2^i} \bmod N \qquad (= x^{\sum_{0 \leq i < k} b_i \cdot 2^i} \bmod N = x^y \bmod N),$$

by at most $k - 1$ multiplications modulo N. – There is a simple recursive way of looking at this, similar to the situation with multiplication à la Français.

$$x^y \bmod N = \begin{cases} 1 & \text{, if } y = 0 \\ x & \text{, if } y = 1 \\ (x^2 \bmod N)^{\lfloor y/2 \rfloor} \bmod N & \text{, if } y \geq 2 \text{ even} \\ ((x^2 \bmod N)^{\lfloor y/2 \rfloor} \bmod N \cdot x) \bmod N & \text{, if } y \geq 2 \text{ odd} \end{cases}$$

The formula is obviously correct. (We use the substitution rule and that $2\lfloor y/2 \rfloor = y$ if y is even and $2\lfloor y/2 \rfloor = y - 1$ if y is odd.) We get the program in Algorithm 5.

Algorithm 5: Fast modular exponentiation

```
1 function modexp(x, y, N):
2 INPUT: nonnegative n-bit integers x, y, N, N ≥ 2, x < N;
3 OUTPUT: x^y mod N
4   if y = 0 then return 1;
5   if y = 1 then return x;
6   z ← modexp((x · x) mod N, ⌊y/2⌋, N);
7   if y is odd then z ← (z · x) mod N;
8   return z
```

An example calculation was presented above ($13^{21} \bmod 19$).

Time analysis: The intermediate results calculated in the algorithm are all modulo N, hence they have length not more than n bits. The modular multiplications in lines **6** and **7** take time $O(n^2)$. The number of recursive calls is not larger than n. Thus the total time is $n \cdot (O(n^2) + O(n^2)) = O(n^3)$.

The procedure becomes faster if faster subroutines for multiplication and division are employed.

Intermezzo: Two's complement. (This refers to the box on page 17 in the book. It is only for interested students and will **not** be **in the exam**!) I assume you know that negative integers are stored and processed in a computer by a technique called "two's complement". This makes it possible to use the same electronic circuits for adding and for subtracting integers. I do not explain here how it is done from the point of view of circuits, only how to look at it from the perspective of modular arithmetic.

Have you seen the following trick for "subtracting by adding"? It is called "9's complement". Assume we want to carry out the subtraction $a-b = 415-378$. We replace every digit t in the subtrahend $b = 378$ by its complement $9-t$ to 9. This yields $\bar{b} = (9-3)(9-7)(9-8) = 621$. Now sum $a+\bar{b}+1 = 415+621+1 = 1037$. Omit the leading 1. The result 37 is the desired difference $a-b$. Why does this work? We calculate modulo $N = 1000$. We have $b+\bar{b} = 999$ by construction, so $b+\bar{b}+1 = 1000$, hence $a+\bar{b}+1 = a+(1000-b-1)+1 = 1000+(a-b)$.

Some things have to be considered when one wants to use this systematically, be able to detect arithmetic overflow like a subtraction $a-b$ with $a < b$, and if one wants to enable representing negative numbers. The whole thing turns out to be about calculating modulo some even number N.

Assume n is the bitlength of the nonnegative integers we want to represent. Let $N = 2^n$. (However, everything works also for arbitrary even numbers N, in particular for the powers of 10. For these we get a systematic version of 9's complement, including representation of negative numbers.) We assume we have a circuit (or algorithm) C that can add two numbers a' and b' from $[0, N)$, with a little twist: The circuit takes as additional input one bit c_0, and it calculates the sum $s' = (a'+b'+c_0) \bmod N$ and an "overflow bit" c which is 0 if $a'+b'+c_0 < N$ and 1 otherwise. So we have:

$$a'+b'+c_0 = s'+cN.$$

Given a circuit or a method for addition, it is normally no problem to accommodate the additional input bit c_0 and also output an overflow bit. We do not worry about the details.[5]

In our examples we use $n = 5$, and hence $N = 32$.

Examples:
$C(9, 20, 0)$ gives $s' = 29$ and $c = 0$ (because $9+20+0 = 29 = 29+0\cdot 32$).
$C(27, 22, 0)$ gives $s' = 17$ and $c = 1$ (because $27+22+0 = 49 = 17+1\cdot 32$).
$C(27, 4, 1)$ gives $s' = 0$ and $c = 1$ (because $27+4+1 = 32 = 0+1\cdot 32$).

In order to represent also negative numbers, the trick in two's complement representation is to keep calculating modulo N but to "interpret" some numbers differently. A number $a' \in [0, N/2)$ represents itself, but $a' \in [N/2, N/2)$ represents $a'-N$, which is negative.

In the following table we list the 5-bit strings as well as their numerical values $a' \in [0, 31]$ and the integer $a \in [-16, 15]$ represented by a', in decimal notation.

[5] In pseudocode, one can look at Algorithm 1. One gets the desired behaviour by replacing **sum3**$(A[0], B[0], 0)$ by **sum3**$(A[0], B[0], c_0)$ in line **2**. The overflow bit is the last carry c in line **6**, which can only be 0 or 1.

binary	a'	a	binary	a'	a
00000	0	0	10000	16	-16
00001	1	1	10001	17	-15
00010	2	2	10010	18	-14
00011	3	3	10011	19	-13
00100	4	4	10100	20	-12
00101	5	5	10101	21	-11
00110	6	6	10110	22	-10
00111	7	7	10111	23	-9
01000	8	8	11000	24	-8
01001	9	9	11001	25	-7
01010	10	10	11010	26	-6
01011	11	11	11011	27	-5
01100	12	12	11100	28	-4
01101	13	13	11101	29	-3
01110	14	14	11110	30	-2
01111	15	15	11111	31	-1

If a' represents a, we always have $a \equiv a' \pmod{N}$. For example: $-14 \equiv 18 \pmod{32}$. We call a representation $a' \in [0, N)$ for a *positive* if a is positive and *negative* if a is negative. We abbreviate this as $\text{sign}(a) = \text{sign}(a') = 0$ for "positive" and $\text{sign}(a) = \text{sign}(a') = 1$ for "negative". Note that in case $N = 2^n$ the sign of a resp. a' is the leading bit of the binary representation.

We next discuss how to add two numbers represented in our system, and a bit. Let a and b be given by their representations a' and b', and let $c_0 \in \{0, 1\}$. We feed a', b', and c_0 into the addition circuit C and get a sum $s' \in [0, N)$ and an overflow bit c. Question: Is s' the representative of $a + b + c_0$? We must consider four cases.

Case "00": $a = a'$ and $b = b'$ are positive. — Then $a, b < N/2$, and hence $0 \leq a + b + c_0 < N$, so $c = 0$ and $s' = a + b + c_0$. If s' is positive (i.e., $\text{sign}(s') = c$), then s' is the representation of $a + b + c_0$. If s' is negative (i.e., $\text{sign}(s') \neq c$), then the sum $a + b + c_0 \geq N/2$ cannot be represented in our system.

Examples: We add 5 and 8 and 0. $C(5, 8, 0)$ gives $s' = 13$ (positive) and $c = 0$. If we add 11 and 8 and 1, the sum 20 does not have a representation. $C(11, 8, 1)$ gives $s' = 20$ (negative) and $c = 0$.

Case "11": $a = a' - N$ and $b = b' - N$ are negative. — Then $N \leq a' + b' + c_0 < 2N$, so circuit C will return $s' = a' + b' + c_0 - N = a + b + c_0 + N$ and $c = 1$. If $s' \geq N/2$, i.e., s' is negative, then s' is the correct representation of $a + b + c_0$. If $s' < N/2$, i.e., positive, then $a + b + c_0 = s' - N < -N/2$ cannot be represented in our system.

Examples: We add -3 and -10 and 0. $C(29, 22, 0)$ gives $s' = (29 + 22) - 32 = 19$ (negative) and $c = 1$. The result s' is the correct representation of -13. Now we add -11 and -10 and 1. There is no representation of the sum -20. $C(21, 22, 1)$ gives $s' = 44 - 32 = 12$ (positive) and carry bit $c = 1$.

Case "01" or "10" (these cases are symmetric): One of a and b is positive, one is negative. For example: $a = a'$ is positive and $b = b' - N$ is negative. — Then $-N/2 \leq$

$a+b+c_0 < N/2$, hence the sum $a+b+c_0$ has a representation. If $a+b+c_0 \geq 0$, we have $a'+b'+c_0 \geq N$, hence circuit C gives result $s' = a'+b'+c_0-N = a+b+c_0$. If $a+b+c_0 < 0$, we have $a'+b'+c_0 < N$, hence circuit C gives result $s' = a'+b'+c_0 = a+b+c_0+N$, which is the representation of $a+b+c_0$. Thus, the result s' is always correct in this case.

Examples: We add -3 and 10 and 0. $C(29,10,0)$ gives $s' = (29+10+0)-32 = 7$, which is the correct representation of 7. Now we add 10 and -16 and 1. $C(10,16,1)$ gives $s' = 27$, which is the correct representation of -5.

Summary of addition in two's complement:
Given are a and b, represented as a' and b', and bit c_0.
1) Apply C to a', b', c_0 to obtain s' and c.
2) If $\text{sign}(a') \neq \text{sign}(b')$, then s' represents $a+b+c_0$.
3) Otherwise s' represents $a+b+c_0$ if and only if $\text{sign}(s') = c$.
(If this is not satisfied, $a+b+c_0$ cannot be represented in our system. An overflow occurs. But the result s' is correct modulo N.)

In one logical expression, with $1 = \textit{true}$ and $0 = \textit{false}$, and $\oplus$ denoting XOR:

The result s' is correct if and only if $(\text{sign}(a') \oplus \text{sign}(b')) \vee (\text{sign}(s') = c)$.

Subtraction: Now we want to carry out subtractions $a-b$, for arbitrary numbers a and b that can be represented in our system, positive or negative.

The first step is to find the representation of $-b-1$, given the representation b' of b. This is very easy:[6] Let $\overline{b'} := N-1-b'$. If $b \geq 0$, then $b = b' < N/2$, and hence $\overline{b'} \geq N/2$, so that $\overline{b'}$ represents $\overline{b'} - N = -b-1$. If $b < 0$, then $b' = b+N \geq N/2$, and hence $\overline{b'} < N/2$, so that $\overline{b'}$ represents itself, and $\overline{b'} = N-b'-1 = -b-1$.

It is worth noting that $\overline{b'}$ is extremely easy to calculate in binary representation: Just flip (i.e., negate) all the bits of b'. This is because $N-1$ has as binary representation a word consisting of n 1's.

Example: The number 12 has binary representation 01100. Flipping all the bits gives 10011, which is $19 = 31-12$, the representation of $-13 = -12-1$. Conversely, flipping all the bits in 10011, which is 19 and represents -13, gives 01100, which represents $31-19 = 12 = -(-13)-1$.

All that remains is to calculate $a'+\overline{b'}+1$. For this, we can use the modified addition circuit C described above. Namely, we let $c_0 = 1$. The correctness criteria can be taken from addition. We get that the result s' is correct if either $\text{sign}(a') = \text{sign}(b')$ or $\text{sign}(s') = c$.

Finally we can explain how to use the modified addition circuit C to carry out either addition or subtraction. Let the inputs a and b be given by their representations a' and b', and the instruction whether we want to add or subtract. (This is usually a bit in the instruction word in the computer.)

[6] Compare the "9's complement" above, where $N = 1000$ and $N-1 = 999$.

Addition: To calculate a representation for $a+b$, feed $(a', b', 0)$ into C, yielding (s', c). Then s' is the correct result if and only if $\text{sign}(a') \neq \text{sign}(b')$ or $\text{sign}(s') = c$.

Subtraction: To calculate a representation for $a - b$, get $\overline{b'}$ by flipping all bits in b' and then feed $(a', \overline{b'}, 1)$ into C, yielding (s', c). The result s' is correct if and only if $\text{sign}(a') = \text{sign}(b')$ or $\text{sign}(s') = c$.

End of intermezzo on two's complement representation

1.2.3 The Euclidean Algorithm

Definition 1.2.1
Let x and y be integers. We say that x divides y, in symbols $x \mid y$, if there is some $q \in \mathbb{Z}$ such that $x \cdot q = y$.
$d \geq 0$ is a common divisor of x and y if $d \mid x$ and $d \mid y$.

$$\gcd(x,y) = \begin{cases} \text{largest } d \text{ that is a common divisor of } x \text{ and } y & \text{if } x^2 + y^2 > 0 \\ 0 & \text{if } x = y = 0. \end{cases}$$

Note that all $x \in \mathbb{Z}$ are divisors of 0, and that 0 only divides 0, no other number. The condition "$x^2 + y^2 > 0$" is a way of expressing that not $x = y = 0$.

How do we find the greatest common divisor of two numbers $a, b \geq 0$? In school one learned that one can find the prime representation of both numbers, and then read off the "common prime factors".

Example: $a = 120$ and $b = 140$. We write $120 = 2^3 \cdot 3 \cdot 5$ and $b = 2^2 \cdot 5 \cdot 7$, hence $\gcd(a,b) = 2^2 \cdot 5 = 20$.

Unfortunately, this "method" is not suitable for use with numbers of the size we want to treat. No algorithm is known that will find the prime factors of numbers of several hundred decimal digits really fast. (As said before, the notorious difficulty of the factoring problem is at the heart of the security of central cryptographic systems that are in everyday use today.) Still, it is possible to calculate greatest common divisors extremely fast, by an algorithm that is more than 2000 years old.

This algorithm is based on the following rules:

(i) $\gcd(x,y) = \gcd(y,x)$.

(ii) $\gcd(x,y) = \gcd(|x|,|y|)$.

(iii) $\gcd(a,0) = a$, for $a \geq 0$.

(iv) $\gcd(a,b) = \gcd(b, a \bmod b)$, for $b > 0$.

These rules are easily justified. (i) holds because the concept of "common divisors" does not look at the order of x and y. (ii) follows from the fact that $x \mid y \Leftrightarrow (-x) \mid y \Leftrightarrow x \mid (-y)$ for arbitrary integers x and y, since divisibility does not change if one replaces x by $-x$ or y by $-y$. For (iii) we first consider the case $a > 0$. Then a is a common divisor of 0 and a, and of course the largest one. If $a = 0$, we have $\gcd(a,0) = 0 = a$ by definition. Finally, for (iv) we observe that the set of common divisors of a and b is the set of common divisors of b and $a \bmod b$. Indeed, write $a = qb + r$ for $r = a \bmod b$. Now if d divides a and b, then d also divides $a - qb = r$. Reversely, if d divides b and r, then d also divides $qb + r = a$.

Rule (ii) allows us to reduce the problem to calculating $\gcd(a, b)$ for $a, b \geq 0$. Rules (iii) and (iv) are utilized in the recursive algorithm 6, the ancient "Euclidean Algorithm". Rule (iv) is used for the recursive step and rule (iii) for the basis of the recursion.

Algorithm 6: The Euclidean Algorithm

```
1 function Euclid(a, b):
2 INPUT: Two n-bit integers a, b ≥ 0;
3 OUTPUT: gcd(a, b)
4   if b = 0 then return a else return Euclid(b, a mod b);
```

Correctness follows from rules (iii) and (iv). If in the first call $a = b = 0$, the result is correct by definition. So assume otherwise. If in the first call we have $b > a \geq 0$, then the first recursive call will have the arguments $(b, a \bmod b) = (b, a)$. From then on in all recursive calls the first argument will be larger than the second one, since for $b > 0$ we have $a \bmod b < b$.

Examples: If the original argument is $(56, 231)$, the next calls will be for $(56, 231)$, $(231, 56)$, $(56, 7)$, $(7, 0)$, and the result is 7. For the original argument $(120, 140)$ we will have calls for $(140, 120)$, for $(120, 20)$, and for $(20, 0)$, with result 20. If the original argument is $(0, 15)$, the next call will be for $(15, 0)$, with result 15.

If we want to calculate $\gcd(x, y)$ for arbitrary integers x, y, we simply call **Euclid**$(|x|, |y|)$.

How long does the Euclidean algorithm take on input (a, b) if both a and b have at most n bits? In one recursive call, we must divide a by b to obtain the remainder $a \bmod b$. This takes time $O(n^2)$. So it remains to determine the number of recursive calls. Consider one call, for (a, b). We can assume that $a > b > 0$, since the other cases are resolved by at most one extra recursive call. If b divides a, the algorithm stops in the next round. Otherwise the next two calls are for $(b, a \bmod b)$, then $(a \bmod b, r)$, for some $r \geq 0$. So two recursive calls reduce the first component from a to $a \bmod b$. We claim that $a \bmod b < a/2$. (The reason for this is the following: If $b \leq a/2$, we have $a \bmod b < b \leq a/2$. If $b > a/2$, or $2b > a$, then $\lfloor a/b \rfloor = 1$ and $a \bmod b = a - b < a - a/2 = a/2$.) So in two rounds the number of bits of the first argument is at least reduced by 1. This entails that there are at most $2n$ recursive calls. Overall, the running time of the algorithm is $2n \cdot O(n^2) = O(n^3)$.

Fact. As a closer analysis shows, which takes the binary length of the intermediate arguments into account, the Euclidean algorithm will even have running time $O(n^2)$ on n-bit arguments. Experience shows that it is also *very fast in practice.*

1.2.4 The Extended Euclidean Algorithm

The following lemma is in the book only in implicit form.

Lemma 1.2.2 ***Bezout's Lemma***
For all integers a and b there are integers x and y such that

$$\gcd(a, b) = xa + yb.$$

Examples: $a = 5$, $b = 8$, $\gcd(5, 8) = 1$, and $1 = 5 \cdot 5 + (-3) \cdot 8 = (-3) \cdot 5 + 2 \cdot 8$.
$a = 21$, $b = -35$, $\gcd(21, -35) = 7$, and $7 = 2 \cdot 21 + 1 \cdot (-35) = (-3) \cdot 21 + (-2) \cdot (-35)$.

From the examples one sees that the coefficients x and y are not unique.

For our purposes it is of central importance that such coefficients x and y do not only exist, but can even be calculated efficiently. This is done by an extension of the Euclidean Algorithm. Of course, existence is proved if we can show these coefficients can be calculated!

The idea of the algorithm is the following. We recursively calculate $d = \gcd(a, b)$ and coefficients x and y as in Bezout's lemma. For this, we run the Euclidean Algorithm, but with a little more bookkeeping. If the argument is (a, b), we calculate the quotient $q = \lfloor a/b \rfloor$ and the remainder $r = a \bmod b = a - qb$. The algorithm is called recursively for (b, r), just as the usual Euclidean algorithm, and yields result (x', y', d'). From these we can easily calculate d, x, y.

Algorithm 7: The Extended Euclidean Algorithm

```
1 function extendedEuclid(a, b):
2 INPUT: Two n-bit integers a, b ≥ 0;
3 OUTPUT: (x, y, d) such that d = gcd(a, b) = xa + yb.
4   if b = 0 then return (1, 0, a);
5   (q, r) ← divide(a, b); // quotient q = ⌊a/b⌋ and remainder r = a − qr
6   (x', y', d) ← extendedEuclid(b, r);
7   return (y', (x' − q · y'), d).
```

Example: Consider input $(20, 14)$. We list the ensuing recursive calls with arguments (a, b) and $(q, r) = (\lfloor a/b \rfloor, a \bmod b)$.

$(20, 14)$: $(q, r) = (1, 6)$.
$(14, 6)$: $(q, r) = (2, 2)$.
$(6, 2)$: $(q, r) = (3, 0)$.
$(2, 0)$: bottom of recursion.

The results of the recursive calls, from bottom to top, are:

$(2,0)$: $(1,0,2)$
$(6,2)$: $(0,(1-1\cdot 0),2) = (0,1,2)$
$(14,6)$: $(1,(0-2\cdot 1),2) = (1,-2,2)$
$(20,14)$: $(-2,(1-1\cdot(-2)),2) = (-2,3,2)$.

In the example we check that indeed we have $(-2)\cdot 20 + 3\cdot 14 = -40 + 42 = 2$.

In order to show that the algorithm is correct we use (generalized) induction on b. If $b = 0$, then $\gcd(a,b) = a = 1\cdot a + 0\cdot b$. (Instead of 0 we could choose any other coefficient for y.) Now assume $b > 0$. Then lines **5**–**7** are carried out. By the induction hypothesis, line **6** yields (x', y', d) with

$$\gcd(b,r) = d = x'b + y'r = x'b + y'(a - qb) = y'a + (x' - qy')b.$$

Since $\gcd(b,r) = \gcd(b, a \bmod b) = \gcd(a,b)$, this shows that the output $(y', (x' - qy'), d)$ produced in line **7** is as required.

The running time for the extended Euclidean algorithm is the same as for the simple algorithms, up to constant factors. When called on n-bit integers, each recursive call takes time $O(n^2)$ for multiplications, divisions and one subtraction. The number of recursive calls is at most $2n$. So one immediately sees that the overall time is $O(n^3)$. A more detailed analysis shows that the running time even is $O(n^2)$. As the simple algorithm, the extended Euclidean algorithm is very fast in practice.

1.2.5 Modular division

In our discussion of modular arithmetic we considered addition, subtraction, multiplication, and exponentiation. What was missing was division, for the good reason that it is a subtle issue, and that we need the extended Euclidean algorithm to master it.

What could "modular division b/a modulo N" mean? We try the following formulation: Given a, b, and N, find some number x such that $ax \bmod N = b$.

For the moment, we study the special case $b = 1$. Then we are given a and N and are looking for some x such that $ax \bmod N = 1$. Such an x is called a *modular inverse of a modulo N*. (We will come back to the case of general b at the end and in the exercises.)

Example: Let $N = 14$. We list some inverses, which can be found by just trying some numbers, and some problematic cases.

a	inverse	comment
5:	3	since $5\cdot 3 \bmod 14 = 1$
7:	??	can't find a suitable x
8:	??	can't find a suitable x
9:	11	since $9\cdot 11 \bmod 14 = 99 \bmod 14 = 1$

About the case $a = 7$: One notices that $7x$ is always divisible by 7, and so it can never be of the form $qN + 1 = 14q + 1$ for any q. Similarly, for $a = 8$: Since $8x$ is always even, it cannot be of the form $qN + 1 = 14q + 1$ for any q. Obviously, we need to find a rule for deciding when a modular inverse does exist and when it does not exist.

Theorem 1.2.3 ***Modular division theorem***
Let $N \geq 1$ and let a be an integer. Then there is some integer x such that $ax \bmod N = 1$ if and only if $\gcd(a, N) = 1$.

A quick check with our example from above: $\gcd(5, 14) = \gcd(9, 14) = 1$, and $\gcd(7, 14) = 7$ and $\gcd(8, 14) = 2$.

Proof: "⇒": Assume that we have $ax \bmod N = 1$. That means there is some q such that $ax = qN + 1$. If d is a common divisor of a and N, then d divides $1 = ax - qN$. So the greatest common divisor of a and N is 1.
"⇐": Now assume $\gcd(a, N) = 1$. By Bezout's lemma there are integers x, y with $xa + yN = 1$. Then clearly $ax \bmod N = (1 - yN) \bmod N = 1$. □

Note that in case a modular inverse x exists, we can always find one in the range $\{1, \ldots, N-1\}$: If $ax \bmod N = 1$, then also $a(x \bmod N) \bmod N = 1$, so $x' = x \bmod N$ is as desired.

How can we calculate a modular inverse? That is easy, since the extended Euclidean algorithm calculates the coefficients x and y whose existence is claimed in Bezout's lemma.

Algorithm 8: Modular inverses

```
1 function modularInverse(a, N):
2 INPUT: Integers N ≥ 1 and a
3 OUTPUT: x ∈ [N] with ax mod N = 1, if it exists, "unsolvable" otherwise.
4     (x, y, d) ← extendedEuclid(a, N);
5     if d = 1 then return x mod N else return "unsolvable".
```

What is the time for calculating modular inverses? Well, the running time of the algorithm is dominated by that of the call to **extendedEuclid**(a, N), which is $O(n^2)$. We can calculate modular inverses just as fast (up to constant factors) as we can do usual division with the school method!

(The following segment was not treated in class, but it is the basis of Exercise 2(c) on problem sheet number 3.)

Let us briefly return to the question of dividing two arbitrary numbers a and b "modulo N". That means, we are looking for some y such that

$$ay \equiv b \pmod{N}, \text{ i.e. } ay + zN = b \text{ for some integer } z. \qquad (*)$$

When can such a "division modulo N" be carried out, and how can we calculate a solution y if it exists?

Consider $d = \gcd(a, N)$. If $ay + zN = b$ for some z, then clearly b must also be divisible by d. This leads to the following:

Fact. If b is not divisible by $\gcd(a, N)$, then $(*)$ is unsolvable.

Example: $15y \equiv 16 \bmod 21$ has no solution, since $\gcd(15, 21) = 3$ does not divide 16.

From here on we assume that $d \mid b$. We let $a' = a/d$, $b' = b/d$, and $N' = N/d$. Then $(*)$ is equivalent to $a'y + zN' = b'$.

Example: From $20y \equiv 28 \bmod 84$ (i.e., $a = 20$, $b = 28$, $N = 84$) we get $d = 4$ and the equivalent relation $5y \equiv 7 \bmod 21$.

This leaves us with an equation $(*)$ in which a and N are relatively prime. We use Algorithm 8 to calculate a modular inverse $x = a^{-1} \bmod N$ of a and let $y = bx \bmod N$. Then $ay \mod N = abx \mod N = (ax \bmod N)b \bmod N = b \bmod N$, as desired.

Example: $a = 5$, $b = 7$, $N = 21$. Then one sees (or finds with the extended Euclidean algorithm) that $17a = 85 \equiv 1 \pmod 7$, so that the modular inverse of a is $x = 17$. Then we let

$$y = bx \bmod 21 = 7 \cdot 17 \bmod 21 = 119 \bmod 21 = 14.$$

We check: $ya = 14 \cdot 5 = 70 \equiv 7 \pmod{21}$.

This y is also a solution to the original relation $20y \equiv 28 \bmod 84$.

Primality testing: Motivation

I tried to get my computer algebra program to its limits by demanding: Give me the smallest prime number that is larger than

134091873409818526874639871487392107450981591384038764198327098157103\
487035178560873607813463412908437998709817309481324601329582137409871\
630587610876143987109384791561653298174313048971364287612847321364987\
132694871390423798315759861875638741632087469123871479132804387183468\
756198724632874612341324986081734087343209813265387560238461938705193\
285709187439132874691382764982137157674314035891058931604130487134601\
783403187356038716487031643876148730487167654764654865847587657813764\
391287436983174698748371496000152038**04**

(Just one big number.) This number has 521 decimal digits, the number of bits in its binary representation is 1728. In a split second the program delivers the answer

134091873409818526874639871487392107450981591384038764198327098157103\
487035178560873607813463412908437998709817309481324601329582137409871\
630587610876143987109384791561653298174313048971364287612847321364987\
132694871390423798315759861875638741632087469123871479132804387183468\
756198724632874612341324986081734087343209813265387560238461938705193\
285709187439132874691382764982137157674314035891058931604130487134601\
783403187356038716487031643876148730487167654764654865847587657813764\
391287436983174698748371496000152041**79**

It seems the program checked something like 375 numbers for being prime. Well, even numbers are never prime, so it will be more like 186. At least the program must have some idea why the number it gives me is supposed to be prime. How does it do this? How does software that creates keys for public key cryptography generate huge primes like this in no time at all?

Of course, methods we all know from school like trying a bunch of possible divisors will never work for immense numbers of this kind, even on a supercomputer.

The following sections will explain.

1.3 Primality Testing (The Fermat Test)

Recall that an integer $N \geq 1$ is a *prime number* if it is divisible by exactly two numbers, namely by 1 and by N. A number N is *composite* if it has a divisor x with $1 < x < N$. Note that 1 is neither a prime number nor composite. An integer $N \geq 2$ is

composite if and only if it is not prime. The smallest prime number is 2, the smallest composite number is 4.

We are looking for algorithms that can distinguish primes and composites. Clearly, even integers $N \geq 3$ cannot be prime, so we only look at odd numbers $N \geq 3$.

An *ideal primality test* is an algorithm that takes as input an odd number $N \geq 3$ and outputs

- 0 ("prime"), if N is a prime number;
- 1 ("composite"), if N is composite.

Is this an easy or a hard problem? Note that the numbers we wish to treat may have thousands of binary digits (see motivating example), or maybe 500 or 1000 decimal digits. If n is the number of binary digits of N, then $2^{n-1} \leq N < 2^n$, so $n - 1 \leq \log N < n$. As before, we measure the running time of any algorithms in terms of n.

From high school we all know a very simple algorithm, called *trial division*: For $x = 3, 5, 7, 9, \ldots, N - 3$ check if x is a divisor of N. The running time is $\Theta(N \cdot n^2) = \Theta(2^n \cdot n^2)$, so it is exponential in terms of n. One can save a lot of time by observing that if there is a factor $x < N$ at all then there must be a factor $x \leq \sqrt{N}$. So we test all odd numbers $x \leq \sqrt{N}$ to see if x divides N. The running time for this is $\Theta(\sqrt{N} \cdot n^2) = \Theta(2^{n/2} \cdot n^2)$. This is still exponential, and it is definitely infeasible for numbers with more than 50 decimal digits. Naive improvements like dividing only by prime numbers smaller than $\sqrt{N}$ help a little, but not much.

We now look at a very old[7] theorem from number theory. This will help us in formulating a faster primality test.

Example 1.3.1
$2^6 \bmod 7 = 8^2 \bmod 7 = 1^2 \bmod 7 = 1$,
$3^6 \bmod 7 = 9^3 \bmod 7 = 2^3 \bmod 7 = 8 \bmod 7 = 1$,
$4^6 \bmod 7 = 16^3 \bmod 7 = 2^3 \bmod 7 = 1$.
$5^{30} \bmod 31 = 125^{10} \bmod 31 = (4 \cdot 31 + 1)^{10} \bmod 31 = 1^{10} \bmod 31 = 1$.

It is no coincidence that we always get 1.

Theorem 1.3.2 ***Fermat's Little Theorem***
If p is a prime number, then for all $a \in \{1, 2, \ldots, p - 1\}$ we have

$$a^{p-1} \bmod p = 1.$$

[7] Pierre de Fermat (1601–1665) was a French mathematician.

Proof of Fermat's Little Theorem. Fix a with $1 \leq a < p$. Multiply each element of $S = \{1, 2, \ldots, p-1\}$ by a (modulo p). This gives numbers

$$(1 \cdot a) \bmod p, (2 \cdot a) \bmod p, \ldots, ((p-1) \cdot a) \bmod p.$$

Example 1.3.3
For $p = 7$ and $a = 4$ we have $S = \{1, 2, 3, 4, 5, 6\}$ and $((1 \cdot 4) \bmod 7, (2 \cdot 4) \bmod 7, \ldots, (6 \cdot 4) \bmod 7) = (4, 1, 5, 2, 6, 3)$, our six numbers in S in different order.

These p numbers are in S, and they must all be different: Since p is prime and $a < p$, we have $\gcd(a, p) = 1$, hence a has a multiplicative inverse a^{-1} mod p, by the modular division theorem. So assume $1 \leq i < j < p$. Then $f(i)$ must be different from $f(j)$, since

$$(a^{-1} \cdot f(i)) \bmod p = (a^{-1} \cdot a \cdot i) \bmod p = i \neq j = (a^{-1} \cdot a \cdot j) \bmod p = (a^{-1} \cdot f(j)) \bmod p.$$

An injective function from the finite set S to S must be bijective. Hence we have $\{(1 \cdot a) \bmod p, (2 \cdot a) \bmod p, \ldots, ((p-1) \cdot a) \bmod p\} = S$. $(*)$

We multiply all numbers in S, once in the natural order, once in the order given in $(*)$, and get (always calculating modulo p):

$$1 \cdot 2 \cdot \cdots \cdot (p-1) \equiv (1 \cdot a) \cdot (2 \cdot a) \ldots ((p-1) \cdot a) \equiv (1 \cdot 2 \cdot \cdots \cdot (p-1)) \cdot a^{p-1}.$$

The number $(p-1)! = 1 \cdot 2 \cdot \cdots \cdot (p-1)$ is not divisible by p, since none of the factors is. (Here we use again that p is prime.) So $X = (p-1)! \bmod p$ is not divisible by p. Hence X^{-1} mod p exists and we can multiply our equation $X \equiv X \cdot a^{p-1} \pmod p$ by X^{-1} mod p on both sides, to obtain $1 \equiv a^{p-1} \pmod p$, or $a^{p-1} \bmod p = 1$. □

Example 1.3.4
$p = 7$ and $a = 4$ (Ex. 1.3.3 continued): $6! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \equiv (1 \cdot 4) \cdot (2 \cdot 4) \cdot (3 \cdot 4) \cdot (4 \cdot 4) \cdot (5 \cdot 4) \cdot (6 \cdot 4) \equiv (6!) \cdot 4^6 \pmod 7$. Multiplying by $((6!) \bmod 7)^{-1} \bmod 7$ gives $1 \equiv 4^6 \pmod 7$, as desired.

Now assume N is given. The idea for the primality test is to pick some a from $\{1, 2, \ldots, N-1\}$ *at random* and calculate a^{N-1} mod N. If N is prime, the result must be 1. If N is composite, we somehow *hope* there is a fair chance that a^{N-1} mod N is not 1.

Running time: The main effort is in the fast exponentiation, hence the whole algorithm takes $O(n^3)$ time.

Behavior: We say that "N passes the Fermat test" if the a chosen by the algorithm satisfies $a^{N-1} \bmod N = 1$ (the output is 0, read as "I guess N is prime"). We say that

Algorithm 9: Fermat Test

```
1 function Fermat(N):
2 INPUT: Odd integer N ≥ 3.
3   choose a at random from {1, 2, ..., N − 1};
4   b ← a^{N−1} mod N;    // fast exponentiation
5   if b ≠ 1 then return 1 else return 0.    // shorter: return (b!=1)
```

good a's with $\gcd(a, 91) > 1$:	
7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84; 13, 26, 39, 52, 65, 78	
bad a's:	*good* a's with $\gcd(a, 91) = 1$:
1, 3, 4, 9, 10, 12, 16, 17, 22,	2, 5, 6, 8, 11, 15, 18, 19, 20,
23, 25, 27, 29, 30, 36, 38, 40, 43,	24, 31, 32, 33, 34, 37, 41, 44, 45,
48, 51, 53, 55, 61, 62, 64, 66, 68,	46, 47, 50, 54, 57, 58, 59, 60, 67,
69, 74, 75, 79, 81, 82, 87, 88, 90	71, 72, 73, 76, 80, 83, 85, 86, 89

Table 1.1: Good and bad a's for $N = 91 = 7 \cdot 13$. There are 36 bad a's and 36 good a's that are relatively prime to 91. It is easy to see that all 18 multiples of the factors 7 and 13 are good.

"N fails the Fermat test" if $a^{N-1} \bmod N \neq 1$ (the output is 1, read as "I know that N is composite"). It is important to notice that if N is composite then it is a matter of chance if N passes or fails the test.

If N is a prime number, then N always passes the Fermat test, no matter which a is chosen, by Fermat's little theorem. So the output 0 is correct.

From here on, we assume that $N \geq 9$ and N is *composite*. What can be said about the behavior of the algorithm then? The output may be 0 or 1, depending on the random choice of a. If the chosen number a is such that N fails the Fermat test (output 1), then $a^{N-1} \bmod N \neq 1$. We can conclude that N is not a prime number, and confidently output 1. Strange but true: In this case we know that N is composite although we have *not found a divisor of* N.

We call an integer $a < N$ *bad* (for our purpose of finding out that N is composite) if N passes the Fermat test when a is chosen, and we say a is *good* if N fails the Fermat test when a is chosen. For the test to work well with a randomly chosen a, many a's should be *good* and few a's should be *bad*. For an illustration of the situation see Table 1.1. There all $a < N = 91$ are classified into *good* ones (54) and *bad* ones (36). So if we pick a at random the probability that N passes the Fermat test (i.e. that we choose a *bad* a) is $\frac{36}{90} = 0.4$.

Observation: If $\gcd(a, N) > 1$, then a is *good*.
(Proof by contraposition. Assume a is bad, i.e., $a^{N-1} \bmod N = 1$. Then $a \cdot ((a^{N-2}) \bmod N) \bmod N = 1$, hence a has a multiplicative inverse modulo N. The modular division theorem [page 23 in the book] says that then $\gcd(a, N) = 1$.)

So from here on we concentrate on the behavior of the algorithm on a with $\gcd(a, N) = 1$. (In the example in Table 1.1 these are the 72 numbers in the two lower parts.) There are "stubborn" numbers for which the Fermat test usually does not help much.

Definition 1.3.5
A composite odd number $N \geq 9$ is called a **Carmichael number** if $a^{N-1} \bmod N = 1$ for all a with $\gcd(a, N) = 1$.

(This means that for a Carmichael number N *all* numbers $a < N$ with $\gcd(a, N) = 1$ are *bad*. If we draw for N the picture given in Figure 1.1 for 91, then the lower right rectangle is completely empty.) The three smallest Carmichael numbers are $561 = 3 \cdot 11 \cdot 17$, $1105 = 5 \cdot 13 \cdot 17$, and $1729 = 7 \cdot 13 \cdot 19$. There are infinitely many such numbers.

For the moment we only consider non-Carmichael composite numbers N. For these the Fermat test has good behaviour.

Proposition 1.3.6
If N is composite non-Carmichael number, then at least half of the numbers a with $\gcd(a, N) = 1$ are *good*.

Proof. Let $B_N := \{a \in \{1, \ldots, N-1\} \mid a^{N-1} \bmod N = 1\}$ be the set of *bad* numbers, and let $G_N := \{a \in \{1, \ldots, N-1\} \mid a^{N-1} \bmod N \neq 1\}$ be the set of *good* numbers. Our plan for the proof is to show that every *bad* number $a \in B_N$ has a "*good* twin" $f(a) \in G_N$ (this statement implies the claim).

Let us define a mapping f, as follows (see Figure 1.1). Since N is not a Carmichael number, there is some $b < N$ with $\gcd(b, N) = 1$ and $b \in G_N$. We define

$$f \colon B_N \to \{1, \ldots, N-1\} \text{ by } f(a) := (b \cdot a) \bmod N.$$

The function f is one-to-one, since we can recover a from $f(a)$ by multiplying with $b^{-1} \bmod N$. Since both b and a are relatively prime to N, also their product is, so $\gcd(f(a), N) = 1$. And $f(a)$ is *good*, since

$$\begin{aligned} f(a)^{N-1} \bmod N &= ((b \cdot a) \bmod N)^{N-1} \bmod N \\ &= (b^{N-1} \cdot \underbrace{(a^{N-1} \bmod N)}_{\equiv 1 \pmod N}) \bmod N = b^{N-1} \bmod N \neq 1. \end{aligned}$$

□

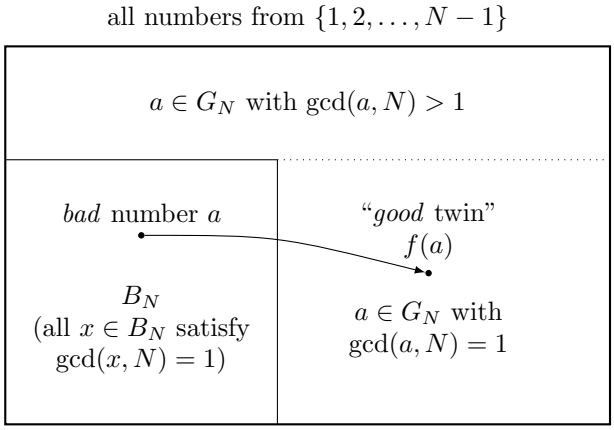

Figure 1.1: Choices of a for the Fermat test, where B_N is the set of *bad* numbers and G_N is the set of *good* numbers. The mapping f maps each bad a to a "*good* twin" $f(a)$ with $\gcd(f(a), N) = 1$. (Cf. Table 1.1 for the subdivision for $N = 91$.)

From the proposition it follows that there are more *good* a's than *bad* a's. Namely, for every *bad* $a \in B_N$ we have its "*good* twin" $f(a) \in G_N$; in addition, G_N contains all numbers $a < N$ with $\gcd(a, N) > 1$ (of which there is at least one, since N is composite). Thus, we have $|B_N| < |G_N|$.

We can turn this into a probability statement. Assume N is not a Carmichael number. We choose a at random. The probability that the answer given by the Fermat test is "0" (which is wrong), i.e., that N passes the test, is

$$\frac{|B_N|}{|B_N| + |G_N|} < \frac{1}{2}.$$

Thus, we have shown:

Theorem 1.3.7
The Fermat test has the following behaviour on input N (an odd number ≥ 3):

- If N is prime, then N passes the Fermat test.
- If N is composite, non-Carmichael, then $\mathbf{Pr}(N \text{ passes the Fermat test}) < \frac{1}{2}$.

(There is no statement for the case that N is a Carmichael number.)

An error probability of close to $\frac{1}{2}$ will be too large in most cases. One obtains a better algorithm by running the Fermat test several times (with a new randomly chosen number a each time), see Algorithm 10.

Running time: For n-bit integers N, Algorithm 10 takes $O(\ell \cdot n^3)$ time.

Algorithm 10: Iterated Fermat Test

```
1 function IterFermat(N, ℓ):
2 INPUT: Odd integer N ≥ 3, number ℓ of iterations
3   repeat ℓ times
4     choose a at random from {1, 2, ..., N − 1};
5     b ← a^{N−1} mod N;
6     if b ≠ 1 then return 1
7   return 0
```

Behavior: If N is a prime number, the output will be 0. Assume N is a composite non-Carmichael number. Then the probability that there will be ℓ rounds in each of which we choose some "bad" a is smaller than

$$\underbrace{\frac{1}{2} \cdot \frac{1}{2} \cdot \cdots \cdot \frac{1}{2}}_{\ell \text{ factors}} = \frac{1}{2^\ell}.$$

Theorem 1.3.8
The iterated Fermat test with ℓ repetitions has the following behaviour on input N (an odd number ≥ 3):

(i) If N is prime, then N passes the iterated Fermat test.

(ii) If N is composite, non-Carmichael, then
$\mathbf{Pr}(N \text{ passes the iterated Fermat test}) < \frac{1}{2^\ell}$.

(There is no statement for the case that N is a Carmichael number.)

We add remarks on changes we can make to also accommodate Carmichael numbers. As said before, the smallest Carmichael numbers are $561 = 3 \cdot 11 \cdot 17$, $1105 = 5 \cdot 13 \cdot 17$, and $1729 = 7 \cdot 13 \cdot 19$. All Carmichael numbers have three or more prime factors, each one occurring only once. If all these prime factors are large, many a's will N make pass the Fermat test, an undesirable behaviour. The good news is that seemingly Carmichael numbers are rare. So if N is picked at random, the probability that it is one of these "stubborn" numbers is very small, and we could even rely on this.

The following algorithm can deal also with Carmichael numbers. As the Fermat test, it picks a number a at random and calculates $a^{N-1} \bmod N$. But in a sense it does this calculation "in slow motion" and looks at intermediate results. More precisely, it writes the exponent $N - 1$ (which is even) as a product $u \cdot 2^t$ of an odd number u and a power of 2. Then it considers $b_0 = a^u \bmod N$, then $b_1 = b_0^2 \bmod N$,..., then $b_i = b_{i-1}^2 \bmod N$, ..., then $b_t = b_{t-1}^2 \bmod N$. One easily sees that

$$b_0 = a^u \bmod N, \ldots, b_i = a^{u \cdot 2^i} \bmod N, \ldots, b_t = a^{u \cdot 2^t} \bmod N = a^{N-1} \bmod N.$$

If $b_0 = 1$ or the sequence $b_0, \ldots, b_{t-1}$ contains $N-1$, the algorithm outputs 0, otherwise it outputs 1.

Algorithm 11: Miller-Rabin Primality Test

```
1  function MillerRabin(N):
2  INPUT: Odd integer N ≥ 3.
3    Find odd u and t ≥ 1 such that N − 1 = u · 2^t;
4    choose a at random from {1, 2, ..., N − 1};
5    b ← a^u mod N; // fast exponentiation
6    if b ∈ {1, N − 1} then return 0; // Case 1
7    for j from 1 to t − 1 do// "repeat t − 1 times"
8      b ← b^2 mod N;
9      if b = N − 1 then return 0; // Case 2
10     if b = 1 then return 1; // Case 3
11   return 1. // Case 4
```

It is easily seen that the numbers $b_0, b_1, \ldots, b_{t-1}$ appear in this order in variable b, unless the algorithm stops somewhere on the way and gives an output. This is because $(a^{u \cdot 2^{i-1}})^2 \bmod N = a^{u \cdot 2^i} \bmod N$. The number b_t is not calculated.

Running time: The fast exponentiation with exponent u needs about $\log u$ rounds with at most two modular multiplications each. The $t-1$ squaring rounds need one modular multiplication each. So the total number of rounds is $\approx \lceil \log u \rceil + t = \lceil \log(N-1) \rceil$, which is $\leq n$. Each round takes time $O(n^2)$. So the total time is $O(n^3)$.

Behavior of the Miller-Rabin algorithm on input N:

(i) If N is a prime number, the output is 0.

(ii) If N is composite, we have $\mathbf{Pr}(\text{output is } 0) \leq \frac{1}{4}$.

The proof of (ii) is somewhat involved and needs a little number theory. But we can explain (i). Output 1 can appear only in the following situations:

"Case 3": $b_0, \ldots, b_{i-1} \notin \{1, N-1\}$ and $b_i = 1$, for some $i < t$. – Then we have found some number $c = b_{i-1}$ with the property that $c^2 \bmod N = 1$, but $c \notin \{1, N-1\}$. (If b_{i-1} were one of these numbers, the algorithm would have stopped in the round before.) Such a thing is impossible if N is a prime number.

(Assume p is prime. If $c^2 \bmod p = 1$, then $c^2 - 1 = (c+1)(c-1)$ is divisible by p. Since p is prime, it is a divisor of $c+1$ or of $c-1$, hence $c \equiv p-1$ or $c \equiv 1 \pmod{p}$.)

"Case 4": We have to discuss the number b_t (which is not computed!). There are two subcases:

Subcase 4a: $b_t \neq 1$. – Since $b_t = a^{N-1} \bmod N$, it follows from Fermat's little theorem that N is not a prime number.

Subcase 4b: $b_t = 1$. – Then $c = b_{t-1}$ has the property that $c^2 \bmod N = 1$, but $c \notin \{1, N-1\}$. We conclude as in Case 3 that N cannot be a prime number.

If the Miller-Rabin test is repeated ℓ times (output 0 if in all ℓ trials the output is 0), we get the following behaviour:

(i) If N is a prime number, the output is 0.

(ii) If N is composite, then $\mathbf{Pr}(\text{output is } 0) \leq \frac{1}{4^\ell}$.

Remark: In 2002, the Indian computer scientists Agrawal, Kayal, and Saxena published a polynomial-time (in n) *deterministic* primality test (no random experiments). It never makes errors. This was a breakthrough result in algorithmics. However, the running time of the algorithm (there are variants that run in time $O(n^6(\log n)^c)$ for a constant c) is not competitive with the randomized tests by Miller and Rabin or by Solovay and Strassen. So in practice these randomized tests are used.

1.3.1 Generating random primes

Assume some n is given and we want to *generate* a prime number with n bits. In cryptographic applications, n could be 512 or 1024 or 2048 or the like.

No efficient algorithm is known for systematically searching for such huge primes. One resorts to a very simple method:

> Pick some odd n-bit number N at random
> and test it with the iterated Miller-Rabin test.

(Random odd n-bit numbers look like this in binary: $1b_{n-2} \ldots b_1 1$, with $n-2$ random bits between the two 1's.) It will turn out that with quite high probability we will have hit a composite number, so we should repeat this step until we find a number that passes the iterated Miller-Rabin test.

Some questions arise: How long does this take? And: Since the Miller-Rabin test has a certain probability of error, what is the risk of outputting a composite number?

To answer the first question, we need to have an idea of how many prime numbers there are in $\{1, 2, 3, \ldots, 2^n - 1\}$. For getting an estimate, we can utilize the famous prime number theorem. For arbitrary $x \geq 0$ we let

$$\pi(x) := |\{p \leq x \mid p \text{ is a prime number}\}|.$$

Algorithm 12: Generating a random prime

```
1 function GeneratePrime(n, ℓ):
2 INPUT: bitlength n, number ℓ of iterations
3 repeat
4     choose odd integer N at random from [2^{n-1}, 2^n) = {2^{n-1}, ..., 2^n − 1}
5 until MillerRabin(N, ℓ) = 0
6 return N.
```

Theorem 1.3.9 ***Prime number theorem***

$$\lim_{x\to\infty} \frac{\pi(x)}{x/\ln x} = 1. \qquad \text{(Often written as: } \pi(x) \sim \tfrac{x}{\ln x}.\text{)}$$

The slight disadvantage of this formulation is that it does not tell us how fast the quotient $\frac{\pi(x)}{x/\ln x}$ approaches 1. But be assured:[8] For $x \geq 500\,000$) we have $\frac{x}{\ln x} < \pi(x) < 1.1\frac{x}{\ln x}$. This implies that in Algorithm 12, line **4**, for $n \geq 20$ the probability that N is a prime number is at least

$$\begin{aligned}
\frac{|\{p \in [2^{n-1}, 2^n) \mid p \text{ is prime}\}|}{2^{n-2}} &= \frac{\pi(2^n) - \pi(2^{n-1})}{2^{n-2}} \\
&\geq \frac{2^n/(n\ln 2) - 1.1 \cdot 2^{n-1}/((n-1)\ln 2)}{2^{n-2}} \\
&\geq \frac{4}{n\ln 2} - \frac{2.2}{n\ln 2} \cdot \frac{n}{n-1} \\
&\geq \frac{2.4}{n}.
\end{aligned}$$

Please make a note of this:

The fraction of prime numbers among the odd n-bit numbers is at least $\frac{12}{5n}$.

It follows that the expected number of repetitions of the loop in Algorithm 12 until a prime number is hit (so that it definitely stops) is at most $n/2.4$.

What about the running time? Well, we know that one execution of MillerRabin(N, ℓ) takes time $O(\ell \cdot n^3)$. How big should we choose ℓ? The answer depends on how close to 1 we want to have the probability that the output is a prime number. Number theorists have shown that for any fixed $\ell \geq 1$ and for n large enough the probability that a composite number is returned is not more than $4^{-\ell} = 2^{-2\ell}$. This gives:

[8] This follows from an elegant estimate by Pierre Dusart, see Corollary 5.2 in: Dusart, Pierre. Explicit estimates of some functions over primes. Ramanujan J 45, 227–251 (2018). `https://doi.org/10.1007/s11139-016-9839-4`.

The expected running time for generating a random n-bit prime with error probability $\leq 4^{-\ell}$ is $O(n^4\ell)$.

For all practical applications it suffices to have an error probability smaller than 10^{-30}, say, that means $2^{2\ell} > 10^{30}$, or $2\ell \ln 2 > 30 \ln 10$, or $\ell > 50$. In practice, much smaller repetition numbers are used, without ever causing any known difficulties. Also, in practice before applying the Miller-Rabin test to N in Algorithm 12 one will use trial division for all prime numbers up to some number like 100 and in this way unmask many composites very quickly, cutting down the running time by a big constant factor. This has the consequence that it is no problem to generate random primes with several thousand bits very fast on a standard computer, in spite of its theoretical running time of $O(n^4)$.

1.4 Cryptography

The situation: Alice wants to send messages to Bob. A *message* or *plaintext* is a bit string of length n, or, equivalently, an integer $x < 2^n$. Typical values are $n = 1020$ or $n = 2040$; we will see why below. If longer messages must be sent, they are split into shorter pieces, which are treated one after the other. There is an "eavesdropper", Eve, who can listen in to the communication channel, which may be a telephone line or an internet connection. Alice and Bob want to make it very hard or even impossible or Eve to find out what the message is.

Let X (a set of bitstrings) be the set of plaintexts. Let Y (also a set of bitstrings) be a set of possible cryptotexts. In principle Alice and Bob want to use

- an *encryption function* $E\colon X \to Y$, which translates each plaintext into a cyphertext and
- a *decryption function* $D\colon Y \to X$, which translates each cyphertext into a plaintext.

Idea: Alice computes $y = E(x)$, sends y to Bob, Bob calculates $z = D(y)$, which should be x. For this to work, we need:

$$D(E(x)) = x \text{ , for all } x \in X. \qquad \textbf{(Decryption condition)}$$

Further, calculating $E(x)$ and $D(y)$ should be efficient.

Eve only sees y, and we hope she cannot calculate x from y. So it should be impossible or at least very "hard" for Eve to calculate x from $y = E(x)$. However, if D is a fixed function, it is sufficient for Eve to know D (get it by bribing Bob's assistant), and she can decrypt all future messages.

Example: Caesar (more than 2000 years ago): Replace each letter by the one that is three positions further down in the Latin alphabet, like[9] `e` $\mapsto$ `H`.

x	t h e q u i c k b r o w n f o x j u m p s o v e r t h e l a z y d o g
y	W K H T X L F N E U R Z Q I R A M X P S V R Y H U W K H O D C B G R J

Decryption: Replace each letter by the one that is three positions before in the alphabet.

1.4.1 Private-key schemes: One-time pad

The big disadvantage of the Ceasar cypher is that everybody who knew the trick and the number 3 could read all encrypted messages. Better: Use a "secret key", $k < 26$, say. Encryption: Replace each letter by the one that is k positions further down in the alphabet. Decryption: Same with "k positions earlier in the alphabet". In the next table we use $k = 13$, so that `a` $\mapsto$ `N`, `b` $\mapsto$ `O`, and so on.

x	t h e q u i c k b r o w n f o x j u m p s o v e r t h e l a z y d o g
y	G U R D H V P X O E B J A S B K W H Z C F B I R E G U R Y N M L Q B T

Even better (this is called the Vigenère cryptosystem): Have a "composed" key $(k_1, \ldots, k_s)$ of s possible shift values in $\{0, \ldots, 25\}$ and use k_1 for the first letter, k_2 for the second, $\ldots$, k_s for the s-th letter, and then start again with k_1. That means: Letter number i is encoded with shift value $k_{((i-1) \bmod s)+1}$. (Warning: This might look quite safe, in particular for larger s, but if s is much smaller than the length of the message there are well-known methods for breaking this system.)

More generally, a *private-key scheme* works as follows. There is a set K of possible "keys". The encryption function is:

$$E\colon X \times K \to Y, (x, k) \mapsto E(x, k);$$

the decryption function is

$$D\colon Y \times K \to X, (y, k) \mapsto D(y, k).$$

(The idea is to have *many* functions available, each one given by a key, and that Eve does not know which key is used.) The *decryption condition* now is:

$$D(E(x, k), k) = x \text{ , for all } x \in X, k \in K.$$

[9] It is customary in discussions of cryptosystems to write plaintext letters in lowercase and cyphertext letters in uppercase.

(Decrypting with the same key that was used for encryption gives back the original plaintext.)

Somehow Alice and Bob have agreed on a key $k \in K$ that only they know. Alice encrypts by calculating $y = E(x, k)$ and sends y to Bob. Bob decrypts $z = D(y, k)$, and then $z = x$, by the decryption condition.

We note that we must have:

Efficiency: It must be possible to calculate $E(x, k)$ and $D(y, k)$ quite fast.

Security: It should be hard, i.e., take a lot of time, for Eve to calculate x, or significant information about x, from y alone, without knowing k.

Example: One-time pad. – Plaintexts, cyphertexts, and keys are bitstrings of length n, i.e., $X = Y = K = \{0, 1\}^n$.

To get started, Alice and Bob secretly generate a *randomly chosen* bitstring k of length n.

To *encrypt* x, Alice computes $E(x, k) = x \oplus_n k$, which means the bitwise XOR of x and k. (We write $a \oplus b$ for $(a + b) \bmod 2$, the XOR of bits a and b.)

Alice sends y to Bob. To *decrypt* y, Bob computes $z = y \oplus_n k$.

If for example $n = 8$, we could have $x = 10010101$, $k = 00101011$. Then $y = E(x, k) = 10111110$ (sent, and read by Eve as well) and $z = D(y, k) = 10010101$.

In general: Since $z_i = y_i \oplus k_i = (x_i \oplus k_i) \oplus k_i = x_i \oplus (k_i \oplus k_i) = x_i \oplus 0 = x_i$ for every bit position i in the strings, we have $z = x$, i.e., Bob gets back the original message x.

Eve listens in and obtains y. What can she learn? It is easy to see that y is a random string no matter what x is. (I.e., every n-bit string y_0 appears as y with the same probability $1/2^n$.) So from seeing y Eve cannot draw any conclusions about x, even if she has unlimited computational power. We call this situation "perfect secrecy".

Some things are inconvenient about this scheme: (1) The necessity to use a secret key, which is only known to Alice and Bob. (2) The fact that the key is as long as the message. (3) The fact that for each message to be sent a new key must be used to have the security guarantee. – There are other symmetric schemes than the one-time pad, which are used in practice and have shorter keys. (In order to arrive at such a system, one starts from a good, but not perfect, cryptosystem for fixed length messages, like AES, "Advanced Encryption Standard". It is assumed, but not proved, that this method is secure against an attacker with limited computing power, even if several messages are encrypted and Eve intercepts several messages. Then a method ("mode") is employed for encrypting several *arbitrarily long sequences* of messages from $X = \{0, 1\}^n$ with one key. **Warning:** It is completely insecure to just

encrypt every single message with the same encryption function! Rather, one has to use "modes" like "cypher block chaining", see the literature.)

Still, (1) is a serious problem, if there are many pairs of partners that want to communicate. For each communication line one would need a new key. Institutions with many communications would have to manage very many keys for pairwise communication. If there are m people with pairwise communication, each one would have to list $m-1$ keys, and there would be $m(m-1)/2$ keys altogether. A lot, if m is a million or so!

1.4.2 Public-key cryptosystems: RSA

Public key cryptosystems provide a way out of problem (1). Here the scheme is a little different. For messages to be sent to Bob a pair $(k_{\text{enc}}, k_{\text{dec}})$ of keys is chosen from a set $K \subseteq K_{\text{enc}} \times K_{\text{dec}}$ of legal key pairs. Everybody knows k_{enc} (Bob's *public key*, it is published e.g. on Bob's webpage or at the bottom of each e-mail he sends), but only Bob knows k_{dec} (his *private key*).

Encryption function: $E\colon X \times K_{\text{enc}} \to Y, (x, k_{\text{enc}}) \mapsto E(x, k_{\text{enc}})$.

Decryption function: $D\colon Y \times K_{\text{dec}} \to X, (y, k_{\text{dec}}) \mapsto D(y, k_{\text{dec}})$.

The decryption condition now reads:

$$D(E(x, k_{\text{enc}}), k_{\text{dec}}) = x, \text{ for all } x \in X, \text{ key pairs } (k_{\text{enc}}, k_{\text{dec}}) \in K.$$

Alice encrypts x as $y = E(x, k_{\text{enc}})$, sends y to Bob, and Bob decrypts $z = D(y, k_{\text{dec}})$. This is x again, by the decryption condition. Eve listens in and she now knows y and also k_{enc} (since everybody does). It should be difficult for her to find out any significant piece of information about x.

We describe the mathematical core of such a system, which was proposed by *Ron Rivest*, *Adi Shamir* und *Leonard Adleman* in 1977 (the "RSA cryptosystem")[10] and is heavily used today, e.g. making the communication with `https://` webpages safe. We prove the decryption condition, and discuss the efficiency of E and D as well as the question of security.

Disclaimer: The "classroom version" of the RSA system as described here cannot and *must not* be used for secure communication. To obtain a secure cryptosystem, the system must be modified as described in the cryptography literature. To be on the safe side: Use a system implemented by experts.

There are two parts, "preprocessing" and "communication".

[10] Wikipedia: *Clifford Cocks*, an English mathematician working for the British intelligence agency Government Communications Headquarters (GCHQ), described an equivalent system in an internal document in 1973, but this was never published.

RSA: The algorithm

Preprocessing: // Done by Bob, or by a certification authority on Bob's request.

(1) Generate two random primes p and q with around $\frac{n}{2}+1$ bits.

(2) Let $N = p \cdot q$.

// *Example*: p and q have 512 binary digits; N has at least 1023 digits.
In general we must have $N \geq 2^{n+1}$, so that $x < N$ for the message x.
Running time in general: $O(n^4)$.

(3) Let $\varphi(N) = (p-1) \cdot (q-1)$.

// "Euler's φ function". This is the number of integers $a \in \{1, \ldots, N-1\}$ that are relatively prime to N.

(4) Choose $e < \varphi(N)$ such that $\gcd(e, \varphi(N)) = 1$.

// Choose randomly, check by Euclidean Algorithm ($O(n^2)$ time).

(5) Calculate $d < \varphi(N)$ such that $e \cdot d \bmod \varphi(N) = 1$.

// Can be done by the Extended Euclidean Algorithm ($O(n^2)$ time).

Bob's **"public key"**: $k_{\text{enc}} = (N, e)$. // Posted on Bob's web page.

Bob's **"private key"**: $k_{\text{dec}} = (N, d)$.
// Never mind that the N part is public anyway. d is secret.

Communication:

Encryption function: $E(x, (N, e)) = x^e \bmod N$, for $x \in X = [N]$.

Decryption function: $D(y, (N, d)) = y^d \bmod N$, for $y \in Y = [N]$.

Assume Alice has a message (or plaintext) $x \in [N] = \{0, \ldots, N-1\}$. She computes

$$y := E(x, (N, e)) = x^e \bmod N$$

and sends y to Bob. Bob computes

$$z := D(y, (N, d)) = y^d \bmod N.$$

For both computations we can use fast exponentiation; the running time is $O(n^3)$ for both. Note that for $n \approx 2000$ or even $n \approx 4000$ this is no problem for today's computers.

RSA *Example*:

Preprocessing: 4-bit primes. Choose $p = 11$, $q = 13$, $N = 143$.

Then $\varphi(N) = (11 - 1)(13 - 1) = 10 \cdot 12 = 120$.

Choose $e = 77$
(from $\{1, \ldots, 119\}$ excepting the multiples of 2, 3, 5, the prime factors of 120).

Calculate $d = 77^{-1} \bmod 120 = 53$ using the extended Euclidean Algorithm
(check: $77 \cdot 53 = 4081 = 34 \cdot 120 + 1$).

Public key (published by Bob): $k_{\text{enc}} = (N, e) = (143, 77)$.

Private key: $k_{\text{dec}} = (143, 53)$. The number 53 is kept secret by Bob.

$p = 11$, $q = 13$, $\varphi(N) = 120$ are *discarded*.

Sending messages:

(1) Plaintext: $x = 33$.

Alice uses fast exponentiation to calculate (see page 6):

$33^{2^i} \bmod 143$	$\lfloor 77/2^i \rfloor$		
33	77		33
$33^2 \bmod 143 = 88$	38	strike out (38 is even)	–
$88^2 \bmod 143 = 22$	19		22
$22^2 \bmod 143 = 55$	9		55
$55^2 \bmod 143 = 22$	4	strike out (4 is even)	–
$22^2 \bmod 143 = 55$	2	strike out (2 is even)	–
$55^2 \bmod 143 = 22$	1		22
Product:			$(33 \cdot 22 \cdot 55 \cdot 22) \bmod 143 = 11$.

She sends the result "11" to Bob.

Bob uses fast exponentiation in a similar way to calculate:

$11^{2^i} \bmod 143$	$\lfloor 53/2^i \rfloor$		
11	53		11
$11^2 \bmod 143 = 121$	26	strike out (26 is even)	–
$121^2 \bmod 143 = 55$	13		55
$55^2 \bmod 143 = 22$	6	strike out (6 is even)	–
$22^2 \bmod 143 = 55$	3		55
$55^2 \bmod 143 = 22$	1		22
Product:			$(11 \cdot 55 \cdot 55 \cdot 22) \bmod 143 = 33$.

The result is $z = 11^{53} \bmod 143 = 33$, which is x again, correct!

(2) Plaintext: $x = 93$. Calculate similarly as above (try it):

Alice calculates $y = 93^{77} \bmod 143 = 58$ and sends "58" to Bob.

Bob calculates $z = 58^{53} \bmod 143 = 93$, correct!

RSA: Proof of correctness

We must check the decryption condition.

Claim: $z = ((x^e \bmod N)^d) \bmod N = x$. Equivalently:

Theorem 1.4.1
For all $x \in [N] = \{0, 1, \ldots, N-1\}$: $x^{ed} \bmod N = x$.

Proof[11]:

Since $0 \leq x < N$, the assertion means the same as $x^{ed} \bmod N = x \bmod N$. This is equivalent to saying that the difference $x^{ed} - x$ is divisible by N.

For this it is sufficient to show that both p and q divide $x^{ed} - x$.
(Recall a basic fact about prime numbers: If the distinct primes p and q both divide some number a, then their product $p \cdot q$ divides a as well.)

We only show that p divides $x^{ed} - x$. (The argument for q is the same.)

If p divides x, then clearly p divides $x^{ed} - x$, and we are done.

Assume from here on that p does not divide x.

Then by Fermat's little theorem we have $x^{p-1} \bmod p = 1$.

Since $ed \bmod \varphi(N) = 1$ by choice of d, we can write
$ed = k \cdot \varphi(N) + 1 = k(p-1)(q-1) + 1$ for some integer $k \geq 1$.

Then (calculating modulo p):

$$\begin{aligned} x^{ed} &\equiv x^{k(p-1)(q-1)} \cdot x \\ &\equiv (x^{p-1})^{k(q-1)} \cdot x \\ &\equiv (x^{p-1} \bmod p)^{k(q-1)} \cdot x \\ &\equiv 1^{k(q-1)} \cdot x \\ &= x, \end{aligned}$$

which means that p divides $x^{ed} - x$, and we are done. □

[11] The proof follows Dasgupta, Papadimitriou, Vazirani, *Algorithms*, page 34, but it replaces the Chinese remainder theorem by a more elementary statement.

RSA: Efficiency and security

Note that RSA can be implemented and used only since certain algorithms for numbers (finding primes, extended Euclidean algorithm for finding inverses, fast exponentiation for encryption and decryption) can be carried out fast for numbers of a few thousand bits.

Security: We can only give remarks. (There is no formal proof that the RSA system is secure in any precise sense.) The security hinges on the fact that **factoring numbers into their prime factors is difficult**, as far as we know today. More precisely: No algorithms with running time $O(n^k)$ for any constant k is known that finds factors p and q from N as in the RSA system. No method is known to factor arbitrary 1024-bit numbers in reasonable time.[12]

This means that the obvious attack from Eve, namely finding p and q by factoring N, does not work (as of the year 2021), as long as N is large enough. It would be enough for Eve to find $\varphi(N)$ or find $d = e^{-1} \bmod \varphi(N)$ by other means than factoring. It turns out, however, that this is not much easier than factoring N, in the sense that if she had $\varphi(N)$ she could easily find the factors, and the same when she had d. Of course, there could be totally different ways of finding x from y and (e, N), which do not imply factoring N, *but no such method is known.*

Remark: RSA encryption and decryption is relatively slow. It is typically not used for encrypting a whole communication between parties. One of its main uses in communication over the internet (think of `https://`), is to establish a "session key" to be used between two participants in a (much faster) symmetric scheme.

Warning: It is **never safe** to encrypt a long text naively, by splitting it into blocks of length $\leq n$ and simply encrypting each single block individually by the simple version of the RSA system.

Among other enhancements, the following trick is helpful and is, in principle, used in practice: Alice splits a long message x into segments $x_1, \ldots, x_m$ of length $n/2$ each. For each block x_i, she chooses a new random string r_i with $n/2$ bits. Then $x'_i = x_i \circ r_i$ (the concatenation) is encoded with Bob's public key, for each i, resulting in $y'_1, \ldots, y'_m$. This sequence is sent to Bob, who decodes to get $z'_1, \ldots, z'_m$. He then drops the second half of each z'_i to obtain $z_1, \ldots, z_m$, which equals $x_1, \ldots, x_m$. Apart from other aspects, with this approach a central problem of deterministic block-wise encryption disappears: even if $x_i = x_j$, we will have $y'_i \neq y'_j$ with high probability.

Warning: With an enhancement like this, RSA as described here may be "secure" against adversaries that are only listening in. But it is vulnerable (like an open barn

[12]The algorithm one would want to use depends on the size of the numbers. For names of available algorithms see, e.g., `https://stackoverflow.com/questions/2267146/what-is-the-fastest-factorization-algorithm`. The asymptotically fastest algorithm, the "general number field sieve", has a running time bound of $\exp(c \cdot n^{1/3}(\log n)^{2/3})$, for a constant $c < 2$, for large n. Here $\exp(x)$ means e^x.

door) to the "man-in-the-middle attack". The "man in the middle", Oscar, is an adversary with stronger abilities than our eavesdropper Eve. He can intercept messages and replace them by messages of his own choice. So he would listen in, intercept an encrypted message $y = E(x, (e, N))$ from Alice to Bob, replace it by $y' = E(x', (e, N))$ for a plaintext x' of his own choice and make Bob believe the message y' is from Alice (inserting a forged sender address, which is not very hard). In the worst case, Oscar can even include in the message a public key (e', N') of his own choosing and induce Bob to respond using this key for encryption. In this case he could even read the answer! In order to avoid these problems, one needs other cryptographic primitives like *authentication*, where the identity of the sender of a message can be checked.

Warning: If you ever implement your own cryptosystem for serious use, have it checked by an expert, or get training to become an expert yourself before.