1. Let $a$ and $b$ be positive integers. The following Extended Euclid algorithm calculates three integers $d$, $x$ and $y$ such that $d$ is the greatest common divisor of $a$ and $b$ and

$$d = ax + by.$$ 

\begin{verbatim}
Extended-Euclid(int a, int b)
1. if b = 0 then
2. return (a,1,0);
3. (d',x',y')<--Extended-Euclid(b, a mod b);
4. (d,x,y)<-- (d', y', x'-floor(a/b)*y');
5. return (d,x,y);
\end{verbatim}

For example, if $a = 99$ and $b = 78$, then $d = 3$, $x = -11$, $y = 14$.

The algorithm can be used to calculate multiplicative inverses in modular arithmetics. If $a$ and $n$ are two positive integers such that their greatest common divisor is 1, then there is an integer $b$ such that $a * b \mod n = 1$. Just give $a$ and $n$ to the algorithm and the output $x$ is the multiplicative inverse. For example, if $a = 19$ and $n = 31$, call the algorithm with $Extended-Euclid(19,31)$ and get $d = 1$, $x = -13$, $y = 8$. Here, $x = -13$ is the multiplicative inverse of 19 modulo 31. In modular arithmetics modulo 31, $-13$ is the same as 18, and $19 \times 18 = 342$ and 342 modulo 31 is 1.

Implement the Extended Euclid algorithm and test it with the above numbers as well as with some other numbers.

2. (Compulsory) The following algorithm divides two polynomials and returns the result plus the remainder. The coefficients of the polynomials are modular integers and all the polynomial and coefficient operations take place in modular arithmetics. Let

$$P(X) = a_nX^n + a_{n-1}X^{n-1} + \cdots + a_1X + a_0,$$

$$Q(X) = b_mX^m + b_{m-1}X^{m-1} + \cdots + b_1X + b_0,$$

and assume that $n \geq m$ and that all the coefficients are integers modulo some prime $p$. The following algorithm divides $P$ by $Q$ and returns the result plus the remainder.
\[ \text{i) } R \leftarrow P \]
\[ \text{ii) for } i = n - m \text{ downto } 0 \text{ do} \]
\[ \text{iii) if } \deg(R) = m + i \text{ then} \]
\[ q_i \leftarrow \text{lc}(R) \cdot (b_m)^{-1} \]
\[ R \leftarrow R - q_iX^iQ \]
\[ \text{else} \]
\[ q_i \leftarrow 0 \]
\[ \text{iv) return } S = q_{n-m}X^{n-m} + q_{n-m-1}X^{n-m-1} + \cdots + q_1X + q_0 \text{ and } R. \]

Notice that \( \text{lc}(R) \) means the leading coefficient of \( R \) and \((b_m)^{-1}\) means the multiplicative inverse of \( b_m \) modulo \( p \). For example, if \( p = 3 \) and
\[ P(X) = 2X^6 + 2X^5 + X^3 + x^2 + 2X + 2, \]
\[ Q(X) = X^4 + X^3 + 2X^2 + X + 2, \]

then the result of the division \( P/Q \) is \( 2X^2 + 2 \) and the remainder is \( 2X^2 + 1 \).

Write a function which gets two polynomials \( P, Q \) and the modulus \( p \) (prime) as parameters and returns the division \( P/Q \) result plus the remainder.

3. The greatest common divisor of the polynomials \( P(X) \) and \( Q(X) \) is a polynomial \( R(X) \) such that \( 1 \leq \deg(R) < \deg(P) \), \( 1 \leq \deg(R) < \deg(Q) \), and \( R(X) \) divides both \( P(X) \) and \( Q(X) \). The euclidean algorithm can be used to find the greatest common divisor (gcd) for polynomials. For example, let
\[ P(X) = 2X^6 + 2X^5 + X^3 + X^2 + 2X + 2, \]
\[ Q(X) = X^4 + X^3 + 2X^2 + X + 2 \]

be two polynomials over mod 3 integers. Then the euclidean algorithm works as follows:

| \( a \) | \( b \) | \( a \mod b \) |
|\( 2X^6 + 2X^5 + X^3 + X^2 + 2X + 2 \) | \( X^4 + X^3 + 2X^2 + X + 2 \) | \( 2X^2 + 1 \) |
| \( x^4 + X^3 + 2X^2 + X + 2 \) | \( 2X^2 + 1 \) | \( 2X + 2 \) |
| \( 2X^2 + 1 \) | \( 2X + 2 \) | \( 0 \) |
| \( 2X + 2 \) | \( 0 \) | \( \) |

The only problem is that the gcd is not unique, if the coefficients are integers modulo \( p \) (\( p \) prime). It is possible to multiply the result with any constant modulo \( p \). Thus \( 2(2X + 2) = X + 1 \) is also a gcd of \( P \) and \( Q \).

Write the function
\[ \text{termNode* polGCD(termNode* P, termNode* Q, int p)} \]
that calculates the greatest common divisor of $P$ and $Q$ using Euclidean algorithm, when the coefficients are integers modulo $p$, $p$ prime. (It is not necessary the check that $p$ is prime.)

4. Write a function

\[
genPolynomials(int n, int p)
\]

which generates all the polynomials of degree $n$ or less, when the coefficients are integers modulo $p$. For example, if $n = 1$ and $p = 3$ then the polynomials are

\[
\begin{array}{c|c|c}
0 & 1 & 2 \\
X & X + 1 & X + 2 \\
2X & 2X + 1 & 2X + 2 \\
\end{array}
\]

You can decide in what form the function returns the polynomials.

5. The following algorithm tests, if a polynomial over modular integers is irreducible:

**Input:** A polynomial $P$ over integers modulo $p$ and of degree $n$.

**Output:** "irreducible" or reducible"

i) Compute $Q(X) = X^{p^n} \text{ rem } P(X)$ (i.e. remainder when $X^{p^n}$ is divided by $P(X)$);
   if $Q(X) \neq X$, then return "reducible";

ii) for all prime divisors $t$ of $n$ do

iii) compute $R(X) = X^{p^{n/t}} \text{ rem } P(X)$;
    if $\gcd(R(X) - X, P(X)) \neq \text{constant}$ then return "reducible".

iv) return "irreducible".

Write a function

\[
\text{int irreducibleP(termNode* P, int p, int factornr, int * factors)}
\]

that returns 1, if $P$ is irreducible and 0 otherwise. The parameter $p$ is the modulus of the coefficients, factornr is the number of prime factors of $n$, factors is a pointer to the factors.

Examples of irreducible polynomials are $X^3 + X^2 + 1$ and $X^3 + X + 1$, when coefficients are integers modulo 2. In the previous exercises there are examples of reducible polynomials.
6. The following Ben-Or’s algorithm generates irreducible polynomials:

**Input:** A prime power \( q \) and \( n \in N \).
**Output:** Random monic (leading constant is 1) irreducible polynomial of degree \( n \) and coefficients are modulo \( q \) numbers.

i) randomly choose a monic polynomial \( P \) of degree \( n \) with coefficients modulo \( q \) integers.

ii) for \( i = 1, \ldots, \lfloor n/2 \rfloor \) do

   \[ g_i \leftarrow \gcd(X^{q^i} - X, P); \text{ if } g_i \neq 1 \text{ then goto 1}; \]

iii) return \( P \).

Write a function

```c
termNode* genIrredicibleP(int q, int n)
```

that returns an irreducible polynomial. It is not necessary to check that \( q \) is a prime power (i.e. of the form \( p^k \), where \( p \) is a prime and \( k \) a positive integer).

7. Write a function

```c
void writeAddMulTables(int p, termNode* P)
```

which generates all the polynomials of degree less than \( \deg(P) \) and with mod \( p \) coefficients (\( p \) prime) and constructs the addition and multiplication tables as follows. Take every pair \( P_1, P_2 \) of the generated polynomials and calculate \( P_1 + P_2 \) and \( P_1 \cdot P_2 \) mod \( P \). The parameter \( P \) should be an irreducible polynomial with mod \( p \) coefficients. See the examples besides the exercise at the web page. (As a matter of fact, if you have done this exercise, you have constructed finite fields which are needed in many applications of cryptography and coding theory.)

8. (In the following bit operation exercises we use int types `uint8_t`, `uint16_t`, `uint32_t`. These types can be found in the library stdint.) Write a function that prints an unsigned 32-bit integer in binary form. The display format should include a separator at byte boundaries.

Example:

```c
class print_bin(uint32_t value) {
  if (value <= 0xff)
    printf("00000000.00000000.00000000.00000000 0x%x\n", value);
  else if (value <= 0xffff)
    printf("00000000.00000000.00000000.00000000 0x%x\n", value);
  else if (value <= 0xffffffff)
    printf("00000000.00000000.00000000.00000000 0x%x\n", value);
  else
    printf("00000000.00000000.00000000.00000000 0x%x\n", value);

print_bin(0) => 00000000.00000000.00000000.00000000
print_bin(1) => 00000000.00000000.00000000.00000001
print_bin(2) => 00000000.00000000.00000000.00000010
print_bin(43) => 00000000.00000000.00000000.00101011
print_bin(123456789) => 00000111.01011011.11001101.00010101
```

Note: You can use this function to track down problems you may encounter while working with bit operations.
9. Write a function

```c
uint16_t make_16bit(uint8_t least_significant, uint8_t most_significant);
```

that combines the given bytes into a 16-bit integer and returns it as the result.
Note: In practice, the first argument contains 8 least significant bits of an unsigned 16-bit integer and the second contains the 8 most significant bits.

Example:

```c
print_bin(1) #=> 00000000.00000000.00000000.00000001
print_bin(7) #=> 00000000.00000000.00000000.00000111
print_bin(make_16bit(7, 1)) #=> 00000000.00000000.00000001.00000111
```

10. Write a function

```c
uint32_t make_32bit(uint8_t byte1, uint8_t byte2, uint8_t byte3, uint8_t byte4);
```

that combines four bytes into a 32-bit integer, similar to the previous function.

Example:

```c
print_bin(make_32bit(1, 2, 3, 4)) #=> 00000100.00000011.00000010.00000001
```