Basic Cryptography

Computer Algebra for Cryptography (B-KUL-H0E74A)

1 RSA Cryptosystem

Implement the basic RSA cryptosystem as described in the lectures. To verify that you implemented the correct API, you can generate a key pair, give the public key to a fellow student and ask him to encrypt a message for you. If everything works properly, you should be able to decrypt the ciphertext.

Exercise 1. Implement the RSA cryptosystem, i.e. implement functions for

- Generating an RSA public and private key: `RSAKeyGen(b::RngIntElt) -> N::RngIntElt, e::RngIntElt, d::RngIntElt` that takes in a bit length b and returns the public key N and e, and the private key d. The bit length of N should be b, i.e. $\lceil \log_2(N) \rceil = b$. If you ask for a random prime in Magma, you should remember to set the optional variable `Proof` to false, otherwise Magma will try to prove that the number is really prime. For large primes, this takes a very long time.
- Encrypting a message: `RSAEncrypt(m::RngIntElt, N::RngIntElt, e::RngIntElt) -> c::RngIntElt` that takes in an integer message m, and a public key N, e. Note that the result should be an integer, and not an element of the ring $\mathbb{Z}_N$.
- Decrypting a ciphertext: `RSADecrypt(c::RngIntElt, N::RngIntElt, d::RngIntElt) -> m::RngIntElt` , given a ciphertext c and private key (N, d), returns the decrypted message m.
- Verify that when you encrypt a message m and then decrypt is, the result is the same.
- Implement a version of the decryption function using the Chinese Remainder Theorem. Magma has a built in function called `CRT(X::SeqEnum[RngIntElt], M::SeqEnum[RngIntElt]) -> RngIntElt`. Why can you only do this for decryption and not for encryption?

Instead of using the built-in exponentiation of Magma, you can also write your own exponentiation routine. You should compare the speed of your implementation vs. the built-in one.

Exercise 2.

- Implement a function `SquareMult(a::FldFinElt, n::RngInt) -> b::FldFinElt` that computes the power $b = a^n$ with a a finite field element using the square and multiply approach. Verify your result with the Magma function `^`.

2 The Hill Cipher

The Hill cipher is an old (and very unsafe) cipher that is based on linear algebra over $\mathbb{Z}_{26}$. With every letter of the alphabet we associate one element of the ring $\mathbb{Z}_{26}$: $A = 0$, $B = 1, \ldots, Z = 25$. To encrypt a message, we first convert every character of the string with the aforementioned bijection. We then break the message in blocks of length k, where k is some (small) integer. The secret key is a matrix $A \in \mathbb{Z}_{26}^{k\times k}$, and encryption is done by considering the blocks of length k as vectors, and (left-)multiplying them with A. The resulting vectors can be concatenated as blocks of length k again, and transformed into capital letters. Decrypting happens exactly the same as encrypting but we now use A^{-1}.

Exercise 3.

- Write a function `StringToHill(s::MonStgElt) -> SeqEnum[RngIntResElt]` in Magma that tests if an input string s consists only of capital letters, and raises an error if this is not the case (you can use the command `error expression, ..., expression;` to this end). If the input is well formed it returns a sequence of elements in $\mathbb{Z}_{26}$ as mentioned above. You can use the Magma functions `Regexp(R, S) : MonStgElt, MonStgElt -> BoolElt, MonStgElt, [ MonStgElt ]` to test that the input consists of all capitals (using the regexp `^[A-Z]+$`) and `Eltseq(s) :  MonStgElt -> [ MonStgElt ]`).

- Write a function `HillToString(a::SeqEnum[RngIntResElt]) -> MonStgElt` which takes in a sequence of elements in $\mathbb{Z}_{26}$ and returns the corresponding string of capital letters.

- Write a function `HillKeyGen(s::MonStgElt) -> Mtrx` that, given as input a string of capital letters of length k^2, tests if the input is well formed (raise an error if not the case) and if it is, outputs a matrix $A \in \mathbb{Z}_{26}^{k\times k}$ where the first row corresponds to the first k letters of the string, the second row to the letters $k+1$ through $2k$, etc. For example, `HillKeyGen("ABXZ")` should result in
$$\begin{bmatrix} 0 & 1 \\ 23 & 25 \end{bmatrix}.$$
You can create a zero matrix using `ZeroMatrix(R::Rng, m::RngIntElt, n::RngIntElt) -> Mtrx`.

- Write a function `HillEncrypt(s::String, A::Mtrx) -> String` in Magma that, given as input a message (a string of capital letters of arbitrary length) and a matrix A representing the private key, outputs the encrypted version of this message with the key A. As an example, given the matrix corresponding to the private key `ABXZ`, encrypting `COMPUTERALGEBRA` would boil down to first converting `COMPUTERALGEBRA` to

```
[ 2, 14, 12, 15, 20, 19, 4, 17, 0, 11, 6, 4, 1, 17, 0 ].
```

Next, we split up this sequence in blocks of length k and interpret them as vectors. In case the message length is not a multiple of k, just pad the end with zeros. Next, multiply these vectors from the left with the private key A, concatenate the resulting vectors into a sequence and transform them back to letters from the alphabet. Using the key of the previous question, the result would be `OGPBTZRXLPEERGAA` (note the one extra character due to `COMPUTERALGEBRA` having an odd number of letters).

- Write a function `HillDecrypt(s::MonStgElt, A::Mtrx) -> String` decrypting a ciphertext (which is a string of capital letters) given a key A.
- Test all your functions a couple of times, e.g. try commands of the form

```
A := HillKeyGen("ABXZ");
HillDecrypt(HillEncrypt("COMPUTERALGEBRA",A),A^-1);
```

for various strings representing both the private key and the message that you want to encrypt. See if the result is the same as the message (possibly with some extra A's at the end). What happens if you use

`A := HillKeyGen("HILLCIPHERFORCRYPTOGRAPHY");`?

What causes this and how could you solve this problem?