Problem Set: Bignums and RSA encryption

Stuart M. Shieber

February 5, 2023

Cryptography is the science of methods for storing or transmitting messages securely and privately.

Cryptographic systems typically use *keys* for encryption and decryption. An encryption key is used to convert the original message (the plaintext) to coded form (the ciphertext). A corresponding decryption key is used to convert the ciphertext back to the original plaintext.

In traditional cryptographic systems, the same key is used for both encryption and decryption, which must be kept secret. Two parties can exchange coded messages only if they share the secret key. Since anyone who learned that key would be able to decode the messages, keys must be carefully guarded and transmitted only under tight security, for example, couriers handcuffed to locked, tamper-resistant briefcases!

Figure 1: Whitfield Diffie (1944–) and Martin Hellman (1948–), co-inventors of public-key cryptography, for which they received the Turing Award in 2015.

In 1976, Diffie and Hellman initiated a new era in cryptography with their discovery of a new approach: public-key cryptography. In this approach, the encryption and decryption keys are different from each other. Knowing the encryption key cannot help you find the decryption key. Thus, you can publish your encryption key publicly – on the web, say – and anyone who wants to send you a secret message can use it to encode a message to send to you. You do not have to worry about key security at all, for even if everyone in the world knew your encryption key, no one could decrypt messages sent to you without knowing your decryption key, which you keep private to yourself. You used public-key encryption when you set up your CS51 git repositories: the command `ssh-keygen` generated a public encryption key and private decryption key for you. You uploaded the public key and (hopefully) kept the private key to yourself.

The best known public-key cryptosystem is due to computer scientists Rivest, Shamir, and Adelman, and is known by their initials, RSA. The security of your web browsing probably depends on RSA encryption. The system relies on the fact that there are fast algorithms for exponentiation and for testing prime numbers, but no known fast algorithms for factoring extremely large numbers. In this problem set you will complete an implementation of a version of the RSA system. (If you're interested in some of the mathematics behind RSA, see Section 3. However, an understanding of that material is not needed to complete the problem set.)

Crucially, RSA requires manipulation of very large integers, much

larger than can be stored, for instance, as an OCaml `int` value. OCaml's `int` type has a size of 63 bits, and therefore can represent integers between -2^{62} and $2^{62}-1$. These limits are available as OCaml constants `min_int` and `max_int`:

```
# min_int, max_int ;;
- : int * int = (-4611686018427387904, 4611686018427387903)
```

The `int` type can then represent integers with up to 18 or so digits, that is, integers in the quintillions, but RSA needs integers with hundreds of digits.

Computer representations for arbitrary size integers are traditionally referred to as BIGNUMS. In this assignment, you will be implementing bignums, along with several operations on bignums, including addition and multiplication. We provide code that will use your bignum implementation to implement the RSA cryptosystem. Once you complete your bignum implementation, you'll be able to encrypt and decrypt messages using this public-key cryptosystem, and discover a hidden message that we've provided encoded in this way.

(This assignment was adapted from CS312 at Cornell.)

1 Big numbers

To handle arbitrarily large integers (as large as the computer's memory allows), we define a new algebraic data type to represent bignums, along with the standard arithmetic operations `plus`, `times`, comparison operators, and so on.

In this bignum implementation, an integer n will be represented as a list of `int` values, namely the coefficients of the expansion of n in some base. For example, suppose the base is 1000. Then the number 123456789 can be written as:

$$(123 \cdot 1000^2) + (456 \cdot 1000^1) + (789 \cdot 1000^0) \quad ,$$

which we will represent by the list `[123; 456; 789]`. Notice that the least significant coefficient (789) appears last in the list. As another example, the number 12000000000000 is represented by the list `[12; 0; 0; 0; 0]`.

The base used by your bignum implementation is defined at the top of the file `bignum.ml`:[1]

```
let cBASE = 1000 ;;
```

To make it easier to implement some of the functions below, you may want to use a base of 1000 while debugging and testing. However, you'll want to make sure that the code works for any value of `cBASE`, always referring to that variable rather than hard-coding a value of

[1] We name the base using this distinctive naming convention, with initial 'c' (for constant) and upper-case mnemonic to emphasize that it is a global constant. Such global constants should be rare, and thus typographically distinctive.

1000, so that if `cBASE` is changed to a different value, your code should still work.[2] This is good programming practice in general, and makes it easier to modify code later.

Here is the actual type definition for bignums:

```
# type bignum = {neg : bool; coeffs : int list} ;;
type bignum = { neg : bool; coeffs : int list; }
```

The `neg` field specifies whether the number is positive (if `neg` is false) or negative (if `neg` is true). The `coeffs` field is the list of coefficients in some base, where each coefficient is between 0 and `cBASE - 1`, inclusive. Assuming a base of 1000, to represent the integer 123456789, we would use:

```
# {neg = false; coeffs = [123; 456; 789]} ;;
- : bignum = {neg = false; coeffs = [123; 456; 789]}
```

and to represent −9999 we would use:

```
# {neg = true; coeffs = [9; 999]} ;;
- : bignum = {neg = true; coeffs = [9; 999]}
```

In other words, the record

`{neg = false, coeffs = [`a_n`; `a_{n-1}`; `a_{n-2}`; ...; `a_0`]}`

represents the integer

$$a_n \cdot base^n + a_{n-1} \cdot base^{n-1} + a_{n-2} \cdot base^{n-2} + \cdots + a_0$$

An empty list thus represents 0. This defines the correspondance between `bignum` and the integers.

1.1 Representation invariants

The implementation of bignums will be simpler if we impose certain representation invariants. A REPRESENTATION INVARIANT is a property of values in your representation that you enforce and that you can thus assume is true when writing functions to handle these values. If the representation invariant is violated, the value is not a valid value for the type. Any function you write that produces a value of the type (for example, `from_int`) should produce a value satisfying the representation invariant.

The invariants for the bignum representation are as follows:

- Zero will always be represented as `{neg = false; coeffs = []}`, and never as `{neg = true; coeffs = []}`.
- There will be no leading zeroes on the list of coefficients. The value `{neg = false; coeffs = [0; 0; 125]}` violates this invariant.

[2] For reasons of simplicity in implementing functions to move between bignums and strings, we'll assume that `cBASE` is a power of 10. Similarly, implementing certain operations on bignums here is simplified by requiring that `cBASE * cBASE < int_max`, that is `cBASE` can be safely squared without overflowing the integer representation. You can assume that both of these conditions hold as invariants.

- Coefficients are never negative and are always strictly less than `cBASE`.

Functions that consume bignums may assume that they satisfy the invariant. We will not test your code using bignums that violate the invariant. Functions that return bignums should preserve these invariants. For example, your functions should never return a bignum representing zero with the `neg` flag set to `true` or a bignum with a coefficients list with leading zeros.

1.2 Using your solution

Using the functions `from_string` and `to_string`, you will be able to test your functions by converting between the bignum representation and a string representation. (These functions further assume that `cBASE` is a power of 10.) Here is a sample interaction with a completed implementation of bignums, in which we multiply 123456789 by 987654321:

```
# let answer = times (from_string "123456789")
#                    (from_string "987654321") ;;
val answer : bignum = {neg = false; coeffs = [121; 932; 631; 112;
  635; 269]}
# to_string answer ;;
- : string = "121932631112635269"
```

1.3 What you need to do

You will implement several functions in `bignum.ml` that operate on bignums. As usual, feel free to change the header of the function definition, for instance by adding a `rec` keyword, if you think it helpful, but do not alter the types of any functions, as we will be unit testing assuming those type signatures.

Problem 1

Implement the function `negate : bignum -> bignum`, which gives the negation of a bignum (the bignum times -1).

Problem 2

Implement the functions `equal : bignum -> bignum -> bool`, `less : bignum -> bignum -> bool`, and `greater : bignum -> bignum -> bool` that compare two numbers `b1` and `b2` and return a boolean indicating whether `b1` is equal to, less than, or greater than `b2`, respectively. You can assume that the arguments satisfy the representation invariants.

Problem 3

Implement conversion functions `to_int : bignum -> int option` and `from_int : int -> bignum` between integers and bignums. The function `to_int : bignum -> int option` should return `None` if the number is too large to fit in an OCaml integer.

You'll want to be careful when checking whether values fit within OCaml integers. In particular, you shouldn't assume that `max_int` is the negative of `min_int`; in fact, it isn't, as seen above. Instead, you may use the fact that `max_int = abs(min_int + 1)`, though by careful design choices you can avoid even that assumption.

Problem 4

We have provided you with a function, `plus_pos : bignum -> bignum -> bignum`, which adds two bignums and provides the result as a bignum. However, it has a limitation: this function only works if the resulting sum is positive. (This might not be the case, for example if `b2` is negative and larger in absolute value than `b1`.) Write the function `plus : bignum -> bignum -> bignum`, which can add arbitrary bignums, without this limitation. It should call `plus_pos` and shouldn't be too complex. *Hints:* How can you use the functions you've written so far to check, without doing the addition, whether the resulting sum will be negative? If the sum will be negative, can you adjust the numbers to find a different way of generating the sum using only additions that obey the limitation? And a hint on your unit tests for this problem: you'll definitely want to test a case where the result comes out negative.

Problem 5

Implement the function `times : bignum -> bignum -> bignum`, which multiplies two bignums. Use the traditional algorithm you learned in grade school (as in Figure 2), but remember that we are representing numbers in base 1000 (say), not 10. The main goal is correctness, so keep your code as simple as possible. Make sure your code works with positive numbers, negative numbers, and zero. Assume that the arguments satisfy the invariant. *Hint:* You may want to write a helper function that multiplies a bignum by a single `int` (which might be one coefficient of a bignum).

						5	4	3
				×		2	2	4
					2	1	7	2
		+		1	0	8	6	*0*
+			1	0	8	6	*0*	*0*
=			1	2	1	6	3	2

Figure 2: Multiplication of 543 and 224 using the grade school algorithm. First, you multiply the first number (543 in this example) by each of the digits of the second number, from least significant to most significant (4, then 2, then 2), adding an increasing number of zeros at the end (shown in italics), no zeros for the least significant digit (resulting in 2172), one for the next (1086 plus one zero yielding 10860), two for the next, and so forth. Then the partial products 2172, 10860, and 108600 are summed to generate the final result 121632.

1.4 Using bignums to implement RSA

We've provided an implementation of the RSA cryptosystem in the file `rsa.ml`. It uses the module `Bignum`, that is, the bignum implementation in `bignum.ml` that you've just completed.

In the file `rsa_puzzle.ml`, we've placed some keys and a ciphertext with a secret message. If your implementation of bignums is correct, you should be able to compile and run the file:

```
% ocamlbuild rsa_puzzle.byte

% ./rsa_puzzle.byte
```

to reveal the secret message!

2 Challenge problem: Multiply faster

As on previous problem sets and several in the future, we are providing an additional problem or two for those who would like an extra challenge. These problems are for your karmic edification only, and will not affect your grade. We encourage you to attempt this problem only once you have done your best work on the rest of the problem set.

The multiplication algorithm you implemented in Section 1.3 will work just fine for most integers of reasonable sizes, including the ones needed by the RSA implementation. However, some exceedingly smart people have devised algorithms which, on very large numbers (think thousands of digits), are considerably faster than the multiplication algorithm you learned in grade school.

Problem 6

Challenge See if you can implement such an algorithm in `times_faster`. You may want to start by looking up the Karatsuba algorithm. This algorithm recursively multiplies smaller and smaller numbers. Note that, when the numbers become small enough (2-4 digits), you can and probably should simply call the `times` function you implemented earlier. However, don't just call this function on any numbers you are given; that's not any faster!

3 More background: How the RSA cryptosystem works

This section is intended for those who are interested in more mathematical details on how the RSA system uses bignum calculations to achieve public-key encryption and decryption. Nothing in this section is needed to complete the pset.

We want to encrypt messages that are strings of characters, but the RSA system does not work with characters, but with integers. To encrypt a piece of text, we first convert it to a number (a bignum in fact) by combining the ASCII codes of the characters; we can then encrypt the resulting number.

To generate public and private keys in the RSA system, you select two very large prime numbers p and q. (Recall that a prime number is a positive integer greater than 1 with no divisors other than itself and 1.) How to find large primes is a subject in itself, but beyond the scope of these notes.

You then compute numbers n and m:

$$n = p\,q$$
$$m = (p-1)(q-1)$$

It turns out that

- With very few exceptions, for almost all numbers $e < n$, $e^m \pmod{n} = 1$.
- No one knows how to compute m, p, or q efficiently, even knowing n.

(Notation: We write $a \pmod{n}$ for the remainder obtained when dividing a by b using ordinary integer division. We use the notation $[a = b] \pmod{n}$ to mean that $a \pmod{n} = b \pmod{n}$. Equivalently, $[a = b] \pmod{n}$ if $a - b$ is divisible by n. For example, $[17 = 32] \pmod{5}$.)

Now you pick a number $e < m$ relatively prime to m; that is, such that e and m have no factors in common except 1. The significance of relative primality is that e is relatively prime to m if and only if e is invertible mod m, that is, if and only if there exists a d such that $[d\,e = 1] \pmod{m}$. Moreover, it is possible to compute d from e and m

using Euclid's algorithm, described below. Your public key, which you can advertise to the world, is the pair (n, e). Your private key is (n, d).

Anyone who wants to send you a secret message s (represented by an integer remember) encrypts it by computing $E(s)$, where

$$E(s) = s^e \pmod{n}$$

That is, if the plaintext is represented by the number s which is less than n, the ciphertext $E(s)$ is obtained by raising s to the power e, then taking the remainder modulo n.

The decryption process is exactly the same, except that d is used instead of e:

$$D(s) = s^d \pmod{n}$$

The operations E and D are inverses:

$$\begin{aligned} D(E(s)) &= (s^e)^d \pmod{n} \\ &= s^{de} \pmod{n} \\ &= s^{1+km} \pmod{n} \\ &= s(s^m)^k \pmod{n} \\ &= s\,1^k \pmod{n} \\ &= s \pmod{n} \\ &= s \end{aligned}$$

For the last step to hold, the integer s representing the plaintext must be less than n. That's why we break the message up into chunks. Also, this only works if s is relatively prime to n, that is, it has no factors in common with n other than 1. If n is the product of two large primes, then all but negligibly few messages $s < n$ satisfy this property. If by some freak chance s and n turned out not to be relatively prime, then the code would be broken; but the chances of this happening by accident are insignificantly small.

In summary, to use the RSA system:

1. Pick large primes p and q.
2. Compute $n = pq$ and $m = (p-1)(q-1)$.
3. Choose e relatively prime to m and use this to compute d such that $[d\,e = 1] \pmod{m}$.
4. Publish the pair (n, e) as your public key, but keep d, p, and q secret.

How secure is RSA? At present, the only known way to obtain d from e and n is to factor n into its prime factors p and q, then compute m

and proceed as above. But despite centuries of effort by number theorists, factoring large integers efficiently is still an open problem. Until someone comes up with an efficient way to factor numbers, or discovers some other way to compute d from e and n, the cryptosystem appears to be secure for large numbers n.

Image Credits

Figure 1. Portrait of Whitfield Diffie by Duncan Hall, courtesy of Wikimedia, licensed under CC BY-SA 4.0. Portrait of Martin Hellman, courtesy of Wikimedia, licensed under CC BY-SA 3.0.

1 Version information: Commit b72ee47 from 2023-02-05 by Stuart Shieber. CI build of I_TAG ().