CS51 PROBLEM SET 3: BIGNUMS AND RSA

STUART M. SHIEBER

This problem set is *not* a partner problem set.

Introduction

In this assignment, you will be implementing the handling of large integers. OCaml's `int` type is only 64 bits, so we need to write our own way to handle very large numbers. Arbitrary size integers are traditionally referred to as "bignums". You'll implement several operations on bignums, including addition and multiplication. The challenge problem, should you choose to accept it, will be to implement part of RSA public key cryptography, the protocol that encrypts and decrypts data sent between computers, which requires bignums as the keys.

To create your repository in GitHub Classroom for this homework, click this link. Then, follow the GitHub Classroom instructions found here.

Reminders.

Compilation errors: In order to submit your work to the course grading server, *your solution must compile against our test suite.* The system will reject submissions that do not compile. If there are problems that you are unable to solve, you must still write a function that matches the expected type signature, or your code will not compile. (When we provide stub code, that code will compile to begin with.) If you are having difficulty getting your code to compile, please visit office hours or post on Piazza. *Emailing your homework to your TF or the Head TFs is* not *a valid substitute for submitting to the course grading server. Please start early, and submit frequently, to ensure that you are able to submit before the deadline.*

Testing is required: As with the previous problem sets, we ask that you explicitly add tests to your code in the file `ps3_tests.ml`.

Time estimates: Please estimate how much time you spent on each part of the problem set by editing the line:

```
let minutes_spent_on_pset : int = _ ;;
```

at the bottom of the file.

Compilation: Write a Makefile if you want one. Take a look at the instructions here if you're stuck.

Style: Style is one of the most important aspects of this and all future problem sets, and will be a significant portion of your grade. Please refer to the CS51 Style Guide for some considerations of writing well-styled code.

Date: February 14, 2018.

As in the previous assignments, please make sure that:

- Your code compiles without warnings on the course grading server.
- Your code adheres to the CS51 Style Guide.
- You have written at least one test per code path per function.
- Your functions have the correct names and type signatures. Do not change the signature of a function that we have provided.

(This assignment was adapted from CS312 at Cornell.)

1. Big Numbers

The byte integer type in OCaml has a size of 63 bits, and therefore can represent integers between -2^{62} and $2^{62} - 1$ (given respectively by the OCaml values `min_int` and `max_int`).

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

Some computations, however, require integers that are larger than these; for instance, RSA public key cryptography requires big integers. In this part of the problem set, you will write code that allows OCaml to handle arbitrarily large integers (as large as the 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.

1.1. Bignum Implementation.

In this bignum implementation, an integer n will be represented as a list of integers, 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 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:[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 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:

[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.

[2] For reasons of simplicity in implementing functions to move between bignums and OCaml integers, 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.

```
# 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 represents 0. If `neg = true`, the record represents the corresponding negative integer. This defines the correspondance between `bignum` and the integers.

1.2. **Using your solution.** Using the functions `from_string` and `to_string`, you will be able to test your functions by converting the result to strings and printing them. (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 ans = times (from_string "123456789") (from_string "987654321") ;;
# let ans = times (from_string "123456789") (from_string "987654321") ;;
Error: Unbound value times
```

The implementation of bignums will be simpler if we impose certain representation invariants. A REPRESENTATION INVARIANT is a statement that you enforce about values in your representation and that you can thus assume is true when writing functions to handle these values. (For example, if we design a data type to hold person names, we might impose the representation invariant that all of the characters stored are letters or spaces.) 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.

REPRESENTATION INVARIANT

The invariants for the bignum representation are as follows:

- Zero will always be represented as `{neg = false; coeffs = []}`. It should never be represented 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.
- 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.

Be sure that your functions preserve this invariant. 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.

1.3. **What you need to do.** Implement the following functions in `ps3.ml`, which operate on bignums. As usual, feel free to change the syntax of the function definition, for instance by adding a `rec` keyword, if you think it helpful, but do not alter the signatures of any functions, as we will be unit testing assuming those 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. Assume that the arguments satisfy the representation invariant.* □

Problem 3. *Implement conversion functions* `to_int : bignum -> int option` *and* `from_int : int -> bignum` *between integers and big numbers.* `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 in the example 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 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, but remember that we are counting in base 1000 (say), not 10. The main goal is correctness, so keep your code simple. 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).*

□

2. Challenge Problem: The RSA Cryptosystem

As on last week's problem set 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.

This problem is a great way to verify that your previous functions work correctly, since it uses almost all of them. Each of the three parts of RSA can be implemented in approximately one line of code if you use the right functions.

In a world that relies increasingly upon digital information, public key encryption plays an important part in achieving private communication. RSA is a popular public-key encryption system. It is based on the fact that there are fast algorithms for exponentiation and for testing prime numbers, but no known fast algorithms for factoring large numbers. In this problem set you will implement a version of the RSA system. The algorithms will you implement have both a deep mathematical foundation and practical importance.

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. Two parties can exchange coded messages only if they share a 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!

Diffie and Hellman (1976) discovered a new approach to encryption and decryption: public key cryptography. In this approach, the encryption and decryption keys are different. Knowing the encryption key cannot help you find the decryption key. Thus you can publish your encryption key on the web, 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 RSA system, due to Rivest, Shamir, and Adelman, is the best-known public-key cryptosystem; the security of your web browsing probably depends on it.

2.1. How RSA Works.

RSA does not work with characters, but with integers. That is, it encrypts numbers. To encrypt a piece of text, just combine the ASCII codes of several characters into a bignum, and then encrypt the resulting number.

Suppose you want to obtain 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.) 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) 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: 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.

2.2. **Your task: Implement RSA.** You have been given a partial implementation of the RSA algorithm. The given code provides additional functions for computations on big numbers; functions to generate prime numbers and code that generates key pairs. We briefly discuss a few of these functions. You should call these functions as needed in your code. They handle most of the mathematical and cryptographic detail for you.

Function `exp_mod` b e m computes $b^e \pmod{m}$. Function `euclid` m n yields numbers (s, t, g) such that $sm + tn = g$, where g is the greatest common divisor of m and n.

Function `is_prime` tests if a number is prime. This is a probabilistic test: it yields true if there is a high probability that n is prime; and yields false if n is definitely not prime. It's worth mentioning that this function uses the well-known Miller-Rabin primality test, partially named after Harvard Professor Michael Rabin, who developed the algorithm.

The function `generate_key_pair r` generates an RSA key pair, and returns a tuple `e, d, n`. It picks random primes p and q that are in the range from 2^r to 2^{r+1} such that $n = pq$ will be in the range 2^{2r} to 2^{2r+2}.

You must add the encryption and decryption functions specified below. Your functions encode an input list of bytes into an output list of bytes. Your functions will operate on *blocks of bytes* (or, equivalently, blocks of chars, since a char takes 1 byte to store). Each block consists of M bytes or $(M-1)$ bytes, where M is the number of bytes required to store the modulus n of the key (n, e) or (n, d). You can use the function `bytes_in_key` to compute this from `n`.

Problem 6. ***Challenge*** *Implement the function* `encrypt_decrypt_bignum`*, which encrypts or decrypts a bignum. (Since the operations are so similar, as described above, we can use the same function. The only difference is that to encrypt you provide e and to decrypt you provide d.)* □

Note: For this problem and the one below, it's probably difficult to write tests that check whether the result matches your expected result. However, you can write tests that check whether a bignum encrypted using RSA and then decrypted using the corresponding decryption key results in the original bignum.

Problem 7. ***Challenge*** *You'll note that RSA acts on bignums, so this gives us no way to directly encode strings. Since very few secret message consist of large integers, we need a way to convert strings (or lists of bytes) into bignums. You may also realize from the description of RSA that each bignum must be less than the modulus n. We have given you the function* `chars_to_bignums` *which transforms a list of chars to a*

list of bignums, each of which is less than n. It does this by breaking the list of chars into blocks of M bytes, where M is the number of bytes required to store n. We've also provided the inverse transformation `bignums_to_chars`*. You may also find the functions* `explode` *and* `implode` *defined higher in the file useful.*

Finally, we've provided the function `encrypt_decrypt_bignums` *which encrypts or decrypts a list of bignums. This simply calls your* `encrypt_decrypt_bignum` *repeatedly.*

Your job is to write the functions `encrypt`*, which takes n, e and a string to encode, and returns the encoded message as a list of bignums, and* `decrypt`*, which takes n, d and a list of bignums provided by* `encrypt`*, decrypts the message and produces a string. Note that these functions are doing more than just a direct translation of chars to bignums and vice versa: we've given you those functions. Your functions should actually encrypt these bignums with RSA, so that nobody will be able to reconstruct the original bignums, or the original message, without the decryption key. Not bad for your second week of CS51!*

To exchange secret messages with your friends, use `generate_key_pair` *to get an RSA keypair. Send the pair* `(n, e)` *to your friend and have them encrypt a message using this key and send it to you. You can then decrypt it using* `n` *and* `d`*, but nobody else can because they don't have* `d`*.* □

3. Challenge Problem: Multiply Faster

The multiplication algorithm you implemented in Section 1 will work just fine for most integers of reasonable sizes, including the ones we will be using in this assignment. 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 8. ***Challenge*** *See if you can implement such an algorithm in* `times_faster`*. You may want to start by googling 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!* □

4. Submission

Before submitting, please estimate how much time you spent on the problem set by editing the lines in each file that look like

```
let minutes_spent_on_pset () : int = failwith "not provided" ;;
```

to replace the value of the function with an approximate estimate of how long (in minutes) this part of the problem set took you to complete. Make sure it still compiles.

Then, to submit the problem set, follow the Gradescope instructions found here.