Project 1: Bignum

Due: 7:00 PM, Oct 12, 2018

Contents

1 Introduction .................................................. 1

2 The Assignment ............................................. 2
   2.1 Describe how bignums are represented ..................... 2
   2.2 ALU Support Code ......................................... 3
   2.3 Two Procedures ........................................... 4
   2.4 Analysis ................................................... 4

3 Restrictions .................................................. 4

4 Handing In .................................................... 5
   4.1 Design Check ............................................... 5
   4.2 Final Handin ............................................... 5
   4.3 Grading .................................................... 6

5 Extra Features ................................................. 7

6 Appendix: Algorithms ........................................ 7
   6.1 Long Addition .............................................. 7
   6.2 Multiplication ............................................. 8

Code Quality

Source: https://xkcd.com/1513/
1 Introduction

...the different branches of Arithmetic—Ambition, Distraction, Uglification, and Derision.

— Lewis Carroll, Alice in Wonderland

The CPU (central processing unit) of a computer is the part of the computer that actually computes. (Okay, there’s also something called the GPU, but let’s not go there...) As part of the computing, it must perform arithmetic and logical operations, and for that it has an ALU (arithmetic/logic unit). Built into the hardware of the ALU is the ability to perform arithmetic operations: addition, subtraction, multiplication, and division. However, there is only a limited amount of circuitry in the ALU, so it is only capable of arithmetic on numbers of a limited number binary digits (bits).

Most programming languages, including C (1972) and Java (1995), therefore provide only limited-precision arithmetic. However, the tradition in Lisp, starting at least with Maclisp (1965), is to provide arbitrary-precision integer arithmetic. (This tradition is carried on by Racket and other languages, including Python.)

As we mentioned in class, arithmetic using huge integers is central to commonly used cryptographic protocols, including those used when you log in remotely using FastX or SSH, or when you access a secure website and communicate your credit card or other private information.

If you have procedures for doing arithmetic using large numbers, these protocols are pretty easy to implement. This means that those of you who want to try out implementing, for example, RSA in Scheme (Racket) can go ahead!

However, it also means that there is a bit of magic in Scheme’s built-in arithmetic procedures, and we don’t like unexplained magic in this class. It also means that assigning unit cost to arithmetic operations (for the purpose of predicting computation times) can lead to inaccurate models if the computation involves huge integers (consisting of hundreds of digits).

We will therefore (1) implement our own arbitrary-precision arithmetic package in Scheme, and (2) analyze the procedures we write to see how long they take as a function of the sizes of the inputs.

Of course, the time these procedures take depend on what algorithms they are based on. It is certainly possible to write “correct” procedures such that multiplying two thousand-digit numbers would take thousands of years. It is also possible to write super-fast multiplication procedures, by drawing on mathematics well beyond the scope of this course. One very early efficient multiplication algorithm was developed by the Persian mathematician al-Khwarizmi (it’s from his name that we get the word “algorithm”).

We will seek a middle ground. We will use algorithms that are familiar to you—they are the first nontrivial algorithms you ever learned: long addition and long multiplication.

I said that you will write an arbitrary-precision arithmetic package, which ordinarily would include addition, subtraction, multiplication, division, but you are only required to write procedures for addition and multiplication.

---

1 You can learn more by reading (shameless plug) A Cryptography Primer: Secrets and Promises, written by your devoted professor, or by reading countless no doubt superior tutorials on the web. (Search for RSA and DSA, to start with.)
The familiar long addition and long multiplication algorithms are based on addition and multiplication of single digits, so your code should also be based on addition and multiplication of single digits. You could in principle use Racket’s built-in addition and multiplication procedures, but that wouldn’t make it clear that your code doesn’t use their full power. We will therefore provide you arithmetic procedures to use in your code; our procedures are intended to be used on single-digit nonnegative integers (i.e. from 0 to 9). (We cut you a little slack—these procedures will accept numbers in the range 0 to 99—as long as the output is no larger than 99.)

2 The Assignment

You will write Racket procedures that perform addition and multiplication of arbitrary-precision nonnegative integers—bignums—using Racket’s primitive arithmetic operations restricted to limited-precision integers. To do this, you will first decide how you want to represent bignums, and then implement two algorithms, long addition and long multiplication, for bignums. Finally, you will analyze the runtime of your procedures.

2.1 Describe how bignums are represented

You should begin by writing how you intend to represent arbitrary-precision nonnegative integers as bignums. In our familiar Base-10 representation, the representation of an integer consists of digits (integers in the range 0 to 9) in different “places” holding different values based on the place and the value of the digit. Your representation of bignums should similarly consist of a sequence of digits. By combining together digits in sequence, you will be able to represent arbitrarily large integers as bignums.

You get to choose how you want to represent a number like 98472. You’re used to the Arabic representation, which is the one in the previous sentence, but just to give you an idea of other possible representations, there are things like Roman numerals (which would be a very bad choice), or the multiples-of-fives-plus-a-few-ones used for working with an abacus, etc. Go ahead and be open to multiple possibilities.

If your program generates or processes any intermediate data structures, you should also describe these.

2.2 ALU Support Code

To mimic a computer with limited ALU, we have provided you with the procedures digit-add, digit-mult, digit-sub, digit-quo, and digit-rem for doing any and all arithmetic. In particular, you are not allowed to use Racket’s built-in +, *, -, quotient, or remainder, all of which can take in inputs and produce outputs much larger than 99.

We have defined these procedures in bignum-operators.rkt file, located at /course/cs0170/src/bignum/bignum-operators.rkt. You'll need to copy this file to your working directory and include the line (require "bignum-operators.rkt") at the top of your bignum.rkt file to gain access to the procedures we’ve defined.

There’s nothing magic about our code. Here’s the entirety of digit-add:
;; Input: a, a natural number between 0 and 99
;; b, a natural number between 0 and 99
;; Output: the sum of a and b, or an error if a, b or their sum is
;; outside the range [0,99]

(define digit-add
  (lambda (a b)
    (if (and (integer? a) (integer? b) (<= 0 a) (<= a 99) (<= 0 b) (<= b 99) (<= (+ a b) 99)) (+ a b) (error 'add "digit - add inputs or output out of range. Tried (+ ~a ~a) " a b)))))

The only new thing you haven’t see here is the error procedure, which prints an error message and halts processing. You’ll never use this procedure yourself, but you can see its role in this procedure.

The other digit operations are implemented with very similar procedures.

Actually, all but one of them is implemented. Your first task, which is required for the design check, is to read through the definitions and fill in the code for digit-sub. Consider carefully which conditions you must check to ensure that digit-sub does not take inputs or return outputs outside the appropriate range. We’ve left a skeleton version of the code, including the appropriate error message, in bignum-operators.rkt. For this task alone, you are permitted to use the standard +, −, *, / operations. You still can’t use them anywhere else in your bignum+ and bignum* code of any of their helpers.

Using our digit operations, you will write procedures for adding and multiplying bignums.

2.3 Two Procedures

Once you choose a suitable bignum representation, your next task is to develop the following procedures by following the design recipe:

- bignum+, which computes the sum of two bignums, producing a new bignum.
- bignum*, which computes the product of two bignums, producing a new bignum

Let $n$ be the sum of the sizes of the inputs. Ideally, the bignum+ procedure should run in $O(n)$ time (linear time), and bignum* should run in $O(n^2)$ time (quadratic time). A portion of your grade will be dependent on writing procedures with the specified runtime, but you will recieve partial credit for less efficient but functional versions of bignum+ and bignum*.

Keep in mind that you should also include comprehensive test cases for each individual procedure. Creating interesting test cases for this project can be a pain in the neck. For example, writing out the bignum that represents the number 1000009 is pretty tough, as it’s easy to write six zeroes
instead of five. That means you’ll want to take particular care in writing your test cases. However, you’ll find that a good testsuite comes in handy when debugging!

As for the algorithms to use—you probably remember how to do long addition and long multiplication. If you need reminders, feel free to check out the appendix or you can check out these Khan Academy videos on the subject: [this video on addition] and [this video on multiplication].

2.4 Analysis

You must also analyze the runtime of your two main procedures. You should formulate recurrence relations for your recursive procedures, and use these to show (drawing on theorems presented in class) the right big-O characterization of the running times. You will receive credit for this part of the assignment if you indeed get the right big-O characterization, even if that big-O characterization is not what we had hoped for.

3 Restrictions

You may not use any mathematical procedures other than the five defined in bignum-operators.rkt when writing bignum+, bignum*, or any of their helpers.

You may use Racket documentation, specifically [http://racket-lang.org], but you may not use any functions not listed in our docs. If in doubt about whether you can use something, ask a TA!

4 Handing In

4.1 Design Check

You are required to pair program this project. We recommend finding a partner as soon as possible, as you will not be able to sign up for a design check until you have one. You can use Piazza to find a partner; check with the TAs if you do not know how to do this yet.

Design checks will be held on Oct 2-4, 2018. You and your partner should sign up for one design check slot as soon as possible. To sign up, one partner should run the script cs0170_signup bignum <other partner's login>. All available design check slots will be printed to the terminal and you will be prompted to choose one. If you have already signed up and wish to change your slot, run the same command again and you will be given the option to cancel your sign-up and choose again.

You and your partner should do the following to prepare for your design check:

- Finish the implementation of digit-sub in ‘bignum-operators.rkt’
- Come up with a description of your representation for bignums and any anticipated intermediate data types.
- Figure out how you will write bignum+ and bignum*, given your representation of bignums. You should be able to show a TA—step-by-step—how you would evaluate (bignum+ bignum1
bignum2) and (bignum* bignum1 bignum2) on a few small bignums of the TA’s choice. You
should understand the algorithms you will use even if you don’t yet know how you will code
them.

Before your design check meeting, upload ‘bignum-operators.rkt’ with your implementation
of digit-sub to Gradescope. You must upload your file before your design check, even if that slot
is before the official Gradescope deadline. Only one partner should hand in the file.

During the design check, you will discuss your solutions to each of the questions above with a TA.
Please come prepared! The better prepared you are, the more productive the design check will be,
and if you come away from the design check with a good understanding of how to proceed, you
should do well on the project. Please also come on time—arriving late to your design check may
result in a deduction.

4.2 Final Handin

The final handin is due by Due: 7:00 PM, Oct 12, 2018. To hand in your files, upload your files via
Gradescope. Only one partner should hand in the project.

For the final handin, your group is required to hand in four files: the source file ‘bignum-operators.rkt’,
a ‘README.txt’ file, a ‘bignum.rkt’ file, and a ‘analysis.txt’ file.

In the ‘bignum.rkt’ file, you should include all your code: bignum+, bignum*, and any other
helper procedures you wrote. As always, you must write check-expects for every procedure you
write.

In the ‘analysis.txt’ file, you should include an run-time analysis of your procedures. You may
use any notation that looks reasonably math-like. We recommend writing things like \(x^2\) to indicate
\(x^2\), for instance. Whatever you do, make sure your writeup is a plain text file, not rich-text format
or a Microsoft Word file. If you don’t know how to be sure you’ve written a plain text file, talk to
the TAs, or even discuss it during the design check.

In the ‘README.txt’ file, you should include:

- an explanation of your representation for bignums (why did you choose this particular
  implementation?)
- instructions describing what sort of expression a user would type using the procedures you’ve
  written to do arithmetic with bignums
- an overview of how all the pieces of your program fit together (when a user provides an input,
  what series of procedures are called in order to produce an output?)
- a description of any possible bugs or problems with your program (You will lose fewer points
  for documented bugs than undocumented ones.)
- a list of the people with whom you collaborated (especially your partner)
- a description of any extra features you chose to implement.
4.3 Grading

The design check counts for 20% of your grade. Specifically,

- Implementation of digit-sub: 1 pt
- Data definition and examples: 6 points
- Walkthrough of adding two bignums: 6 points
- Walkthrough of multiplying two bignums: 7 points

Note: For this project only, you can get back any points you lose on the design check if you correct your mistakes on your final handin. (Note that if you don't show up to your design check, or if you show up and haven't done any work, you can't get these points back.)

Functionality counts for 50% of your grade. Specifically,

- bignum+: 25 points
- bignum*: 25 points

Partial functionality merits partial credit.

Your analysis counts for 20% of your grade.

The final 10% of your grade is reserved for style.

You should always:

- Follow the design recipe, including recursion diagrams, input/output specifications, and test cases, for all procedures you write.
- Spend time trying to find elegant solutions to your problems. Clearer solutions usually result in shorter code that is easier to understand and (more important to you) to debug.
- Follow the CS 17 Racket style guide.

You will not receive full credit for style if a TA is ever in doubt about what a section of your code is doing or how it works.

5 Extra Features

If you finish your project early, but want to keep playing with bignums, we suggest you implement long division. Once you have done that, you can implement modular exponentiation:

- **input**: bigint b, bigint e, bigint m
- **output**: the value of $b^e \mod m$
There is a clever algorithm for writing an efficient exponentiation procedure based on multiplication. The other trick is this: to keep the intermediate numbers from getting too big, you should reduce each intermediate number mod \(m\). If you are interested in doing this, get in touch with the professor for guidance.

Note that these extra features don’t count towards your grade.

6 Appendix: Algorithms

6.1 Long Addition

Long addition is an algorithm that reduces the problem of adding up large numbers to the problem of adding up their digits one by one. The algorithm works by aligning the numbers’ columns, and then adding their corresponding digits, one at a time. In grade school, you were taught to sum the columns from right to left, and to carry a 1 to the next column if ever a sum exceeded 9. With your bignums, we suggest one of two methods: either have a procedure add each column, and then account for the carries by fixing the result, or write a procedure that can keep track of carry digits. Whichever method you choose, be sure to have all of your data types explicitly defined.

For example, here’s how you could add 99 and 89:

\[
\begin{array}{c}
9 \\
+ 8 \\
\hline
17 \\
\end{array}
\]

But the sequence 17 18 is not our answer. We would have to apply some procedure to convert this result to the number 188.

Or you could add 99 and 89 by keeping track of a carry value, like so:

\[
\begin{array}{c}
1 \\
+ 8 \\
\hline
18 \\
\end{array}
\]

6.2 Multiplication

Multiplication You may have already learned the classic long multiplication algorithm. The following description of this algorithm appears in Wolfram’s Mathworld:

The long multiplication algorithm starts with multiplying the multiplicand by the least significant digit of the multiplier to produce a partial product, then continuing this process for all higher order digits in the multiplier. Each partial product is right-aligned with the corresponding digit in the multiplier, and the partial products are summed.

Here’s how you would multiply 17 and 18 using this algorithm:

\[
\begin{array}{c}
1 \\
+ 8 \\
\hline
18 \\
\end{array}
\]
After setting up the problem, you begin with the rightmost (least significant) digit of the bottom number, and multiply it by every digit of the number on the top. Here, we would first multiply 8 by 7, yielding 56. We place the 6 down below the current digit of the top number, and place the carry digit, 5, above the next place in the top number.

\[
\begin{array}{c}
5 \\
1 \\
\times 1 \\
\hline
7 \\
6
\end{array}
\]

Next, we multiply 8 by the next digit of the top number and add the carry digit, yielding \(8 \times 1 + 5 = 13\). Again, the 3 goes into the result and we carry the 1.

\[
\begin{array}{c}
1 \\
1 \\
\times 1 \\
\hline
5 \\
7 \\
\hline
1 \\
3 \\
6
\end{array}
\]

Since we’ve reached the end of our top number our next multiplication is \(8 \times 0 + 1 = 1\) and we just move down the 1 to the result. we have now computed the first partial product, 136.

\[
\begin{array}{c}
1 \\
1 \\
\times 1 \\
\hline
5 \\
7 \\
\hline
1 \\
3 \\
6
\end{array}
\]

Having reached the end of our top number, we now repeat the process described above for the second least significant digit in our bottom number, to achieve our second partial product, 170.

\[
\begin{array}{c}
1 \\
1 \\
\times 1 \\
\hline
7 \\
7 \\
\hline
1 \\
3 \\
6 \\
\hline
1 \\
7
\end{array}
\]

Once you have repeated the above process to get a partial product corresponding to each digit in the bottom number, we add together all those partial products to get our answer. In this case we would add together 136 and 170 to get the final product 306.

\[
\begin{array}{c}
1 \\
1 \\
\times 1 \\
\hline
7 \\
6 \\
\hline
1 \\
3 \\
6 \\
\hline
1 \\
7
\end{array}
\]

\[
\begin{array}{c}
+ \\
3 \\
\hline
3 \\
0 \\
6
\end{array}
\]

9
Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: http://cs.brown.edu/courses/csci0170/feedback