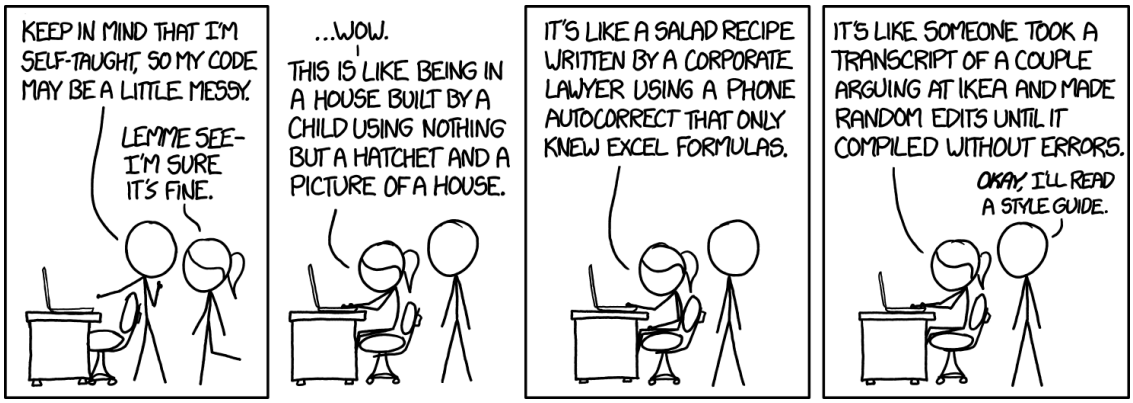
Project 1: Bignum

7:00 PM, Oct 11, 2019

Contents

> Hofstadter's Law: It always takes longer than you think it will, even if you take into account Hofstadter's Law.
>
> –Douglas Hofstadter.
> *Gödel, Escher, Bach: An Eternal Golden Braid.*

Code Quality



Source: `https://xkcd.com/1513/`

1 Introduction

Think about a child who only knows the digits 0 through 9. Adding 2 to 3 makes sense because the answer, $2 + 3 = 5$, is a number the child knows about, but $7 + 8$ is hopeless: the child might say "That's too big. There's no such number!"

Later, though, the child learns about numbers like 15, where the 5 represents 5, but the 1 shifted over in the next place represents 1×10^1. The idea of using "places" to let you represent bigger numbers is pretty cool. But learning to add these multi-digit numbers together requires that you use a rule that involves "carrying".

It happens that the way you "carry" when adding or multiplying multi-digit numbers really constitutes an "algorithm," and in fact was one of the first algorithms ever written down. The Persian mathematician al-Khwarizmi was the first to write down this algorithm, and it's from his name that we get the word "algorithm."

Arithmetic on a computer is done in much the same way as we do it by hand, except that while our "addition table" and "multiplication table" are 10 rows by 10 columns, because there are ten Arabic numeral digits, a computer's addition table is typically much larger. For a small computer, the basic "arithmetic logic unit" (ALU) can be used to add or multiply any two numbers up to about $60,000$. When we need to multiply larger numbers, we use sequences of these basic numbers, just as we write sequences of digits when we work on paper[1].

Of course, $60,000$ is a lot of "digits" to have in your number system, so for this project, we're going to do the same kind of task — using limited-size arithmetic to let us do arbitrary-size arithmetic — but on a computer whose basic ALU can only work with numbers from 0 to 99. And even with those numbers, there are operations it can't do: if you try to multiply 18 by 13, it'll fail, because the result is bigger than 99, and our system can't handle numbers of that size. So about the largest thing you can reliably do is multiply 9 by 9. We'll call the numbers between 0 and 9—the numbers on which we can reliably perform arithmetic operations—"digits".

This trick of using limited-size arithmetic to compute arbitrary size arithmetic is more or less what languages like Racket do for *all* numbers, so Racket will happily compute for you the value 2^{10000}, which has about 3000 digits. Your program will mimic this behavior by adding and multiplying arbitrary-size numbers called "bignums" that are represented by sequences of digits.

To do so, you'll be allowed to use a "pretend ALU" that we've provided for you. It has functions like `digit-add`, which takes two arguments, each of which is an integer between 0 and 99, and adds them. If the result is less than 100, `digit-add` returns it, but generates an error otherwise. You will use our ALU to do "digit-level operations" on the digits of your bignum. What you may *not* do is use the ordinary `+, -, *, /` operations of Racket anywhere in your code. If you want to multiply something, you'll need to use `digit-mult`.

Let's look a little closer at the task at hand.

[1] *Inside* the ALU, just the opposite sort of thing happens: numbers are represented as sequences of *bits* (i.e., zeroes and ones) and the addition and multiplication table are both 2×2.

2 The Assignment

You will write Racket procedures that enable addition and multiplication of arbitrary-size integers—**bignums**—using the various digit operations. 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 Data Definition

Following the design recipe, you should begin by writing a data definition for a **bignum**, which will be used to represent an arbitrary natural number. The way that we represent numbers in our familiar decimal representation consists of digits 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**, or integers ranging from 0 to 9, inclusive. By stringing 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 types, you should also include data definitions for those types.

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`:

```
;; digit-add : num * num -> num

;; Input: two natural numbers between 0 and 99, a and b
;; 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 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 what its role is 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 will be discussed at 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`.

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 length of the longer bignum input to one of these procedures. The `bignum+` procedure should run in $O(n \mapsto n)$ (linear time), while `bignum*` should run in $O(n \mapsto n^2)$ (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/example cases. However, you'll find that a good testsuite comes in handy when debugging!

2.4 Analysis

You must also analyze the runtime of your two main procedures. For `bignum+`, you must find a recurrence that the operation-counting function satisfies, and then show that any solution of that recurrence is in a particular big-O class.

For `bignum*`, you need only derive the recurrence, but need not solve it. The details for each of these are presented below.

Let $A(n)$ be the largest number of elementary operations involved in applying `bignum+` to any two operands of exactly n digits each. You should consider `digit-add` and `digit-mult`, etc., to be elementary operations (along with the usual ones like `first` and `cons`).

Write a recurrence relation that that the function A satisfies. Then, use your knowledge of recurrence relations and big-O to make a conjecture about the big-O class of A, just as we did for the procedure `length` in class. Be sure to explain the reasoning behind your conjecture.

Now, we'll consider `bignum*`. Leaving aside your answer above, let $A(n)$ be the largest number of elementary operations required to apply `bignum+` to any two bignums, each of length no greater than n. Furthermore, let $P(n)$ be the largest number of operations required to take get a partial product by multiplying each digit of a bignum of length n by a single digit. Now, let $M(n, k)$ be the largest number of elementary operations required to apply your bignum* procedure to a bignum of length n and a bignum of length k. Write a recurrence for $M(n, k)$. Feel free to refer to $A(n)$ and $P(n)$ as necessary in your answer.

Note: There are multiple base cases to consider in this recurrence relation! Multiplying two bignums both of length 0, for example, takes some small constant number of operations, call it Q. We've laid out the cases for you below; all you have to do is fill in the last three.

$$M(n,k) \leq \begin{cases} Q & \text{if } n = k = 0 \\ ? & \text{if } n = 0, k > 0 \\ ? & \text{if } n > 0, k = 0 \\ ? & \text{otherwise} \end{cases}$$

The missing expressions in the last three lines may depend on n or k.

Note: Even if you do not achieve the runtime we specified for `bignum+` or `bignum*`, you can still recieve full credit for this portion of the assignment; if your analysis explains that your `bignum+` runs in $O(n \mapsto n^n)$, say, and your code actually does run in $O(n \mapsto n^n)$, that is OK: you've analyzed it correctly. On the other hand, its failure to run in $O(n \mapsto n)$ time means that you'll lose some credit for the efficiency requirement.

3 Restrictions

You may not use any mathematical procedures other than the five defined in bignum-operators.rkt.

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 1-3, 2019. You and your partner should sign up for one design check slot as soon as possible. To sign up, one partner should go to the Design Check signup link on Piazza. All available design check slots will be listed in the form and you will be prompted to choose one. If you have already signed up and wish to change your slot, you can edit your response to the form.

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 data definition 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 don't have to code these procedures for the design check, but it'd be a good idea to sketch out the algorithms on paper.

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.

Make sure you come prepared to the design check. 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. Plus, the design check itself is worth a significant portion of your grade! Also, please arrive on time. Arriving late to a design check may result in a deduction.

Ideally, when the design-check is over, all you have left to do is *typing*, not *thinking*. You should aim for this.

4.2 Final Handin

The final handin is due by 7:00 PM, Oct 11, 2019. 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.

In the '`analysis.txt`' file, you should include an analysis of the functions A and M. 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 data definition 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 (rest assured, 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.

- Follow the design recipe, including data definitions and examples of the data, type signatures, specification, 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, there are lots of additional features you can implement. If you'd like to hear about possible extra features, email the TA list and we will be happy to discuss them. Note that these are strictly for fun: they do not count towards your grade!

6 Appendix: Algorithms

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:

$$\begin{array}{rcc} & 9 & 9 \\ + & 8 & 9 \\ \hline & 17 & 18 \end{array}$$

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:

$$\begin{array}{rcc} 1 & 1 & 0 \\ \hline & 9 & 9 \\ + & 8 & 9 \\ \hline 1 & 8 & 8 \end{array}$$

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:

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.

$$\begin{array}{rrr} & & \mathbf{5} \\ \hline & & 1 \quad \mathbf{7} \\ \times & & 1 \quad \mathbf{8} \\ \hline & & \mathbf{6} \end{array}$$

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

$$\begin{array}{rrr} & \mathbf{1} & 5 \\ \hline & & \mathbf{1} \quad 7 \\ \times & & 1 \quad \mathbf{8} \\ \hline & & \mathbf{3} \quad 6 \end{array}$$

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

$$\begin{array}{rrr} & 1 & 5 \\ \hline & & 1 \quad 7 \\ \times & & 1 \quad \mathbf{8} \\ \hline & \mathbf{1} & 3 \quad 6 \end{array}$$

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.

$$\begin{array}{rrr} & & 1 \quad \mathbf{7} \\ \times & & \mathbf{1} \quad 8 \\ \hline & 1 & 3 \quad 6 \\ & & \mathbf{7} \end{array}$$

$$\begin{array}{rrr} & & \mathbf{1} \quad 7 \\ \times & & \mathbf{1} \quad 8 \\ \hline & 1 & 3 \quad 6 \\ & \mathbf{1} & 7 \end{array}$$

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

$$\begin{array}{rrrr} & & 1 & 7 \\ & \times & 1 & 8 \\ \hline & 1 & 3 & 6 \\ + & 1 & 7 & \\ \hline & \mathbf{3} & \mathbf{0} & \mathbf{6} \end{array}$$

If this algorithm is new or confusing to you, we recommend you check out this video for some in depth examples.

You should implement long multiplication using your solution to long addition as a helper, but you may not call that solution 17 times to multiply a number by 17. (We didn't explicitly say so above, but you also cannot call `add1` or anything equivalent 17 times to add 17 to a number.)

Primitives You may find the built-in Racket procedures `quotient` and `remainder` useful to you in this assignment (although you may only use their digit-equivalents, `digit-quo` or `digit-rem`, of course, within `bignum+` and `bignum*`!). Evaluating (`quotient num1 num2`) produces the integer quotient of `num1`, the dividend, divided by `num2`, the divisor. Evaluating (`remainder num1 num2`) produces the remainder. For example:

```
(quotient 100 3)
=> 33

(remainder 100 3)
=> 1
```

These operators can be used to extract digits from a (Racket) number, which can be a useful way to create a bignum on which to test your two main procedures, although creating input by hand is every bit as good. But within the bignum addition and multiplication procedures, and within any helper procedures, only `digit-quo` and `digit-rem` may be used.

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`.