Algorithms & Data Structures: Lab 17

week of 4th March 2019

1 Setup

1.1 Saving your work from last week

By now you should be familiar with the operations needed to save your work. Make sure you commit your work to version control often, and always have a backup copy, ideally remotely (for example in your own account on the department's gitlab installation.)

1.2 Downloading this week's distribution

Once you have successfully saved your changes from last week, you can get my updates by doing

```
git pull
```

which *should* automatically merge in new content. After the `git pull` command, you should have a new directory containing this week's material (named `17/`) alongside your existing directories.

2 Big Integers

This lab exercise involves implementing a Big Integer data structure, and some arithmetic operations involving Big Integers.

A Big Integer behaves more like a mathematical integer than the fixed-point numbers usually found as primitives in computers: there is no hard upper limit for Big Integers; instead, if larger numbers are computed then space is made for them. Mathematical operations on them behave like pen-and-paper integers, without any kind of overflow or wraparound behaviour.

In this implementation we will use base 10 for our digits, just as with standard arithmetical notation, and restrict ourselves to unsigned integers.

2.1 Basic Data Structure

Implement the basic storage for a Big Integer as an array named `data` (to store the digits of the integer) of type `char` (Java) / `unsigned char` (C++). You will need a constructor, whose argument should be the size of the array of digits, and that size also needs to be stored in the `ndigits` member variable.

Implement the `get(i)` accessor, which should return the i^{th} digit from the digit array, counting from the units position upwards – or `0` if the position requested is off the end of the Big Integer. Once this is done correctly, some of the tests in the test suite should pass.

2.2 Addition

Implement addition of two big integers. You will need to loop over each digit position in turn, performing the sum of the two digits along with a possible carry from the previous position, storing the relevant digit in the right position in the result, and storing the carry for the next part of the addition. The following pseudocode will give you some idea.

```
function ADD(a,b)
    n ⟵ 1 + max(NDIGITS(a), NDIGITS(b))
    r ⟵ new BigInt(n)
    c ⟵ 0
    for 0 ≤ i < n do
        (c,s) ⟵ DIGITSOF(GET(a,i)+GET(a,i)+c)
        SET(r,i,s)
    end for
    return r
end function
```

Again, once you have addition working, some of the tests in the test suite will start passing.

2.3 Subtraction

Subtraction is similar to addition, with borrowing instead of carrying. Implement subtraction $B_a - B_b$ as `Ba->Sub(Bb)` (C++) / `Ba.Sub(Bb)` (Java). You may assume that the subtrahend B_b is smaller than or equal to the minuend B_a. Verify that your implementation is correct by running the tests once more and seeing that the subtraction-related ones pass.

2.4 Multiplication

This part of the lab walks you through implementing schoolchild multiplication.

2.4.1 Shift

Implement `Shift()`, which returns a new big integer with the digits of the given Big Integer shifted by the specified number of places, effectively multiplying the original number by a power of 10. Run the tests to check your implementation.

2.4.2 Multiplication by a single digit

Implement `MulByDigit()`, which takes a digit between 0 and 9 and produces the Big Integer which is the product of the digit and this Big Integer. Once again, check your implementation using the provided tests.

2.4.3 Schoolchild multiplication

Combine the shift, multiply-by-digit and addition operations to implement schoolchild multiplication: choose one of the two Big Integers, loop over its digits, multiply each digit by the other Big Integer, shift the answer appropriately and add it to the result so far. Once this is correctly implemented, all the provided tests should pass.

2.5 Division and Remainder

Implement operations to compute the quotient `Div()` and the remainder `Rem()` of division of two Big Integers. You may assume that the divisor is smaller than or equal to the dividend. You will have to write your own tests for these methods.

2.6 [Optional] Karatsuba multiplication

Implement multiplication using the Karatsuba divide-and-conquer recursion shown in the lectures. Use an `OpCounter` to keep track of how many digit × digit multiplications you do for numbers with N digits, and compare that against the number of digit × digit multiplies the schoolchild method uses.

3 Code submission

Submit your work on Big Integers, along with answers to the optional extra questions, to the VLE submission area for this lab by **16:00** on **Friday 15th March**. As usual, you may submit more than once, and the highest-achieved score is retained.

4 Draft summaries submission

Your draft summaries, for peer-assessment, are due on the VLE Workshop activity by **16:00** on **Friday 15th March** (submissions are open from **16:00** on **Wednesday 6th March**). Failure to submit a draft will lead to an automatic mark of zero for both the submission and the assessment part of this activity.