Handout 1
Multiplication and Division

Construction and Verification of Software
FCT-NOVA
Bernardo Toninho

29 March, 2022

This handout is due on Sunday, April 16, at 23h59m. The exact details on how to turn in your solution will be made available at a later date, but you will only have to submit a single Dafny file with your answers.

The handout consists of two verification exercises where you will implement and verify two simple algorithms: one which computes the multiplication of two integers and another which computes the integer (or Euclidian) division of two integers. Simple enough. . . right?

1 Multiplication

The algorithm we have in mind is one which calculates the result of multiplying any two integers a and b (with b positive), only by performing addition, multiplication by two and division by two. Variations of this algorithm have existed since ancient Egypt, so here's the basic idea: to calculate $a \times b$ we instead compute $(2 \times a) \times \frac{b}{2}$, if b is even, and $a + (2 \times a) \times \frac{(b-1)}{2}$ if b is odd (where the fraction stands for real division). In both cases, the multiplier is going down by a factor of two or more, at the cost of one halving, one doubling and possibly one addition. Of course, the algorithm computes each successive multiplication using this approach, so that in the end we obtain our result while performing less work than the "traditional" approach.

This algorithm is often dubbed *Russian Peasant Multiplication* in the literature, although its origins likely pre-date its russian naming. As mentioned above, similar procedures have been found in historical documents from ancient Egypt and Ethiopia. Lets see how the algorithm works by trying it out with two large, but not too large, integers: lets calculate 93×39 using this approach. Since 39 is odd, we want to calculate 93×39 by calculating:

$$\mathbf{93} + (2 \times 93) \times \frac{38}{2} = \mathbf{93} + (186 \times 19)$$

Well, now we keep going to calculate 186×19:

$$186 \times 19 = \mathbf{186} + (2 \times 186) \times \frac{18}{2} = \mathbf{186} + (372 \times 9)$$

and again for 372×9:

$$372 \times 9 = \mathbf{372} + (2 \times 372) \times \frac{8}{2} = \mathbf{372} + (744 \times 4)$$

and again, but now note that our second operand is even:

$$744 \times 4 = (2 \times 744) \times \frac{4}{2} = 1488 \times 2 = (2 \times 1488) \times 1 = \mathbf{2976} + 0$$

What we did was just unroll our initial multiplication all the way until our second operand became 0, at which point we dont have any multiplications left. Putting it all together, with the conveniently highlighted numbers in bold:

$$93 \times 39 = \mathbf{93} + \mathbf{186} + \mathbf{372} + \mathbf{2976} = 3627$$

And so we have simplified our "complicated" multiplication down to a few additions (and some "easy" multiplications and divisions by two). In fact, if we assume that halving, doubling and addition are all constant-time operations we can show that the algorithm's running time is $\Theta(\log b)$, which is better than the traditional linear-time algorithm (or quadratic if we reason at the level of bits, using $\Omega(k)$ time to add k bits).

While the procedure above was described in a recursive fashion, it is not so hard to see how to turn it into an iterative procedure: effectively what we need to do to multiply a and b is repeatedly double a and halve b or $b - 1$, depending on whether b is even or odd, respectively. When b is even, we keep going, when b is odd, we add the current a to our running total. We stop this procedure once b reaches 0. Moreover, if we stick to using only integers and integer operations, we don't need to worry about the subtraction at all, just use integer division of b by 2 regardless.

Another way to understand what the algorithm is doing is to realize that the successive divisions by two are actually converting the number to *binary* and effectively multiplying each non-zero bit by 93, which produces the correct result due to the distributive property of multiplication. Lets see how it works: 39 is 100111 in base 2. If we expand back to decimal we have that $39 = 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$ and so:

$$93 \times 39 = 93 \times (2^5 + 2^2 + 2^1 + 2^0) = \mathbf{2976} + \mathbf{372} + \mathbf{186} + \mathbf{93}$$

Essentially, the cases where the remainder of the division by two is 1 correspond to a 1 bit, whereas no remainder corresponds to a 0 bit.

So there you have it – now all you need to do is prove this procedure is correct. Write a Dafny method:

```
method peasantMult(a: int, b: int) returns (r: int)
```

that implements the procedure described above, specifying and verifying the appropriate weakest pre- and strongest post-conditions. Note that for the sake of the verification, it is easier for the automation if you **don't** use negation in your code (i.e. don't test if the remainder of the division is different from zero, test what it should be equal to in that case).

Note that you will have to help Dafny with this verification, since it doesn't know outright that halving and multiplying preserves the value of multiplication. As indicated above, the property that makes this entire procedure work is (for positive b):

$$a \times b = \begin{cases} (2 \times a) \times \dfrac{b}{2} & \text{if } b \text{ is even} \\ a + (2 \times a) \times \dfrac{b-1}{2} & \text{if } b \text{ is odd} \end{cases}$$

Note: If you can't manage to convince Dafny that this property holds, you may *assume* it holds (remember that lemmas without bodies are assumed to hold in Dafny). However, if you do this you **will not get full credit** for the exercise.

As a side note, you may be wondering why this algorithm matters at all, especially when modern machines can multiply numbers in hardware. While this is definitely true, it is only the case for some fixed bit length representation of numbers (32 bits, 64 bits, etc.)! Sometimes, we need to work with arbitrarily large numbers and perform arithmetical operations on them (e.g. cryptography where we need to work with large primes, scientific computing, etc.). This is where these techniques kick in! While the algorithm above still relies on multiplication and division, it only uses a special case: halving and doubling,which can be implemented efficiently using other (specialized) techniques.

However, in reality, multiplication of arbitrary precision integers is not implemented using this algorithm. A common approach is to use different algorithms depending on the magnitudes of the numbers involved: if the numbers are "not too big" (e.g. less than 50 "words" in length), use the long multiplication algorithm, also known as the "grade school" algorithm (i.e. literally the one you learn in school), which is quadratic in the number of words or bits (just as the one above). For larger numbers, more sophisticated algorithms exist. The Karatsuba algorithm [KO62] is one such algorithm, the first shown to perform less than n^2 elementary operations to multiply two n-digit numbers ($O(n^{\log(3)})$ elementary operations). The algorithm uses a recursive divide-and-conquer approach. While asymptotically faster than long multiplication, the algorithm typically runs slower for small values of n, due to the use of extra additions and shifts.

For "intermediately large" numbers (e.g. above 240 words in length), Toom-Cook multiplication [Knu97] is generally the chosen algorithm, using $\Theta(n^{\frac{\log(5)}{\log(3)}}) \approx \Theta(n^{1.46})$ elementary operations. The algorithm uses much more sophisticated operations than Karatsuba multiplication and so only "catches up" for larger numbers. For really large numbers, the asymptotically faster Schönhage-Strassen algorithm [SS71] is the best choice ($\Theta(n \cdot \log n \cdot \log\log n)$). This algorithm uses Fast Fourier transforms in rings with $2^n + 1$ elements and outperforms Toom-Cook multiplication for numbers (approximately) beyond $2^{2^{15}}$ (over 9000 decimal digits).

2 Euclidian Division

Now that we know all about how multiplication was done in antiquity, lets think of division. In particular *euclidian* division. Given two integers a and b (with $b \neq 0$), one can prove that there exist unique integers q and r such that:

$$a = b \times q + r$$

and $0 \leq r <\mid b \mid$, where $\mid b \mid$ stands for the absolute value of b. In the statement above, a is called the dividend, b the divisor, q the quotient and r the remainder, with the computation of the quotient and the remainder from the dividend and the divisor called *Euclidian division*.

For the purposes of computation, we do not care about the uniqueness part of the statement. Lets think of a constructive proof of the above existence statement and see what kind of algorithm it suggests. First, observe that we need only consider strictly positive b and non-negative a: If b is negative, we consider $b' = -b$ and $q' = -b$ and rewrite the equation above as $a = b' \times q' + r$ and the inequality as $0 \leq r <\mid b' \mid$. If $a < 0$ we can consider $a' = -a$, $q' = -q - 1$ and $r' = b - r$ and perform a similar rewrite.

So, without loss of generality, let us consider $a \geq 0$ and $b > 0$. Let $q' = 0$ and $r' = a$. It is clearly the case that $a = b \times q' + r'$. If $r' < b$ then we are done, so lets consider $r' \geq b$. Then we can set $q'' = q' + 1$ and $r'' = r' - b$ and observe that $a = b \times q'' + r''$ with $0 \leq r'' < r'$. Since there are only r' non-negative integers less than r', we can repeat this process at most $'$ times to reach the final quotient and remainder, and so we have that for some $k \leq r'$, $a = b \times q_k + r_k$ and $0 \leq r_k < |b'|$.

Your task now is to prove this algorithm correct! That is, write a Dafny method:

```
method euclidianDiv(a: int,b : int) returns (q: int, r: int)
```

that implements Euclidian division. Write the appropriate pre- and post-conditions and show your algorithm satisfies its specification.

The algorithm described above is quite slow, since it performs $\Omega(a/b)$ elementary operations to compute its result (and is actually exponential if we think of costs at the level of bits). For the arbitrary precision case, Knuth's D algorithm [Knu97] is often used (a form of long division that is $O(n^2)$ at the bit level). For very large numbers (860 words or more), Burnikel-Ziegler recursive division [BZ98] is a better choice, with running time $2K(n) + O(n \log n)$ for division of a $2n$-digit number by an n-digit number, where $K(n)$ is the Karatsuba multiplication time.

References

[BZ98] Christoph Burnikel and Joachim Ziegler. Fast recursive division, 1998.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., USA, 1997.

[KO62] A. Karatsuba and Yu. Ofman. Multiplication of many-digital numbers by automatic computers. *Dokl. Akad. Nauk SSSR*, 145(2):293–294, 1962.

[SS71] Arnold Schönhage and Volker Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7(3-4):281–292, 1971.