1. Multiple Precision Arithmetic

We need to go back to school and remember how we did basic arithmetic! Recall that we first learnt how to operate with small numbers i. e. the digits. Then we learnt how to represent and manipulate bigger numbers. Thus for us "human"-sized symbols are the symbols 0-9 which we consider "small"; in a similar way the numbers 0–$2^{32}-1$ or the numbers made (in base 2) of thirty-two 0's and 1's are "small" for a computer. We must already know the following operations with these numbers; we use W to denote the set of basic symbols which represent natural numbers from 0 to $M-1$ ($M = 10$ for humans and $M = 2^{32}$ for computers):

addition: The operation of addition with carry

$$\texttt{add} : W \times W \times \{0,1\} \to W \times \{0,1\}$$

where $(a, b, \mu) \mapsto (c, \rho)$ which satisfies

$$a + b + \mu = c + \rho \cdot M$$

subtraction: The operation of subtraction with borrow

$$\texttt{sub} : W \times W \times \{0,1\} \to W \times \{0,1\}$$

where $(a, b, \mu) \mapsto (c, \rho)$ which satisfies

$$a - b - \mu = c - \rho \cdot M$$

multiplication: The operation of multiplication with remainder

$$\texttt{mul} : W \times W \to W \times W$$

where $(a, b) \mapsto (c, d)$ which satisfies

$$a \cdot b = c \cdot M + d$$

division: The operation of division with remainder (this is a partial function)

$$\texttt{div} : W \times W \times W \to W \times W$$

where $(a, b, c) \mapsto (d, e)$ when $b < a$ and satisfies

$$b \cdot M + c = a \cdot d + e$$

with $e < a$.

There may be some further operations that we may need to check whether on element of W is less than another or equal to another and so on.

The main aim of this section is to write down methods of computing with integers written in the "usual" way as strings of symbols $u_1 u_2 \cdots u_p$ representing $u_1 \cdot M^{p-1} + \cdots + u_1$. We will also count the number of steps taken to make our calculations under the assumption that each of the above operations counts as one step.

1.1. Addition and Subtraction Algorithm.

We now write the addition recipe using the usual method

$$\begin{array}{cccccc} & u_1 & u_2 & \cdots & u_p \\ & v_1 & v_2 & \cdots & v_p \\ \hline \rho & w_1 & w_2 & \cdots & w_p \end{array}$$

Or more concisely

$$(u_1 \cdots u_p) + (v_1 \cdots v_p) = (w_1 \cdots w_p) + \rho M^p = (\rho w_1 \cdots w_p)$$

The identities that are satisfied are

$$\begin{aligned}(w_p, \rho_0) &= \texttt{add}(u_p, v_p, 0)\\ (w_{p-i}, \rho_i) &= \texttt{add}(u_{p-i}, v_{p-i}, \rho_{i-1})\end{aligned}$$

We can clearly write these identities quite concisely by defining $\rho_{-1} = 0$. It is thus clear how to calculate this inductively for all i upto $i = p$. At the last stage the carry over is the extra $\rho = \rho_p$ in the answer.

A similar technique applies to the subtraction method. We write the identity

$$(u_1 \cdots u_p) - (v_1 \cdots v_p) = (w_1 \cdots w_p) - \rho M^p$$

And find the identities

$$\begin{aligned}(w_p, \rho_0) &= \texttt{sub}(u_p, v_p, 0)\\ (w_{p-i}, \rho_i) &= \texttt{sub}(u_{p-i}, v_{p-i}, \rho_{i-1})\end{aligned}$$

Again, putting $\rho_{-1} = 0$ combines the identities and we can easily compute these inductively. Note that $\rho = \rho_p$ is the "extra" borrow element and so if this is non-zero then $(v_1 \cdots v_p)$ is in fact more than $(u_1 \cdots u_p)$. We will see that this calculation is useful even in this case.

We note that the number of steps taken by both methods is equal to p.

1.2. **Multiplication Algorithm.** As is to be expected, multiplication is more complicated. The usual method of multiplication requires simultaneous summation of multiple rows of decimal numbers. The carry over from addition thus becomes quite large, whereas our basic routines can only handle a carry of 0 or 1. It is best to proceed slowly!

First of all we look at a procedure to multiply by a single "digit" u and perform the calculation, $u \times (v_1 \cdots v_p) = (w_0 w_1 \cdots w_p)$. Actually we will need the more general "linear form" operation (we will assume that $q \geq p + 1$ by padding t by zeros if necessary)

$$u \times (v_1 \cdots v_p) + (t_0 \cdots t_q) = (w_0 \cdots w_{q+1})$$

More diagrammatically,

			v_1	v_2	$\cdots$	v_{p-1}	v_p
$\times$							u
$+$	t_0	t_1	t_2	t_3	$\cdots$	t_{q-1}	t_q
		x_1	x_2	x_3	$\cdots$	x_p	
			y_1	y_2	$\cdots$	y_{p-1}	y_p
z_0	z_1	z_2	z_3	$\cdots$	z_{q-1}	z_q	z_{q+1}
w_0	w_1	w_1	w_2	$\cdots$	w_{q-1}	w_q	w_{q+1}

The usual multiplication procedures leads to certain intermediate digits which we denote by x, y, z as above. We obtain the identities

$$\begin{aligned}(x_p, y_p) &= \texttt{mul}(u, v_p)\\ (w_{q+1}, \rho_1) &= \texttt{add}(t_p, y_0, 0)\\ (z_q, \mu_1) &= \texttt{add}(t_{q-1}, x_p, 0)\\ (x_{p-i}, y_{p-i}) &= \texttt{mul}(u, v_{p-i})\\ (w_{q+1-i}, \rho_{i+1}) &= \texttt{add}(z_{q+1-i}, y_{p-i}, \rho_i)\\ (z_{q-i}, \mu_{i+1}) &= \texttt{add}(t_{q-i-1}, x_{p-i}, \mu_i)\end{aligned}$$

As before the first three steps are a special case of the latter three when we put $\rho_0 = 0 = \mu_0$ and $z_p = t_p$. There are $q+2$ cycles. The `mul` operation actually takes place only in the first p cycles (in the remaining cases x_j and y_j are zero). The last `add` operation does not take place on the last cycle. Thus, are at most $p + 2q + 3$ steps.

Now we use the above routine repeatedly to obtain the general linear form, (where as before we assume that $s \geq p + q + 1$)

$$(u_1 \cdots u_p) \times (v_1 \cdots v_q) + (t_0 \cdots t_{s-1}) = (w_0 \cdots w_s)$$

We see that this is achieved by intermediate computations of the form

$$\begin{aligned} u_p \times (v_1 \cdots v_q) + (t_0 \cdots t_{s-1}) &= (z_{0,p} \cdots z_{s,p}) \\ u_{p-i} \times (v_1 \cdots v_q) + (z_{0,p-i+1} \cdots z_{s-i,p-i+1}) &= (z_{0,p-i} \cdots z_{s-i,p-i}) \end{aligned}$$

Then $w_{s-i} = z_{s-i,p-i}$ since the $(s-i)$-th "digit" $z_{s-i,p-i}$ is not affected by subsequent computations. If we initialise $(z_{0,p+1} \cdots z_{s,p+1})$ to be equal to $(t_0, \cdots t_{s-1})$, then first computation is a special case of the second. We will perform p cycles, each a linear form as above of the length $q + 2s + 3$. Thus the number of steps is order of $pq + 2ps + 3p$ steps.

While this looks a bit complex, it is still *not* the most efficient when the p and q are large enough to allow us to use more (order linear in p and q) "book-keeping" steps in exchange for efficiency.

1.3. **Long Division Algorithm.** This is by far the most complicated procedure. The first point to note is that the school method for long division involves a "guess". We need to ensure that this guess is replaced a multiple-choice procedure (also characterised by a "case" statement). Moreover, the number of choices should be very small and not of the same size as the base M.

As for the case of multiplication we can understand some aspects of the relevant book-keeping by performing the "short" division

$$(v_1 \cdots v_q) = u \times (w_1 \cdots w_q) + t$$

where $t < u$. We first compute $\texttt{div}(u, 0, v_1) = (w_1, t_1)$. From then on we have the inductive approach $\texttt{div}(u, t_i, v_{i+1}) = (w_{i+1}, t_{i+1})$. Clearly, the first step is a special case of the second by putting $v_0 = 0$. At the end $t = t_q$. Moreover, there are exactly q steps.

There are two procedures for long division, each with its own level of complexity. We note that in the division

$$(v_0 \cdots v_q) = (u_1 \cdots u_q) \times w + (t_1 \cdots t_q)$$

where $v_0 < u_1$ and $(t_1 \cdots t_q) < (u_1 \cdots u_q)$, a "good guess" for w is provided by x which is obtained by $\texttt{div}(u_1, v_0, v_1) = (x, y)$ (so that $y < u_1$). The following lemma tells us just how good the guess is

Lemma 1. *If $2 \cdot u_1 + 1 \geq M$ then we have $x - 2 \leq w \leq x$.*

Proof. Using $(u_2 \cdots u_q) \leq M^{q-1} - 1$ and $(v_2 \cdots v_q) \leq M^{q-1}$ we obtain the following inequalities

$$\begin{aligned} u_1 \cdot M^{q-1} &\leq (u_1 \cdots u_q) \leq u_1 M^{q-1} + M^{q-1} - 1 \\ (v_0 \cdot M + v_1) \cdot M^{q-1} &\leq (v_0 \cdots v_q) \leq (v_0 \cdot M + v_1) \cdot M^{q-1} + M^{q-1} - 1 \end{aligned}$$

which we can combine with the Euclidean inequalities

$$\begin{aligned} u_1 \cdot x &\leq v_0 \cdot M + v_1 \leq & u_1 \cdot (x+1) - 1 \\ (u_1 \cdots u_q) \times w &\leq (v_0 \cdots v_q) \leq & (u_1 \cdots u_q) \times (w+1) - 1 \end{aligned}$$

We then obtain

$$\begin{aligned} u_1 \cdot M^{q-1} \cdot w &\leq (u_1 \cdots u_q) \times w \\ &\leq (v_0 \cdots v_q) \\ &\leq (v_0 \cdot M + v_1) \cdot M^{q-1} + M^{q-1} - 1 \\ &\leq (u_1 \cdot (x+1) - 1) \cdot M^{q-1} + M^{q-1} - 1 \\ &= u_1 \cdot (x+1) \cdot M^{q-1} - 1 \end{aligned}$$

As a consequence $w < (x+1)$ or equivalently (since these are integers) $w \leq x$. The reverse comparison is provided by

$$\begin{aligned} u_1 \cdot x \cdot M^{q-1} &\leq (v_0 \cdot M + v_1) \cdot M^{q-1} \\ &\leq (v_0 \cdots v_q) \\ &\leq (u_1 \cdots u_q) \times (w+1) - 1 \\ &\leq (u_1 M^{q-1} + M^{q-1} - 1) \cdot (w+1) - 1 \\ &= (u_1 + 1) \cdot (w+1) \cdot M^{q-1} - (w+2) \end{aligned}$$

It follows that $u_1 \cdot x < (u_1+1) \cdot (w+1)$ or (since we have integers on both sides) $u_1 \cdot x \leq u_1 \cdot (w+1) + w$. Now the condition $v_0 < u_1$ implies that $x \leq (M-1)$. So if $w \geq M-1$, then we certainly have the inequality $x \leq w+2$ as required. Thus we may assume that $w < M-1$. From the assumption $2 \cdot u_1 \geq M-1$ we obtain $w < u_1 \cdot 2$. Combining this with the above we see that

$$u_1 \cdot x \leq u_1 \cdot (w+1) + w < u_1 \cdot (w+3)$$

or equivalently $x \leq w+2$ as required. □

Using this lemma, we see that we can give an algorithm for the long division

$$(v_1 \cdots v_p) = (u_1 \cdots u_q) \times (w_0 \cdots w_r) + (t_1 \cdots t_s)$$

that involves making one out of three choices. In order to apply the lemma we must first normalise the divisor $(u_1 \cdots u_q)$. So as a first step we compute $\texttt{add}(u_1, 1, 0) = (t, \rho)$. If $\rho = 1$, then we take $d = 1$, other wise we compute $\texttt{div}(t, 1, 0) = (d, r)$. Thus in every case we have d is the integer part of $M/(u_1+1)$. We now take

$$\begin{aligned} (x_1 \cdots x_q) &= d \times (u_1 \cdots u_q) \\ (y_0 \cdots y_p) &= d \times (v_1 \cdots v_p) \end{aligned}$$

Here we take $y_0 = 0$ if necessary. We check easily that this step also ensures that $y_0 < x_1$. Next we initialise $(t_{0,0} \cdots t_{0,q-1}) = (y_0 \cdots y_{q-1})$. We can then perform the long division by induction as follows. Let $\texttt{div}(x_1, t_{i,0}, t_{i,1}) = (g_i, r)$; then g_i is our guess. We compute (and note that the z sequence cannot be longer than q due to the choice of g_i)

$$(t_{i,0} \cdots t_{i,q-1} y_{q+i}) - g_i \times (x_1 \cdots x_q) = (z_{i,0} \cdots z_{i,q-1}) - \rho \cdot M^q$$

If $\rho = 0$ then we take $(t_{i+1,0} \cdots t_{i+1,q-1}) = (z_{i,0} \cdots z_{i,q-1})$ and $w_i = g_i$. Otherwise (we have $\rho = 1$ and) we compute

$$(z_{i,0} \cdots z_{i,q-1}) + (x_1 \cdots x_q) = (\tilde{z}_{i,0} \cdots \tilde{z}_{i,q-1}) + \mu \cdot M^q$$

Now, if $\mu = 1$ then we take $(t_{i+1,0} \cdots t_{i+1,q-1}) = (\tilde{z}_{i,0} \cdots \tilde{z}_{i,q-1})$ and $\texttt{add}(g_i, 1, 0) = (w_i, 0)$. Finally, if we have $\mu = 0$, then we put

$$(\tilde{z}_{i,0} \cdots \tilde{z}_{i,q-1}) + (x_1 \cdots x_q) = (t_{i+1,0} \cdots t_{i+1,q-1}) + \mu' \cdot M^q$$

We note that $\mu' = 1$ is ensured by the lemma above. We put $\texttt{add}(g_i, 2, 0) = (w_i, 0)$ in this case. After $r = p - q$ steps we obtain the identity

$$(y_0 \cdots y_p) = (x_1 \cdots x_q) \times (w_0 \cdots w_r) + (t_{r+1,0} \cdots t_{r+1,q-1})$$

From the choices for x's and y's we see that d exactly divides the remainder, so we perform short division to obtain

$$(t_{r+1,0} \cdots t_{r+1,q-1}) = d \times (t_1 \cdots t_q)$$

This completes the long division which takes about $3rq$ steps upto terms linear in p and q (such as additions and subtractions).

Another way to simplify the guessing process is to take $(u_0 u_1 \cdots u_q)$ of the form $(10u_2 \cdots u_q)$. In this case, the division of $(v_0 \cdots v_q)$ always yields one of v_0 or $v_0 - 1$ as the quotient. The main question is then how to perform the necessary normalisation (to bring the u into this form). Starting with $(u_1 \cdots u_q)$ we first compute $\texttt{add}(u_1, 1, 0) = (t, \rho)$ and if $\rho = 1$ we put $d = 1$. Otherwise, we compute $\texttt{div}(t, 1, 1) = (d, r)$. Thus, in every case d is the integer part of $(M + 1)/(u_1 + 1)$. As before, we multiply by d

$$\begin{aligned} (x_{1,0} \cdots x_{1,q}) &= d \times (u_1 \cdots u_q) \\ (y_{1,0} \cdots y_{1,p}) &= d \times (v_1 \cdots v_p) \end{aligned}$$

Where we put $x_{1,0} = 0$ and $y_{1,0} = 0$ if there is no carry over. Now, if $x_{i,1} = 0$ (then $x_{i,0} = 1$ is forced) we stop. Otherwise, we perform the following steps

$$\begin{aligned} \texttt{sub}(0, x_{1,1}, 0) = (c_i, 1) \qquad & \texttt{add}(x_{1,1}), 1, 0) = (t_i, \rho) \\ \text{if } \rho = 0 \qquad & \texttt{div}(t_i, c_i, 0) = (d_i, r) \\ \text{if } \rho = 1 \qquad & d_i = c_i \end{aligned}$$

Thus in every case, we have d_i is the integer part of $M(M - x_{1,1})/(x_{i,i} + 1)$. Now, we multiply by $(1.d_i)$

$$\begin{aligned} (x_{i+1,0} \cdots x_{i+1,q}.x_{i+1,q+1} \cdots) &= (1.d_i) \times (x_{i,0} \cdots x_{i,q}.x_{i,q+1} \cdots) \\ (y_{i+1,0} \cdots y_{i+1,p}.y_{i+1,q+1} \cdots) &= (1.d_i) \times (y_{i,0} \cdots y_{i,q}.y_{i,q+1} \cdots) \end{aligned}$$

where we use the (.) to keep the notation simple and note that there are only finitely many terms after it. We now iterate over i.

Lemma 2. *For some $i \leq 3$ we obtain $(x_{i,0} x_{i,1} x_{i,2} \cdots) = (10 x_{i,2} \cdots)$ as required.*

Assuming this for the moment we see that have removed all `div` operations in the iterative part of the long division process. We leave it to the reader to make an algorithm out of this procedure. The proof of the lemma is a somewhat complicated exercise which we skip as well.

1.4. **Using the shift operations.** So far we have been discussing things in generalities but when we make some assumptions about the machine then the above algorithms can be improved or simplified. For example, in the first algorithm for division, when $M = b^n$ and there is a "left shift" operation on the symbol that amounts to multiplication by b, it is quicker to multiply iteratively by b to get something close enough to the required d. In this case division by d in the last step is also replaced by an appropriate "right shift".

On the other hand no such simplification is possible in the second algorithm for division, but (as is often true about "real" machines) if `div` is a significantly slower operation than the others, then it may be worthwhile to perform the extra steps.

Since most of our operations are on *binary* computers (so that $M = 2^n$), we have special procedures to multiply and divide by powers of 2. We now describe these procedures based on some new fundamental operations.

left shift: The operation of multiplication by a power of 2

$$\texttt{lshift} : W \times \{0, \ldots, n-1\} \to W \times W$$

where $(a, k) \mapsto (c, d)$ which satisfies

$$2^k \cdot a = c \cdot M + d$$

right shift: The operation of division by a power of 2

$$\texttt{rshift} : W \times \{0, \ldots, n-1\} \to W \times W$$

where $(a, k) \mapsto (c, d)$ which satisfies

$$a \cdot M/2^k = c \cdot M + d$$

zero: The operation of finding out whether something is non-zero; this is also the "bit-wise and" operation

$$\texttt{zero} : W \to \{0, 1\}$$

where $a \mapsto 0$ if a is zero and 1 if it is non-zero.

bitwise negation: The difference from $M - 1$; this is also the operation of bitwise negation

$$\texttt{bneg} : W \to W$$

where $a \mapsto M - a - 1$.

integer log to the base 2: The operation of finding the integer part of the logarithm to base 2

$$\texttt{log2} : W \to \{0, \ldots, n-1\}$$

where $a \mapsto k$ which satisfies $2^k \leq a < 2^k$. This also counts the number of right shifts possible without carry.

left shift count: The number of left shifts possible without carry.

$$\texttt{gol2} : W \to \{0, \ldots, n-1\}$$

where $a \mapsto k$ which satisfies $M/2 \leq 2^k a < M$.

We can use these operations to produce some special algorithms. We note that we have already used the `zero` operation without mention!

Given a small integer k and an integer $(u_1 \cdots u_q)$, we can construct $(v_0 \cdots v_q)$ satisfying $(v_0 \cdots v_q) = 2^k \cdots (u_0 \cdots v_q)$ as follows. We compute $\texttt{lshift}(u_i, k) = (s_{i-1}, t_i)$, $\texttt{add}(s_i, t_i, 0) = (v_i, 0)$ and $v_0 = s_0$. Similarly, we can easily compute division by 2^k. We can also find the precise power of 2 that divides any integer

$(u_1 \cdots u_q)$ (also called the 2-adic value of this integer). Clearly all these operations are of order linear in q. We can further extend this procedure quite easily to integers $(k_1 \cdots k_r)$ of size r with only r extra steps (to divide k by M).

Finally, we can also use the given basic operations to implement comparisons between integers which are linear in the size of the integers concerned.

We summarise our investigations in the following table

Operation	Time taken
Addition/Subtraction	Linear
Multiplication/Division by small integers	Linear
Multiplication/Division	Quadratic
Multiplication/Division by powers of 2	Linear
Comparison	Linear
2-adic value	Linear

Beyond this point we will not refer to the basic operations like `add` any more and make use of the higher level operations on integers mentioned above. If we do find a need for some specialised higher level operations to be implemented faster we should come back here and add to this section rather than complicate later sections!

1.5. **Faster Algorithms.** There are algorithms for multiplication and division that are asymptotically much faster than the above procedures; when multiplying numbers of size p we have taken p^2 steps above, which can asymptotically be reduced to a constant multiple of $p\log(p)$. However, the constant is large and it is likely that we will not need to look at those methods for the integers of the size we will be dealing with.

Similarly, we have taken $3p^2$ steps for division whereas division can be reduced to multiplication upto a constant factor and hence is again accomplished in a constant multiple of $p\log(p)$ steps.

It will probably suffice for cryptological applications to make the asymmetric assumption that the cryptographer (encryption/decryption process) uses multiplication that takes p^2 steps while the cryptanalyst (the "code-breaker") uses multiplication that takes $p\log(p)$ steps. If the cryptosystems that we design are secure under such asymmetric assumptions they will also be secure under more symmetric assumptions!

1.6. **GMP and other implementations.** The GNU multi-precision package implements the algorithms required to make the required computations. It provides a library of functions that can be used to initialise integers of arbitrarily large size and make computations with them. It not only implements the algorithms which we have described in detail but also the "faster" ones that we have not.

There are many other implementations both as libraries and as interpretive programs. Probably the program `gpari` is most suitable for our purpose. However, it is probably worthwhile to read through the code to understand how these things are implemented and also to spot bugs!