COURSEWORK 1 FOR INF2B: MULTIPLICATION OF LARGE INTEGERS
ISSUED: 8 FEBRUARY 2018

The coursework consists of two parts (of a different nature) relating to one problem. As shown below these have separate deadlines, you *cannot* swap the parts round or submit either part later than the stated deadline (unless you have permission to do so from the year organiser).

- Part 1 consists of Exercise 1 on page 12, §2.5, this involves fairly straightforward analysis.
  Submit Part 1 by: 4.00PM, 26 February 2018.
- Part 2 consists of the tasks in §3; these are all software related.
  Submit Part 2 by: 4.00PM, 9 March 2018.

You will be given the marks for each part, after the second part is marked the two marks will be added together to produce the overall total for this coursework (this is the mark that will be used for calculating the contribution of this coursework to your overall course mark once all other constituent marks are known). See §4 at the end of this document for instructions on how to submit.

§1. Introduction

In this practical we will consider two methods of multiplying integers and compare them by means of asymptotic analysis as well as implementations in Java. We will consider integers of very large size so that bare machine arithmetic is not sufficient. Naturally we cannot maintain the simplifying assumption made elsewhere in the course that arithmetic operations take constant time.

The practical has several aims:

1. Practice at asymptotic analysis (with guidance).
2. Careful implementation of one algorithm (an implementation of the other algorithm is supplied).
3. Carry out timing experiments in order to:
   (a) Compare the algorithm that you implement with one that is supplied and decide the point from which one is more efficient than the other (i.e., the overheads are overweighed by the advantages).
   (b) Determine the constant for the asymptotic analysis as it applies to the particular implementation.

You will find it very beneficial to read and understand the entire document before starting any further work. Note that although the document is fairly long the tasks you are required to carry out are quite modest. The emphasis here is on understanding the problem well and so plenty of discussion has been included.

In marking the exercise there will be a very high premium placed on answers that are clear, concise and correct[1]. This is a good time to have another look at the advice in Note 2 on Mathematical writing. The exercise is as much about giving you an opportunity to demonstrate your ability to express things appropriately as showing your technical mastery of the material. Please note that there are various ways to express an answer that fit the requirement, i.e., there isn't a unique form of expression (just as there are many ways to write a good program for a given task reflecting a good programmer's style). I recommend very strongly that you make an early start and produce a first draft version of your answer to each part, especially for the asymptotic analysis. Work this up into your final version then leave it for a day or so. After that read your proposed final version with an impartial view, pretend that your answer was part of the notes supplied for the course. Ask questions such as: Is the line of reasoning clear? Are claims justified? In answering these the appropriate attitude is that of a fair minded judge who does not wish to be overly picky but is also not going to be so lenient as to accept flawed arguments. As a practical matter if the marker cannot even start to follow the answer to a given part within 1 minute he will deem it to be incorrect and award it 0. The amount of work you have to submit has been kept moderate so that everybody has time to produce fluent answers.

Notes

Good Scholarly Practice: Please remember the University requirement as regards all assessed work for credit. Details and advice about this can be found at:

http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct

and links from there. Note that, in particular, you are required to take reasonable measures to protect your assessed work from unauthorised access. For example, if you put any such work on a public repository then you must set access permissions appropriately (generally permitting access only to yourself, or your group in the case of group practicals).

Following instructions: The various parts of this practical place certain requirements with a penalty if these are not followed. *No negotiation of any kind will be entered into where the requirements are not followed.* The point of the requirements is to ensure that in doing the various parts you practice and demonstrate skills relating to the appropriate areas of the course. As discussed above, what you submit must be considered work; imagine that your work was being given to you and others as a sample answer. If you, or others, would find it unclear and unhelpful then your work needs further revision.

Regrettably the various requirements and penalties can appear somewhat censorious and limiting. This is not the intention, indeed much of what is stated can be seen in the much more positive light of reassurance that the various tasks have quite simple answers.

§2. Multiplication of integers

We will simplify the discussion by assuming that our integers are all non-negative. This just saves us the trouble of carrying round a sign in some agreed way; a necessary feature of any application but one that has no significant effect on the nature of the problem discussed.

[1] I hope this handout is clear and correct. It is hardly concise but it does not consist of a answers to a simple questions, its aim is to explain various things leaving no gaps.

When representing integers we choose a convenient base $B \geq 2$ and then use *digits* in this base, a digit is a number d satisfying $0 \leq d < B$. Note carefully here that we have B digits, not necessarily just the usual 9 familiar with base 10 (we can notate the digits in any convenient way, e.g., as base 10 integers if all else fails[2]). The integer denoted by the sequence of digits $a_{n-1}a_{n-2}\dots a_0$ is precisely $a_{n-1}B^{n-1}+a_{n-2}B^{n-2}+\dots+a_0$; when there is no possible confusion about the base B it is not indicated in the pure digit notation, otherwise it is sometimes shown as a subscript, e.g., $(a_{n-1}a_{n-2}\dots a_0)_B$. We call a_{n-1} the *most significant digit* and a_0 the *least significant digit*. It is a simple matter to show that every integer has such a representation for any base $B \geq 2$. The most familiar situations are for $B = 10$ and $B = 2$ (in this second case we refer to *bits* rather than "digits," this is a convenient shorthand for indicating that we are working in binary).

Note that if we have two bases B_1, B_2 and $B_2 = B_1^r$ for some $r \geq 1$ then integers in base B_1 can be converted to base B_2 very easily. For example if $r = 2$ and $a = a_0 + a_1B_1 + a_2B_1^2 + a_3B_1^3 + \cdots$ in base B_1 then in base B_2 we have $a = (a_0 + a_1B_1) + (a_2 + a_3B_1)B_1^2 + \cdots = (a_0 + a_1B_1) + (a_2 + a_3B_1)B_2 + \cdots$. Thus if the digits in base B_1 are $a_0, a_1, \dots$ (starting with the least significant upwards) then the digits in base B_2 are $a_0 + a_1B_1, a_2 + a_3B_1, \dots$ As a consequence, if we have already fixed a notation for the digits in base B_1, say $d_0, d_1, \dots, d_{B_1-1}$ then we can use pairs of these as our notation for the digits of base $B_2 = B_1^2$, i.e., our base B_2 digits are notated as d_id_j for $0 \leq i, j \leq B_1 - 1$. More generally if $B_2 = B_1^r$ then for base B_2 digits we can use r-tuples of base B_1 digits. In this exercise our base will be a proper power of 10, thus our digits will *not* just be the usual ones $0, 1, \dots, 9$ but they will be appropriately sized tuples of them. [At this point it would be a good idea for you to think about how we could go from base B_2 to base B_1 assuming we have followed the suggested convention for notating base B_2 digits. The process is very easy.]

§2.1 The school method

We now consider the problem of multiplying two integers represented in base B. We will simplify matters further by assuming that the two integers have the same number of digits which we will denote by n (not really a restriction as we can always pad out the shorter integer with leading zeros though this can have a bad effect on efficiency for pairs of integers of significantly different lengths). Put

$$a = a_0 + a_1B + \cdots + a_{n-1}B^{n-1},$$
$$b = b_0 + b_1B + \cdots + b_{n-1}B^{n-1}.$$

(As we give the appropriate powers of the base we do not need to stick to the convention of writing digits from most significant to least significant in left to right order. The notation above reverses things; this makes some things below a little more natural, e.g., when we think of digits as being stored in an array.)

One way of multiplying the two integers is by the familiar method taught at school: multiply digit by digit, add them from least significant upwards and take care of the carry if there is one. We will proceed to give an indication of the general case but if you bear in mind the very familiar case of $B = 10$ you will see that there is nothing new or unusual here. We have

$$\begin{aligned} ab = a_0b_0 + a_1b_0B + a_2b_0B^2 + a_3b_0B^3 + \cdots + a_{n-1}b_0B^{n-1} & \quad + \\ a_0b_1B + a_1b_1B^2 + a_2b_1B^3 + \cdots + a_{n-2}b_1B^{n-1} + a_{n-1}b_1B^n & \quad + \\ a_0b_2B^2 + a_1b_2B^3 + \cdots + a_{n-3}b_2B^{n-1} + a_{n-2}b_2B^n + a_{n-1}b_2B^{n+1} & \quad + \\ \ddots \qquad\qquad & \quad \ddots \end{aligned}$$

(One reason for writing things with lowest digit first is that the preceding diagram is neater this way round.) To obtain the lowest digit we look at a_0b_0. If this satisfies $a_0b_0 < B$ then it is the digit. Otherwise we take the remainder r_0 when a_0b_0 is divided by B and this is the digit while the carry is the quotient q_0 (since we have $a_0b_0 = r_0 + q_0B$). The next digit is then obtained by considering $q_0 + a_0b_1 + a_1b_0$. Again if we have $q_0 + a_0b_1 + a_1b_0 < B$ we are done for this digit. Otherwise we find the remainder r_1 when $q_0 + a_0b_1 + a_1b_0$ is divided by B; this is the digit while the carry is the quotient q_1. The idea is now clear.

What is the cost of this familiar algorithm? Since B is fixed for any particular application it follows that arithmetic on a pair of digits takes constant time. We are therefore justified in simply counting the number of operations on digits (admittedly in any implementation there will need to be some further controlling logic but this does not contribute significantly to the runtime). Without going into details it can be seen that this is $\Theta(n^2)$. The fact that it is $\Omega(n^2)$ is obvious from the illustration above: each row involves n digit by digit multiplications and there are n rows (remember that the powers of B are there to indicate the place of each digit, we don't actually multiply by B at any stage). It is not hard to see that the remaining operations are $O(n^2)$.

An implementation of this approach is supplied as part of the practical, the relevant method is called `schoolMul`.

§2.2 The Karatsuba-Ofman algorithm

Can we do better than the school method? In asymptotic terms we can do considerably better. There is an algorithm due to Schönhage and Strassen [3] that takes $O(n \log n \log \log n)$ time[3]. Unfortunately the constant involved is very large and so systems do not use this method. In any case the techniques used are beyond the scope of this course[4]. We consider instead an older method due to Karatsuba and Ofman [1] that is very easily understood. Moreover the overheads are much smaller and the method is used in practice (once there are enough digits, see below). Note that in the literature the algorithm is normally referred to as "Karatsuba's algorithm."

Let as assume for now that n is even and put

$$a = \alpha_0 + \alpha_1B^{n/2}$$
$$b = \beta_0 + \beta_1B^{n/2}$$

[2] As another example hexadecimal (base 16) uses as digits 0–9 and A, B, C, D, E, F to denote the digits in base 16 that correspond to 10, 11, 12, 13, 14 ,15. Of course we can extend this to other bases but for large enough ones we run out of letters and must use a different convention.

[3] More recently algorithms that are asymptotically a bit faster have been designed by Martin Fürer as well as Anindya De, Chandan Saha, Piyush Kurur and Ramprasad Saptharishi.

[4] Aside from their intrinsic interest the algortihms by Schönhage and Strassen and others are of importance if we wish to prove a *lower bound* for the cost of integer multiplication, i.e., that all algorithms for the problem require at least a certain claimed runtime. No lower bound can have a growth rate larger than the runtime of any of the algorithms. In fact we do not have any lower bound that matches the best known upper bound.

where $\alpha_0 = a_0 + a_1 B + \cdots + a_{n/2-1} B^{n/2-1}$ and $\alpha_1 = a_{n/2} + a_{n/2+1} B + \cdots + a_{n-1} B^{n/2-1}$. Similarly for β_0 and β_1. In other words α_0, β_0 are the lower order digits of a, b respectively and α_1, β_1 are the upper order ones. Now

$$ab = \alpha_0\beta_0 + (\alpha_0\beta_1 + \alpha_1\beta_0)B^{n/2} + \alpha_1\beta_1 B^n. \qquad (\dagger)$$

This suggests a divide and conquer strategy: compute $\alpha_0\beta_0$, $\alpha_0\beta_1$, $\alpha_1\beta_0$, $\alpha_1\beta_1$, add the middle two results and then make any adjustments to digits due to carries. However this strategy is doomed to failure for the simple reason that although the four multiplications are on smaller numbers (of size $n/2$) each of them costs around $(n/2)^2$ and we have a total cost for these of $4(n/2)^2 = n^2$. As an aside it is very important that you develop your understanding to the point where you can spot such blind alleys without excessive work or the need for a very formal analysis.

The critical observation is that

$$\alpha_0\beta_1 + \alpha_1\beta_0 = (\alpha_0 + \alpha_1)(\beta_0 + \beta_1) - \alpha_0\beta_0 - \alpha_1\beta_1.$$

This helps because:

1. Computing $\alpha_0 + \alpha_1$ and $\beta_0 + \beta_1$ is cheap, around $2(n/2) = n$ operations; we just add digit by digit taking any carry into account.

2. The two numbers computed in the preceding steps have around $n/2$ digits (in fact at most $n/2 + 1$ if there is a carry at the highest digit).

3. The numbers $\alpha_0\beta_0$ and $\alpha_1\beta_1$ are already computed, see $(\dagger)$, so can be reused.

So, ignoring for now any linear terms, the overall cost is around $3(n/2)^2$. Well this is no better as an asymptotic runtime, though saving $(n/2)^2$ operations is of practical significance. The second crucial (and rather obvious) observation is to use the algorithm recursively. If we do this we make a significant saving at *each* stage, enough to ensure that the exponent of the runtime is significantly smaller than 2 (it is $\lg 3 \approx 1.585$, see Figure 1 for a comparison of n^2 with $n^{\lg 3}$). The cost of the three multiplications at the top level is of course no longer $3(n/2)^2$ but roughly $3T(n/2)$ where T is the runtime of the algorithm.

Let us now put all this together into a step by step description for multiplying a and b.

1. If $n = 1$ then return $a_0 b_0$ (there might be a carry to take care of, i.e., two digits are returned). Otherwise proceed as follows.

2. Define α_0 to be the integer obtained from the lower $\lfloor n/2 \rfloor$ digits of a and α_1 to be the integer obtained from the upper $\lceil n/2 \rceil$ digits, i.e.,

   $$a = \alpha_0 + \alpha_1 B^{\lfloor n/2 \rfloor}.$$

   (Think carefully about the power of B above, it is *not* an error.) Define β_0 and β_1 similarly with respect to b.

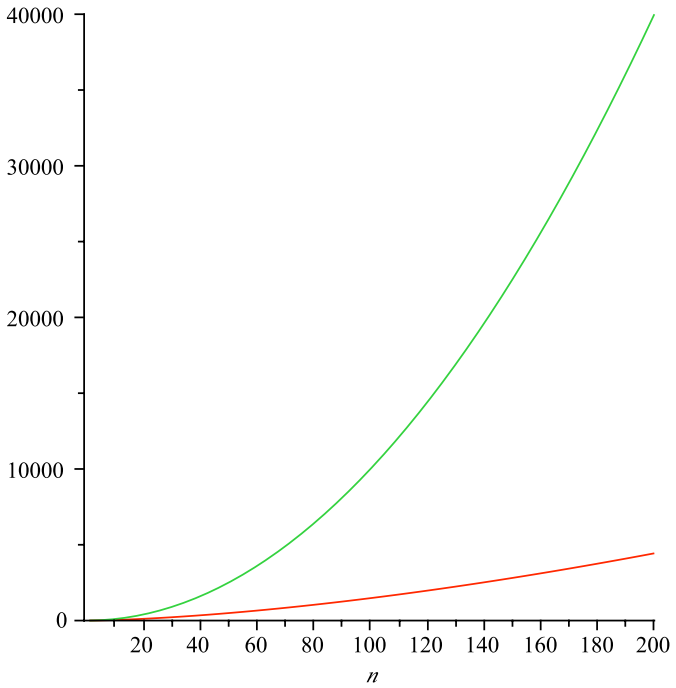

Figure 1: The graph of n^2 and $n^{\lg 3}$.

3. Compute recursively
$$l = \alpha_0\beta_0,$$
$$h = \alpha_1\beta_1,$$
$$m = (\alpha_0 + \alpha_1)(\beta_0 + \beta_1) - l - h.$$

4. Return $l + mB^{\lfloor n/2 \rfloor} + hB^{2\lfloor n/2 \rfloor}$.

§2.3 Operations on digits

Before proceeding further it is worthwhile thinking a little more about the most basic operations we have in this process: addition, subtraction and multiplication of pairs of digits. Clearly we need a way to add or multiply pairs of digits and obtain the resulting digit as well as any carry, similarly for subtraction though normally we think of "borrowing 1" (see below). As mentioned above these operations take constant time for a *given* base B. This follows since there are only B digits so only $B(B+1)/2$ pairs to consider for addition and multiplication [why?]. For subtraction the situation is slightly different but in any case the number of pairs is just B^2 [why?]. Thus we could, if desired, implement the operations by a look up table (however we must take any carry into account so the tables need to be adjusted for this). Note also that, with our representation, multiplying an integer by B^s is achieved by performing s right shifts of the digits[5].

Suppose we have two digits d_1 and d_2 (i.e., $0 \le d_1, d_2 \le B-1$). Let us examine what can happen in adding or multiplying them.

Addition: We have $0 \le d_1 + d_2 \le 2B - 2 < 2B - 1$. If $d_1 + d_2 < B$ there is no carry (or, for uniformity, the carry is 0) and the digit is $d_1 + d_2$.

On the other hand if $d_1 + d_2 \ge B$ then we know that $0 \le d_1 + d_2 - B < B - 1$ so that the digit is $d_1 + d_2 - B$ and the carry is 1 (i.e., we have $d_1 + d_2 = (d_1 + d_2 - B) + 1 \cdot B$).

Subtraction: In this application we always subtract an integer from a larger one. However at the level of digits we might well have to subtract a larger digit from a smaller one. For example consider $51 - 26$ in base 10; all we do is "borrow" 1 so that instead of subtracting 6 from 1 we subtract 6 from 11 giving us 5 as the first digit. Now we must "pay back" the borrowed 1 so we subtract 3 (i.e., $2 + 1$) from 5 giving us 2 (all this amounts to is the fact that $51 = 5 \cdot 10 + 1 = (5-1) \cdot 10 + 11$, we take care of subtracting 1 from 5 by adding the "borrowed" 1 onto the first digit of 26, the number we are subtracting). Thus the answer is 25. Rather than introduce new terminology we will regard the borrowed digit as a carry and this is fine so long as we agree that we always add the carry to the second digit of the operation. In other words if d_1, d_2 are the current digits and c is the carry then we think of addition and subtraction as $d_1 + (d_2 + c)$ and $d_1 - (d_2 + c)$ respectively. Of course for addition it does no harm to add the carry to d_1 but why not keep things simple?

[5] In saying "right shifts" we are assuming a representation that puts the least significant digit first then the next one and so on. This is different from the normal written order but corresponds to the natural representation by arrays which we will use in the software.

Multiplication: We have $0 \le d_1 d_2 \le (B-1)^2$. If $d_1 d_2 \le B - 1$ then as before the digit is $d_1 d_2$ and the carry is 0. However if this fails to be the case we cannot proceed simply as before since we might have $d_1 d_2 - B \ge B$. Well all this means is that we need to take some more away from $d_1 d_2$, let's say it has to be done q times leaving us with $r = d_1 d_2 - qB$ where $0 \le r \le B-1$. How large can q be? Since $d_1 d_2 = r + qB \le (B-1)^2$ and $r \ge 0$ it follows that $q \le (1 - 1/B)(B-1)$ and so $q < B - 1$. What this discussion has shown is the intuitively obvious fact that when multiplying two digits we have to carry at most $B - 2$, i.e., the carry is just to the next digits along (i.e., power of the base B) but no further.

We note here that we can do better when finding the carry than the obvious method of repeatedly subtracting B. Even so this is constant time since everything is bounded by B^2 which is a constant. As mentioned above one approach is to use a lookup table. However if the base B is large the lookup table approach is not so attractive, there are efficient ways of "guessing" the quotient so that it is within 1 of the correct result, we can then check and adjust if necessary.

Note that, for this practical, arithmetic on digits is provided for you.

§2.3.1 Representing large integers

Having fixed our base B there is a very obvious data structure that can be used to represent integers:

1. Keep a record of the base B.

2. Keep the digits $a_0, a_1, \ldots, a_{n-1}$ in an array that is large enough to hold this many digits (we use dynamic arrays).

There are two implications of this:

1. The base B should not be so large that we cannot hold it in machine arithmetic, e.g., no larger than a computer word.

2. The digits should not be so large that they cannot be held in a single array location. Fortunately this follows automatically from the first requirement.

In general the base B is chosen to be either a power of 10 or a power of 2. The advantage of the first possibility is that reading in and printing out results requires very little processing. the disadvantage is that machines are more efficient with powers of 2, at least for now. Using powers of 2 reverses the situation. There is no forced choice here and it depends on intended usage. Whatever base is used it makes sense to ensure that B^2 can be carried out in machine arithmetic; this avoids various complications when optimising things (e.g., no overflow flags to worry about).

As discussed in §2, whatever base is used we choose some method for representing the digits. When the base B is no more than 10 we can just use the familiar decimal digits from 0 to $B - 1$. Otherwise if the base is not too large we can invent new digit names (e.g., for base 16 we use

A, B, C, D, E, F for 10, 11, 12 ,13, 14 ,15). However for a large base this is clearly impractical. Generally we use digits that are in fact numbers in an appropriate smaller base. Thus if our base is 10^4 we can use as digits normal base 10 numbers with at most 4 digits. As observed above, this makes conversion between base 10 and base 10^4 very simple.

For this practical the base used is

$$B = 10^4.$$

In fact you do not need to know this value for your software. It is provided partly for the sake of interest but more importantly because it determines the digits that can be used in input for testing (we discuss this below).

The most obvious way to implement the algorithm given above is to create new data structures to hold α_0, α_1, β_0, β_1 and then make the recursive calls. This does have implications for efficiency. One possibility is to reserve a certain amount of "scratch space" that gets reused. For this practical we will keep things simple and face the penalty; our interest is to carry out some experiments and learn from them rather than provide the most efficient implementation (in which case we would not be using Java).

Finally note that for this practical you will handle all operations on integers as part of an abstract data structure (again there is an efficiency penalty). The implementation is as discussed but you do not need to know this in order to write your software.

§2.4 Simulation of the Karatsuba-Ofman algorithm

At this stage it would be a very good idea to simulate the Karatsuba-Ofman algorithm presented at the end of §2.2 with moderate size integers in base 10. Most of the stages of one simulation are presented below, you should fill in at least one or two of the gaps for the latter part.

Consider $a = 31421$ and $b = 13546$. We will move to a representation that corresponds to the data structure used as this gives us a much better feeling for the way the algorithm works when implemented. We have:

$$a = \langle 1, 2, 4, 1, 3\rangle,$$
$$b = \langle 6, 4, 5, 3, 1\rangle.$$

Recall that in going from conventional digit notation to our data structure we reverse the digits (and we must remember to do this also at the end when we have obtained the answer and want to present it in conventional notation). Finally note that we have abused notation by equating a and b with their representations, well there are limits to pedantry (and in any case the same is true when we write $a = 31421$ and $b = 13546$).

Enter 1: [Top level call.] Here $n = 5$ so $\lfloor n/2\rfloor = 2$. Let

$$\alpha_0 = \langle 1, 2\rangle, \qquad \alpha_1 = \langle 4, 1, 3\rangle,$$
$$\beta_0 = \langle 6, 4\rangle, \qquad \beta_1 = \langle 5, 3, 1\rangle.$$

Enter 1.1: [Recursive call on $\alpha_0 = \langle 1, 2\rangle$, $\beta_0 = \langle 6, 4\rangle$] Here $n = 2$ so $\lfloor n/2\rfloor = 1$. Let

$$\alpha_0 = \langle 1\rangle, \qquad \alpha_1 = \langle 2\rangle,$$
$$\beta_0 = \langle 6\rangle, \qquad \beta_1 = \langle 4\rangle.$$

Enter 1.1.1: [Recursive call on $\alpha_0 = \langle 1\rangle$, $\beta_0 = \langle 6\rangle$] Base case, `return` $\langle 6\rangle$.

Back to 1.1: $l = \langle 6\rangle$.

Enter 1.1.2: [Recursive call on $\alpha_1 = \langle 2\rangle$, $\beta_1 = \langle 4\rangle$] Base case, `return` $\langle 8\rangle$.

Back to 1.1: $h = \langle 8\rangle$.

Enter 1.1.3 [Recursive call on $\alpha_0 + \alpha_1 = \langle 3\rangle$, $\beta_0 + \beta_1 = \langle 0, 1\rangle$] Here $n = 2$ so $\lfloor n/2\rfloor = 1$. Let

$$\alpha_0 = \langle 3\rangle, \qquad \alpha_1 = \langle 0\rangle,$$
$$\beta_0 = \langle 0\rangle, \qquad \beta_1 = \langle 1\rangle.$$

Enter 1.1.3.1 [Recursive call on $\alpha_0 = \langle 3\rangle$, $\beta_0 = \langle 0\rangle$] Base case, `return` $\langle 0\rangle$.

Back to 1.1.3 $l = \langle 0\rangle$.

Enter 1.1.3.2 [Recursive call on $\alpha_1 = \langle 0\rangle$, $\beta_0 = \langle 1\rangle$] Base case, `return` $\langle 0\rangle$.

Back to 1.1.3 $h = \langle 0\rangle$.

Enter 1.1.3.3 [Recursive call on $\alpha_0 + \alpha_1 = \langle 3\rangle$, $\beta_0 + \beta_1 = \langle 1\rangle$] Base case, `return` $\langle 3\rangle$.

Back to 1.1.3 $m = \langle 3\rangle - \langle 0\rangle - \langle 0\rangle = \langle 3\rangle$.
`return` $l + mB + hB^2 = \langle 0\rangle + \langle 0, 3\rangle + \langle 0\rangle = \langle 0, 3\rangle$.

Back to 1.1: $m = \langle 0, 3\rangle - l - h = \langle 0, 3\rangle - \langle 6\rangle - \langle 8\rangle = \langle 6, 1\rangle$.

$$\texttt{return } l + mB + hB^2 = \langle 6\rangle + \langle 6, 1\rangle B + \langle 8\rangle B^2$$
$$= \langle 6\rangle + \langle 0, 6, 1\rangle + \langle 0, 0, 8\rangle$$
$$= \langle 6, 6, 9\rangle$$

Back to 1: $l = \langle 6, 6, 9\rangle$.

Enter 1.2: [Recursive call on $\alpha_1 = \langle 4, 1, 3\rangle$, $\beta_1 = \langle 5, 3, 1\rangle$] Here $n = 3$ so $\lfloor n/2\rfloor = 1 \ldots$

$$\vdots$$

Back to 1.2: $l = \langle 0, 2\rangle$.

$$\vdots$$

Back to 1.2: $h = \langle 3, 0, 4\rangle$.

$$\vdots$$

Back to 1.2: $m = \langle 7, 0, 2\rangle$.

$$\texttt{return } l + mB + hB^2 = \langle 0, 2\rangle + \langle 7, 0, 2\rangle B + \langle 3, 0, 4\rangle B^2$$
$$= \langle 0, 2\rangle + \langle 0, 7, 0, 2\rangle + \langle 0, 0, 3, 0, 4\rangle$$
$$= \langle 0, 9, 3, 2, 4\rangle$$

Back to 1: $h = \langle 0, 9, 3, 2, 4\rangle$.

Enter 1.3: [Recursive call on $\alpha_0 + \alpha_1 = \langle 5, 3, 3\rangle$, $\beta_0 + \beta_1 = \langle 1, 8, 1\rangle$] Here $n = 3$ so $\lfloor n/2\rfloor = 1 \ldots$

$$\vdots$$

Back to 1.3: $l = \langle 5 \rangle$.

$$\vdots$$

Back to 1.3: $h = \langle 4, 9, 5 \rangle$.

$$\vdots$$

Back to 1.3: $m = \langle 3, 2, 1 \rangle$.

$$\begin{aligned}\texttt{return } l + mB + hB^2 &= \langle 5 \rangle + \langle 3, 2, 1 \rangle B + \langle 4, 9, 5 \rangle B^2 \\ &= \langle 5 \rangle + \langle 0, 3, 2, 1 \rangle + \langle 0, 0, 4, 9, 5 \rangle \\ &= \langle 5, 3, 6, 0, 6 \rangle\end{aligned}$$

Back to 1: $m = \langle 5, 3, 6, 0, 6 \rangle - \langle 6, 6, 9 \rangle - \langle 0, 9, 3, 2, 4 \rangle = \langle 9, 7, 2, 7, 1 \rangle$.

$$\begin{aligned}\texttt{return } l + mB^2 + hB^4 &= \langle 6, 6, 9 \rangle + \langle 9, 7, 2, 7, 1 \rangle B^2 + \\ &\qquad \langle 0, 9, 3, 2, 4 \rangle B^4 \\ &= \langle 6, 6, 9 \rangle + \langle 0, 0, 9, 7, 2, 7, 1 \rangle + \\ &\qquad \langle 0, 0, 0, 0, 0, 9, 3, 2, 4 \rangle \\ &= \langle 6, 6, 8, 8, 2, 6, 5, 2, 4 \rangle\end{aligned}$$

So the algorithm tells us that $31\,421 \times 13\,546 = 425\,628\,866$ which is indeed the case.

Such simulations not only illustrate the overall workings of the algorithm but can also bring some issues out. For example it is tempting to assume and perhaps insist that the two input numbers both genuinely have n digits (i.e., we have not counted some leading zeros as digits for one of the numbers). The simulation shows that even if this is the case at the top level call there might well be points in the recursion where one number has more digits than the other. Of course the algorithm is perfectly correct under these conditions. The lesson to draw is that when implementing the algorithm we should look at the number of digits for each integer and take the larger of the two as the value of n. There is of course an issue of efficiency, in a more sophisticated version we would avoid this padding. However this complicates the algorithm and its implementation, worth doing if we intend to rely on it heavily.

Another point to note is that at times we make a recursive call even though one of the numbers to be multiplied is 0. Naturally this could be avoided very easily. However there is a fine judgement to be made when trying to catch out special cases: does the cost of carrying out a test every single time outweigh the savings made? The answer depends on the cost of the test, the frequency of its success and the cost saved on those occasions when it does succeed. In fact the simulation also makes it clear that when numbers involve only a few digits the overheads are very significant and will outweigh any savings in digit by digit multiplications. It therefore makes sense to use a hybrid algorithm with a break point for a certain number d of digits. If the integers have strictly more than d digits we use the Karatsuba–Ofman approach, otherwise we switch to the school method. This is the strategy employed by applications that are in use. You will explore both (there is very little extra cost in terms of code). Naturally the asymptotic analysis is not affected but the constants are.

Finally, in the simulation we have not followed the details of addition or subtraction. Had we done so there would have been far too much detail to follow thus obscuring the main structure. If

we did feel the need to simulate the missed out operations it would be best to do them on separate examples as stand alone simulations. Choosing the granularity of a simulation is important if we are to benefit from the process. Essentially we want to illustrate the key structural aspects rather than incidental details.

§2.5 Asymptotic analysis

This is the "pencil and paper" part of the practical.

Exercise 1. Let $T(n)$ denote the worst case runtime of the Karatsuba-Ofman algorithm on inputs of size n.

1. Consider the step $m = (\alpha_0 + \alpha_1)(\beta_0 + \beta_1) - l - h$ of the algorithm (see page 7). Prove that the cost of this step is $T(\lceil n/2 \rceil) + \Theta(n)$.

   You may assume without proof that $T(m+1) = T(m) + \Theta(m)$, for all m. [20%]

2. Prove that $T(n)$ satisfies the recurrence

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1; \\ T(\lfloor n/2 \rfloor) + 2T(\lceil n/2 \rceil) + \Theta(n), & \text{otherwise.} \end{cases}$$

   [20%]

3. Use an appropriate theorem from the course (which you should identify but need not state) to obtain the asymptotic runtime of the algorithm; explain carefully how you obtained the answer (i.e., show your working). [10%]

Note: You *cannot* ignore floors and ceilings in the first two parts above.

If you have time it would be a good idea to consider the hybrid algorithm that switches to the school method for integers with at most d digits. Do you expect the asymptotic runtime to be different? Write down the recurrence for $T(n)$ in this case and solve it. Note that this is not part of the assessment so do not submit anything for this.

Before moving on to the software tasks we note that the Karatsuba-Ofman approach can be refined further to decrease the exponent as close to 1 as we like. This is carried out by Knuth [2] who shows:

Theorem For each $\epsilon > 0$ there is an algorithm for multiplying integers of length n in time proportional to $n^{1+\epsilon}$.

Note that this is still not as fast (asymptotically) as the Schönhage–Strassen algorithm [why?]. Moreover the smaller we make ϵ the greater are the (constant) overheads.

§3. Software tasks

We first give a guide to some of the suppled classes and methods (some more are mentioned later).

§3.1 Supplied software

The files that you will need for this coursework can be found on the coursework webpage:

`http://www.inf.ed.ac.uk/teaching/courses/inf2b/coursework/cwk1.html`

This page contains a file called `inf2bcw1.tar` which you should download. The file is tarred so you need to extract it, e.g. by `tar -xf inf2bcw1.tar`, this will create a subdirectory `src` of your current one that contains the software. The `java` package for this coursework is `package imult`. The only class that should be changed is the nearly empty class in the file `StudentCode.java` which is where you will implement your part of the project. In fact the amount of code you need to write is fairly modest.

There is a supplied class `BigInt` that represents large integers in base B (which you cannot change). In most examples below we will assume that $B = 10$ just to keep things simple.

Recall that we are using dynamic arrays to represent integers but this need not concern you directly since we abstract away from it. In the illustrative examples the integer $a_{n-1}B^{n-1} + a_{n-2}B^{n-2} + \cdots + a_0$ will be denoted by $\langle a_{n-1}, a_{n-2}, \cdots, a_0\rangle$, i.e., following the normal written order[6]. It could also be represented by $\langle 0, \ldots, 0, a_{n-1}, a_{n-2}, \ldots, a_0\rangle$, this accords with normal notational practice where leading zeros are usually discounted (except for the number 0 itself of course, we leave at least one digit). Note that we "index" our digits with the least significant one having index 0 which occurs at the right hand end of this purely local notation; this is only important for following the examples. In the unlikely event of an integer (i.e., dynamic array) being too large an exception will be thrown; you do not need to handle this.

The supplied methods of class `BigInt` are :

- `public BigInt(),`
  `public BigInt(int n),`
  `public BigInt(String s),`
  `public BigInt(List<Unsigned> l)`

  Constructors to create an empty BigInt object, initialise a new BigInt object with n random digits, load a new BigInt with digits from a string (the digits must be separated by spaces), or initialise a BigInt object from a pre-existing digit list respectively.

  `Unsigned` is a (class) type for non-negative integers whereas `int` is the base Java type for integers.

- `public BigInt split(Unsigned r, Unsigned s)`

  Returns a new structure consisting of the integer we obtain by taking the digits of `a` starting from position `r` and ending at `s` (using these as the first, second etc. digits of the new integer). An exception is thrown if the parameters do not satisfy $0 \leq \mathtt{r} \leq \mathtt{s}$. For example if the digits of a BigInt a are

  $$\langle 1, 4, 0, 6, 3, 1, 5, 0, 0\rangle$$

[6] It is true that in §2.4 we used the array representation order. It is good training to be able to switch between different conventions and appreciate that in this context there is nothing to be concerned about, you are working with abstract data structures after all.

then a`.split(2,5)` returns a new BigInt object with digits

$$\langle 6, 3, 1, 5\rangle.$$

- `public Unsigned getDigit(int d)`

  Returns the digit of `a` at position `d`. For example if the digits of a are

  $$\langle 0, 1, 0, 4, 3, 0, 0, 0, 2\rangle$$

  then the result of a`.getDigit(0)` is `2`, the result of a`.getDigit(5)` is `4` and the result of a`.getDigit(6)` is `0`.

  Note that if d is larger than the length of `a` the method returns 0, this is consistent with the fact that inserting leading zeros does not change the integer.

- `public int length()`

  If all digits are 0 then 0 is returned; arguably it should return 1 since the length of the *representation* of the number 0 is 1 but this is not important here. Otherwise the number of digits up to the highest non-zero digit is returned. For example if the digits of a are

  $$\langle 0, 1, 0, 3, 0, 0, 0\rangle$$

  then the result of a`.length()` is 6.

- `public void lshift(int n)`

  This shifts the digits of a by n places to the left, filling places on the right with 0. As an example if the digits of a are

  $$\langle 1, 0, 2, 5\rangle$$

  then after a`.lshift(3)` the digits of a are

  $$\langle 1, 0, 2, 5, 0, 0, 0\rangle.$$

- `public void inputDigits(String s)`

  This function will clear and initialise the BigInt objects with digits from the string parameter `s`. This method can be invoked by calling the `BigInt` string constructor.

  A helper method in the `Utils` class,

  `public static BigInt[] readDigitFile(File f),`

  can be used to read in saved digits from a file and returns them in as an array of BigInt objects. The convention we use is that base B digits are supplied in decimal each digit being a sequence of at most 4. Such base B digits are separated by one or more spaces (or newline) and the sequence is terminated with a semicolon (";"). Examples of valid input are:

```
6845 1023 000 012 556;
7700 543 0 012 376;
76 3069 000 7763
     887 1999     1  23;
```

Thus blocks do not have to be of size exactly 4, smaller ones are padded with 0 at the left. An exception is raised if an invalid character is met or a sequence of consecutive decimal digits is longer than 4.

- `public void randomDigits(int n)`

  Generates a random big integer with `n` digits (the most significant digit is guaranteed to be non-zero). `randomDigits(..)` can be invoked by using the `BigInt(int n)` integer constructor.

- `public void print()`

  Print the BigInt as an integer.

- `public void dump()`

  Dumps the contents of the BigInt array as stored internally.

§3.2 Tasks

Note: You are asked to write some code that is in fact fairly simple thanks to the supplied methods. *Any code that is incorrect will be awarded 0*. Correct code will be marked for style as well. This means that it should have useful comments and helpful indentation. Note that pointless comments (e.g., stating the obvious such as "now we add the two variables together") or excessive indentation are almost as bad as the complete absence of either.

Important: The checking of your code includes an automated procedure. Partly for this reason, you *must* follow the specification exactly and must not change any of the method names or their parameters. Otherwise your code will fail the test and be awarded 0.

Your programing tasks are as follows.

1. In the `StudentCode` class, provide implementations of the methods

   - `public static BigInt add(BigInt a, BigInt b)` [10%]
   - `public static BigInt sub(BigInt a, BigInt b)` [10%]

   For the second method it can be assumed that whenever it is called we have $a \geq b$ so that the result is never negative. In implementing these you will need appropriate methods from the supplied class `Arithmetic`:

   - `public static DigitAndCarry addDigits(Unsigned a, Unsigned b, Unsigned c)`

   - `public static DigitAndCarry subDigits(Unsigned a, Unsigned b, Unsigned c)`
   - `public static DigitAndCarry mulDigits(Unsigned a, Unsigned b)`

   The returned instance of the class `DigitAndCarry` has methods called `getDigit()`, `getCarry()`, `setDigit()` and `setCarry()` which do the obvious thing. For the first two methods `c` is the carry (from a previous operation). Remember our convention discussed in §2.3 regarding carries and the subtraction of digits. Note that `subDigits` subtracts `b+c` from `a`.

   - `public static BigInt StudentCode.koMul(BigInt a, BigInt b)` [15%]

   Keep your code straightforward, follow the algorithm as outlined in §2.2 just before §2.3.1. Note that here you *cannot* assume that the input integers have even length (this would do no good anyway unless their length was a power of 2). You must not assume that they have the same length, just take the largest length as being common to both; recall the discussion in §2.4. Note also that in the unlikely event of an integer being too large an exception will be thrown; you do not need to handle this.

   Test your implementation by using the following methods in the `BigIntMul` class:

   - `public static Boolean mulVerify(BigInt a, BigInt b, BigInt m)`

     Returns `true` if $m = ab$ otherwise `false`.

   - `public static void mulTest(Unsigned t, Unsigned l)`

     Generates `t` random pairs of `BigInt`s of length `l` and multiplies them using the method `koMul`. If all results are correct then it returns an appropriate message. Else it reports on the failure.

2. Provide an implementation of the method

   - `public static BigInt StudentCode.koMulOpt(BigInt a, BigInt b)` [5%]

   Here the Karatsuba–Ofman algorithm is used to multiply integers where the shortest one has strictly more than 10 digits, otherwise the school method is used. The `Arithmetic` class contains an implementation of `schoolMul`, the school method for multiplication (same parameters as `koMul`). All you need to do here is to copy your code for `koMul` and add a bit of control. Of course we could combine the two pieces of code but let's keep things simple (less chance of confusion between experiments).

   Test your code by using the method `mulVerify`. You should *not* include any code you write for your tests as part of your submission.

3. The aim of this part is to find out when your implementations of `koMul` and `koMulOpt` are more efficient than `schoolMul`. We would expect that initially `schoolMul` has the advantage because it has fewer overheads but from some point on this is not the case. In this part you will determine when this happens.

   The supplied `BigIntMul` class method

   - `public static void getRuntimes(Unsigned m, Unsigned n, Unsigned t, File f, Boolean opt)`

   does the following:

   (a) Generates `m` pairs of size `n` (the pairs are *not* random so that experiments are repeatable).

   (b) Multiplies each pair using both `schoolMul` and `koMul` if `opt` is false, otherwise it uses `koMulOpt` instead of `koMul`, and takes the cpu times for these.

   (c) Records the worst case times over these pairs (note that it is pefectly possible that the worst case runtimes for the two algorithm implementations occur for different pairs).

   (d) Repeats the above with pairs of size $2n, 3n, \ldots, tn$.

   (e) Outputs the result for each iteration on the file `f` in the format

   integer-size `schoolMul-`*worst-case-runtime* `koMul-`*worst-case-runtime*

   or

   integer-size `schoolMul-`*worst-case-runtime* `koMulOpt-`*worst-case-runtime*

   (as appropriate) one line at a time; note that these figures are in standard base 10 notation.

   Carry out experiments with increasing values of `t` until you find the point at which `koMul` is more efficient than `schoolMul`. Likewise for `koMulOpt`. In order to avoid excessive waiting choose the value of `m` to be 1 and the value of `n` to be 20. Naturally it makes sense to continue the experiment a little beyond the crossover point (indeed due to various factors there might be a temporary turn around, this doesn't happen for long). Normally doing only one experiment per input size is unwise. However our algorithms go through pretty much the same number of steps for all inputs of the same size. You could check what happens by changing the value of `m`. Do bear in mind the usual fly in the ointment with Java that it might decide to carry out garbage collection thus producing an overly large figure for one run.

   Task: run the experiment for `koMulOpt` with `m=1`, `n=10`, `t=90` and keep the results in a file called `koMulOptTimes`[7]. [5%]

   From now on we will focus on `koMulOpt` and `schoolMul`.

[7] You are likely to see a trend for about half the 'digits' axis and then a sudden drop in runtime for both algorithms with the trend resuming. This drop is due to Java applying some optimization to its runtime environment which is not within user control. Note that this is *not* an optimization of the algorithms, it is just that everything runs faster.

4. We know that the worst case runtime $T(n)$ of `koMulOpt` satisfies $T(n) \leq cn^{\lg 3}$ for some constant c. Of course asymptotic notation allows for a settling in period, i.e., from some value n_0 of n onwards. In fact this period is not really necessary at least from $n = 1$ onwards because we can just take a larger constant if needed (with the obvious downside that this gives a pessimistic performance guarantee for all large enough values of n). In the preceding part you determined the point from which we would consider using `koMultOpt` so it makes sense for us to obtain a value of c for values of n from that point onwards.

   The supplied `BigIntMul` method

   - `public static void getRatios(Unsigned m, Unsigned n, Unsigned t, File fout, Unsigned xOver)`

   is similar to `getRuntimes` except that for each experiment it only uses `koMultOpt` (the parameters are discussed below). Once it has found the worst case runtime for an integer size n it records the value of that runtime divided by $n^{\lg 3}$. The results are output to the file `fout` in the format

   integer-size *ratio*

   one per line as before. At the end of this output it also appends three extra lines:

   `Ignoring ratios before cross over point <Xover>.`
   `Sorted ratios are:` $r_1, r_2, \ldots$
   `Maximum ratio is:` *max-ratio*
   `Average ratio is:` *ave-ratio*

   Thus the final two lines give us the overall maximum as well as the average of all the ratios. The case for taking the average is that it smooths out to some extent the effects of any garbage collection. In a more detailed study we need to be more sophisticated but here we will keep things simple and rely on the plot (see below) to give us an idea of how useful the average is. You can also use the sorted ratios to get an idea of the effects of garbage collection (we would expect ratios affected by this to be significantly bigger than the "real" ones).

   Use the following values for the parameters:

   - `m`, `n`, `t`: use the same values as in the task of the preceding part (remember that we are focusing on `koMulOpt`),
   - `xOver`: use the crossover point you found where `koMultOpt` becomes more efficient than `schoolMult`.

Note that ratios below the crossover point are ignored (we are treating this as the settling in period). The ratios should all be fairly similar but it would be unrealistic to expect them to be exactly the same.

Finally you will plot the data from the file `koMulOptTimes` of the previous part and the worst case runtime as determined by the theoretical analysis together with the constants found by the preceding experiment. Use the supplied `BigIntMul` method

- `public static void plotRuntimes(double c, double a, file f)`

to produce your plot. For `c` use the maximum constant found above, for `a` use the average and for the file pass the data in the file `koMulOptTimes.txt` from the preceding part. This will bring up a plot which you should save in a file called `plot.jpg`. To be precise this plot will show graphs of the recorded times for `schoolMul`, `koMulOpt` as well as the curves $cn^{\lg 3}$ and $an^{\lg 3}$. [5%]

Compiling and Running

From the commandline, type `cd` *`/path/to/src/folder`* and compile with the command `javac imult/*.java`. Run the program with the command `java imult/`*`ClassName`* where *ClassName* is the `.java` file where your `main()` target is.

Enabling Assertions: There are many assertions that verify conditions are met. To include them when compiling run the command: `javac -source 1.8 imult/*.java`. When running the program, to enable assertions type: `java -ea imult/`*`ClassName`*.

Notes

1. Java has a `BigInteger` class. You *must not* use this in your implementation. The point is to do things from scratch. If you ignore this injunction you will be awarded 0 for the algorithm implementation part.
2. As stated above your code must be well laid out showing its logical structure. Just like good prose or good mathematical writing it must be set out to aid understanding. This way you and others can maintain it as well as spot any errors more easily.

§3.3 Points to consider

If time permits you should think about following points.

Data structure used. The supplied class uses dynamic arrays to hold the digits. An alternative is to use linked lists. Compare briefly the two approaches mentioning at least one advantage and one disadvantage for each.

Experimental conclusions. What do you conclude from the experiments you have carried out, in particular the crossover points of the two algorithms?

You are not required to submit anything here.

§4. Submitting the Coursework

You must submit the two parts of your work by the deadlines given at the head of this document. If you complete the coursework in full, you should be submitting:

Part A: Analysis, in *hard copy only*.

1. Your answer to Exercise 1 on page 12, §2.5.

Part B: Software, for this part the submissions are *electronic only*.

1. Your code in the class file `StudentCode.java` .
2. `koMulOptTimes.txt`.
3. `plot.jpg`.

Note: Your answers to the two exercises of Part 1 are best handwritten (unless you have a medical condition that prevents this) and neatly presented following the guidelines of Lecture Note 2 (and the notes in general). Your hardcopy submission should be made at the appropriate time to:

- ITO, Appleton Tower Room 6.05, Crichton Street, Edinburgh, EH8 9LE

Important: Put your *matriculation number* very clearly at the top of your submission. You do not have to put your name; marks will be returned to the ITO by matriculation number (this is how the `submit` command (see below) organises things. If you need to submit more than one piece of paper please staple the sheets together and put your matriculation number on the top of each sheet (in case they get separated). Do not use folders to hold several sheets as this holds up the marking process significantly.

For electronic submission follow these instructions.

- First put your submission files (and *nothing else*) into one directory called `inf2b-cw1`.
- Submit this directory using the following command (when logged into DICE):

```
submit inf2b cw1 inf2b-cw1
```

Warning: The rule for courseworks is "We mark what is submitted." Before you submit your coursework, make sure that you are submitting the correct files. In some previous years students submitted the wrong files and lost marks that way.

§5. Final observations

In this exercise we have implemented an algorithm using a high level language and abstracted most steps into suitable classes with relevant methods. While this is an excellent way of maintaining correctness and focusing on the essential structure of the algorithm we do pay a heavy price in many ways. All the checks that are carried out slow things down significantly; we have seen from the simulation in §2.4 that the multiplication of even fairly small integers involves a lot of operations and data structures. In applications we might use integers with hundreds of digits, clearly the price is too great for serious use.

In terms of Java, an alternative to the creation of data structures every time we build an integer is to use some scratch space that has been pre-declared; as noted earlier in §2.3.1. We use space as necessary and then release it.

Finally, operations on very large integers are crucial in computer algebra systems (these provide environments for carrying out mathematical operations of many kinds, not just numerical). It follows that these must be as fast as possible and so assembly language code is worth the effort there.

Bibliography

1. A. Karatsuba and Yu. Ofman, *Dokl. Akad. Nauk SSSR* **145** (1962) 293–294. (English translation: Multiplication of multi-digit numbers on automata, *Soviet Phys. Dokl.* **7** (1963) 595–596).

2. D. E. Knuth, *Seminumerical Algorithms*, (Second Edition), Addison-Wesley (1981).

3. A. Schönhage and V. Strassen, Schnelle Multiplikation großer Zahlen, *Computing*, **7**, (1971) 281–292.

Kyriakos Kalorkoti

Software by Abby Levenberg modified by Shahzad Asif and Naums Mogers

Monday 19th February, 2018