11x11=121

Now that calculators are readily available they are sometimes used in classrooms to explore patterns in sequences of numbers. It has been pointed out that some patterns such as

11x11=121 111x111=12321 1111*1111=1234321 ...

only really become interesting when the numbers get too big for a calculator to handle accurately. Computers are usually no better but, in the article that follows, **David Tall** explains that he has the answer.

Arithmetic with large numbers

Modern microcomputers do not usually cope well with the arithmetic of large integers; instead they store and display numbers to an accuracy of a few digits, with inevitable rounding errors. This is a consequence of the way computer languages represent and process numbers, rather than an inherent weakness of the computer. By using a different representation it is possible to obtain accuracy to any specified degree, subject only to the limitations of available memory. In this article I shall show how BBC BASIC may be extended to calculate exact sums, differences, products, quotients, remainders and powers for whole numbers in the range $\pm 10^{254}$ and shall demonstrate how this facility can be used to factorize large numbers.

Normally BBC BASIC can only cope with integers in the range $\pm(2^{31}-1)$. The command

```
A%=2↑31 – 1
```

gives an integer variable its maximum value whilst

```
A%=2↑31
```

produces the error message 'Too big'.

The technical reason for this is that integers are stored internally in four memory locations, each of which can hold an eight-digit binary number. The number of binary digits available is therefore thirty-two. One of them is used to represent the sign. Thus the largest integer that can be represented is $2^{31}-1$. This can give unsatisfactory results. Even in the accepted integer range the numbers are not printed accurately in decimal notation.

```
A%=2↑31 – 1 : PRINT A%
```

gives the expression

```
2.14748365E9
```

instead of the exact answer

2,147,483,647.

```
A%=2↑31 – 1 : PRINT 2*A% : PRINT A%+A%
```

gives two radically different answers:

```
2*A%=4.29496729E9
```

and

```
A%+A%= –2.
```

The difference arises from an error in the BASIC interpreter. Integer arithmetic is used wherever possible, and the interpreter switches to floating-point routines when integers become inappropriate. The computation 2*A% clearly exceeds the maximum integer size and is performed in floating-point arithmetic. But A%+A% is erroneously carried out as an integer calculation, with the sum of two thirty-one-digit binary numbers producing an error in the thirty-second place. This sets the minus sign, but fails to trigger a 'Too big' error, thus producing the number −2 instead of the true value 2 ↑ 32 − 2.

To cope with large-number arithmetic and get it right more memory must be used to store the numbers and new arithmetic routines must be written to manipulate them. The routines can be written in BASIC and methods have been devised independently by Trevor Fletcher and Joe Watson. Joe's package is available on disk from Keele Department of Education for £5. The central idea is to hold the number in memory as a string, giving a maximum length of 255 digits. To perform the arithmetic the strings of digits are broken into manageable chunks (four digits at a time, say) which are turned into numbers so that they can be operated on using ordinary BASIC arithmetic. The results are then reassembled to give the answer as another long string of digits. Addition and subtraction are fast; multiplication and division are a little slower. The methods are ideal for the demonstration of what can be done in BASIC and are quite satisfactory for normal arithmetic. But time becomes a problem when a large number of operations is required. In such circumstances a method of speeding up the calculations becomes highly desirable.

Faster arithmetic is made possible by writing the routines in machine code. The '6502 processor' that drives the BBC Microcomputer includes routines for decimal arithmetic which are totally ignored by BBC BASIC. The program given on the disk available in connection with this issue of *MICROMATH* uses these neglected resources to provide arithmetic for large numbers.

I shall illustrate some of the operations that can be carried out with large numbers.

To load the procedures into your computer, type

```
CHAIN "BIGNUM"
```

Typing

```
PROCcalculate
```

turns the computer into a simple large-number calculator that also prints the time taken for each operation. For instance, the value of 2^{200} will be printed as

```
1606938044258990275541962092341162602
522202993782792835301376
```

in just 0.15 seconds

Investigations confirm that the longer the strings are, the longer it takes, and addition and subtraction are much faster than the other operations.

Playing with vast numbers may soon pall, as meaningless strings of large digits are flashed up before your eyes, though you may get some perverse pleasure from causing calculation overflow even with these large numbers. To exit from the calculator mode, press ESCAPE.

All the user-defined functions may be used direct from the keyboard without running any program. For instance,

```
PRINT FNadd("123123123","15452345")
```

immediately prints out the result of adding the numbers contained in the strings. Corresponding results will be obtained from FNsubtract and FNmultiply.

```
PRINT FNpower ("2","200")
```

gives the calculation of 2^{200} mentioned earlier.

Calculating the quotient or remainder in a division involves producing both answers simultaneously. Whichever is calculated first, 'FNother' can be used to obtain the other one. Thus

```
PRINT FNquotient("14","3") : PRINT FNother
```

gives the quotient, 4, followed by the remainder, 2, whilst

```
PRINT FNremainder("31","5") : PRINT
FNother
```

gives the remainder, 1, followed by the quotient, 6. Functions may be combined; typing

```
PRINT FNsubtract (FNpower("2","31"),"1")
```

reveals the true value of 2 ↑ 31 − 1.

Values may be held in integer strings, so that the previous calculation can be performed in three steps, as follows:

```
P$=FNpower("2"."31") : A$=FNsubtract(P$,"1") :
 PRINT A$
```

with the obvious bonus that the result of A$ is stored in memory, should it be required again.

A number of useful procedures for factorizing large numbers are included in the program. One of these is for the highest common factor of two large integers and has exactly the same form as THE BASIC procedure which can be written to find the highest common factor of two integers of normal size.

```
DEF FNgcd(X%,Y%) : IFX%=0 THEN=Y%
ELSE=FNgcd(Y%MODX%,Y%)
```

gives the highest common factor (or greatest common divisor) of the two integers, X% and Y%, of normal size. Typing

```
PRINT FNgcd(3108,4095)
```

will give the greatest common divisor of 3108 and 4095 as 21.

A function for finding the highest common factor of two large integers can be defined having exactly the same structure:

```
DEF FNhcf(X$, Y$) : IF  X$="0" THEN=Y$
ELSE=FNhcf(FNremainder(Y$, X$), Y$)
```

and gives a fast and accurate result even with very large numbers.

Finding the factors of a single number is a different kettle of fish. The machine code is much faster than the BASIC routines, but even on a very fast main-frame the time taken for calculating factors increases exponentially with size. It has been estimated that the factorization of a random 200-digit number on a main-frame computer could take 3.8 billion years, so do not expect miracles! The simple routine to be described copes with numbers containing less than twenty digits in a reasonable time.

An elementary algorithm for finding factors uses the property that, if a number n has a factor it has one which does not exceed the square root of n. The function FNfactor (X%), which can be used for normal numbers, seeks factors of X% by starting at 2 and trying successively each number until the square root of X% is reached. When a factor is found it is checked to see if it repeats: the quotient is calculated and this searched for further factors. The square root of the quotient now provides a new maximum for any remaining factors; because this is less than the square root of X% the extent of the search is reduced. The function FNfactor(X%) produces a string containing the product of the prime factors:

```
PRINT FNfactor(120)
```

yields the string

```
"2*2*2*3*5".
```

while

```
F$=FNfactor(X%) : PRINT EVAL F$
```

evaluates the product of the factors and gives the original number, X%.

FNfactors(X$) is used to find factors of large numbers by an analogous process. No square-root function is available, but the length of the string can be used instead: if a number n has a factor, it must have one whose length does not exceed half the length of n.

You can gain an idea of the time this method takes to calculate factors by using the routine:

```
REPEAT : INPUT "number="X$ : TIME=0 :
PRINT "factor=" FNfactors(X$) :
PRINT "time taken="; TIME/100;
"seconds" : UNTIL FALSE
```

Here are results of a few typical runs.

```
number=120
factors=2*2*2*3*5
time taken=0.2 seconds

number=1024
factors=2*2*2*2*2*2*2*2*2*2
time taken=0.37 seconds

number=243912827
factors=7591*30677
time taken=212.61 seconds
```

The larger the prime factors, the longer it takes. The arithmetic operations for large numbers always take at least 1/100 of a second and often take much longer. Even at 1/100 of a second per search it would require one second to try 100 numbers and 10^{n-2} seconds to try 10^n numbers. A number containing a prime factor with forty digits would cause the search to continue until the divisor had more than twenty digits. This would take far in excess of 10^{18} seconds or about thirty-billion years.

Should you allow your fingers to type a random forty-digit number for the computer to factorize you may reach an impasse to which the only solution is to press ESCAPE.

An attempt to

```
PRINT FNfactors(FNpower("2","200"))
```

fails for another reason: the string of factors

```
2*2*2*....
```

becomes 'too long'. The alternative routine FNpowerfactors copes better by grouping together powers of a given prime number.

```
PRINT FNpowerfactors("120")
```

gives

```
(2 ↑ 3)*3*5
```

so that

```
PRINT FNpowerfactors(FNpower("2","200"))
```

successfully yields the response

```
(2 ↑ 200).
```

The routines may easily be used in other programs. The exact way in which this can be done is described in the notes accompanying the disk. ■

David Tall works at the Mathematics Education Research Centre, University of Warwick.