WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Català
Español
Esperanto
Français
Italiano
Português
Türkçe
Edit links

Multiplication algorithm

From Wikipedia, the free encyclopedia

A **multiplication algorithm** is an algorithm (or method) to multiply two numbers. Depending on the size of the numbers, different algorithms are in use. Efficient multiplication algorithms have existed since the advent of the decimal system.

Contents

Grid method [edit]

Main article: Grid method multiplication

The grid method (or box method) is an introductory method for multiple-digit multiplication that is often taught to pupils at primary school or elementary school level. It has been a standard part of the national primary-school mathematics curriculum in England and Wales since the late 1990s.[1]

Both factors are broken up ("partitioned") into their hundreds, tens and units parts, and the products of the parts are then calculated explicitly in a relatively simple multiplication-only stage, before these contributions are then totalled to give the final answer in a separate addition stage.

The calculation 34 × 13, for example, could be computed using the grid:

×	30	4
10	300	40
3	90	12

```
  300
   40
   90
 + 12
 ————
  442
```

followed by addition to obtain 442, either in a single sum (see right), or through forming the row-by-row totals (300 + 40) + (90 + 12) = 340 + 102 = 442.

This calculation approach (though not necessarily with the explicit grid arrangement) is also known as the partial products algorithm. Its essence is the calculation of the simple multiplications separately, with all addition being left to the final gathering-up stage.

The grid method can in principle be applied to factors of any size, although the number of sub-products becomes cumbersome as the number of digits increases. Nevertheless it is seen as a usefully explicit method to introduce the idea of multiple-digit multiplications; and, in an age when most multiplication calculations are done using a calculator or a spreadsheet, it may in practice be the only multiplication algorithm that some students will ever need.

Long multiplication [edit]

If a positional numeral system is used, a natural way of multiplying numbers is taught in schools as **long multiplication**, sometimes called **grade-school multiplication**, sometimes called **Standard Algorithm**: multiply the multiplicand by each digit of the multiplier and then add up all the properly shifted results. It requires memorization of the multiplication table for single digits.

This is the usual algorithm for multiplying larger numbers by hand in base 10. Computers initially used a very similar shift and add algorithm in base 2, but modern processors have optimized circuitry for fast multiplications using more efficient algorithms, at the price of a more complex hardware realization. A person doing long multiplication on paper will write down all the products and then add them together; an abacus-user will sum the products as soon as each one is computed.

Example [edit]

This example uses *long multiplication* to multiply 23,958,233 (multiplicand) by 5,830 (multiplier) and arrives at 139,676,498,390 for the result (product).

```
      23958233
×         5830
———————————————
      00000000 ( =      23,958,233 ×     0)
     71874699  ( =      23,958,233 ×    30)
   191665864   ( =      23,958,233 ×   800)
+ 119791165    ( =      23,958,233 × 5,000)
———————————————
  139676498390 ( = 139,676,498,390        )
```

Pseudocode [edit]

Below pseudocode describes the process of above multiplication. It keeps only one row to maintain sum which finally becomes the result. Note that we use '+=' operator to denote sum to existing value and store operation (akin to languages such as Java and C) for compactness.

```
multiply(a[1..p], b[1..q], base)
  m = [1..p+q]   //Allocate space for result
  for bi = 1 to q
    carry = 0
    for ai = 1 to p
      m[ai + bi - 1] += carry + a[ai] * b[bi]
      carry = m[ai + bi - 1] / base
      m[ai + bi - 1] = m[ai + bi - 1] mod base
    m[bi + p] += carry
  return m
```

Space complexity [edit]

This section **does not cite any references or sources**. Please help improve this section by adding citations to reliable sources. Unsourced material may be challenged and removed. *(September 2012)*

Let *n* be the total number of digits in the two input numbers in base *D*. If the result must be kept in memory then the space complexity is trivially Θ(*n*). However in certain applications, the entire result need not be kept in memory and instead digits of results can be streamed out as they are computed (for example, to system console or file). In these scenarios, long multiplication has the advantage that it can easily be formulated as a log space algorithm; that is, an algorithm that only needs working space proportional to the logarithm of the number of digits in the input (Θ(log *n*)). This is the *double* logarithm of the numbers being multiplied themselves (log log *N*). Note that operands themselves still need to be kept in memory and their Θ(*n*) space is not considered in this analysis.

The method is based on the observation that each digit of the result can be computed from right to left with only knowing the carry from the previous step. Let a_i and b_i be the *i*-th digit of the operand, r_i be the *i*-th digit of the result and c_i be the carry generated for r_i (i=1 is the right most digit) then

$$r_i = (c_{i-1} + \sum_{j+k=i} a_j b_k) \mod D$$

$$c_i = \lfloor (c_{i-1} + \sum_{j+k=i} a_j b_k)/D \rfloor$$

$$c_0 = 0$$

A simple inductive argument shows that the carry can never exceed *n* and the total sum for r_i can never exceed *D* * *n*: the carry into the first column is zero, and for all other columns, there are at most *n* digits in the column, and a carry of at most *n* from the previous column (by the induction hypothesis). The sum is at most *D* * *n*, and the carry to the next column is at most *D* * *n* / *D*, or *n*. Thus both these values can be stored in O(log *n*) digits.

In pseudocode, the log-space algorithm is:

```
multiply(a[1..p], b[1..q], base) // Operands containing rightmost digits at index 1
    sum ← 0
    for ri = 1 to p + q - 1                     //For each digit of result
        for bi = MAX(1, ri - p + 1) to MIN(ri, q) //Digits from b that need to be considered
            ai ← ri − bi + 1                    //Digits from a follow "symmetry"
            sum ← sum + (a[ai] × b[bi])
        result[ri] ← sum mod base
        sum ← floor(sum / base)
    result[p+q] ← sum mod base                  //Last digit of the result comes from last
carry
```

Electronic usage [edit]

Some chips implement this algorithm for various integer and floating-point sizes in computer hardware or in microcode. In arbitrary-precision arithmetic, it's common to use long multiplication with the base set to 2^w, where *w* is the number of bits in a word, for multiplying relatively small numbers.

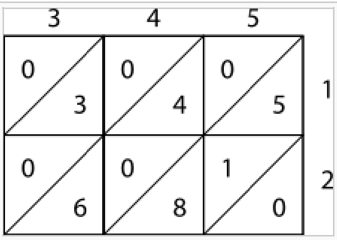
To multiply two numbers with *n* digits using this method, one needs about n^2 operations. More formally: using a natural size metric of number of digits, the time complexity of multiplying two *n*-digit numbers using long multiplication is $\Theta(n^2)$.

When implemented in software, long multiplication algorithms have to deal with overflow during additions, which can be expensive. For this reason, a typical approach is to represent the number in a small base *b* such that, for example, $8b^2$ is a representable machine integer (for example Richard Brent used this approach in his Fortran package MP[2]); we can then perform several additions before having to deal with overflow. When the number becomes too large, we add part of it to the result or carry and map the remaining part back to a number less than *b*; this process is called *normalization*.

Lattice multiplication [edit]

Main article: Lattice multiplication

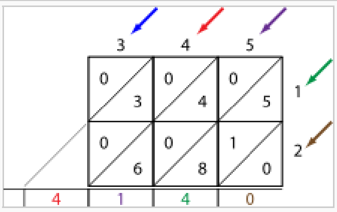
Lattice, or sieve, multiplication is algorithmically equivalent to long multiplication. It requires the preparation of a lattice (a grid drawn on paper) which guides the calculation and separates all the multiplications from the additions. It was introduced to Europe in 1202 in Fibonacci's Liber Abaci. Leonardo described the operation as mental, using his right and left hands to carry the intermediate calculations. Matrakçı Nasuh presented 6 different variants of this method in this 16th-century book, Umdet-ul Hisab. It was widely used in Enderun schools across the Ottoman Empire.[3] Napier's bones, or Napier's rods also used this method, as published by Napier in 1617, the year of his death.



First, set up the grid by marking its rows and columns with the numbers to be multiplied. Then, fill in the boxes with tens digits in the top triangles and units digits on the bottom.

As shown in the example, the multiplicand and multiplier are written above and to the right of a lattice, or a sieve. It is found in Muhammad ibn Musa al-Khwarizmi's "Arithmetic", one of Leonardo's sources mentioned by Sigler, author of "Fibonacci's Liber Abaci", 2002.

- During the multiplication phase, the lattice is filled in with two-digit products of the corresponding digits labeling each row and column: the tens digit goes in the top-left corner.
- During the addition phase, the lattice is summed on the diagonals.
- Finally, if a carry phase is necessary, the answer as shown along the left and bottom sides of the lattice is converted to normal form by carrying ten's digits as in long addition or multiplication.



Finally, sum along the diagonal tracts and carry as needed to get the answer

Example [edit]

The pictures on the right show how to calculate 345 × 12 using lattice multiplication. As a more complicated example, consider the picture below displaying the computation of 23,958,233 multiplied by 5,830 (multiplier); the result is 139,676,498,390. Notice 23,958,233 is along the top of the lattice and 5,830 is along the right side. The products fill the lattice and the sum of those products (on the diagonal) are along the left and bottom sides. Then those sums are totaled as shown.

```
     2   3   9   5   8   2   3   3
   +---+---+---+---+---+---+---+---+-
   |1 /|1 /|4 /|2 /|4 /|1 /|1 /|1 /|
   | / | / | / | / | / | / | / | / | 5
```

```
 01
 002
 0017
 00024
 01|/ 0|/ 5|/ 5|/ 5|/ 0|/ 0|/ 5|/ 5|
   +---+---+---+---+---+---+---+---+-
   |1 /|2 /|7 /|4 /|6 /|1 /|2 /|2 /|
   | / | / | / | / | / | / | / | / | 8
 02|/ 6|/ 4|/ 2|/ 0|/ 4|/ 6|/ 4|/ 4|
   +---+---+---+---+---+---+---+---+-
   |0 /|0 /|2 /|1 /|2 /|0 /|0 /|0 /|
   | / | / | / | / | / | / | / | / | 3
 17|/ 6|/ 9|/ 7|/ 5|/ 4|/ 6|/ 9|/ 9|
   +---+---+---+---+---+---+---+---+-
   |0 /|0 /|0 /|0 /|0 /|0 /|0 /|0 /|
   | / | / | / | / | / | / | / | / | 0
 24|/ 0|/ 0|/ 0|/ 0|/ 0|/ 0|/ 0|/ 0|
   +---+---+---+---+---+---+---+---+-
     26  15  13  18  17  13  09  00
```

```
 000026
 0000015
 00000013
 000000018
 0000000017
 00000000013
 000000000009
 0000000000000
 —————————————
  139676498390
```

```
 = 139,676,498,390
```

Peasant or binary multiplication [edit]

Main article: Peasant multiplication

This section **does not cite any references or sources**. Please help improve this section by adding citations to reliable sources. Unsourced material may be challenged and removed. *(January 2013)*

In base 2, long multiplication reduces to a nearly trivial operation. For each '1' bit in the multiplier, shift the multiplicand an appropriate amount and then sum the shifted values. Depending on computer processor architecture and choice of multiplier, it may be faster to code this algorithm using hardware bit shifts and adds rather than depend on multiplication instructions, when the multiplier is fixed and the number of adds required is small.

This algorithm is also known as Peasant multiplication, because it has been widely used among those who are unschooled and thus have not memorized the multiplication tables required by long multiplication. The algorithm was also in use in ancient Egypt.

On paper, write down in one column the numbers you get when you repeatedly halve the multiplier, ignoring the remainder; in a column beside it repeatedly double the multiplicand. Cross out each row in which the last digit of the first number is even, and add the remaining numbers in the second column to obtain the product.

The main advantages of this method are that it can be taught quickly, no memorization is required, and it can be performed using tokens such as poker chips if paper and pencil are not available. It does however take more steps than long multiplication so it can be unwieldy when large numbers are involved.

Examples [edit]

This example uses peasant multiplication to multiply 11 by 3 to arrive at a result of 33.

```
Decimal:      Binary:
11   3        1011  11
5    6        101  110
2   ~~12~~        10  ~~1100~~
1   24        1  11000
    ——          ——————
    33          100001
```

Describing the steps explicitly:

- 11 and 3 are written at the top
- 11 is halved (5.5) and 3 is doubled (6). The fractional portion is discarded (5.5 becomes 5).
- 5 is halved (2.5) and 6 is doubled (12). The fractional portion is discarded (2.5 becomes 2). The figure in the left column (2) is **even**, so the figure in the right column (12) is discarded.
- 2 is halved (1) and 12 is doubled (24).
- All not-scratched-out values are summed: 3 + 6 + 24 = 33.

The method works because multiplication is distributive, so:

$$\begin{aligned}3 \times 11 &= 3 \times (1 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3)\\ &= 3 \times (1 + 2 + 8)\\ &= 3 + 6 + 24\\ &= 33.\end{aligned}$$

A more complicated example, using the figures from the earlier examples (23,958,233 and 5,830):

```
Decimal:                 Binary:
5830  ~~23958233~~          1011011000110  ~~1011011011001001011011001~~
2915  47916466          101101100011  10110110110010010110110010
1457  95832932          10110110001  101101101100100101101100100
728  ~~191665864~~          1011011000  ~~1011011011001001011011001000~~
364  ~~383331728~~          101101100  ~~10110110110010010110110010000~~
182  ~~766663456~~          10110110  ~~101101101100100101101100100000~~
91  1533326912          1011011  1011011011001001011011001000000
45  3066653824          101101  10110110110010010110110010000000
22  ~~6133307648~~          10110  ~~101101101100100101101100100000000~~
11 12266615296          1011  1011011011001001011011001000000000
5  24533230592          101  10110110110010010110110010000000000
2  ~~49066461184~~          10  ~~101101101100100101101100100000000000~~
1  98132922368          1  1011011011001001011011001000000000000
  ————————————             1022143253354344244353353243222210110 (before carry)
  139676498390            10000010000101010111100011100111010110
```

Shift and add [edit]

Historically, computers used a "shift and add" algorithm to multiply small integers. Both base 2 long multiplication and base 2 peasant multiplication reduce to this same algorithm. In base 2, multiplying by the single digit of the multiplier reduces to a simple series of logical AND operations. Each partial product is added to a running sum as soon as each partial product is computed. Most currently available microprocessors implement this or other similar algorithms (such as Booth encoding) for various integer and floating-point sizes in hardware multipliers or in microcode.

On currently available processors, a bit-wise shift instruction is faster than a multiply instruction and can be used to multiply (shift left) and divide (shift right) by powers of two. Multiplication by a constant and division by a constant can be implemented using a sequence of shifts and adds or subtracts. For example, there are several ways to multiply by 10 using only bit-shift and addition.

```
((x << 2) + x) << 1 # Here 10*x is computed as (x*2^2 + x)*2
(x << 3) + (x << 1) # Here 10*x is computed as x*2^3 + x*2
```

In some cases such sequences of shifts and adds or subtracts will outperform hardware multipliers and especially dividers. A division by a number of the form 2^n or $2^n \pm 1$ often can be converted to such a short sequence.

These types of sequences have to always be used for computers that do not have a "multiply" instruction,[4] and can also be used by extension to floating point numbers if one replaces the shifts with computation of *2*x* as *x+x*, as these are logically equivalent.

Quarter square multiplication [edit]

Two quantities can be multiplied using quarter squares by employing the following identity involving the floor function that some sources[5][6] attribute to Babylonian mathematics (2000–1600 BC).

$$\left\lfloor \frac{(x+y)^2}{4} \right\rfloor - \left\lfloor \frac{(x-y)^2}{4} \right\rfloor = \frac{1}{4}\left(\left(x^2+2xy+y^2\right)-\left(x^2-2xy+y^2\right)\right) = \frac{1}{4}\left(4xy\right) = xy.$$

If one of $x+y$ and x-y is odd, the other is odd too; this means that the fractions, if any, will cancel out, and discarding the remainders does not introduce any error. Below is a lookup table of quarter squares with the remainder discarded for the digits 0 through 18; this allows for the multiplication of numbers up to $9{\times}9$.

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$\lfloor n^2/4 \rfloor$	0	0	1	2	4	6	9	12	16	20	25	30	36	42	49	56	64	72	81

If, for example, you wanted to multiply 9 by 3, you observe that the sum and difference are 12 and 6 respectively. Looking both those values up on the table yields 36 and 9, the difference of which is 27, which is the product of 9 and 3.

Antoine Voisin published a table of quarter squares from 1 to 1000 in 1817 as an aid in multiplication. A larger table of quarter squares from 1 to 100000 was published by Samuel Laundy in 1856,[7] and a table from 1 to 200000 by Joseph Blater in 1888.[8]

Quarter square multipliers were used in analog computers to form an analog signal that was the product of two analog input signals. In this application, the sum and difference of two input voltages are formed using operational amplifiers. The square of each of these is approximated using piecewise linear circuits. Finally the difference of the two squares is formed and scaled by a factor of one fourth using yet another operational amplifier.

In 1980, Everett L. Johnson proposed using the quarter square method in a digital multiplier.[9] To form the product of two 8-bit integers, for example, the digital device forms the sum and difference, looks both quantities up in a table of squares, takes the difference of the results, and divides by four by shifting two bits to the right. For 8-bit integers the table of quarter squares will have 2^9-1=511 entries (one entry for the full range 0..510 of possible sums, the differences using only the first 256 entries in range 0..255) or 2^9-1=511 entries (using for negative differences the technique of 2-complements and 9-bit masking, which avoids testing the sign of differences), each entry being 16-bit wide (the entry values are from $(0^2/4)=0$ to $(510^2/4)=65025$).

The Quarter square multiplier technique has also benefitted 8-bit systems that do not have any support for a hardware multiplier. Steven Judd implemented this for the 6502.[10]

Ancient Indian algorithm for multiplying numbers close to a round number [edit]

This section **does not cite any references or sources**. Please help improve this section by adding citations to reliable sources. Unsourced material may be challenged and removed. *(May 2013)*

Suppose we want to multiply two numbers x and y that are close to a round number N. Writing $x = N + x'$ and $y = N + y'$, allows one to express the product as:

$$xy = (N + x')\,(N + y') = N^2 + N\,(x' + y') + x'y' = N\,(N + x' + y') + x'y' = N\,(x + y') + x'y'$$

Example. Suppose we want to multiply 92 by 87. We can then take $N = 100$ and implement the above formula as follows. We write the numbers below each other and next to them the amounts we have to add to get to 100:

$$\begin{array}{r|r} 92 & 8 \\ 87 & 13 \end{array}$$

Since the numbers on the right are $-x'$ and $-y'$, the product is obtained by subtracting from the top left number the bottom right number (or subtract from the bottom left number the top right number), multiply that by 100 and add to that the product of the two numbers on the right. We have 87 - 8 = 79; 79*100 = 7900; 8*13 = 104; 7900+104 = 8004.

The multiplication of 8 by 13 could also have been done using the same method, by taking $N = 10$. The above table can then be extended to:

$$\begin{array}{r|r|r} 92 & 8 & 2 \\ 87 & 13 & -3 \end{array}$$

The product is then computed by evaluating the differences 87-8=79; 13-2 = 11, and the product 2*(-3) = -6. We then have 92*87 = 79*100 + 11*10 - 6 = 7900 + 104 = 8004.

Fast multiplication algorithms for large inputs [edit]

List of unsolved problems in computer science

What is the fastest algorithm for multiplication of two n-digit numbers?

Gauss's complex multiplication algorithm [edit]

Complex multiplication normally involves four multiplications and two additions.

$$(a + bi)(c + di) = (ac - bd) + (bc + ad)i.$$

Or

$$\begin{array}{c|c|c} \times & a & bi \\ \hline c & ac & bci \\ \hline di & adi & -bd \end{array}$$

By 1805 Gauss had discovered a way of reducing the number of multiplications to three.[11]

The product (*a* + *bi*) · (*c* + *di*) can be calculated in the following way.

$k_1 = c \cdot (a + b)$
$k_2 = a \cdot (d - c)$
$k_3 = b \cdot (c + d)$
Real part = $k_1 - k_3$
Imaginary part = $k_1 + k_2$.

This algorithm uses only three multiplications, rather than four, and five additions or subtractions rather than two. If a multiply is more expensive than three adds or subtracts, as when calculating by hand, then there is a gain in speed. On modern computers a multiply and an add can take about the same time so there may be no speed gain. There is a trade-off in that there may be some loss of precision when using floating point.

For fast Fourier transforms (FFTs) (or any linear transformation) the complex multiplies are by constant coefficients *c* + *di* (called twiddle factors in FFTs), in which case two of the additions (*d*−*c* and *c*+*d*) can be precomputed. Hence, only three multiplies and three adds are required.[12] However, trading off a multiplication for an addition in this way may no longer be beneficial with modern floating-point units.[13]

Karatsuba multiplication [edit]

Main article: Karatsuba algorithm

For systems that need to multiply numbers in the range of several thousand digits, such as computer algebra systems and bignum libraries, long multiplication is too slow. These systems may employ **Karatsuba multiplication**, which was discovered in 1960 (published in 1962). The heart of Karatsuba's method lies in the observation that two-digit multiplication can be done with only three rather than the four multiplications classically required. This is an example of what is now called a *divide and conquer algorithm*. Suppose we want to multiply two 2-digit numbers: $x_1x_2\cdot y_1y_2$:

1. compute $x_1 \cdot y_1$, call the result A
2. compute $x_2 \cdot y_2$, call the result B
3. compute $(x_1 + x_2) \cdot (y_1 + y_2)$, call the result C
4. compute $C - A - B$, call the result K; this number is equal to $x_1 \cdot y_2 + x_2 \cdot y_1$
5. compute $A \cdot 100 + K \cdot 10 + B$.

Bigger numbers x_1x_2 can be split into two parts x_1 and x_2. Then the method works analogously. To compute these three products of m-digit numbers, we can employ the same trick again, effectively using recursion. Once the numbers are computed, we need to add them together (step 5.), which takes about n operations.

Karatsuba multiplication has a time complexity of $O(n^{\log_2 3}) \approx O(n^{1.585})$, making this method significantly faster than long multiplication. Because of the overhead of recursion, Karatsuba's multiplication is slower than long multiplication for small values of n; typical implementations therefore switch to long multiplication if n is below some threshold.

Karatsuba's algorithm is the first known algorithm for multiplication that is asymptotically faster than long multiplication,[14] and can thus be viewed as the starting point for the theory of fast multiplications.

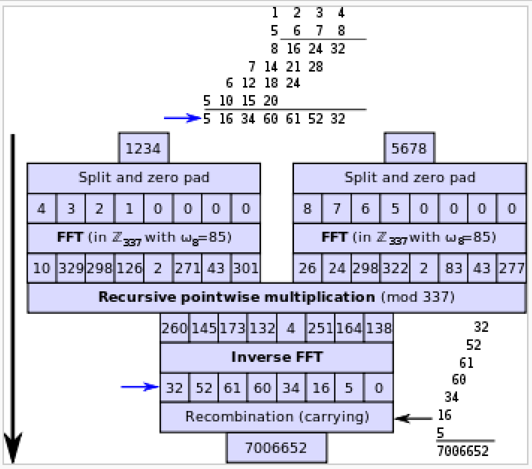
Toom–Cook [edit]

Main article: Toom–Cook multiplication

Another method of multiplication is called Toom–Cook or Toom-3. The Toom–Cook method splits each number to be multiplied into multiple parts. The Toom–Cook method is one of the generalizations of the Karatsuba method. A three-way Toom–Cook can do a size-*3N* multiplication for the cost of five size-*N* multiplications, improvement by a factor of 9/5 compared to the Karatsuba method's improvement by a factor of 4/3.

Although using more and more parts can reduce the time spent on recursive multiplications further, the overhead from additions and digit management also grows. For this reason, the method of Fourier transforms is typically faster for numbers with several thousand digits, and asymptotically faster for even larger numbers.

Fourier transform methods [edit]



Demonstration of multiplying 1234 × 5678 = 7006652 using fast Fourier transforms (FFTs). Number-theoretic transforms in the integers modulo 337 are used, selecting 85 as an 8th root of unity. Base 10 is used in place of base 2^w for illustrative purposes.

The basic idea due to Strassen (1968), is to use fast polynomial multiplication to perform fast integer multiplication. The algorithm was made practical and theoretical guarantees were provided in 1971 by Schönhage and Strassen resulting in the Schönhage–Strassen algorithm. [15] The details are the following: We choose the largest integer w that will not cause overflow during the process outlined below. Then we split the two numbers into m groups of w bits as follows

$$a = \sum_{i=0}^{m-1} a_i 2^{wi} \text{ and } b = \sum_{j=0}^{m-1} b_j 2^{wj}.$$

We look at these numbers as polynomials in x, where $x = 2^w$, to get,

$$a = \sum_{i=0}^{m-1} a_i x^i \text{ and } b = \sum_{j=0}^{m-1} b_j x^j.$$

Then we can then say that,

$$ab = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} a_i b_j x^{(i+j)} = \sum_{k=0}^{2m-2} c_k x^k$$

Clearly the above setting is realized by polynomial multiplication, of two polynomials a and b. The crucial step now, is to use Fast Fourier multiplication of polynomials, to realize the multiplications above faster than in naive $O(m^2)$ time.

To remain in the modular setting of Fourier transforms, we look for a ring with a $2m^{th}$ root of unity. hence we do multiplication modulo N (and thus in the Z/NZ ring). Further, N must be chosen so that there is no 'wrap around', essentially, no reductions modulo N occur. Thus, the choice of N is crucial. For example, it could be done as,

$$N = 2^{3w} + 1$$

The ring Z/NZ would thus have a $2m^{th}$ root of unity, namely 8. Also, it can be checked that $c_k < N$, and thus no wrap around will occur.

The algorithm has a time complexity of $\Theta(n \log(n) \log(\log(n)))$ and is used in practice for numbers with more than 10,000 to 40,000 decimal digits. In 2007 this was improved by Martin Fürer (Fürer's algorithm) [16] to give a time complexity of $n \log(n)\, 2^{\Theta(\log^*(n))}$ using Fourier transforms over complex numbers. Anindya De, Chandan Saha, Piyush Kurur and Ramprasad Saptharishi[17] gave a similar algorithm using modular arithmetic in 2008 achieving the same running time. In context of the above material, what these latter authors have achieved is to find N much less than $2^{3k} + 1$, so that Z/NZ has a $2m^{th}$ root of unity. This speeds up computation and reduces the time complexity. However, these latter algorithms are only faster than Schönhage–Strassen for impractically large inputs.

Using number-theoretic transforms instead of discrete Fourier transforms avoids rounding error problems by using modular arithmetic instead of floating-point arithmetic. In order to apply the factoring which enables the FFT to work, the length of the transform must be factorable to small primes, and must be a factor of N-1, where N is the field size. In particular, calculation using a Galois Field GF(k^2), where k is a Mersenne Prime, allows the use of a transform sized to a power of 2; e.g. $k = 2^{31}$-1 supports transform sizes up to 2^{32}.

Lower bounds [edit]

There is a trivial lower bound of $\Omega(n)$ for multiplying two n-bit numbers on a single processor; no matching algorithm (on conventional Turing machines) nor any better lower bound is known. Multiplication lies outside of $AC^0[p]$ for any prime p, meaning there is no family of constant-depth, polynomial (or even subexponential) size circuits using AND, OR, NOT, and MOD_p gates that can compute a product. This follows from a constant-depth reduction of MOD_q to multiplication.[18] Lower bounds for multiplication are also known for some classes of branching programs.[19]

Polynomial multiplication [edit]

This section requires expansion.
(October 2008)

All the above multiplication algorithms can also be expanded to multiply polynomials. For instance the Strassen algorithm may be used for polynomial multiplication[20] Alternatively the Kronecker substitution technique may be used to convert the problem of multiplying polynomials into a single binary multiplication.[21]

See also [edit]

- Binary multiplier
- Division algorithm
- Logarithm
- Mental calculation
- Prosthaphaeresis
- Slide rule
- Trachtenberg system
- Horner scheme for evaluation of a polynomial

References [edit]

1. ^ Gary Eason, Back to school for parents, *BBC News*, 13 February 2000
Rob Eastaway, Why parents can't do maths today, *BBC News*, 10 September 2010
2. ^ Richard P. Brent. A Fortran Multiple-Precision Arithmetic Package. Australian National University. March 1978.
3. ^ Corlu, M. S., Burlbaw, L. M., Capraro, R. M., Corlu, M. A.,& Han, S. (2010). The Ottoman Palace School Enderun and The Man with Multiple Talents, Matrakçı Nasuh. Journal of the Korea Society of Mathematical Education Series D: Research in Mathematical Education. 14(1), pp. 19–31.
4. ^ "Novel Methods of Integer Multiplication and Division" by G. Reichborn-Kjennerud
5. ^ McFarland, David (2007), *Quarter Tables Revisited: Earlier Tables, Division of Labor in Table Construction, and Later Implementations in Analog Computers*, p. 1
6. ^ Robson, Eleanor (2008). *Mathematics in Ancient Iraq: A Social History*. p. 227. ISBN 978-0691091822.
7. ^ "Reviews", *The Civil Engineer and Architect's journal*, 1857: 54–55.
8. ^ Holmes, Neville (2003), "Multiplying with quarter squares", *The Mathematical Gazette* **87** (509): 296–299, JSTOR 3621048.
9. ^ Everett L., Johnson (March 1980), "A Digital Quarter Square Multiplier", *IEEE Transactions on Computers* (Washington, DC, USA: IEEE Computer Society) **C–29** (3): 258–261, doi:10.1109/TC.1980.1675558, ISSN 0018-9340, retrieved 2009-03-26
10. ^ Judd, Steven (Jan 1995), *C=Hacking* (9) http://www.ffd2.com/fridge/chacking/c=hacking9.txt Missing or empty `|title=` (help)
11. ^ Knuth, Donald E. (1988), *The Art of Computer Programming volume 2: Seminumerical algorithms*, Addison-Wesley, pp. 519, 706
12. ^ P. Duhamel and M. Vetterli, Fast Fourier transforms: A tutorial review and a state of the art", *Signal Processing* vol. 19, pp. 259–299 (1990), section 4.1.
13. ^ S. G. Johnson and M. Frigo, "A modified split-radix FFT with fewer arithmetic operations," *IEEE Trans. Signal Processing* vol. 55, pp. 111–119 (2007), section IV.
14. ^ D. Knuth, *The Art of Computer Programming*, vol. 2, sec. 4.3.3 (1998)

15. ^ A. Schönhage and V. Strassen, "Schnelle Multiplikation großer Zahlen", *Computing* **7** (1971), pp. 281–292.
16. ^ Fürer, M. (2007). "Faster Integer Multiplication" in Proceedings of the thirty-ninth annual ACM symposium on Theory of computing, June 11–13, 2007, San Diego, California, USA
17. ^ Anindya De, Piyush P Kurur, Chandan Saha, Ramprasad Saptharishi. Fast Integer Multiplication Using Modular Arithmetic. Symposium on Theory of Computation (STOC) 2008.
18. ^ Sanjeev Arora and Boaz Barak, *Computational Complexity: A Modern Approach*, Cambridge University Press, 2009.
19. ^ Farid Ablayev and Marek Karpinski, *A lower bound for integer multiplication on randomized ordered read-once branching programs*, Information and Computation 186 (2003), 78–89.
20. ^ "Strassen algorithm for polynomial multiplication". Everything2.
21. ^ von zur Gathen, Joachim; Gerhard, Jürgen (1999), *Modern Computer Algebra*, Cambridge University Press, pp. 243–244, ISBN 978-0-521-64176-0.

External links [edit]

Basic arithmetic [edit]

- The Many Ways of Arithmetic in UCSMP Everyday Mathematics
- A Powerpoint presentation about ancient mathematics
- Lattice Multiplication Flash Video

Advanced algorithms [edit]

- Multiplication Algorithms used by GMP

v · t · e **Number-theoretic algorithms**

Primality tests	AKS test · APR test · Baillie–PSW · ECPP test · Elliptic curve · Pocklington · Fermat · Lucas · *Lucas–Lehmer* · *Lucas–Lehmer–Riesel* · *Proth's theorem* · *Pépin's* · Quadratic Frobenius test · Solovay–Strassen · Miller–Rabin
Prime-generating	Sieve of Atkin · Sieve of Eratosthenes · Sieve of Sundaram · Wheel factorization
Integer factorization	Continued fraction (CFRAC) · Dixon's · Lenstra elliptic curve (ECM) · Euler's · Pollard's rho · *p* − 1 · *p* + 1 · Quadratic sieve (QS) · General number field sieve (GNFS) · *Special number field sieve (SNFS)* · Rational sieve · Fermat's · Shanks' square forms · Trial division · Shor's
Multiplication	Ancient Egyptian · Long · Karatsuba · Toom–Cook · Schönhage–Strassen · Fürer's
Discrete logarithm	Baby-step giant-step · Pollard rho · Pollard kangaroo · Pohlig–Hellman · Index calculus · Function field sieve
Greatest common divisor	Binary · Euclidean · Extended Euclidean · Lehmer's
Modular square root	Cipolla · Pocklington's · Tonelli–Shanks
Other algorithms	Chakravala · Cornacchia · Integer relation · Integer square root · Modular exponentiation · Schoof's

Italics indicate that algorithm is for numbers of special forms · Smallcaps indicate a deterministic algorithm

Categories: Computer arithmetic algorithms | Multiplication