MULTIPLE-LENGTH DIVISION REVISITED: A TOUR OF THE MINEFIELD

Long division of natural numbers plays a crucial role in Cobol arithmetic, cryptography, and primality testing. Only a handful of textbooks discuss the theory and practice of long division, and none of them do it satisfactorily. This paper attempts to fill this surprising gap in the literature on computer algorithms. I illustrate the subtleties of long division by examples, define the problem concisely, summarize the theory, and develop a complete SuperPascal algorithm using a consistent terminology.

INTRODUCTION

Long division of natural numbers plays a crucial role in *Cobol arithmetic* (Brinch Hansen 1966), *cryptography* (Rivest 1978), and *primality testing* (Rabin 1980). While writing a program for primality testing, I learned two lessons the hard way (Brinch Hansen 1992):

1. Only a handful of textbooks discuss the theory and practice of long division, and none of them do it satisfactorily.

2. A correct, efficient algorithm for long division cannot be reinvented with minimal effort.

This chapter attempts to fill this surprising gap in the literature on computer algorithms.

A helpful description of an algorithm should include three elements:

1. An informal introduction that illustrates the problem and its solution by well-chosen examples.

2. A concise definition of the general problem and an explanation of the computational theory.

P. Brinch Hansen, Multiple-length division revisited: A tour of the minefield. *Software—Practice and Experience 24*, (June 1994), 579–601.

3. A well-structured complete algorithm written in a standard programming language using the same terminology as the theoretical discussion.

The best textbooks on algorithms satisfy all three requirements. However, computer scientists do not always appreciate that a professional programmer cannot rely on a complex algorithm unless it is written in a programming language that runs on an available computer. There are too many pitfalls in purely theoretical arguments about "pseudocode" that cannot be compiled and executed on any computer.

I am aware of only four texts that deal with multiple-length arithmetic. I will briefly discuss how well they satisfy the above criteria.

In 1969, *Donald Knuth* published a volume on *Seminumerical Algorithms*. This work contains the most comprehensive treatment of multiple-length arithmetic:

1. In his introduction to long division, Knuth gives a three-line hint of an example and remarks: "It is clear that this same idea works in general."

2. Knuth covers most of the relevant theory from the literature.

3. Knuth presents a division algorithm in three different ways. The first version is a mixture of English, mathematical notation, and *goto* statements. The second one is an informal flowchart that merely shows the flow of control. The third attempt is written in the assembly language MIX with informal comments. Each of these versions conveys insight, but none of them inspire complete confidence in the finer details. They are obviously not well-structured algorithms written in a standard programming language.

However, you must keep in mind that Knuth wrote the first edition of his book in the late 1960s, when structured programming was still in its infancy.

Ten years later, *Suad Alagić* and *Michael Arbib* wrote a book on *The Design of Well-Structured and Correct Programs* (1978). This text has two pages on long division:

1. Alagić and Arbib do not illustrate long division by examples.

2. They state four theorems from Knuth without motivation, proof, or reference.

3. Their algorithm for long division consists of 56 lines written in Pascal. The entire program is a single compound statement composed of shorter statements according to the principles of structured programming. However, the lack of procedures makes it hard to study the algorithm bottom-up (or top-down) at different levels of detail. The program assumes that array dimensions correspond exactly to the lengths of operands. As it stands, the program cannot be compiled and executed. (It has one syntax error and two undeclared names.) The algorithm is helpful, but not sufficiently developed for software design.

Clearly, this program could have been developed further. If the authors had finished the job, there would have been no need to write this tutorial. Alagić and Arbib make a devastating remark about this part of their book: "The reader who has struggled through [our program]—which is typical of the way in which programs are presented and documented—may come to understand the advantages of a top-down approach to presenting a program."

This brings me to *Derick Wood's* textbook on *Paradigms and Programming with Pascal* (1984), which presents a slightly different form of long division:

1. Wood carefully explains long division by an example.

2. His method underestimates quotient digits and corrects them by a slow loop that uses multiple-length arithmetic. The main problem is the lack of a theory that predicts the maximum number of corrections required.

3. Wood's algorithm consists of seven procedures, three of which are left as exercises. With one exception, the remaining procedures are well-structured and well-explained. However, the main procedure for long division contains a complicated *while* statement that tests and prepares its own termination in the middle of the loop.

In his book on *Prime Numbers and Computer Methods for Factorization*, *Hans Riesel* (1985) makes the following observation: "Unfortunately...the performance of exact computations on large integers has a limited appeal, and computer manufacturers do not find it profitable to include such facilities in the software that goes with their hardware. This means that the reader may have to construct such a package himself for the computer he is using... [We] shall discuss ways in which this can be done."

1. Riesel gives no examples of long division.

2. According to Riesel: "Division is by far the most complicated of the four elementary operations." In spite of that, he immediately adds: "We shall only sketch. . .division." This warning is followed by a single page of hints with no theoretical analysis.

3. Without further explanation, Riesel presents a Pascal procedure of 45 lines, which uses *goto* statements (instead of *while* and *for* statements) to implement iteration. Although written in a standard language, this procedure is too hard to follow.

These evaluations of existing textbooks are written not for the sake of criticizing the authors, but to explain why it is necessary to discuss a fundamental algorithm that has been known for five centuries (Smith 1988). When you need multiple-length division on a computer, you will look in vain for a textbook that combines an elegant algorithm with a simple explanation.

In the following, I illustrate the subtleties of long division by examples, define the problem concisely, summarize the theory, and develop a complete SuperPascal algorithm using a consistent terminology. I also derive the complexity of the algorithm and explain how the radix is selected. The Appendix contains proofs of the theorems on which the algorithm is based.

LONG DIVISION

Most computers limit integer arithmetic to operands of 32–64 bits, corresponding to 8–17 decimal digits. A larger integer must be represented by an array of digits, each occupying a single machine word. The arithmetic operations on multiple-length integers are serial operations that imitate paper-and-pencil methods.

If a machine word represents a decimal digit, a 100-digit decimal number requires 100 machine words. However, if you use radix 1000 (instead of ten), the same number occupies only 34 words. A large radix reduces both the memory space of multiple-length integers and the execution time of serial arithmetic.

Multiple-length division is surprisingly difficult. Example 1 illustrates *long division* of decimal numbers, as we learned it in school. The initial remainder is the dividend 316097 extended with a leading zero: 0316097. (The purpose of the extra digit will soon become apparent.)

Example 1:

```
     3098
102 0316097
    0306
     0100
     0000
      1009
      0918
       0917
       0816
        101
```

The quotient digits are computed one at a time:

1. Since the divisor has three digits, you divide the four leading digits of the remainder by the divisor:

   $$0316 \text{ div } 102 = 3$$

   This gives you the leading digit of the quotient. The remainder is then reduced to 010097, as shown.

2. You divide the four leading digits of the new remainder by the divisor to get the next quotient digit:

   $$0100 \text{ div } 102 = 0$$

   This leaves a remainder of 10097.

3. You use the same method to compute the last two digits of the quotient:

   $$1009 \text{ div } 102 = 9$$
   $$0917 \text{ div } 102 = 8$$

   The final remainder is 101.

In each step, you treat the four leading digits of the remainder r as an integer $r\{4\}$ and use the divisor d to compute a *quotient digit* q_k:

$$q_k = r\{4\} \text{ div } d$$

The integer $r\{4\}$ is called a *prefix* of the remainder.

The following table is a different representation of the division steps in Example 1:

k	$r\{4\}$	q_k
3	0316	3
2	0100	0
1	1009	9
0	0917	8

Without a zero in front of the initial remainder, the computation of the first quotient digit would be a special case

$$316 \text{ div } 102 = 3$$

requiring three digits of the remainder (instead of four).

If the divisor has many digits, a quotient digit can seldom be computed directly, but must be estimated and corrected, if necessary. The main challenge is to replace human intuition about this process by an efficient iterative algorithm.

The three leading digits of the remainder define a shorter *prefix* $r\{3\}$. Similarly, the two leading digits of the divisor define a *prefix* $d\{2\}$. I will use

$$r\{3\} \text{ div } d\{2\}$$

as an *initial estimate* of the quotient digit q_k. Since a decimal digit must be less than 10, the initial estimate q_e is defined as follows:

$$q_e = \min(r\{3\} \text{ div } d\{2\}, 9)$$

The *error* of the initial estimate is the difference

$$\Delta q = q_e - q_k$$

In Example 1, where $d\{2\} = 10$, the initial estimate produces the following sequence of digits:

k	$r\{4\}$	q_k	$r\{3\}$	q_e	Δq
3	0316	3	031	3	0
2	0100	0	010	1	1
1	1009	9	100	9	0
0	0917	8	091	9	1

In every step,

$$0 \leq \Delta q \leq 1$$

Later, you will see that this inequality always holds. So, the most obvious idea is to correct a *trial digit* q_t as follows:

$q_t := q_e$;
if $r\{4\} < d * q_t$ then $q_t := q_t - 1$

At the end of this iteration, $q_t = q_k$.

In Example 1, half of the initial guesses require a single correction. The number of corrections can be reduced dramatically by *scaling* the operands before the division: You multiply the divisor and the dividend by the same digit. The scaling must make the leading digit of the divisor at least equal to half of the radix without changing the length of the divisor. This is called *normalization*.

Example 2:

```
      3098
510 ⌊1580485
    1530
     0504
     0000
      5048
      4590
       4585
       4080
        505
```

Example 1 is normalized by multiplying both operands by five:

$$\begin{aligned} 102 * 5 &= 510 \\ 0316097 * 5 &= 1580485 \end{aligned}$$

The leading digit of the normalized divisor is equal to 5, which is half of the radix 10. The normalized divisor still has only three digits. Since the original dividend already has an extra digit-position, normalization does not change its length either. This is another reason for adding the extra digit.

After normalization, division proceeds as usual.

Finally, the remainder is divided by the scaling factor to obtain the remainder of the original problem:

$$505 \text{ div } 5 = 101$$

In each step, the initial estimate q_e is the prefix $r\{3\}$ divided by the prefix $d\{2\} = 51$:

k	$r\{4\}$	q_k	$r\{3\}$	q_e	Δq
3	1580	3	158	3	0
2	0504	0	050	0	0
1	5048	9	504	9	0
0	4585	8	458	8	0

In this example, the initial estimates are exact estimates of the corresponding quotient digits. However, in general, normalized division occasionally requires correction of an initial estimate. In decimal arithmetic, the average number of corrections is less than 20%. For higher radices, corrections are rarely necessary.

After this intuitive exploration, I am ready to analyze the problem concisely.

THE ESSENCE OF THE PROBLEM

I am considering long division of two natural numbers, x and y

$$\begin{aligned} q &= x \; div \; y \\ r &= x \; mod \; y \end{aligned} \tag{1}$$

where $x \geq 0$ and $y > 0$.

The quotient q and the remainder r are natural numbers which satisfy the constraints:

$$x = y * q + r \tag{2}$$

$$0 \leq r \leq y - 1 \tag{3}$$

Each number is represented by an array of digits in *radix b*. The *dividend* x has n digits

$$x = x_{n-1}b^{n-1} + x_{n-2}b^{n-2} + \cdots + x_0 \tag{4}$$

while the *divisor* y has m digits

$$y = y_{m-1}b^{m-1} + y_{m-2}b^{m-2} + \cdots + y_0 \tag{5}$$

Two *special cases* immediately arise:

1. $m = 1$: If the divisor is a single-digit number, I will use a very simple division algorithm. Since zero is a single-digit number, this algorithm will also detect overflow.

2. $m > n$: If the divisor is longer than the dividend, the quotient is zero, and the remainder is x.

In the theoretical analysis, I will concentrate on the remaining case

$$2 \le m \le n \tag{6}$$

where the divisor has at least two digits, and the dividend has at least as many digits as the divisor.

Before the division, the operands are multiplied by a *scaling factor*, $f \ge 1$, that will be defined later. The following abstract program defines *normalized division:*

```
var x, y, q, r, d, f: integer;
begin
  r := x * f;
  d := y * f;
  q := r div d;
  r := (r mod d) div f
end
```

The *normalized divisor*, $d = y * f$, has m digits

$$d = d_{m-1}b^{m-1} + d_{m-2}b^{m-2} + \cdots + d_0 \tag{7}$$

Since $y > 0$, and $f \ge 1$, you also have $d > 0$.

The *quotient* q has $n - m + 1$ digits:

$$q = q_{n-m}b^{n-m} + q_{n-m-1}b^{n-m-1} + \cdots + q_0 \tag{8}$$

The initial *remainder*, $r = x * f$, has $n + 1$ digits. Immediately before the computation of quotient digit q_k, the remainder has been reduced to $k + m + 1$ digits

$$r = r_{k+m}b^{k+m} + r_{k+m-1}b^{k+m-1} + \cdots + r_0 \tag{9}$$

where $0 \leq k \leq n - m$.

The leading $m + 1$ digits of the remainder define a *prefix* $r\{m + 1\}$:

$$r\{m + 1\} = r_{k+m}b^m + r_{k+m-1}b^{m-1} + \cdots + r_k \tag{10}$$

The essence of multiple-length division is the computation of a single *quotient digit* q_k

$$q_k = r\{m + 1\} \; div \; d \tag{11}$$

by iteration.

To simplify the algebra a bit, I will assume that the *radix* b is *even*, say, a power of two or ten:

$$b \; div \; 2 = b/2 \geq 1 \tag{12}$$

TRIAL ITERATION

The Initial Estimate

The computation of a quotient digit q_k is an iteration that decrements an initial estimate until it equals q_k. The most conservative guess is $b-1$, which requires $O(b)$ corrections. Fortunately, there is a much better choice.

The three leading digits of the remainder r define a *prefix* $r\{3\}$:

$$r\{3\} = r_{k+m}b^2 + r_{k+m-1}b + r_{k+m-2} \tag{13}$$

The *prefix* $d\{2\}$ consists of the two leading digits of the divisor d

$$d\{2\} = d_{m-1}b + d_{m-2} \tag{14}$$

where

$$2 \leq m \leq k + m \leq n \tag{15}$$

I assume that $r\{3\}$ and $d\{2\}$ can be represented as single-length integers.

I will use

$$q_e = \min(r\{3\} \; div \; d\{2\}, b - 1) \tag{16}$$

as the *initial estimate* of q_k. Obviously $0 \leq q_e \leq b - 1$.

The following theorems show that q_e is an excellent guess. They are due to Krishnamurthy and Nandi (1967). (The Appendix includes the proofs of all theorems used.)

Theorem 1: $q_k \le q_e$

Theorem 2: $q_e \le q_k + 1$

By combining Theorems 1 and 2 with the assumption that q_k is a digit, you obtain the inequality:

$$0 \le q_k \le q_e \le q_k + 1 \le b \tag{17}$$

So, the initial estimate q_e is either correct or off by 1.

Trial Correction

The computation of a *quotient digit* q_k is based on two simple theorems about any *trial digit* q_t:

Theorem 3: *If* $r\{m+1\} < d * q_t$ *then* $q_k < q_t$

Theorem 4: *If* $r\{m+1\} \ge d * q_t$ *then* $q_t \le q_k$

These theorems and (17) suggest the following *trial iteration:*

$q_t := q_e$;
if $r\{\text{m+1}\} < d * q_t$ then $q_t := q_t - 1$

Here is the same algorithm with assertions added:

{ $0 \le q_k \le q_e \le q_k + 1 \le b$ by (17) }
$q_t := q_e$;
{ $0 \le q_k \le q_t \le q_k + 1 \le b$ }
if $r\{\text{m+1}\} < d * q_t$ then
  { $0 \le q_k < q_t \le q_k + 1 \le b$ by Theorem 3 }
  $q_t := q_t - 1$
  { $0 \le q_k \le q_t < q_k + 1 \le b$ }
else
  { $0 \le q_k \le q_t \le q_k < b$ by Theorem 4 }

The *if* statement terminates with the postcondition

$$0 \le q_k \le q_t \le q_k \le b - 1$$

which implies that $q_t = q_k$.

I will show that the frequency of trial corrections depends on the *leading digit* $d\{1\}$ of the divisor d

$$d\{1\} = d_{m-1} \tag{18}$$

where $1 \leq d\{1\} \leq b - 1$.

The computation of quotient digit q_k is equivalent to the integer division

$$q_k = r\{m+1\}\ div\ d$$

which leaves a remainder

$$r\{m+1\}\ mod\ d = r\{m+1\} - d * q_k$$

where $0 \leq r\{m+1\} \bmod d < d$.

I assume that the relative remainder

$$z_k = (r\{m+1\}\ mod\ d)/d = r\{m+1\}/d - q_k$$

is a random variable with a uniform distribution between 0 and 1.

At this point, another theorem is needed:

Theorem 5: *If* $q_k = q_e - 1$ *then* $r\{m+1\}/d - q_k\ >\ 1 - 1/d\{1\}$

The probability that a correction is required is

$$\text{Prob}(q_k = q_e - 1) = \text{Prob}(z_k\ >\ 1 - 1/d\{1\})$$

In other words,

$$\text{Prob}(q_k = q_e - 1)\ <\ 1/d\{1\} \tag{19}$$

To reduce the number of corrections, you should obviously make the leading digit of the divisor as large as possible. This requires normalization of the operands.

Normalization

The divisor d is *normalized* if its leading digit is at least half of the radix b:

$$1 \leq b/2 \leq d\{1\} \leq b - 1 \tag{20}$$

This requirement implies that

$$1 \leq b^m/2 \leq d \leq b^m - 1 \tag{21}$$

The *leading digit* $y\{1\}$ of the original divisor y is

$$y\{1\} = y_{m-1} \tag{22}$$

where $1 \le y\{1\} \le b-1$. This digit determines the *scaling factor* f:

$$f = b \; div \; (y\{1\} + 1) \tag{23}$$

Obviously, $1 \le f \le b/2$.

The following theorem shows that normalization is done correctly:

Theorem 6: $b^m/2 \le y * f \le b^m - 1$

From now on, I assume that the divisor, $d = y * f$, is normalized. By (19) and (20), the probability that an initial estimate q_e needs to be decremented is bounded as follows:

$$\text{Prob}(q_k = q_e - 1) < 2/b$$

In decimal division, at least 80% of the initial estimates are correct. For radix 1,000, each guess is correct with probability 0.998.

SUPERPASCAL ALGORITHM

With this background, I will describe a SuperPascal algorithm for multiple-length division. The theoretical discussion introduced names for the most important constants and variables. The algorithm uses exactly the same terminology and is presented in bottom-up form. (If you prefer top-down design, please read the rest of this section backwards.)

Number Representation

A natural *number* x is represented by an array of $w+1$ digits in *radix* b, say

```
const b = 1000; w = 100;
type number = array [0..w] of integer;
var x: number;
```

The algorithm can divide natural numbers with 1 to w digits. During division, the remainder is extended with an additional position.

Every *digit* $x[k]$ is an integer in the range

$$0 \le x[k] \le b-1 \qquad \text{for } 0 \le k \le w$$

The kth digit of x represents the integer $x[k] * b^k$. I will use the words *left* and *right* to refer to the high-order and low-order positions of a number.

SupePascal extends Pascal with structured function types. Algorithm 1, defines the conversion of a single-length integer x to a multiple-length integer of type *number*.

```
function value(x: integer): number;
var y: number; i: integer;
begin
  for i := 0 to w do
    begin
      y[i] := x mod b;
      x := x div b
    end;
  value := y
end;
```

Algorithm 1: Value.

The *length* of a natural number is the number of significant digits in the corresponding array. Algorithm 2 determines the length of a number x by a linear search from left to right. The search stops at the first nonzero digit or at the rightmost digit, whichever is reached first. This variant of linear searching was derived in Brinch Hansen (1990).

```
function length(x: number): integer;
var i, j: integer;
begin
  i := w; j := 0;
  while i <> j do
    if x[i] <> 0 then j := i
    else i := i - 1;
  length := i + 1
end;
```

Algorithm 2: Length.

Partial Arithmetic

The simplest part of multiple-length division is the multiplication or division of a natural number by a single digit. To avoid confusing these *partial operations* with complete multiple-length operations, they are called *product, quotient,* and *remainder* (instead of multiply, divide, and modulo):

Function	Result
product(x,k)	x ∗ k
quotient(x,k)	x div k
remainder(x,k)	x mod k

Each operation involves a natural number x and a digit k. The functions are straightforward implementations of familiar paper-and-pencil methods.

A partial *product* is computed, digit by digit, from right to left using a *carry* (Algorithm 3). The effect of the *assume* statement is to halt the execution if *overflow* occurs.

```
function product(x: number;
  k: integer): number;
var carry, i, m, temp: integer;
begin
  m := length(x); carry := 0;
  for i := 0 to m − 1 do
    begin
      temp := x[i]∗k + carry;
      x[i] := temp mod b;
      carry := temp div b
    end;
  if m <= w then x[m] := carry
  else assume carry = 0;
  product := x
end;
```

Algorithm 3: Product.

The *quotient* of a partial division is calculated serially from left to right (Algorithm 4).

The *remainder* of a partial division is the last carry (compare Algorithms 4 and 5).

```
function quotient(x: number;
  k: integer): number;
var carry, i, m, temp: integer;
begin
  m := length(x); carry := 0;
  for i := m − 1 downto 0 do
    begin
      temp := carry∗b + x[i];
      x[i] := temp div k;
      carry := temp mod k
    end;
  quotient := x
end;
```

Algorithm 4: Quotient.

```
function remainder(x: number;
  k: integer): number;
var carry, i, m: integer;
begin
  m := length(x); carry := 0;
  for i := m − 1 downto 0 do
    carry := (carry∗b + x[i]) mod k;
  remainder := value(carry)
end;
```

Algorithm 5: Remainder.

Prefix Arithmetic

The computation of a quotient digit $q[k]$ breaks down into simpler *prefix operations*. In the following, $x[i..j]$ denotes digits i through j of a natural number x.

The assignment

$$q_t := \text{trialdigit}(r,\ d,\ k,\ m)$$

defines a *trial digit*, $q_t = q_e$, which is an initial estimate of q_k. The operands

of the *trial digit* function are prefixes of the remainder r and the divisor d

$$r\{3\} = r[k+m..k+m-2] \qquad d\{2\} = d[m-1..m-2]$$

where

$$2 \le m \le k+m \le w \tag{24}$$

The initial estimate is computed as described earlier (Algorithm 6). Strictly speaking, the trial function should verify that its *precondition* holds. However, since (24) turns out to be an invariant of long division, this assumption is described by a comment only.

```
function trialdigit(r, d: number;
  k, m: integer): integer;
var d2, km, r3: integer;
begin
  { 2 <= m <= k+m <= w }
  km := k + m;
  r3 := (r[km]*b + r[km−1])*b + r[km−2];
  d2 := d[m−1]*b + d[m−2];
  trialdigit := min(r3 div d2, b − 1)
end;
```

Algorithm 6: Trial-digit.

Two functions define prefix comparison and subtraction:

Function	Result
smaller(r, dq, k, m)	$r\{m+1\} < dq$
difference(r, dq, k, m)	$r - dq * b^k$

The (m+1)-place operands are

$$r\{m+1\} = r[k+m..k] \qquad d * q_t = dq[m..0]$$

where

$$0 \le k \le k+m \le w \tag{25}$$

Since the division invariant (24) implies the precondition (25), these functions do not verify this assumption.

The boolean function that determines if the prefix $r\{m+1\}$ is *smaller* than the product dq is a variant of the *length* function (see Algorithms 2 and 7).

```
function smaller(r, dq: number;
   k, m: integer): boolean
{ r[k+m..k] < dq[m..0] };
var i, j: integer;
begin
   { 0 <= k <= k+m <= w }
   i := m; j := 0;
   while i <> j do
      if r[i+k] <> dq[i]
         then j := i
         else i := i - 1;
   smaller := r[i+k] < dq[i]
end;
```

Algorithm 7: Smaller.

The subtraction of the product dq from the $m + 1$ high-order digits of the remainder r proceeds from right to left using a *borrow*. If the *difference* is negative, the execution halts (Algorithm 8).

Division Algorithms

Algorithm 9 defines *long division*

$$x \text{ div } y$$

of two natural numbers

$$x[n-1..0] \qquad y[m-1..0]$$

where

$$2 \leq m \leq n \leq w \tag{26}$$

After normalization of the operands, the *quotient* q is computed, digit by digit, from left to right. The quotient iteration maintains the invariant

$$2 \leq m \leq k+m \leq n \leq w \tag{27}$$

The *long modulo* operation

$$x \text{ mod } y$$

```
function difference(r, dq: number;
  k, m: integer): number;
{ r[k+m..k] :=
    r[k+m..k] − dq[m..0];
  difference := r }
var borrow, diff, i: integer;
begin
  { 0 <= k <= k+m <= w }
  borrow := 0;
  for i := 0 to m do
    begin
      diff := r[i+k] − dq[i]
        − borrow + b;
      r[i+k] := diff mod b;
      borrow := 1 − diff div b
    end;
  assume borrow = 0;
  difference := r
end;
```

Algorithm 8: Difference.

is very similar (Algorithm 10). Notice that the final *remainder* r is divided by the *scaling factor* f.

The complete algorithm for computing the quotient of a *multiple-length division* uses simpler methods for the special cases, where the *divisor* y has only a single digit or is longer than the *dividend* x (Algorithm 11). In all other cases, the algorithm uses long division after establishing the precondition (26).

Algorithm 12 shows the complete algorithm for computing the remainder of a multiple-length division.

Complexity

The complexity of long division is determined by the $n-m+1$ quotient steps. Each step is dominated by product and difference operations on $(m+1)$-place operands. Consequently, the complexity is

$$O((n-m+1)(m+1))$$

```
function longdiv(x, y: number;
   n, m: integer): number;
{ longdiv = x div y }
var d, dq, q, r: number;
   f, k, qt: integer;
begin
   { 2 <= m <= n <= w }
   f := b div (y[m−1] + 1);
   r := product(x, f);
   d := product(y, f);
   q := value(0);
   for k := n − m downto 0 do
      begin
         { 2 <= m <= k+m <= n <= w }
         qt := trialdigit(r, d, k, m);
         dq := product(d, qt);
         if smaller(r, dq, k, m) then
            begin
               qt := qt − 1;
               dq := product(d, qt)
            end;
         q[k] := qt;
         r := difference(r, dq, k, m)
      end;
   longdiv := q
end;
```

Algorithm 9: Long divide.

For random divisors with uniformly distributed lengths between 2 and n, the average run time T is close to

$$T \approx \frac{1}{n}\int_2^n (n-m+1)(m+1)dm = O(n^2) \qquad \text{for } n \gg 2$$

Since

$$10^N \approx b^n \qquad \text{for } n \approx N/\log b$$

a decimal number with N digits corresponds to a radix-b number with $N/\log b$

```
function longmod(x, y: number;
  n, m: integer): number;
{ longmod = x mod y }
var d, dq, r: number;
  f, k, qt: integer;
begin
  { 2 <= m <= n <= w }
  f := b div (y[m−1] + 1);
  r := product(x, f);
  d := product(y, f);
  for k := n − m downto 0 do
    begin
      { 2 <= m <= k+m <= n <= w }
      qt := trialdigit(r, d, k, m);
      dq := product(d, qt);
      if smaller(r, dq, k, m) then
        begin
          qt := qt − 1;
          dq := product(d, qt)
        end;
      r := difference(r, dq, k, m)
    end;
  longmod := quotient(r, f)
end;
```

Algorithm 10: Long modulo.

digits. Consequently,

$$T \approx O\left((N/\log b)^2\right)$$

Using radix 1000, instead of ten, reduces the average division time by a factor of 9. Radix 10,000 makes multiple-length division 16 times faster than decimal division.

Radix Selection

The algorithms in Algorithms 4–5 include integer expressions of the form

$$d * b + d$$

```
function divide(x, y: number): number;
var m, n, y1: integer; q: number;
begin
  m := length(y);
  if m = 1 then
    begin
      y1 := y[m−1];
      assume y1 > 0;
      q := quotient(x, y1)
    end
  else
    begin
      n := length(x);
      if m > n then q := value(0)
      else { 2 <= m <= n <= w }
        q := longdiv(x, y, n, m)
    end;
  divide := q
end;
```

Algorithm 11: Divide.

where the d's denote arbitrary digits. The corresponding values can be represented by standard integers if you choose a radix b that satisfies the inequality

$$d * b + d \leq b^2 - 1 \leq maxint$$

where *maxint* is the largest standard integer.

For 32-bit integers, this means

$$b^2 \leq 2^{31}$$

or $b \leq 46{,}340$. A possible choice is $b = 10{,}000$.

However, the *trial digit* function (Algorithm 6) has an expression of the form

$$(d * b + d) * b + d$$

The corresponding inequality is

$$(d * b + d) * b + d \leq b^3 - 1 \leq maxint$$

```
function modulo(x, y: number): number;
var m, n, y1: integer; r: number;
begin
  m := length(y);
  if m = 1 then
    begin
      y1 := y[m−1];
      assume y1 > 0;
      r := remainder(x, y1)
    end
  else
    begin
      n := length(x);
      if m > n then r := x
      else { 2 <= m <= n <= w }
        r := longmod(x, y, n, m)
    end;
  modulo := r
end;
```

Algorithm 12: Modulo.

For 32-bit integers, you must therefore use a smaller radix, for which

$$b^3 \leq 2^{31}$$

that is, $b \leq 1{,}290$. For example, $b = 1{,}000$.

If the available computer supports both 32 and 64-bit integers, a radix of 10,000 is feasible if you use *double-precision arithmetic* in the trial digit function.

FINAL REMARKS

I have developed a Pascal algorithm for long division of natural numbers and have explained the theory behind the algorithm. This chapter is merely an interpretation and formalization of the original ideas of the authors referenced in the text. It was written only because I was unable to find an elegant, complete algorithm described with convincing clarity in the literature.

Acknowledgements

I thank Konrad Jahn who taught me how to divide in the fourth grade in Denmark. I am indebted to Jonathan Greenfield who rediscoverecd Theorem 2 and brought it to my attention. It is also a pleasure to acknowledge the comments of Suad Alagić, Christian Gram, Hans Riesel and Peter Villemoes.

APPENDIX: PROOF OF THEOREMS

The appendix summarizes the laws of prefixes and integer division and proves the theorems about long division.

Prefix Laws

By (5), (22):

$$y\{1\}b^{m-1} \;\le\; y \;<\; (y\{1\}+1)b^{m-1} \tag{28}$$

By (7), (18):

$$d\{1\}b^{m-1} \;\le\; d \;<\; (d\{1\}+1)b^{m-1} \tag{29}$$

By (7), (14):

$$d\{2\}b^{m-2} \;\le\; d \;<\; (d\{2\}+1)b^{m-2} \tag{30}$$

By (10), (13):

$$r\{3\}b^{m-2} \;\le\; r\{m+1\} \;<\; (r\{3\}+1)b^{m-2} \tag{31}$$

Division Laws

Any integer division, $q = x \text{ div } y$, satisfies three equivalent laws:

$$x/y - 1 \;<\; q \;\le\; x/y \tag{32}$$

$$x \;<\; y(q+1) \;\le\; x+y \tag{33}$$

$$x - y + 1 \;\le\; y * q \;\le\; x \tag{34}$$

The following instances of these laws apply to the integer divisions that define the quotients, f, q_e, and q_k, where

$$\begin{aligned} f &= b \; div \; (y\{1\}+1) \\ q_e &= \min(r\{3\} \; div \; d\{2\}, b-1) \\ q_k &= r\{m+1\} \; div \; d \end{aligned}$$

By (32):

$$b/(y\{1\}+1) - 1 \;<\; f \;\leq\; b/(y\{1\}+1) \tag{35}$$

If $q_e = r\{3\}\ div\ d\{2\}$, then by (34):

$$r\{3\} - d\{2\} + 1 \;\leq\; d\{2\} * q_e \;\leq\; r\{3\} \tag{36}$$

By (34):

$$r\{m+1\} - d + 1 \;\leq\; d * q_k \;\leq\; r\{m+1\} \tag{37}$$

By (33):

$$r\{m+1\} \;<\; d(q_k+1) \;\leq\; r\{m+1\} + d \tag{38}$$

Proof of Theorem 1

Theorem: $q_k \;\leq\; q_e$

Proof:

1. If $q_e = r\{3\}\ div\ d\{2\}$, then

$$\begin{array}{rcl} d(q_k - q_e) & \leq & r\{m+1\} - d\{2\}q_e b^{m-2} \\ & & \text{by (30), (37)} \\ & < & ((r\{3\}+1) - (r\{3\} - d\{2\} + 1))b^{m-2} \\ & & \text{by (31), (36)} \\ & = & d\{2\}b^{m-2} \\ & \leq & d \qquad \text{by (30)} \end{array}$$

   Since $d > 0$, you have $q_k - q_e < 1$, which implies $q_k - q_e \leq 0$.

2. If $q_e = b - 1$, then

$$q_k \;\leq\; q_e \qquad \text{by } q_k \;\leq\; b-1$$

Proof of Theorem 2

Theorem: $q_e \;\leq\; q_k + 1$

Proof by contradiction: If $q_e \;\geq\; q_k + 2$, then

$$\begin{array}{rcll} r\{m+1\} & < & d(q_k+1) & \text{by (38)} \\ & \leq & d(q_e-1) & \text{by } q_e \;\geq\; q_k+2 \\ & < & (d\{2\}+1)q_e b^{m-2} - d & \text{by (30)} \\ & \leq & (r\{3\}+q_e)b^{m-2} - d & \text{by (36)} \\ & < & (r\{3\}+b)b^{m-2} - d & \text{by } q_e \;<\; b \\ & \leq & r\{3\}b^{m-2} & \text{by } d \;\geq\; b^{m-1} \\ & \leq & r\{m+1\} & \text{by (31)} \end{array}$$

that is, $r\{m+1\} < r\{m+1\}$, a contradiction.

Proof of Theorem 3

Theorem: *If* $r\{m+1\} < d * q_t$ *then* $q_k < q_t$

Proof:

$$\begin{array}{rcll} d(q_k - q_t) & \leq & r\{m+1\} - d * q_t & \text{by (37)} \\ & < & 0 & \text{by } r\{m+1\} < d * q_t \end{array}$$

Since $d > 0$, you have $q_k - q_t < 0$.

Proof of Theorem 4

Theorem: *If* $r\{m+1\} \geq d * q_t$ *then* $q_t \leq q_k$

Proof by contradiction: If $q_t \geq q_k + 1$, then

$$\begin{array}{rcll} r\{m+1\} & < & d(q_k+1) & \text{by (38)} \\ & \leq & d * q_t & \text{by } q_t \geq q_k + 1 \\ & \leq & r\{m+1\} & \text{by } r\{m+1\} \geq d * q_t \end{array}$$

that is, $r\{m+1\} < r\{m+1\}$, a contradiction.

Proof of Theorem 5

Theorem: *If* $q_k = q_e - 1$ *then* $r\{m+1\}/d - q_k > 1 - 1/d\{1\}$

Proof:

$$\begin{array}{rcll} r\{m+1\} - d * q_k & = & d + r\{m+1\} - d * q_e & \\ & & \text{by } q_k = q_e - 1 & \\ & > & d + (r\{3\} - (d\{2\} + 1)q_e)b^{m-2} & \\ & & \text{by (30), (31)} & \\ & \geq & d - q_e b^{m-2} & \text{by (36)} \\ & > & d - b^{m-1} & \text{by } q_e < b \\ & = & d(1 - b^{m-1}/d) & \text{by } d > 0 \\ & \geq & d(1 - 1/d\{1\}) & \text{by (29)} \end{array}$$

Consequently, $r\{m+1\}/d - q_k > 1 - 1/d\{1\}$.

Proof of Theorem 6

Theorem: $b^m/2 \le y * f \le b^m - 1$

1. Upper bound:

$$\begin{array}{rcll} y * f & \le & ((y\{1\}+1)b^{m-1}-1)f & \text{by (28)} \\ & \le & (y\{1\}+1)f * b^{m-1} - 1 & \text{by } f \ge 1 \\ & \le & b^m - 1 & \text{if } f \le b/(y\{1\}+1) \end{array}$$

   The largest possible scaling factor is

$$f = b \; div \; (y\{1\}+1)$$

2. Lower bound:

   (a) If $y\{1\} < b/2$, then

$$\begin{array}{rcl} y\{1\}f & > & y\{1\}(b/(y\{1\}+1)-1) \qquad \text{by (35)} \\ & = & (b/2 - y\{1\} - 1)(y\{1\}-1)/(y\{1\}+1) + b/2 - 1 \\ & & \text{by algebra} \\ & \ge & b/2 - 1 \qquad \text{by } 1 \le y\{1\} \le b/2 - 1 \end{array}$$

   Since $y\{1\}f > b/2 - 1$ and $b/2$ is an integer, you have $y\{1\}f \ge b/2$. Consequently,

$$\begin{array}{rcll} y * f & \ge & y\{1\}f * b^{m-1} & \text{by (28)} \\ & \ge & b^m/2 & \text{by } y\{1\}f \ge b/2 \end{array}$$

   (b) If $y\{1\} \ge b/2$, then

$$\begin{array}{rcll} y * f & \ge & y\{1\}b^{m-1} & \text{by } f \ge 1, \text{ (28)} \\ & \ge & b^m/2 & \text{by } y\{1\} \ge b/2 \end{array}$$

References

P. Brinch Hansen and R. House, The Cobol compiler for the Siemens 3003. *BIT 6*, 1–23 (1966).

R. L. Rivest, A. Shamir and L. M. Adleman, A method for obtaining digital signatures for public-key cryptography. *Communications of the ACM 21*, 120–126 (1978).

M. O. Rabin, Probabilistic algorithms for testing primality. *Journal of Number Theory 12*, 128–138 (1980).

P. Brinch Hansen, Primality testing. School of Computer and Information Science, Syracuse University, Syracuse, NY, 1992.

D. Knuth, *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, MA, 1969.

S. A. Alagic and M. A. Arbib, *The Design of Well-Structured and Correct Programs*. Springer-Verlag, New York, 1978.

D. Wood, *Paradigms and Programming with Pascal*. Computer Science Press, Rockville, MD, 1984.

H. Riesel, *Prime Numbers and Computer Methods for Factorization*. Birkhäuser, Boston, MA, 1985.

D. E. Smith and J. Ginsburg, From numbers to numerals and from numerals to computation. in J. R. Newman (ed.), *The World of Mathematics, Volume 1*, Tempus, Redmond, WA, 1988, 433–454.

E. V. Krishnarnurthy and S. K. Nandi, On the normalization requirement of divisor in divide-and-correct methods. *Communications of the ACM 12*, 809–813 (1967).

P. Brinch Hansen, The linear search rediscovered. *Structured Programming 11*, 53–55 (1990).