1

Appetizer: Integer Arithmetic

An appetizer is supposed to stimulate the appetite at the beginning of a meal. This is exactly the purpose of this chapter. We want to stimulate your interest in algorithmic[1] *techniques by showing you a surprising result. Although, the school method for multiplying integers has been familiar to all of us since our school days and seems to be the natural way to multiply integers, it is* not *the best multiplication algorithm; there are much faster ways to multiply large integers, i.e., integers with thousands or even millions of digits, and we shall teach you one of them.*

Algorithms for arithmetic are not only among the oldest algorithms, but are also essential in areas such as cryptography, geometric computing, computer algebra, and computer architecture. The three former areas need software algorithms for arithmetic on long integers, i.e., numbers with up to millions of digits. We shall present the algorithms learned in school, but also an improved algorithm for multiplication. The improved multiplication algorithm is not just an intellectual gem but is also extensively used in applications. Computer architecture needs very fast algorithms for moderate-length (32 to 128 bits) integers. We shall present fast parallel algorithms suitable for hardware implementation for addition and multiplication. On the way, we shall learn basic algorithm analysis and basic algorithm engineering techniques in a simple setting. We shall also see the interplay of theory and experiment.

We assume that integers[2] are represented as digit strings. In the base B number system, where B is an integer larger than 1, there are digits $0, 1, \ldots, B-1$, and a digit string $a_{n-1}a_{n-2}\ldots a_1a_0$ represents the number $\sum_{0\le i<n} a_iB^i$. The most important systems with a small value of B are base 2, with digits 0 and 1, base 10, with digits 0

[1] The Soviet stamp on this page shows Muhammad ibn Musa al-Khwarizmi (born approximately 780; died between 835 and 850), Persian mathematician and astronomer from the Khorasan province of present-day Uzbekistan. The word "algorithm" is derived from his name.

[2] Throughout this chapter, we use "integer" to mean a nonnegative integer.

to 9, and base 16, with digits 0 to 15 (frequently written as 0 to 9, A, B, C, D, E, and F). Larger bases, such as 2^8, 2^{16}, 2^{32}, and 2^{64}, are also useful. For example,

$$\begin{aligned} \text{“10101” in base 2 represents} \quad & 1\cdot 2^4+0\cdot 2^3+1\cdot 2^2+0\cdot 2^1+1\cdot 2^0 = 21, \\ \text{“924” in base 10 represents} \quad & 9\cdot 10^2+2\cdot 10^1+4\cdot 10^0 = 924. \end{aligned}$$

We assume that we have two primitive operations at our disposal: the addition of three digits with a two-digit result (this is sometimes called a full adder), and the multiplication of two digits with a two-digit result.[3] For example, in base 10, we have

$$\begin{array}{r} 3 \\ 5 \\ 5 \\ \hline 13 \end{array} \qquad \text{and} \qquad 6\cdot 7 = 42.$$

We shall measure the efficiency of our algorithms by the number of primitive operations executed.

We can artificially turn any n-digit integer into an m-digit integer for any $m \geq n$ by adding additional leading 0's. Concretely, “425” and “000425” represent the same integer. We shall use a and b for the two operands of an addition or multiplication and assume throughout this chapter that a and b are n-digit integers. The assumption that both operands have the same length simplifies the presentation without changing the key message of the chapter. We shall come back to this remark at the end of the chapter. We refer to the digits of a as a_{n-1} to a_0, with a_{n-1} being the most significant digit (also called leading digit) and a_0 being the least significant digit, and write $a = (a_{n-1}\dots a_0)$. The leading digit may be 0. Similarly, we use b_{n-1} to b_0 to denote the digits of b, and write $b = (b_{n-1}\dots b_0)$.

1.1 Addition

We all know how to add two integers $a = (a_{n-1}\dots a_0)$ and $b = (b_{n-1}\dots b_0)$. We simply write one under the other with the least significant digits aligned, and sum the integers digitwise, carrying a single digit from one position to the next. This digit is called a *carry*. The result will be an $(n+1)$-digit integer $s = (s_n\dots s_0)$. Graphically,

$$\begin{array}{rr} a_{n-1}\ \dots\ a_1\ a_0 & \text{first operand} \\ b_{n-1}\ \dots\ b_1\ b_0 & \text{second operand} \\ c_n\ c_{n-1}\ \dots\ c_1\ \ 0 & \text{carries} \\ \hline s_n\ s_{n-1}\ \dots\ s_1\ s_0 & \text{sum} \end{array}$$

[3] Observe that the sum of three digits is at most $3(B-1)$ and the product of two digits is at most $(B-1)^2$, and that both expressions are bounded by $(B-1)\cdot B^1+(B-1)\cdot B^0 = B^2-1$, the largest integer that can be written with two digits. This clearly holds for $B=2$. Increasing B by 1 increases the first expression by 3, the second expression by $2B-1$, and the third expression by $2B+1$. Hence the claim holds for all B.

where $c_0, \ldots, c_n$ is the sequence of carries and $s = (s_n \ldots s_0)$ is the sum. We have $c_0 = 0$, $c_{i+1} \cdot B + s_i = a_i + b_i + c_i$ for $0 \leq i < n$, and $s_n = c_n$. As a program, this is written as

$c = 0 : \mathit{Digit}$ // Variable for the carry digit
for $i := 0$ **to** $n-1$ **do** add a_i, b_i, and c to form s_i and a new carry c
$s_n := c$

We need one primitive operation for each position, and hence a total of n primitive operations.

Theorem 1.1. *The addition of two n-digit integers requires exactly n primitive operations. The result is an* $(n+1)$*-digit integer.*

1.1.1 Parallel Addition

Our addition algorithm produces the result digits one after the other, from least significant to most significant. In fact, the ith carry depends on the $(i-1)$th carry, which in turn depends on the $(i-2)$nd carry, and so on. So, it seems natural to compute the result digits sequentially. Is there a parallel algorithm for addition that produces all digits in time less than linear in the number of digits? Parallel addition is not an academic exercise but crucial for microprocessor technology, because processors need fast, hardware-implemented algorithms for arithmetic. Suppose that engineers had insisted on using serial addition algorithms for microprocessors. In that case it is likely that we would still be using 8-bit processors wherever possible, since they would be up to eight times faster than 64-bit processors.

Our strategy for parallel addition is simple and ambitious – we want to perform all digit additions $a_i + b_i + c_i$ in parallel. Of course, the problem is how to compute the carries c_i in parallel. Consider the following example of the addition of two 8-digit binary numbers a and b:

position	8	7	6	5	4	3	2	1	0
a	1	0	0	0	0	1	1	1	
b	0	1	0	0	1	0	1	0	
c	0	0	0	0	1	1	1	0	0
x	p	p	s	s	p	p	g	p	
y	s	s	s	s	g	g	g	p	

Row c indicates the carries. Why is there a carry into position 4, why is $c_4 = 1$? Because position 1 generates a carry, since $a_1 + b_1 = 2$, and positions 2 and 3 propagate it, since $a_2 + b_2 = a_3 + b_3 = 1$. Why is there no carry into position 8, why is $c_8 = 0$? Positions 6 and 7 would propagate a carry, since $a_6 + b_6 = a_7 + b_7 = 1$, but there is nothing to propagate, since no carry is generated in position 5 since $a_5 + b_5 = 0$. The general rule is: We have a carry into a certain position if a carry is generated somewhere to the right and then propagated through all intermediate positions.

Rows x and y implement this rule. Row x indicates whether a position generates (g), propagates (p), or stops (s) a carry. We have

$$x_i = \begin{cases} \mathtt{g} & \text{if } a_i + b_i = 2, \\ \mathtt{p} & \text{if } a_i + b_i = 1, \\ \mathtt{s} & \text{if } a_i + b_i = 0. \end{cases}$$

The different x_i's are independent and can be computed in parallel. Row y gives information whether we have a carry into a certain position. For $i \geq 1$, we have a carry into position i, i.e., $c_i = 1$, if and only if $y_{i-1} = \mathtt{g}$. We never have a carry into position 0, i.e., $c_0 = 0$. We give two rules for determining the sequence of y's. Consider the maximal subsequences of the x-sequence consisting of a string of p's followed by either an s or a g or the right end. Within each subsequence, turn all symbols into the last symbol; leave the sequence of trailing p's unchanged. In our example, the maximal subsequences are $x_7x_6x_5$, x_4, and $x_3x_2x_1$, and the sequence of trailing p's consists of x_0. Therefore $y_7y_6y_5$ become s's, y_4 becomes s, $y_3y_2y_1$ become g's, and y_0 becomes p. It would be equally fine to turn the trailing p's into s's. However, this would make the formal treatment slightly more cumbersome. Alternatively, we may state the rule for the y_i's as follows:

$$y_i = \begin{cases} x_i & \text{if } i = 0 \text{ or } x_i \in \{\mathtt{s}, \mathtt{g}\}, \\ y_{i-1} & \text{otherwise, i.e., } i > 0 \text{ and } x_i = \mathtt{p}. \end{cases}$$

Our task is now to compute the y_i's in parallel. The first step is to rewrite the definition of y_i as

$$y_i = \begin{cases} x_0 & \text{if } i = 0, \\ x_i \otimes y_{i-1} & \text{if } i > 0, \end{cases} \qquad \text{where} \qquad \begin{array}{c|ccc} \otimes & \mathtt{s} & \mathtt{p} & \mathtt{g} \\ \hline \mathtt{s} & \mathtt{s} & \mathtt{s} & \mathtt{s} \\ \mathtt{p} & \mathtt{s} & \mathtt{p} & \mathtt{g} \\ \mathtt{g} & \mathtt{g} & \mathtt{g} & \mathtt{g} \end{array}$$

The operator $\otimes$[4] returns its left argument if this argument is s or g and returns its right argument when the left argument is p. We next expand the definition of y_i and obtain

$$y_i = x_i \otimes y_{i-1} = x_i \otimes (x_{i-1} \otimes y_{i-2}) = x_i \otimes (x_{i-1} \otimes (x_{i-2} \ldots (x_1 \otimes x_0) \ldots)).$$

This formula corresponds to the sequential computation of y_i. We first compute $x_1 \otimes x_0$, then left-multiply by x_2, then by x_3, and finally by x_i. The operator $\otimes$ is associative (we shall prove this below) and hence we can change the evaluation order without changing the result. Compare the following formulae for y_6:

$$y_6 = x_6 \otimes (x_5 \otimes (x_4 \otimes (x_3 \otimes (x_2 \otimes (x_1 \otimes x_0)))))$$

and

$$y_6 = (x_6 \otimes ((x_5 \otimes x_4) \otimes ((x_3 \otimes x_2) \otimes (x_1 \otimes x_0)).$$

[4] Pronounced "otimes".

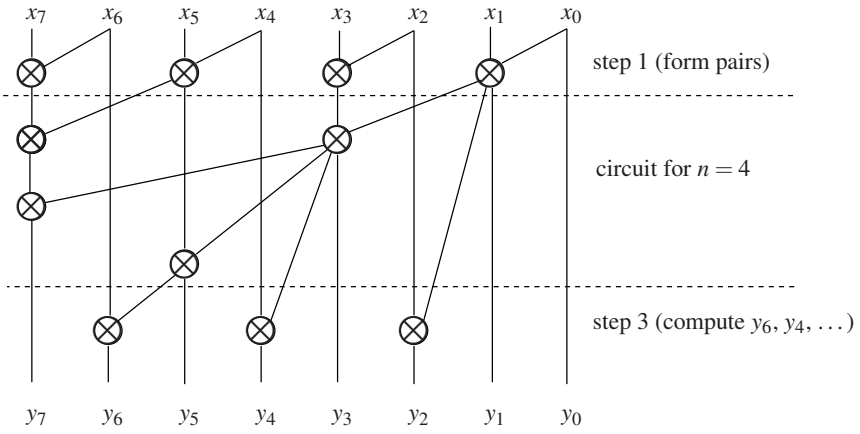

Fig. 1.1. The computation of y_7 to y_0 from x_7 to x_0. The horizontal partition of the computation corresponds to the general description of the algorithm. We first form pairs. In the second row of gates, we combine pairs, and in the third row, we combine pairs of pairs. Then we start to fill in gaps. In row four, we compute y_5, and in the last row, we compute y_6, y_4, and y_2.

The latter formula corresponds to a parallel evaluation of y_6. We compute $x_5 \otimes x_4$, $x_3 \otimes x_2$, and $x_1 \otimes x_0$ in parallel, then $x_6 \otimes (x_5 \otimes x_4)$ and $(x_3 \otimes x_2) \otimes (x_1 \otimes x_0)$, and finally y_6. Thus three rounds of computation suffice instead of the six rounds for the sequential evaluation. Generally, we can compute y_i in $\lceil \log i \rceil$ rounds of computation. The computation of the different y_i's can be intertwined, as Fig. 1.1 shows.

We next give a more formal treatment for a general base B. We start with the simple observation that the carry digit is either 0 or 1.

Lemma 1.2. *$c_i \in \{0,1\}$ for all i.*

Proof. We have $c_0 = 0$ by definition. Assume inductively, that $c_i \in \{0,1\}$. Then $a_i + b_i + c_i \leq 2(B-1) + 1 = 2B - 1 = 1 \cdot B + B - 1$ and hence $c_{i+1} \in \{0,1\}$. □

Two input digits a_i and b_i will *generate* a carry no matter what happens to the right of them if $a_i + b_i > B - 1$. On the other hand, the addition involving a_i and b_i will *stop* any carry if $a_i + b_i < B - 1$; this holds because an incoming carry is at most 1. If $a_i + b_i = B - 1$, an incoming carry will be *propagated* to the left. Hence, we define

$$x_i := \begin{cases} \mathtt{s} & \text{if } a_i + b_i < B - 1 \text{ (stop)}, \\ \mathtt{p} & \text{if } a_i + b_i = B - 1 \text{ (propagate)}, \\ \mathtt{g} & \text{if } a_i + b_i > B - 1 \text{ (generate)}. \end{cases} \tag{1.1}$$

Lemma 1.3. *The operator $\otimes$ is associative, i.e., $(u \otimes v) \otimes w = u \otimes (v \otimes w)$ for any u, v, and w. Let $y_i = \bigotimes_{0 \leq j \leq i} x_j$. Then $c_i = 1$ if and only if $y_{i-1} = \mathtt{g}$.*

Proof. If $u \neq \mathtt{p}$, then $(u \otimes v) \otimes w = u \otimes w = u = u \otimes (v \otimes w)$. If $u = \mathtt{p}$, then $(u \otimes v) \otimes w = v \otimes w = u \otimes (v \otimes w)$.

We have a carry into position i if and only if there is a $k < i$ such that a carry is generated in position k and propagated by positions $k+1$ to $i-1$, i.e., $x_k = \mathsf{g}$ and $x_{k+1} = \ldots x_{i-1} = \mathsf{p}$. Then $y_k = x_k = \mathsf{g}$. This completes the argument when $i-1 = k$. Otherwise, $y_{k+1} = \mathsf{p} \otimes y_k = \mathsf{g}$, $y_{k+2} = \mathsf{p} \otimes y_{k+1} = \mathsf{g}$, $\ldots$, $y_{i-1} = \mathsf{p} \otimes y_{i-2} = \mathsf{g}$. □

We now come to the parallel computation of the y_i's. For simplicity, we assume n to be a power of 2. If $n = 1 = 2^0$, we simply return x_0. Otherwise, we do the following:

(a) Combine consecutive pairs of inputs: $z_0 = x_1 \otimes x_0$, $z_1 = x_3 \otimes x_2$, $\ldots$, $z_{n/2-1} = x_{n-1} \otimes x_{n-2}$.
(b) Apply the algorithm recursively to the sequence $z_{n/2-1}, \ldots, z_0$ to obtain its sequence of prefix sums $w_{n/2-1}, \ldots, w_0$.
(c) Assemble the output as $y_{2i+1} = w_i$ for $i \in 0..n/2-1$ and $y_0 = x_0$ and $y_{2i} = x_{2i} \otimes w_{i-1}$ for $i \in 1..n/2-1$.

Figure 1.1 illustrates the computation for $n = 8$. The number $N(n)$ of gates satisfies $N(1) = 0$ and $N(n) = n/2 + N(n/2) + n/2 - 1 \le N(n/2) + n$. Note that we use $n/2$ gates in step (a), $n/2-1$ gates in step (c), and $N(n/2)$ gates in step (b). By repeated substitution, we obtain $N(n) \le n + N(n/2) \le n + n/2 + n/4 + n/8 + \ldots = \mathrm{O}(n)$. The number of rounds of computation is 0 for $n = 1$, is 1 for $n = 2$, and grows by 2 whenever n doubles. Thus the number of rounds is $2\log n - 1$.

The algorithm for computing the y_i's in parallel is an example of a *prefix sum* computation – a frequently useful collective computation described in more detail in Sect. 13.3.

1.2 Multiplication: The School Method

We all know how to multiply two integers. In this section, we shall review the "school method". In a later section, we shall get to know a method which is significantly faster for large integers.

We shall proceed slowly. We first review how to multiply an n-digit integer a by a one-digit integer b_j. We use b_j for the one-digit integer, since this is how we need it below. For any digit a_i of a, we form the product $a_i \cdot b_j$. The result is a two-digit integer $(c_i d_i)$, i.e.,

$$a_i \cdot b_j = c_i \cdot B + d_i.$$

We form two integers, $c = (c_{n-1} \ldots c_0\, 0)$ and $d = (d_{n-1} \ldots d_0)$ from the c_i's and d_i's, respectively. Since the c's are the higher-order digits in the products, we add a 0 digit at the end. We add c and d to obtain the product $p_j = a \cdot b_j$. Graphically,

$$(a_{n-1} \ldots a_i \ldots a_0) \cdot b_j \quad \longrightarrow \quad \begin{array}{r} c_{n-1}\ c_{n-2}\ \ldots\ c_i \quad\ c_{i-1}\ \ldots\ c_0\ 0 \\ \underline{\quad d_{n-1}\ \ldots\ d_{i+1}\ d_i \quad\ \ldots\ d_1\ d_0} \\ \text{sum of } c \text{ and } d \end{array}$$

Let us determine the number of primitive operations. For each i, we need one primitive operation to form the product $a_i \cdot b_j$, for a total of n primitive operations. Then

we add two $(n+1)$-digit numbers $(c_{n-1}\dots c_0 0)$ and $(0d_{n-1}\dots d_0)$. However, we may simply copy the digit d_0 to the result and hence effectively add two n-digit numbers. This requires n primitive operations. So the total number of primitive operations is $2n$. The leftmost addition of c_{n-1} and the carry into this position generates carry 0 as $a \cdot b_j \leq (B^n - 1) \cdot (B-1) < B^{n+1}$, and hence the result can be written with $n+1$ digits.

Lemma 1.4. *We can multiply an n-digit number by a one-digit number with $2n$ primitive operations. The result is an $(n+1)$-digit number.*

When you multiply an n-digit number by a one-digit number, you will probably proceed slightly differently. You combine[5] the generation of the products $a_i \cdot b_j$ with the summation of c and d into a single phase, i.e., you create the digits of c and d when they are needed in the final addition. We have chosen to generate them in a separate phase because this simplifies the description of the algorithm.

Exercise 1.1. Give a program for the multiplication of a and b_j that operates in a single phase.

We can now turn to the multiplication of two n-digit integers. The school method for integer multiplication works as follows: We first form partial products p_j by multiplying a by the jth digit b_j of b, and then sum the suitably aligned products $p_j \cdot B^j$ to obtain the product of a and b. Graphically,

$$\begin{array}{cccccccc}
 & & & p_{0,n} & p_{0,n-1} & \cdots & p_{0,2} & p_{0,1}\ p_{0,0} \\
 & & p_{1,n} & p_{1,n-1} & p_{1,n-2} & \cdots & p_{1,1} & p_{1,0} \\
 & p_{2,n} & p_{2,n-1} & p_{2,n-2} & p_{2,n-3} & \cdots & p_{2,0} & \\
 & & & \cdots & & & & \\
\hline
p_{n-1,n} & \cdots & p_{n-1,3} & p_{n-1,2} & p_{n-1,1} & & p_{n-1,0} &
\end{array}$$

sum of the n partial products

The description in pseudocode is more compact. We initialize the product p to 0 and then add to it the partial products $a \cdot b_j \cdot B^j$ one by one:

$p = 0 : \mathbb{N}$
for $j := 0$ **to** $n-1$ **do** $p := p + a \cdot b_j \cdot B^j$

Let us analyze the number of primitive operations required by the school method. Each partial product p_j requires $2n$ primitive operations, and hence all partial products together require $2n^2$ primitive operations. The product $a \cdot b$ is a $2n$-digit number, and hence all summations $p + a \cdot b_j \cdot B^j$ are summations of $2n$-digit integers. Each such addition requires at most $2n$ primitive operations, and hence all additions together require at most $2n^2$ primitive operations. Thus, we need no more than $4n^2$ primitive operations in total.

[5] In the literature on compiler construction and performance optimization, this transformation is known as *loop fusion*.

n	T_n / sec	$T_n/T_{n/2}$
8	0.00000469	
16	0.0000154	3.28
32	0.0000567	3.67
64	0.000222	3.91
128	0.000860	3.87
256	0.00347	4.03
512	0.0138	3.98
1 024	0.0547	3.95
2 048	0.220	4.01
4 096	0.880	4.00
8 192	3.53	4.01
16 384	14.2	4.01
32 768	56.7	4.00
65 536	227	4.00
131 072	910	4.00

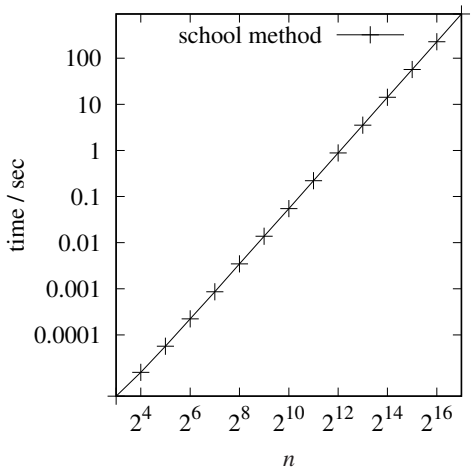

Fig. 1.2. The running time of the school method for the multiplication of n-digit integers. The three columns of the table on the *left* give n, the running time T_n in seconds of the C++ implementation given in Sect. 1.8, and the ratio $T_n/T_{n/2}$. The plot on the *right* shows $\log T_n$ versus $\log n$, and we see essentially a line. Observe that if $T_n = \alpha n^\beta$ for some constants α and β, then $T_n/T_{n/2} = 2^\beta$ and $\log T_n = \beta \log n + \log \alpha$, i.e., $\log T_n$ depends linearly on $\log n$ with slope β. In our case, the slope is 2. Please use a ruler to check.

A simple observation allows us to improve this bound. The number $a \cdot b_j \cdot B^j$ has $n+1+j$ digits, the last j of which are 0. We can therefore start the addition in the $(j+1)$th position. Also, when we add $a \cdot b_j \cdot B^j$ to p, we have $p = a \cdot (b_{j-1} \ldots b_0)$, i.e., p has $n+j$ digits. Thus, the addition of p and $a \cdot b_j \cdot B^j$ amounts to the addition of two $n+1$-digit numbers and requires only $n+1$ primitive operations. Therefore, all $n-1$ additions together require no more than $(n-1)(n+1) < n^2$ primitive operations. We have thus shown the following result.

Theorem 1.5. *The school method multiplies two n-digit integers with* $3n^2$ *primitive operations.*

We have now analyzed the numbers of primitive operations required by the school methods for integer addition and integer multiplication. The number M_n of primitive operations for the school method for integer multiplication is essentially $3n^2$. We say that M_n *grows quadratically*. Observe also that

$$\frac{M_n}{M_{n/2}} = \frac{3n^2}{3(n/2)^2} = 4,$$

i.e., quadratic growth has the consequence of essentially quadrupling the number of primitive operations when the size of the instance is doubled.

Assume now that we actually implement the multiplication algorithm in our favorite programming language (we shall do so later in the chapter), and then time the

program on our favorite machine for various n-digit integers a and b and various n. What should we expect? We want to argue that we shall see quadratic growth. The reason is that *primitive operations are representative of the running time of the algorithm*. Consider the addition of two n-digit integers first. What happens when the program is executed? For each position i, the digits a_i and b_i have to be moved to the processing unit, the sum $a_i + b_i + c$ has to be formed, the digit s_i of the result needs to be stored in memory, the carry c is updated, the index i is incremented, and a test for loop exit needs to be performed. Thus, for each i, the same number of machine cycles is executed. We have counted one primitive operation for each i, and hence the number of primitive operations is representative of the number of machine cycles executed. Of course, there are additional effects, for example pipelining and the complex transport mechanism for data between memory and the processing unit, but they will have a similar effect for all i, and hence the number of primitive operations is also representative of the running time of an actual implementation on an actual machine. The argument extends to multiplication, since multiplication of a number by a one-digit number is a process similar to addition and the second phase of the school method for multiplication amounts to a series of additions.

Let us confirm the above argument by an experiment. Figure 1.2 shows execution times of a C++ implementation of the school method; the program can be found in Sect. 1.8. For each n, we performed a large number[6] of multiplications of n-digit random integers and then determined the average running time T_n; T_n is listed in the second column of the table. We also show the ratio $T_n/T_{n/2}$. Figure 1.2 also shows a plot of the data points[7] $(\log n, \log T_n)$. The data exhibits approximately quadratic growth, as we can deduce in various ways. The ratio $T_n/T_{n/2}$ is always close to four, and the double logarithmic plot shows essentially a line of slope 2. The experiments are quite encouraging: *Our theoretical analysis has predictive value. Our theoretical analysis showed quadratic growth of the number of primitive operations, we argued above that the running time should be related to the number of primitive operations, and the actual running time essentially grows quadratically.* However, we also see systematic deviations. For small n, the growth factor from one row to the next is by less than a factor of four, as linear and constant terms in the running time still play a substantial role. For larger n, the ratio is very close to four. For very large n (too large to be timed conveniently), we would probably see a factor larger than four, since the access time to memory depends on the size of the data. We shall come back to this point in Sect. 2.2.

Exercise 1.2. Write programs for the addition and multiplication of long integers. Represent integers as sequences (arrays or lists or whatever your programming language offers) of decimal digits and use the built-in arithmetic to implement the primitive operations. Then write ADD, MULTIPLY1, and MULTIPLY functions that add

[6] The internal clock that measures CPU time returns its timings in some units, say milliseconds, and hence the rounding required introduces an error of up to one-half of this unit. It is therefore important that the experiment whose duration is to be timed takes much longer than this unit, in order to reduce the effect of rounding.

[7] Throughout this book, we use $\log x$ to denote the logarithm to base 2, $\log_2 x$.

integers, multiply an integer by a one-digit number, and multiply integers, respectively. Use your implementation to produce your own version of Fig. 1.2. Experiment with using a larger base than 10, say base 2^{16}.

Exercise 1.3. Describe and analyze the school method for division.

1.3 Result Checking

Our algorithms for addition and multiplication are quite simple, and hence it is fair to assume that we can implement them correctly in the programming language of our choice. However, writing software[8] is an error-prone activity, and hence we should always ask ourselves whether we can check the results of a computation. For multiplication, the authors were taught the following technique in elementary school. The method is known as *Neunerprobe* in German, "casting out nines" in English, and *preuve par neuf* in French.

Add the digits of a. If the sum is a number with more than one digit, sum its digits. Repeat until you arrive at a one-digit number, called the checksum of a. We use s_a to denote this checksum. Here is an example:

$$4528 \to 19 \to 10 \to 1.$$

Do the same for b and the result c of the computation. This gives the checksums s_b and s_c. All checksums are single-digit numbers. Compute $s_a \cdot s_b$ and form its checksum s. If s differs from s_c, c is not equal to $a \cdot b$. This test was described by al-Khwarizmi in his book on algebra.

Let us go through a simple example. Let $a = 429$, $b = 357$, and $c = 154153$. Then $s_a = 6$, $s_b = 6$, and $s_c = 1$. Also, $s_a \cdot s_b = 36$ and hence $s = 9$. So $s_c \neq s$ and hence c is not the product of a and b. Indeed, the correct product is $c = 153153$. Its checksum is 9, and hence the correct product passes the test. The test is not foolproof, as $c = 135153$ also passes the test. However, the test is quite useful and detects many mistakes.

What is the mathematics behind this test? We shall explain a more general method. Let q be any positive integer; in the method described above, $q = 9$. Let s_a be the remainder, or residue, in the integer division of a by q, i.e., $s_a = a - \lfloor a/q \rfloor \cdot q$. Then $0 \leq s_a < q$. In mathematical notation, $s_a = a \bmod q$.[9] Similarly, $s_b = b \bmod q$ and $s_c = c \bmod q$. Finally, $s = (s_a \cdot s_b) \bmod q$. If $c = a \cdot b$, then it must be the case that $s = s_c$. Thus $s \neq s_c$ proves $c \neq a \cdot b$ and uncovers a mistake in the multiplication (or a mistake in carrying out casting out nines). What do we know if $s = s_c$? We know that q divides the difference of c and $a \cdot b$. If this difference is nonzero, the mistake will be detected by any q which does not divide the difference.

[8] The bug in the division algorithm of the floating-point unit of the original Pentium chip became infamous. It was caused by a few missing entries in a lookup table used by the algorithm.

[9] The method taught in school uses residues in the range 1 to 9 instead of 0 to 8 according to the definition $s_a = a - (\lceil a/q \rceil - 1) \cdot q$.

Let us continue with our example and take $q = 7$. Then $a \bmod 7 = 2$, $b \bmod 7 = 0$ and hence $s = (2 \cdot 0) \bmod 7 = 0$. But $135153 \bmod 7 = 4$, and we have uncovered the fact that $135153 \neq 429 \cdot 357$.

Exercise 1.4. Explain why casting out nines corresponds to the case $q = 9$. Hint: $10^k \bmod 9 = 1$ for all $k \geq 0$.

Exercise 1.5 (*Elferprobe*, casting out elevens). Powers of ten have very simple remainders modulo 11, namely $10^k \bmod 11 = (-1)^k$ for all $k \geq 0$, i.e., $1 \bmod 11 = +1$, $10 \bmod 11 = -1$, $100 \bmod 11 = +1$, $1\,000 \bmod 11 = -1$, etc. Describe a simple test to check the correctness of a multiplication modulo 11.

1.4 A Recursive Version of the School Method

We shall now derive a recursive version of the school method. This will be our first encounter with the *divide-and-conquer* paradigm, one of the fundamental paradigms in algorithm design.

Let a and b be our two n-digit integers which we want to multiply. Let $k = \lfloor n/2 \rfloor$. We split a into two numbers a_1 and a_0; a_0 consists of the k least significant digits and a_1 consists of the $n-k$ most significant digits.[10] We split b analogously. Then

$$a = a_1 \cdot B^k + a_0 \quad \text{and} \quad b = b_1 \cdot B^k + b_0,$$

and hence

$$a \cdot b = a_1 \cdot b_1 \cdot B^{2k} + (a_1 \cdot b_0 + a_0 \cdot b_1) \cdot B^k + a_0 \cdot b_0.$$

This formula suggests the following algorithm for computing $a \cdot b$:

(a) Split a and b into a_1, a_0, b_1, and b_0.
(b) Compute the four products $a_1 \cdot b_1$, $a_1 \cdot b_0$, $a_0 \cdot b_1$, and $a_0 \cdot b_0$.
(c) Add the suitably aligned products to obtain $a \cdot b$.

Observe that the numbers a_1, a_0, b_1, and b_0 are $\lceil n/2 \rceil$-digit numbers and hence the multiplications in step (b) are simpler than the original multiplication if $\lceil n/2 \rceil < n$, i.e., $n > 1$. The complete algorithm is now as follows. To multiply one-digit numbers, use the multiplication primitive. To multiply n-digit numbers for $n \geq 2$, use the three-step approach above.

It is clear why this approach is called *divide-and-conquer*. We reduce the problem of multiplying a and b to some number of *simpler* problems of the same kind. A divide-and-conquer algorithm always consists of three parts: In the first part, we split the original problem into simpler problems of the same kind (our step (a)); in the second part, we solve the simpler problems using the same method (our step (b)); and, in the third part, we obtain the solution to the original problem from the solutions to the subproblems (our step (c)). Instead of "we solve the simpler problems using

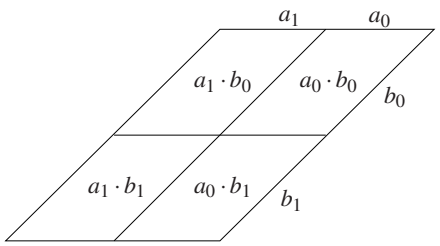

Fig. 1.3. Visualization of the school method and its recursive variant. The rhombus-shaped area indicates the partial products in the multiplication $a \cdot b$. The four subareas correspond to the partial products $a_1 \cdot b_1$, $a_1 \cdot b_0$, $a_0 \cdot b_1$, and $a_0 \cdot b_0$. In the recursive scheme, we first sum the partial products in the four subareas and then, in a second step, add the four resulting sums.

the same method", one usually says more elegantly "we solve the simpler problems recursively".

What is the connection of our recursive integer multiplication to the school method? The two methods are strongly related. Both methods compute all n^2 digit products $a_i b_j$ and then sum the resulting two-digit results (appropriately shifted). They differ in the order in which these summations are performed. Figure 1.3 visualizes the digit products and their place values as rhombus-shaped regions. The school method adds the digit products row by row. The recursive method first computes the partial results corresponding to the four subregions, which it then combines to obtain the final result with three additions. We may almost say that our recursive integer multiplication is just the school method in disguise. In particular, the recursive algorithm also uses a quadratic number of primitive operations.

We next derive the quadratic behavior without appealing to the connection to the school method. This will allow us to introduce recurrence relations, a powerful concept for the analysis of recursive algorithms.

Lemma 1.6. *Let $T(n)$ be the maximum number of primitive operations required by our recursive multiplication algorithm when applied to n-digit integers. Then*

$$T(n) \leq \begin{cases} 1 & \text{if } n = 1, \\ 4 \cdot T(\lceil n/2 \rceil) + 2 \cdot 2 \cdot n & \text{if } n \geq 2. \end{cases}$$

Proof. Multiplying two one-digit numbers requires one primitive multiplication. This justifies the case $n = 1$. So, assume $n \geq 2$. Splitting a and b into the four pieces a_1, a_0, b_1, and b_0 requires no primitive operations.[11] Each piece has at most $\lceil n/2 \rceil$ digits and hence the four recursive multiplications require at most $4 \cdot T(\lceil n/2 \rceil)$ primitive operations. The products $a_0 \cdot b_0$ and $a_1 \cdot b_1 \cdot B^{2k}$ can be combined into a single number without any cost as the former number is a $2k$-digit number and the latter number ends with $2k$ many 0's. Finally, we need two additions to assemble the final result. Each addition involves two numbers of at most $2n$ digits and hence requires at most $2n$ primitive operations. This justifies the inequality for $n \geq 2$. □

[10] Observe that we have changed notation; a_0 and a_1 now denote the two parts of a and are no longer single digits.

[11] It will require work, but it is work that we do not account for in our analysis.

In Sect. 2.8, we shall learn that such recurrences are easy to solve and yield the already conjectured quadratic execution time of the recursive algorithm.

Lemma 1.7. *Let $T(n)$ be the maximum number of primitive operations required by our recursive multiplication algorithm when applied to n-digit integers. Then $T(n) \leq 5n^2$ if n is a power of 2, and $T(n) \leq 20n^2$ for all n.*

Proof. We refer the reader to Sect. 1.9 for a proof. □

1.5 Karatsuba Multiplication

In 1962, the Soviet mathematician Karatsuba [174] discovered a faster way of multiplying large integers. The running time of his algorithm grows like $n^{\log 3} \approx n^{1.58}$. The method is surprisingly simple. Karatsuba observed that a simple algebraic identity allows one multiplication to be eliminated in the divide-and-conquer implementation, i.e., one can multiply n-digit numbers using only *three* multiplications of integers half the size.

The details are as follows. Let a and b be our two n-digit integers which we want to multiply. Let $k = \lfloor n/2 \rfloor$. As above, we split a into two numbers a_1 and a_0; a_0 consists of the k least significant digits and a_1 consists of the $n-k$ most significant digits. We split b in the same way. Then

$$a = a_1 \cdot B^k + a_0 \quad \text{and} \quad b = b_1 \cdot B^k + b_0$$

and hence (the magic is in the second equality)

$$\begin{aligned} a \cdot b &= a_1 \cdot b_1 \cdot B^{2k} + (a_1 \cdot b_0 + a_0 \cdot b_1) \cdot B^k + a_0 \cdot b_0 \\ &= a_1 \cdot b_1 \cdot B^{2k} + ((a_1 + a_0) \cdot (b_1 + b_0) - (a_1 \cdot b_1 + a_0 \cdot b_0)) \cdot B^k + a_0 \cdot b_0. \end{aligned}$$

At first sight, we have only made things more complicated. A second look, however, shows that the last formula can be evaluated with only three multiplications, namely, $a_1 \cdot b_1$, $a_0 \cdot b_0$, and $(a_1 + a_0) \cdot (b_1 + b_0)$. We also need six additions.[12] That is three more than in the recursive implementation of the school method. The key is that additions are cheap compared with multiplications, and hence saving a multiplication more than outweighs the additional additions. We obtain the following algorithm for computing $a \cdot b$:

(a) Split a and b into a_1, a_0, b_1, and b_0.
(b) Compute the three products

$$p_2 = a_1 \cdot b_1, \quad p_0 = a_0 \cdot b_0, \quad p_1 = (a_1 + a_0) \cdot (b_1 + b_0).$$

[12] Actually, five additions and one subtraction. We leave it to readers to convince themselves that subtractions are no harder than additions.

(c) Add the suitably aligned products to obtain $a \cdot b$, i.e., compute $a \cdot b$ according to the formula

$$a \cdot b = (p_2 \cdot B^{2k} + p_0) + (p_1 - (p_2 + p_0)) \cdot B^k.$$

The first addition can be performed by concatenating the corresponding digit strings and requires no primitive operation.

The numbers a_1, a_0, b_1, b_0, $a_1 + a_0$, and $b_1 + b_0$ are $(\lceil n/2 \rceil + 1)$-digit numbers and hence the multiplications in step (b) are simpler than the original multiplication if $\lceil n/2 \rceil + 1 < n$, i.e., $n \geq 4$. The complete algorithm is now as follows: To multiply three-digit numbers, use the school method, and to multiply n-digit numbers for $n \geq 4$, use the three-step approach above.

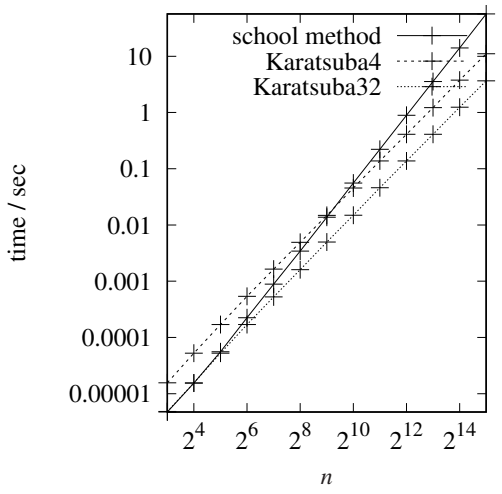

Fig. 1.4. The running times of implementations of the Karatsuba and school methods for integer multiplication. The running times of two versions of Karatsuba's method are shown: Karatsuba4 switches to the school method for integers with fewer than four digits, and Karatsuba32 switches to the school method for integers with fewer than 32 digits. The slopes of the lines for the Karatsuba variants are approximately 1.58. The running time of Karatsuba32 is approximately one-third the running time of Karatsuba4.

Figure 1.4 shows the running times $T_S(n)$, $T_{K4}(n)$, and $T_{K32}(n)$ of C++ implementations of the school method and of two variants of the Karatsuba method for the multiplication of n-digit numbers. Karatsuba4 (running time $T_{K4}(n)$) uses the school method for numbers with fewer than four digits and Karatsuba32 (running time $T_{K32}(n)$) uses the school method for numbers with fewer than 32 digits; we discuss the rationale for this variant in Sect. 1.7. The scales on both axes are logarithmic. We see, essentially, straight lines of different slope. The running time of the school method grows like n^2, and hence the slope is 2 in the case of the school method. The slope is smaller in the case of the Karatsuba method, and this suggests that its running time grows like n^β with $\beta < 2$. In fact, the ratios[13] $T_{K4}(n)/T_{K4}(n/2)$ and $T_{K32}(n)/T_{K32}(n/2)$ are close to three, and this suggests that β is such that $2^\beta = 3$ or $\beta = \log 3 \approx 1.58$. Alternatively, you may determine the slope from Fig. 1.4. We shall prove below that $T_K(n)$ grows like $n^{\log 3}$. We say that *the Karatsuba method has*

[13] $T_{K4}(1024) = 0.0455$, $T_{K4}(2048) = 0.1375$, and $T_{K4}(4096) = 0.41$.

better asymptotic behavior than the school method. We also see that the inputs have to be quite big before the superior asymptotic behavior of the Karatsuba method actually results in a smaller running time. Observe that for $n = 2^8$, the school method is still faster, that for $n = 2^9$, the two methods have about the same running time, and that the Karatsuba method wins for $n = 2^{10}$. The lessons to remember are:

- Better asymptotic behavior ultimately wins.
- An asymptotically slower algorithm can be faster on small inputs.

In the next section, we shall learn how to improve the behavior of the Karatsuba method for small inputs. The resulting algorithm will always be at least as good as the school method. It is time to derive the asymptotics of the Karatsuba method.

Lemma 1.8. *Let $T_K(n)$ be the maximum number of primitive operations required by the Karatsuba algorithm when applied to n-digit integers. Then*

$$T_K(n) \leq \begin{cases} 3n^2 & \text{if } n \leq 3, \\ 3 \cdot T_K(\lceil n/2 \rceil + 1) + 8n & \text{if } n \geq 4. \end{cases}$$

Proof. Multiplying two n-digit numbers using the school method requires no more than $3n^2$ primitive operations, according to Theorem 1.5. This justifies the first line. So, assume $n \geq 4$. Splitting a and b into the four pieces a_1, a_0, b_1, and b_0 requires no primitive operations.[14] Each piece and the sums $a_0 + a_1$ and $b_0 + b_1$ have at most $\lceil n/2 \rceil + 1$ digits, and hence the three recursive multiplications require at most $3 \cdot T_K(\lceil n/2 \rceil + 1)$ primitive operations. We need two additions to form $a_0 + a_1$ and $b_0 + b_1$. The results of these additions have fewer than n digits and hence the additions need no more than n elementary operations each. Finally, we need three additions in order to compute the final result from the results of the multiplications. These are additions of numbers with at most $2n$ digits. Thus these additions require at most $3 \cdot 2n$ primitive operations. Altogether, we obtain the bound stated in the second line of the recurrence. □

In Sect. 2.8, we shall learn some general techniques for solving recurrences of this kind.

Theorem 1.9. *Let $T_K(n)$ be the maximum number of primitive operations required by the Karatsuba algorithm when applied to n-digit integers. Then $T_K(n) \leq 153n^{\log 3}$ for all n.*

Proof. We refer the reader to Sect. 1.9 for a proof. □

1.6 Parallel Multiplication

Both the recursive version of the school method and the Karatsuba algorithm are good starting points for parallel algorithms. For simplicity, we focus on the school

[14] It will require work, but remember that we are counting primitive operations.

method. Recall that the bulk of the work is done in the recursive multiplications a_0b_0, a_1b_1, a_0b_1, and a_1b_0. These four multiplications can be done independently and in parallel. Hence, in the first level of recursion, up to four processors can work in parallel. In the second level of recursion, the parallelism is already $4 \cdot 4 = 16$. In the ith level of recursion, 4^i processors can work in parallel. Fig. 1.5 shows a graphical representation of the resulting computation for multiplying two numbers $a_{11}a_{10}a_{01}a_{00}$ and $b_{11}b_{10}b_{01}b_{00}$.

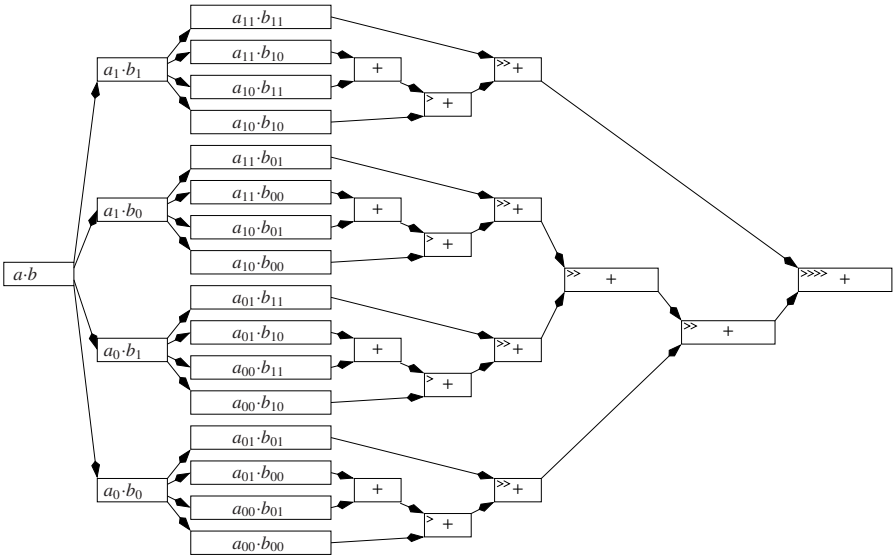

Fig. 1.5. Task graph for parallel recursive multiplication with two levels of parallel recursion and using the school method on two-digit numbers as the base case. A "$>$" stands for a shift by two digits.

What is the running time of the algorithm if an unlimited number of processors are available? This quantity is known as the *span* of a parallel computation; see Sect. 2.10. For simplicity, we assume a sequential addition algorithm. As before, we shall count only arithmetic operations on digits. For the span $S(n)$ for multiplying two n-digit integers, we get

$$S(n) \leq \begin{cases} 1 & \text{if } n = 1, \\ S(\lceil n/2 \rceil) + 3 \cdot 2 \cdot n & \text{if } n \geq 2. \end{cases}$$

This recurrence has the solution $S(n) \leq 12n$. Note that this is much less than the quadratic number of operations performed. Hence, we can hope for considerable speedup by parallel processing – even without parallel addition.

Exercise 1.6. Show that by using parallel addition (see Sect. 1.1.1), we can achieve a span $O(\log^2 n)$ for parallel school multiplication.

Parallel programming environments for multicore processors make it relatively easy to exploit parallelism in the kind of divide-and-conquer computations described above, see Sect. 2.14 for details. Roughly, they create a *task* for recursive calls and automatically assign cores to tasks. They ensure that enough tasks are created to keep all available cores busy, but also that tasks stay reasonably large. Section 14.5 explains the *load-balancing* algorithms behind this programming model.

Exercise 1.7. Describe a parallel divide-and-conquer algorithm for the Karatsuba method. Show that its span is linear in the number of input digits.

1.7 Algorithm Engineering

Karatsuba integer multiplication is superior to the school method for large inputs. In our implementation, the superiority only shows for integers with more than 1000 digits. However, a simple refinement improves the performance significantly. Since the school method is superior to the Karatsuba method for short integers, we should stop the recursion earlier and switch to the school method for numbers which have fewer than n_0 digits for some yet to be determined n_0. We call this approach the *refined Karatsuba method*. It is never worse than either the school method or the original Karatsuba algorithm as long as n_0 is not chosen too large.

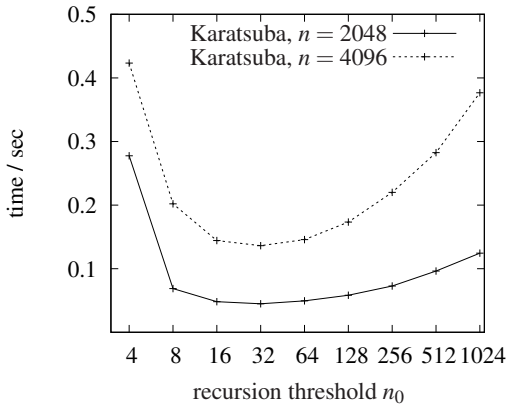

Fig. 1.6. The running time of the Karatsuba method as a function of the recursion threshold n_0. The times consumed for multiplying 2048-digit and 4096-digit integers are shown. The minimum is at $n_0 = 32$.

What is a good choice for n_0? We shall answer this question both experimentally and analytically. Let us discuss the experimental approach first. We simply time the refined Karatsuba algorithm for different values of n_0 and then adopt the value giving the smallest running time. For our implementation, the best results were obtained for $n_0 = 32$ (see Fig. 1.6). The asymptotic behavior of the refined Karatsuba method is shown in Fig. 1.4. We see that the running time of the refined method still grows like $n^{\log 3}$, that the refined method is about three times faster than the basic Karatsuba

method and hence the refinement is highly effective, and that the refined method is never slower than the school method.

Exercise 1.8. Derive a recurrence for the worst-case number $T_R(n)$ of primitive operations performed by the refined Karatsuba method.

We can also approach the question analytically. If we use the school method to multiply n-digit numbers, we need $3n^2$ primitive operations. If we use one Karatsuba step and then multiply the resulting numbers of length $\lceil n/2 \rceil + 1$ using the school method, we need about $3(3(n/2+1)^2) + 7n$ primitive operations. The latter is smaller for $n \geq 23$, and hence a recursive step saves primitive operations as long as the number of digits is more than 23. You should not take this as an indication that an actual implementation should switch at integers of approximately 23 digits, as the argument concentrates solely on primitive operations. You should take it as an argument that it is wise to have a nontrivial recursion threshold n_0 and then determine the threshold experimentally.

Exercise 1.9. Throughout this chapter, we have assumed that both arguments of a multiplication are n-digit integers. What can you say about the complexity of multiplying n-digit and m-digit integers? (a) Show that the school method requires no more than $\alpha \cdot nm$ primitive operations for some constant α. (b) Assume $n \geq m$ and divide a into $\lceil n/m \rceil$ numbers of m digits each. Multiply each of the fragments by b using Karatsuba's method and combine the results. What is the running time of this approach?

1.8 The Programs

We give C++ programs for the school and Karatsuba methods below. These programs were used for the timing experiments described in this chapter. The programs were executed on a machine with a 2 GHz dual-core Intel Core 2 Duo T7200 processor with 4 MB of cache memory and 2 GB of main memory. The programs were compiled with GNU C++ version 3.3.5 using optimization level `-O2`.

A digit is simply an unsigned int and an integer is a vector of digits; here, "vector" is the vector type of the standard template library. A declaration *integer* $a(n)$ declares an integer with n digits, $a.size()$ returns the size of a, and $a[i]$ returns a reference to the ith digit of a. Digits are numbered starting at 0. The global variable B stores the base. The functions *fullAdder* and *digitMult* implement the primitive operations on digits. We sometimes need to access digits beyond the size of an integer; the function $getDigit(a,i)$ returns $a[i]$ if i is a legal index for a and returns 0 otherwise:

```
typedef unsigned int digit;
typedef vector<digit> integer;
unsigned int B = 10;                          // Base, 2 <= B <= 2^16

void fullAdder( digit a, digit b, digit c, digit & s, digit & carry)
{ unsigned int sum = a + b + c; carry = sum/B; s = sum - carry*B; }
```

```
void digitMult ( digit  a,  digit  b,  digit & s,  digit & carry)
{ unsigned int prod = a*b; carry = prod/B; s = prod - carry*B; }

digit  getDigit (const integer& a, int  i )
{ return ( i < a.size()? a[ i ] : 0 ); }
```

We want to run our programs on random integers: *randDigit* is a simple random generator for digits, and *randInteger* fills its argument with random digits.

```
unsigned int X = 542351;

digit  randDigit () { X = 443143*X + 6412431; return X % B ; }

void randInteger(integer& a)
{ int n = a.size (); for (int i=0; i<n; i++) a[ i ] = randDigit ();}
```

We now come to the school method of multiplication. We start with a routine that multiplies an integer a by a digit b and returns the result in *atimesb*. We need a carry-digit *carry*, which we initialize to 0. In each iteration, we compute d and c such that $c * B + d = a[i] * b$. We then add d, the c from the previous iteration, and the *carry* from the previous iteration, store the result in $atimesb[i]$, and remember the *carry*. The school method (the function *mult*) multiplies a by each digit of b and then adds it at the appropriate position to the partial result (the function *addAt*):

```
void mult(const integer& a, const digit& b,  integer& atimesb)
{ int n = a.size ();  assert(atimesb.size()  == n+1);
   digit  carry = 0,  c,  d,  cprev = 0;
  for (int i = 0; i < n; i++)
  {  digitMult (a[ i ], b,d,c);
     fullAdder(d,  cprev,  carry,  atimesb[ i ],  carry );
     cprev = c;
  }
  d = 0;
  fullAdder(d,  cprev,  carry,  atimesb[n],  carry );   assert(carry == 0);
}

void addAt(integer& p, const integer& atimesbj, int  j )
{ //   p has length n+m,
   digit  carry = 0;  int L = p.size ();
  for (int i = j ; i < L; i++)
      fullAdder(p[ i ],  getDigit (atimesbj, i - j ),  carry,  p[ i ],  carry );
  assert(carry == 0);
}

integer  mult(const integer& a, const integer& b)
{ int n = a.size ();  int m = b.size ();
  integer  p(n + m,0);   integer  atimesbj(n+1);
  for (int j = 0; j < m; j++)
  { mult(a,  b[ j ],  atimesbj); addAt(p, atimesbj,  j ); }
  return p;
}
```

For Karatsuba's method, we also need algorithms for general addition and subtraction. The subtraction method may assume that the first argument is no smaller than the second. It computes its result in the first argument:

```
integer add(const integer& a, const integer& b)
{ int n = max(a.size(),b.size());
  integer s(n+1); digit carry = 0;
  for (int i = 0; i < n; i++)
    fullAdder(getDigit(a,i), getDigit(b,i), carry, s[i], carry);
  s[n] = carry;
  return s;
}
void sub(integer& a, const integer& b) // requires a >= b
{ digit carry = 0;
  for (int i = 0; i < a.size(); i++)
    if ( a[i] >= ( getDigit(b,i) + carry ))
      { a[i] = a[i] - getDigit(b,i) - carry; carry = 0; }
    else { a[i] = a[i] + B - getDigit(b,i) - carry; carry = 1;}
  assert(carry == 0);
}
```

The function *split* splits an integer into two integers of half the size:

```
void split (const integer& a,integer& a1, integer& a0)
{ int n = a.size(); int k = n/2;
  for (int i = 0; i < k; i++) a0[i] = a[i];
  for (int i = 0; i < n - k; i++) a1[i] = a[k+ i];
}
```

The function *Karatsuba* works exactly as described in the text. If the inputs have fewer than *n0* digits, the school method is employed. Otherwise, the inputs are split into numbers of half the size and the products *p0*, *p1*, and *p2* are formed. Then *p0* and *p2* are written into the output vector and subtracted from *p1*. Finally, the modified *p1* is added to the result:

```
integer Karatsuba(const integer& a, const integer& b, int n0)
{ int n = a.size(); int m = b.size(); assert(n == m); assert(n0 >= 4);
  integer p(2*n);
  if (n < n0) return mult(a,b);
  int k = n/2; integer a0(k), a1(n - k), b0(k), b1(n - k);
  split (a,a1,a0); split (b,b1,b0);
  integer p2 = Karatsuba(a1,b1,n0),
          p1 = Karatsuba(add(a1,a0),add(b1,b0),n0),
          p0 = Karatsuba(a0,b0,n0);
  for (int i = 0; i < 2*k; i++) p[i] = p0[i];
  for (int i = 2*k; i < n+m; i++) p[i] = p2[i - 2*k];
  sub(p1,p0); sub(p1,p2); addAt(p,p1,k);
  return p;
}
```

The following program generated the data for Fig. 1.4:

```
inline double cpuTime() { return double(clock())/CLOCKS_PER_SEC; }
int main(){
for (int n = 8; n <= 131072; n *= 2)
{ integer a(n), b(n); randInteger(a); randInteger(b);
  double T = cpuTime(); int k = 0;
  while (cpuTime() - T < 1) { mult(a,b); k++; }
  cout << "\n" << n << " school = " << (cpuTime() -T)/k;
  T = cpuTime(); k = 0;
  while (cpuTime() - T < 1) { Karatsuba(a,b,4); k++; }
  cout << " Karatsuba4 = " << (cpuTime() -T) /k; cout.flush ();
  T = cpuTime(); k = 0;
  while (cpuTime() - T < 1) { Karatsuba(a,b,32); k++; }
  cout << " Karatsuba32 = " << (cpuTime() -T) /k; cout.flush ();
}
return 0;
}
```

1.9 Proofs of Lemma 1.7 and Theorem 1.9

To make this chapter self-contained, we include proofs of Lemma 1.7 and Theorem 1.9. We start with an analysis of the recursive version of the school method. Recall that $T(n)$, the maximum number of primitive operations required by our recursive multiplication algorithm when applied to n-digit integers, satisfies the recurrence relation

$$T(n) \leq \begin{cases} 1 & \text{if } n = 1, \\ 4 \cdot T(\lceil n/2 \rceil) + 4n & \text{if } n \geq 2. \end{cases}$$

We use induction on n to show $T(n) \leq 5n^2 - 4n$ when n is a power of 2. For $n = 1$, we have $T(1) \leq 1 = 5n^2 - 4n$. For $n > 1$, we have

$$T(n) \leq 4T(n/2) + 4n \leq 4(5(n/2)^2 - 4n/2) + 4n = 5n^2 - 4n,$$

where the second inequality follows from the induction hypothesis. For general n, we observe that multiplying n-digit integers is certainly no more costly than multiplying $2^{\lceil \log n \rceil}$-digit integers and hence $T(n) \leq T(2^{\lceil \log n \rceil})$. Since $2^{\lceil \log n \rceil} \leq 2n$, we conclude $T(n) \leq 20n^2$ for all n.

Exercise 1.10. Prove a bound on the recurrence $T(1) \leq 1$ and $T(n) \leq 4T(n/2) + 9n$ when n is a power of 2.

How did we know that "$5n^2 - 4n$" is the bound to be shown? There is no magic here. For $n = 2^k$, repeated substitution yields

$$\begin{aligned}
T(2^k) &\le 4 \cdot T(2^{k-1}) + 4 \cdot 2^k \le 4 \cdot (4 \cdot T^{k-2} + 4 \cdot 2^{k-1}) + 4 \cdot 2^k \\
&\le 4 \cdot (4 \cdot (4 \cdot T(2^{k-3}) + 4 \cdot 2^{k-2}) + 4 \cdot 2^{k-1}) + 4 \cdot 2^k \\
&\le 4^3 T(2^{k-3}) + 4 \cdot (4^2 \cdot 2^{k-2} + 4^1 \cdot 2^{k-1} + 2^k) \le \ldots \\
&\le 4^k T(1) + 4 \sum_{0 \le i \le k-1} 4^i 2^{k-i} \le 4^k + 4 \cdot 2^k \sum_{0 \le i \le k-1} 2^i \\
&\le 4^k + 4 \cdot 2^k (2^k - 1) = n^2 + 4n(n-1) = 5n^2 - 4n.
\end{aligned}$$

We now turn to the proof of Theorem 1.9. Recall that T_K satisfies the recurrence

$$T_K(n) \le \begin{cases} 3n^2 & \text{if } n \le 3, \\ 3 \cdot T_K(\lceil n/2 \rceil + 1) + 8n & \text{if } n \ge 4. \end{cases}$$

The recurrence for the school method has the nice property that if n is a power of 2, the arguments of T on the right-hand side are again powers of two. This is not true for T_K. However, if $n = 2^k + 2$ and $k \ge 1$, then $\lceil n/2 \rceil + 1 = 2^{k-1} + 2$, and hence we should now use numbers of the form $n = 2^k + 2$, $k \ge 0$, as the basis of the inductive argument. We shall show that

$$T_K(2^k + 2) \le 51 \cdot 3^k - 16 \cdot 2^k - 8$$

for $k \ge 0$. For $k = 0$, we have

$$T_K(2^0 + 2) = T_K(3) \le 3 \cdot 3^2 = 27 = 51 \cdot 3^0 - 16 \cdot 2^0 - 8.$$

For $k \ge 1$, we have

$$\begin{aligned}
T_K(2^k + 2) &\le 3T_K(2^{k-1} + 2) + 8 \cdot (2^k + 2) \\
&\le 3 \cdot \left(51 \cdot 3^{k-1} - 16 \cdot 2^{k-1} - 8\right) + 8 \cdot (2^k + 2) \\
&= 51 \cdot 3^k - 16 \cdot 2^k - 8.
\end{aligned}$$

Again, there is no magic in coming up with the right induction hypothesis. It is obtained by repeated substitution. Namely,

$$\begin{aligned}
T_K(2^k + 2) &\le 3T_K(2^{k-1} + 2) + 8 \cdot (2^k + 2) \\
&\le 3^k T_K(2^0 + 2) + 8 \cdot \left(3^0(2^k + 2) + 3^1(2^{k-1} + 2) + \ldots + 3^{k-1}(2^1 + 2)\right) \\
&\le 27 \cdot 3^k + 8 \cdot \left(2^k \frac{(3/2)^k - 1}{3/2 - 1} + 2\frac{3^k - 1}{3 - 1}\right) \\
&\le 51 \cdot 3^k - 16 \cdot 2^k - 8,
\end{aligned}$$

where the first inequality uses the fact that $2^k + 2$ is even, the second inequality follows from repeated substitution, the third inequality uses $T_K(3) = 27$, and the last inequality follows by a simple computation.

It remains to extend the bound to all n. Let k be the minimum integer such that $n \le 2^k + 2$. Then $k \le 1 + \log n$. Also, multiplying n-digit numbers is no more costly than multiplying $(2^k + 2)$-digit numbers, and hence

$$T_K(n) \le 51 \cdot 3^k - 16 \cdot 2^k - 8 \le 153 \cdot 3^{\log n} \le 153 \cdot n^{\log 3},$$

where we have used the equality $3^{\log n} = 2^{(\log 3)\cdot(\log n)} = n^{\log 3}$.

Exercise 1.11. Solve the recurrence

$$T_R(n) \le \begin{cases} 3n^2 + 2n & \text{if } n < n_0, \\ 3 \cdot T_R(\lceil n/2 \rceil + 1) + 8n & \text{if } n \ge n_0, \end{cases}$$

where n_0 is a positive integer. Optimize n_0.

1.10 Implementation Notes

The programs given in Sect. 1.8 are not optimized. The base of the number system should be a power of 2 so that sums and carries can be extracted by bit operations. Also, the size of a digit should agree with the word size of the machine and a little more work should be invested in implementing the primitive operations on digits.

1.10.1 C++

GMP [127] and LEDA [194] offer exact arithmetic on integers and rational numbers of arbitrary size, and arbitrary-precision floating-point arithmetic. Highly optimized implementations of Karatsuba's method are used for multiplication here.

1.10.2 Java

java.math implements exact arithmetic on integers of arbitrary size and arbitrary-precision floating-point numbers.

1.11 Historical Notes and Further Findings

Is the Karatsuba method the fastest known method for integer multiplication? No, much faster methods are known. Karatsuba's method splits an integer into two parts and requires three multiplications of integers of half the length. The natural extension is to split integers into k parts of length n/k each. If the recursive step requires ℓ multiplications of numbers of length n/k, the running time of the resulting algorithm grows like $n^{\log_k \ell}$. In this way, Toom [315] and Cook [78] reduced the running time to[15] $O\bigl(n^{1+\varepsilon}\bigr)$ for arbitrary positive ε. Asymptotically even more efficient are the

[15] The $O(\cdot)$ notation is defined in Sect. 2.1.

algorithms of Schönhage and Strassen [285] and Schönhage [284]. The former multiplies n-bit integers with $O(n \log n \log\log n)$ bit operations, and can be implemented to run in this time bound on a Turing machine. The latter runs in linear time $O(n)$ and requires the machine model discussed in Sect. 2.2. In this model, integers with $\log n$ bits can be multiplied in constant time. The former algorithm was improved by Fürer [117] and De et al. [84] to $O((n \log n)2^{c \log^*(n)})$ bit operations. Here, $\log^* n$ is the smallest integer $k \geq 0$ such that $\log(\log(\ldots \log(n) \ldots)) \leq 1$ (k-fold application of the logarithm function). The function $\log^* n$ grows extremely slowly.

Modern microprocessors use a base $B = 2$ multiplication algorithm that does quadratic work but requires only $O(\log n)$ wire delays [148]. The basic idea is to first compute n^2 digit products in parallel. Bits with the same position in the output are combined together using full adders, which reduce three values at position i to one value at position i and one value at position $i+1$. This way, $\log_{3/2} n$ layers of full adders suffice to reduce up to n initial bits at position i to two bits. The remaining bits can be fed into an adder with logarithmic delay as in Sect. 1.1.1.