Chapter 2

Algorithms for Integer Arithmetic

We will develop efficient algorithms for the main operations on *nonnegative integers*, starting from the simple primitives of addition, subtraction, and multiplication by a power of two and finally dealing with the more complex case of integer multiplication, for which we will develop a divide-and-conquer algorithm which is faster than the elementary algorithm based on sum of partial products.

We consider a nonnegative integer A as being represented by the string of bits forming its binary representation:

$$A = a_{n-1}a_{n-2}\ldots a_0,$$

with $a_i \in \{0,1\}$, for $0 \leq i < n$. For convenience, in the algorithms we will often assume that bits at positions $\geq n$ are also defined, and conventionally set them to zero (i.e., $a_i = 0$ for $i \geq n$). For what concerns the implementation of basic integer operations, a natural measure of the instance size is the number of bits needed to represent a single input, or, in case of a fixed number of inputs, the number of bits needed to represent the largest one. Finally, when computing the running time of an algorithm, we will assign unit cost to each occurrence of a logical operator between bits: $\wedge$ (**and**), $\vee$ (**or**), $\neg$ (**not**), and $\oplus$ (**xor**). We will disregard the cost of all other instructions. Observe that in a more realistic cost model with might want to assign unit cost to other basic instructions such as, for instance, bit assignments.

2.1 Integer Addition

Le $A = a_{n-1}a_{n-2}\ldots a_0$ and $B = b_{m-1}b_{m-2}\ldots b_0$ be the two integers to be summed, and assume that $n \geq m$. Moreover, let $S = A + B$, with $S = s_n s_{n-1}\ldots s_0$ (observe that the

result of a sum may be one bit longer that the largest of the two operands).

An optimal, linear time summation algorithm can be obtained by simply implementing the well-known elementary algorithm based on carry propagation. Recall that such algorithm obtains each bit s_i of S, for $0 \leq i < n$, as a boolean function of three bits: a_i, b_i and c_i, where the latter bit represents the *carry* generated in position i, with $c_0 = 0$ and c_{i+1} itself a function of a_i, b_i and c_i. A simple logical synthesis by means of Karnaugh maps suffices to determine the following analytic description for these two functions:

$$\begin{cases} c_i &= 0 & \text{for } i = 0, \\ s_i &= a_i \oplus b_i \oplus c_i & \text{for } 0 \leq i < n, \\ c_{i+1} &= (a_i \wedge b_i) \vee (a_i \wedge c_i) \vee (b_i \wedge c_i) & \text{for } 0 \leq i < n, \\ s_i &= c_i & \text{for } i = n. \end{cases}$$

The algorithm follows immediately from the above discussion:

```
SUM(A, B)
n ← MAX(length(A), length(B))
c ← 0
for i ← 0 to n − 1 do
    s_i ← a_i xor b_i xor c
    c ← (a_i and b_i) or (a_i and c) or (b_i and c)
s_n ← c
return S
```

Observe that in the algorithm we have used a single bit-variable c, which iteratively stores the carries are generated in sequence. The running time of the resulting algorithm is easily established by observing that there are n iterations, and that in each iteration we execute 7 logical bit operations for a total running time

$$T_{\text{SUM}}(n) = 7n \in \Theta(n)$$

which is clearly optimal.

2.2 Integer Subtraction

Le $A = a_{n-1}a_{n-2}\ldots a_0$ and $B = b_{m-1}b_{m-2}\ldots b_0$ be, respectively, the minuend and the subtrahend of the subtraction to be performed, that is, $S = A - B$. Observe that we are working with nonnegative integers, therefore we assume that $A \geq B$, which in turn implies

that $n \geq m$. Observe that, unlike the case of addition, S cannot contain more than n significant bits.

An optimal, linear time subtraction algorithm can be obtained by simply implementing the well-known elementary algorithm based on borrowing. Recall that such algorithm obtains each bit s_i of S, for $0 \leq i < n$, as a boolean function of three bits: a_i, b_i and p_i, where the latter bit represents the *borrow* generated in position i, with $p_0 = 0$ and p_{i+1} itself a function of a_i, b_i and p_i. A simple logical synthesis by means of Karnaugh maps suffices to determine the following analytic description for these two functions:

$$\begin{cases} p_i &= 0 & \text{for } i = 0, \\ s_i &= a_i \oplus b_i \oplus p_i & \text{for } 0 \leq i < n, \\ p_{i+1} &= (\neg a_i \wedge b_i) \vee (\neg a_i \wedge p_i) \vee (b_i \wedge p_i) & \text{for } 0 \leq i < n. \end{cases}$$

(Observe the symmetry with addition.) The algorithm follows immediately from the above discussion:

```
SUB(A, B)
n ← length(A)
p ← 0
for i ← 0 to n − 1 do
    s_i ← a_i xor b_i xor p
    p ← ((not a_i) and b_i) or ((not a_i) and p) or (b_i and p)
return S
```

As for SUM, the running time of SUB is immediately established. We have:

$$T_{\text{SUB}}(n) = 9n \in \Theta(n)$$

which is, again, optimal.

2.3 Multiplication by a power of two

When working with binary numbers it is well known that multiplication by a power of two, $P = A \cdot 2^k$, is a simple operation, since the representation of P is simply obtained by shifting the representation of A by k positions to the left and setting the k least significant bits to zero. Let $A = a_{n-1}a_{n-2}\ldots a_0$. Formally,

$$P = A2^k \quad = \quad 2^k \sum_{i=0}^{n-1} a_i 2^i$$

$$
\begin{aligned}
&= \sum_{i=0}^{n-1} a_i 2^{i+k} \\
&= \left(\sum_{i=0}^{k-1} 0 \cdot 2^i\right) + \left(\sum_{i=k}^{n+k-1} a_{i-k} \cdot 2^i\right) \\
&= \sum_{i=0}^{n+k-1} p_i \cdot 2^i,
\end{aligned}
$$

where, in the last formula, $p_i = 0$ for $0 \leq i < k$, and $p_i = a_{i-k}$ for $k \leq i < n + k$. A simple algorithm implementing this operation follows.

```
SHIFT(A, k)
n ← length(A)
for i ← 0 to k − 1 do
    p_i ← 0
for i ← k to n + k − 1 do
    p_i ← a_{i−k}
return P
```

Observe that under our cost model, the above algorithm has null complexity (which is clearly optimal!). This is justifiable in actual machines, where shifts are performed much more quickly than arithmetic/logic operations. In a model where bit assignments cost unit time, however, the running time would become $O(n + k)$, which is linear in the size of the output, hence still optimal.

2.4 Integer Multiplication

We have finally come to the most complex operation of our basic suite for integer arithmetic: general multiplication. We will first examine the elementary algorithm, whose quadratic complexity is however rather far from optimal. Then, we discuss a simple divide-and-conquer approach which does not yield a faster algorithm. Improving the inefficient divide-and-conquer algorithm, using a trick devised by the Russian mathematician Karatsuba, we finally succeed in reducing the asymptotic complexity to $O\left(n^{\log_2 3}\right) = O\left(n^{1.58\cdots}\right)$. In what follows, we assume that the numbers being multiplied have the same size. Minor modifications are sufficient to deal with the more general case.

2.4.1 Elementary algorithm

As we did for addition and subtraction, let us start with the familiar elementary algorithm for multiplication, which basically obtains the product P of two integers $A = a_{n-1}a_{n-2}\ldots a_0$

and $B = b_{n-1}b_{n-2}\ldots b_0$ by summing a number of partial products. More specifically, for each "set bit" $b_i = 1$ of the multiplicator B, we sum a partial product obtained by shifting the multiplicand A by i positions to the left. The correctness of the above approach is based on the following simple derivation:

$$\begin{aligned} P = AB &= A\sum_{i=0}^{n-1} b_i 2^i \\ &= \sum_{i=0}^{n-1} b_i (A \cdot 2^i) \\ &= \sum_{\{i:b_i=1\}} \text{SHIFT}(A, i). \end{aligned}$$

The algorithm follows.

```
MUL(A, B)
n ← length(B)
P ← 0
for i ← 0 to n − 1 do
    if b_i = 1  then P ← SUM(P, SHIFT(A, i))
return P
```

Let us now evaluate the running time of the algorithm. Observe that in the iteration of index i of the **for** loop, we perform an addition between two numbers of at most $n+i$ bits (this property can be easily proved by induction on i: we leave it to the reader). Therefore, the ith iteration has cost $T_{\text{SUM}}(n+i) = 7(n+i)$, in the worst case. The total cost of the algorithm is therefore

$$\begin{aligned} T_{\text{MUL}}(n) &= \sum_{i=0}^{n-1} 7(n+i) \\ &= 7n^2 + 7n(n-1)/2 \\ &= \Theta\left(n^2\right). \end{aligned}$$

We have obtained a quadratic algorithm! Since we are far from the natural linear lower-bound on integer multiplication, we will now examine alternative strategies with the hope of obtaining a better algorithm.

2.4.2 A simple divide-and-conquer approach

In what follows, we make the simplyfing assumption that n is a power of two. Let $A = a_{n-1}a_{n-2}\ldots a_0$ and $B = b_{n-1}b_{n-2}\ldots b_0$ be the numbers that have to be multiplied. Split in half the two representations as follows:

$$A = \underbrace{a_{n-1}\ldots a_{n/2}}_{A_1}\underbrace{a_{n/2-1}\ldots a_0}_{A_0} \text{ and } B = \underbrace{b_{n-1}\ldots b_{n/2}}_{B_1}\underbrace{b_{n/2-1}\ldots b_0}_{B_0}.$$

Consider now the four halfs A_1, A_0, B_1, B_0 as four integers whose representation is

$$A_1 = \sum_{i=0}^{n/2-1} a_{i+n/2}2^i, A_0 = \sum_{i=0}^{n/2-1} a_i 2^i, B_1 = \sum_{i=0}^{n/2-1} b_{i+n/2}2^i, B_0 = \sum_{i=0}^{n/2-1} b_i 2^i. \tag{2.1}$$

Observe that

$$\begin{aligned} A &= \sum_{i=0}^{n-1} a_i 2^i \\ &= \sum_{i=0}^{n/2-1} a_i 2^i + \sum_{i=n/2}^{n-1} a_i 2^i \\ &= A_0 + 2^{n/2} \sum_{j=0}^{n/2-1} a_{j+n/2} 2^j \\ &= A_0 + 2^{n/2} A_1, \end{aligned}$$

where the third passage is easily obtained by substituting $j = i - n/2$ as the index of the second summation. Clearly, a similar derivation can be made for B. Observe that we have expressed an n-bit number as a linear combination of the two $n/2$-bit numbers corresponding to its two halves. This is crucial in setting up a recursive strategy. Indeed, we have that

$$\begin{aligned} P = AB &= (A_0 + 2^{n/2}A_1)(B_0 + 2^{n/2}B_1) \\ &= A_0B_0 + 2^{n/2}(A_0B_1 + A_1B_0) + 2^n A_1B_1, \end{aligned} \tag{2.2}$$

therefore we can obtain P by a simple combination (based on sums and shifts) of the four subproducts of the $n/2$-bit halves defined in Equation 2.1, and these subproducts can be obtained recursively. As for the base case, observe that when $n = 1$ numbers A and B are simply the bits a_0 and b_0, hence their product is $a_o \wedge b_0$. The algorithm follows.

RMUL(A, B)
$n \leftarrow \text{length}(A)$
if $n = 1$
  then return $(a_0$ **and** $b_0)$
$A_1 \leftarrow a_{n-1} \dots a_{n/2}$ $A_0 \leftarrow a_{n/2-1} \dots a_0$
$B_1 \leftarrow b_{n-1} \dots b_{n/2}$ $B_0 \leftarrow b_{n/2-1} \dots b_0$
$P_{00} \leftarrow \text{RMUL}(A_0, B_0)$ $P_{01} \leftarrow \text{RMUL}(A_0, B_1)$
$P_{10} \leftarrow \text{RMUL}(A_1, B_0)$ $P_{11} \leftarrow \text{RMUL}(A_1, B_1)$
$P_{01} \leftarrow \text{SHIFT}(\text{SUM}(P_{01}, P_{10}), n/2)$
$P_{11} \leftarrow \text{SHIFT}(P_{11}, n)$
return $\text{SUM}(\text{SUM}(P_{00}, P_{01}), P_{11})$

The correctness of RMUL follows immediately from induction and the decomposition of Equation 2.2. Since the conquer phase consists of a constant number of calls to the SUM and SHIFT subroutines on numbers whose size is at most $2n$, and we execute four recursive calls of size $n/2$, the running time obeys to the following recurrence:

$$T_{\text{RMUL}}(n) = \begin{cases} 1 & n = 1, \\ 4T_{\text{RMUL}}(n/2) + cn & n > 1. \end{cases} \tag{2.3}$$

where c is a given constant (depending on the calls to SUM in the conquer). Since $cn = O\left(n^{\log_2 4}\right)$, by appliying the Master Theorem (case 1) we obtain

$$T_{\text{RMUL}}(n) = \Theta\left(n^2\right).$$

Clearly, the new algorithm does not improve upon the running time of the elementary one. In fact, in practice algorithm MUL is definitively preferable to RMUL, since there is no overhead due to recursion. However, in the next subsection we will start from the recursive framework which lead to RMUL and modify it cleverly to obtain a faster multiplication algorithm.

2.4.3 Karatsuba's Algorithm

In order to build an intuition on possible ways of improving RMUL, let us have a closer look at its recurrence, specified in Equation 2.3. Since the recurrence falls in Case 1 of the Master Theorem, it means that it is the large number of recursive calls that dominates the running time, rather than the cost of the conquer phase. In other words, if we are able to bring down the number of recursive calls, even at the expense of increasing the time for the conquer phase, we will obtain a better algorithm.

Karatsuba's algorithm implements the above intuition by bringing down the number of recursive calls down to three, at the expense of performing more work in the divide and conquer phases of the algorithm. The combined cost of the divide and conquer phases remains linear but exhibits a rather larger constant than the one associated to algorithm RMUL.

Karatsuba's way of decomposing $P = AB$ into a linear combination of smaller products is the following. Let A_1, A_0, B_1, B_0 be the four numbers obtained by halving the representation of A and B, as illustrated in Equation 2.1, and consider the following three products:

$$U = (A_0 + A_1) \cdot (B_0 + B_1) \quad V = A_1 B_1 \quad W = A_0 B_0$$

Observe that while V and W appeared as subproducts in Equation 2.2, U is a new subproduct obtained by multiplying the sums of the two halves of each operand. The crucial idea behind the algorithm is the following: *we can obtain the same decomposition of Equation 2.2 as a linear combination of subproducts U, V, W.* To this aim, it is sufficient to observe that

$$\begin{aligned} U - V - W &= (A_0 + A_1) \cdot (B_0 + B_1) - A_1 B_1 - A_0 B_0 \\ &= A_0 B_1 + A_1 B_0 \end{aligned}$$

hence

$$\begin{aligned} P = AB &= A_0 B_0 + 2^{n/2}(A_0 B_1 + A_1 B_0) + 2^n A_1 B_1 \\ &= W + 2^{n/2}(U - V - W) + 2^n V \end{aligned} \tag{2.4}$$

Note that the conquer phase implied by Equation 2.4 performs a constant number of sums and subtractions on operands of linear size, hence its total cost is still linear (with a higher constant than before).

Before writing the code of the new algorithm, let us make sure that we have obtained a better algorithm and that there are no "hidden" problems with the above approach. At a first glance, everything seems to work: by keeping the same base case of RMUL, the running time of an algorithm implementing the above approach seems to comply with the recurrence:

$$T(n) = \begin{cases} 1 & n = 1, \\ 3T(n/2) + kn & n > 1, \end{cases}$$

where k is a constant depending on the sums and subtractions executed both in the divide

and in the conquer phase of the algorithm. By the Master Theorem, the recurrence would yield a running time $T(n) = \Theta\left(n^{\log_2 3}\right) = \Theta\left(n^{1.584\ldots}\right)$ which is asymptotically much better than $\Theta(n^2)$. However, there are two minor problems that have to be fixed, in order to obtain a working algorithm:

1. Recall that the sum of two numbers of $n/2$ bits may yield a result with $n/2+1$ significant bits: this implies that, even if we start with operands whose size is a power of two, we generate subinstances whose size is not necessarly a power of two. In turn, this implies that we end up generating instances of different sizes. Consider, for instance, the case $n = 8$: we generate an instance of size $8/2+1 = 5$ (to obtain U) and two instances of size $8/2 = 4$ (for V and W). In turn, the instance of size 5 would generate an instance of size $\lceil 5/2 \rceil + 1 = 4$ (U), an instance of size $\lceil 5/2 \rceil = 3$ and an instance of size $\lfloor 5/2 \rfloor = 2$ (V and W). In general, the size of the three instances, in the worst case, is $\lceil n/2 \rceil + 1$, $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$. Finally, note that Equation 2.2 must be re-written to account for odd values of n. By letting A_0 and B_0 be of size $\lfloor n/2 \rfloor$ we obtain:

$$\begin{aligned} P = AB &= A_0B_0 + 2^{\lfloor n/2 \rfloor}(A_0B_1 + A_1B_0) + 2^{2\lfloor n/2 \rfloor}A_1B_1, \\ &= W + 2^{\lfloor n/2 \rfloor}(U - V - W) + 2^{2\lfloor n/2 \rfloor}V \end{aligned} \tag{2.5}$$

2. Because of these new instance sizes, the divide phase may fail to generate subinstances whose size is strictly smaller than the size of the input instance: indeed, for $n \leq 3$ we have $\lceil n/2 \rceil + 1 \geq n$. In order to fix this problem, we have to increase the value n_0 for which the base case applies, and solve the instances of size at most n_0 nonrecursively, for instance using algorithm MUL. Clearly, it suffices to choose $n_0 = 3$.

We are now ready to give the code for Karatsuba's algorithm:

KMUL(A, B)
$n \leftarrow$ length(A)
if $n \leq 3$
  then return MUL(A, B)
$A_1 \leftarrow a_{n-1} \ldots a_{\lfloor n/2 \rfloor}$ $A_0 \leftarrow a_{\lfloor n/2 \rfloor - 1} \ldots a_0$
$B_1 \leftarrow b_{n-1} \ldots b_{\lfloor n/2 \rfloor}$ $B_0 \leftarrow b_{\lfloor n/2 \rfloor - 1} \ldots b_0$
$\bar{A} \leftarrow$ SUM(A_1, A_0)
$\bar{B} \leftarrow$ SUM(B_1, B_0)
$U \leftarrow$ KMUL$(\bar{A}, \bar{B})$ $V \leftarrow$ KMUL(A_1, B_1) $W \leftarrow$ KMUL(A_0, B_0)
$U \leftarrow$ SHIFT(SUB$(U,$ SUM$(V, W))), \lfloor n/2 \rfloor)$
$V \leftarrow$ SHIFT$(V, 2\lfloor n/2 \rfloor)$
return SUM(SUM$(U, V), W)$

The correctness of KMUL follows from induction and from the considerations made above. Indeed, in the base case $n \leq 3$ KMUL is correct since algorithm MUL is correct. For $n > 3$, the divide phase correctly generates instances of size smaller than n, while the conquer phase performs the merge correctly as prescribed by Equation 2.5.

The running time of the above algorithm obeys the following recurrence:

$$T_{\text{KMUL}}(n) = \begin{cases} k_1 & n \leq 3, \\ T_{\text{KMUL}}(\lceil n/2 \rceil + 1) + T_{\text{KMUL}}(\lceil n/2 \rceil) + T_{\text{KMUL}}(\lfloor n/2 \rfloor) + k_2 n & n > 3, \end{cases}$$

were k_1 and k_2 are two constants. Note that this recurrence is nonstandard and cannot be handled by the Master Theorem. As a simple consequence of the Master Theorem, however, we obtain the lower bound $T_{\text{KMUL}}(n) = \Omega\left(n^{\log_2 3}\right)$ (we leave the proof to the reader). To obtain a matching upper bound, consider the recursion tree associated to the recurrence. The tree is ternary but, unfortunately, it is not necessarily complete and the sizes of the subinstances at a given level are not all the same. However, we can easily prove by induction that the maximum subinstance size s_ℓ at level $\ell \geq 0$ of the tree satisfies the inequality $s_\ell \leq n/2^\ell + 3$. The base case $\ell = 0$ is trivial, since the size of the instance at the root is $n \leq n + 3$. If the bound holds at level $\ell - 1$, at level ℓ we have:

$$\begin{aligned} s_\ell &\leq \lceil s_{\ell-1}/2 \rceil + 1 \\ &\leq (s_{\ell-1} + 1)/2 + 1 \\ &\leq (n/2^{\ell-1} + 3 + 1)/2 + 1 \\ &= n/2^\ell + 3. \end{aligned}$$

The above bound also helps us bound the number of levels in the tree, since $s_{\lceil \log_2 n \rceil} \leq 4$. This is all we need to obtain an upper bound, since the real recursion tree surely yields a smaller time than a complete, ternary tree with all the leaves at level $\lceil \log_2 n \rceil + 1$, where the size of the 3^ℓ instances at level $\ell \leq \lceil \log_2 n \rceil$ is s_ℓ, and contributes time $k_2 s_\ell$, while leaves at level $\lceil \log_2 n \rceil + 1$ contribute time k_1 . Putting it all together, we obtain:

$$\begin{aligned} T_{\text{KMUL}}(n) &\leq \underbrace{\sum_{\ell=0}^{\lceil \log_2 n \rceil} 3^\ell k_2 s_\ell}_{\text{internal nodes}} + \underbrace{3^{\lceil \log_2 n \rceil + 1} k_1}_{\text{leaves}} \\ &\leq \sum_{\ell=0}^{\lceil \log_2 n \rceil} 3^\ell k_2 (n/2^\ell + 3) + k_1 3^{\log_2 n + 2} \\ &\leq ((9/2)k_2 + (27/2)k_2 + 9k_1) n^{\log_2 3} \end{aligned}$$

$$
\begin{aligned}
&= 9(2k_2 + k_1)n^{\log_2 3} \\
&= O\left(n^{\log_2 3}\right).
\end{aligned}
$$

We have shown that $T_{\text{KMUL}}(n)$ is both $\Omega\left(n^{\log_2 3}\right)$ and $O\left(n^{\log_2 3}\right)$, hence it follows that $T_{\text{KMUL}}(n) = \Theta\left(n^{\log_2 3}\right)$.

Although asymptotically faster, the large constants hidden in the running time of algorithm KMUL make it superior to algorithm MUL only for relatively large integers. This situation is even worse in practice, since it is much more complex to obtain a good implementation of recursive KMUL than iterative MUL. However, nowadays cryptographic applications require extremely large integers whose representation is often thousands of bits long. For such applications, a clever implementation of KMUL outperforms MUL.

Karatsuba's algorithm is not the fastest algorithm known for integer multiplication. A more complex algorithm due to Swiss scientists Schönhage and Strassen, based on a sophisticated application of the Fast Fourier Transform, attains an $O(n \log n \log\log n)$ running time. As for Karatsuba, the Schönhage-Strassen algorithm features large constants in the running time, hence its application is again limited to very large instances.