1

Integer arithmetic

In this chapter, our main topic is integer arithmetic. However, we shall see that many algorithms for polynomial arithmetic are similar to the corresponding algorithms for integer arithmetic, but simpler due to the lack of carries in polynomial arithmetic. Consider for example addition: the sum of two polynomials of degree n always has degree at most n, whereas the sum of two n-digit integers may have $n+1$ digits. Thus, we often describe algorithms for polynomials as an aid to understanding the corresponding algorithms for integers.

1.1 Representation and notations

We consider in this chapter algorithms working on integers. We distinguish between the logical – or mathematical – representation of an integer, and its physical representation on a computer. Our algorithms are intended for "large" integers – they are not restricted to integers that can be represented in a single computer word.

Several physical representations are possible. We consider here only the most common one, namely a dense representation in a fixed base. Choose an integral *base* $\beta > 1$. (In case of ambiguity, β will be called the *internal* base.) A positive integer A is represented by the length n and the digits a_i of its base β expansion

$$A = a_{n-1}\beta^{n-1} + \cdots + a_1\beta + a_0,$$

where $0 \leq a_i \leq \beta - 1$, and a_{n-1} is sometimes assumed to be non-zero. Since the base β is usually fixed in a given program, only the length n and the integers $(a_i)_{0\leq i<n}$ need to be stored. Some common choices for β are 2^{32} on a 32-bit computer, or 2^{64} on a 64-bit machine; other possible choices

are respectively 10^9 and 10^{19} for a decimal representation, or 2^{53} when using double-precision floating-point registers. Most algorithms given in this chapter work in any base; the exceptions are explicitly mentioned.

We assume that the sign is stored separately from the absolute value. This is known as the "sign-magnitude" representation. Zero is an important special case; to simplify the algorithms we assume that $n = 0$ if $A = 0$, and we usually assume that this case is treated separately.

Except when explicitly mentioned, we assume that all operations are *off-line*, i.e. all inputs (resp. outputs) are completely known at the beginning (resp. end) of the algorithm. Different models include *lazy* and *relaxed* algorithms, and are discussed in the Notes and references (§1.9).

1.2 Addition and subtraction

As an explanatory example, here is an algorithm for integer addition. In the algorithm, d is a *carry* bit.

Our algorithms are given in a language that mixes mathematical notation and syntax similar to that found in many high-level computer languages. It should be straightforward to translate into a language such as C. Note that ":=" indicates a definition, and "←" indicates assignment. Line numbers are included if we need to refer to individual lines in the description or analysis of the algorithm.

Algorithm 1.1 IntegerAddition

Input: $A = \sum_0^{n-1} a_i\beta^i$, $B = \sum_0^{n-1} b_i\beta^i$, carry-in $0 \le d_{\text{in}} \le 1$
Output: $C := \sum_0^{n-1} c_i\beta^i$ and $0 \le d \le 1$ such that $A + B + d_{\text{in}} = d\beta^n + C$

1: $d \leftarrow d_{\text{in}}$
2: **for** i **from** 0 **to** $n-1$ **do**
3: $\quad s \leftarrow a_i + b_i + d$
4: $\quad (d, c_i) \leftarrow (s \operatorname{div} \beta, s \bmod \beta)$
5: return C, d.

Let T be the number of different values taken by the data type representing the coefficients a_i, b_i. (Clearly, $\beta \le T$, but equality does not necessarily hold, for example $\beta = 10^9$ and $T = 2^{32}$.) At step 3, the value of s can be as large as $2\beta - 1$, which is not representable if $\beta = T$. Several workarounds are possible: either use a machine instruction that gives the possible carry of $a_i + b_i$, or use the fact that, if a carry occurs in $a_i + b_i$, then the computed

sum – if performed modulo T – equals $t := a_i + b_i - T < a_i$; thus, comparing t and a_i will determine if a carry occurred. A third solution is to keep a bit in reserve, taking $\beta \le T/2$.

The subtraction code is very similar. Step 3 simply becomes $s \leftarrow a_i - b_i + d$, where $d \in \{-1, 0\}$ is the *borrow* of the subtraction, and $-\beta \le s < \beta$. The other steps are unchanged, with the invariant $A - B + d_{\text{in}} = d\beta^n + C$.

We use the *arithmetic complexity* model, where *cost* is measured by the number of machine instructions performed, or equivalently (up to a constant factor) the *time* on a single processor.

Addition and subtraction of n-word integers cost $O(n)$, which is negligible compared to the multiplication cost. However, it is worth trying to reduce the constant factor implicit in this $O(n)$ cost. We shall see in §1.3 that "fast" multiplication algorithms are obtained by replacing multiplications by additions (usually more additions than the multiplications that they replace). Thus, the faster the additions are, the smaller will be the thresholds for changing over to the "fast" algorithms.

1.3 Multiplication

A nice application of large integer multiplication is the *Kronecker–Schönhage trick*, also called *segmentation* or *substitution* by some authors. Assume we want to multiply two polynomials, $A(x)$ and $B(x)$, with non-negative integer coefficients (see Exercise 1.1 for negative coefficients). Assume both polynomials have degree less than n, and the coefficients are bounded by ρ. Now take a power $X = \beta^k > n\rho^2$ of the base β, and multiply the integers $a = A(X)$ and $b = B(X)$ obtained by evaluating A and B at $x = X$. If $C(x) = A(x)B(x) = \sum c_i x^i$, we clearly have $C(X) = \sum c_i X^i$. Now since the c_i are bounded by $n\rho^2 < X$, the coefficients c_i can be retrieved by simply "reading" blocks of k words in $C(X)$. Assume for example that we want to compute

$$(6x^5 + 6x^4 + 4x^3 + 9x^2 + x + 3)(7x^4 + x^3 + 2x^2 + x + 7),$$

with degree less than $n = 6$, and coefficients bounded by $\rho = 9$. We can take $X = 10^3 > n\rho^2$, and perform the integer multiplication

$$\begin{aligned} &6\,006\,004\,009\,001\,003 \times 7\,001\,002\,001\,007 \\ &= 42\,048\,046\,085\,072\,086\,042\,070\,010\,021, \end{aligned}$$

from which we can read off the product

$$42x^9 + 48x^8 + 46x^7 + 85x^6 + 72x^5 + 86x^4 + 42x^3 + 70x^2 + 10x + 21.$$

Conversely, suppose we want to multiply two integers $a = \sum_{0 \le i < n} a_i \beta^i$ and $b = \sum_{0 \le j < n} b_j \beta^j$. Multiply the polynomials $A(x) = \sum_{0 \le i < n} a_i x^i$ and $B(x) = \sum_{0 \le j < n} b_j x^j$, obtaining a polynomial $C(x)$, then evaluate $C(x)$ at $x = \beta$ to obtain ab. Note that the coefficients of $C(x)$ may be larger than β, in fact they may be up to about $n\beta^2$. For example, with $a = 123$, $b = 456$, and $\beta = 10$, we obtain $A(x) = x^2 + 2x + 3$, $B(x) = 4x^2 + 5x + 6$, with product $C(x) = 4x^4 + 13x^3 + 28x^2 + 27x + 18$, and $C(10) = 56088$. These examples demonstrate the analogy between operations on polynomials and integers, and also show the limits of the analogy.

A common and very useful notation is to let $M(n)$ denote the time to multiply n-bit integers, or polynomials of degree $n - 1$, depending on the context. In the polynomial case, we assume that the cost of multiplying coefficients is constant; this is known as the *arithmetic complexity* model, whereas the *bit complexity* model also takes into account the cost of multiplying coefficients, and thus their bit-size.

1.3.1 Naive multiplication

Algorithm 1.2 BasecaseMultiply

Input: $A = \sum_0^{m-1} a_i \beta^i$, $B = \sum_0^{n-1} b_j \beta^j$
Output: $C = AB := \sum_0^{m+n-1} c_k \beta^k$

1: $C \leftarrow A \cdot b_0$
2: **for** j **from** 1 **to** $n - 1$ **do**
3: $\quad C \leftarrow C + \beta^j (A \cdot b_j)$
4: return C.

Theorem 1.1 *Algorithm* **BasecaseMultiply** *computes the product AB correctly, and uses $\Theta(mn)$ word operations.*

The multiplication by β^j at step 3 is trivial with the chosen dense representation; it simply requires shifting by j words towards the most significant words. The main operation in Algorithm **BasecaseMultiply** is the computation of $A \cdot b_j$ and its accumulation into C at step 3. Since all fast algorithms rely on multiplication, the most important operation to optimize in multiple-precision software is thus the multiplication of an array of m words by one word, with accumulation of the result in another array of $m + 1$ words.

We sometimes call Algorithm **BasecaseMultiply** *schoolbook multiplication* since it is close to the "long multiplication" algorithm that used to be taught at school.

Since multiplication with accumulation usually makes extensive use of the pipeline, it is best to give it arrays that are as long as possible, which means that A rather than B should be the operand of larger size (i.e. $m \geq n$).

1.3.2 Karatsuba's algorithm

Karatsuba's algorithm is a "divide and conquer" algorithm for multiplication of integers (or polynomials). The idea is to reduce a multiplication of length n to three multiplications of length $n/2$, plus some overhead that costs $O(n)$.

In the following, $n_0 \geq 2$ denotes the threshold between naive multiplication and Karatsuba's algorithm, which is used for n_0-word and larger inputs. The optimal "Karatsuba threshold" n_0 can vary from about ten to about 100 words, depending on the processor and on the relative cost of multiplication and addition (see Exercise 1.6).

Algorithm 1.3 KaratsubaMultiply

Input: $A = \sum_0^{n-1} a_i\beta^i$, $B = \sum_0^{n-1} b_j\beta^j$
Output: $C = AB := \sum_0^{2n-1} c_k\beta^k$
  if $n < n_0$ **then** return **BasecaseMultiply**(A, B)
  $k \leftarrow \lceil n/2 \rceil$
  $(A_0, B_0) := (A, B) \bmod \beta^k$, $(A_1, B_1) := (A, B) \operatorname{div} \beta^k$
  $s_A \leftarrow \operatorname{sign}(A_0 - A_1)$, $s_B \leftarrow \operatorname{sign}(B_0 - B_1)$
  $C_0 \leftarrow$ **KaratsubaMultiply**(A_0, B_0)
  $C_1 \leftarrow$ **KaratsubaMultiply**(A_1, B_1)
  $C_2 \leftarrow$ **KaratsubaMultiply**$(|A_0 - A_1|, |B_0 - B_1|)$
  return $C := C_0 + (C_0 + C_1 - s_A s_B C_2)\beta^k + C_1\beta^{2k}$.

Theorem 1.2 *Algorithm* **KaratsubaMultiply** *computes the product AB correctly, using $K(n) = O(n^\alpha)$ word multiplications, with $\alpha = \lg 3 \approx 1.585$.*

Proof. Since $s_A|A_0 - A_1| = A_0 - A_1$ and $s_B|B_0 - B_1| = B_0 - B_1$, we have $s_A s_B|A_0 - A_1||B_0 - B_1| = (A_0 - A_1)(B_0 - B_1)$, and thus $C = A_0B_0 + (A_0B_1 + A_1B_0)\beta^k + A_1B_1\beta^{2k}$.

Since A_0, B_0, $|A_0 - A_1|$ and $|B_0 - B_1|$ have (at most) $\lceil n/2 \rceil$ words, and A_1 and B_1 have (at most) $\lfloor n/2 \rfloor$ words, the number $K(n)$ of word multiplications

satisfies the recurrence $K(n) = n^2$ for $n < n_0$, and $K(n) = 2K(\lceil n/2 \rceil) + K(\lfloor n/2 \rfloor)$ for $n \geq n_0$. Assume $2^{\ell-1}n_0 < n \leq 2^\ell n_0$ with $\ell \geq 1$. Then $K(n)$ is the sum of three $K(j)$ values with $j \leq 2^{\ell-1}n_0$, so at most 3^ℓ $K(j)$ with $j \leq n_0$. Thus, $K(n) \leq 3^\ell \max(K(n_0), (n_0-1)^2)$, which gives $K(n) \leq Cn^\alpha$ with $C = 3^{1-\lg(n_0)}\max(K(n_0), (n_0-1)^2)$. □

Different variants of Karatsuba's algorithm exist; the variant presented here is known as the *subtractive* version. Another classical one is the *additive* version, which uses A_0+A_1 and B_0+B_1 instead of $|A_0-A_1|$ and $|B_0-B_1|$. However, the subtractive version is more convenient for integer arithmetic, since it avoids the possible carries in $A_0 + A_1$ and $B_0 + B_1$, which require either an extra word in these sums, or extra additions.

The efficiency of an implementation of Karatsuba's algorithm depends heavily on memory usage. It is important to avoid allocating memory for the intermediate results $|A_0 - A_1|$, $|B_0 - B_1|$, C_0, C_1, and C_2 at each step (although modern compilers are quite good at optimizing code and removing unnecessary memory references). One possible solution is to allow a large temporary storage of m words, used both for the intermediate results and for the recursive calls. It can be shown that an auxiliary space of $m = 2n$ words – or even $m = O(\log n)$ – is sufficient (see Exercises 1.7 and 1.8).

Since the product C_2 is used only once, it may be faster to have auxiliary routines **KaratsubaAddmul** and **KaratsubaSubmul** that accumulate their results, calling themselves recursively, together with **KaratsubaMultiply** (see Exercise 1.10).

The version presented here uses $\sim 4n$ additions (or subtractions): $2 \times (n/2)$ to compute $|A_0 - A_1|$ and $|B_0 - B_1|$, then n to add C_0 and C_1, again n to add or subtract C_2, and n to add $(C_0 + C_1 - s_A s_B C_2)\beta^k$ to $C_0 + C_1\beta^{2k}$. An improved scheme uses only $\sim 7n/2$ additions (see Exercise 1.9).

When considered as algorithms on polynomials, most fast multiplication algorithms can be viewed as evaluation/interpolation algorithms. Karatsuba's algorithm regards the inputs as polynomials A_0+A_1x and B_0+B_1x evaluated at $x = \beta^k$; since their product $C(x)$ is of degree 2, Lagrange's interpolation theorem says that it is sufficient to evaluate $C(x)$ at three points. The subtractive version evaluates[1] $C(x)$ at $x = 0, -1, \infty$, whereas the additive version uses $x = 0, +1, \infty$.

1.3.3 Toom–Cook multiplication

Karatsuba's idea readily generalizes to what is known as Toom–Cook r-way multiplication. Write the inputs as $a_0+\cdots+a_{r-1}x^{r-1}$ and $b_0+\cdots+b_{r-1}x^{r-1}$,

[1] Evaluating $C(x)$ at ∞ means computing the product A_1B_1 of the leading coefficients.

with $x = \beta^k$, and $k = \lceil n/r \rceil$. Since their product $C(x)$ is of degree $2r-2$, it suffices to evaluate it at $2r-1$ distinct points to be able to recover $C(x)$, and in particular $C(\beta^k)$. If r is chosen optimally, Toom–Cook multiplication of n-word numbers takes time $n^{1+O(1/\sqrt{\log n})}$.

Most references, when describing subquadratic multiplication algorithms, only describe Karatsuba and FFT-based algorithms. Nevertheless, the Toom–Cook algorithm is quite interesting in practice.

Toom–Cook r-way reduces one n-word product to $2r-1$ products of about n/r words, thus costs $O(n^\nu)$ with $\nu = \log(2r-1)/\log r$. However, the constant hidden by the big-O notation depends strongly on the evaluation and interpolation formulæ, which in turn depend on the chosen points. One possibility is to take $-(r-1), \ldots, -1, 0, 1, \ldots, (r-1)$ as evaluation points.

The case $r = 2$ corresponds to Karatsuba's algorithm (§1.3.2). The case $r = 3$ is known as Toom–Cook 3-way, sometimes simply called "the Toom–Cook algorithm". Algorithm **ToomCook3** uses the evaluation points 0, 1, -1, 2, ∞, and tries to optimize the evaluation and interpolation formulæ.

Algorithm 1.4 ToomCook3
Input: two integers $0 \le A, B < \beta^n$
Output: $AB := c_0 + c_1\beta^k + c_2\beta^{2k} + c_3\beta^{3k} + c_4\beta^{4k}$ with $k = \lceil n/3 \rceil$
Require: a threshold $n_1 \ge 3$
1: **if** $n < n_1$ **then** return **KaratsubaMultiply**(A, B)
2: write $A = a_0 + a_1x + a_2x^2$, $B = b_0 + b_1x + b_2x^2$ with $x = \beta^k$.
3: $v_0 \leftarrow$ **ToomCook3**(a_0, b_0)
4: $v_1 \leftarrow$ **ToomCook3**$(a_{02}+a_1, b_{02}+b_1)$ where $a_{02} \leftarrow a_0+a_2, b_{02} \leftarrow b_0+b_2$
5: $v_{-1} \leftarrow$ **ToomCook3**$(a_{02} - a_1, b_{02} - b_1)$
6: $v_2 \leftarrow$ **ToomCook3**$(a_0 + 2a_1 + 4a_2, b_0 + 2b_1 + 4b_2)$
7: $v_\infty \leftarrow$ **ToomCook3**(a_2, b_2)
8: $t_1 \leftarrow (3v_0 + 2v_{-1} + v_2)/6 - 2v_\infty$, $t_2 \leftarrow (v_1 + v_{-1})/2$
9: $c_0 \leftarrow v_0$, $c_1 \leftarrow v_1 - t_1$, $c_2 \leftarrow t_2 - v_0 - v_\infty$, $c_3 \leftarrow t_1 - t_2$, $c_4 \leftarrow v_\infty$.

The divisions at step 8 are exact; if β is a power of two, the division by 6 can be done using a division by 2 – which consists of a single shift – followed by a division by 3 (see §1.4.7).

Toom–Cook r-way has to invert a $(2r-1) \times (2r-1)$ Vandermonde matrix with parameters the evaluation points; if we choose consecutive integer points, the determinant of that matrix contains all primes up to $2r-2$. This proves that division by (a multiple of) 3 can not be avoided for Toom–Cook 3-way with consecutive integer points. See Exercise 1.14 for a generalization of this result.

1.3.4 Use of the fast Fourier transform (FFT)

Most subquadratic multiplication algorithms can be seen as evaluation-interpolation algorithms. They mainly differ in the number of evaluation points, and the values of those points. However, the evaluation and interpolation formulæ become intricate in Toom–Cook r-way for large r, since they involve $O(r^2)$ scalar operations. The fast Fourier transform (FFT) is a way to perform evaluation and interpolation efficiently for some special points (roots of unity) and special values of r. This explains why multiplication algorithms with the best known asymptotic complexity are based on the FFT.

There are different flavours of FFT multiplication, depending on the ring where the operations are performed. The Schönhage–Strassen algorithm, with a complexity of $O(n \log n \log\log n)$, works in the ring $\mathbb{Z}/(2^n + 1)\mathbb{Z}$. Since it is based on modular computations, we describe it in Chapter 2.

Other commonly used algorithms work with floating-point complex numbers. A drawback is that, due to the inexact nature of floating-point computations, a careful error analysis is required to guarantee the correctness of the implementation, assuming an underlying arithmetic with rigorous error bounds. See Theorem 3.6 in Chapter 3.

We say that multiplication is *in the FFT range* if n is large and the multiplication algorithm satisfies $M(2n) \sim 2M(n)$. For example, this is true if the Schönhage–Strassen multiplication algorithm is used, but not if the classical algorithm or Karatsuba's algorithm is used.

1.3.5 Unbalanced multiplication

The subquadratic algorithms considered so far (Karatsuba and Toom–Cook) work with equal-size operands. How do we efficiently multiply integers of different sizes with a subquadratic algorithm? This case is important in practice, but is rarely considered in the literature. Assume the larger operand has size m, and the smaller has size $n \leq m$, and denote by $M(m, n)$ the corresponding multiplication cost.

If evaluation-interpolation algorithms are used, the cost depends mainly on the size of the result, i.e. $m + n$, so we have $M(m, n) \leq M((m + n)/2)$, at least approximately. We can do better than $M((m+n)/2)$ if n is much smaller than m, for example $M(m, 1) = O(m)$.

When m is an exact multiple of n, say $m = kn$, a trivial strategy is to cut the larger operand into k pieces, giving $M(kn, n) = kM(n) + O(kn)$. However, this is not always the best strategy, see Exercise 1.16.

When m is not an exact multiple of n, several strategies are possible:

- split the two operands into an equal number of pieces of unequal sizes;
- or split the two operands into different numbers of pieces.

Each strategy has advantages and disadvantages. We discuss each in turn.

First strategy: equal number of pieces of unequal sizes

Consider for example Karatsuba multiplication, and let $K(m,n)$ be the number of word-products for an $m \times n$ product. Take for example $m = 5$, $n = 3$. A natural idea is to pad the smaller operand to the size of the larger one. However, there are several ways to perform this padding, as shown in the following figure, where the "Karatsuba cut" is represented by a double column:

a_4	a_3	a_2	a_1	a_0
		b_2	b_1	b_0

$A \times B$

a_4	a_3	a_2	a_1	a_0
	b_2	b_1	b_0	

$A \times (\beta B)$

a_4	a_3	a_2	a_1	a_0
b_2	b_1	b_0		

$A \times (\beta^2 B)$

The left variant leads to two products of size 3, i.e. $2K(3,3)$, the middle one to $K(2,1)+K(3,2)+K(3,3)$, and the right one to $K(2,2)+K(3,1)+K(3,3)$, which give respectively 14, 15, 13 word-products.

However, whenever $m/2 \le n \le m$, any such "padding variant" will require $K(\lceil m/2 \rceil, \lceil m/2 \rceil)$ for the product of the differences (or sums) of the low and high parts from the operands, due to a "wrap-around" effect when subtracting the parts from the smaller operand; this will ultimately lead to a cost similar to that of an $m \times m$ product. The "odd–even scheme" of Algorithm **OddEvenKaratsuba** (see also Exercise 1.13) avoids this wrap-around. Here is an example of this algorithm for $m = 3$ and $n = 2$. Take $A = a_2x^2 + a_1x + a_0$ and $B = b_1x + b_0$. This yields $A_0 = a_2x + a_0$, $A_1 = a_1$, $B_0 = b_0$, $B_1 = b_1$; thus, $C_0 = (a_2x + a_0)b_0$, $C_1 = (a_2x + a_0 + a_1)(b_0 + b_1)$, $C_2 = a_1b_1$.

Algorithm 1.5 OddEvenKaratsuba

Input: $A = \sum_0^{m-1} a_i x^i$, $B = \sum_0^{n-1} b_j x^j$, $m \ge n \ge 1$
Output: $A \cdot B$
  if $n = 1$ **then** return $\sum_0^{m-1} a_i b_0 x^i$
  write $A = A_0(x^2) + xA_1(x^2)$, $B = B_0(x^2) + xB_1(x^2)$
  $C_0 \leftarrow$ **OddEvenKaratsuba**(A_0, B_0)
  $C_1 \leftarrow$ **OddEvenKaratsuba**$(A_0 + A_1, B_0 + B_1)$
  $C_2 \leftarrow$ **OddEvenKaratsuba**(A_1, B_1)
  return $C_0(x^2) + x(C_1 - C_0 - C_2)(x^2) + x^2C_2(x^2)$.

We therefore get $K(3,2) = 2K(2,1) + K(1) = 5$ with the odd–even scheme. The general recurrence for the odd–even scheme is

$$K(m,n) = 2K(\lceil m/2 \rceil, \lceil n/2 \rceil) + K(\lfloor m/2 \rfloor, \lfloor n/2 \rfloor),$$

instead of

$$K(m,n) = 2K(\lceil m/2 \rceil, \lceil m/2 \rceil) + K(\lfloor m/2 \rfloor, n - \lceil m/2 \rceil)$$

for the classical variant, assuming $n > m/2$. We see that the second parameter in $K(\cdot,\cdot)$ only depends on the smaller size n for the odd–even scheme.

As for the classical variant, there are several ways of padding with the odd–even scheme. Consider $m = 5$, $n = 3$, and write $A := a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 = xA_1(x^2) + A_0(x^2)$, with $A_1(x) = a_3x + a_1$, $A_0(x) = a_4x^2 + a_2x + a_0$; and $B := b_2x^2 + b_1x + b_0 = xB_1(x^2) + B_0(x^2)$, with $B_1(x) = b_1$, $B_0(x) = b_2x + b_0$. Without padding, we write $AB = x^2(A_1B_1)(x^2) + x((A_0 + A_1)(B_0 + B_1) - A_1B_1 - A_0B_0)(x^2) + (A_0B_0)(x^2)$, which gives $K(5,3) = K(2,1) + 2K(3,2) = 12$. With padding, we consider $xB = xB_1'(x^2) + B_0'(x^2)$, with $B_1'(x) = b_2x + b_0$, $B_0' = b_1x$. This gives $K(2,2) = 3$ for A_1B_1', $K(3,2) = 5$ for $(A_0 + A_1)(B_0' + B_1')$, and $K(3,1) = 3$ for A_0B_0' – taking into account the fact that B_0' has only one non-zero coefficient – thus, a total of 11 only.

Note that when the variable x corresponds to say $\beta = 2^{64}$, Algorithm **OddEvenKaratsuba** as presented above is not very practical in the integer case, because of a problem with carries. For example, in the sum $A_0 + A_1$ we have $\lfloor m/2 \rfloor$ carries to store. A workaround is to consider x to be say β^{10}, in which case we have to store only one carry bit for ten words, instead of one carry bit per word.

The first strategy, which consists in cutting the operands into an equal number of pieces of unequal sizes, does not scale up nicely. Assume for example that we want to multiply a number of 999 words by another number of 699 words, using Toom–Cook 3-way. With the classical variant – without padding – and a "large" base of β^{333}, we cut the larger operand into three pieces of 333 words and the smaller one into two pieces of 333 words and one small piece of 33 words. This gives four full 333×333 products – ignoring carries – and one unbalanced 333×33 product (for the evaluation at $x = \infty$). The "odd–even" variant cuts the larger operand into three pieces of 333 words, and the smaller operand into three pieces of 233 words, giving rise to five equally unbalanced 333×233 products, again ignoring carries.