Computer Science & Engineering Department
IIT Kharagpur
Computational Number Theory: CS60094
Lecture I

Instructor: Goutam Biswas *Spring Semester 2014-2015*

1 Computation with Large Integers

1.1 Introduction

Most imperative programming languages support integers as a built-in type. But they are of fixed size, restricted by the machine architecture or the language specification. Typical size of a C language integer is 32-bit or 64-bit. The corresponding ranges of integers are -2^{31} to $2^{31}-1$ for a 32-bit representation, and -2^{63} to $2^{63}-1$ for a 64-bit representation[1]. But in number theoretic applications it is necessary to deal with integers of much larger size with full precision[2].

Languages like C provides support for *multi-precision integers* in the form of library e.g. *GNU Multiple Precision Arithmetic Library (GMP)*([GMP]). There are programming languages like *Python* that directly support arithmetic over multi-precision integers. Also there are softwares e.g. *PARI/gp* ([PARI/gp]) that supports programming in number theory and other algebraic application.

In this lecture we shall discuss basic data structure for multi-precision positive integers and arithmetic operations on them.

1.2 Representation of Multi-Precision Integer

A multi-precision integer is represented using positional number system, as a vector of digits in some suitable base B. The sign may be indicated by a *flag*. Let $a \in \mathbb{Z}$, we write

$$a = \pm \sum_{i=0}^{k-1} a_i B^i \equiv \pm(a_{k-1}a_{k-2}\cdots a_1 a_0),$$

Where $0 \leq a_i < B$, for $i = 0, 1, \cdots, k-1$. For uniqueness $a_{k-1} \neq 0$. It may be also necessary to store the number of digits. For most of our discussion we shall only consider *unsigned integers*.

The base B is often chosen to be a power of 2 to get the benefit of the binary world of a computer. As an example, we may choose the base $B = 2^{16} = 65536$. A digit of this radix can be stored in a C program as an element in an array of type `unsigned int` (size 32-bit). The range of digits with this base is 0 to $2^{16}-1$. The maximum value of the product of two digits, including a carry, can be $(B-1)^2 + (B-2) = B^2 - B - 1$. With $B = 65536$, the value is $4294901759 < 2^{32} - 1 = 4294967295$. In this scheme the space utilization is not good, as half of the 32-bit word is not used. The number of elementary digit operations are also more due to larger number of digits compared to base $B = 2^{32}$. This increases the constants of time complexity of different operations e.g. addition, multiplication etc.

[1] If we use unsigned integer the corresponding ranges are 0 to $2^{32}-1$ and 0 to $2^{64}-1$ respectively.
[2] Due to the loss of precision, floating point numbers are ruled out.

Example 1. Decimal number 123456789 in base 2^{16} is $(1883, 52501)_{2^{16}}$, where the digits of are coded in decimal. The place value of 1883 is $1883 \times 2^{16} = 123404288_{10}$.
Similarly $2^{100} = 1267650600228229401496703205376_{10}$ and its base-2^{16} representation is $(16, 0, 0, 0, 0, 0, 0)_{2^{16}}$.

We can do better if we have support for 64-bit (`unsigned long long int`) operations (available in GCC). We may choose $b = 2^{32}$, so there is no loss of bits. But then we have to use "`unsigned long long int`" operations which may turn out to be slower.
Example 2. The decimal number 123456789123456789 represented in base-2^{16} is $(438, 39755, 44240, 24341)$ and if represented in base-2^{32} is $(28744523, 2899336981)$. In terms of memory usage, base-2^{16} uses 16-bytes and base-2^{32} uses 8-bytes. Two add two such numbers, the first one requires four additions and the second one requires two.

We may also take the base as a power of 10. If $B = 10^4$ (10^9 is less than 2^{32}), every word can store 0 to 9999 and the product will not exceed $2^{32} - 1$. In this case the space utilisation is bad but the I/O will be simple. Cost of operation on every digit will be more.

A sample C data type may be as follows:

```
struct uli {
       unsigned int digits[DIG_LEN], sDigCount ;
};
typedef struct uli uli;   // unsigned large integer
uli n;
```

If we want a signed integer, we may use a *sign flag*, or we may store both the sign and the number of digits in the `sDigCount` field of the record[3]. If the sign is $s \in \{0(+), 1(-)\}$ and the number of digits are $n+1$, $d_n \cdots d_0$, we store $s \times 2^{31} + n + 1$ in `sDigCount`. So $0 < n+1 < 2^{31}$. The scheme can be used to accommodate $2^{31} - 1 = 2147483647$ digits of base 2^{16}.

In the digit array, the i^{th} location stores the i^{th} digit and its place value is `n.digits[i]`$*B^i$.
Example 3. The representation for $-$`123456789123456789` in our scheme is

index$\rightarrow$	3	2	1	0
`digits`	438	39755	44240	24341
`sDigCount`	$1 \times 2^{31} + 4 = 2147483649$			

We may also store a base-2^{31} number in a 32-bit word. In this case the sum of two digits will remain within 32-bit and the carry can be detected from b_{31}. But the result of multiplication will exceed 32-bits and is to be handled carefully.
Example 4. The base 2^{31} representation of $a = 123456789123456789$ is $(57489047, 751853333)$ and the representation of $b = 98765432109876543210$ is $(21, 894081649, 1697021674)$.
When we add them, $751853333 + 1697021674 = 2448875007$, the sum is greater than 2^{31} but less than 2^{32}. We get the digit of the sum as $2448875007 \mod 2^{31} = 301391359$ and the carry as $\lfloor \frac{2448875007}{2^{31}} \rfloor = 1$.

But in case of multiplication, $751853333 * 1697021674 = 1275911401770139442$ is larger than $2^{32} - 1$ and we have to use a double-word arithmetic.

1.3 Basic Operations on Multi-Precision Integers

We shall now consider how to input and output *multi-precision* integers and how to perform basic arithmetic operations on them.

[3]Or may be as the first element of the array itself.

- I/O functions - `readuli()`, `printuli()`,
- Arithmetic operations - `adduli()`, `subuli()`, `multuli()`, `divuli()` etc.
- Relational operations - `eq()`, `le()` etc.
- Other functions e.g. constructors - `makeuli()` etc.

It is essential to input data and print the result. So first of all we discuss the I/O functions. A long integer input comes as a long string of decimal digits, a stream of ASCII or similar code. It is necessary to convert it to internal representation (to a suitable base). So the *input:* "123456789123456789" is to be converted to $(438, 39755, 44240, 24341)_{2^{16}}$.

Let the input be a string of encoded decimal digits from the most significant side: $d_{n-1} \cdots d_i \cdots d_0$. Assume that we have already converted and stored the value of the digits $d_{n-1} \cdots d_i$ as a *base-B* numeral $D_{m-1} \cdots D_j$. Let the value of the partial input be N_i. We read the next decimal digit d_{i-1}. The value corresponding to $d_{n-1} \cdots d_i d_{i-1}$ is $N_{i-1} = 10N_i + d_{i-1}$ in *base-B*. So we multiply the representation of N_i in *base-B* by ten and add the *base-B* equivalent of the digit d_{i-1}. It essentially calls for a multiplication and addition functions in *base-B*.

$$N_i = \begin{cases} 0 & \text{if } i = m, \\ 10N_{i+1} + d_i & 0 \le i < n. \end{cases}$$

We may consume more than one decimal digits at a time. If we consume two digits at a time, we have $N_i = 100N_{i+2} + 10d_{i+1} + d_i$.

Example 5. Our input is "123456789123456789". Let the base be $2^{16} = 65536$. We start with (0).

Digit Consumed	Value
–	(0)
1	$10 \times (0) + 1 = (1)$
2	$10 \times (1) + 2 = (12)$
3	$10 \times (12) + 3 = (123)$
4	$10 \times (123) + 4 = (1234)$
5	$10 \times (1234) + 5 = (12345)$
6	$10 \times (12345) + 6 = (1, 57920)$
7	$10 \times (1, 57920) + 7 = (18, 54919)$
...	...

Now we consider how to print multi-precision integer. Let $N = D_{m-1} \cdots D_0$ be a m-digit *base-B* numeral we wish to print. Naturally we wish to print it as a decimal number. The essential idea is to convert N to a *base-10* or *power of 10* numeral.

Let us assume that we already have computed *base-10* numeral corresponding to $N_i = D_{m-1} \cdots D_i$. When the next digit D_{i-1} is included, the value of the number is $N_{i-1} = N_i \times B + D_{i-1}$. So we have to represent B and D_{i-1} in *base-10* and perform multiplication and addition. We start with $N_m = 0$ and compute N_0.

Example 6. Consider the base $2^{16} = 65536$ numeral $n = (438, 39755, 44240, 24341)$. We have

$$\begin{aligned} N_0 &= (0), \\ N_1 &= 0 \times 65536 + 438 = (4, 3, 8), \\ N_2 &= N_1 \times 65536 + 39755 = (2, 8, 7, 4, 4, 5, 2, 3), \end{aligned}$$

$$
\begin{aligned}
N_3 &= N_2 \times 65536 + 44240 = (1,8,8,3,8,0,1,1,0,3,5,6,8),\\
N_4 &= N_3 \times 65536 + 24341 = (1,2,3,4,5,6,7,8,9,1,2,3,4,5,6,7,8,9).
\end{aligned}
$$

In fact in the first step, 65536 is to be taken as $(6,5,5,3,6)$ and 438 is taken as $(4,3,8)$, and so on.

We can make it more efficient by taking the base for printing as a power of 10. It should be slightly smaller than the base of our multi-precision numeral. For example, if the base of the multi-precision numeral is $B = 2^{16}$, we may take our print numeral base to be 10^4. In this case we do the arithmetic in base 10^4. In this connection it is necessary to remember that each *base*-10^4 digit must have four decimal digits while printing.

Example 7. We start with the decimal number $n_{10} = 10200300040000500000$. The *base*-$2^{16}$ representation is $(36238, 48855, 50849, 61728)$. We choose our output base to be 10000. So the internal base is $(6, 5536)$ in the output base. We compute as follows:

$$
\begin{aligned}
N_0 &= (0),\\
N_1 &= N_0 \times (6,5536) + (3,6238) = (3,6238),\\
N_2 &= N_1 \times (6,5536) + (4,8855) = (23,7494,2423),\\
N_3 &= N_2 \times (6,5536) + (5,0849) = (155,6442,2668,4577),\\
N_4 &= N_3 \times (6,5536) + (6,1728) = (1020,0300,0400,0050,0000).
\end{aligned}
$$

During the computation we cannot ignore '00' of '0050' etc.

1.3.1 Addition and Multiplication

Algorithm for addition is simple. Consider base-B data $a = (a_{k-1}, a_{k-2}, \cdots, a_1, a_0)$ and $b = (b_{l-1}, b_{l-2}, \cdots, b_1, b_0)$. Without any loss of generality we can assume that $k \geq l \geq 1$ (if otherwise, we may exchange a and b). We take $b_{k-1}, \cdots, b_l \leftarrow 0$. The sum $c = a + b$, where $c = (c_k, c_{k-1}, c_{k-2}, \cdots, c_1, c_0)$. The value of c_k may be 0 or 1.

```
add(a,b)
cy ← 0
for i ← 0 to l − 1
      val ← a_i + b_i + cy
      c_i ← val mod B
      cy ← ⌊val/B⌋
for i ← l to k − 1
      val ← a_i + cy
      c_i ← val mod B
      cy ← ⌊val/B⌋
if cy = 1 then
      digits_C ← k + 1
      c_k ← 1
else digits_C ← k
return c
```

We assume that addition, mod , and division of two words can be performed in $O(1)$ time cost. So the running time of this algorithm is $O(k)$, where k is the word size of a multi-precision data.

Note that on a 32-bit machine if the base is $B = 2^{32}$, the value of $a_i + b_i + c_i$ may exceed B and will generate a *carry*, c_{i+1}. This can be detected by comparing the operands and the sum under different conditions of incoming carry.
Our job is to detect whether the carry out c_{i+1} is 0 or 1. We consider two cases, *carry-in* c_i is 0 or 1. We know that $0 \leq a_i, b_i < B$ and $a_i + b_i + c_i = B \times c_{i+1} + s_i$, where $c_{i+1} \in \{0, 1\}$ and $0 \leq s_i < B$.
If $c_i = 0$ and the carry out is 0, then $a_i + b_i = s_i$. So $s_i \geq a_i, b_i$. But if the carry out is 1, then $a_i + b_i = B + s_i$. So $s_i < a_i, b_i$ as $B > a_i, b_i$.
If $c_i = 1$ and the carry out is 0, then $a_i + b_i + 1 = s_i$. So $s_i > a_i, b_i$. If the carry out is 1, $a_i + b_i + 1 = B + s_i$ i.e. $a_i + b_i = (B - 1) + s_i$. So $s_i \leq a_i, b_i$. This gives an algorithm to detect carry. The advantage of base $B = 2^{32}$ is that the 'mod' operation can be performed at a lower cost.
Example 8. Let the base $B = 2^16$, following table shows different conditions:

c_i	a_i	b_i	s_i	condition	c_{i+1}
0	37940	40213	12617	$s_i < a_i, b_i$	1
0	37940	12111	50051	$s_i > a_i, b_i$	0
0	37940	0	37940	$s_1 = a_i, b_i = 0$	0
1	37940	12111	50052	$s_i > a_i, b_i$	0
1	37940	65535	37940	$s_i = a_i, b_i = 65535$	1
1	37940	40213	12618	$s_i < a_i, b_i$	1

Exercise: 1 Give an algorithm for performing subtraction.

Our next algorithm is for multiplication of two unsigned numbers. The inputs are $a = (a_{k-1}, a_{k-2}, \cdots, a_1, a_0)$ and $b = (b_{l-1}, b_{l-2}, \cdots, b_1, b_0)$, where $k, l \geq 1$. We take a as the multiplier. The output is $c = (c_{k+l-1}, c_{k+l-2}, \cdots, c_1, c_0)$. The time complexity is $O(kl)$.

```
mult(a,b)
for i ← 0 to k + l − 1
    c_i ← 0
for i ← 0 to k − 1
    cy ← 0
    for j ← 0 to l − 1
        val ← a_i × b_j + c_{i+j} + cy
        c_{i+j} ← val mod b
        cy ← ⌊val/b⌋
    c_{i+l} ← cy
if c_{k+l−1} > 0 then digits_C ← digits_A + digits_B
else digits_C ← digits_A + digits_B − 1
return c
```

Example 9. We start with $a = 9876556789 = (2, 19632, 19445)$ and $b = 1234567891234567 89_{10} = (438, 39755, 44240, 24341)_{65536}$. So the product can have at most 7-digits. We initialise

$c = (0, 0, 0, 0, 0, 0, 0)$. First we multiply $19445 \times (438, 39755, 44240, 24341)$:

$a_0 = 19445$						
j	b_j	cy_{in}	c_{i+j}	val	c_{i+j}	cy_{out}
0	24341	0	0	473310745	$c_0 = 9753$	7222
1	44240	7222	0	860254022	$c_1 = 28486$	13126
2	39755	13126	0	773049101	$c_2 = 51981$	11795
3	438	11795	0	8528705	$c_3 = 9025$	130

So we have the *partial product* $c = (0, 0, 130, 9025, 51981, 28486, 9753)$. Similarly,

$a_1 = 19632$						
j	b_j	cy_{in}	c_{i+j}	val	c_{i+j}	cy_{out}
0	24341	0	28486	477890998	$c_1 = 2486$	7292
1	44240	7292	51981	868578953	$c_2 = 30345$	13253
2	39755	13253	9025	780492438	$c_3 = 24214$	11909
3	438	11909	130	8610855	$c_4 = 25639$	131

The second *partial product* $c = (0, 131, 25639, 24214, 30345, 2486, 9753)$. Finally,

$a_1 = 2$						
j	b_j	cy_{in}	c_{i+j}	val	c_{i+j}	cy_{out}
0	24341	0	30345	79027	$c_2 = 13491$	1
1	44240	1	24214	112695	$c_3 = 47159$	1
2	39755	1	25639	105150	$c_4 = 39614$	1
3	438	1	131	1008	$c_5 = 1008$	0

The *final product* is $c = (0, 1008, 39614, 47159, 13491, 2486, 9753)$. This is equivalent to 1219327988765418508546090521 in decimal.

1.3.2 Karatsuba-Ofman Algorithm

This school-book algorithm mentioned earlier takes $O(n^2)$ basic operations (multiplication of two digits, addition with carry; division by constant etc). The question is whether anything better can be done. It seems Andrey Kolmogorov conjectured (1952) that this algorithm is asymptotically optimal. The conjecture was presented in a seminar at the Moscow State University in 1960. Within a week a 23-year old student Anatolii Alexeevitch Karatsuba found an algorithm that can multiply two n-digit numbers using $\Theta(n^{\log_2 3})$ single digit multiplications ([AKYO] [AD]).

We give an example of the algorithm with 2-digit and 4-digit numbers in radix-10.
Example 10. Let m, n be two 2-digit positive integers. So $m = 10a + b$ and $n = 10c + d$, where a, b, c, d are decimal digits. The product is

$$m \times n = (10a + b) \times (10c + d) = 100(a \cdot c) + 10(a \cdot d + b \cdot c) + b \cdot d.$$

It requires four multiplications of single-digit numbers[4], and three additions. Karatsuba's trick was to reduce the number of multiplication to three. He computed $a \cdot d + b \cdot c$ as

[4]Multiplication by constant can be converted to low cost shift operations.

$a \cdot c + b \cdot d - (b-a)(d-c)$. He computed the following products.

$$\begin{aligned} p &= a \cdot c, \\ q &= b \cdot d, \\ r &= (b-a)(d-c), \\ m \times n &= 100p + 10(p+q-r) + q \end{aligned}$$

In this case it is necessary to perform subtraction operation. So it is necessary to store signed numbers. If $m = 85$ and $n = 69$, then we have $p = 8 \times 6 = 48$, $q = 5 \times 9 = 45$, $r = (5-8)(9-6) = -9$. So the product is $100 \times 48 + 10 \times (48 + 45 - (-9)) + 45 = 5865$. The total number of operations are three single-digit multiplications, three single digit subtractions and three additions.

A 4-digit number e.g. 6592 can be written as $65 \times 10^2 + 92$. Let m and n be two 4-digit numbers such that $m = a \times 10^2 + b$ and $n = c \times 10^2 + d$, where a, b, c, d are each two-digit numbers. We again compute

$$\begin{aligned} p &= a \cdot c, \\ q &= b \cdot d, \\ r &= (b-a)(d-c), \\ m \times n &= 10^4 \cdot p + 10^2 \cdot (p+q-r) + q \end{aligned}$$

Example 11. Let $m = 5987 = 10^2 \times 59 + 87$ and $n = 7823 = 10^2 \times 78 + 23$. So $p = 59 \times 78 = 4602$, $q = 87 \times 23 = 2001$, $r = (87-59)(23-78) = -1540$. So the product is $10^4 \times 4602 + 10^2 \times (4602 + 2001 + 1540) + 2001 = 46836301$.

In this case we have used $3 \times 3 = 9$ single-digit multiplications to compute the product $m \times n$ (compute p, q and r). A school-book multiplication algorithm would have taken 16 single-digit multiplications. If the number of digits of the multiplier and the multiplicand are same and a power of 2, i.e. $n = 2^k$, then the *Karatsuba algorithm* will take three multiplications of 2^{k-1} digits. If we apply the algorithm recursively, we get 3^k single-digit multiplications. Whereas the school-book algorithm performs it with $n^2 = 2^{2k} = 4^k$ multiplications. Let $3^k = 3^{\log_2 n} = x$. So we have $\log_2 n \times \log_2 3 = \log_2 x$, implies that $\log_2 x = \log_2 n^{\log_2 3}$. So, $3^k = x = n^{\log_2 3} = n^{1.58}$ single digit multiplications.

It is necessary to take the addition and subtraction operations for exact analysis. It seems the school-book multiplication is faster than Karatsuba when the number of digits are small. But with a proper representation of multi-precision integers, Karatsuba is faster for larger numbers.

1.3.3 Toom-Cook Algorithm

This is a modification and generalisation of Karatsuba's method. First we review the Karatsuba-Ofman algorithm in a slightly different way. Let a and b two n digit numbers in base-B number system. We take $x = B^{\lceil n/2 \rceil}$ and express $a = a_1 x + a_0$ and $b = b_1 x + b_0$. We treat x as a formal variable, so we have the polynomials $a(x) = a_1 x + a_0$ and $b(x) = b_1 x + b_0$. The product $c(x) = a(x)b(x) = c_2 x^2 + c_1 x + c_0$, where $c_2 = a_1 b_1$, $c_1 = (a_1 b_0 + a_0 b_1)$, and $c_0 = a_0 b_0$.

The coefficients of $c(x)$, a quadratic polynomial, can be uniquely determined by evaluating it at three points. If we choose the points to be -1, 0 and ∞[5], we have $c(\infty) = a(\infty)b(\infty) =$

[5]If $p(x)$ is a polynomial of degree k, then by $p(\infty)$ we mean $\lim_{x \to \infty} \frac{p(x)}{x^k}$. This gives the coefficient of x^k.

a_1b_1, $c(0) = a(0)b(0) = a_0b_0$, and $c(-1) = a(-1)b(-1) = (a_0 - a_1)(b_0 - b_1)$. This gives us the coefficients of products in Karatsuba-Ofman algorithm.

Example 12. Let $a = 349$ and $b = 732$, $n = 3$, in decimal system. So $10^{\lceil \frac{n}{2} \rceil} = 10^2 = 100$. So $a(x) = 3x + 49$ and $b(x) = 7x + 32$ and $c(x) = (3 \times 7)x^2 + (3 \times 32 + 49 \times 7)x + 49 \times 32$. Now $c(\infty) = a(\infty)b(\infty) = 3 \times 7$, $c(0) = a(0)b(0) = 49 \times 32$, $c(-1) = a(-1)b(-1) = (49-3)(32-7) = 46 \times 25 = 1150$. So $10^4 \times (3 \times 7) + 10^2 \times (21 + 1568 - 1150) + 1568 = 210000 + 43900 + 1568 = 255468$.

Toom-Cook algorithm uses higher order polynomial to represent the numbers. We consider the special case of quadratic polynomial. The corresponding algorithm is known as Toom-3 algorithm.

Let a and b be two n digit base-B numbers. We take $x = B^{\lceil n/3 \rceil}$. Corresponding polynomials are

$$\begin{aligned} a(x) &= a_2x^2 + a_1x + a_0, \\ b(x) &= b_2x^2 + b_1x + b_0; \end{aligned}$$

and the product $a(x)b(x) = c(x) = c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0$, where the coefficients are

$$\begin{aligned} c_4 &= a_2b_2, \\ c_3 &= a_2b_1 + a_1b_2, \\ c_2 &= a_2b_0 + a_0b_2 + a_1b_1, \\ c_1 &= a_1b_0 + a_0b_1, \\ c_0 &= a_0b_0. \end{aligned}$$

A direct computation of five coefficients require 9 multiplications (in radix-$xB^{\lceil n/3 \rceil}$) which is same as the number of multiplications i by the school book method.

But we can improve this with a suitable choice of five evaluation points. Our choices are $-2, -1, 0, 1, \infty$. And we have

$$\begin{array}{rcll} c(\infty) &=& c_4 & = a_2b_2, \\ c(1) &=& c_4 + c_3 + c_2 + c_1 + c_0 & = (a_2 + a_1 + a_0)(b_2 + b_1 + b_0), \\ c(-1) &=& c_4 - c_3 + c_2 - c_1 + c_0 & = (a_2 - a_1 + a_0)(b_2 - b_1 + b_0), \\ c(-2) &=& 16c_4 - 8c_3 + 4c_2 - 2c_1 + c_0 & = (4a_2 - 2a_1 + a_0)(4b_2 - 2b_1 + b_0), \\ c(0) &=& c_0 & = a_0b_0. \end{array}$$

In this case we use five multiplications. Using the matrix notation we get

$$\begin{pmatrix} c(\infty) \\ c(1) \\ c(-1) \\ c(-2) \\ c(0) \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 \\ 16 & -8 & 4 & -2 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} c_4 \\ c_3 \\ c_2 \\ c_1 \\ c_0 \end{pmatrix}$$

The coefficient matrix is fixed and has inverse. So we have

$$\begin{pmatrix} c_4 \\ c_3 \\ c_2 \\ c_1 \\ c_0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 2 & 1/6 & 1/2 & -1/6 & -1/2 \\ -1 & 1/2 & 1/2 & 0 & -1 \\ -2 & 1/3 & -1 & 1/6 & 1/2 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} c(\infty) \\ c(1) \\ c(-1) \\ c(-2) \\ c(0) \end{pmatrix}.$$

So we have the following solution for the coefficients.

$$\begin{aligned}
c_4 &= c(\infty),\\
c_3 &= \frac{1}{6}(12c(\infty) + c(1) + 3c(-1) - c(-2) - 3c(0)),\\
c_2 &= \frac{1}{2}(-2c(\infty) + c(1) + c(-1) - 2c(0),\\
c_1 &= \frac{1}{6}(-12c(\infty) + 2c(1) - 6c(-1) + c(-2) + 3c(0),\\
c_0 &= c(0).
\end{aligned}$$

The time complexity for multiplying $\lceil n/3 \rceil$ digit integers by small constants like 2, 3, 12 is of $O(n)$. The main contribution to time complexity comes from five multiplications of $\lceil n/3 \rceil$ digit integers.

These $\lceil n/3 \rceil$ multiplications can be done recursively in a similar manner until we come to single digits. To simplify our analysis of time complexity we assume that n, the number of digits, is 3^k. So in the school-book method we need $n^2 = 9^k$ single digit multiplication. But in Toom-3 we require $5^k = n^{\log_3 5} = n^{1.465}$ multiplications.

Example 13. Let $a = 123456789$ and $b = 87654321$ with the base $B = 10^3$. So the actual data is $a = (12, 345, 678)$ and $b = (87, 654, 321)$. As the number of digits $n = 3$, $x = B^{3/3} = B = 10^3$. We write $a(x) = 12x^2 + 345x + 678$ and $b(x) = 87x^2 + 654x + 321$ i.e. $a_2 = 12, a_1 = 345, a_0 = 678$ and $b_2 = 87, b_1 = 654, b_0 = 321$. The product is $c(x) = c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c - 0$. The values of

$$\begin{aligned}
c(\infty) &= a_2b_2 = 12 \times 87 = (1, 044),\\
c(1) &= (a_2 + a_1 + a_0)(b_2 + b_1 + b_0) = (12 + 345 + 678)(87 + 654 + 321) = (1, 099, 170),\\
c(-1) &= (a_2 - a_1 + a_0)((b_2 - b_1 + b_0) = (12 - 345 + 678)(87 - 654 + 321) = -(84, 870),\\
c(-2) &= (4a_2 - 2a_1 + a_0)(4b_2 - 2b_1 + b_0) = (4 \times 12 - 2 \times 345 + 678)(4 \times 87 - 2 \times 654 + 321)\\
&= -(23, 004),\\
c(0) &= a - 0b_0 = 678 \times 321 = (217, 638).
\end{aligned}$$

Now we find the values of the coefficients of $c(x)$.

$$\begin{aligned}
c_4 &= c(\infty) = (1, 044),\\
c_3 &= \frac{1}{6}[12c(\infty) + c(1) + 3c(-1) - c(-2) - 3c(0)]\\
&= \frac{1}{6}[12 \times (1, 044) + (1, 099, 170) + 3(84, 870) + (23, 004) - 3 \times (217, 638)]\\
&= \frac{1}{6} \times (227, 178) = (37, 863),\\
c_2 &= \frac{1}{2}[-2c(\infty) + c(1) + c(-1) - 2c(0)]\\
&= \frac{1}{2}[-2 \times (1, 044) + (1, 099, 170) - (84, 870) - 2 \times (217, 638)\\
&= \frac{1}{2} \times (576, 936) = (288, 468),\\
c_1 &= \frac{1}{6}[-12c(\infty) + 2c(1) - 6c(-1) + c(-2) + 3c(0)]
\end{aligned}$$

$$= \frac{1}{6}[-12 \times (1,044) + 2 \times (1,099,170) + 6 \times (84,870) - (23,004) + 3 \times (217,638)]$$
$$= \frac{1}{6} \times (3,324,942) = ((554,157)],$$
$$c_0 = c(0) = (217,638)$$

So the product is

c_4x^4	1	044					+
$+ c_3x^3$		037	863				+
$+ c_2x^2$			288	468			+
$+ c_1x$				554	157		+
$+ c_0$					217	638	=
$= c$	1	082	152	022	347	638	

1.3.4 Schönhage-Strassen Algorithm

The best known multiplication algorithm (asymptotically) is based on *Fast Fourier Transform (FFT)* algorithm of *Discrete Fourier Transform (DFT)*. Here the multiplication is viewed as convolution. We shall present a simplified version of the algorithm [AD]. The running time of the algorithm is $O(n \log n \log\log n)$.

We start with the definition of DFT over complex field. Let our *signal* be a finite sequence $a = (a_i)$ of length n, where $i = 0, 1, \cdots, n-1$[6]. An element a_i may be viewed as a coefficient of the polynomial $a(x) = a_{n-1}x^{n-1} + \cdots + a_1x + a_0$.
In general a_i's are complex numbers. Let ω_n be the *primitive* n^{th} root of unity i.e. $\omega_n = e^{j\frac{2\pi}{n}}$. The *Discrete Fourier Transform (DFT)* of a, $DFT(a)$, is the signal $A = (A_l)$, $l = 0, 1, \cdots, n-1$, where

$$A_l = \sum_{k=0}^{n-1} a_k \omega_n^{kl}.$$

Essentially A_l is $a(\omega_n^l)$.
The signal A may also be viewed as a polynomial $A(x) = A_0 + A_1x + \cdots + A_{n-1}x^{n-1}$. The *Inverse Discrete Fourier Transform* of A, $DFT^{-1}(A)$ gives a. The $DFT^{-1}(A)$ is defined as,

$$a_l = \frac{1}{n} \sum_{k=0}^{n-1} A_k \omega_n^{-kl},\ l = 0, 1, \cdots, n-1.$$

So $a_l = \frac{A(\omega_n^{-l})}{n}$, and we claim that $a = DFT^{-1}(DFT(a))$. The time complexity of this method of DFT computation is $\Theta(n^2)$. Subsequently we shall show a better method for this computation. It is known as *Fast Fourier Transform (FFT)* algorithm.
Example 14. Let $a(x) = x^3 + 2x^2 + 3x + 4 : (1, 2, 3, 4)$. So $a_0 = 1, a_1 = 2, a_2 = 3, a_3 = 4$. The primitive 4^{th}-root of 1 is $\omega_4 = e^{i\frac{2\pi}{4}} = \cos\frac{\pi}{2} + i\sin\frac{\pi}{2} = i = \sqrt{-1}$.
So $\omega_4^0 = 1, \omega_4^1 = \omega_4 = i, \omega_4^2 = -1$ and $\omega_4^3 = -i$. The components of the $DFT(a)$ vector is

[6] We shall not bother much about the notion of "signal". These may be a sequence of digitized audio signal and in communication engineering people are interested about correlation between the signal and its sinusoidal frequency components.

(A_3, A_2, A_1, A_0), where

$$\begin{array}{rcrcrcr} A_0 & = & a(\omega_4^0) & = & 1+2+3+4 & = & 10 \\ A_1 & = & a(\omega_4^1) & = & i^3+2i^2+3i+4 & = & 2+2i \\ A_2 & = & a(\omega_4^2) & = & (-1)^3+2(-1)^2+3(-1)+4 & = & 2 \\ A_3 & = & a(\omega_4^1) & = & (-i)^3+2(-i)^2+3(-i)+4 & = & 2-2i \end{array}$$

We use the polynomial $A(x) : (2-2i)x^3 + 2x^2 + (2+2i)x + 10$ to evaluate $DFT^{-1}(A)$. We have $(\omega_4)^{-1} = \frac{1}{i} = -i$.

$$\begin{array}{rcrcrcr} a_0 & = & \frac{1}{4}A((-i)^0) & = & \frac{1}{4}A(1) & = & \frac{1}{4}[(2-2i)+2+(2+2i)+10] & = & 4 \\ a_1 & = & \frac{1}{4}A((-i)^1) & = & \frac{1}{4}A(-i) & = & \frac{1}{4}[(2-2i)(-i)^3+2(-i)^2+(2+2i)(-i)+10] & = & 3 \\ a_2 & = & \frac{1}{4}A((-i)^2) & = & \frac{1}{4}A(-1) & = & \frac{1}{4}[(2-2i)(-1)^3+2(-1)^2+(2+2i)(-1)+10] & = & 2 \\ a_3 & = & \frac{1}{4}A((-i)^3) & = & \frac{1}{4}A(i) & = & \frac{1}{4}[(2-2i)(i)^3+2(i)^2+(2+2i)(i)+10] & = & 1 \end{array}$$

It is important that we mathematically justify our claim. Let the polynomial $a(x) = a_0 + a_1x + \cdots, a_{n-1}x^{n-1}$. We evaluate it at n different points $x_0, x_1, \cdots, x_{n-1}$ and the corresponding values are $y_0 = a(x_0), \cdots, y_{n-1} = a(x_{n-1})$. This can be expressed as a $n \times n$ matrix equation.

$$\begin{pmatrix} 1 & x_0 & \cdots & x_0^{n-1} \\ 1 & x_1 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_{n-1} & \cdots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

The coefficient matrix is known as *Vandermonde* matrix $V(x_0, x_1, \cdots, x_{n-1})$. It is known to be invertible if $x_i \neq x_j$. The determinant is $\prod_{0 \leq i < j \leq n-1}(x_j - x_i)$. In DFT the *Vandermonde* matrix is $V_n(1, \omega_n, \omega_n^2, \cdots, \omega_n^{n-1})$, where ω_n is the primitive n^{th} roots of unity.

$$\begin{pmatrix} 1 & \omega_n^0 & \cdots & (\omega_n^0)^{n-1} \\ 1 & \omega_n & \cdots & (\omega_n)^{n-1} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & \omega_n^{n-1} & \cdots & (\omega_n^{n-1})^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

V_n is invertible, so $a = V_n^{-1}y$.

Proposition 1. $V_n^{-1}[k,l] = \frac{\omega_n^{-lk}}{n}$, $k, l = 0, 1, \cdots, n-1$.

We prove the following lemma used in the proof of the proposition.

Lemma 2. Let n, k be positive integers and $n \not| k$.

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = 0.$$

Proof:

$$\begin{array}{rl} & \sum_{j=0}^{n-1} (\omega_n^k)^j \\ = & (\omega_n^k)^0 + (\omega_n^k)^1 + \cdots + (\omega_n^k)^{n-1} \end{array}$$

$$
\begin{aligned}
&= \frac{(\omega_n^k)^n - 1}{\omega_n^k - 1} \\
&= \frac{(\omega_n^n)^k - 1}{\omega_n^k - 1} \\
&= \frac{1^k - 1}{\omega_n^k - 1} \\
&= 0.
\end{aligned}
$$

If $n|k$, then $\omega_n^k = 1$, and the result does not hold. □

Proof: (*proposition*) We assume $V_n^{-1}[k,l] = \frac{\omega_n^{-lk}}{n}$, $k, l = 0, 1, \cdots, n-1$, and prove that $V_n^{-1}V_n = I_n$. We know that $(V_n^{-1}V_n)[p,q]$, the $(p,q)^{th}$ entry of $V_n^{-1}V_n$, is the inner product of the p^{th}-row of V_n^{-1} i.e. $[1/n\ (\omega_n^{-p})^1/n\ \cdots\ (\omega_n^{-p})^{n-1}/n]$, and the q^{th}-column of V_n i.e. $[1\ (\omega_n^q)^1\ \cdots\ (\omega_n^q)^{n-1}]^T$. So,

$$
(V_n^{-1}V_n)[p,q] = \frac{\omega_n^{0\cdot(q-p)}}{n} + \frac{\omega_n^{1\cdot(q-p)}}{n} + \cdots + \frac{\omega_n^{(n-1)\cdot(q-p)}}{n}.
$$

This sum is 1 if $p = q$, otherwise it is 0 by the previous lemma. Note that $q - p$ is not divisible by n as $0 \le p, q \le n-1$. So $V_n^{-1}V_n = I_n$. □

Our next job is to show how multiplication can be performed using DFT.
Let a and b be the multiplier and the multiplicand. Each of them has n digits in some base B. We take $N = 2n$ and pad both a and b with leading zeros (digits $a_{N-1}, b_{N-1}, \cdots, a_n, b_n$). Now we have

$$
\begin{aligned}
a: &\quad a_{N-1}B^{N-1} + \cdots + a_1 B + a_0 \\
b: &\quad b_{N-1}B^{N-1} + \cdots + b_1 B + b_0,
\end{aligned}
$$

where $a_i = 0 = b_i$, $\frac{N}{2} \le i < N$. The reason for this zero-extension is that the product of two n digit numbers can have $2n$ digits.
The *cyclic convolution* of two *signals* $a : (a_{N-1}, \cdots, a_1, a_0)$ and $b : (b_{N-1}, \cdots, b_1, b_0)$ is $c : (c_{N-1}, \cdots, c_1, c_0)$, where

$$
\begin{aligned}
c_k &= \sum_{(i+j)\equiv k(\bmod N)} a_i b_j,\ k = \{0, 1, \cdots, N-1\}, \\
&= (a_0 b_k + a_1 b_{k-1} + \cdots + a_k b_0) + (a_{k+1} b_{N-1} + a_{k+2} b_{N-2} + \cdots + a_{N-1} b_{k+1}), \\
&= (a_0 b_k + a_1 b_{k-1} + \cdots + a_k b_0), \\
&= \sum_{i+j=k} a_i b_j,\ k = \{0, 1, \cdots, N-1\}.
\end{aligned}
$$

The contribution of $a_{k+1}b_{N-1} + a_{k+2}b_{N-2} + \cdots + a_{N-1}b_{k+1}$ is zero as $i + j = k + N$ and $a_i = 0 = b_i$, $\frac{N}{2} \le i < N$.
So the product

$$
a \times b = \sum_{i=0}^{N-1} c_i B^i.
$$

We compute $DFT(a)$ and $DFT(b)$.

$$
\begin{aligned}
DFT(a_{N-1}, \cdots, a_1, a_0) &\Rightarrow (A_{N-1}, \cdots, A_1, A_0), \\
DFT(b_{N-1}, \cdots, b_1, b_0) &\Rightarrow (B_{N-1}, \cdots, B_1, B_0),
\end{aligned}
$$

where A_k is $a(x)$ evaluated at ω_N^k, B_k is $b(x)$ evaluated at ω_N^k. Let $DFT(c_{N-1}, \cdots, c_1, c_0) \Rightarrow (C_{N-}, \cdots, C_1, C_0)$, where C_k is $c(x)$ evaluated at ω_N^k. So we have

$$C_k = c(\omega_N^k) = a(\omega_N^k) \cdot b(\omega_N^k) = A_k B_k.$$

The product $c = (c_{N-1}, \cdots, c_1, c_0)$ is obtained by $\text{DFT}^{-1}(C_{N-1}, \cdots, C_1, C_0)$.
Example 15. Let $a = 123$ and $b = 456$ (decimal numbers). The corresponding signals are $a : (0, 0, 0, 1, 2, 3)$ and $b : (0, 0, 0, 4, 5, 6)$. The primitive 6^{th} root of unity is

$$\omega_6 = e^{i\frac{2\pi}{6}} = \cos\frac{\pi}{3} + i\sin\frac{\pi}{3} = \frac{1+\sqrt{3}i}{2}.$$

Different powers of ω_6 are

ω_6^0	ω_6^1	ω_6^2	ω_6^3	ω_6^4	ω_6^5
1	$\frac{1+\sqrt{3}i}{2}$	$\frac{-1+\sqrt{3}i}{2}$	-1	$\frac{-1-\sqrt{3}i}{2}$	$\frac{1-\sqrt{3}i}{2}$
$(\omega_6^{-1})^0$	$(\omega_6^{-1})^1$	$(\omega_6^{-1})^2$	$(\omega_6^{-1})^3$	$(\omega_6^{-1})^4$	$(\omega_6^{-1})^5$
1	$\frac{1-\sqrt{3}i}{2}$	$\frac{-1-\sqrt{3}i}{2}$	-1	$\frac{-1+\sqrt{3}i}{2}$	$\frac{1+\sqrt{3}i}{2}$

The $DFT(a) = A = (A_0, A_1, A_2, A_3, A_4, A_5)$, where $A_k = a(\omega_6^k)$, $k = 0, \cdots, 5$.

A_0	$=$	$a(\omega_6^0)$	$=$	$1+2+3$	$=$	6
A_1	$=$	$a(\omega_6^1)$	$=$	$\left(\frac{1+\sqrt{3}i}{2}\right)^2 + 2\left(\frac{1+\sqrt{3}i}{2}\right) + 3$	$=$	$\left(\frac{7+3\sqrt{3}i}{2}\right)$
A_2	$=$	$a(\omega_6^2)$	$=$	$\left(\frac{-1+\sqrt{3}i}{2}\right)^2 + 2\left(\frac{-1+\sqrt{3}i}{2}\right) + 3$	$=$	$\left(\frac{3+\sqrt{3}i}{2}\right)$
A_3	$=$	$a(\omega_6^3)$	$=$	$1-2+3$	$=$	2
A_4	$=$	$a(\omega_6^4)$	$=$	$\left(\frac{-1-\sqrt{3}i}{2}\right)^2 + 2\left(\frac{-1-\sqrt{3}i}{2}\right) + 3$	$=$	$\left(\frac{3-\sqrt{3}i}{2}\right)$
A_5	$=$	$a(\omega_6^5)$	$=$	$\left(\frac{1-\sqrt{3}i}{2}\right)^2 + 2\left(\frac{1-\sqrt{3}i}{2}\right) + 3$	$=$	$\left(\frac{7-3\sqrt{3}i}{2}\right)$

Similarly we have

$$DFT(b) = B = (B_0, B_1, B_2, B_3, B_4, B_5) = \left(15, \frac{13+9\sqrt{3}i}{2}, \frac{3+3\sqrt{3}i}{2}, 5, \frac{3-3\sqrt{3}i}{2}, \frac{13-9\sqrt{3}i}{2}\right).$$

Let $c = a \times b$. The

$$\begin{aligned}
& DFT(c) \\
= \; & C \\
= \; & DFT(a) \cdot DFT(b) \\
= \; & (A_5 \times B_5, A_4 \times B_4, A_3 \times B_3, A_2 \times B_2, A_1 \times B_1, A_0 \times B_0) \\
= \; & \left(\frac{5-51\sqrt{3}i}{2}, \frac{3-3\sqrt{3}i}{2}, 10, \frac{3+3\sqrt{3}i}{2}, \frac{5+51\sqrt{3}i}{2}, 90\right).
\end{aligned}$$

So,

$$C(x) = \frac{5-51\sqrt{3}i}{2}x^5 + \frac{3-3\sqrt{3}i}{2}x^4 + 10x^3 + \frac{3+3\sqrt{3}i}{2}x^2 + \frac{5+51\sqrt{3}i}{2}x + 90.$$

Finally, $c = DFT^{-1}(C)$, where $c_k = \frac{1}{6}C((\omega_6^{-1})^k)$.

c_0	=	$\frac{1}{6}C((\omega_6^{-1})^0)$	=	18
c_1	=	$\frac{1}{6}C((\omega_6^{-1})^1)$	=	27
c_2	=	$\frac{1}{6}C((\omega_6^{-1})^2)$	=	28
c_3	=	$\frac{1}{6}C((\omega_6^{-1})^3)$	=	13
c_4	=	$\frac{1}{6}C((\omega_6^{-1})^4)$	=	4
c_5	=	$\frac{1}{6}C((\omega_6^{-1})^5)$	=	0

So

$$a \times b = 123 \times 456 = 18 + 270 + 2800 + 13000 + 40000 = 56088.$$

The *Fast Fourier Transform (FFT)* algorithm uses *divide-and-conquer* strategy, takes the advantage of the properties of complex roots of 1, and computes DFT in $\Theta(n \log n)$ time. Let $a(x) = a_0 + a_1x + \cdots + a_{n-1}x^{n-1}$ be a polynomial where n is a power of 2. We express it as $a(x) = a^{(e)}(x^2) + xa^{(o)}(x^2)$, where

$$a^{(e)}(x) = a_0 + a_2x + \cdots + a_{n-2}x^{\frac{n}{2}-1},\ a^{(o)}(x) = a_1 + a_3x + \cdots + a_{n-1}x^{\frac{n}{2}-1}.$$

The evaluation of $a(x)$ at $\omega_n^0, \omega_n^1, \cdots, \omega_n^{n-1}$ is equivalent to the evaluation of $a^{(o)}(x)$ and $a^{(e)}(x)$ at $(\omega_n^0)^2, (\omega_n^1)^2, \cdots, (\omega_n^{n-1})^2$.
Example 16. Let $a(x) = 4+3x+2x^2+x^3$ (1.3.4), so we have $a^{(o)}(x) = 3+x$ and $a^{(e)}(x) = 4+2x$. The components of $DFT(a)$ using $a^{(o)}(x)$ and $a^{(e)}(x)$ are, are,

A_0	=	$a(\omega_4^0)$	=	$a^{(e)}((\omega_4^0)^2) + \omega_4^0 a^{(o)}((\omega_4^0)^2)$	=	$(4+2) + 1 \cdot (3+1)$	=	10
A_1	=	$a(\omega_4^1)$	=	$a^{(e)}((\omega_4^1)^2) + \omega_4^1 a^{(o)}((\omega_4^1)^2)$	=	$(4+2(-1)) + i(3+(-1))$	=	$2+2i$
A_2	=	$a(\omega_4^2)$	=	$a^{(e)}((\omega_4^2)^2) + \omega_4^2 a^{(o)}((\omega_4^2)^2)$	=	$(4+2) + (-1)(3+1)$	=	2
A_3	=	$a(\omega_4^3)$	=	$a^{(e)}((\omega_4^3)^2) + \omega_4^3 a^{(o)}((\omega_4^3)^2)$	=	$(4-2) + (-i)(3-1)$	=	$2-2i$

Lemma 3. (*Cancellation lemma*) Let n, k, and d be integers such that $n, k \geq 0$, $d > 0$. The $(dk)^{th}$ power of the $(dn)^{th}$ root of 1 is same as the k^{th} power of the n^{th} root of 1.
Proof:

$$\begin{aligned} \omega_{dn}^{dk} &= (e^{i\frac{2\pi}{dn}})^{dk} \\ &= (e^{i\frac{2\pi}{n}})^k \\ &= \omega_n^k. \end{aligned}$$

□

Lemma 4. (*Halving Lemma*) If the positive integer n is even, then the square of n complex n^{th}-roots of 1 are the $\frac{n}{2}$ complex $\frac{n}{2}^{th}$-roots of 1.
Proof: We have $(\omega_n^{k+\frac{n}{2}})^2 = \omega_n^{2k}\omega_n^n = (\omega_n^k)^2$, where $k = 0, 1, \cdots, \frac{n}{2} - 1$. Again by the cancellation lemma, $(\omega_n^k)^2 = \omega_{\frac{n}{2}}^k$, where $k = 0, 1, \cdots, \frac{n}{2} - 1$. So, $(\omega_n^{k+\frac{n}{2}})^2 = (\omega_n^k)^2 = \omega_{\frac{n}{2}}^k$, when n is even and $k = 0, 1, \cdots, \frac{n}{2} - 1$. □

We know that the evaluation of $a(x)$ at $\omega_n^0, \omega_n^1, \cdots, \omega_n^{n-1}$ is equivalent to the evaluation of $a^{(e)}(x^2)+xa^{(o)}(x^2)$ at $(\omega_n^0)^2, (\omega_n^1)^2, \cdots, (\omega_n^{n-1})^2$. But then the sequence $(\omega_n^0)^2, (\omega_n^1)^2, \cdots, (\omega_n^{n-1})^2$ is the sequence of $\frac{n}{2}$ complex $\frac{n}{2}^{th}$-roots of 1 repeated twice.

We also have $\omega_n^{k+\frac{n}{2}} = \omega_n^k \cdot \omega_n^{\frac{n}{2}} = -\omega_n^k$, where $k = 0, 1, \cdots, \frac{n}{2}$. So finally we have $A_k = a^{(e)}(\omega_{\frac{n}{2}}^k) + \omega_n^k a^{(o)}(\omega_{\frac{n}{2}}^k) = DFT(a^{(e)}) + \omega_n^k DFT(a^{(o)})$ and $A_{k+\frac{n}{2}} = a^{(e)}(\omega_{\frac{n}{2}}^k) - \omega_n^k a^{(o)}(\omega_{\frac{n}{2}}^k) = DFT(a^{(e)}) - \omega_n^k DFT(a^{(o)})$, $k = 0, 1, \cdots, \frac{n}{2} - 1$.
Example 17. We have

$$\begin{array}{rclcrcl} a^{(e)}((\omega_4^k)^2) & = & a^{(e)}(\omega_2^k), & & a^{(o)}((\omega_4^k)^2) & = & a^{(o)}(\omega_2^k), \\ a^{(e)}((\omega_4^{k+2})^2) & = & a^{(e)}(\omega_2^k), & & a^{(o)}((\omega_4^{k+2})^2) & = & a^{(o)}(\omega_2^k), \end{array}$$

for $k = 0, 1$.
Again $\omega_4^{k+2} = \omega_4^k \cdot \omega_4^2 = -\omega_4^k$, for $k = 0, 1$. So,

$$\begin{array}{rclcrcl} A_0 & = & a^{(e)}(\omega_2^0) + \omega_4^0 a^{(o)}(\omega_2^k) & = & 10 \\ A_1 & = & a^{(e)}(\omega_2^1) + \omega_4^1 a^{(o)}(\omega_2^1) & = & 2 + 2i \\ A_2 & = & a^{(e)}(\omega_2^0) - \omega_4^0 a^{(o)}(\omega_2^0) & = & 2 \\ A_3 & = & a^{(e)}(\omega_2^1) - \omega_4^1 a^{(o)}(\omega_2^1) & = & 2 - 2i. \end{array}$$

Following is the recursive FFT algorithm [CLRS]. We take the length or the number of coefficients of the polynomial n to be power of 2.

```
FFT(a)
    n ← length(a)
    if n = 1 return a
    ω_n ← e^{i 2π/n}
    ω ← 1
    a^(e) ← (a_0, a_2, ··· , a_{n-2})
    a^(o) ← (a_1, a_3, ··· , a_{n-1})
    A^(e) ← FFT(a^(e))
    A^(o) ← FFT(a^(o))
    for k ← 0 to n/2 − 1
        A_k ← A_k^(e) + ω A_k^(o)
        A_{k+n/2} ← A_k^(e) − ω A_k^(o)
        ω ← ω ω_n
    return A
```

This FFT algorithm evaluates DFT in $\Theta(n \log n)$ time. If the number of digits of a and b are n, so that $2^{t-1} < n \leq 2^t$, then for the purpose of multiplication we start with $N = 2^{t+1}$ digits with proper padding of the most significant digits by zeros. The power of 2 is for the ease of implementation of FFT. The multiplication by 2 is for the product.
Example 18. Let $a = (12, 34)$ in base $B = 100$ i.e in decimal $a = 1234$. We compute $DFT(a)$ and also by FFT algorithm. We extend a to 4-digits (for multiplication) by putting zeros in the two most significant digits. So, $a = (0, 0, 12, 34)$ and $a(x) = 12x + 34$.
Let $DFT(a) = A = (A_3, A_2, A_1, A_0)$ i.e. $A(x) = A_3x^3 + A_2x^2 + A_1x + A_0$, where $\omega_4 = i$ and

$$\begin{array}{rclcrcl} A_0 & = & a(\omega_4^0) & = & 12 + 34 & = & 46 \\ A_1 & = & a(\omega_4^1) & = & 34 + 12i & & \\ A_2 & = & a(\omega_4^2) & = & 34 - 12 & = & 22 \\ A_3 & = & a(\omega_4^3) & = & 34 - 12i & & \end{array}$$

Now we use FFT algorithm to do the same calculation. After the first recursive call we have,

$$\begin{array}{rclcrclcrcl}
A_0 & = & a(\omega_4^0) & = & a^{(e)}((\omega_4^0)^2) + \omega_4^0 a^{(o)}((\omega_4^0)^2) & = & a^{(e)}(\omega_2^0) + \omega_4^0 a^{(o)}(\omega_2^0) & = & A_0^{(e)} + \omega_4^0 A_0^{(o)} \\
A_1 & = & a(\omega_4^1) & = & a^{(e)}((\omega_4^1)^2) + \omega_4^1 a^{(o)}((\omega_4^1)^2) & = & a^{(e)}(\omega_2^1) + \omega_4^1 a^{(o)}(\omega_2^1) & = & A_1^{(e)} + \omega_4^1 A_1^{(o)} \\
A_2 & = & a(\omega_4^2) & = & a^{(e)}((\omega_4^2)^2) - \omega_4^2 a^{(o)}((\omega_4^2)^2) & = & a^{(e)}(\omega_2^2) + \omega_4^0 a^{(o)}(\omega_2^2) & = & A_0^{(e)} - \omega_4^0 A_0^{(o)} \\
A_3 & = & a(\omega_4^3) & = & a^{(e)}((\omega_4^3)^2) + \omega_4^3 a^{(o)}((\omega_4^3)^2) & = & a^{(e)}(\omega_2^1) - \omega_4^1 a^{(o)}(\omega_2^1) & = & A_1^{(e)} - \omega_4^1 A_1^{(o)}
\end{array}$$

where $a^{(e)} = (0, 34)$ i.e. $a^{(e)}(x) = 34$ and $a^{(o)} = (0, 12)$ i.e $a^{(o)}(x) = 12$. So we have $a^{(e)(e)} = (34)$, $a^{(e)(o)} = (0)$ and $a^{(o)(e)} = (12)$, $a^{(o)(o)} = (0)$.
These are base case and can be computed directly

$$\begin{array}{rclcrclcl}
A_0^{(e)} & = & a^{(e)}(\omega_2^0) & = & a^{(e)(e)}(\omega_1^0) + \omega_2^0 a^{(e)(o)}(\omega_1^0) & = & 34 \\
A_1^{(e)} & = & a^{(e)}(\omega_2^1) & = & a^{(e)(e)}(\omega_1^1) - \omega_2^0 a^{(e)(o)}(\omega_1^1) & = & 34 \\
A_0^{(o)} & = & a^{(o)}(\omega_2^0) & = & a^{(o)(e)}(\omega_1^0) + \omega_2^0 a^{(o)(o)}(\omega_1^0) & = & 12 \\
A_1^{(o)} & = & a^{(o)}(\omega_2^1) & = & a^{(o)(e)}(\omega_1^1) - \omega_2^0 a^{(o)(o)}(\omega_1^1) & = & 12
\end{array}$$

Finally we get the values of

$$\begin{array}{rclcrcl}
A_0 & = & A_0^{(e)} + \omega_4^0 A_0^{(o)} & = & 34 + 12 & = & 46 \\
A_1 & = & A_1^{(e)} + \omega_4^1 A_1^{(o)} & = & 34 + 12i & & \\
A_2 & = & A_0^{(e)} - \omega_4^0 A_0^{(o)} & = & 34 - 12 & = & 22 \\
A_3 & = & A_1^{(e)} - \omega_4^1 A_1^{(o)} & = & 34 - 12i & &
\end{array}$$

So we have $DFT(a) = FFT(a) = A = (34 - 12i, 22, 34 + 12i, 46)$.
If we take another number $b = (0, 0, 56, 78)$, then in the similar way we get $DFT(b) = FFT(b) = B = (78 - 56i, 22, 78 + 56i, 134)$. If we now take the inner product of A and B we get
$A \cdot B = C = (1980 - 2840i, 484, 1980 + 2840i, 6164)$.
Now we run the FFT algorithm with ω_n^{-1}. We know that $c = (c_3, c_2, c_1, c_0) = DFT^{-1}(C)$. A direct computation shows that $c = (0, 672, 2840, 2652)$ in base 100. This gives us a product in decimal as $a \times b = 6720000 + 284000 + 2652 = 7006652$. Now we apply FFT algorithm on C with $\omega_4^{-1} = -i$ and $\omega_2^{-1} = -1$.

Call 1: $FFT(C) = FFT(1980 - 2840i, 484, 1980 + 2840i, 6164)$, where
$n = 4$, $\omega_4^{-1} = e^{-i\frac{2\pi}{4}} = \cos\pi/2 - i\sin\pi/2 = -i$, $\omega = 1$, $C^{(e)} = (484, 6164)$, $C^{(o)} = (1980 - 2840i, 1980 + 2840i)$.

Call 1.1: $FFT(C^{(e)}) = FFT((484, 6164))$,
`returns` $(6164 - 484, 6164 + 484) = (5680, 6648)$.

Call 1.2: $FFT(C^{(o)}) = FFT((1980 - 2840i, 1980 + 2840i))$,
`returns` $(1980 + 2840i - 1980 + 2840i, 1980 + 2840i + 1980 - 2840i) = (5680i, 3960)$.

Loop for $k = 0$, $\omega = 1$, $d_0 = 6648 + 3960 = 10608$, $d_2 = 6648 - 3960 = 2688$.
Loop for $k = 1$, $\omega = -i$, $d_1 = 5680 + (-i)5680i = 11360$, $d_3 = 5680 - (-i)5680i = 0$.
`returns` $(0, 2688, 11360, 10608)$.

Finally $c = (c_3, c_2, c_1, c_0) = \frac{1}{4}(d_3, d_2, d_1, d_0) = \frac{1}{4}(0, 2688, 11360, 10608) = (0, 672, 2840, 2652)$.
In *base*-100, this is equivalent to $6720000 + 284000 + 2652 = 7006652 = 1234 \times 5678$.
Now we look at recursive calls:

Call 1.1: $FFT(C^{(e)}) = FFT((484, 6164))$, where
$n = 2$, $\omega_2^{-1} = e^{-i\frac{2\pi}{2}} = \cos\pi - i\sin\pi = -1$, $\omega = 1$, $C^{(e)(e)} = (6164)$, $C^{(e)(o)} = (484)$.
Call 1.1.1: $FFT(C^{(e)(e)}) = FFT(6164)$,
`returns` 6164, as it is a base case.
Call 1.1.2: $FFT(C^{(e)(o)}) = FFT(484)$,
`returns` 484, as it is a base case.
Loop for $k = 0$, $\omega = 1$, $e_0 = 6164 + 484 = 6648$, $e_1 = 6164 - 484 = 5680$.
`returns` $(5680, 6648)$.
Call 1.2: $FFT(C^{(o)}) = FFT(1980 - 2840i, 1980 + 2840i)$ similarly returns $(5680i, 3960)$.

1.3.5 Division

The integer division algorithm that generates quotient and remainder is more involved than other operations. We have the *dividend* $a = (a_{k-1}, a_{k-2}, \cdots, a_1, a_0)$ and the *divisor* $a = (b_{l-1}, b_{l-2}, \cdots, b_1, b_0)$ $(b_{l-1} \neq 0)$ in a base-B number system. We wish to find out the quotient q and the remainder r so that $a = bq + r$ where $0 \leq r < b$. We further assume that $k \geq l$, otherwise, $a < b$ implies that the quotient is zero and the remainder is a. The smallest value of b with l digits in base B is B^{l-1} and the largest value of a with k digits is $B^k - 1$. So the number of digits of the quotient q can be at most $k - (l - 1) = k - l + 1 = m$ digits. Let the quotient bits be $q = (q_{m-1}, q_{m-2}, \cdots, q_1, q_0)$. Following example shows the basic division algorithm.
Example 19. Let the radix $B = 10$, $a = 3596$ and $b = 38$. So $k = 4$ and $l = 2$. The maximum number of quotient digits can be $m = 4 - 2 + 1 = 3$. Let r be the partial remainder, and to start with r is a. We perform the following computation for steps $m - 1$ to 0.

i	r_{in}	$q_i \leftarrow \lfloor r_{in}/(10^i \cdot b) \rfloor$	$r_{out} \leftarrow r_{in} - 10^i \times q_i \times b$
2	3596	$3596/(10^2 \times 38) = 0$	$3596 - 10^2 \times 0 \times 38 = 3596$
1	3596	$3596/(10^1 \times 38) = 9$	$3596 - 10^1 \times 9 \times 38 = 176$
0	176	$176/(10^0 \times 38) = 4$	$176 - 10^0 \times 4 \times 38 = 24$

The quotient is 094 and the remainder is 24.

In the algorithmic form it is as follows:

```
div(a,b)
r ← a
for i ← m − 1 downto 0
    q_i ← ⌊r/(B^i · b)⌋
    r ← r − (B^i · b) · q_i
```

After the initialisation of r,

$$0 \leq r = a < B^k = (B^{k-l+1} \cdot B^{l-1}) \leq (B^{k-l+1} \cdot b) = (B^m \cdot b).$$

We observe the following loop-invariance at the beginning of each iteration, $0 \leq r < B^{i+1} \cdot b$. So, when r is divided by $B^i \cdot b$, the value of the quotient is in the range of 0 to $B - 1$, a single digit.

The main problem is the efficient computation of $\lfloor r_i/(B^i \cdot b) \rfloor$. In a pencil-paper calculation we '*try*' different values of q_i, multiply it with b, subtract it from r_i etc.. But how can we restrict the number of trials? We prove the following proposition.

Proposition 5. Let x and y be non-negative integers so that

$$x = x_0 \cdot B^n + s, \text{ and } 0 < y = y_0 \cdot B^n,$$

such that $n > 0$ and $0 \leq s < B^n$, then $\lfloor \frac{x}{y} \rfloor = \lfloor \frac{x_0}{y_0} \rfloor$.
Proof: We have

$$\frac{x}{y} = \frac{x_0 \cdot B^n + s}{y_0 \cdot B^n} = \frac{x_0}{y_0} + \frac{s}{y_0 \cdot B^n} \geq \frac{x_0}{y_0}.$$

So $\lfloor \frac{x}{y} \rfloor \geq \lfloor \frac{x_0}{y_0} \rfloor$. Again,

$$\frac{x}{y} = \frac{x_0 \cdot B^n + s}{y_0 \cdot B^n} < \frac{x_0}{y_0} + \frac{1}{y_0}, \text{ as } s < B^n.$$

Let $x_0 = ky_0 + z_0$, where $0 \leq z_0 < y_0$ and $k = \lfloor \frac{x_0}{y_0} \rfloor$. So we have

$$\frac{x_0}{y_0} + \frac{1}{y_0} = \frac{ky_0 + z_0}{y_0} + \frac{1}{y_0} = k + \frac{z_0 + 1}{y_0} \leq k + 1 = \lfloor \frac{x_0}{y_0} \rfloor + 1.$$

So, $\lfloor \frac{x}{y} \rfloor < \lfloor \frac{x_0}{y_0} \rfloor + 1$, hence the proof. □

This shows that we can have simple algorithm when the divisor is single digit. Let the dividend be $a = (a_{k-1}, a_{k-2}, \cdots, a_1, a_0)$ and a single digit divisor be $b = b_0$. Following the previous proposition the k digit quotient $q = (q_{k-1}, q_{k-2}, \cdots, q_1, q_0)$ can be computed as follows:

div(a,b_0)
$r \leftarrow 0$
`for` $i \leftarrow k - 1$ `downto` 0
    $t \leftarrow r \cdot B + a_i$
    $q_i \leftarrow \lfloor t/b_0 \rfloor$
    $r \leftarrow t \bmod b_0$

The *quotient* is $q = q_{k-1} \cdots q_0$ and the remainder is r.

Where $0 \leq r < b_0 < B$, and $0 \leq t = r \cdot B + a_i < (B-1)B + (B-1) = (B^2 - 1)$. So t is of size at most two digits.
Example 20. Let $B = 100$, $a = 123456789_{10} = (1, 23, 45, 67, 89)_{100}$ and $b = 67_{10} = (67)_{100}$. Number of digits of the divisor is $k = 5$.

r_{in}	i	a_i	t	q	r_{out}
0	4	1			
0	4	1	1	0	1
1	3	23			
1	3	23	123	1	56
56	2	45			
56	2	45	5645	84	17
17	1	67	1767		
17	1	67	1767	26	25
25	0	89			
25	0	89	2589	38	43

So the quotient is $(1, 84, 26, 38)_{100} = 1842638_{10}$ and the remainder is 43.
Example 21. Let the dividend be $a = 123456789123456789_{10}$ and the divisor be $b = 50129_{10} = 50129_{2^{16}}$. So $a = (438, 39755, 44240, 24341)_{2^{16}}$. The quotient $q = 2462781805411 \equiv (573, 26940, 4963)_{2^{16}}$ and the remainder $r = 8770$. Here $k = 4$, $b = 2^{16} = 65536$.

r_{in}	i	a_i	t	q	r_{out}
0	3	438			
438	3	438	438	0	438
438	2	39755			
438	2	39755	28744523	573	20606
20606	1	44240	1350479056		
20606	1	44240	1350479056	26940	3796
3796	0	24341			
3796	0	24341	248798997	4963	8770

Division is simple when the divisor is single digit i.e. $l = 1$. Now we try to *estimate the quotient digit* when the number of digits are one or more.
Proposition 6. Let x and y be non-negative integers such that

$$x = x_0 \cdot B^n + s, \text{ and } 0 < y = y_0 \cdot B^n + t$$

where $n > 0$, and $0 \leq s, t < B^n$. We further suppose that $2B > 2 \cdot y_0 \geq B$ i.e the dividend is *normalised*, then

$$\left\lfloor \frac{x}{y} \right\rfloor \leq \left\lfloor \frac{x_0}{y_0} \right\rfloor \leq \left\lfloor \frac{x}{y} \right\rfloor + 2.$$

Proof: As $y \geq y_0 \cdot B^n$, so

$$\frac{x}{y} \leq \frac{x}{y_0 \cdot B^n} = \frac{x_0 \cdot B^n + s}{y_0 \cdot B^n} = \frac{x_0}{y_0} + \frac{s}{y_0 \cdot B^n} \leq \frac{x_0}{y_0}.$$

So, $\left\lfloor \frac{x}{y} \right\rfloor \leq \left\lfloor \frac{x_0}{y_0} \right\rfloor$.

We have $\frac{x}{y} = \frac{x_0 \cdot B^n + s}{y_0 \cdot B^n + t} = \frac{x_0 + s/B^n}{y_0 + t/B^n} \geq \frac{x_0}{y_0 + 1}$. Both s/B^n and t/B^n are less than 1, and we reduce the numerator and increase the denominator. This gives us $x_0 y - x y_0 - x \leq 0$.
According to our assumption $2 \cdot y_0 \geq B$. Intutively it is clear that $\frac{x}{y} < B$ ($\frac{x}{y} = \frac{x_0 B^n + s}{y_0 B^n + t} < \frac{(B-1)B^n + (B^n - 1)}{B^n} < B$). This implies that $2y_0 \geq x/y$ i.e. $2yy_0 - x \geq 0$. So we have $x_0 y - x y_0 - x \leq 0 \leq 2yy_0 - x$. Dividing both sides by yy_0 we get $\frac{x_0}{y_0} \leq \frac{x}{y} + 2$. So, the second inequality. □

If we take $b_{l-1} \geq B/2$ (normalised number), then the difference between the actual quotient $\lfloor x/y \rfloor$ and the 'guess' from $\lfloor x_0/y_0 \rfloor$ may differ by at most 2. So we need at most two iterations for each quotient digit.

Following this proposition we have the restoring division algorithm that works well with a normalised divisor.

We assume that a digit-location can accommodate values in the range $-B+1$ to $B-1$. So it is possible to have a negative digit.

div(a,b)
1. `for` $i \leftarrow 0$ `to` $k-1$
2. $\quad r_i \leftarrow a_i$
3. $r_k \leftarrow 0$
4. `for` $i \leftarrow k-l$ `downto` 0
5. $\quad t \leftarrow \lfloor \frac{r_{i+l} \cdot B + r_{i+l-1}}{b_{l-1}} \rfloor$
6. $\quad$ `if` $t \geq B$ `then` $q_i \leftarrow B-1$
7. $\quad$ `else` $q_i \leftarrow t$
8. $\quad cy \leftarrow 0$
9. $\quad$ `for` $j \leftarrow 0$ `to` $l-1$
10. $\quad\quad t \leftarrow r_{i+j}$ - $q_i b_j + cy$
11. $\quad\quad r_{i+j} \leftarrow t \bmod B$
12. $\quad\quad cy \leftarrow \lfloor t/B \rfloor$
13. $\quad r_{i+l} \leftarrow r_{i+l} + cy$
14. $\quad$ `while` $r_{i+l} < 0$
15. $\quad\quad cy \leftarrow 0$
16. $\quad\quad$ `for` $j \leftarrow 0$ `to` $l-1$
17. $\quad\quad\quad t \leftarrow r_{i+j} + b_j + cy$
18. $\quad\quad\quad r_{i+j} \leftarrow t \bmod B$
19. $\quad\quad\quad cy \leftarrow \lfloor t/B \rfloor$
20. $\quad\quad r_{i+l} \leftarrow r_{i+l} + cy$
21. $\quad\quad q_i \leftarrow q_i - 1$

1. Line 1-3: Initializes the *remainder*, r, with the *dividend*.

2. Line 4-7: Estimates the i^{th} quotient digit q_i. It is greater than or equal to the actual quotient by at most 2. It cannot exceed $B-1$ and we immediately adjust it.

3. Line 8-13: We compute

$$(r_{i+l} \cdots r_i) \leftarrow (r_{i+l} \cdots r_i) - q_i \cdot B.$$

   The value of carry in line 8-12 is $-(b-1)$ to 0, and the value of 't' is $-q_i b_i = -b(b-1)$ to $r_{i+j} = b-1$.

4. Line 14-21: If the partial remainder is negative, then it is restored to its previous value, and the quotient digit is reduced by 1.

The theorem on normalised data ensures that the correction loop is not executed more than twice. The running time is not difficult to compute. The outer loop is executed $k-l+1 = m = \log q$ times and both the inner loops are executed $l = \log b$ times each. So the running time is $O(\log q \cdot \log b)$.

If the *divisor is not normalised*, we may multiply both the *dividend* and the *divisor* by 2^w and get a_s and b_s. Multiplication by a power of 2 is a left shift operation in a computer. Now we have $a_s = b_s q + r_s$, where $q = \lfloor a/b \rfloor = \lfloor a_s/b_s \rfloor$. But the remainder is multiplied by 2^w, $r_s = r2^w$. So r_s is to be right shifted to get the actual remainder.

Left and right shift operations required for normalisation and adjustment of the remainder takes $O(\log a + \log b)$ time. So the running time complexity remains as $O(\log q \cdot \log b)$.

Example 22. Let the base $B = 100$, the *dividend* $a = (1, 20, 05, 67, 89)$ and the *divisor* $b = (60, 98)$. The number of digits of the *dividend* is $k = 5$ and that of the *divisor* is $l = 2$. So the maximum possible digits in *quotient* is $m = k - l + 1 = 4$. We initialise the *partial remainder* $r = (0, 1, 20, 05, 67, 89)$. Quotient digits are computed starting from the most significant side. We start with,

- $i \leftarrow k - l = 5 - 2 = 3$:
  $t \leftarrow \lfloor \frac{r_5 \times b + r_4}{b_1} \rfloor = \lfloor \frac{0 \times 100 + 1}{60} \rfloor = 0.$
  $q_3 \leftarrow 0.$
  $cy \leftarrow 0.$

  The tentative value of $q_3 = 0$. As it is the lowest possible value of the quotient digit, the remaining portion of the code does not have any effect. We compute the next digit.

  - $j \leftarrow 0$: No effect
  - $j \leftarrow 1$: No effect

- $i \leftarrow 2$: $t \leftarrow \lfloor \frac{r_4 \times b + r_3}{b_1} \rfloor = \lfloor \frac{1 \times 100 + 20}{60} \rfloor = 2.$
  $q_2 \leftarrow 2.$
  $cy \leftarrow 0.$

  The tentative value of $q_2 = 2$. We verify it by doing the following computation.

  - $j \leftarrow 0$:
    $t \leftarrow r_{i+j} - b_j q_i + cy = r_2 - b_0 q_2 + cy = 05 - 98 \times 2 + 0 = -191.$
    $r_2 \leftarrow -191 \mod 100 = 09.$
    $cy \leftarrow \lfloor \frac{-191}{100} \rfloor = -2.$
  - $j \leftarrow 1$:
    $t \leftarrow r_{i+j} - b_j q_i + cy = r_3 - b_1 q_2 + cy = 20 - 60 \times 2 - 2 = -102.$
    $r_2 \leftarrow -102 \mod 100 = 98.$
    $cy \leftarrow \lfloor \frac{-102}{100} \rfloor = -2.$

  The value of $r_{i+l} = r_4 \leftarrow r_4 + cy = 1 - 2 = -1$. As it is negative, the remainder is to be restored.

  $$r = (0, -1, 98, 09, 67, 89)$$

  $cy \leftarrow 0.$

  - $j \leftarrow 0$:
    $t \leftarrow r_{i+j} + b_j + cy = r_2 + b_0 + cy = 09 + 98 + 0 = 107.$
    $r_{i+j} \leftarrow t \mod B = 107 \mod 100 = 07.$ $cy \leftarrow \lfloor \frac{t}{B} \rfloor = \lfloor 107/100 \rfloor = 1.$
  - $j \leftarrow 1$:
    $t \leftarrow r_{i+j} + b_j + cy = r_3 + b_1 + cy = 98 + 60 + 1 = 159.$
    $r_{i+j} \leftarrow t \mod B = 159 \mod 100 = 59.$
    $cy \leftarrow \lfloor \frac{t}{B} \rfloor = \lfloor 159/100 \rfloor = 1.$

New value of $r_{i+l} = r_4 \leftarrow r_4 + cy = -1 + 1 = 0$. The quotient bit $q_i = q_3$ is modified, $q_3 \leftarrow q_3 - 1 = 2 - 1 = 1$. After this stage we have $r = (0, 0, 59, 07, 67, 89)$ and $q = (0, 1, -, -)$.

- $i \leftarrow 1$:
- $i \leftarrow 0$:

Example 23. Let the base of the system be $b = 2^8 = 256$, the dividend is $2550731057_{10} \equiv (152, 9, 17, 49)_b$, the divisor is $38650_{10} \equiv (150, 250)_b$. So $k = 4, l = 2$. We initialize $r = (0, 152, 9, 17, 49)_b$.

- $i = k - l = 4 - 2 = 2$:
  $q_2 = \lfloor \frac{r_4 \times b + r_3}{b_1} \rfloor = \lfloor \frac{0 \times 256 + 152}{150} \rfloor = 1$,
  $cy = 0$,
  - $j = 0$:
    $t = r_2 - q_2 \times b_0 + cy = 9 - 1 \times 250 + 0 = -241$
    $cy = -1$
    $r_2 = 15$
    $r = (0, 152, 15, 17, 49)_b$.
  - $j = 1$:
    $t = r_3 - q_2 \times b_1 + cy = 152 - 1 \times 150 - 1 = 1$
    $cy = 0$
    $r_2 = 1$
    $r = (0, 1, 15, 17, 49)_b$.

  $r_4 = r_4 + cy = 0$.
- $i = 1$:
  $q_1 = \lfloor \frac{r_3 \times b + r_2}{b_1} \rfloor = \lfloor \frac{1 \times 256 + 15}{150} \rfloor = 1$,
  $cy = 0$,
  - $j = 0$:
    $t = r_1 - q_1 \times b_0 + cy = 17 - 1 \times 250 + 0 = -233$
    $cy = -1$
    $r_1 = 23$
    $r = (0, 1, 15, 23, 49)_b$.
  - $j = 1$:
    $t = r_2 - q_1 \times b_1 + cy = 15 - 1 \times 150 - 1 = -136$
    $cy = -1$
    $r_2 = 120$
    $r = (0, 1, 120, 23, 49)_b$.

  $r_3 = r_3 + cy = 1 - 1 = 0$. $r = (0, 0, 120, 23, 49)_b$.
- $i = 0$:
  $q_0 = \lfloor \frac{r_2 \times b + r_1}{b_1} \rfloor = \lfloor \frac{120 \times 256 + 23}{150} \rfloor = 204$,
  $cy = 0$,

- $j = 0$:
  $t = r_0 - q_0 \times b_0 + cy = 49 - 204 \times 250 + 0 = -50951$
  $cy = -200$
  $r_0 = 249$
  $r = (0, 1, 15, 23, 249)_b$.
- $j = 1$:
  $t = r_1 - q_0 \times b_1 + cy = 23 - 204 \times 150 - 200 = 30777$
  $cy = -121$
  $r_1 = 199$
  $r = (0, 1, 120, 199, 249)_b$.

$r_2 = r_2 + cy = 120 - 121 = -1$.
$r = (0, 0, -1, 23, 249)_b$.
As the digit r_2 is -ve, we enter the `while` loop.
$cy = 0$

- $j = 0$:
  $t = r_0 + b_0 + cy = 249 + 250 + 0 = 499$
  $r_0 = 243$
  $cy = 1$
  $r = (0, 0, -1, 199, 243)_b$.
- $j = 1$:
  $t = r_1 + b_1 + cy = 199 + 150 + 1 = 350$
  $r_0 = 94$
  $cy = 1$
  $r = (0, 0, -1, 94, 243)_b$.

$r_2 = r_2 + cy = -1 + 1 = 0$.
$q_0 = q_0 - 1 = 204 - 1 = 203$.
$r = (0, 0, 0, 94, 243)_b$. $q = (1, 1, 203)_b$.

1.4 Software

GNU has a multiple precision arithmetic library (GMP: `http://gmplib.org/`). It is necessary to install it on your machine. Following is a small sample program that computes n factorial.

```
// file name gmp1.c: $ cc -Wall gmp1.c -lgmp
#include <stdio.h>
#include <gmp.h>                          // header

int main()
{
    mpz_t fact, im ;                      // multiple precision integer
    int i, n;

    mpz_init(fact);                       // initialization
    mpz_init(im);
    printf("Enter a positive integer: ");
```

```
    scanf("%d", &n);
    mpz_set_si(fact, 1);                // setting a value
    for(i=1; i<=n; ++i) {
        mpz_set_si(im,i);               //
        mpz_mul(fact, fact, im);        // fact <-- fact + im
    }
    gmp_printf("%d! = %Zd\n", n, fact);  // print
    return 0;
}
```

The code is compiled (provided the library is installed) with the following command:
`$ cc -Wall gmp1.c -lgmp`.
The library has its own manual.

The programming language *Python* supports multi-precision integers as its data type and we can write normal program.

```
#! /usr/bin/python
# fact2.py calculates factorial of a number
#
n = input("Enter a +ve integer: ")
i, fact = 1, 1
while i <= n:
     fact = fact*i
     i = i + 1
print n, "!=", fact
```

Another important software is PARI/GP available from `http://pari.math.u-bordeaux.fr/`. It is usable in different forms:

1. A library `libpari` that can be linked with a C program.
2. A programmable calculator `gp`.
3. A `gp2c` compiler that translates a `gp` code to C code.

The first example is a use of `libpari` -

```
#include <stdio.h>
#include <pari/pari.h>
int main()
{
   GEN n, i, fact;

   pari_init(1000000,2);
   printf("Enter a non-ve integer: ") ;
   n = gp_read_stream(stdin);
   fact = gen_1;
   for(i=gen_1; gsigne(gsub(n,i))>=0; i=gadd(i,gen_1))
       fact = gmul(fact,i) ;
```

```
   pari_printf("%Ps! = %Ps\n", n, fact) ;
   pari_close();
   return 0 ;
} // cc -Wall pariFact1.c -lpari
```

The program can be compiled with the following command provided the library is in place:
`$ cc -Wall pariFact1.c -lpari`.

GP has a built-in operator (also a function) for computing factorial.

```
gp > 100!
%1 = 933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000000
```

A simple GP script to compute $n!$ is as follows:

```
gp > fact(n) = if(n, n*fact(n-1),1)
%5 = (n)->if(n,n*fact(n-1),1)
(15:37) gp > fact(0)
%6 = 1
(15:38) gp > fact(5)
%7 = 120
(15:38) gp > fact(100)
%8 = 933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000000
```

References

[AD] *Computational Number Theory* by Abhijit Das, Pub. CRC Press, 2013, ISBN 978-1-4398-6615-3.

[AKYO] A. Karatsuba & Yu. Ofman, Multiplication of Multidigit Numbers on Automata (in Russian), *Doklady Akad. Nauk SSSR* 145 (1962), pp. 293-294, English translation in *Soviet Physics Doklady* 7 (1963), pp. 595-596.

[DEK] *The Art of Computer Programming, volume 2, Seminumerical Algorithms*, by Donald E. Knuth, (3rd. ed.), Addison-Wesley, 1999, ISBN 981-235-883-8.

[GMP] `http://gmplib.org/`

[PARI/gp] `http://pari.math.u-bordeaux.fr/`

[VS] *A Computational Introduction to Number Theory and Algebra* by Victor Shoup, 2nd ed., Pub. Cambridge University Press, 2009, ISBN 978-0-521-51644-0.

[CLRS] *Introduction to Algorithms*, by T H Cormen, C E Leiserson, R L Rivest, C Stein, (2nd ed). Prentice-Hall of India, 2002, 81-203-2141-3.