CHAPTER 1 | Arithmetic with Large Integers

The goal of this and the next lecture is to explain how computers and other hardware platforms perform basic operations with large integers in the context of public key cryptographic protocols. On a hardware platform, non-negative integers will be represented in base B where B is typically a power of 2 (e.g., B could be 2^{32} for a 32-bit architecture). Each integer n is then represented by

$$n = \sum_{i=0}^{s-1} n_i B^i, \qquad 0 \leq n_i \leq B-1.$$

Here, the n_i's are blocks, the least significant one being n_0 and the most significant one being n_{s-1}.

1.1 Complexity of algorithms

Often, we will need to give mathematical estimates for the complexity of various algorithms. We do that via a special notation called big-$\mathcal{O}$ notation that we now introduce.

1.1.1 Big-$\mathcal{O}$-notation

Suppose that $f, g\colon \mathbf{Z}_{>0} \to \mathbf{R}$ are two functions of the positive integers n that take real values. We say that $g(n) = \mathcal{O}(f(n))$ if there exist constants $C, N > 0$ such that $|g(n)| \leq C|f(n)|$ for all $n \geq N$. Moreover, we say that $g(n) = \Theta(f(n))$ if $g(n) = \mathcal{O}(f(n))$ and $f(n) = \mathcal{O}(g(n))$. Very often, we will give the run-time of an algorithm via this notation. We will write $g(n) = o(f(n))$ if $f(n) \neq 0$ for all large enough n and $g(n)/f(n)$ tends to 0 as $n \to +\infty$. Finally, we write $f(n) \sim g(n)$ if $f(n) \neq 0$ for all large enough n and $g(n)/f(n)$ tends to 1 as $n \to +\infty$.

As an example, $2n^2 + n = \mathcal{O}(n^2)$ since $2n^2 + n \leq 3n^2$ for all $n \geq 1$. Moreover, $2n^2 + n = \Theta(n^2)$, but $n \neq \Theta(n^2)$ since $n^2 \neq \mathcal{O}(n)$ even if $n = \mathcal{O}(n^2)$. As we see examples of algorithms, we will get more and more used to this notation.

1.1.2 Remarks on big-$\mathcal{O}$ notation to keep in mind

Measuring run-time of various algorithms is a subtle question. The big-$\mathcal{O}$ notation is one kind of asymptotic notation, an idea from mathematics that describes the behavior of functions *in the limit*.

In practice, defining how long an algorithm takes to run is difficult: one cannot usually give an answer in milliseconds because of the dependency on the machine architecture. One can neither give an answer in CPU clock cycles (or as an operation count) because that would be too specific to the particular data to be useful.

The simple way of looking at asymptotic notation is that it discards all the constant factors in a function. In other words, an^2 will always be bigger that bn if n is sufficiently large (assuming everything is positive). Changing the constant factors a and b does not change that - it changes the specific value of n where n^2 is bigger, but does not change that it happens. So we say that $\mathcal{O}(n^2)$ is asymptotically larger than $\mathcal{O}(n)$, and forget about those constants that we probably do not know anyway. That is useful because the problems with large n are usually the ones where things slow down enough that we really care. If n is small enough, the time taken is small and the gains available from choosing different algorithms are small. When n gets large, choosing a different algorithm can thus make a huge difference.

The big-$\mathcal{O}$ notation is a useful mathematical model that abstracts away enough awkward-to-handle details that useful results can be found, but it is certainly not a perfect measure for the complexity of algorithms. We do not deal with infinite problems in practice and there are plenty of times when problems are small enough that those constants are relevant for real-world performance and sometimes you just have to time things with a clock rather than asymptotically.

1.2 Schoolbook, Karatsuba and Toom–Cook multiplication

1.2.1 Schoolbook multiplication

Let us recall the basic multiplication of integers that we learned in school: to compute $123 \cdot 323$, we do

$$\begin{array}{cccccc} & & & 3 & 6 & 9 \\ & & 2 & 4 & 6 & \\ 3 & 6 & 9 & & & \\ \hline 3 & 9 & 7 & 2 & 9 & \end{array}$$

Simple as that, we will try to analyze the run-time and express it in big-$\mathcal{O}$ notation. Suppose first that the two numbers have n digits each. To compute each of the rows, we need n single digit multiplications, a total of n^2 single digit multiplications. To compute the final result out of these, we need to add n integers of at most $2n$ digits each. This gives us a total of $3n^2$ elementary operations, i.e., the run-time of the algorithm is $\mathcal{O}(n^2)$. Note that in this estimate, we are, for example, not accounting for the time it takes to shift an integer to the left. Yet, in computers, bit shifts are

typically considered very fast operations taking almost negligible amount of time.

1.2.2 Karatsuba multiplication

Let x and y be two base-b numbers of n digits each (in their b-base representation). For simplicity, assume that n is even, i.e., $n = 2m$. Write $x = x_0 + b^m x_1$ and $y = y_0 + b^m y_1$. Then

$$xy = x_0 y_0 + (x_0 y_1 + x_1 y_0) b^m + x_1 y_1 b^{2m}.$$

To compute xy, it thus suffices to perform the four multiplications $x_0 y_0$, $x_1 y_0$, $x_0 y_1$ and $x_1 y_1$. Karatsuba [**?**] observed that one only needs to do three multiplications: $(x_0 + x_1)(y_0 + y_1)$, $x_0 y_0$ and $x_1 y_1$, and perform four extra additions in order to get the above product, since

$$x_0 y_1 + x_1 y_0 = (x_0 + x_1)(y_0 + y_1) - x_0 y_0 - x_1 y_1.$$

Applying this algorithm recursively, one computes the product xy in $\mathcal{O}(n^{\log_2 3})$ elementary steps (you will see in the homework exercise why this is the run time). This is asymptotically faster than the schoolbook algorithm (in fact, $\log_2 3 \sim 1,585 < 2$).

1.2.3 Toom–Cook multiplication

Toom–Cook's method is based on the observation that, given two polynomials of degree d (for some d),

$$p(x) = p_0 + p_1 x + \cdots + p_{d-1} x^{d-1}$$

and

$$q(x) = q_0 + q_1 x + \cdots + q_{d-1} x^{d-1},$$

the product polynomial $h(x) = p(x)q(x) = h_0 + h_1 x + \cdots + h_{2d-2} x^{2d-2}$ is completely determined by the values at $2d-1$ distinct points, e.g., at $t = -d+1, -n+2, \ldots, 0, 1, \ldots, d-1$. If we now think of each polynomial as the base-b expansion of an n-bit number, we can turn this idea into an algorithm for multiplying integers.

We illustrate this with an example: consider $d = 3$ and write the symbolic expressions

$$\begin{aligned}
r_{-2} &= (p_0 - 2p_1 + 4p_2)(q_0 - 2q_1 + 4q_2) \\
r_{-1} &= (p_0 - p_1 + p_2)(q_0 - q_1 + q_2) \\
r_0 &= p_0 q_0 \\
r_1 &= (p_0 + p_1 + p_2)(q_0 + q_1 + q_2) \\
r_2 &= (p_0 + 2p_1 + 4p_2)(q_0 + 2q_1 + 4q_2).
\end{aligned}$$

But now, one can use symbolic linear algebra to express the coefficients of $h(x)$ in terms of the r_t's (think of this as a precomputation, depending only on the parameter

d):

$$\begin{aligned} h_0 &= r_0 \\ h_1 &= r_{-2}/12 - 2r_{-1}/3 + 2r_1/3 - r_2/12 \\ h_2 &= -r_{-2}/24 + 2r_{-1}/3 - 5r_0/4 + 2r_1/3 - r_2/24 \\ h_3 &= -r_{-2}/12 + r_{-1}/6 - r_1/6 + r_2/12 \\ h_4 &= r_{-2}/24 - r_{-1}/6 + r_0/4 - r_1/6 + r_2/24. \end{aligned}$$

Now, to multiply two base-b numbers $x = x_0 + x_1 b + x_2 b^2$ and $y = y_0 + y_1 b + y_2 b^2$, we compute the corresponding r_t's by performing the corresponding additions and 5 multiplications. We then compute the h_i's above for $i = 0, \ldots, 4$. Note that these h_i's are not yet the base-b digits (they need not be in $[0, b-1)$). We adjust the carries by a classical procedure known as *carrying*.

To compute the gain in the above procedure, we compare to Karatsuba's algorithm where we multiplied two size b^2 numbers using 3 instead of 4 multiplications of size b numbers. Here, we multiply two size b^3 numbers using 5 instead of 9 multiplication, so the gain factor is $9/5$. If we recursively use the same Toom–Cook procedure to multiply the smaller-size numbers in the computation of the r_t's, we obtain a total of $\mathcal{O}(n^{\log_3 5})$ small size multiplications (here, by small, we mean multiplications of constant size independent of n). Since $\log_3 5 < \log_2 3$ (in Karatsuba's run-time), the Toom–Cook algorithm runs asymptotically faster in our case.

We can generalize the above procedure for any degree d: there are $2d-1$ multiplications in the computations of $r_{-d+1}, \ldots, r_{d-1}$ and again, by a recursive application, we obtain a total of $\mathcal{O}(n^{\log_d(2d-1)})$ small multiplications. We can thus bring the complexity to $\mathcal{O}(n^{1+\varepsilon})$ for any positive number $\varepsilon > 0$.

Note, however, that this is just an asymptotic analysis and is far from what one expects in practice - in particular, the simple analysis above ignores completely the bitwise operations used in the additions and multiplications by small constants (which contribute significantly as the size of the numbers grows and as $n \to \infty$). Even if often multiplications by small constants are cheap, they grow significantly with the size of the coefficients in the computations of the h_i's. Yet, the algorithm is certainly of theoretical interest and in addition, it can be useful for special hardware where multiplications are particularly expensive compared to additions.

1.3 Fast Fourier Transform (FFT) methods

To simplify the exposition, we will present a very similar algorithm that multiplies quickly two polynomials.

1.3.1 Version for polynomials

Let $p(X), q(X) \in \mathbf{R}[X]$, both of degree $n-1$. Without too much loss of generality, we will assume that $n = 2^k$ is a power of 2. If

$$p(X) = a_0 + a_1 X + \cdots + a_{n-1}X^{n-1} \qquad \text{and} \qquad q(X) = b_0 + b_1 X + \cdots + b_{n-1}X^{n-1},$$

then the product polynomial $r(X) = p(X)q(X) = c_0 + c_1 X + \cdots + c_{2n-2}X^{2n-2}$ has degree $2n-2$ and the coefficients are given by $c_k = \sum_{\substack{i+j=k \\ 0 \le i,j \le n-1}} a_i b_j$. Computing the above product requires multiplying each a_i with each b_j and hence, n^2 multiplications (or less if some coefficients are 0). For each coefficient c_k, the number of additions is equal to the number of pairs (i,j) satisfying the conditions minus one. This yields a total of $\sum_{k=0}^{n-1} k + \sum_{k=n}^{2n-2} (2n-2-k) = n^2 - 2n + 1$ additions We thus have a run-time of $\mathcal{O}(n^2)$ elementary operations in $\mathbf{R}$ which makes the algorithm rather slow in practice (here, multiplications in $\mathbf{R}$ are considered much more expensive than additions in $\mathbf{R}$).

One can use another approach instead based on Fast Fourier Transform (FFT). The idea is that any polynomial $f(X) = u_0 + u_1 X + \cdots + u_{n-1}X^{n-1}$ of degree exactly $n-1$ is uniquely determined by its values at n distinct points (finding the polynomial from the evaluations is known as interpolation). A good choice of evaluation points are the numbers ω_n^k for $0 \le k \le n-1$, where ω_n is a primitive nth root of unity (for example $\omega_n = e^{2\pi i/n}$). In what follows, we explain why this choice is good.

We now represent any polynomial $f(X)$ of degree $n-1$ by the vector in $\mathbf{R}^n$ of its coefficients. The Discrete Fourier Transform (DFT) of $f(X)$ for ω_n is

$$\texttt{DFT}_{\omega_n} : \mathbf{u} = (u_0, u_1, \ldots, u_{n-1}) \qquad \mapsto \qquad \widehat{\mathbf{u}} = \left(\widehat{u_0}, \widehat{u_1}, \ldots, \widehat{u_{n-1}}\right), \tag{1.1}$$

where $\widehat{u_j} = \sum_{t=0}^{n-1} u_t \omega_n^{jt} = f(\omega_n^j)$. The main point here is that there are algorithms such as Algorithm **??** computing the DFT with a complexity better than the naïve approach requiring $\mathcal{O}(n^2)$ elementary operations in $\mathbf{R}$. If this is the case, we speak of Fast Fourier Transform (FFT).

The DFT-based algorithm to compute the product $r(X)$ of two degree $n-1$ polynomials $p(X)$ and $q(X)$ will essentially evaluate $p(X)$ and $q(X)$ at $X = \omega_{2n}^m$ (here, ω_{2n} is a primitive $2n$-roots of unity) for every $0 \le m < 2n$ using the more efficient algorithm and will then compute the products $p(\omega_{2n}^m)q(\omega_{2n}^m) = r(\omega_{2n}^m)$. This is because of the basic convolution theorem according to which

$$\texttt{DFT}_{\omega_{2n}}(r(X)) = \texttt{DFT}_{\omega_{2n}}(p(X) \cdot q(X)) = \texttt{DFT}_{\omega_{2n}}(p(X)) \cdot \texttt{DFT}_{\omega_{2n}}(q(X)).$$

We will thus get more efficiently the discrete Fourier transform of the coefficient vector of $r(X)$. To get $r(X)$ itself, we will use an inverse-FFT algorithm to compute the inverse $\texttt{IDFT}_{\omega_{2n}}$ of the transformation (**??**). But the inverse can be performed essentially via the same algorithm as `RECURSIVE-DFT` with the same complexity. The

inversion is based on the relation

$$\frac{1}{2n}\,\mathtt{DFT}_{\omega_{2n}^{-1}} \circ \mathtt{DFT}_{\omega_{2n}} = \mathrm{id}\,.$$

Note that we consider a $2n$th roots instead of a nth root because of r.

Algorithm 1 `DFT-Product`

Require: Two degree $n-1$ polynomials p and q seen in $\mathbf{R}^n$, ω_{2n} a primitive $2n$-root of unity

Ensure: The vector of the coefficients of $r = p \cdot q$ in $\mathbf{R}^{2n-1}$

1: $\widehat{p} := \mathtt{DFT}_{\omega_{2n}}(p)$
2: $\widehat{q} := \mathtt{DFT}_{\omega_{2n}}(q)$
3: $\widehat{r} := \widehat{p} \cdot \widehat{q}$
4: $r := \dfrac{1}{2n}\,\mathtt{DFT}_{\omega_{2n}^{-1}}(\hat{r})$
5: **return** r

Example 1. For example, let $p(X) = a_0 + a_1 X$ and $q(X) = b_0 + b_1 X$, so that $n = 2$. Consider $\omega_{2n} = e^{2\pi i/(2n)} = i$. Then,

$$\mathtt{DFT}_i(a_0, a_1) = [a_0 + a_1, a_0 + a_1 i, a_0 - a_1, a_0 - a_1 i]$$

and similarly for q. The product of these two vectors is

$$\begin{bmatrix} (a_0 + a_1)(b_0 + b_1), \\ (a_0 + a_1 i)(b_0 + b_1 i), \\ (a_0 - a_1)(b_0 - b_1), \\ (a_0 - a_1 i)(b_0 - b_1 i) \end{bmatrix}$$

and the DFT of this last vector at $\omega_{2n}^{-1} = -i$ is $[4a_0b_0, 4(a_0b_1 + a_1b_0), 4a_1b_1, 0]$.

We now present a fast recursive algorithm (Algorithm **??**) to compute the DFT. It is based on the remark that, for $n = 2^k$, $f(X) = a_0 + a_1 X + \cdots + a_{n-1}X^{n-1} = (a_0 + a_2 X^2 + \cdots + a_{n-2}X^{n-2}) + X(a_1 + a_3 X^2 + \cdots + a_{n-1}X^{n-2})$ so that evaluating $f(X)$ at a primitive nth root of unity can be done by evaluating two degree $(n-2)/2$ polynomials at a primitive $(n/2)$th root of unity.

Let $T(n)$ be the run-time (number of elementary steps, i.e., "addition" and "multiplication" of complex numbers). Steps 7. and 8. take a total of $2T(n/2)$ elementary steps whereas steps $12 - 16$ take a linear (in n) number of steps, i.e., $\Theta(n)$. We thus get a recurrence relation

$$T(n) = 2T(n/2) + \Theta(n),$$

from which we compute $T(n) = \Theta(n \log_2 n)$. This is certainly asymptotically faster than the naïve algorithm described in the beginning.

Algorithm 2 `RECURSIVE-DFT`

Require: An integer $n = 2^k$ and a vector $\mathbf{a} = (a_0, a_1, \ldots, a_{n-1})$.
Ensure: $\texttt{DFT}_{\omega_n}(\mathbf{a}) = \widehat{\mathbf{a}} = (\widehat{a_0}, \widehat{a_1}, \ldots, \widehat{a_{n-1}})$.

```
a^even := (a_0, a_2, ..., a_{n-2})
a^odd := (a_1, a_3, ..., a_{n-1})
if n = 2 then
    â^even := a^even
    â^odd := a^odd
else
    â^even := RECURSIVE-DFT(n/2, a^even)
    â^odd := RECURSIVE-DFT(n/2, a^odd)
end if
ω_n = e^{2πi/n}
w = 1
for i = 0, ..., 2^{k-1} − 1 do
    â_i = â_i^even + w â_i^odd
    â_{i+2^{k-1}} = â_i^even − w â_i^odd.
    w := w · ω_n
end for
return â = (â_0, ..., â_{n-1}).
```

1.3.2 Cooley–Tukey algorithm

The above recursive FFT algorithm is a particular case of a more general FFT algorithm due to Cooley and Tukey [**?**], one of the most common and general variants of FFT. Here, we assume that $n = n_1 n_2$ and we arrange the coefficients into a 2D array. One then expresses

$$\begin{aligned}
\widehat{a_{k_1 n_2 + k_2}} &= \sum_{m_1=0}^{n_1-1} \sum_{m_2=0}^{n_2-1} a_{n_1 m_2 + m_1} e^{-\frac{2\pi i}{n_1 n_2}(n_1 m_2 + m_1)(n_2 k_1 + k_2)} = \\
&= \sum_{m_1=0}^{n_1-1} e^{-\frac{2\pi i}{n_1 n_2} m_1 k_2} \left(\sum_{m_2=0}^{n_2-1} a_{n_1 m_2 + m_1} e^{-\frac{2\pi i}{n_2} m_2 k_2} \right) = \\
&= \sum_{m_1=0}^{n_1-1} \left(\sum_{m_2=0}^{n_2-1} a_{n_1 m_2 + m_1} e^{-\frac{2\pi i}{n_2} m_2 k_2} \right) e^{-\frac{2\pi i}{n_1 n_2} m_1 (n_2 k_1 + k_2)}.
\end{aligned}$$

The above formula shows that to perform a DFT of size $n_1 n_2$, it suffices to perform:

1. n_1 DFTs of size n_2,
2. Multiplications by the appropriate roots of unity,
3. n_2 DFTs of size n_1.

1.3.2.1 `RecursiveDFT` as Radix-2 DIT

The algorithm `RECURSIVE-DFT` described above is a 2-radix decimation-in-time (radix-2 DIT) form of the algorithm of Cooley–Tukey. There are other forms of Cooley–Tukey that are useful in practice.

1.3.2.2 Data ordering, data access and variations

There are different variations of the above FFT algorithms according to the application or the architecture. See [**?**] for a good overview of some of these variations of FFT.

One particular aspect is designing an *in-place* algorithm: i.e., an algorithm that overwrites its input with its output data that uses constant auxiliary storage. To get an in-place Radix-2 DIT, one uses a technique known as bit-reversal.

Algorithm 3 `IterativeDFT`

Require: An integer $n = 2^k$ and a vector $\mathbf{a} = (a_0, a_1, \ldots, a_{n-1})$.
Ensure: A vector r containing $\texttt{DFT}_{\omega_n}(\mathbf{a}) = \widehat{\mathbf{a}} = (\widehat{a_0}, \widehat{a_1}, \ldots, \widehat{a_{n-1}})$.

```
1: bit-reverse-copy(a, r)
2: for s = 1, ..., k do
3:    m = 2^s and ω_m := exp(−2πi/m)
4:    for ℓ = 0, m, 2m, ..., 2^k − m do
5:       ω := 1
6:       for j = 0, ..., m/2 − 1 do
7:          t := ωr[k + j + m/2]
8:          u := r[k + j]
9:          r[k + j] := u + t
10:         r[k + j + m/2] := u − t
11:         ω := ωω_m
12:      end for
13:   end for
14: end for
15: return r.
```

Dimitar : **Complete!**

1.3.3 Multiplying large integers - the method of Schönhage–Strassen

An efficient algorithm based on FFT that multiplies two n-digit numbers was discovered in 1971 by Schönhage and Strassen [**?**]. The run-time of this algorithm is $\mathcal{O}(n \log n \log\log n)$ which is less than the run-time of Karatsuba multiplication. Note that it outperforms the method of Karatsuba and other older methods for numbers of more than 10,000 decimal digits. It is currently a part of the GNU Multiprecision Library and is used for at least 1728 to 7808 64-bit words (33,000 to 150,000 decimal digits), depending on architecture. In addition, a Java implementation of the method is used for implementing multiplication in Java for big integers of more than 74,000 decimal digits[1].

The basic idea is the following: to multiply two numbers, e.g., 156×723, we

[1] The Java `BigInteger` class (`https://docs.oracle.com/javase/7/docs/api/java/math/BigInteger.html`) uses the algorithm of Schönhage and Strassen.

consider first the linear convolution sequence:

$$\begin{array}{ccccc} & & 1 & 5 & 6 \\ & & 7 & 2 & 3 \\ \hline & & 3 & 15 & 18 \\ & 2 & 10 & 12 & \\ 7 & 35 & 42 & & \\ \hline 7 & 37 & 55 & 27 & 18 \end{array}$$

The sequence $(7, 37, 55, 27, 18)$ is known as the *linear (or acyclic) convolution* of the two sequences $(1, 5, 6)$ and $(7, 2, 3)$. In general, for two n-digit numbers, the length of that sequence is always $2n - 1$. There are two useful convolutions computed out of that sequence: the *cyclic convolution* and the *negacyclic convolution*. The cyclic convolution in the example above yields the sequence

$$\begin{array}{ccccc} & & 55 & 27 & 18 \\ + & & & 7 & 37 \\ \hline & & 55 & 34 & 55 \end{array}$$

The negacyclic convolution is

$$\begin{array}{ccccc} & & 55 & 27 & 18 \\ - & & & 7 & 37 \\ \hline & & 55 & 20 & -19 \end{array}$$

The key observation is that the product of two n-digit base-b numbers is equivalent modulo $b^n - 1$ to the cyclic convolution obtained from the two sequences corresponding to the base-b digits of the numbers. Similarly, the product of two n-digit base-b numbers is equivalent modulo $b^n + 1$ to the negacyclic convolution. The Schönhage–Strassen's algorithm relies on the negacyclic convolution rather than the cyclic one for various efficiency (and other) reasons that we explain below.

The idea uses a weighted version of the DFT algorithm in order to compute what is called the negacyclic convolution. More precisely, letting v_x and v_y be the vectors of base-b digits of the numbers x and y, we have

$$\texttt{CyclicConvolution}(v_x, v_y) = \texttt{IDFT}_\omega(\texttt{DFT}_\omega(v_x) \cdot \texttt{DFT}_\omega(v_y)),$$

and

$$\texttt{NegacyclicConvolution}(v_x, v_y) = A^{-1} \cdot \texttt{IDFT}_\omega(\texttt{DFT}_\omega(A \cdot v_x) \cdot \texttt{DFT}_\omega(A \cdot v_y)),$$

where $A = \mathrm{diag}(\omega^i)_{i=0}^n$ where ω is a primitive $2n$th root of unity.

In order to apply the recursive DFT algorithm in practice to integers in b-base representation, we need to work modulo some number N (what is known as the *Number Theoretic Transform, or NTT*). Over the real numbers $\mathbf{R}$ or the complex numbers $\mathbf{C}$, it was natural to use a primitive $2n$th root of unity. Yet, it is not automatic that such a root of unity would exist modulo N. We thus need to determine

a special modulus N for which we have a primitive root of unity. If N is sufficiently large, then computing $xy \mod N$ yields the product xy.

A major part of the algorithm is the careful choice of N to ensure that multiplications the primitive root and reductions modulo N are performed very efficiently (essentially, using only bit shifts and additions).

Algorithm 4 `Schönhage-Strassen's algorithm`

Require: Integers x and y; an integer n

Ensure: Computes $xy \mod 2^n + 1$ using the negacyclic convolution

1: Decompose both x and y into 2^k equal parts where $2^k \mid n$ and set n' to be the smallest integer that is at least $2n/2^k + k$ and is divisible by 2^k (n' is the recursion length)
2: Compute $A \cdot v_x$ and $A \cdot v_y$ via shifts (the jth component shifted by $n'j/2^k$)
3: Compute $\texttt{DFT}_\omega(A \cdot v_x)$ and $\texttt{DFT}_\omega(A \cdot v_y)$ via the NTT variant of `RECURSIVE-DFT` (using $\omega = 2^{2n'}/2^k$ as a primitive 2^kth root of unity, one performs multiplications as shifts)
4: Apply recursively the algorithm to compute the element-wise product $\texttt{DFT}_\omega(A \cdot v_x) \cdot \texttt{DFT}_\omega(A \cdot v_y)$
5: Compute $\texttt{IDFT}_\omega(\texttt{DFT}_\omega(A \cdot v_x) \cdot \texttt{DFT}_\omega(A \cdot v_y))$ using the NTT variant of `RECURSIVE-DFT` (multiplications are again shifts)
6: Multiply the result vector by A^{-1} using shifts only
7: **return** Result after carrying modulo $2^n + 1$.

The complexity is thus expressed in terms of the parameter k in Step 1. The optimal k is when $2^k \sim \sqrt{n}$ and in this case, we obtain a complexity $\mathcal{O}(n \log n \log\log n)$.

1.3.3.1 An algorithm for polynomial multiplication in $\mathbf{Q}[x]$.

Note that the rational numbers $\mathbf{Q}$ do not contain a primitive $2n$th root of unity. Instead, assuming that $n = 2^k$, we consider the polynomials in $\mathbf{Q}[x]$ modulo $x^n + 1$. We have the following congruences of polynomials:

$$x^n \equiv -1 \bmod x^n + 1 \qquad \text{and} \qquad x^{2n} \equiv 1 \bmod x^n + 1.$$

This means that the polynomial $\omega = x \mod x^n + 1$ is a $2n$th root of unity. Since 2 is invertible in $\mathbb{Q}$, ω is a primitive $2n$th root of unity. If we are able to replace arithmetic in $\mathbf{Q}$ with arithmetic with rational polynomials modulo $x^n + 1$, we will then be able to compute the product of two polynomials in $\mathbf{Q}[x]$ of degrees at most n in $\mathcal{O}(n \log n)$ operations with polynomials mod $x^n + 1$.

To make this precise, consider $f, g \in \mathbf{Q}[x]$ with $\deg(fg) \leq 2n = 2^k$ and let $m = 2^{\lfloor k/2 \rfloor}$, $t = 2n/m$. Write the polynomials f and g as

$$f(x) = f_0(x) + x^m f_1(x) + \cdots + x^{m(t-1)} f_{t-1}(x)$$

and

$$g(x) = g_0(x) + x^m g_1(x) + \cdots + x^{m(t-1)} g_{t-1}(x),$$

where the degrees of the f_i's and g_i's are less than m. We then see that

$$f(x) = F(x, x^m) \text{ for } F(x, y) = f_0(x) + yf_1(x) + \cdots + y^{t-1}f_{t-1}(x),$$

and

$$g(x) = G(x, x^m) \text{ for } G(x, y) = g_0(x) + yg_1(x) + \cdots + y^{t-1}g_{t-1}(x).$$

The key observation is that in order to compute fg, it suffices to compute FG modulo $y^t + 1$. Indeed, if

$$F(x, y)G(x, y) = H(x, y) + q(x, y)(y^t + 1).$$

then

$$f(x)g(x) \equiv H(x, x^m) \bmod x^{mt} + 1 = H(x, x^m) \bmod x^n + 1.$$

1.3.3.2 An algorithm for large integer multiplication.

We are not presenting the version of the algorithm of Schönhage and Strassen, but rather, a version based on the Chinese remainder theorem. Suppose that one has two large integers $a = \sum_{0 \le i < \ell} a_i 2^{Bi}$ and $b = \sum_{0 \le i < \ell} b_i 2^{Bi}$. We will also assume that ℓ does not exceed 2^B (typical examples that occurs in practice is $B = 64$). In this case, we can choose three auxiliary primes p_1, p_2 and p_3 between 2^{B-1} and 2^B. In order to obtain ab, it suffices to compute $ab \bmod p_i$ for each $i = 1, 2, 3$ and then use the Chinese remainder theorem. If the p_i's are chosen appropriately (i.e., such that $p_i - 1$ is divisible by a large power of 2) then $a \bmod p_1$ can be computed using FFT modulo p_i (by the choice of p_i, there will be a primitive $2n$th root of unity modulo p_i).

1.4 Recent developments and summary

In 2007, Fürer [**?**] presented an algorithm for asymptotically faster multiplication of very large integers compared to the algorithm of Schönhage and Strassen.

Surprisingly, a recent algorithm of Harvey and van der Hoeven [**?**] multiplies two n-bit integers in $\mathcal{O}(n \log n)$ operations, thus, proving a long-standing conjecture of Schönhage–Strassen.

In summary, we have seen the following multiplication algorithms with indicated complexities:

Algorithm	**Complexity**
Schoolbook Multiplication	$\mathcal{O}(n^2)$
Karatsuba	$\mathcal{O}(n^{\log_2 3})$
Toom–Cook	$\mathcal{O}(n^{1+\varepsilon})$
Fast Fourier Transform	$\mathcal{O}(n \log n)$
Schönhage–Strassen	$\mathcal{O}(n \log n \log \log n)$
Harvey–vdHoeven	$\mathcal{O}(n \log n)$

1.5 Exercises

Exercise 1.1.
1. Prove in detail that if two maps $T, g : \mathbf{R}_{>0} \to \mathbf{R}_{>0}$ are bounded on any bounded interval, satisfy $T(x) = 2T(x/2) + g(x)$ for every real number $x \geq 1$ and $g(x) = \mathcal{O}(x)$, then $T(x) = \mathcal{O}(x \log(x))$.
2. Let $a, b \geq 2$ be integers. Prove that if a map $T : \mathbf{Z}_{>0} \to \mathbf{R}_{>0}$ satisfies $T(n) \leq aT(\lceil n/b \rceil)$ for all $n \geq 1$, then $T(n) = \mathcal{O}(n^{\log_b(a)})$.

Exercise 1.2 (SAGE)**.** Write a SAGE function `Karatsuba(A, B)` that takes two integers A and B and returns their product using the algorithm from **??**. Run and time the above algorithm on 100 pairs of uniformly random positive integers less than 2^{512}.

Exercise 1.3. Prove the correctness of the `RECURSIVE-DFT` algorithm supposing $n = 2^k$.

Exercise 1.4 (SAGE)**.**
1. Write the function `RecursiveDFT` taking as input a vector $a \in \mathbf{Z}^n$ (e.g., the coefficients of a polynomial $p(x)$ of degree n) as well as a parameter ω (e.g., a complex root of unity) and that outputs the vector $\widehat{a} = \mathrm{DFT}_\omega(a)$. Here we can assume that $n = 2^k$ is a power of 2.
2. Write the function `InverseRecursiveDFT`.
3. Write a function `DFT_product` that computes the product of two integers, using fast Fourier transform. It is not needed to implement Schönhage-Strassen's algorithm, but you can use the idea that 2 is a primitive $2n$-th root of unity in $\mathbf{Z}/(2^n + 1)\mathbf{Z}$. Otherwise working with complex roots of unity is fine.