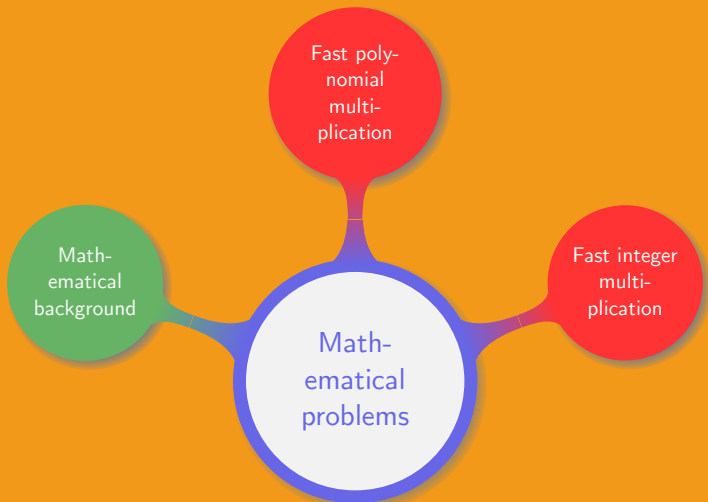


```
1001011111010100010101001111000111000100010111000000100110100001111101
1010010110011100111011101010001110000010001101110101011011111101010000
11011100111000110100110101100110110101010100111101000010101010101000
000110011110001000000110101011011110111100110000001010101011110100001
1100101110000100001001010110000011100010101000100110110010001101100100
0101011010111101010011001011101001111100101111001011110011110100
01100011111010000110100111110101110011001101111011001100110111101001101
11101001000001001111011110100110001100110011001100110011001100110011001
100111100011110100101011101010111001100110011001100110011001100110011001
011111011111101001001001001001001001001001001001001001001001001001001001
1110110110011100001001010101010101010101010101010101010101010101010101
1101101000100110010100100111100001010011100001000101110010011011010011
0000110101010001100110000111000100101110011110000101001101100100000110
110111101001010101110101111000010100010011101100000010110111100000011
0111101000100101011011001110011101011011110010110101011001100110101100
1011011011011111010000000011101110001111010111011001110011010011100000
```

Introduction to Algorithms

6. Mathematical problems

Manuel – Fall 2020



Many applications require large numbers to be multiplied.

Common multiplication algorithms:

- Simple strategy: $\mathcal{O}(n^2)$
- Karatsuba: $\mathcal{O}(n^{\log_2 3})$

Many applications require large numbers to be multiplied.

Common multiplication algorithms:

- Simple strategy: $\mathcal{O}(n^2)$
- Karatsuba: $\mathcal{O}(n^{\log_2 3})$

Fast Fourier Transform (FFT):

- Fast polynomial and number multiplication
- One of the most important and used algorithms

Definitions

Let S and S' be two sets.

- ① An *internal composition law* (\circ) is an map from $S \times S$ into S such that

$$S \times S \longrightarrow S$$

$$(x, y) \longmapsto x \circ y.$$

- ② An *external composition law* $(*)$ is an map from $S' \times S$ into S such that

$$S' \times S \longrightarrow S$$

$$(\alpha, x) \longmapsto \alpha * x.$$

Example. For a set S , the intersection (\cap) and union (\cup) define two internal composition laws for the class of subsets of S .

Definition (Group)

A *group* is a pair (G, \circ) consisting of a set G and an internal composition law that verifies the following properties:

- i *Associativity*: $a \circ (b \circ c) = (a \circ b) \circ c$ for all $a, b, c \in G$
- ii *Existence of a unit element*: there exists an element $e \in G$ such that $a \circ e = e \circ a = a$ for all $a \in G$
- iii *Existence of inverse*: for every $a \in G$ there exists an element $a^{-1} \in G$ such that $a \circ a^{-1} = a^{-1} \circ a = e$

A group is called *abelian* if in addition to the above properties

- iv *Commutativity*: $a \circ b = b \circ a$ for all $a, b \in G$.

Definition (Ring)

A *ring* is a triple $(R, +, \cdot)$ consisting of a set R and two internal composition laws $(+)$ and (\cdot) , such that

- i $(R, +)$ is an abelian group
- ii *Multiplicative unit*: there exists an element $1 \in R$ such that

$$a \cdot 1 = 1 \cdot a = a \quad \text{for all } a \in R$$

- iii *Associativity*: for any $a, b, c \in R$,

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c$$

- iv *Distributivity*: for any $a, b, c \in R$,

$$a \cdot (b + c) = (a \cdot b) + (a \cdot c), \quad (b + c) \cdot a = (b \cdot a) + (c \cdot a)$$

A ring is called *commutative* if in addition to the above properties

- v *Commutativity*: $a \cdot b = b \cdot a$ for all $a, b \in R$

Definition (Field)

Let $(F, +, \cdot)$ be a commutative ring with unit element of addition 0 and unit element of multiplication 1. Then F is a *field* if

- i $0 \neq 1$
- ii For every $a \in F \setminus \{0\}$ there exists an element a^{-1} such that

$$a \cdot a^{-1} = 1.$$

Definition (Field)

Let $(F, +, \cdot)$ be a commutative ring with unit element of addition 0 and unit element of multiplication 1. Then F is a *field* if

- i $0 \neq 1$
- ii For every $a \in F \setminus \{0\}$ there exists an element a^{-1} such that

$$a \cdot a^{-1} = 1.$$

Remark. Another way of writing this definition is to say that $(F, +, \cdot)$ is a field if $(F, +)$ and $(F \setminus \{0\}, \cdot)$ are abelian groups, $0 \neq 1$, and \cdot distributes over $+$.

Example. Let n be an integer, and $\mathbb{Z}/n\mathbb{Z}$ be the set of the integers modulo n

- $(\mathbb{Z}/n\mathbb{Z}, +)$ also denoted $(\mathbb{Z}_n, +)$ is a group
- $(\mathbb{Z}/n\mathbb{Z}, +, \cdot)$ is a ring
- If n is prime then $(\mathbb{Z}/n\mathbb{Z}, +, \cdot)$ is the field \mathbb{F}_n
- The invertible elements of $\mathbb{Z}/n\mathbb{Z}$, with respect to ' \cdot ', form a group denoted $U(\mathbb{Z}/n\mathbb{Z})$ or sometimes \mathbb{Z}_n^\times or \mathbb{Z}_n^*
- $(\mathbb{Z}/n\mathbb{Z}[X], +, \cdot)$ is the ring of the polynomials over $\mathbb{Z}/n\mathbb{Z}$
- If n is prime and the polynomial $P(X)$ is irreducible then

$$(\mathbb{F}_n[X]/\langle P(X) \rangle, +, \cdot)$$

is a field; this is $\mathbb{F}_{n^{\deg P(X)}}$

Definitions

Let R be a ring and n be a strictly positive integer.

- ① An element a of R is a *zero divisor* if there exists $b \in R \setminus \{0\}$ such that $ab = 0$.
- ② If 0 is the only zero divisor in R , then R is an *integral domain*.
- ③ An element $\omega \in R$ is an *n th root of unity* if $\omega^n = 1$.
- ④ An element $\omega \in R$ is a *primitive n th root of unity* if $\omega^n = 1$ and for any $1 \leq k \leq n - 1$, $\omega^k \neq 1$.

Example.

- In \mathbb{C} , $e^{2i\pi/8}$ is a primitive 8th root of unity;
- In \mathbb{Z}_{17} , 2 is not a primitive 16th root of unity;

Lemma

Let R be a ring, l, n be two integers such that $1 < l < n$, and ω be a primitive n th root of unity. Then (i) $\omega^l - 1$ is not a zero divisor in R , and (ii) $\sum_{j=0}^{n-1} \omega^{lj} = 0$.

Proof. (i) By definition of a primitive n th root of unity ω^l is not 1 unless l is 0 or larger or equal to n .

(ii) Note that for any $c \in R$ and m in \mathbb{N}

$$c^m - 1 = (c - 1)(1 + c + c^2 + \cdots + c^{m-1}).$$

In particular for $c = \omega^l$ and $m = n$

$$\omega^{ln} - 1 = (\omega^l - 1)(1 + \omega^l + \cdots + \omega^{l(n-1)}).$$

As $\omega^{ln} = 1$, and $\omega^l - 1$ is not a zero divisor, $\sum_{j=0}^{n-1} \omega^{lj} = 0$. □

Definition

Let R be a ring, and $\omega \in R$ be a primitive n th root of unity. We denote a polynomial $P(X)$ of degree less than n in $R[X]$ by its coefficients

$$P(X) = \sum_{i=0}^{n-1} a_i X^i = (a_0, \dots, a_{n-1}).$$

The linear map

$$\text{DFT}_\omega : R^n \longrightarrow R^n$$

$$(a_0, \dots, a_{n-1}) \longmapsto (P(1), P(\omega), \dots, P(\omega^{n-1}))$$

evaluates P at the powers of ω and is called *Discrete Fourier Transform* (DFT).

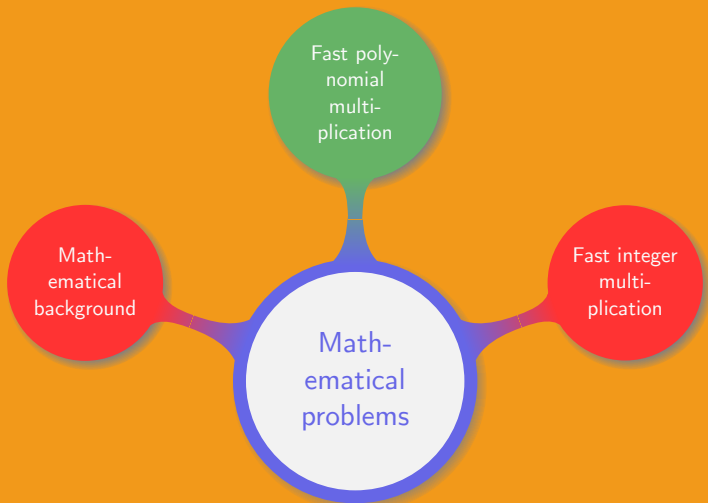
As DFT_ω is a linear map it is expressed as a matrix transformation

$$\begin{pmatrix} P(1) \\ P(\omega) \\ P(\omega^2) \\ \vdots \\ P(\omega^{n-1}) \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)^2} \end{pmatrix}}_{V_\omega} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix}.$$

Note that if ω is a primitive n th root of unity then so is ω^{-1} . Otherwise there would be a $1 \leq k < n$ such that $(\omega^{-1})^k$ is 1. And as $\omega^n = 1$ we would have $1 = \omega^{n+k}$ and $\omega^k = 1$.

Then using lemma 6.10 observe that $V_\omega V_{\omega^{-1}} = nI_n$, where I_n is the identity matrix of size $n \times n$. Thus the inverse of DFT_ω is

$$\text{DFT}_\omega^{-1} = \frac{1}{n} \text{DFT}_{\omega^{-1}}.$$



Two main cases depending on the structure of the polynomial:

- Dense: use an array where the coefficient of each monomial is stored at index “degree of the monomial”
- Sparse: use a structure composed of two arrays storing the degrees and the corresponding coefficients, respectively

Two main cases depending on the structure of the polynomial:

- Dense: use an array where the coefficient of each monomial is stored at index “degree of the monomial”
- Sparse: use a structure composed of two arrays storing the degrees and the corresponding coefficients, respectively

Alternative strategy: over an integral domain a polynomial of degree strictly less than n can be represented using its value at n distinct points

Let $P(X) = \sum_{i=0}^n a_i X^i$ be a polynomial of degree n . Evaluating P costs $\mathcal{O}(n^2)$ if naively computed. However note that $P(X)$ can be rewritten

$$P(X) = a_0 + X(a_1 + X(a_2 + \cdots + X(a_{n-1} + Xa_n))).$$

This remark dramatically decreases the complexity as it drops to $\mathcal{O}(n)$, and yields the following simple algorithm.

Algorithm. (*Horner*)

Input : a polynomial P and x the value to evaluate P at

Output: Px the evaluation of P at x

```
1 Function Horner( $P, x$ ):  
2    $Px \leftarrow 0$ ;  
3   for  $i \leftarrow \deg P$  to  $0$  do  $Px \leftarrow Px \cdot x + \text{coeff}[i]$  ;  
4   return  $Px$ ;  
5 end
```

Two main cases depending on the polynomial representation:

- Dense: usual approach, multiply the various coefficients together; complexity $\mathcal{O}(n^2)$
- Evaluation: evaluate the polynomials in n points, multiply them two by two as

$$PQ(x_i) = P(x_i)Q(x_i).$$

Complexity is $\Omega(n^2)$ since n evaluations are necessary, to which have to be added the cost of the multiplications and of the interpolation to get the final polynomial (usually $\mathcal{O}(n^2)$).

Looking back at DFT_ω , it can be viewed as a special multipoint evaluation of a polynomial in the powers $1, \omega, \dots, \omega^{n-1}$ of a primitive n th root of unity ω . Then its inverse DFT_ω^{-1} , which given n evaluations of a polynomial allows to recover its coefficients, is just the interpolation at the powers of ω .

From the previous discussion on the DFT (6.12), it is clear that knowing how to compute it efficiently means being able to also compute its inverse efficiently.

For the sake of simplicity assume $n = 2^k$, $k \in \mathbb{N}$, and observe that

$$\begin{aligned} P(X) &= \sum_{i=0}^{n-1} a_i X^i \\ &= (a_0 + a_2 X^2 + \dots + a_{n-2} X^{n-2}) + (a_1 X + a_3 X^3 + \dots + a_{n-1} X^{n-1}) \\ &= P_1(X^2) + X P_2(X^2) \end{aligned} \tag{6.1}$$

with both P_1 and P_2 of degree less than $(n-2)/2 < n/2$.

The structure of equation (6.1) suggests a “divide and conquer” approach in order to determine

$$P(\omega^i) = P_1(\omega^{2i}) + \omega^i P_2(\omega^{2i}), \quad 0 \leq i < n. \quad (6.2)$$

This formulation can be further rewritten by noticing that

$$\begin{aligned} 0 &= \omega^n - 1 \\ &= (\omega^{n/2} - 1)(\omega^{n/2} + 1). \end{aligned}$$

By lemma 6.10 none of the two factors is a zero divisor and as $\omega^{n/2} \neq 1$, ω being a primitive n th root of the unity, we conclude that $\omega^{n/2} = -1$. Hence, for all $0 \leq i < n/2$, $\omega^i = -\omega^{n/2+i}$, and (6.2) can be rewritten

$$\begin{aligned} P(\omega^i) &= P_1(\omega^{2i}) + \omega^i P_2(\omega^{2i}), \quad 0 \leq i < n/2, \\ P(\omega^{n/2+i}) &= P_1(\omega^{2i}) - \omega^i P_2(\omega^{2i}), \quad 0 \leq i < n/2. \end{aligned} \quad (6.3)$$

Algorithm. (*Fast Fourier Transform (FFT)*)

Input : a polynomial P of degree $< n$, with n a power of 2, and ω a primitive n th root of unity

Output : $\text{DFT}_\omega(P)$

```
1 Function FFT( $P, \omega$ ):  
2    $n \leftarrow \deg P + 1$ ;  
3   if  $n = 1$  then return  $P$ ;  
4    $P_1 \leftarrow \text{FFT}(\sum_{j=0}^{n/2-1} a_{2j} X^{2j}, \omega^2)$ ;  
5    $P_2 \leftarrow \text{FFT}(\sum_{j=0}^{n/2-1} a_{2j+1} \omega X^{2j}, \omega^2)$ ;  
6   for  $i \leftarrow 0$  to  $n/2$  do  
7      $P_\omega[i] \leftarrow P_{1,\omega^2}[i] + \omega^i P_{2,\omega^2}[i]$ ;  
8      $P_\omega[n/2 + i] \leftarrow P_{1,\omega^2}[i] - \omega^i P_{2,\omega^2}[i]$ ;  
9   end for  
10  return  $P_\omega$ ;  
11 end
```

Theorem

Given a polynomial P over a commutative ring and ω a primitive n th root of unity, the FFT algorithm correctly computes $\text{DFT}_\omega(P)$ in time $\mathcal{O}(n \log n)$.

Proof. The correctness clearly follows from the previous discussion, more particularly from the recurrence relation (6.3).

Let $T(n)$ be the time to compute a DFT. Then in relation (6.3) two smaller DFT are computed, plus $n/2$ multiplications and n additions, that is

$$\begin{aligned} T(n) &\leq 2T(n/2) + n/2 + n \\ &\leq 2T(n/2) + 3n/2. \end{aligned}$$

By the Master theorem (2.31) the time complexity of a DFT is $\mathcal{O}(n \log n)$. □

Let R be a ring containing a primitive n th root of unity. Given two polynomials P and Q with degrees less than $n/2$, defined over $R[X]$, we want to determine $S = PQ$. Note that n is still taken to be a power of 2.

As both P and Q are of degrees less than n they can be efficiently evaluated using the FFT algorithm (6.19). Then n multiplications in R are enough to determine S , represented using its evaluation in n points.

Applying DFT_{ω}^{-1} to the n evaluations of S , is achieved through the calculation of $1/n \text{DFT}_{\omega^{-1}} S$ (6.12). This computation returns the interpolation of S in n points (6.17), that is it determines the unique polynomial of degree less than n passing through the n points. Hence we obtain a fast strategy to compute the product of two polynomials.

Algorithm. (*Fast polynomial multiplication*)

Input : P and Q two polynomials of degree $< n/2$, with n a power of 2,
a primitive n th root of unity ω

Output : $S = PQ$

```
1 Function FPMult( $P, Q, \omega$ ):  
2    $P_\omega \leftarrow \text{FFT}(P, \omega);$   
3    $Q_\omega \leftarrow \text{FFT}(Q, \omega);$   
4    $S_\omega \leftarrow P_\omega Q_\omega;$   
5    $S \leftarrow \frac{1}{n} \text{FFT}(S_\omega, \omega^{-1});$   
6   return  $S$   
7 end
```

Algorithm. (*Fast polynomial multiplication*)

Input : P and Q two polynomials of degree $< n/2$, with n a power of 2,
a primitive n th root of unity ω

Output : $S = PQ$

```
1 Function FPMult( $P, Q, \omega$ ):  
2    $P_\omega \leftarrow \text{FFT}(P, \omega);$   
3    $Q_\omega \leftarrow \text{FFT}(Q, \omega);$   
4    $S_\omega \leftarrow P_\omega Q_\omega;$   
5    $S \leftarrow \frac{1}{n} \text{FFT}(S_\omega, \omega^{-1});$   
6   return  $S$   
7 end
```

Definition

A commutative ring containing a primitive 2^k th root of unity for any k in \mathbb{N}^* is said to *support the FFT*.

Theorem

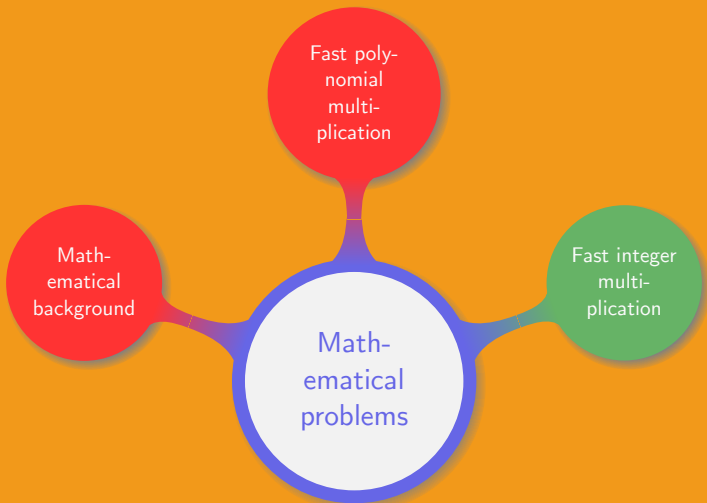
Let R be a ring supporting the FFT and n be 2^k , with k in \mathbb{N}^* . Then for two polynomials P and Q in $R[X]$, with $\deg PQ < n$, the fast polynomial multiplication algorithm computes their product in time $\mathcal{O}(n \log n)$.

Proof. The correctness of the algorithm is clear when considering the previous discussion (6.21).

The algorithm computes three DFT, n component-wise products in R , as well as n multiplications by the inverse of n in R . Therefore the overall complexity is dominated by $\mathcal{O}(n \log n)$. \square

In order to run the fast polynomial multiplication algorithm (6.22) the underlying ring R must support the FFT. In the case where R does not contain any primitive 2^k th root of unity a “virtual” one can be attached to the ring.

The Schönage-Strassen algorithm handles this special case at the cost of a slightly worse complexity. In fact their result states that over any ring the product of two polynomials of degree less than n can be computed in $\mathcal{O}(n \log n \log \log n)$ operations.



Let a and b be two N -bit long integers. For the sake of simplicity we assume N to be of the form 2^{2^l} for some integer $l > 0$. Let a_N, \dots, a_0 and b_N, \dots, b_0 denote the binary representations of a and b , respectively.

Since N was chosen to be a square it is easy to split a and b into blocks of size \sqrt{N} and write them

$$a = \sum_{i=0}^{\sqrt{N}-1} A_i 2^{i\sqrt{N}}, \text{ and } b = \sum_{i=0}^{\sqrt{N}-1} B_i 2^{i\sqrt{N}}, \quad 0 \leq A_i, B_i \leq 2^{\sqrt{N}} - 1.$$

If we consider the polynomials $A(X) = \sum_{i=0}^{\sqrt{N}-1} A_i X^i$ and $B(X) = \sum_{i=0}^{\sqrt{N}-1} B_i X^i$, then the product AB evaluated at $2^{\sqrt{N}}$ is exactly the product ab . Therefore integer multiplication can be performed via polynomial multiplication: (i) write the two integers as polynomials, (ii) apply the fast polynomial multiplication on them, and (iii) finally evaluate the product at $2^{\sqrt{N}}$.

While the first step is simple to achieve the second one requires more technical considerations. In fact the two polynomials A and B are both of degree less than $2\sqrt{N}$ and as such R must contain a primitive $2\sqrt{N}$ th root of unity.

As \mathbb{Z} does not support FFT some extra work is needed in order to attach a new “virtual” element to the ring without altering the final result.

Consider the ring $\mathbb{Z}_{2^{\sqrt{N}}+1}$ and observe that 2 is a primitive $2\sqrt{N}$ th root of unity. This is clear as $2^{\sqrt{N}}$ is -1 modulo $2^{\sqrt{N}} + 1$. Thus $t = 2\sqrt{N}$ is the smallest power for which 2^t is 1.

An obvious idea is then to perform the computation in the ring $\mathbb{Z}_{2^{\sqrt{N}}+1}[X]$. However as both A and B can feature coefficients as large as $2^{\sqrt{N}} - 1$, the polynomial C resulting from their product can have coefficients up to $\sqrt{N}2^{2\sqrt{N}}$, which is larger than $2^{\sqrt{N}} + 1$.

As a result, if coefficients in C happen to be too large they will be reduced modulo $2^{\sqrt{N}}+1$, ruining the whole calculation. A simple solution consists in performing all the computation in a larger ring. At that stage two points must be taken into consideration: (i) the use of a large ring increases the computational cost and (ii) the ring must contain a primitive $2\sqrt{N}$ th root of unity.

Note that for any $N \geq 1$, $2^{3\sqrt{N}} > \sqrt{N}2^{2\sqrt{N}}$, while 8 is a primitive $2\sqrt{N}$ th root of unity in $\mathbb{Z}_{2^{3\sqrt{N}}+1}$. The latter being a consequence of 2 being a primitive $6\sqrt{N}$ th root of unity.

Therefore it suffices to consider A and B as polynomials over the ring $\mathbb{Z}_{2^{3\sqrt{N}}+1}[X]$. Then applying the fast polynomial multiplication algorithm (6.22) and evaluating the product at $2^{\sqrt{N}}$ yields the result.

Algorithm. (*Fast integer multiplication*)

Input : two N bit integers a and b , a ring $R = \mathbb{Z}_{2^{3\sqrt{N}}+1}$, $\omega = 8$ a primitive $2\sqrt{N}$ th root of unity

Output: $c = a \cdot b$

```
1  $A \leftarrow \text{poly}(a);$                                 /* encode  $a$  as a polynomial */
2  $B \leftarrow \text{poly}(b);$                                 /* encode  $b$  as a polynomial */
3  $C \leftarrow \text{FPMult}(A, B, \omega);$ 
4  $c \leftarrow \text{Horner}(C, 2^{\sqrt{N}});$ 
5 return  $c;$ 
```

Theorem

Given two integers of length N in a ring R , the fast integer multiplication algorithm computes their product in $\mathcal{O}(\sqrt{N} \log \sqrt{N})$ arithmetic operations in R .

Proof. The correctness results from the previous discussion (6.26).

The pre-dominant computation in the algorithm is the fast polynomial multiplication of A and B , which by theorem 6.23 takes $\mathcal{O}(\sqrt{N} \log \sqrt{N})$.

□

Remark. With a bit more work it is possible to determine the complexity in term of bit operations, instead of arithmetic operations. In that case the complexity becomes

$$\mathcal{O}(N \log^{2+\log_2 3-1} N).$$

Other interesting remarks:

- At the bit level, most operations can be performed using “shifts”, incurring a linear cost in the length of the integers. This is, in particular the main reason for choosing ω to be a power of 2.
- The integer $2^{\sqrt{N}}$ has inverse $2^{6\sqrt{N}-\log_2 \sqrt{N}-1}$ in $\mathbb{Z}_{2^{3\sqrt{N}+1}}$. Observe that $2^{6\sqrt{N}} \equiv 1 \pmod{2^{3\sqrt{N}+1}}$, and $2^{\log_2 \sqrt{N}} = \sqrt{N}$.
- Although in the algorithm no coefficient reaches $2^{3\sqrt{N}} + 1$ all the calculations are performed in the ring $\mathbb{Z}_{2^{3\sqrt{N}+1}}$ and the special case of adding two $3\sqrt{N}$ bits long elements has to be considered when defining addition.
- To date the asymptotically fastest integer multiplication algorithm, due to Fürer, takes $N \log N 2^{\mathcal{O}(\log^* N)}$ bit operations.

Definition

The complexity of multiplying two polynomials of degree less than n is denoted $M(n)$, while the complexity of multiplying two n -bit integer is denoted $M_I(n)$.

For all n and m in \mathbb{N} ,

$$M(n + m) \geq M(m) + M(n) \quad \text{and} \quad M_I(n + m) \geq M_I(n) + M_I(m).$$

These simpler notations allows an easier complexity study of more advanced algorithms such as fast multi-point evaluation and fast interpolation.

1001011111010100010101001111000111000100010111000000100110100001111101
1010010110011100111011101010001110000010001101110101011011111101010000
1101110011100011010011010110011011010101010100111101000010101010101000
00011001111000100000011010101101111011110011000000101010101110100001
1100101110000100001001010110000011100010101000100110110010001101100100
01010110101110000010101100100101111000101001111000001100111001110100
01100011111010000011010011111000110111001100110111101001101
111010010000010011111011110100110000110011001001001111010001001001
100111100011111010010101110100111101000111100111101110100010
0111110111111001111100100100100110100110010010001111110000110
111011011001111000010010101011011001101011111011111011111010
1101101000100110010100100111100001010011100001000101110010011011010011
0000110101010001100110000111000100101110011110000101001101100100000110
110111101001010101110101111100001010001001110110000001011011110000011
01111010001001010110110011100111010110111100101101011001100110101100
10110110110111110100000000111011100011110101110110011100111010011100000

Thank you!