Integer multiplication with generalized Fermat primes

Svyatoslav Covanov

CARAMEL Team, LORIA, University of Lorraine
Supervised by: Emmanuel Thomé and Jérémie Detrey

Journées nationales du Calcul Formel 2015 (Cluny)

November 4, 2015

How to multiply two numbers $a = (a_0 \cdots a_N)$ and $b = (b_0 \cdots b_N)$ where a and b are given in binary representation?

How to multiply two numbers $a = (a_0 \cdots a_N)$ and $b = (b_0 \cdots b_N)$ where a and b are given in binary representation?

First idea:
Sum all $a_i * b$ using 2-shift. This method has a $O(N^2)$ bit complexity.

How to multiply two numbers $a = (a_0 \cdots a_N)$ and $b = (b_0 \cdots b_N)$ where a and b are given in binary representation?

First idea:
Sum all $a_i * b$ using 2-shift. This method has a $O(N^2)$ bit complexity.

People believed long enough it was the best complexity we could reach (Kolmogorov). Karatsuba proved that it was wrong...

The fastest known algorithm is based on the evaluation-interpolation paradigm.

Input: 2 numbers a and b of N bits.

We decompose the input into 2 polynomials $A = \sum_i a_i x^i$ and $B = \sum_i b_i x^i$ ($\deg A = \deg B = n$, $|a_i| = |b_i| = N/(2n) = k$, and $a_i = b_i = 0$ for $i > n$).

$$A(2^k) = a_0 + 2^k \times a_1 + \cdots + a_{2n-1} \times 2^{(2n-1)k} = a$$
$$B(2^k) = b_0 + 2^k \times b_1 + \cdots + b_{2n-1} \times 2^{(2n-1)k} = b$$

The fastest known algorithm is based on the evaluation-interpolation paradigm.

Input: 2 numbers a and b of N bits.

We decompose the input into 2 polynomials $A = \sum_i a_i x^i$ and $B = \sum_i b_i x^i$ ($\deg A = \deg B = n$, $|a_i| = |b_i| = N/(2n) = k$, and $a_i = b_i = 0$ for $i > n$).

$$\begin{aligned} A(2^k) &= a_0 + 2^k \times a_1 + \cdots + a_{2n-1} \times 2^{(2n-1)k} = a \\ B(2^k) &= b_0 + 2^k \times b_1 + \cdots + b_{2n-1} \times 2^{(2n-1)k} = b \end{aligned}$$

We work in some ring $\mathcal{R}$ in which we have a $2n$-th principal root of unity ω.

The fastest known algorithm is based on the evaluation-interpolation paradigm.
Input: 2 numbers a and b of N bits.
We decompose the input into 2 polynomials $A = \sum_i a_i x^i$ and $B = \sum_i b_i x^i$ ($\deg A = \deg B = n$, $|a_i| = |b_i| = N/(2n) = k$, and $a_i = b_i = 0$ for $i > n$).

$$A(2^k) = a_0 + 2^k \times a_1 + \cdots + a_{2n-1} \times 2^{(2n-1)k} = a$$
$$B(2^k) = b_0 + 2^k \times b_1 + \cdots + b_{2n-1} \times 2^{(2n-1)k} = b$$

We work in some ring $\mathcal{R}$ in which we have a $2n$-th principal root of unity ω.
We compute the $A(\omega^i)$ and $B(\omega^i)$.

The fastest known algorithm is based on the evaluation-interpolation paradigm.
Input: 2 numbers a and b of N bits.
We decompose the input into 2 polynomials $A = \sum_i a_i x^i$ and $B = \sum_i b_i x^i$ ($\deg A = \deg B = n$, $|a_i| = |b_i| = N/(2n) = k$, and $a_i = b_i = 0$ for $i > n$).

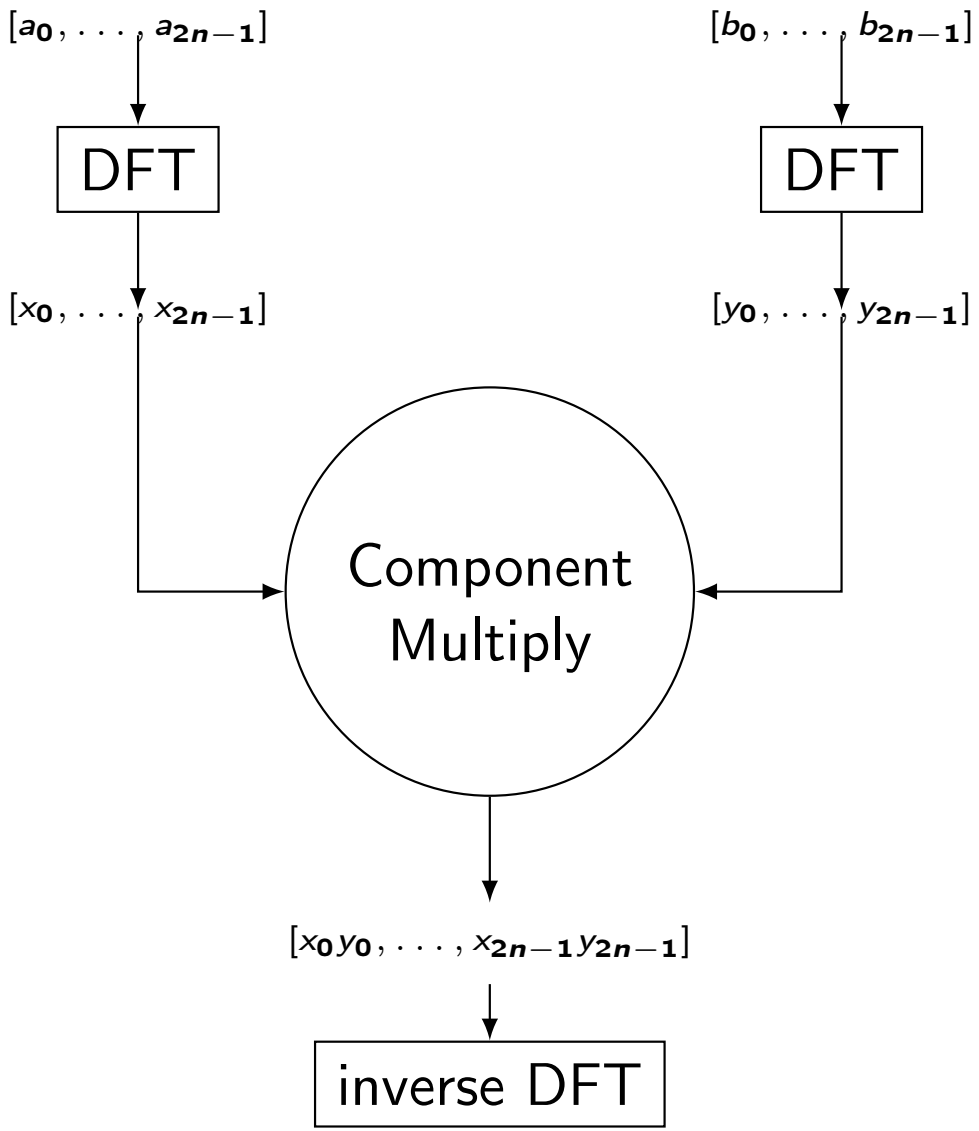
$$A(2^k) = a_0 + 2^k \times a_1 + \cdots + a_{2n-1} \times 2^{(2n-1)k} = a$$
$$B(2^k) = b_0 + 2^k \times b_1 + \cdots + b_{2n-1} \times 2^{(2n-1)k} = b$$

We work in some ring $\mathcal{R}$ in which we have a $2n$-th principal root of unity ω.
We compute the $A(\omega^i)$ and $B(\omega^i)$.
We recover $A \cdot B$ from the points $A(\omega^i) \cdot B(\omega^i)$ with Lagrange interpolation for a polynomial of degree $2n - 1$.

The fastest known algorithm is based on the evaluation-interpolation paradigm.
Input: 2 numbers a and b of N bits.
We decompose the input into 2 polynomials $A = \sum_i a_i x^i$ and $B = \sum_i b_i x^i$ ($\deg A = \deg B = n$, $|a_i| = |b_i| = N/(2n) = k$, and $a_i = b_i = 0$ for $i > n$).

$$A(2^k) = a_0 + 2^k \times a_1 + \cdots + a_{2n-1} \times 2^{(2n-1)k} = a$$
$$B(2^k) = b_0 + 2^k \times b_1 + \cdots + b_{2n-1} \times 2^{(2n-1)k} = b$$

We work in some ring $\mathcal{R}$ in which we have a $2n$-th principal root of unity ω.
We compute the $A(\omega^i)$ and $B(\omega^i)$.
We recover $A \cdot B$ from the points $A(\omega^i) \cdot B(\omega^i)$ with Lagrange interpolation for a polynomial of degree $2n - 1$.
The DFT algorithm allows one to compute the $A(\omega^i)$ and $B(\omega^i)$.


$[a_0, \ldots, a_{2n-1}]$
$[b_0, \ldots, b_{2n-1}]$
DFT
DFT
$[x_0, \ldots, x_{2n-1}]$
$[y_0, \ldots, y_{2n-1}]$
Component Multiply
$[x_0 y_0, \ldots, x_{2n-1} y_{2n-1}]$
inverse DFT

P is a polynomial of degree $2n-1$ (n is a power of 2) and ω is a $2n$-th principal root of unity in $\mathcal{R}$ ($\mathbb{C}$ or $\mathbb{Z}/p\mathbb{Z}$ for example), which means that $\sum_{j\in[0,2n-1]} \omega^{ij} = 0$ for $i \in [1, 2n-1]$.

P is a polynomial of degree $2n-1$ (n is a power of 2) and ω is a $2n$-th principal root of unity in $\mathcal{R}$ ($\mathbb{C}$ or $\mathbb{Z}/p\mathbb{Z}$ for example), which means that $\sum_{j\in[0,2n-1]} \omega^{ij} = 0$ for $i \in [1, 2n-1]$.

FFT$(P, \omega, 2n-1)$

---

if $n = 1$ **then**
  return $P_0 + P_1 + X(P_0 - P_1)$
end if
$P^{even} \leftarrow (P_{2i})_i$
$P^{odd} \leftarrow (P_{2i+1})_i$
$Q^{even} \leftarrow$ FFT$(P^{even}, \omega^2, n-1)$
$Q^{odd} \leftarrow$ FFT$(P^{odd}, \omega^2, n-1)$
$Q \leftarrow Q^{even}(X) + Q^{odd}(\omega X) + X^n \cdot (Q^{odd}(X) - Q^{even}(\omega X))$
return Q

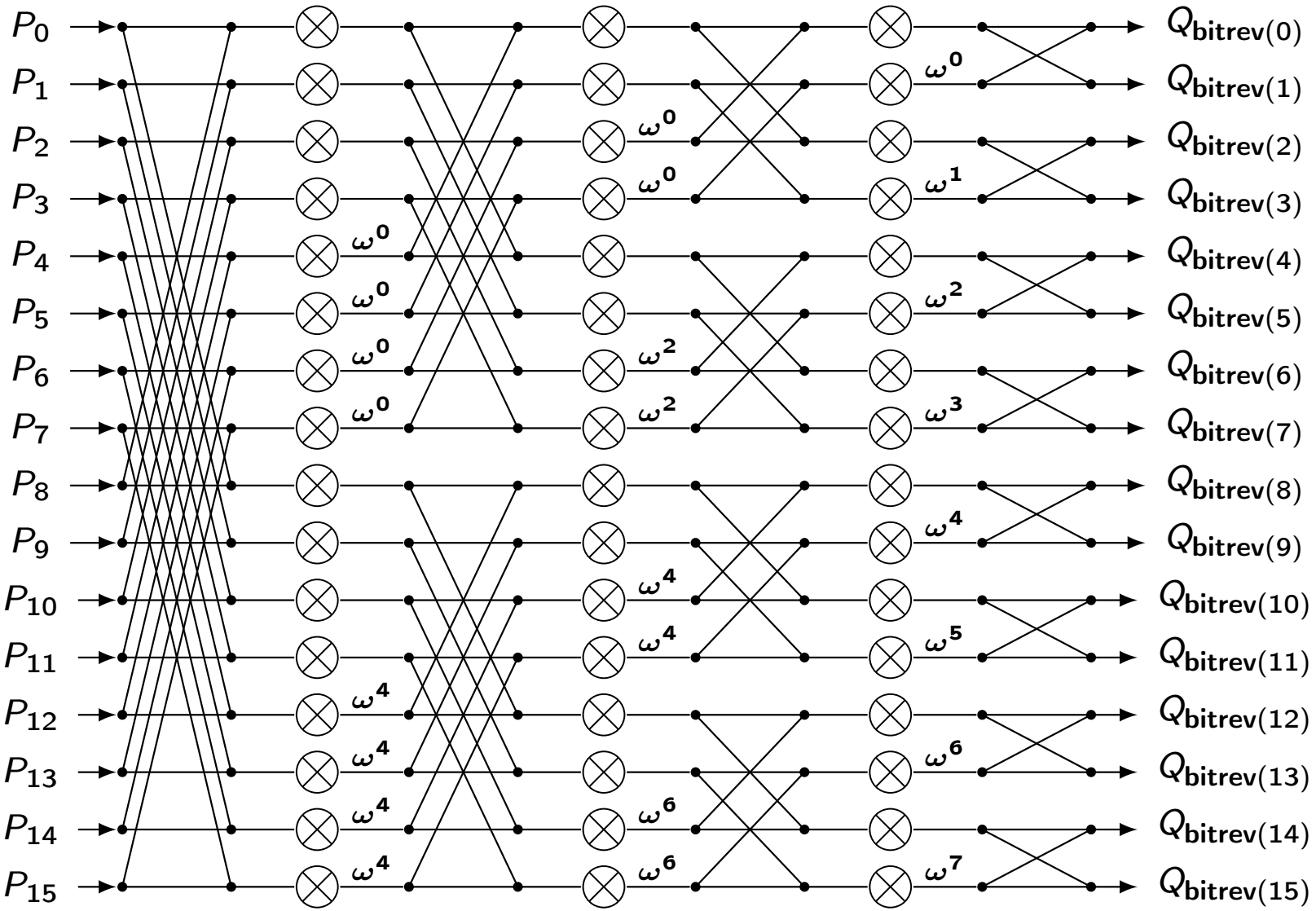
P is a polynomial of degree $2n-1$ (n is a power of 2) and ω is a $2n$-th principal root of unity in $\mathcal{R}$ ($\mathbb{C}$ or $\mathbb{Z}/p\mathbb{Z}$ for example), which means that $\sum_{j\in[0,2n-1]} \omega^{ij} = 0$ for $i \in [1, 2n-1]$.

FFT$(P, \omega, 2n-1)$

---

if $n = 1$ **then**
  return $P_0 + P_1 + X(P_0 - P_1)$
end if
$P^{even} \leftarrow (P_{2i})_i$
$P^{odd} \leftarrow (P_{2i+1})_i$
$Q^{even} \leftarrow \text{FFT}(P^{even}, \omega^2, n-1)$
$Q^{odd} \leftarrow \text{FFT}(P^{odd}, \omega^2, n-1)$
$Q \leftarrow Q^{even}(X) + Q^{odd}(\omega X) + X^n \cdot (Q^{odd}(X) - Q^{even}(\omega X))$
return Q

Complexity: $O(n \log n)$ operations in $\mathcal{R}$, among which multiplications by some powers of ω, additions and subtractions.


P_0
P_1
P_2
P_3
P_4
P_5
P_6
P_7
P_8
P_9
P_{10}
P_{11}
P_{12}
P_{13}
P_{14}
P_{15}
ω^0 ω^0 ω^0 ω^0
ω^4 ω^4 ω^4 ω^4
ω^0 ω^0 ω^2 ω^2
ω^4 ω^4 ω^6 ω^6
ω^0 ω^1 ω^2 ω^3 ω^4 ω^5 ω^6 ω^7
$Q_{\text{bitrev}(0)}$
$Q_{\text{bitrev}(1)}$
$Q_{\text{bitrev}(2)}$
$Q_{\text{bitrev}(3)}$
$Q_{\text{bitrev}(4)}$
$Q_{\text{bitrev}(5)}$
$Q_{\text{bitrev}(6)}$
$Q_{\text{bitrev}(7)}$
$Q_{\text{bitrev}(8)}$
$Q_{\text{bitrev}(9)}$
$Q_{\text{bitrev}(10)}$
$Q_{\text{bitrev}(11)}$
$Q_{\text{bitrev}(12)}$
$Q_{\text{bitrev}(13)}$
$Q_{\text{bitrev}(14)}$
$Q_{\text{bitrev}(15)}$

1. N: # bits of the integers that we multiply.
2. $2n$: degree of the polynomials A and B used to represent a and b.
3. k: # bits used to encode the coefficients of A and B: $a = A(2^k)$ and $b = B(2^k)$.

1. N: # bits of the integers that we multiply.
2. $2n$: degree of the polynomials A and B used to represent a and b.
3. k: # bits used to encode the coefficients of A and B: $a = A(2^k)$ and $b = B(2^k)$.

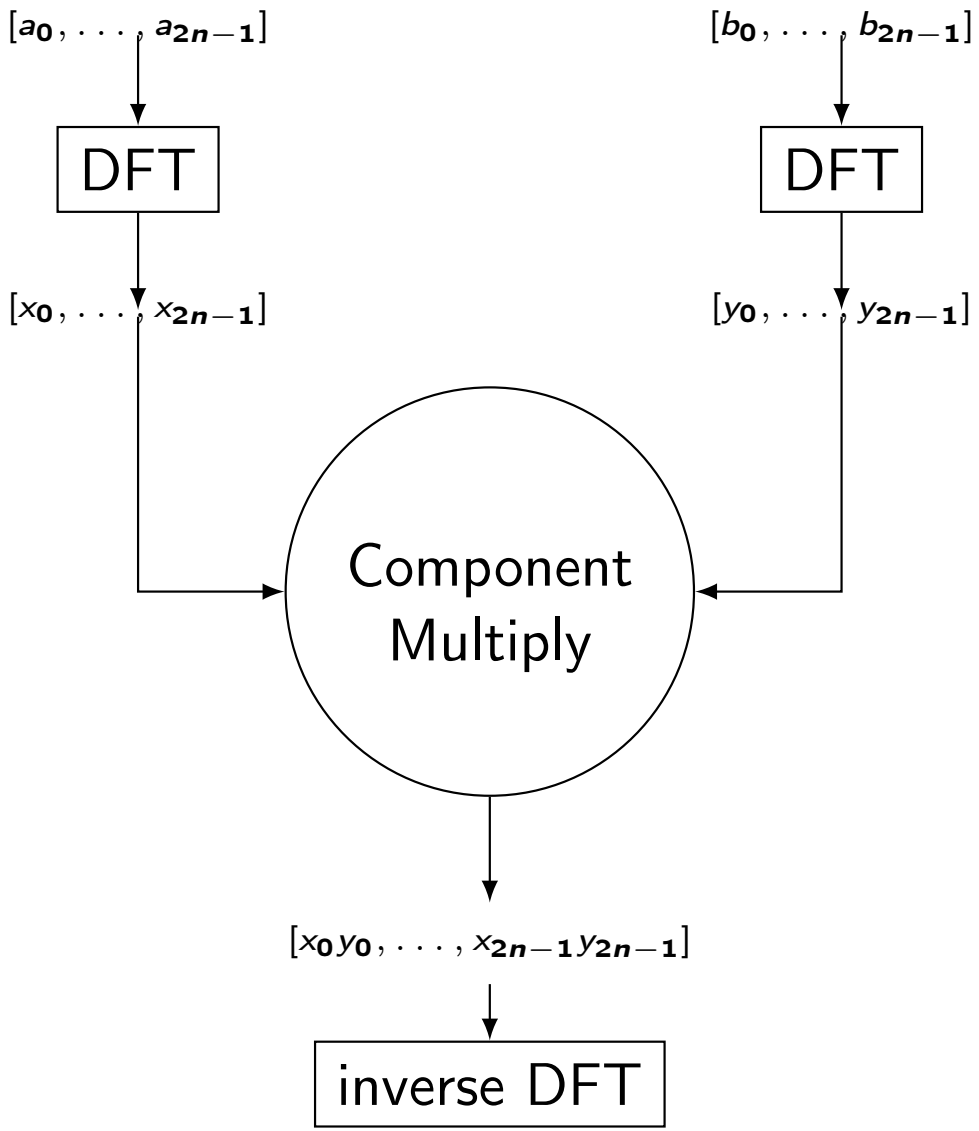
Examples:

- If $\mathcal{R} = \mathbb{C}$, then $\omega = \exp(i\pi/n)$. $k = O(\log N)$ is the best choice. Thus, the recursive calls are manipulating $O(\log N)$-sized data during convolution step.

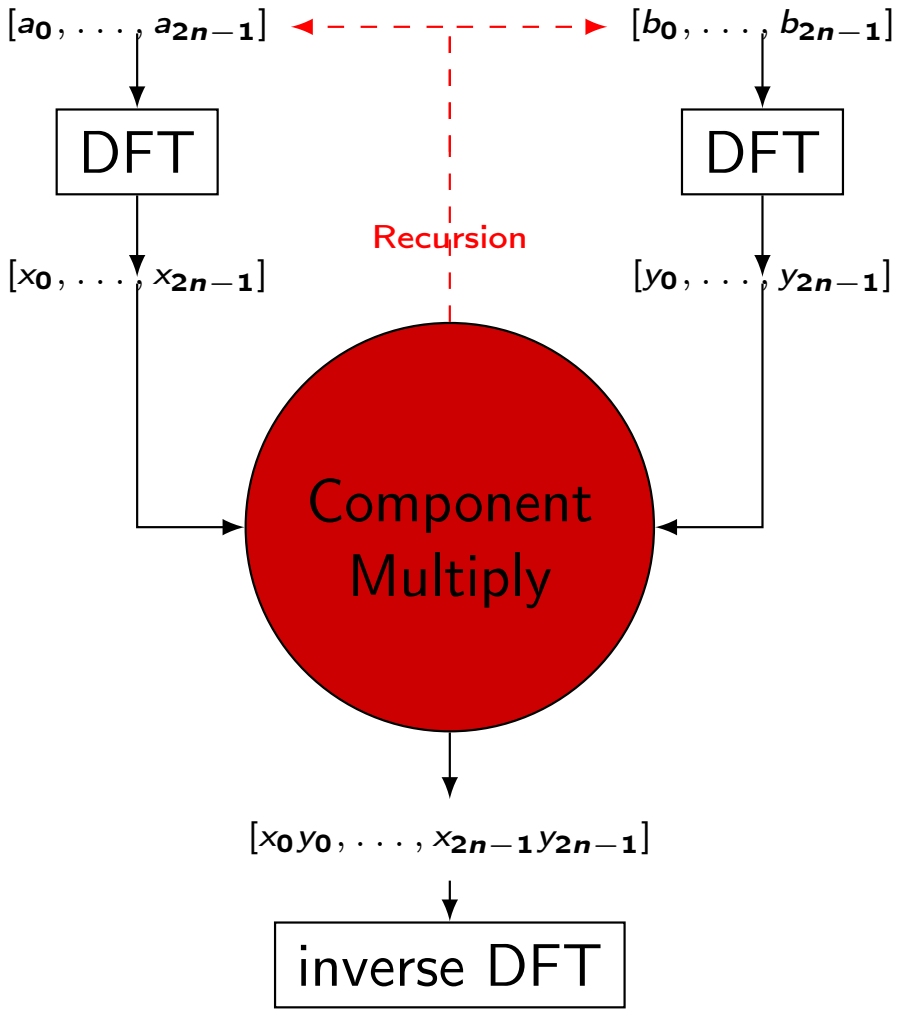
1. N: # bits of the integers that we multiply.
2. $2n$: degree of the polynomials A and B used to represent a and b.
3. k: # bits used to encode the coefficients of A and B: $a = A(2^k)$ and $b = B(2^k)$.

Examples:

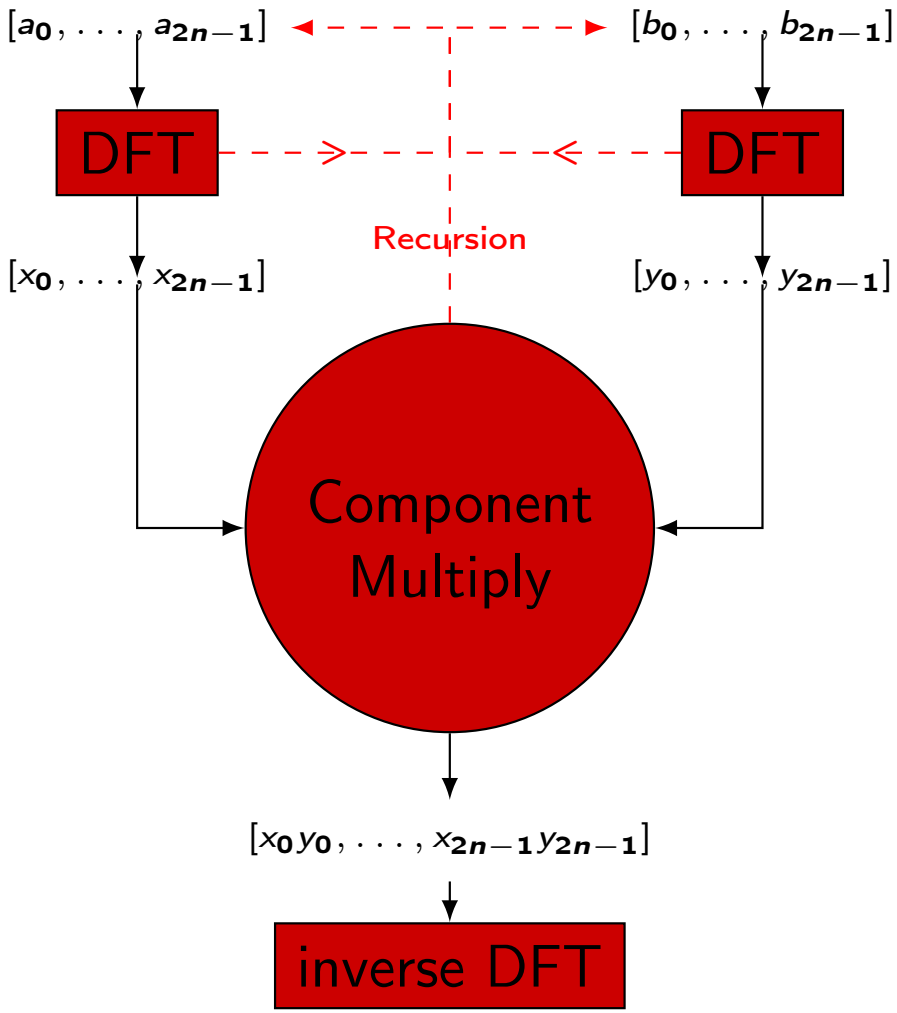
- If $\mathcal{R} = \mathbb{C}$, then $\omega = \exp(i\pi/n)$. $k = O(\log N)$ is the best choice. Thus, the recursive calls are manipulating $O(\log N)$-sized data during convolution step.
- If $\mathcal{R} = \mathbb{Z}/(2^e + 1)\mathbb{Z}$, then, $k \simeq e \simeq O(\sqrt{N})$ (smallest k possible).
  Then $\omega = 2^j$ with $j = e/n$. Multiplications by powers of ω are negacyclic permutations.

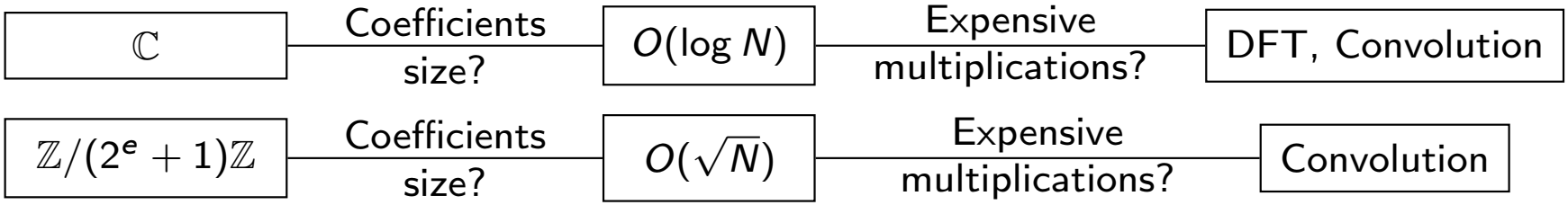

$[a_0, \ldots, a_{2n-1}]$
$[b_0, \ldots, b_{2n-1}]$
DFT
DFT
$[x_0, \ldots, x_{2n-1}]$
$[y_0, \ldots, y_{2n-1}]$
Component Multiply
$[x_0 y_0, \ldots, x_{2n-1} y_{2n-1}]$
inverse DFT

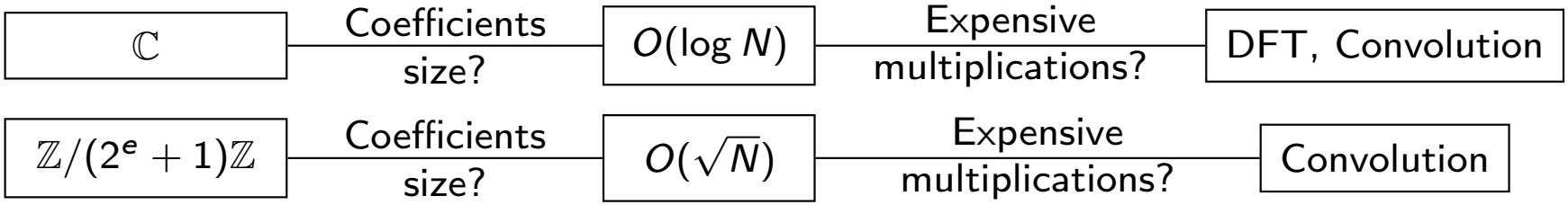
Modular Case



Complex Case



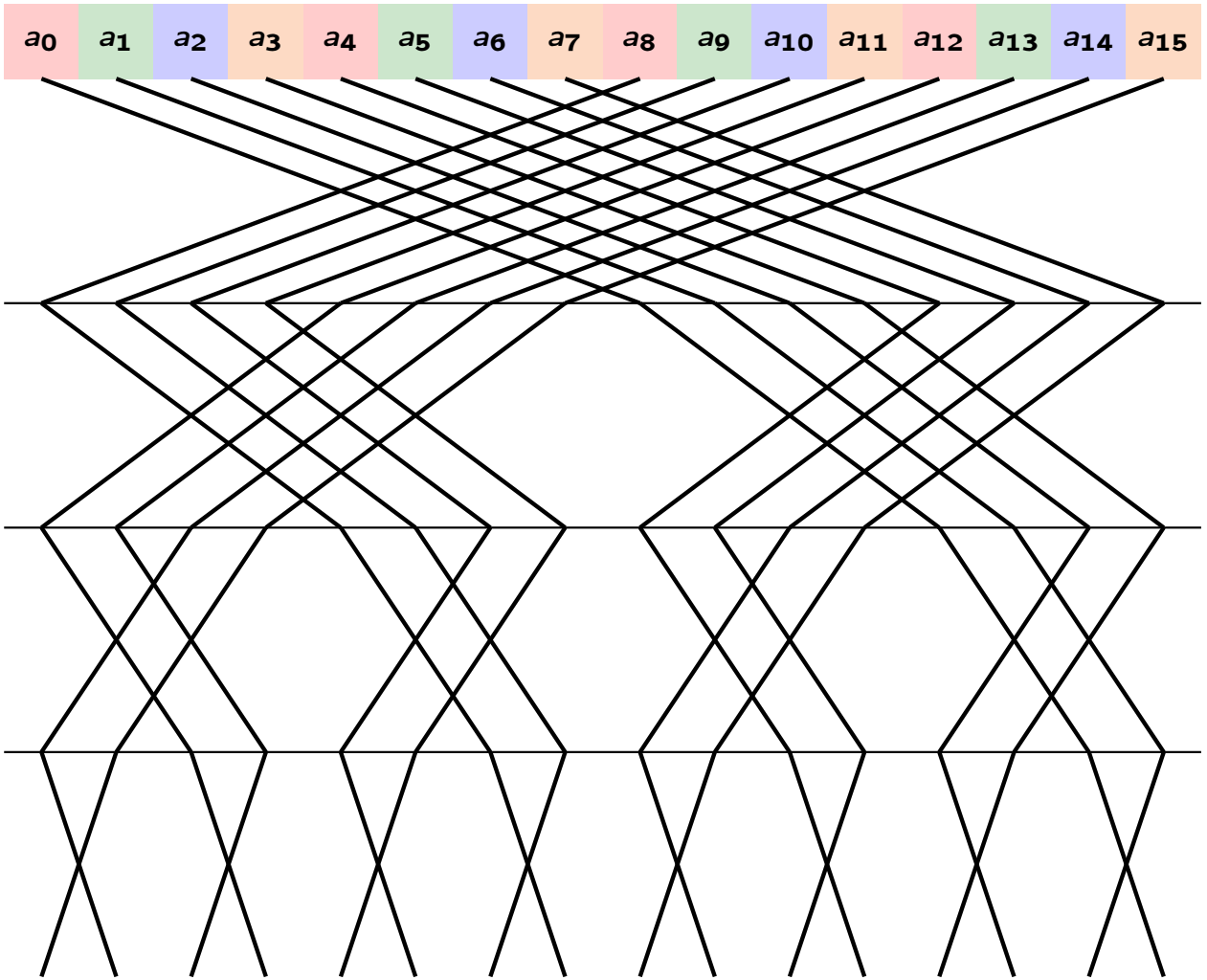

$\mathbb{C}$
Coefficients size?
$O(\log N)$
Expensive multiplications?
DFT, Convolution
$\mathbb{Z}/(2^e+1)\mathbb{Z}$
Coefficients size?
$O(\sqrt{N})$
Expensive multiplications?
Convolution



Case	Degree	Mult. by a root	Recursion	Complexity
$\mathbb{C}$	$O(N/\log N)$	expensive	$O(\log N)$	$N \log N \log\log N \cdots 2^{O(\log^* N)}$
$\mathbb{Z}/(2^e+1)\mathbb{Z}$	$O(\sqrt{N})$	cheap	$O(\sqrt{N})$	$N \log N \log\log N$

In $\mathbb{C}$, computing an FFT in $\{1, -1, i, -i\}$ is quite easy. But less obvious for superior orders...

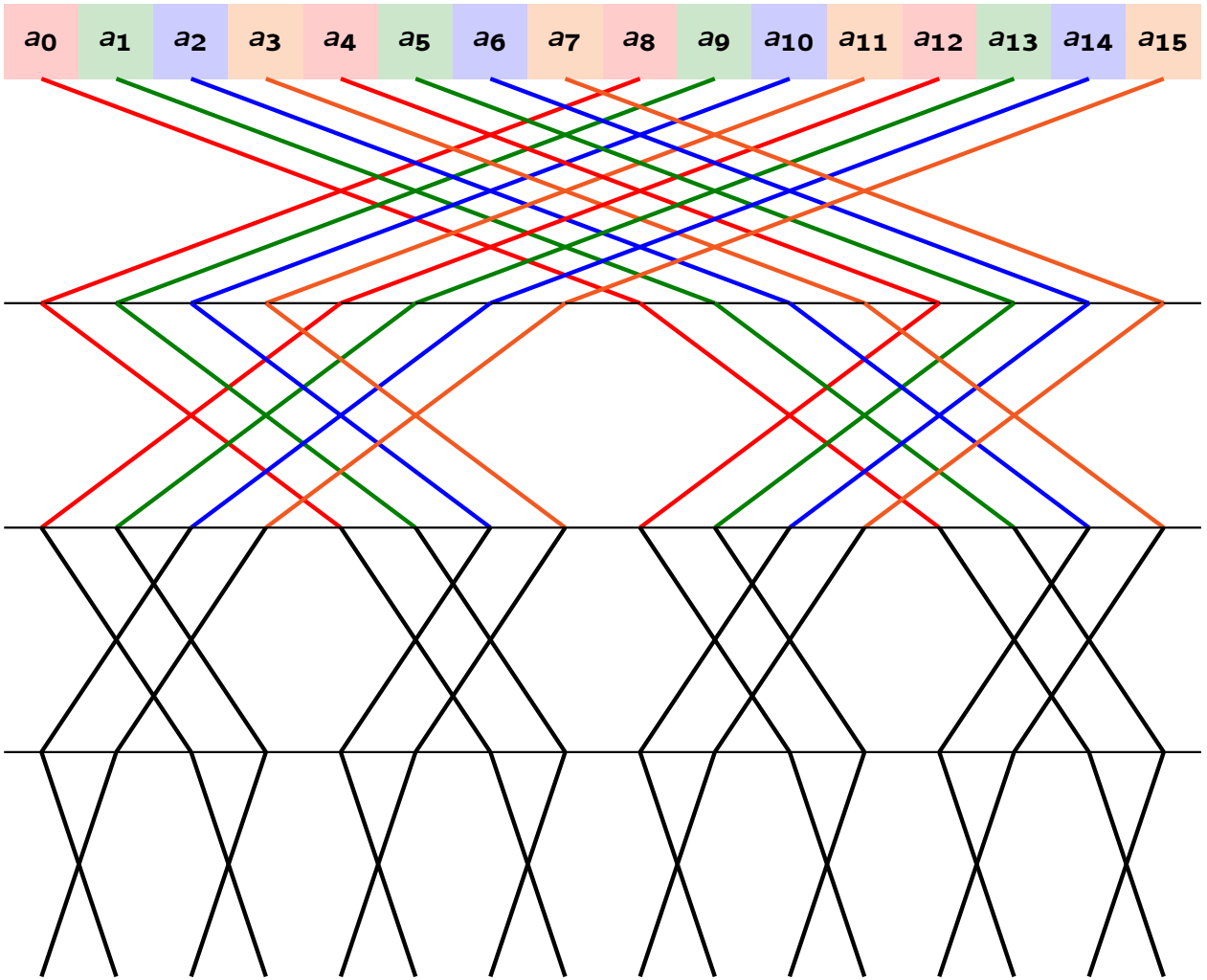
Here you can see the butterfly graph for 16-point transform.



Below, the matrix representation:

$$\begin{pmatrix} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ a_8 & a_9 & a_{10} & a_{11} \\ a_{12} & a_{13} & a_{14} & a_{15} \end{pmatrix}$$

We start by inner transforms.

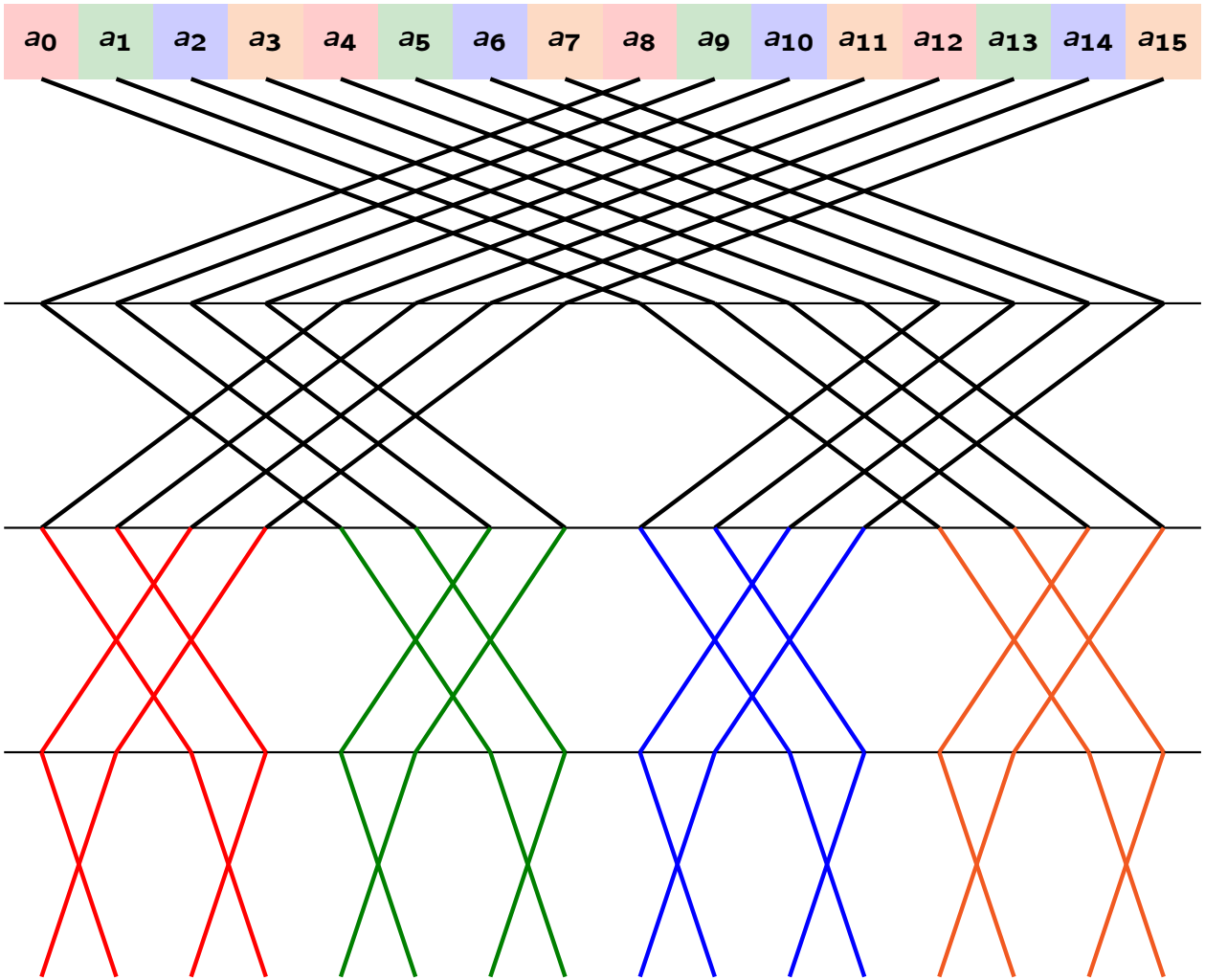


Below, the matrix representation:

$$\begin{pmatrix} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ a_8 & a_9 & a_{10} & a_{11} \\ a_{12} & a_{13} & a_{14} & a_{15} \end{pmatrix}$$

There are 4 4-points FFT.

Then the outer ones.



Below, the matrix representation:

$$\begin{pmatrix} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ a_8 & a_9 & a_{10} & a_{11} \\ a_{12} & a_{13} & a_{14} & a_{15} \end{pmatrix}$$

There are 4 4-points FFT.

- We use a polynomial ring $\mathcal{R}$ of the form $\mathcal{R} = \mathbb{C}[x]/(x^P + 1)$ or $\mathcal{R} = \mathbb{Z}[x]/(q^c, x^P + 1)\mathbb{Z}$
- There exists a $2n$-th root of unity ρ such that $\rho^{n/P} = x$ (Lagrange interpolation)
- The computation of $2n$-points FFT is factored into the computation of $\log_{2P} 2n$ times n/P $2P$-points FFT
- $P = \Theta(\log N)$ and coefficients of elements of $\mathcal{R}$ are stored on $\Theta(\log N)$ bits

- We use a polynomial ring $\mathcal{R}$ of the form $\mathcal{R} = \mathbb{C}[x]/(x^P + 1)$ or $\mathcal{R} = \mathbb{Z}[x]/(q^c, x^P + 1)\mathbb{Z}$
- There exists a $2n$-th root of unity ρ such that $\rho^{n/P} = x$ (Lagrange interpolation)
- The computation of $2n$-points FFT is factored into the computation of $\log_{2P} 2n$ times n/P $2P$-points FFT
- $P = \Theta(\log N)$ and coefficients of elements of $\mathcal{R}$ are stored on $\Theta(\log N)$ bits

Case	Degree	Mult. by a root	Recursion	Complexity
$\mathbb{C}$	$O(N/\log N)$	expensive	$O(\log N)$	$N \log N \log\log N \cdots 2^{O(\log^* N)}$
$\mathbb{Z}/(2^e + 1)\mathbb{Z}$	$O(\sqrt{N})$	cheap	$O(\sqrt{N})$	$N \log N \log\log N$
$\mathbb{C}[x]/(x^P + 1)$	$O(N/\log^2 N)$	it depends	$O(\log^2 N)$	$N \log N\, 2^{O(\log^* N)}$
$\mathbb{Z}[x]/(q^c, x^P + 1)\mathbb{Z}$	$O(N/\log^2 N)$	it depends	$O(\log^2 N)$	$N \log N\, 2^{O(\log^* N)}$

Let us multiply integers by associating to them 2 polynomials of degree $2n-1$ for which the coefficients are embedded in a finite field $\mathcal{R} = \mathbb{Z}/q\mathbb{Z}$.

Let us multiply integers by associating to them 2 polynomials of degree $2n-1$ for which the coefficients are embedded in a finite field $\mathcal{R} = \mathbb{Z}/q\mathbb{Z}$.

The prime q must verify: $2n \mid q-1$. Thus, there exists a $2n$-th principal root of unity.

Let us multiply integers by associating to them 2 polynomials of degree $2n-1$ for which the coefficients are embedded in a finite field $\mathcal{R} = \mathbb{Z}/q\mathbb{Z}$.

The prime q must verify: $2n \mid q-1$. Thus, there exists a $2n$-th principal root of unity.

1. N: # bits of the integers that we multiply.
2. $2n$: degree of the polynomials A and B used to represent a and b.
3. k: # bits used to encode the coefficients of A and B: $a = A(2^k)$ and $b = B(2^k)$; this number is given by roughly $\frac{1}{2}\log_2 q$.

Let us multiply integers by associating to them 2 polynomials of degree $2n-1$ for which the coefficients are embedded in a finite field $\mathcal{R} = \mathbb{Z}/q\mathbb{Z}$.

The prime q must verify: $2n \mid q-1$. Thus, there exists a $2n$-th principal root of unity.

1. N: # bits of the integers that we multiply.
2. $2n$: degree of the polynomials A and B used to represent a and b.
3. k: # bits used to encode the coefficients of A and B: $a = A(2^k)$ and $b = B(2^k)$; this number is given by roughly $\frac{1}{2}\log_2 q$.

A choice of q such that $k = O(\log N)$ is optimal.

- q is chosen such that $q = b^P + 1$. There exists b such that $b < P \cdot (\log P)^{1+\epsilon}$ for any $\epsilon > 0$. Thus, $\log_2 q \approx P \log P$.
- Let ρ be a $2n$-th root of unity in $\mathbb{Z}/q\mathbb{Z}$ such that $\rho^{n/P} = b$.

- q is chosen such that $q = b^P + 1$. There exists b such that $b < P \cdot (\log P)^{1+\epsilon}$ for any $\epsilon > 0$. Thus, $\log_2 q \approx P \log P$.
- Let ρ be a $2n$-th root of unity in $\mathbb{Z}/q\mathbb{Z}$ such that $\rho^{n/P} = b$.

Working in radix b is like working with "polynomials" of degree P whose coefficients are bounded by b.

- q is chosen such that $q = b^P + 1$. There exists b such that $b < P \cdot (\log P)^{1+\epsilon}$ for any $\epsilon > 0$. Thus, $\log_2 q \approx P \log P$.
- Let ρ be a $2n$-th root of unity in $\mathbb{Z}/q\mathbb{Z}$ such that $\rho^{n/P} = b$.

Working in radix b is like working with "polynomials" of degree P whose coefficients are bounded by b.

	Naive Way	New way
x	x	$X(b) = x_0 + x_1 \cdot b + x_2 \cdot b^2 \cdots x_{P-1} \cdot b^{P-1}$
y	y	$Y(b) = y_0 + y_1 \cdot b + y_2 \cdot b^2 \cdots y_{P-1} \cdot b^{P-1}$
x $*$ y	$z = x \cdot y$ and $x * y = z \mod q$	$Z = X \cdot Y \mod (X^P + 1)$ and $x * y = Z(b)$

Using Fürer's algorithm, we got:

Case	Degree	Mult. by a root	Recursion	Complexity
$\mathbb{C}$	$O(N/\log N)$	expensive	$O(\log N)$	$N \log N \log\log N \cdots 2^{O(\log^* N)}$
$\mathbb{Z}/(2^e+1)\mathbb{Z}$	$O(\sqrt{N})$	cheap	$O(\sqrt{N})$	$N \log N \log\log N$
$\mathbb{C}[x]/(x^P+1)$	$O(N/\log^2 N)$	it depends	$O(\log^2 N)$	$N \log N\, 2^{O(\log^* N)}$
$\mathbb{Z}[x]/(q^c, x^P+1)\mathbb{Z}$	$O(N/\log^2 N)$	it depends	$O(\log^2 N)$	$N \log N\, 2^{O(\log^* N)}$

Using Fürer's algorithm, we got:

Case	Degree	Mult. by a root	Recursion	Complexity
$\mathbb{C}$	$O(N/\log N)$	expensive	$O(\log N)$	$N \log N \log\log N \cdots 2^{O(\log^* N)}$
$\mathbb{Z}/(2^e+1)\mathbb{Z}$	$O(\sqrt{N})$	cheap	$O(\sqrt{N})$	$N \log N \log\log N$
$\mathbb{C}[x]/(x^P+1)$	$O(N/\log^2 N)$	it depends	$O(\log^2 N)$	$N \log N\, 2^{O(\log^* N)}$
$\mathbb{Z}[x]/(q^c, x^P+1)\mathbb{Z}$	$O(N/\log^2 N)$	it depends	$O(\log^2 N)$	$N \log N\, 2^{O(\log^* N)}$

Using the last trick, we get the following data:

Case	Degree	Mult. by a root	Recursion	Complexity
$\mathbb{C}$	$O(N/\log N)$	expensive	$O(\log N)$	$N \log N \log\log N \cdots 2^{O(\log^* N)}$
$\mathbb{Z}/(2^e+1)\mathbb{Z}$	$O(\sqrt{N})$	cheap	$O(\sqrt{N})$	$N \log N \log\log N$
$\mathbb{C}[x]/(x^P+1)$	$O(N/\log^2 N)$	it depends	$O(\log^2 N)$	$N \log N\, 2^{O(\log^* N)}$
$\mathbb{Z}[x]/(q^c, x^P+1)\mathbb{Z}$	$O(N/\log^2 N)$	it depends	$O(\log^2 N)$	$N \log N\, 2^{O(\log^* N)}$
$\mathbb{Z}/(b^P+1)\mathbb{Z}$	$O(N/(\log N \log\log N))$	it depends	$O(\log N \log\log N)$	$N \log N\, 2^{O(\log^* N)}$

Removing the polynomial layer improves the complexity of the algorithm.

Removing the polynomial layer improves the complexity of the algorithm.

With a careful complexity analysis, we get the following complexity estimate: $N \log N \cdot 4^{\log^* N}$ (the same as the one obtained by Harvey, Lecerf and Van der Hoeven with Mersenne primes).

Removing the polynomial layer improves the complexity of the algorithm.

With a careful complexity analysis, we get the following complexity estimate: $N \log N \cdot 4^{\log^* N}$ (the same as the one obtained by Harvey, Lecerf and Van der Hoeven with Mersenne primes).

An efficient implementation has to be done.

Removing the polynomial layer improves the complexity of the algorithm.

With a careful complexity analysis, we get the following complexity estimate: $N \log N \cdot 4^{\log^* N}$ (the same as the one obtained by Harvey, Lecerf and Van der Hoeven with Mersenne primes).

An efficient implementation has to be done.

Some limitations: efficient multiplication of two polynomials modulo $X^P + 1$ (bilinear rank), strategy for choosing a good prime, an algorithm for the decompositions...